

Vaibbhav Taraate

Digital Design from the VLSI Perspective

Concepts for VLSI Beginners

Digital Design from the VLSI Perspective

Vaibhav Taraate

Digital Design from the VLSI Perspective

Concepts for VLSI Beginners



Springer

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

ISBN 978-981-19-4651-6 ISBN 978-981-19-4652-3 (eBook)
<https://doi.org/10.1007/978-981-19-4652-3>

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2023

This work is subject to copyright. All rights are solely and exclusively licensed 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, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

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

*Dedicated to my Inspiration
Respected Gordon Moore
and
Late Respected Ajit Shelat*

Preface

I am delighted to have the first edition of the *Digital Logic Design from VLSI Perspective* book. During past twenty-one years I have taught this subject in various multinational corporations.

Even during the VLSI-based product and system designs I have used various techniques, and I am delighted to share the domain knowledge with you.

This edition includes the digital design and various logic design techniques from the VLSI and system perspectives. The manuscript also discusses about the performance improvement of the design, and the design optimization and improvement techniques are also included in this edition!

Especially more than hundreds of practical design scenarios are included in this manuscript, and manuscript contents are useful to the researchers and engineers.

For the schematic of few designs, I have used Xilinx Vivado and ISE 14.7. The readers can go to www.xilinx.com and can download the EDA tool and even can purchase the XILINX FPGA boards and tools to implement the products and ideas using the architecture case studies discussed in this book.

The book has 19 chapters and mainly useful to understand about the logic design concepts, performance improvement of the design, designs from VLSI perspective, FSM-based designs, data and control path designs, and case studies from VLSI perspective.

The book even covers the advanced concepts used during the architecture designs, low power, and multiple clock domain design concepts.

Chapter 1 Introduction: The chapter discusses about the basics of digital systems and number representations.

Chapter 2 Basics of Design Elements: The chapter is useful to understand the basics of combinational and sequential design elements.

Chapter 3 System and Architecture Design: The chapter is useful to understand the architecture, micro-architecture design, and the digital design concepts useful to design the efficient architecture and micro-architecture.

Chapter 4 Combinational Logic and Design Techniques: The techniques such as K-maps and design using mux are always helpful during the design phase. In this context the chapter discusses about the various design techniques, arithmetic

resources, design using multiplexers, universal logic. The objective of the design engineer is to have the design with less area, more speed, and less power.

Chapter 5 Data Control Elements and Applications: The chapter is useful to understand the multiplexers and their use from VLSI perspective.

Chapter 6 Decoders and Encoders: The chapter is useful to understand various decoders, encoders, and priority logic and their applications.

Chapter 7 Combinational Design Scenarios: The chapter is useful to understand the applications of the multiplexers and design using the combinational elements. Various design scenarios are discussed in this chapter.

Chapter 8 Synchronous Sequential Design: The main important sequential design techniques are discussed in this chapter. The chapter focuses on the synchronous counters, shift registers, and design using D flip-flops.

Chapter 9 Logic Design Scenarios and Objectives: The chapter is useful to understand the various asynchronous design techniques and practical scenarios and why asynchronous designs are not recommended in the design of SOCs. Even this chapter is useful to understand the important logic design scenarios and objectives of the designer to have design with minimum area, high speed, and low power.

Chapter 10 Sequential Design Scenarios: The chapter is useful to understand various sequential design scenarios.

Chapter 11 Timing Parameters and Maximum Frequency Calculations: To understand much more about the timing and frequency calculations and to design the architectures, the first step is the conceptual understanding of the sequential circuit parameters. The chapter discusses about the delays, sequential circuit parameters, and frequency calculations.

Chapter 12 FSM Designs: The chapter is useful to understand about the basics of FSM designs, state diagrams and FSM representations, FSM encoding techniques, and design of Moore and Mealy FSMs for the given functional specifications.

Chapter 13 Design of Sequence Detectors: The FSM logic design of various sequence detectors with the minimum area, maximum speed, and minimum power are discussed in this chapter. The chapter is even useful to understand about the FSM optimizations, data, and control paths for the design.

Chapter 14 Performance Improvement for the Design: The performance of the design can be improved by using area, power, and speed improvement techniques. In this context the chapter discusses about the basics of the performance improvement for the design and how we can tweak the architectures and micro-architectures to improve the design performance.

Chapter 15 Optimization Techniques: In the VLSI context we need to understand the various area, speed, and power optimization techniques. The chapter discusses about the various techniques used during the logic and architecture design to have the lesser area, maximum speed, and lesser power.

Chapter 16 Case Study: Speed Improvement for the Design: The chapter is useful to understand the speed improvement techniques such as register balancing, pipelining, and the design tweaks at the architecture and logic level. The chapter is even useful to understand the design of the pipelined processor.

Chapter 17 Case Study: Multiple Clock Domains and FIFO Architecture

Design: This chapter discusses about the multiple clock domain design techniques and the control and data path synchronizers and their use! Even the chapter is useful to understand about the asynchronous FIFO architecture design.

Chapter 18 Hardware Description for Design: The chapter discusses about the basics of the hardware description and the role of the hardware description languages to have the efficient VLSI-based designs.

Chapter 19 FPGA Architecture and Design Flow: The chapter is useful to understand about the FPGA architecture and FPGA design flow. The chapter is also useful to understand about the PLD classification and programmable features and use of the FPGAs during the prototype phase.

The book covers more than hundreds of practical design scenarios, examples to understand how to design the logic and architecture from the specifications? The book is also useful to understand the optimization and performance improvement techniques. The book even covers the important case studies of pipelined processor architecture and micro-architecture design, FIFO architecture design.

This book is useful to the engineering undergraduates, postgraduates, VLSI beginners, RTL design beginners, system and logic design engineers, and professionals those who wish to design the architectures and micro-architectures using logic design concepts!

Thank you very much in advance for buying, reading, and enjoying this lengthy manuscript. I am sure that you will be benefited with the domain knowledge through this manuscript!

Vaibbhav Taraate
Entrepreneur and Mentor
1 Rupee S T (Semiconductor Training
@ Rs 1)
Pune, Maharashtra, India
<https://www.onerupeest.com>

Acknowledgements

Most of the engineers requested me to write a book on *Digital Design from VLSI Perspective* during the corporate training programs. Over the period of time whatever experience which I have gained I thought to document the practical scenarios in this manuscript.

This book is possible due to help of many people. I am thankful to all the participants to whom I taught the subject *Digital Design from VLSI Perspective* in various multinational corporations. I am thankful to all those entrepreneurs, design/verification engineers, and managers with whom I worked in the past almost around 20 years.

Especially I am thankful to my mother, father, and my mother-in-law for their great support during the pandemic duration 2020–2021.

I am thankful to my dearest friends, well-wishers, and family members for their constant support. Special thanks to Neeraj, Annu and Deepesh for their best wishes and for their valuable help during the manuscript work.

Special thanks to Somi, Siddhesh, and Kajal for their faith and belief on me and for their indirect support.

Finally, I am thankful to Springer Nature staff, especially Swati Meherishi, Aparajita Singh, Muskan Jaiswal, Ashok Kumar, Vishnu Muthuswamy and Rini Christy, for their great support during the various phases of the manuscript.

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

Contents

1	Introduction	1
1.1	Number Representation	1
1.2	Digital Systems: System Perspective	4
1.3	Processors and Their Role	5
1.4	The Important Terminology: System Perspective	6
1.5	System Design Components	7
1.6	Few Important Considerations	8
1.7	Summary	10
2	Basics of Design Elements	11
2.1	Combinational Design Elements	11
2.1.1	Logic Gates and Their Use in the Design	12
2.2	De Morgen's Theorems	20
2.2.1	NAND is Equal to Bubbled OR	20
2.2.2	NOR is Equal to Bubbled and	21
2.3	Level Versus Edge Sensitive Elements	22
2.3.1	Latches and Their Use in the Design	22
2.3.2	Edge Sensitive Elements and Their Role	24
2.4	Summary	26
3	System and Architecture Design	27
3.1	Architecture of the Design	27
3.2	Micro-Architecture of the Design	29
3.3	System Design Architecture	29
3.4	Design for the Glue Logic	32
3.5	Application of 2-variable Karnaugh Maps	33
3.6	Let Us Design Two Variable Function	34
3.7	SOP Terms and Boolean Expression	36
3.8	POS Terms and Expression	37
3.9	Design of Glue or Combinational Using Minimum Logic Gates	37
3.10	Summary	40

4	Combinational Logic and Design Techniques	41
4.1	Let Us Design Few Boolean Functions	41
4.2	Arithmetic Resources	46
4.2.1	Half Adder	46
4.2.2	Half Subtractor	47
4.2.3	Full Adder	49
4.2.4	Full Adder Using Half Adders	51
4.3	Role of Data Control Elements	52
4.4	The Multi-Bit Adder and Subtractor	52
4.5	The Multi-Bit Adder with Area Optimization	53
4.6	3-Variable K-Map and Code Converters	55
4.6.1	3-bit Binary to Gray Code Converter	55
4.6.2	3-Bit Gray to Binary Code Converter	57
4.7	Summary	60
5	Data Control Elements and Applications	61
5.1	Data and Control Paths in Design	61
5.2	Multiplexers to Control the Data	62
5.3	Lowest Order Mux in the Design	63
5.4	The 2:1 MUX Using NAND	64
5.5	The 4:1 MUX	66
5.6	The Design of 4:1 Mux Using 2:1 Mux	66
5.7	Design Using Multiplexers	68
5.7.1	NOT Using 2:1 Mux	68
5.7.2	NAND Using Mux	69
5.7.3	NOR Using 2:1 Mux	69
5.7.4	Design of XOR Gate to Get the SUM Output Using Mux	72
5.7.5	Design of XNOR Gate to Get the Even Parity	73
5.8	Boolean Functions and Implementation Using Mux	74
5.9	Mux as Universal Logic	76
5.10	Summary	77
6	Decoders and Encoders	79
6.1	Demultiplexers and Use in the Design	79
6.2	Decoder 2 to 4 Having Active High Output	80
6.3	Decoder 2 to 4 Having Active Low Output	83
6.4	Design for the Given Specifications	84
6.5	Design of 3:8 Encoder Using 2:4 Decoders	85
6.6	Encoders and Their Applications	86
6.7	Practical Encoder Design	89
6.8	Priority Encoders	91
6.9	Practical Design Scenario	92
6.10	Summary	93

7	Combinational Design Scenarios	95
7.1	Mux-Based Designs and Optimization	95
7.2	Right and Left Shift Using Multiplexers	96
7.3	Design of 8:1 Mux Using 4:1 Mux	97
7.4	Design of 8:1 Mux Using 2:1 Mux	98
7.5	Boolean Expression from the Logic	99
7.6	Boolean Expression for Mux-Based Design	100
7.7	Stuck at Faults	102
7.8	Design Using Decoders	102
7.9	Design Using Decoder and NAND Gates	103
7.10	Summary	104
8	Synchronous Sequential Design	105
8.1	Sequential Design Elements	105
8.2	Synchronous Design	109
8.3	Why to Use Synchronous Design?	109
8.4	Asynchronous Design	111
8.4.1	<i>D</i> Flip-Flop and Use in the Design	111
8.5	Design of the Synchronous Counters	113
8.6	Design of the Synchronous Down-Counters	115
8.7	Design of the Synchronous Gray Counter	118
8.8	Few Important Guidelines	120
8.9	Summary	120
9	Logic Design Scenarios and Objectives	123
9.1	What is Asynchronous Design?	123
9.2	Synchronous Versus Asynchronous Reset	124
9.2.1	<i>D</i> Flip-Flop Having Asynchronous Reset	124
9.2.2	Synchronous Reset <i>D</i> Flip_flop	125
9.3	Asynchronous MOD Counters	126
9.3.1	Frequency Divider Network	127
9.3.2	Ripple Counter Design	128
9.4	Design Scenario	129
9.5	PIPO Register	131
9.6	Shift Register	131
9.6.1	Shift Operation and Clock Cycles	132
9.7	Bidirectional Shift Register	132
9.8	Important Design Guidelines	134
9.9	Summary	134
10	Sequential Design Scenarios	137
10.1	Design Scenario I	137
10.2	Four-Bit Latch	137
10.3	Positive Edge Sensitive Flip-Flop Using Multiplexers	139
10.4	Flip-Flop Negative Edge Sensitive	139
10.5	Timing Sequence of Design	140

10.6	Load and Shift Register	141
10.7	Design Scenario II	141
10.8	Design Scenario III	143
10.9	Design Scenario III	143
10.10	Design Scenario IV	144
10.11	Design of 4-bit Ring Counter	144
10.12	Design of 4-bit Johnson Counter	147
10.13	Duty Cycle Control	148
10.13.1	Counter Design with 50% Duty Cycle	150
10.14	Summary	155
11	Timing Parameters and Maximum Frequency Calculations	157
11.1	What is Delay in the System?	157
11.1.1	Cascade Logic Elements in Design	158
11.1.2	Parallel Logic Elements in Design	159
11.2	How Delays Affect the Performance of the Design?	159
11.3	Sequential Circuit and Timing Parameters	160
11.4	Timing Paths in Design	162
11.4.1	Input to Register Path	163
11.4.2	Register to Output Path	163
11.4.3	Register to Register Path	164
11.4.4	Input to Output Path	164
11.5	Maximum Frequency Calculations	164
11.5.1	Design 1: Toggle Flip-Flop	165
11.5.2	Design II: The 2-bit Synchronous Up-Counter	166
11.6	Maximum Operating Frequency	167
11.6.1	Maximum Operating Frequency for Synchronous Designs	168
11.7	Clock Skew	169
11.7.1	Positive Clock Skew and Maximum Operating Frequency	169
11.7.2	Negative Clock Skew and Maximum Operating Frequency for the Design	170
11.8	VLSI Specific Scenarios	171
11.8.1	VLSI Specific Design Scenario I	171
11.8.2	VLSI Specific Design Scenario II	172
11.9	Hold Slack	173
11.9.1	VLSI Specific Design Scenario III	174
11.10	Summary	174
12	FSM Designs	177
12.1	Introduction to FSM	177
12.1.1	Moore FSM	178
12.1.2	Mealy FSM	178
12.1.3	Moore Versus Mealy FSM	179
12.2	State Encoding Methods	179

12.3	Moore FSM Design	181
12.4	Mealy FSM Design	185
12.5	Applications and Design Strategies	189
12.6	State Diagrams	189
12.6.1	Moore Machine State Diagram	190
12.6.2	Mealy Machine State Diagram	190
12.7	Summary	191
13	Design of Sequence Detectors	193
13.1	Moore Machine Non-overlapping 101 Sequence Detector	193
13.2	Mealy Machine Non-overlapping 101 Sequence Detector	194
13.3	One-Hot Encoding	194
13.4	FSM Area and Power Optimization	196
13.5	Moore Sequence Detector for 101 Overlapping Sequence	197
13.6	Mealy Sequence Detector for 101 Overlapping Sequence	201
13.7	Mealy Sequence Detector for 1010 Overlapping Sequence	205
13.8	Various Paths in the Design	209
13.9	Data and Control Path Design Techniques	210
13.10	Summary	211
14	Performance Improvement for the Design	213
14.1	What Is Design Performance?	213
14.2	How to Use the Minimum Arithmetic Resources	214
14.3	Multibit Adders and Subtractors	214
14.4	Four-Bit Full Adder	215
14.4.1	4-Bit Full Subtractor	215
14.4.2	4-Bit Adder and Subtractor	216
14.4.3	Area Optimization of 4-Bit Adder and Subtractor	218
14.4.4	Optimization of Design Using Only Adders	218
14.4.5	Optimization by Tweaking the Logic to Have Least Area and Least Power	218
14.5	Optimization of the Design for Least Area and Power	220
14.6	Comparators and Parity Detectors with Lesser Area	220
14.6.1	Binary Comparator Design with Least Area	220
14.6.2	Parity Detector Design with Least Area	222
14.7	Processor Designs and Speed Improvement Techniques (Source: www.onerupeest.com)	224
14.8	Avoid Asynchronous Designs to Improve the Speed	227
14.9	Power Improvement	228
14.9.1	Gated Clocks and Dynamic Power Reduction	228
14.10	Summary	229

15 Optimization Techniques	231
15.1 Let Us Understand About the Area Optimization	231
15.2 Arithmetic Resource Sharing	231
15.3 Resource Sharing for Sequential Circuits	233
15.4 Logic Duplications	233
15.5 Design Scenario: Performance Improvement	236
15.6 Use of Pipelining in Design	239
15.6.1 Design Without Pipelining	239
15.6.2 Speed Improvement Using Register Balancing or Pipelining	241
15.7 Power Improvement of Design	242
15.8 Dynamic Power Reduction	245
15.9 Summary	247
16 Case Study: Speed Improvement for the Design	249
16.1 Case Study: Speed Improvement at Logic-Level Case Study	249
16.2 Speed Improvement at Architecture Level (Source: www.onerupeest.com)	251
16.2.1 Top-Level Pin Interface	253
16.2.2 Pin Description	253
16.2.3 Case Study: Micro-architecture Design	253
16.3 Summary	257
17 Case Study: Multiple Clock Domains and FIFO Architecture Design	259
17.1 Single Clock Domain Designs	259
17.2 Multiple Clock Domain Designs	259
17.3 Metastability	261
17.4 Control Path Synchronizer	261
17.5 Data Path Synchronizers	262
17.5.1 Why We Need FIFO?	262
17.5.2 FIFO Depth Calculation	263
17.5.3 Case Study: FIFO as a Data Path Synchronizer	266
17.5.4 Micro-architecture of FIFO	267
17.6 Design Guidelines	270
17.7 Summary	271
18 Hardware Description for Design	273
18.1 Verilog HDL	273
18.2 Use of Continuous Assignments	274
18.3 The always Procedural Block	274
18.4 The Procedural Block always@*	275
18.5 Use of the case Construct	276
18.6 Continuous Versus Procedural Assignments	277
18.7 Multiple Blocking Assignments Within the always Block	278

Contents	xix	
18.8	Design Scenario I: Blocking Assignments	279
18.9	Non-blocking Assignments	280
18.10	Design Scenario II: Example Using Non-blocking Assignments	281
18.11	The 4-bit Register	281
18.12	Asynchronous Reset	282
18.13	Synchronous Reset	283
18.14	Design Guidelines and Summary	284
19	FPGA Architecture and Design Flow	285
19.1	Basics of Programmable Logic	285
19.2	CPLD Versus FPGA	286
19.3	ASIC Versus FPGA	287
19.4	FPGA Architecture	288
19.5	FPGA Design Flow	290
19.5.1	Design Planning	290
19.5.2	RTL Design	291
19.5.3	Design Verification and Synthesis	291
19.5.4	Design Implementation	292
19.5.5	Device Programming and Testing	292
19.6	FPGA-Based Product Design	293
19.7	Summary	294
19.8	Further Reading	294
	Bibliography	297
	Index	299

About the Author

Vaibbhav Taraate is Entrepreneur and Mentor at “1 Rupee S T”. He holds a B.E. (Electronics) degree from Shivaji University, Kolhapur (1995) and secured a gold medal for standing first in all engineering branches. He has completed his M.Tech. (Aerospace Control and Guidance) in 1999 from Indian Institute of Technology (IIT) Bombay. He has over 18 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



The basics of digital logic design and the various useful techniques plays important role in the architecture and system design.

Most of the time the beginners in VLSI gets confused about the role of digital design and the important elements. In such scenario it is very much important to understand about the need of the digital design from VLSI perspective. The chapter is mainly focused on the goal and objective of the logic designer while designing the VLSI-based systems or embedded systems. The chapter discusses the basics of the digital systems.

1.1 Number Representation

We can have the weighted numbers and unweighted numbers. The weighted numbers are binary, octal, hexadecimal, and BCD. Unweighted numbers are gray codes and Excess 3 codes.

The digital system and the design elements use the binary inputs as 1 or 0. But we should be familiar with the various number representations. So let us discuss the various number representations.

1. Weighted number: Four-bit Binary

The base of binary number is 2. The base of hexadecimal number is 16. The binary and its equivalent hexadecimal are shown in Table 1.1.

The decimal number has the base 10, and the binary equivalent of the decimal is documented in Table 1.2.

2. Weighted Number: Octal

The octal numbers have base of 8 and the Table 1.3 shows the binary and its equivalent octal number (Table 1.3). The BCD and its equivalent Excess-3 codes are shown in the Table 1.4.

Table 1.1 Binary and its equivalent hex number

4-bit binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Table 1.2 Decimal and its equivalent binary

4-bit binary	Decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

3. Weighted Number: 4-bit BCD or 8421 Code

The binary coded decimals are also called as 8421 code and if we consider combination of 4-bit binary, then the total number of representations is $2^4 = 16$. But as the name indicates it is binary coded decimal; the valid BCD codes are 10, and invalid BCD codes are 6. During the arithmetic operations if we find any invalid

Table 1.3 Binary and its equivalent octal

3-bit binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Table 1.4 BCD and Excess 3 codes

Decimal	Valid BCD 8421	Excess 3 code
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

BCD, then we need to add 0110 to get the correct output. The BCD code is not self-complementing.

4. Unweighted Number: Excess 3 code

If we need to get the Excess 3 code, then we can get it from BCD code by adding 0011 to each coded number. Excess 3 is unweighted and self-complementing code (Table 1.4).

5. Unweighted Number: Gray code

Gray codes are unweighted codes, and the 4-bit gray equivalent of binary is shown in Table 1.5. The gray codes are also called as reflected codes and used in error correction algorithms and in multiple clock domains.

Depending on the design requirement we use these number representations. And in the subsequent chapter we will discuss the basics of combinational and sequential elements.

Table 1.5 Four-bit gray

4-bit binary	4-bit gray
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

1.2 Digital Systems: System Perspective

Most of the logic designers are familiar with the two types of systems that is analog systems and digital systems. The system can be designed using analog and digital components such as ADC, DAC, sensors, processors, memories, and IO devices. These kinds of systems are called as embedded systems.

The analog systems use the analog input which is function of the time and processes that to get the desired analog output. For example, consider the analog system to detect the various parameters such as temperature, pressure. Such type of system should consist of sensors, ADC, processor to process the data, and DAC to generate the desired analog output.

The digital systems are used to process the binary data which is in the form of logic 0 and logic 1. The numbers can be represented using the binary, gray, octal, BCD, or hexadecimal formats. The binary number system has base of 2, the octal number system has base of 8, and the hexadecimal number has base of 16.

Even most of the time we need to have the signed, unsigned number representations, fixed point, and floating-point numbers. To design the system which can perform the processing using the floating-point numbers we need to use the desired IEEE standards.

Now consider the digital system which is used to process the digital data and to perform the communication of IO and memory devices. The memory devices can be of type EPROM, ROM, RAM, etc. and IO devices can be for the interfacing of keyboard, display or to transfer the serial or parallel data from processor to external

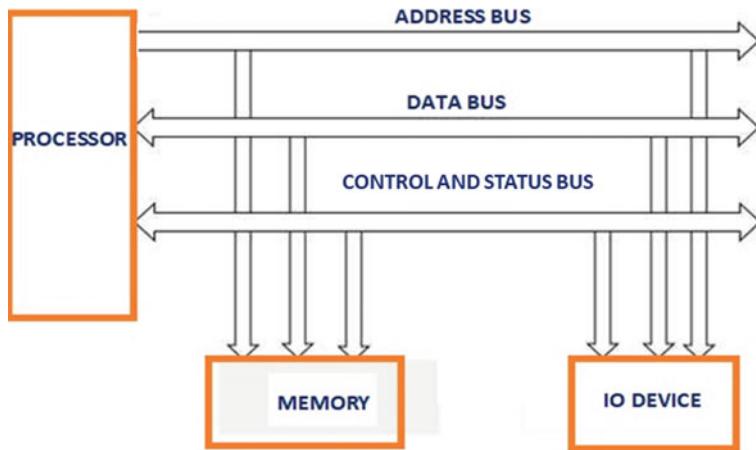


Fig. 1.1 Basic digital system

world. Such type of system is very basic system having limited number of components. Figure 1.1 shows the digital system which consists of processor, IO, and memory devices.

In the system perspective we need to think about the digital system, and such kind of system should have the

1. Microprocessors
2. Memories
3. General purpose IO devices
4. ADC
5. DAC.

For the system designer it is essential to understand about the role of the above-mentioned components and their use to design the efficient system. In the subsequent chapters we will discuss this in much more detail.

1.3 Processors and Their Role

In the system design the selection of the processors plays very important role. There is different types of the microcontrollers and microprocessors available in the market, and the objective of designer is to have the processor or microcontroller which has the desired speed, low power dissipation, and should perform the desired operations on the binary data. Even important aspect to investigate it is the number of pins and how easy to use them to interface with the memory and IO devices.

The block representation of the processor interfaced with the memory is shown in Fig. 1.2. As shown the processor is interfaced with the memory using system bus. System bus is combination of the address, data, and control bus.

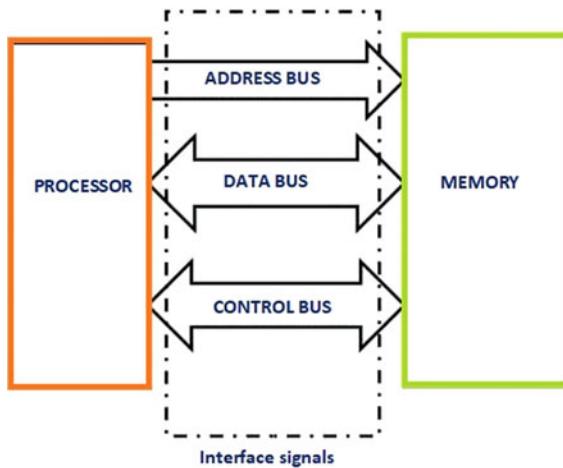


Fig. 1.2 System bus

Table 1.6 System bus

Bus	Description
Address Bus	Address bus is unidirectional and used to send the address of IO or memory devices
Data Bus	The data bus is bidirectional and used to carry data to/from processor and IO devices
Control and status Bus	The control and status bus are used to poll the status of IO devices and used to generate the control signals such as read and write for the IO or memory devices

Table 1.6 gives information about the buses used to interface the memories and IO devices.

1.4 The Important Terminology: System Perspective

Now let us try to understand the important terminology used in the design of the digital systems. We will think about the system perspective.

1. **System specifications:** The overall specifications for the design which includes the objective and application of the systems, components, electrical characteristics and the information about the external interfaces.
2. **System components:** The system components to design the digital systems, these components are microprocessors, IO devices, memory devices, sensors, power supply units, serial devices, ADCs, DACs, keyboard, and display. We need to identify them depending on the need and the various design considerations.

3. **Design considerations:** The important design considerations during system design are compatibility, fanout, load, propagation delay, power, speed which should be understood by the system designer to design the digital system.
4. **Power requirements and backup strategies:** The overall power requirement for the system and how to design the efficient power supply network will play very important role. Even for the volatile memories and programmable logic devices we need to have the backup strategies in place.
5. **External memories and management:** The processors or microcontrollers have limited memory capacity and system designer should understand about the need of the external memories and should work on the better and flexible memory management and interfaces.
6. **External IO or peripheral Devices:** The system designer should know about the need of the external IO devices and should identify them to have the better IO selections and IO managements.
7. **Multiple power domain strategies:** Most of the systems have the multiple power domains, for example, we need to have isolation of the analog and digital ground if the system has analog and digital components.
8. **Mechanical Assembly:** The system or product designer should have the separate team to design the mechanical assembly and the compact products.

The system designers need to consider all above while designing the digital systems. By considering all above the main goals of system designers are to design the system, which is compact, should have long life, more user friendly and flexible to use. The system should dissipate the lesser power and should have maximum area.

1.5 System Design Components

As discussed in the previous section we need to understand about the important system design components such as microprocessors, memories, and IO devices and their selection strategies.

1. **Microprocessor:** The microprocessors are used to process the digital data and to communicate with the external peripheral devices, memories. Most of the time while selecting the microprocessor we need to consider about the
 - a. Instruction set and processing features
 - b. The overall pin count of the processor
 - c. The power and speed requirements
 - d. The system bus features.
2. **Memories:** As discussed in the previous section the processor or microcontroller has limited internal memory and hence it is essential to identify the external memory such as ROM, RAM, and EPROM depending on the design requirements.

3. **IO devices:** The need of the different input and output devices and their compatibility with the external world and processors is very much important. For example, consider the digital system which needs the keyboard to input the data and LCD to display the data.

1.6 Few Important Considerations

Following are few of the important considerations while designing the system.

1. **Area of the system:** Use the minimum components while developing the top-level design of the system and justify their role with the compatibility considerations.
2. **Speed of the system:** This is one of the factors which plays important role in the performance of the system. The clocking network and selection of the crystal of desired frequency are very much required while designing the digital system.
3. **Power requirements:** Have the strategy while designing the system to have lower power dissipation and have better power network.
4. **Compatibility:** The two modules or designs are compatible means their logic levels and current profiles are matching with each other. As shown in Fig. 1.3 the AND gate is followed by NOT and both logic gates are compatible with each other, and they understand the voltage and current levels of each other.
5. **Fanout:** The maximum load which can be driven by driver is called as fanout. As shown in the Fig. 1.4 the driver drives the n number of loads and the fanout is n. If number of loads are 100 which are driven by the driver, then the fanout of the gate is 100.

Consider the sink current of driver is 16 microamperes and the source current of each load is 1.6 micro-amperes; then the fanout of the driver is $16/1.6 = 10$ loads.

6. **Noise margin:** The maximum allowable noise in the system is called as noise margin. The Fig. 1.5 gives information about the voltage profiles, that is input and output minimum and maximum voltage levels.

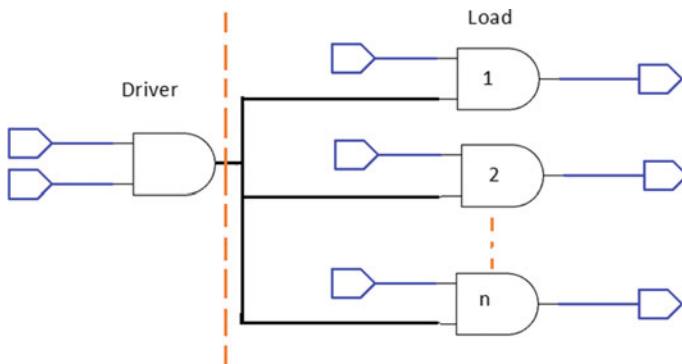
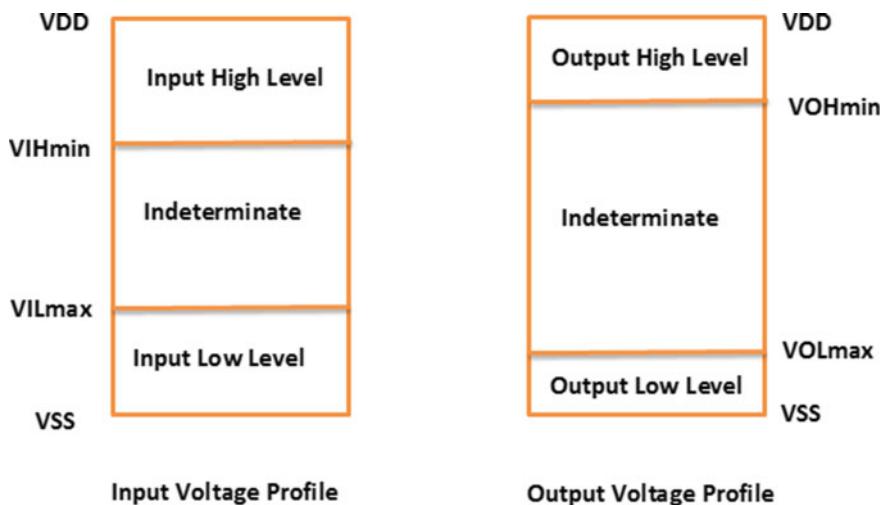
The noise margin is V_n and it is defined as

$$V_n = V_{IL\ max} - V_{OL\ max}$$

where $V_{IL\max}$ = Maximum low-level input voltage



Fig. 1.3 Design compatibility

**Fig. 1.4** Fanout of the design**Fig. 1.5** Voltage profiles

$V_{OL\max}$ = Maximum low-level output voltage

The noise margin is V_n is also defined as

$$V_n = V_{OH\min} - V_{IH\min}$$

where $V_{IH\min}$ = Minimum high-level input voltage

$V_{OH\min}$ = Minimum high-level output voltage.

As discussed in this chapter we need to use the understanding of number representations and various elements to design the system. Even we need to consider the compatibility, maximum load, and noise margin of the selected element. The goal is always to have the lesser number of resources while designing the system.

The important objective of the manuscript is to share the knowledge with readers so they can use the digital elements to design the digital systems to have minimum area, maximum speed, and lesser power.

In this context the next chapter is useful to understand the basics of digital design elements.

1.7 Summary

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

1. The digital logic operates on the binary data that is logic 0 and logic 1.
2. The weighted numbers are binary, octal, hexa-decimal, BCD.
3. Unweighted numbers are gray codes and Excess 3 codes.
4. The Excess 3 is unweighted and self-complementing code.
5. The BCD is unweighted and not a self-complementing code.
6. The analog design has input which is function of the time and should be sampled by ADC for the processing.
7. The important components of the digital system are microprocessors, memories, and IO devices.
8. The microprocessor is heart of the digital systems and should communicate with one of the memories or IO devices at a time.
9. The memories and IO devices use the system bus to communicate with the processor.
10. The devices interfaced should be compatible with each other that is they should have the same logic levels and voltage profiles.
11. The number of loads driven by driver is called as the fanout of the system.
12. The noise margin is the maximum noise acceptable by the system.

Chapter 2

Basics of Design Elements



The basics of digital logic design and the various useful design elements plays important role in the architecture and system design.

As discussed in the previous chapter we need to design the digital systems or architectures using the basic elements. The goal is the design should have lesser area, maximum speed, and lesser power. In this context the chapter is useful to understand the various combinational and sequential design elements.

The following are types of digital logic:

1. **Combinational Logic:** In this type of logic an output is function of the present input only. For example, logic gates and multiplexers are combinational logic elements.
2. **Sequential Logic:** In this type of logic an output is function of present input and past output. For example, latches and flip-flop are sequential logic elements.

The subsequent sessions discuss more about the combinational and sequential logic elements.

2.1 Combinational Design Elements

As we know that in the combinational logic an output is function of the present input. That means if one of the input changes then an output change. Practically the output changes after the propagation delay that is t_{pd} time duration.

Examples of the combinational logic are

1. Logic gates
2. Adders, subtractors
3. Code converters
4. Multiplexers
5. Demultiplexers
6. Decoders

7. Encoders
8. Priority encoders.

In the subsequent chapters we will discuss this in much more detail. In the subsequent session let us discuss in detail about the logic gates and their Boolean equations.

2.1.1 Logic Gates and Their Use in the Design

The logic gates are important logic design elements. The focus of this book is more on the use of the logic elements to design the area, power, and speed efficient logic. As name indicates that the logic gates are used to perform the desired logic function. The logic gates have inputs and output, and they are used to build the combinational and sequential logic.

Although most of the engineers are familiar with the logic gates, let us discuss them in the context of the logic design and optimization of the logic!

1. NOT gate

NOT is complement logic and it is used to generate the output which is complement of the input.

The truth table of the NOT gate is shown in Table 2.1. As shown the NOT gate has an input A and an output Y. The relationship between the input and output is given by

$$Y = \overline{A}$$

The NOT gate symbol is shown in Fig. 2.1 and the output is 180 degrees out of phase of an input. The complement of logic 0 is logic 0 and vice versa.

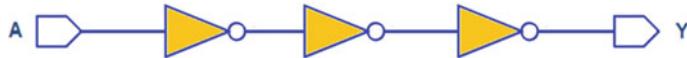
Consider even number of NOT gates in cascade; then output $Y = A$. If each NOT gate has propagation delay of 1 nsec; then the overall propagation delay is 2 nsec as two NOT gates are connected in cascade (Fig. 2.2).

Table 2.1 Truth table of NOT gate

A	Y
0	1
1	0

Fig. 2.1 NOT gate



**Fig. 2.2** Even number of NOT gates in cascade**Fig. 2.3** Odd number of NOT gates connected in cascade

Consider odd number of NOT gates in cascade; then output Y is equal to complement of A. If each NOT gate has propagation delay of 1 nsec; then the overall propagation delay is 3 nsec as three NOT gates are connected in cascade (Fig. 2.3).

2. OR gate

OR is the logical OR of the inputs and in simple words indicates that this OR this! The 2-input OR gate performs logical OR on the two binary inputs to generate a single-bit binary output.

The truth table of the OR gate is shown in Table 2.2. It has inputs A, B and an output Y. The relationship between the inputs and output is given by

$$Y = A + B$$

Figure 2.4 is symbolical representation of OR gate, and it indicates that either A or B should be logic 1 to get an output as logic 1. Hence the logic is OR logic.

Now consider that one of the inputs of OR gate is logic 0. Then the output of OR gate is equal to A (Fig. 2.5).

$$Y = A + 0 = A$$

Table 2.2 Truth table of OR gate

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

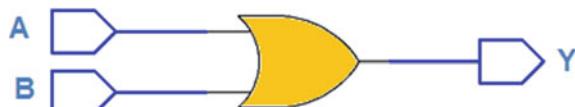
Fig. 2.4 OR gate

Fig. 2.5 2-input OR gate having one of the inputs as logic 0

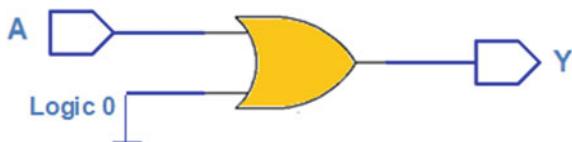


Fig. 2.6 2-input OR gate having one of the inputs as logic 1



Now consider that one of the inputs of OR gate is logic 1. Then the output of OR gate is equal to A (Fig. 2.6).

$$Y = A + 1 = 1$$

3. NOR gate

The NOR is NOT of OR and the output of NOR gate is logic 1 when all the inputs are logic 0. If one of the inputs is logic 1, then an output of NOR gate is logic 0. The NOR gate is universal gate because by using the number of NOR gates any Boolean function can be implemented.

The truth table of NOR gate is shown in Table 2.3 and has inputs as A, B and output as Y. The relationship between the inputs and an output is given by

$$Y = \overline{A + B}$$

The NOT of OR is shown in Fig. 2.7 which is the cascade of OR and NOT.

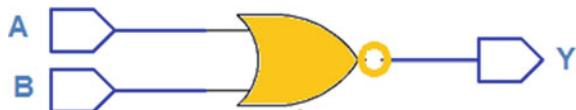
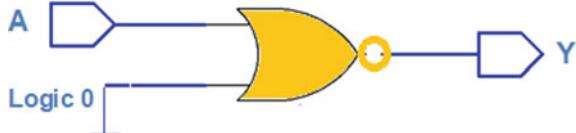
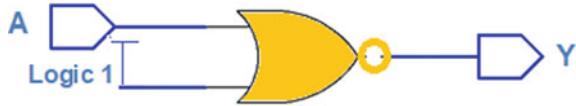
The symbol of the NOR gate is shown in Fig. 2.8 and the minimum number of NOR gates should be used to implement the Boolean function. **By using the minimum number of NOR gates any Boolean function can be realized, and hence, the NOR gate is called as universal logic gate.**

Table 2.3 Truth table of NOR gate

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Fig. 2.7 NOT of OR



Fig. 2.8 NOR gate**Fig. 2.9** 2-input NOR gates having one of the inputs as logic 0**Fig. 2.10** 2-input NOR gate having one of the inputs as logic 1

Now consider that one of the inputs of NOR gate is logic 0. Then the output of NOR gate is equal to complement of A (Fig. 2.9).

$$\begin{aligned} Y &= \overline{A + 0} \\ &= \overline{A} \end{aligned}$$

Now consider that one of the inputs of NOR gate is logic 1. Then the output of NOR gate is equal to logic 0 (Fig. 2.10).

$$\begin{aligned} Y &= \overline{A + 1} \\ &= \overline{1} \\ &= 0 \end{aligned}$$

4. AND Gate

The AND logic output is logic 1 when both the inputs A, B are at logic 1. Hence the logic is represented by using A AND B . The logic expression of 2-input AND gate is given by

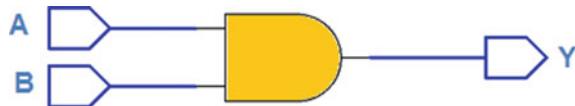
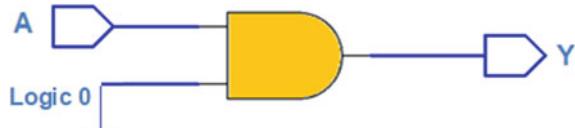
$$Y = A \cdot B$$

The (Pronounced as dot) indicates the AND operation. The truth table is shown in Table 2.4. As described in the truth table when both the inputs A, B are logic 1, an output of the AND gate is logic 1. When one of the inputs is logic 0, the output of AND gate remains at logic 0.

The symbol of the 2-input AND is shown in Fig. 2.11 and as shown the AND gate has two inputs A, B , and an output as Y .

Table 2.4 Truth table of AND gate

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Fig. 2.11 AND gate**Fig. 2.12** 2-input AND gate having one of the inputs as logic 0

Consider that one of the inputs of AND gate is logic 0. Then the output of AND gate is equal to logic 0 (Fig. 2.12).

$$\begin{aligned} Y &= A \cdot 0 \\ &= 0 \end{aligned}$$

Consider that one of the inputs of AND gate is logic 1. Then the output of AND gate is equal to another input A (Fig. 2.13).

$$\begin{aligned} Y &= A \cdot 1 \\ &= A \end{aligned}$$

5. NAND Gate

NAND is NOT of AND; the output of NAND is logic 0 when both the inputs are at logic 1. If one of the inputs of NAND gate is at logic 0, then an output of NAND gate is logic 1. The truth table of 2-input NAND gate is described in Table 2.5. The logic expression is given by

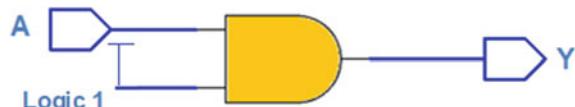
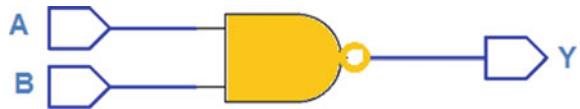
Fig. 2.13 2-input AND gate having one of the input as logic 1

Table 2.5 Truth table of NAND gate

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

Fig. 2.14 NOT of AND**Fig. 2.15** NAND gate

$$Y = \overline{A \cdot B}$$

The cascade of the AND, NOT is shown in Fig. 2.14. As discussed earlier the design engineers should avoid the cascading of the stages.

The NAND gate symbolical representation is shown in Fig. 2.15 and as shown it has two inputs A, B and an output as Y .

Consider that one of the inputs of NAND gate is logic 0. Then the output of NAND gate is equal to logic 1 (Fig. 2.16).

$$\begin{aligned} Y &= \overline{A \cdot 0} \\ &= \overline{0} \\ &= 1 \end{aligned}$$

Consider that one of the inputs of NAND gate is logic 1. Then the output of NAND gate is equal to complement of another input A (Fig. 2.17).

$$= \overline{A \cdot 1}$$

$$= \overline{A}.$$

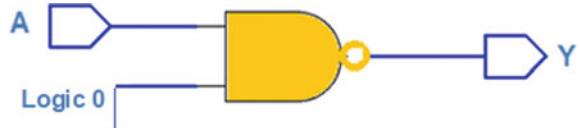
Fig. 2.16 2-input NAND gate having one of the inputs as logic 0

Fig. 2.17 2-input NAND gate having one of the inputs as logic 1



6. XOR Gate

The XOR gate is also called as Exclusive OR. The truth table of the XOR gate is shown in Table 2.6. As shown the output of the 2-input XOR gate is logic 1 when both the inputs are not equal. The logic expression of XOR gate is

$$Y = A \oplus B$$

$$Y = A \cdot \overline{B} + \overline{A} \cdot B$$

The symbol of the XOR gate is shown in Fig. 2.18 and as shown the 2-input XOR gate has inputs A, B and an output as Y .

Consider that one of the inputs of XOR gate is logic 0. Then the output of XOR gate is equal to another input (Fig. 2.19).

$$Y = A \oplus 0$$

$$\begin{aligned} Y &= A \cdot \overline{0} + \overline{A} \cdot 0 \\ &= A \end{aligned}$$

Table 2.6 Truth table of XOR gate

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 2.18 XOR gate

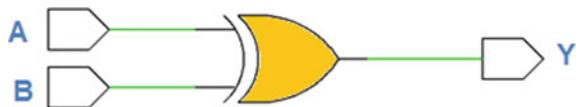


Fig. 2.19 Two input XOR gate having one of the inputs as logic 0

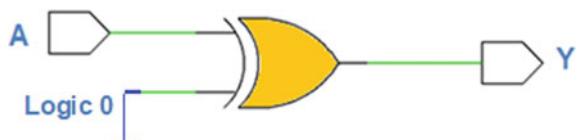


Fig. 2.20 2-input XOR gate having one of the inputs as logic 1



Consider that one of the inputs of XOR gate is logic 1. Then the output of XOR gate is equal to complement of another input (Fig. 2.20).

$$\begin{aligned} Y &= A \oplus 1 \\ Y &= A \cdot \bar{1} + \bar{A} \cdot 1 \\ &= \bar{A} \end{aligned}$$

7. XNOR Gate

The XNOR is NOT of XOR. The symbol of the XNOR is \odot (*Pronounced as EXNOR*). The logic expression is given by

$$Y = A \odot B$$

The EXNOR or exclusive NOR or XNOR are few names which we use while implementing the design. The XNOR is cascade of the XOR and NOT and hence called as NOT of XOR. The XNOR using XOR is shown in Fig. 2.21.

The truth table of the XNOR gate is described in Table 2.7. As described the output of 2-input XNOR gate is logic 1 when both the inputs are equal.

As discussed earlier avoid the cascading of the stages as it increases the overall propagation delay.

The symbol of the XNOR gate is shown in Fig. 2.22 and as shown the logic has A, B inputs and output as Y .

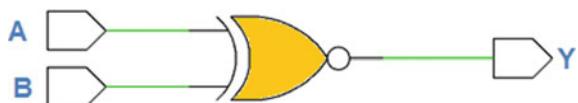
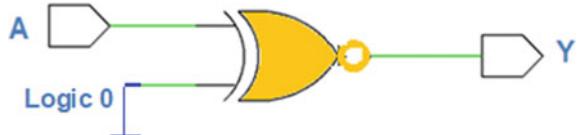
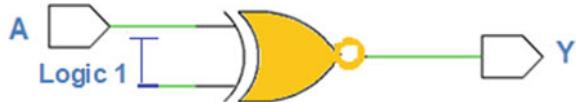
Consider that one of the inputs of XNOR gate is logic 0. Then the output of XNOR gate is equal to complement of another input (Fig. 2.23).

Fig. 2.21 NOT of XOR



Table 2.7 Truth table of XNOR gate

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Fig. 2.22 XNOR gate**Fig. 2.23** 2-input XNOR gate having one of the inputs as logic 0**Fig. 2.24** 2-input XNOR gate having one of the inputs as logic 1

$$Y = A \odot 0$$

$$\begin{aligned} Y &= A \cdot 0 + \overline{A} \cdot \overline{0} \\ &= \overline{A} \end{aligned}$$

Consider that one of the inputs of XNOR gate is logic 1. Then the output of XNOR gate is equal to another input (Fig. 2.24).

$$Y = A \odot 1$$

$$\begin{aligned} Y &= A \cdot 1 + \overline{A} \cdot \overline{1} \\ &= A \end{aligned}$$

In the next subsequent chapters, we will use the logic gates to implement the combinational logic with goal to have minimum area and least propagation delay.

2.2 De Morgen's Theorems

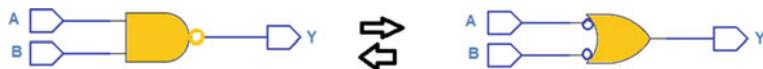
As we have good understanding of the logic gates now, let us try to understand the important theorems as De Morgen's theorems. For the Boolean simplification, the two important theorems are as follows.

2.2.1 NAND is Equal to Bubbled OR

$$\overline{A \cdot B} = \overline{A} + \overline{B}.$$

Table 2.8 Truth table of bubbled OR is equal to NAND

A	\bar{A}	B	\bar{B}	Y
0	1	0	1	1
0	1	1	0	1
1	0	0	1	1
1	0	1	0	0

**Fig. 2.25** Bubbled OR is equal to NAND

The truth table is shown in Table 2.8. The output if bubbled OR matches with the output of 2-input NAND and hence in simple words we can consider bubbled OR is equal to NAND or vice versa.

For Boolean simplifications we can use the De Morgen's theorems. We can use the bubbled OR as a NAND during Boolean simplifications (Fig. 2.25).

2.2.2 NOR is Equal to Bubbled AND

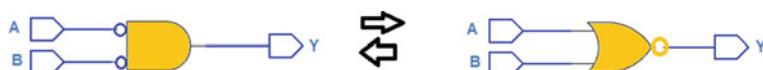
$$A + \bar{B} = \bar{A} \cdot \bar{B}$$

The truth table is shown in Table 2.9. The output if bubbled AND matches with the output of 2-input NOR and hence in simple words we can consider bubbled AND is equal to NOR or vice versa.

For Boolean simplifications we can use the De Morgen's theorems. We can use the bubbled AND as a NOR during Boolean simplifications (Fig. 2.26).

Table 2.9 Truth table of bubbled AND is equal to NOR

A	\bar{A}	B	\bar{B}	Y
0	1	0	1	1
0	1	1	0	0
1	0	0	1	0
1	0	1	0	0

**Fig. 2.26** Bubbled AND is equal to NOR

In the digital design we need to use the level and edge sensitive elements. The following section discusses the latches and flip-flops useful to design the sequential logic.

2.3 Level Versus Edge Sensitive Elements

Most of the time we need to use the sequential design elements such as latches and flip-flops. The latches are level sensitive, and flip-flops are edge triggered. The following section discusses these elements.

2.3.1 Latches and Their Use in the Design

As we know that, the latches are level sensitive. Latches are transparent during active level of enable input. For example, the output of the positive level sensitive D latch is equal to the data input during the active level that is positive level.

Similarly, the negative level sensitive D latch output is equal to the data input during negative level.

2.3.1.1 Positive Level Sensitive D Latch

The positive level sensitive latch is transparent during active high level of enable (EN). For $EN = 1$ output $Q = D$. For the $EN = 0$ the latch holds previous output. The latch truth table is shown in Table 2.10.

The D latch schematic is shown in Fig. 2.27, as shown the data input of latch is D , latch enable input EN which is active high and output of latch Q .

The timing diagram of the positive level sensitive D latch is shown in Fig. 2.28. As shown the latch output $Q = D$ for $EN = 1$. The latch holds previous output either 1 or 0 for the $EN = 0$.

Table 2.10 Positive level sensitive D latch

Enable (EN)	Data input (D)	Output (Q)
1	0	0
1	1	1
0	X	Hold the previous output

Fig. 2.27 Positive level sensitive D latch

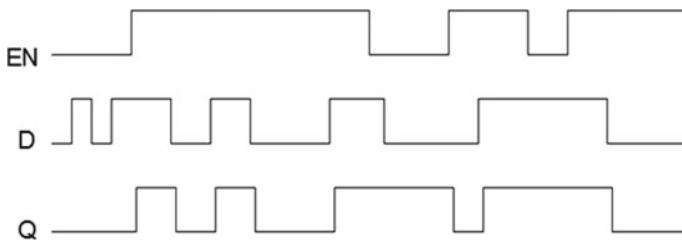
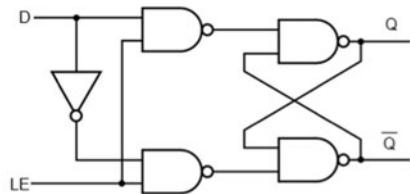
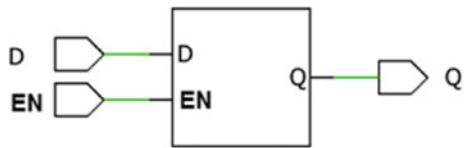


Fig. 2.28 Waveform of the positive level sensitive latch

2.3.1.2 Negative Level Sensitive D Latch

The positive level sensitive latch is transparent during active high level of enable (EN). For $EN = 1$ output $Q = D$. For the $EN = 0$ the latch holds previous output. The latch truth table is shown in Table 2.11.

The D latch schematic is shown in Fig. 2.29, as shown the data input of latch is D , active low latch enable input EN and output of latch Q .

The timing diagram of the negative level sensitive D latch is shown in Fig. 2.30. As shown the latch output $Q = D$ for $EN = 0$. The latch holds previous output either 1 or 0 for the $EN = 1$.

Applications: Latches are used in the system design to hold the previous output during inactive enable condition. For example, consider the multiplexed addressed

Table 2.11 Negative level sensitive D latch

Enable (EN)	Data input (D)	Output (Q)
0	0	0
0	1	1
1	X	Hold the previous output

Fig. 2.29 Negative level sensitive D latch

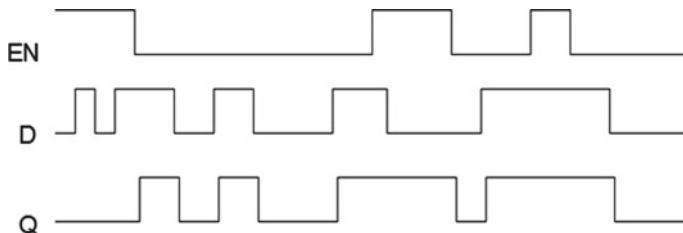
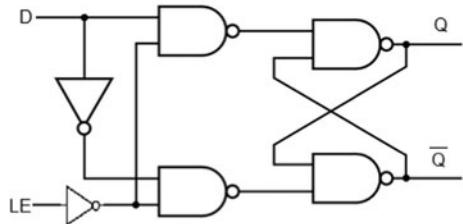


Fig. 2.30 Waveform of the negative level sensitive latch

and data bus as output from microprocessor. The multiplexed bus acts as address bus during active high enable and acts as data bus during active low enable. In such scenario we can use latch to hold the address during inactive enable condition.

2.3.2 Edge Sensitive Elements and Their Role

As we know that most of the time, we use the flip-flops as they are sensitive to the active edge of the clock. That means during one clock cycle data is sampled on either positive edge of the clock or on the negative edge of the clock. The main advantage of the flip-flop is that the data is stable for one clock cycle.

We use the D flip-flops either positive or negative edge sensitive during the design of the counters and shift registers. The logic circuit of the D flip-flops is discussed in this section.

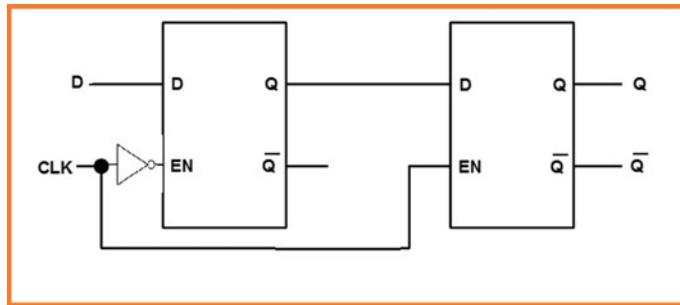


Fig. 2.31 Positive edge triggered D flip-flop

2.3.2.1 Positive Edge Sensitive D Flip-Flop

The positive edge means the low to high transition; it is also called as rising edge. The positive edge sensitive flip-flop is designed by using two latches in cascade. The data input D is given as input to the negative level sensitive D latch, and an output of the negative level sensitive D latch acts as input to the positive level sensitive D latch.

The output of the positive level sensitive D latch acts as a flip-flop output. The positive edge sensitive D flip-flop is shown in Fig. 2.31.

2.3.2.2 Negative Edge Sensitive D Flip-Flop

The negative edge means the high to low transition; it is also called as falling edge. The negative edge sensitive D flip-flop is designed by using two latches in cascade. The data input D is given as input to the positive level sensitive D latch, and an output of the positive level sensitive D latch acts as input to the negative level sensitive D latch.

The output of the negative level sensitive D latch acts as a flip-flop output. The negative edge sensitive D flip-flop is shown in Fig. 2.32.

Applications: As discussed flip-flops are edge sensitive and used in the design of the counters, state machines, shift registers.

The next subsequent chapters focus on the architecture design and system design from the given specifications, and we will use the edge sensitive elements with the combinational resources to design the system.

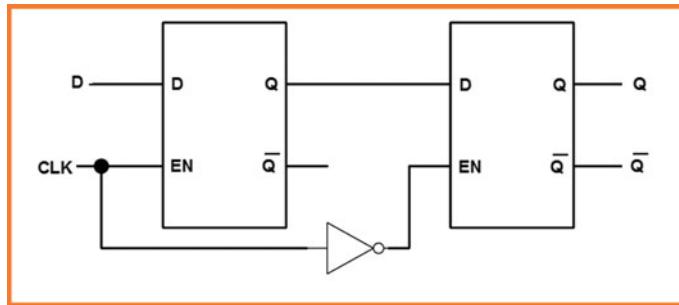


Fig. 2.32 Negative edge triggered D flip-flop

2.4 Summary

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

1. The NAND is universal logic gate and minimum number of NAND gates can be used to implement any other logic gate.
2. The NOR is universal logic gate, and minimum number of NOR gates can be used to implement any other logic gate.
3. NAND is equal to bubbled OR.
4. NOR is equal to bubbled AND.
5. The latches are level sensitive, and flip-flops are edge triggered elements.
6. We will use the flip-flops in the design of the digital systems to have the registered boundaries.

Chapter 3

System and Architecture Design



The architecture and micro-architecture design of VLSI-based system is one of the major milestones during VLSI-based system design.

Most of the time we experience the need of the higher-level understanding about the VLSI-based systems. The architecture and micro-architecture team uses the specification understanding to design and document the architecture of the VLSI-based systems. The chapter is useful to understand the architecture, micro-architecture design and the digital design concepts useful to design the efficient architecture and micro-architecture.

3.1 Architecture of the Design

The architecture of any VLSI-based system is the block-level representation and is evolved using the design functional specifications. The architecture design phase is one of the important milestones, and the following are objectives of the architecture design team.

1. Understand the functional specifications of design.
2. Try to understand the desired functional blocks or elements.
3. Create the top-level block representation of the design.
4. Identify the interface details.
5. Try to estimate the area, speed, and power for the design.
6. Try to understand the timing details.
7. Have the design partitioning according to the multiple clock domain requirements.
8. Try to understand the multiple power domains and incorporate the low power-aware architecture design.

Consider the single clock domain top-level architecture of the pipelined processor.

The architecture is evolved through the design and functional specifications. Consider that, we need to have the RISC processor having few arithmetic, logical and branching instructions and should have the following specifications.

1. The size of data bus 16 bit.
2. The size of address bus 20 bit.
3. The processor should perform the read and write operations.
4. The processor should have the IO or memory identification and should communicate with the single IO or memory device at a time.
5. The processor should have the provision to accept the level, edge triggered hardware interrupt.
6. The processor should have at least 16 instructions and maximum 32 instructions.
7. The internal memory provision should be specified.
8. The processor should have the high-speed IO interfaces and bus interfaces.
9. There should be provision to have serial IO ports.

By considering all the above points the architecture of the processor can be evolved. As shown in Fig. 3.1, the following are the important design blocks in the architecture design.

1. Arithmetic logic unit (ALU)
2. Control and timing unit
3. Interrupt control unit
4. Serial IO
5. Instruction decoder

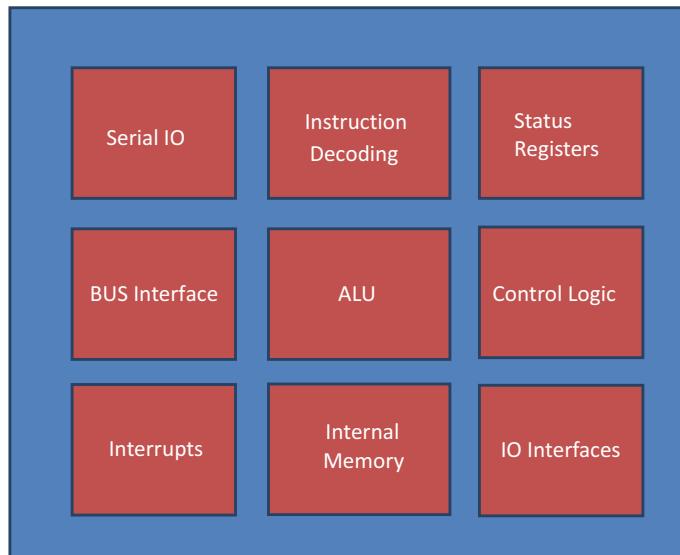


Fig. 3.1 Single clock domain processor architecture

6. Status register
7. IO interfaces
8. Internal memory
9. Bus interface logic

3.2 Micro-Architecture of the Design

The micro-architecture of the design is sub-block-level representation and is always evolved from the architecture of the design. Consider the processor which has 16-bit ALU and it performs the 16 operations then we can use the following thought process to design the micro-architecture.

1. The partition of the ALU by considering the number of arithmetic and logic instructions
 - a. Arithmetic unit
 - b. Logic unit
2. The sub-block representation of the arithmetic unit
 - a. Arithmetic resources such as adders, subtractors, multipliers, and dividers
 - b. Selection logic such as multiplexers
3. The sub-block representation of the logic unit
 - a. Logic operations and need of the logic gates such as NOT, OR, AND, XOR, NAND, NOR, and XNOR
 - b. Multiplexers as selection logic
4. Registered inputs and registered outputs for the arithmetic unit and logic unit
5. Selection logic to select either arithmetic or logic operation.
6. Pipelined stages to perform the pipelining that is fetch, decode, execute, and overlap during the top-level micro-architecture design.

3.3 System Design Architecture

Consider the system design using the components such as microprocessors, memories, and IO devices. In the system design perspective, the system design architecture should focus on the following.

1. How to establish communication between the microprocessor and memory devices?
2. How to establish the communication between the microprocessor and IO devices?
3. How to create the memory and IO mapping so that processor can communicate with one of the memory or IO devices at a time?

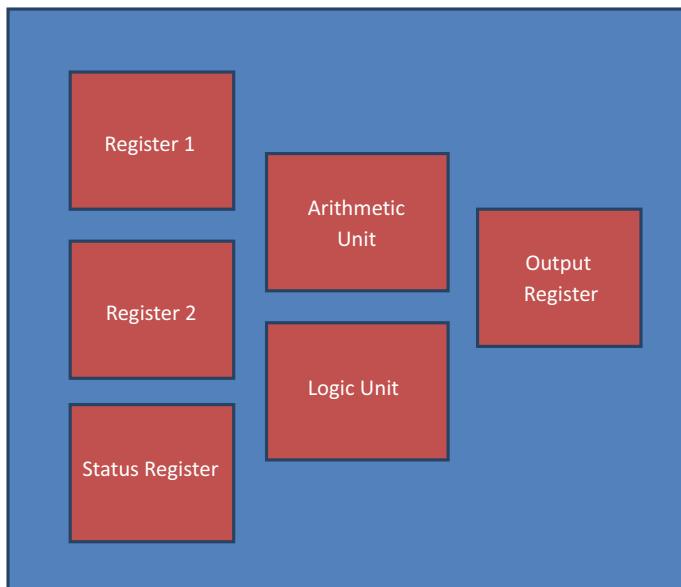


Fig. 3.2 Micro-architecture of the ALU

Consider an example of the system design, we need to have 8 register bank as internal general-purpose registers. Each register is 4-bit wide. We can have the system-level architecture as shown in Fig. 3.2.

What we need to perform the read and write operations with the registers.

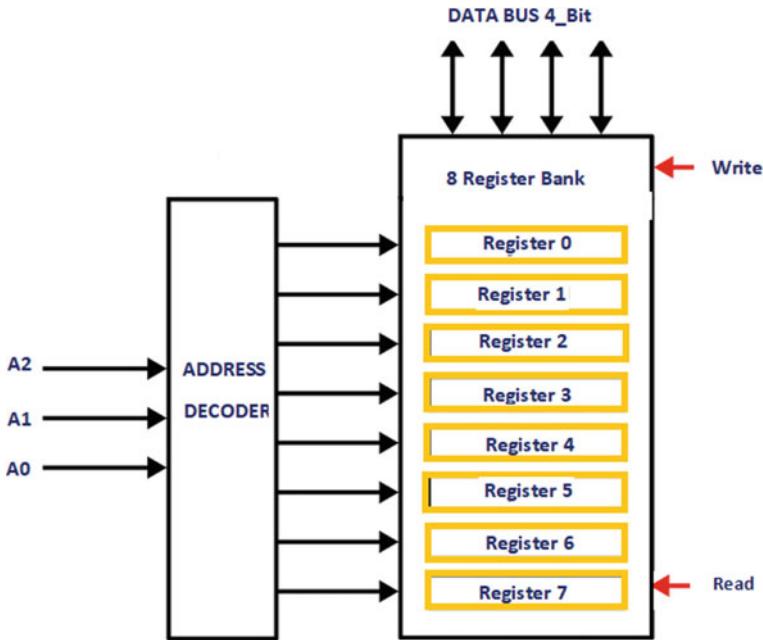
1. The address bus to select one of the registers at a time. As 8-registers we need to have 3 address lines A_2 , A_1 , and A_0 .
2. The data bus to carry data to/from microprocessor. The size of the data bus is 4 bit, and hence, we need to have the 4-bit data bus D_3-D_0 .
3. The read and write control signals to perform either read or write operation at a time.

By using all the above we can identify one of the registers at a time using the 3:8 decoder. The decoder outputs are used as enable input to the respective register (Fig. 3.3).

One of the outputs of the decoder is active at a time and used to select the desired register depending on the status of address bus (Table 3.1). As shown in Table 3.1 for the $A_2 = 1$, $A_1 = 1$, and $A_0 = 0$ the register 6 is enabled and the processor can perform either read or write operation with the register 6.

The main objective of this manuscript is to design the digital systems, architecture of the VLSI chips and micro-architecture using the digital design concepts. To do this in better way the subsequent chapters will discuss about the combinational and sequential elements and the design techniques. The focus is on the area, speed, and power efficient designs.

We will discuss in the subsequent chapters about the.

**Fig. 3.3** System-level design**Table 3.1** Register bank selection

Address bus (A2 A1 A0)	Register selection
000	Register 0
001	Register 1
010	Register 2
011	Register 3
100	Register 4
101	Register 5
110	Register 6
111	Register 7

1. K-map and their use
2. Code converters
3. Arithmetic resources
4. Control elements such as multiplexers, demultiplexers
5. Decoders and encoders
6. Priority design and priority encoders
7. Area and speed optimization
8. Sequential design
9. Area, speed, and power optimization for the sequential design

3.4 Design for the Glue Logic

Most of the time we need to have the glue logic or combinational logic between the modules. Although it is always good practice to use the register boundaries to transfer the data between the modules. That is used in the registered inputs and registered outputs.

Consider the example to design the binary to gray code converter. The simple strategy is use the 4-bit binary as an input and 4-bit gray at output (Table 3.2).

To get the gray outputs consider the binary inputs as B_3, B_2, B_1, B_0 and gray outputs as G_3, G_2, G_1, G_0 . So, if we compare the truth table of the binary and gray then we can arrive with the following Boolean equations to get the 4-bit gray number.

$$G_3 = B_3$$

$$G_2 = \overline{B_3} \cdot B_2 + B_3 \cdot \overline{B_2} = B_3 \oplus B_2$$

$$G_1 = \overline{B_2} \cdot B_1 + B_2 \cdot \overline{B_1} = B_2 \oplus B_1$$

$$G_0 = \overline{B_1} \cdot B_0 + B_1 \cdot \overline{B_0} = B_1 \oplus B_0$$

Table 3.2 4-bit binary and gray code

4-bit binary	4-bit gray
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

But this method to derive the Boolean equations I am not recommending to you. As just by observing the truth table we may land up into the errors in the design.

So, we need to understand the 2-variable, 3-variable, and 4-variable K-maps. So let us understand the 2-variable K-maps and their applications to deduce the Boolean expression.

3.5 Application of 2-variable Karnaugh Maps

As most of us are familiar with the fact that the Karnaugh maps are used to simplify the Boolean expressions. The Karnaugh maps (K-maps) were introduced during year 1953 by Maurice Karnaugh. The K-maps are refinement of the Veitch chart which was rediscovery of the Allan Marquand's 1881 logical diagrams.

Using the K-map we can get the Sum of Product (SOP) as well as Product of Sum (POS) expressions. Important applications of K-maps are listed below.

1. The K-maps are used to get the SOP Boolean expressions by grouping number of 1s.
2. The K-maps are used to get the POS Boolean expressions by grouping number of 1s.
3. The K-maps has the potential to get the Boolean expression with minimum terms.
4. The K-maps are useful to find the essential prime implicants to get the Boolean expressions.

Let us discuss the 2-bit binary to gray code converter design. To get the 2-bit gray output $G1, G0$ from the binary number $B1, B0$ we can use the following steps.

Step 1: Let us document the 2-bit binary and 2-bit gray code entries in Table 3.3

Step 2: Let us use K-maps to deduce the Boolean expression of $G1, G0$.

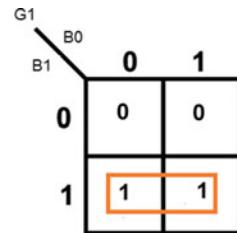
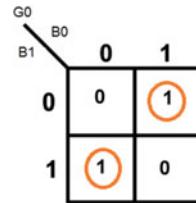
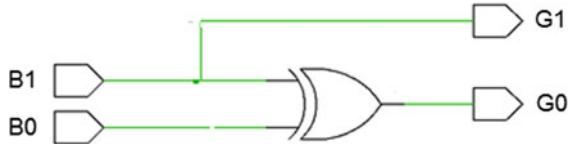
1. K-map for $G1$

As shown in the K-map (Fig. 3.4) the two 1s are grouped together and the Boolean expression for $G1$ is

$$G1 = B1$$

Table 3.3 Two-bit binary to gray codes

2-bit binary ($B1\ B0$)	2-bit gray ($G1\ G0$)
00	00
01	01
10	11
11	10

Fig. 3.4 K-map of $G1$ **Fig. 3.5** K-map for $G0$ **Fig. 3.6** Two-bit binary to gray code design

2. K-map for $G0$

As shown in the K-map (Fig. 3.5) the two 1s cannot be grouped together and the Boolean expression for $G0$ is

$$\begin{aligned} G0 &= \overline{B1} \cdot B0 + B1 \cdot \overline{B0} \\ &= B1 \oplus B0 \end{aligned}$$

3. The logic diagram

Let us sketch the logic diagram for the 2-bit binary to gray code converter (Fig. 3.6).

3.6 Let Us Design Two Variable Function

Let us discuss the 2-bit gray to binary code converter design. To get the 2-bit binary output $B1, B0$ from the gray number $G1, G0$ we can use the following steps.

- Step 1:* Let us document the 2-bit gray and 2-bit binary code entries in Fig. 3.7.
- Step 2:* Let us use K-maps to deduce the Boolean expression of $B1, B0$.

Fig. 3.7 Two-bit gray to binary code

2-bit Gary (G1 G0)	2-bit Binary (B1 B0)
00	00
01	01
11	10
10	11

1. K-map for $B1$

As shown in the K-map (Fig. 3.8) the two 1s are grouped together and the Boolean expression for $B1$ is

$$B1 = G1$$

2. K-map for $B0$

As shown in the K-map (Fig. 3.9) the two 1s cannot be grouped together and the Boolean expression for $B0$ is

$$\begin{aligned} B0 &= \overline{G1} \cdot G0 + G1 \cdot \overline{G0} \\ &= G1 \oplus G0 \end{aligned}$$

Fig. 3.8 K-map of $B1$

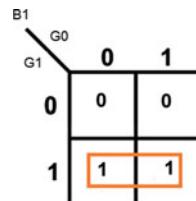


Fig. 3.9 K-map of $B0$

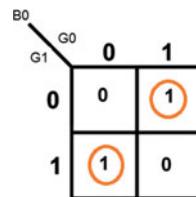
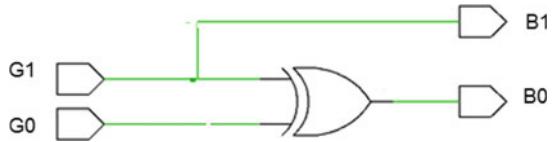


Fig. 3.10 Two-bit gray to binary code converter



3. The logic diagram

Let us sketch the logic diagram for the 2-bit gray to binary code converter (Fig. 3.10).

The binary to gray and gray to binary code converters are useful in the architecture design which has multiple clock domains. More about the multiple clock domain designs is discussed in Chap. 12.

3.7 SOP Terms and Boolean Expression

Let us get the SOP expression for the design shown (Fig. 3.11).

Let us find the Boolean equation for y_1 , y_2 , y_3

$$y_1 = \overline{AB}$$

$$y_2 = \overline{A \cdot \overline{AB}} = \overline{A \cdot (\overline{A} + \overline{B})} = \overline{A \cdot \overline{B}}$$

$$y_3 = \overline{B \cdot \overline{AB}} = \overline{B \cdot (\overline{A} + \overline{B})} = \overline{\overline{B} \cdot \overline{\overline{A}}}$$

$$y = \overline{y_1 \cdot y_2} = \overline{\overline{(A \cdot \overline{B})} \cdot \overline{(B \cdot \overline{A})}}$$

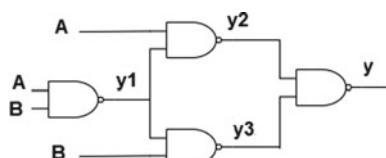
Using De Morgan's theorem that is NAND is equal to bubbled OR we will get

$$y = \overline{\overline{(A \cdot \overline{B})} \cdot \overline{(B \cdot \overline{A})}}$$

$$y = \overline{\overline{(A \cdot \overline{B})}} + \overline{\overline{(B \cdot \overline{A})}}$$

$$y = A \cdot \overline{B} + \overline{A} \cdot B$$

Fig. 3.11 Logic realization using NAND



3.8 POS Terms and Expression

Now let us quickly understand the Product of Sum that is POS expression. We will use the maxterm to get the POS expression. The maxterms are those where we get the output as logic 0.

Consider the SOP expression $y = A + \overline{A} \cdot B$; here the Boolean expression has two min terms.

Now the expression we can split as shown below to get to product terms. The POS expression is shown below

$$y = (A + \overline{A}) \cdot (A + B)$$

The expression simplifies as OR logic function and given below.

$$y = (A + B)$$

3.9 Design of Glue or Combinational Using Minimum Logic Gates

In this section let us try to understand few design scenarios using minimum number of logic gates.

1. **Scenario 1:** Let us design the AND using minimum number of NAND gates (Fig. 3.12).

Let us find the output expression.

$$y_1 = \overline{A \cdot B}$$

$$y = \overline{\overline{A \cdot B}}$$

$$y = A \cdot B$$

The expression $y = A \cdot B$ is Boolean expression of AND gate.

Fig. 3.12 AND using NAND

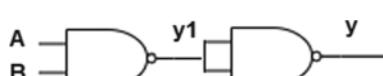
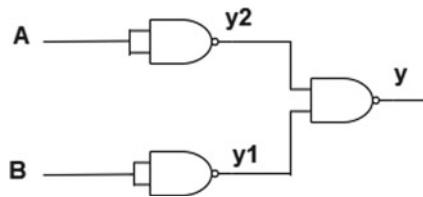


Fig. 3.13 OR using NAND

2. **Scenario 2:** Consider the following digital circuit using 2 input NAND gates (Fig. 3.13).

Let us find the output expression.

$$y2 = \overline{A}$$

$$y1 = \overline{B}$$

$$y = \overline{\overline{y1} \cdot \overline{y2}}$$

$$y = \overline{\overline{B} \cdot \overline{A}}$$

$$y = \overline{\overline{B} + \overline{A}}$$

$$y = A + B$$

The expression $y = A + B$ is Boolean expression of OR gate.

3. **Scenario 3:** Consider the following digital circuit using 2 input NAND gates (Fig. 3.14).

Let us find the output expression.

$$y2 = \overline{A}$$

$$y1 = \overline{B}$$

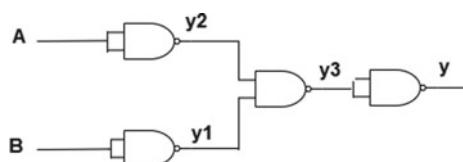
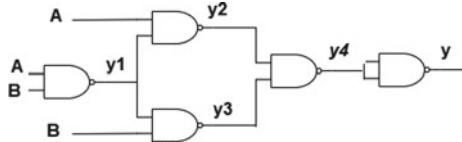
Fig. 3.14 NOR using NAND

Fig. 3.15 XNOR using NAND



$$y_3 = \overline{y_1 \cdot y_2}$$

$$y_3 = \overline{\overline{B} \cdot \overline{A}}$$

$$y_3 = \overline{\overline{B}} + \overline{\overline{A}}$$

$$y_3 = A + B$$

$$y = \overline{y_3}$$

$$y = \overline{A + B}$$

The expression $y = \overline{A + B}$ is Boolean expression of NOR gate.

4. **Scenario 4:** Consider the following digital circuit using 2 input NAND gates (Fig. 3.15).

Let us find the Boolean equation for output y.

$$y_1 = \overline{AB}$$

$$y_2 = \overline{A \cdot \overline{AB}} = \overline{A \cdot (\overline{A} + \overline{B})} = \overline{A \cdot \overline{B}}$$

$$y_3 = \overline{B \cdot \overline{AB}} = \overline{B \cdot (\overline{A} + \overline{B})} = \overline{B \cdot \overline{A}}$$

$$y_4 = \overline{y_2 \cdot y_3} = \left(\overline{\overline{A \cdot \overline{B}}} \cdot \overline{\overline{B \cdot \overline{A}}} \right)$$

Using De Morgan's theorem that is NAND is equal to bubbled OR we will get

$$y_4 = \overline{\overline{(A \cdot \overline{B})} \cdot \overline{\overline{(B \cdot \overline{A})}}}$$

$$y_4 = \overline{\overline{(A \cdot \overline{B})}} + \overline{\overline{(B \cdot \overline{A})}}$$

$$y_4 = A \cdot \overline{B} + \overline{A} \cdot B$$

$$y = \overline{A \cdot \overline{B} + \overline{A} \cdot B}$$

Thus $y = \overline{A \cdot \overline{B} + \overline{A} \cdot B}$.
y is equal to A XNOR B.

From this it is clear that we can use minimum number of NAND gates to implement any of another gate, and hence, the NAND gate is called as universal gate.

3.10 Summary

The following are few of the important points to conclude this chapter.

1. Architecture is block-level representation of the system.
2. Architecture of any design is important milestone and is evolved from the system or design specifications.
3. The micro-architecture is always evolved from the architecture of the design and is sub-block-level representation.
4. The important goal in the architecture design is the logic optimization and area improvements.
5. The speed of the design is the important parameter, and the objective of the architecture design team is to improve the speed using various speed improvement techniques.
6. In most of the VLSI-based applications, we need to have the low power-aware architectures.
7. For any VLSI-based designs the area, speed, and power constraints need to be met.
8. The glue logic glues between two modules.
9. The K-maps are useful to get the Boolean expression for the given Boolean function.
10. The SOP expression uses the min terms and POS expression uses the max terms.
11. The code converters such as binary to gray and gray to binary are useful in the multiple clock domain architecture designs.
12. NAND is universal logic gate, and using minimum number of NAND gates any logic function can be realized.

Chapter 4

Combinational Logic and Design Techniques



The basics of combinational logic design and the various useful techniques are used to design the combinational logic.

The basic design techniques to design the combinational logic are helpful to deduce the Boolean expression. The techniques such as K-maps and design using mux are always helpful during the design phase. In this context the chapter discusses about the various design techniques, arithmetic resources, design using multiplexers, and universal logic. The objective of the design engineer is to have the design with less area, more speed, and less power.

4.1 Let Us Design Few Boolean Functions

In this section let us try to understand how to design the Boolean functions using minimum number of universal gates.

1. **OR gate:** Consider the cascade stage of 2 NOR Gates (Fig. 4.1).

Let us find the output expression.

$$y_1 = \overline{A + B}$$

$$y = \overline{\overline{y_1}}$$

$$y = \overline{\overline{A + B}}$$

$$y = A + B$$

The expression $y = A + B$ is Boolean expression of OR gate.

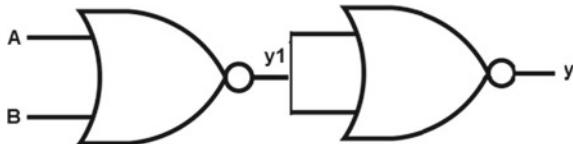


Fig. 4.1 OR using NOR gates

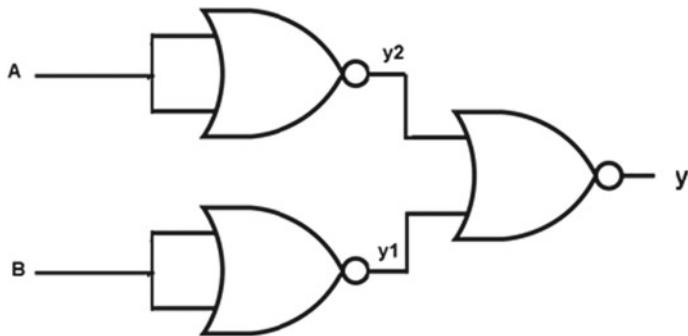


Fig. 4.2 AND using NOR gates

2. **AND Gate:** Consider the following digital circuit using 2-input NOR Gates. The goal is to have the use of the minimum number of NOR gates (Fig. 4.2).

Let us find the output expression.

$$y_2 = \overline{A}$$

$$y_1 = \overline{B}$$

$$y = \overline{y_1 + y_2}$$

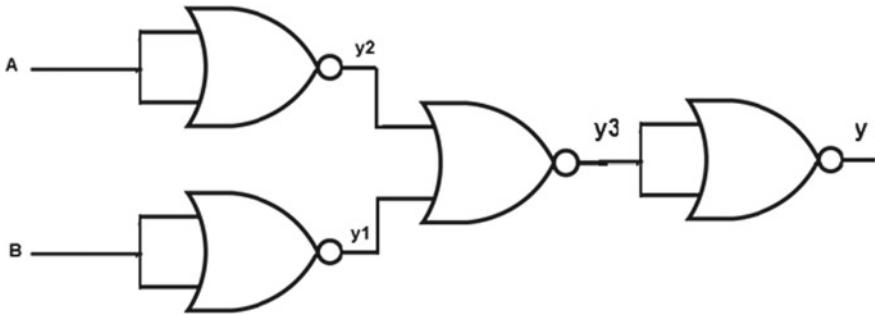
$$y = \overline{\overline{B} + \overline{A}}$$

$$y = \overline{\overline{B}} \cdot \overline{\overline{A}}$$

$$y = A \cdot B$$

The expression $y = A \cdot B$ is Boolean expression of AND gate.

3. **NAND Gate:** Consider the following digital circuit using 2-input NOR Gates. The goal is to have the use of the minimum number of NOR gates to implement the NAND gate (Fig. 4.3).

**Fig. 4.3** NAND using NOR gates

Let us find the output expression.

$$y_2 = \overline{A}$$

$$y_1 = \overline{B}$$

$$y_3 = \overline{\overline{y}_1 + y_2}$$

$$y_3 = \overline{\overline{B} + \overline{A}}$$

$$y_3 = \overline{\overline{B}} \cdot \overline{\overline{A}}$$

$$y_3 = A \cdot B$$

$$y = \overline{y_3}$$

$$y = \overline{A \cdot B}$$

The expression $y = \overline{A \cdot B}$ is Boolean expression of NAND gate.

4. **XOR Gate:** Consider the following digital circuit using 2-input NOR Gates (Fig. 4.4).

Let us find the Boolean equation for y.

$$y_1 = \overline{A}$$

$$y_2 = \overline{B}$$

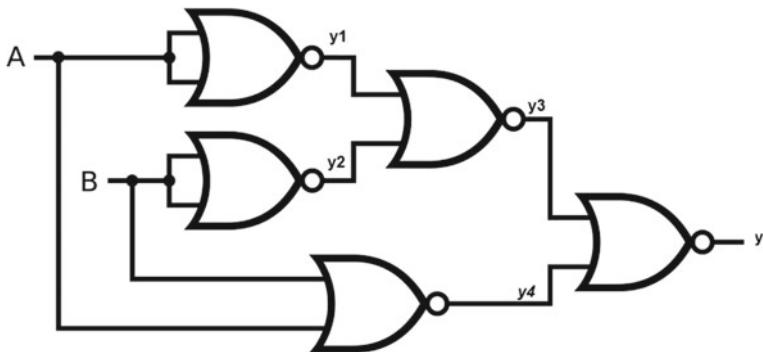


Fig. 4.4 The logic realization using NAND

$$y_3 = \overline{y_1 + y_2} = \overline{(\overline{A} + \overline{B})} = \overline{\overline{A}} \cdot \overline{\overline{B}} = A \cdot B$$

$$y_4 = \overline{A + B} = \overline{A} \cdot \overline{B}$$

$$y = \overline{y_3 + y_4}$$

$$y = \overline{A \cdot B + \overline{A} \cdot \overline{B}}$$

Using De Morgan's theorem, we will get

$$y = \overline{A} \cdot B + A \cdot \overline{B}$$

y is equal to $A \text{ XOR } B$.

5. **XNOR Gate:** Consider the following digital circuit using 2-input NOR Gates (Fig. 4.5).

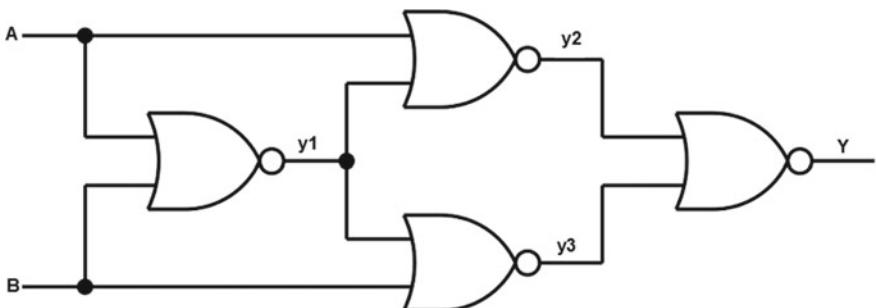


Fig. 4.5 XNOR using NOR gates

Let us find the Boolean equation for y

$$y1 = \overline{A + B}$$

$$y2 = \overline{\overline{A} + \overline{A + B}}$$

Using De Morgan's law, we can get

$$y2 = \overline{A} \cdot (\overline{A + B})$$

$$y2 = \overline{A} \cdot (A + B)$$

$$y2 = \overline{A} \cdot B$$

$$y3 = \overline{B} + \overline{\overline{A} + B}$$

Using De Morgan's law, we can get

$$y3 = \overline{B} \cdot (\overline{\overline{A} + B})$$

$$y3 = \overline{B} \cdot (A + B)$$

$$y3 = \overline{B} \cdot A$$

$$y = \overline{y2 + y3}$$

$$y = \overline{\overline{A} \cdot B + A \cdot \overline{B}}$$

Using De Morgan's theorem we will get

$$y = \overline{A} \cdot \overline{B} + A \cdot B$$

y is equal to A XNOR B.

From this it is clear that we can use minimum number of NOR gates to implement any of another gate, and hence the NOR gate is called as universal gate.

4.2 Arithmetic Resources

The various arithmetic resources such as adders, subtractors, multipliers, and dividers can be designed using the minimum number of logic gates. These resources are used in the processor ALU to perform the desired arithmetic operation. The following section discusses about them in the context of the VLSI design.

This section discusses about the arithmetic resources and their role in the digital design.

4.2.1 Half Adder

As the name indicates this resource is used to perform the addition without any carry input, and hence the resource is named as a half adder. It performs the addition of binary inputs A and B to generate the result at output as SUM and CARRY. The truth table of the half adder is shown in Table 4.1.

Now from the truth table we can get the Boolean function for the SUM and CARRY.

$$\text{SUM}(A, B) = \sum m(1, 2)$$

The K-map of the SUM is shown in Fig. 4.6.

The Boolean equation is derived from the number of 1's which are circled in the K-map. Each term in the Boolean equation is product term, and the expression of

Table 4.1 Half-adder truth table

A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Fig. 4.6 K-map for the SUM

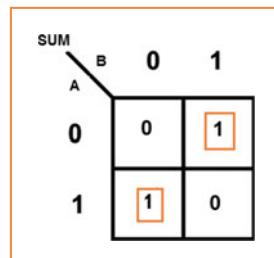


Fig. 4.7 The K-map for the CARRY

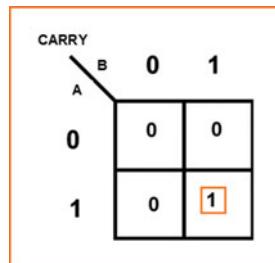
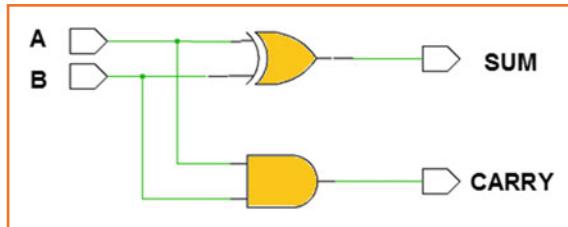


Fig. 4.8 The half adder using logic gates



SUM is called as SOP expression.

$$\text{SUM} = \overline{A} \cdot B + \overline{B} \cdot A$$

$$\text{SUM} = A \oplus B$$

The K-map of the CARRY is shown in Fig. 4.7.

The Boolean equation is derived from the number of 1's which are circled in the K-map. The expression has single product term.

$$\text{CARRY}(A, B) = \sum m(3)$$

$$\text{CARRY} = A \cdot B$$

The design of the half adder is shown in Fig. 4.8.

4.2.2 Half Subtractor

As name indicates the half subtractor is used to perform the subtraction of binary inputs A and B and it generates the output as difference (Diff) and Borrow. The truth table of the half subtractor is shown in Table 4.2.

Now let us use the truth table to get the Boolean function for the Diff and Borrow.

Table 4.2 Half-subtractor truth table

<i>A</i>	<i>B</i>	<i>B</i>	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

$$\text{Diff} (A, B) = \sum m(1, 2)$$

The K-map of the Diff is shown in Fig. 4.9.

The Boolean equation is derived from the number of 1's which are circled in the K-map. The Boolean equation for the Diff is SOP expression and has two product terms. The function is XOR function to generate the difference.

$$\text{Diff} = \overline{A} \cdot B + \overline{B} \cdot A$$

$$\text{Diff} = A \oplus B$$

The K-map of the Borrow is shown in Fig. 4.10.

The Boolean equation is derived from the number of 1's which are circled in the K-map. The borrow equation has single product term.

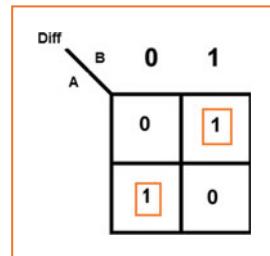
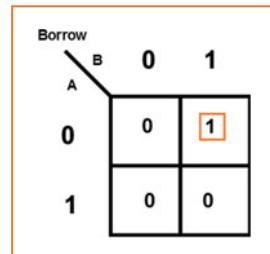
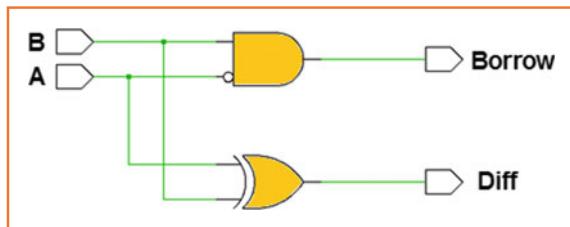
Fig. 4.9 The K-map for the Diff**Fig. 4.10** The K-map for the Borrow

Fig. 4.11 Half subtractor using logic gates



$$\text{Borrow } (A, B) = \sum m(1)$$

$$\text{Borrow} = \overline{A} \cdot B$$

The design of the half subtractor is shown in Fig. 4.11.

In the VLSI Design Context, the Following are Few of the Important Points While Using Adders

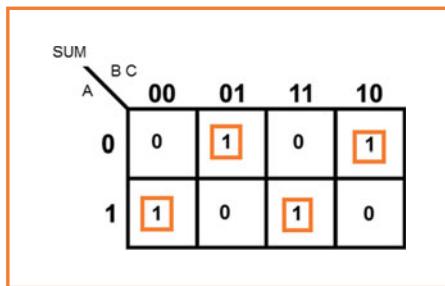
1. *The adders consume more areas.*
2. *The sum and difference functionality are designed using minimum number of XOR gates.*
3. *The gates cascaded will incur more propagation delay, and hence the designer should avoid the cascade stages.*

4.2.3 Full Adder

The full adder uses the binary inputs A and B and carry input C and performs the addition to generate the result as SUM and CARRY (Table 4.3).

Table 4.3 The truth table of full adder

A	B	C	SUM	CARRY
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

Fig. 4.12 K-map of SUM**Step 1: Let us find the SUM expression**

The K-map of SUM is shown in Fig. 4.12, and as shown the number 1's is circled. By using the number of 1's circled the Boolean expression is derived as below.

$$\begin{aligned}
 \text{SUM} &= \overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot B \cdot \overline{C} + A \cdot \overline{B} \cdot \overline{C} + A \cdot B \cdot C \\
 &= \overline{A} \cdot (\overline{B} \cdot C + B \cdot \overline{C}) + A \cdot (\overline{B} \cdot \overline{C} + B \cdot C) \\
 &= \overline{A} \cdot (B \oplus C) + A \cdot (\overline{B} \oplus C) \\
 &= A \oplus B \oplus C
 \end{aligned}$$

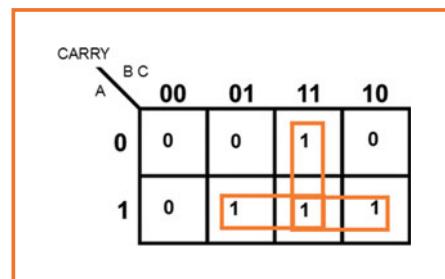
The SUM expression is XOR of A , B , and C , and if we use 2-input XOR gates, then we need two XOR gates to have the SUM functionality.

Step 2 : Let us find the CARRY Expression

The K-map of CARRY is shown in Fig. 4.13, and as shown the number 1's are circled.

By using the group of 1's we can get the product terms and the Boolean expression is deduced as below.

$$\begin{aligned}
 \text{CARRY} &= A \cdot B + A \cdot C + B \cdot C \\
 &= A \cdot B + (A + B) \cdot C
 \end{aligned}$$

**Fig. 4.13** K-map of CARRY

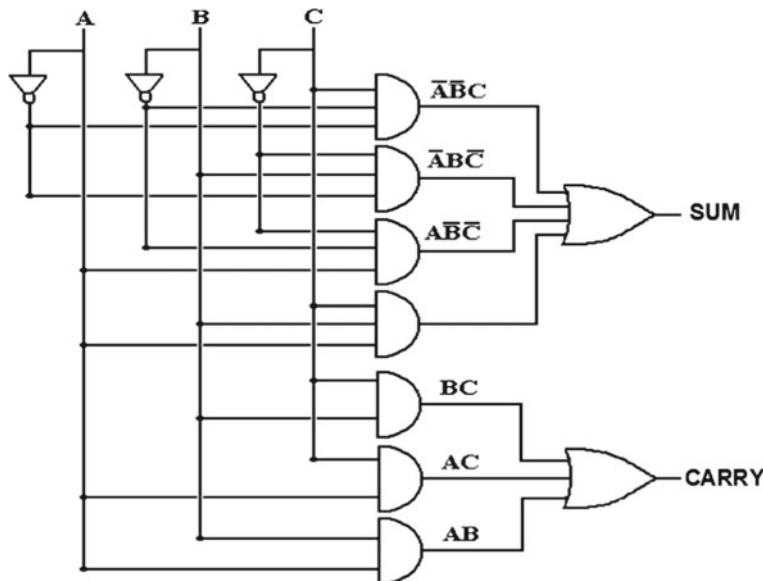


Fig. 4.14 Full adder using logic gates

The resources needed to implement the full adder are two XOR gates and three 2-input AND gates and three input single OR gate. So, this is the gate-level design and has more area. So let us design the full adder using the half adders and minimum additional logic gates (Fig. 4.14).

4.2.4 Full Adder Using Half Adders

Now let us think about the design of the full adder using the following components

1. Half adder
2. OR gate

The full adder can be designed using the half adders connected in cascade. To generate the carry output, use the OR gate. The design of the full adder using the two half adders and OR gate is shown in Fig. 4.15.

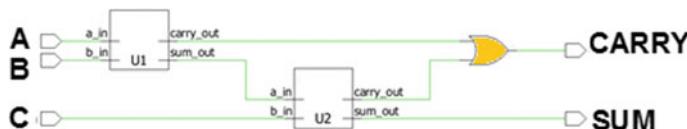
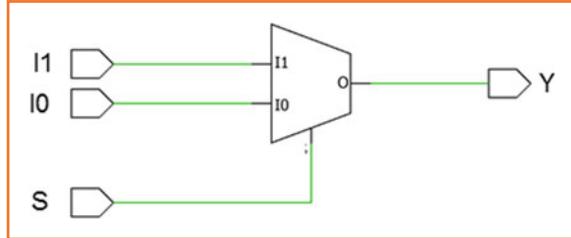


Fig. 4.15 Full adder using half adders and OR gate

Table 4.4 Truth table of 2:1 MUX

S	Y
0	I0
1	I1

Fig. 4.16 Symbolical representation of 2:1 MUX



4.3 Role of Data Control Elements

Multiplexer is many to one switch and has many inputs and single output. The status on the select inputs decides the output. For more details about the multiplexers please refer this chapter. Consider 2:1 mux having inputs I_0 and I_1 and select input S . The output Y of mux is I_0 for $S = 0$, and for the $S = 1$ output is I_1 (Table 4.4).

The symbol of the 2:1 mux is shown in Fig. 4.16 and as shown depending on the status of S it passes one of the inputs either I_0 or I_1 to an output Y .

The multiplexers are used as data control elements or data selection elements and treated as universal logic elements. Let us discuss the design to perform the addition and subtraction. We can use the arithmetic resources and multiplexers to implement the adder–subtractor.

4.4 The Multi-Bit Adder and Subtractor

We need to have the multi-bit adders and subtractors as dedicated resources to perform the addition and subtraction, respectively. Consider the 4-bit processor and has the instructions as addition and subtraction. Then the design strategy is use of the adder chain in one of the data paths and use of the subtractor chain in the other data path. To get one of the operations' results we can use the chain of multiplexers at the output.

Consider the design of the 4-bit adder and subtractor. The operations are listed in Table 4.5.

Table 4.5 The 4-bit addition and subtraction truth table

Operation	Description	Expression
Addition	Unsigned addition of A, B	$A + B$
Subtraction	Unsigned subtraction of A, B	$A - B$

To design the 4-bit adder and subtractor use the architecture level design understanding and document the resources

1. 4-bit adder
2. 4-bit subtractor
3. Chain of four, 2:1 mux to get sum or difference output
4. Chain of four, 2:1 mux to get the carry or borrow output.

The design is shown in Fig. 4.17 and has the above-mentioned resources as discussed earlier.

The issue in the design is more area as two separate data paths. Another important issue is that the design performs both operations addition and subtraction at a time and hence more power. The area and power optimization are the main goal of the logic design engineer, and it can be achieved by using the resource sharing concepts. For more details, please refer the subsequent chapters on the area and power optimization.

4.5 The Multi-Bit Adder with Area Optimization

We have discussed about the arithmetic resources such as half adder, full adder, and half subtractor. Now consider the practical scenario to perform the addition and subtraction of 4-bit numbers. What we need to use is 4-bit adder, 4-bit subtractor, and the selection logic to select the result of either adder or subtractor.

This approach is not helpful to minimize area; hence we can use common resource as adder to perform the subtraction. The subtraction is performed using 2's complement addition (Table 4.6).

So, the addition is performed when operational code or control bit is 0, $Y = A + B + 0$. The subtraction is performed when the operational code or control bit is logic 1, $Y = A + \sim B + 1$.

What we need is multi-bit adder as common resource and the additional logic gates to control the input of the adder. If you observe the table entries, then we can conclude that for both the operations one of the inputs of adder is always A . The other input of adder is either B or complement of B depending on the operation. So, if we use the logic gate as XOR having one input as B and other input as control bit then we can get B or B complement depending on the status of the control bit.

Figure 4.18 is the design of the 4-bit adder and subtractor.

The design shown in Fig. 4.18 has significant improvement in the area and power as common resources adders are shared, and additional gates are used at the input side to eliminate the chain of output multiplexers. This technique is called as resource sharing and useful in the area optimization.

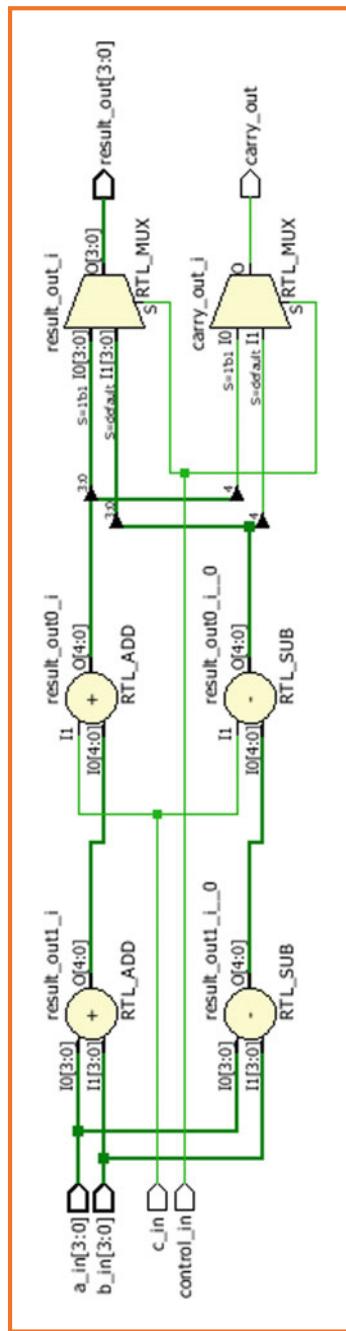


Fig. 4.17 The 4-bit adder and subtractor without area optimization

Table 4.6 The multi-bit adder and subtractor using resource sharing

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

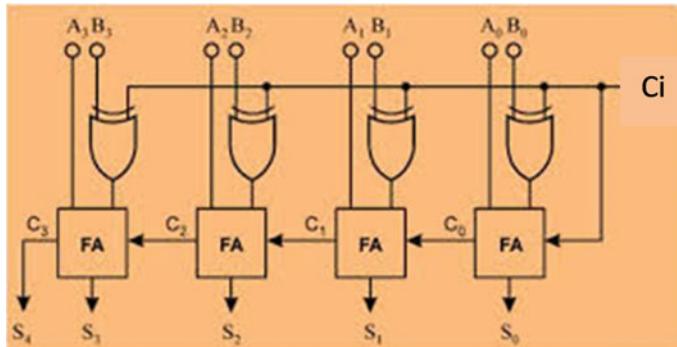


Fig. 4.18 The 4-bit adder and subtractor with area optimization

4.6 3-Variable K-Map and Code Converters

Now let us discuss the 3-variable K-map and its role in the design.

4.6.1 3-bit Binary to Gray Code Converter

The 3-bit binary to gray code truth table (Table 4.7) is shown and has $B2, B1$, and $B0$ as binary number and $G2, G1$, and $G0$ as gray number.

Table 4.7 The 3-bit binary and gray code

3-bit binary code $B2\ B1\ B0$	3-bit gray code $G2\ G1\ G0$
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

The design of the 3-bit binary to gray code converter is discussed in this section using 3-variable K-map. As 8 entries of the binary and gray code let us use the 3-variable K-map to deduce the equations for $G2$, $G1$, and $G0$.

As the design is combinational logic $G2$, $G1$, and $G0$ is function of the $B2$, $B1$, and $B0$. That is output is function of the present inputs.

1. As there is group of four 1's the logic expression using the K-map for $G2$ (Fig. 4.19) is

$$G2 = B2$$

2. Let us group the terms as shown in the K-map (Fig. 4.20). The logic expression is

$$G1 = \overline{B2} \cdot B1 + \overline{B1} \cdot B2$$

$$G1 = B2 \oplus B1$$

3. Let us group the terms as shown in the K-map (Fig. 4.21). The logic expression is

$$G0 = \overline{B1} \cdot B0 + \overline{B0} \cdot B1$$

Fig. 4.19 K-map for $G2$

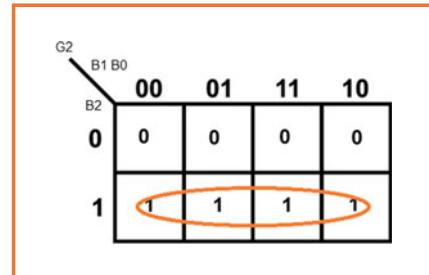


Fig. 4.20 K-map for $G1$

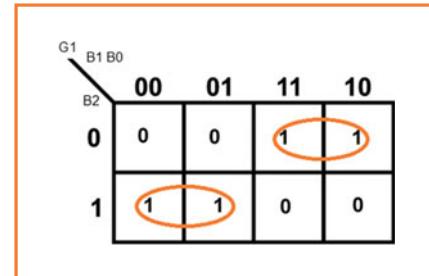
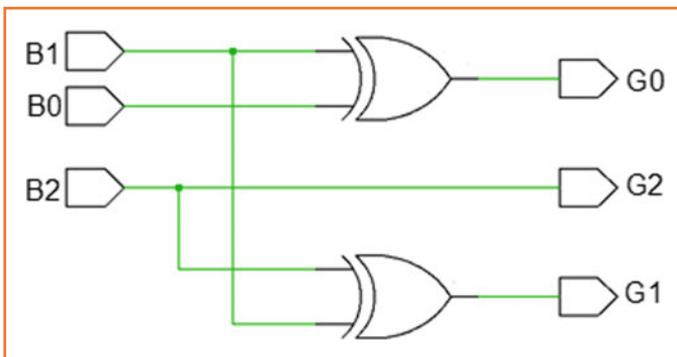
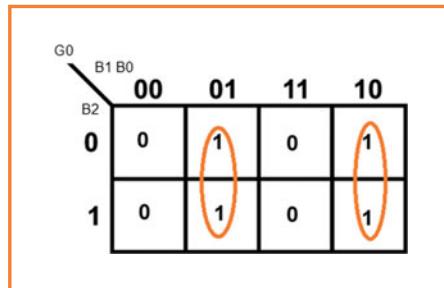


Fig. 4.21 K-map for G0**Fig. 4.22** The 3-bit binary to gray code converter

$$G0 = B1 \oplus B0$$

From the above Boolean expressions, it is clear that to implement the 3-bit binary to gray code converter we need to have the 2-XOR gates. 4.11 the 3-bit binary to gray code converter is shown in Fig. 4.22.

4.6.2 3-Bit Gray to Binary Code Converter

The 3-bit gray to binary code converter truth table (Table 4.8) gives the relationship between the 3-bit gray code (G_2 , G_1 , and G_0) and binary code (B_2 , B_1 , and B_0).

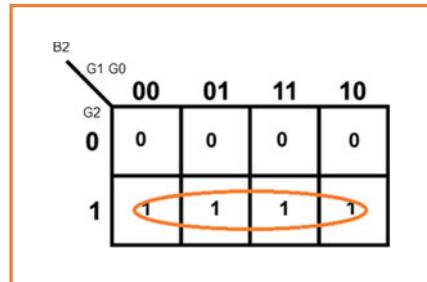
The design of the 3-bit gray to binary code converter is discussed in this section. As 8 entries of the gray and binary number let us use the 3-variable K-map to deduce the equations for B_2 , B_1 , and B_0 .

As a combinational logic the output of code converter is function of the present input. That is the B_2 , B_1 , and B_0 is function of the G_2 , G_1 , and G_0 .

Table 4.8 The 3-bit gray and binary codes

3-bit gray code $G2\ G1\ G0$	3-bit binary code $B2\ B1\ B0$
000	000
001	001
011	010
010	011
110	100
111	101
101	110
100	111

Fig. 4.23 The K-map for the $B2$



- As group of four 1's the logic expression using the K-map for $B2$ (Fig. 4.23) is

$$B2 = G2$$

- Let us group the terms as shown in the K-map (Fig. 4.24). The logic expression is

$$B1 = \overline{G2} \cdot G1 + \overline{G1} \cdot G2$$

$$B1 = G2 \oplus G1$$

- Let us group the terms as shown in the K-map (Fig. 4.25). The logic expression is

$$B0 = \overline{G2} \cdot \overline{G1} \cdot G0 + \overline{G2} \cdot \overline{G0} \cdot G1 + G2 \cdot \overline{G1} \cdot \overline{G0} + G2 \cdot G1 \cdot G0$$

$$B0 = \overline{G2} \cdot (\overline{G1} \cdot G0 + \overline{G0} \cdot G1) + G2 \cdot (\overline{G1} \cdot \overline{G0} + G1 \cdot G0)$$

$$B0 = G2 \oplus G1 \oplus G0$$

Fig. 4.24 The K-map for the B_1

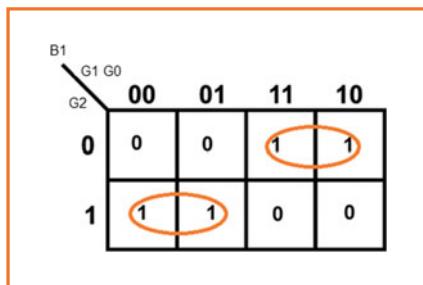
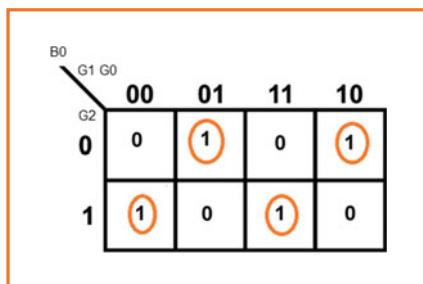


Fig. 4.25 The K-map for the B_0



To implement the 3-bit gray to binary code converter we need to have the 2-XOR gates. The design of the 3-bit binary to gray code converter is shown in Fig. 4.26.

The gray codes are used in the multiple clock domain designs as in the successive gray codes only one-bit changes. For more information about the gray codes please refer the subsequent chapters.

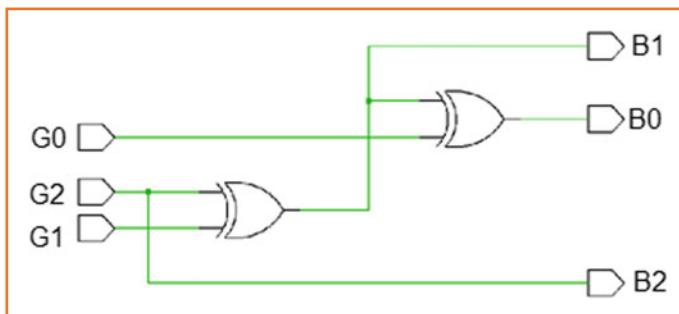


Fig. 4.26 The 3-bit gray to binary code converter

4.7 Summary

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

1. NOR gate is universal gate; as using minimum number of NOR gates we can realize any Boolean function.
2. The arithmetic resources are adders and subtractors and used to perform the arithmetic operations.
3. The subtraction is 2's complement addition.
4. The common resources can be shared to optimize the area.
5. Full adder can be designed by using half adders and additional OR gate.
6. In the design avoid the cascade stages as it will incur more propagation delay.
7. Multiplexers are universal logic and used to implement any kind of Boolean function.
8. In two consecutive gray numbers only one-bit changes.
9. Gray codes are used in the multiple clock domain designs.

Chapter 5

Data Control Elements and Applications



The various data control elements their understanding and applications are useful during the architecture design phase.

In the previous few chapters, we have discussed about the various combinational elements, arithmetic resources, and the basics of the combinational design techniques. In this chapter let us discuss about the various data control elements and their role in the design. The chapter focuses on the multiplexers and their role in the design.

5.1 Data and Control Paths in Design

Most of the time we use the terms such as data path and control path, and the architecture design engineer need to consider the better data and control path during the architecture design. The data path name itself indicates that the path is used to carry the data. For example, in the multiple clock domain design we have the FIFO to carry the data from the sender clock domain to receiver clock domain. We treat the FIFO as data path synchronizer.

Figure 5.1 shows various parallel paths in the design to improve the parallelism. Similarly, we can have various data and control paths in any of the design.

The control path is used to carry the multiple control signals, and these control signals are useful to control the various read and write transactions, for example, the control signal generation for the multiple clock domain designs. Considering the read and write control signals need to pass between the multiple clock domain designs, we will use the control signal generators or FSM-based controllers in the control path. We can treat even the various synchronizers used to carry the control signals as the control path elements.

Now by considering the need of the data and control path-based designs and the applications in the VLSI domain, let us discuss in detail the various elements such as multiplexers, demultiplexers, decoders, encoders, and priority logic in this chapter.

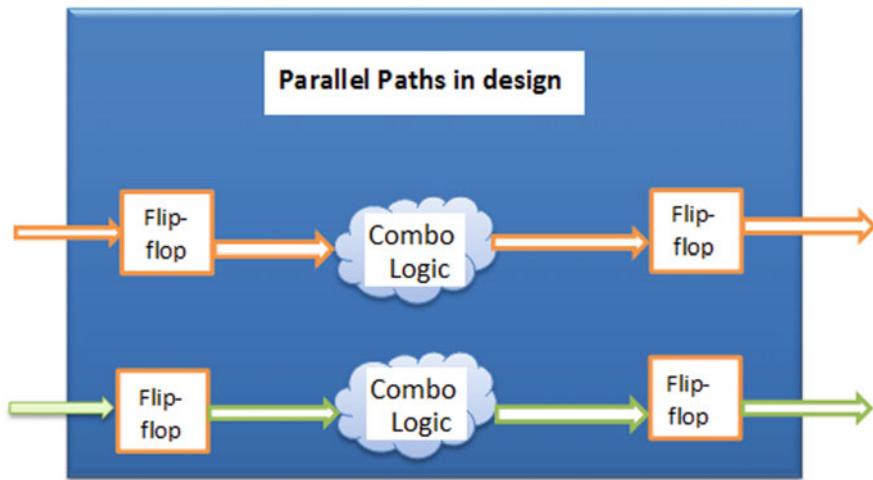


Fig. 5.1 Design having parallel paths

5.2 Multiplexers to Control the Data

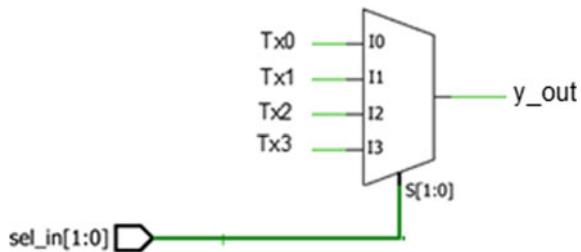
The multiplexer is universal logic and has many inputs and single output. It is also called as many to one switch, and they are used extensively in various VLSI applications such as

1. Bus multiplexing
2. Clock multiplexing
3. As a universal logic to implement the combinational logic
4. As a data control element in the transmitter channels.

As we know, the multiplexer has many inputs and single output. To control the data transfer the multiplexer has select inputs and the status of the select inputs decides the data transfer. If we have m inputs and n select lines, then the relationship between the inputs and select lines is given by $m = 2^n$.

The multiplexer is also treated as universal logic and named as mux throughout this book. Consider the 4:1 mux as shown in Fig. 5.2.

Fig. 5.2 The 4:1 multiplexer



As shown the multiplexer has 4 inputs and 2 select lines. The 4 is equal to 2 to the power 2, and mux has single output line.

1. Consider 2:1 mux, the mux has 2 input lines and single select line.
2. For 4:1 mux, as the $4 = 2^2$ the mux has 4 input lines and 2 select lines.
3. For 8:1 mux, as the $8 = 2^3$ the mux has 8 input lines and 3 select lines.
4. For 10:1 mux, as the 10 is greater than 8, $8 = 2^3$ the mux has 10 input lines and 4 select lines.

5.3 Lowest Order Mux in the Design

In the design we use the 2:1 mux as the lowest order mux. The 2:1 mux has 2 inputs: single select line and single output line. Consider inputs as A and B and select line as S , the truth table is documented as shown below. As described the output of mux is B for $S = 1$, for the $S = 0$ output is A (Table 5.1).

The symbolic representation of the 2:1 mux is shown in Fig. 5.3; as shown, output Y is function of the select input S and inputs A, B .

Now, let us implement the gate-level design of 2:1 mux. From the truth table we can get the product term and we can use the Sum of Product (SOP) expression to implement the 2:1 mux (Table 5.2).

Using the product term, the SOP expression for the 2:1 mux is given by

$$Y = Y_0 + Y_1$$

Table 5.1 Truth table of 2:1 MUX

S	Y
0	A
1	B

Fig. 5.3 Symbolical representation of 2:1 MUX

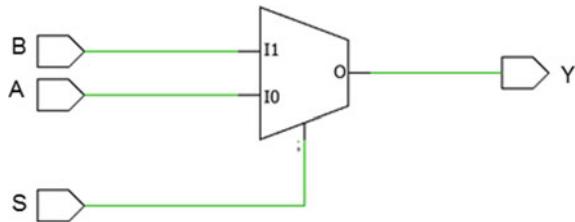


Table 5.2 The 2:1 mux product terms

S	Y	Product term
0	A	$Y = \bar{S} \cdot A$
1	B	$Y = S \cdot B$

$$Y = \overline{S} \cdot A + S \cdot B$$

5.4 The 2:1 MUX Using NAND

As we are familiar with the fact that the NAND is the universal gate, we can implement any logic function using minimum number of NAND gates. Now let us implement 2:1 mux using minimum number of 2-input NAND gates.

What should be our strategy to implement the 2:1 mux. Use the following steps to implement the 2:1 mux using minimum number of NAND gates.

1. *Step I:* Let us use the 2:1 mux design from Figs. 5.4 and 5.5.
2. *Step II:* To get the realization of 2:1 mux using NAND gates use the bubbles at the output of each AND gate. As we know that bubble indicates complement or NOT, we will be able to get NAND gate. But this will change the Boolean function so place bubble at each input of OR gate. This will give us same Boolean function as even NOT gates in cascade will give output same as input (Fig. 5.6).
3. *Step III:* Use De Morgan's theorem. As we know bubbled OR is equal to NAND replace the bubbled OR using 2-input NAND (Fig. 5.7).
4. *Step IV:* Replace the bubble shown in the yellow box by using NOT gate which is designed by using 2-input NAND (Fig. 5.8).

Fig. 5.4 Gate level structure of 2:1 MUX

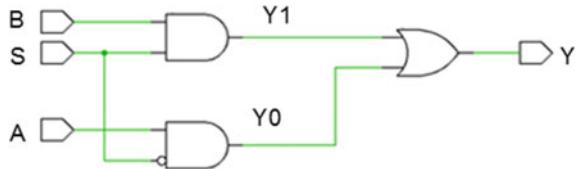


Fig. 5.5 The 2:1 mux using logic gates

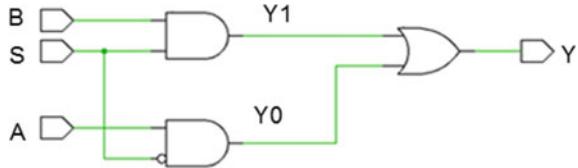


Fig. 5.6 Modifications in 2:1 mux design

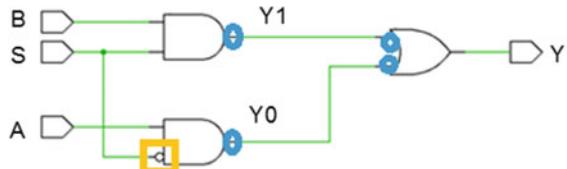


Fig. 5.7 The 2:1 mux using NAND and NOT

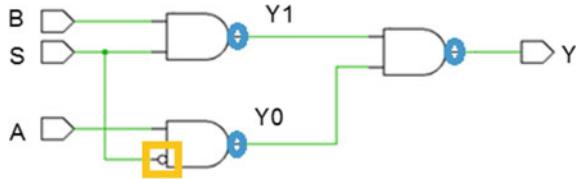
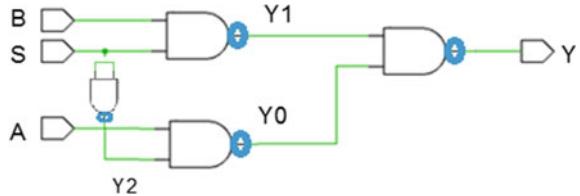


Fig. 5.8 The 2:1 mux using minimum number of NAND gates



5. *Step V:* Now let us deduce the expression for output Y .

$$Y_2 = \bar{S}$$

$$Y = Y_1 \cdot Y_0$$

$$Y_0 = \overline{A \cdot Y_2}$$

$$Y_0 = \overline{A \cdot \bar{S}}$$

$$Y_1 = \overline{\bar{S} \cdot B}$$

$$Y = \overline{Y_1 \cdot Y_0}$$

$$Y = \overline{(S \cdot B)} \cdot \overline{(A \cdot \bar{S})}$$

Using De Morgan's theorem, we can get

$$Y = \overline{\overline{\overline{S}} \cdot \overline{B}} + \overline{\overline{\overline{A}} \cdot \overline{\overline{\bar{S}}}}$$

Thus, we will get

$$Y = S \cdot B + \bar{S} \cdot A$$

Table 5.3 Truth table of 4:1 mux

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Thus, we need to have 4 NAND gates to implement the 2:1 mux.

5.5 The 4:1 MUX

The lowest order multiplexer we can use to design any of the highest order multiplexers. Consider the 4:1 mux (Table 5.3) which has two select inputs S_1 and S_0 , and an output is function of the inputs I_0 , I_1 , I_2 , and I_3 , and select lines S_1 and S_0 . Now let us use the minimum number of 2:1 mux to implement the 4:1 mux

The logic expression of 4:1 mux we can deduce by using following product terms.

For $s_1 = 0, s_0 = 0$ the product term is $I_0 \cdot \overline{S_1} \cdot \overline{S_0}$.

For $s_1 = 0, s_0 = 1$ the product term is $I_1 \cdot S_1 \cdot \overline{S_0}$.

For $s_1 = 1, s_0 = 0$ the product term is $I_2 \cdot S_1 \cdot \overline{S_0}$.

For $S_1 = 1, s_0 = 1$ the product term is $I_3 \cdot S_1 \cdot S_0$.

The SOP expression of 4:1 mux is given by

$$Y = Y_0 + Y_1 + Y_2 + Y_3$$

$$Y = I_0 \cdot \overline{S_1} \cdot \overline{S_0} + I_1 \cdot \overline{S_1} \cdot S_0 + I_2 \cdot S_1 \cdot \overline{S_0} + I_3 \cdot S_1 \cdot S_0$$

5.6 The Design of 4:1 Mux Using 2:1 Mux

Now let us realize the 4:1 mux using minimum number of 2:1 multiplexer. To design the 4:1 mux using minimum number of 2:1 mux let us partition the 4:1 mux table into three sections as shown in Table 5.4. From the partition we need to have three 2:1 mux to implement the 4:1 mux.

From Table 5.4 it is clear that we need to have three 2:1 mux to implement the 4:1 mux. Let us now document the entries of Fig. 5.9 as shown below to get the output multiplexer (Fig. 5.10).

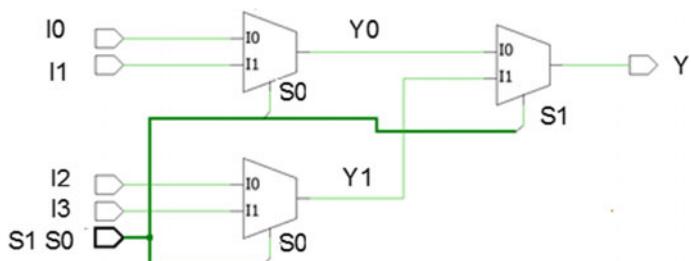
Another strategy in the design is the use of the logic expression to get the number of 2:1 mux needed to implement the design

Table 5.4 Truth table to implement the 4:1 mux using 2:1 multiplexer

S1	S0	Y	
0	0	I0	2:1 MUX
	1	I1	
1	0	I2	2:1 MUX
	1	I3	

2:1 MUX

S1	Y
0	y0
1	y1

Fig. 5.9 The output mux entries**Fig. 5.10** The 4:1 MUX using only 2:1 multiplexers

$$Y = I0 \cdot \overline{S1} \cdot \overline{S0} + I1 \cdot \overline{S1} \cdot S0 + I2 \cdot S1 \cdot \overline{S0} + I3 \cdot S1 \cdot S0$$

$$Y = \overline{S1}(I0 \cdot \overline{S0} + I1 \cdot S0) + S1 \cdot (I2 \cdot \overline{S0} + I3 \cdot S0)$$

$$Y = Y0 + Y1$$

5.7 Design Using Multiplexers

So as discussed in the previous few sections we can use the minimum number of 2:1 mux in the design of the combinational logic. In the VLSI context we can use the minimum number of multiplexers to implement the following:

- Combinational design
- Clock multiplexing
- Address data bus multiplexing
- Pin multiplexing

The objective of the logic designer is to use the minimum number of 2:1 mux. Even the cascade multiplexer stages reduce the speed due to increase in propagation delay.

5.7.1 NOT Using 2:1 Mux

Now let us design the NOT gate using single 2:1 mux. Use the following steps.

1. *Step I:* Let us tabulate entries to have NOT of b_{in} (Table 5.5).
2. *Step II:* Let us document the entries of 2:1 mux. Consider for $S = 0$ $Y = A$ and for $S = 1$ $Y = B$ (Table 5.6).

Table 5.5 The NOT gate truth table

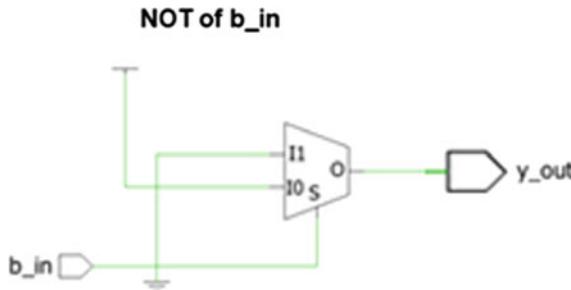
b_{in}	y_{out}
0	1
1	0

Table 5.6 The truth table of 2:1 mux

S	Y_{out}
0	$I0$
1	$I1$

Table 5.7 The truth table of NOT using 2:1 mux

$S = b_{\text{in}}$	Y_{out}
0	$I0 = 1$
1	$I1 = 0$

**Fig. 5.11** NOT using single 2:1 mux

3. *Step III:* Compare the Table entries of step I and step II and document the entries as shown below (Table 5.7).
4. *Step IV:* Sketch the logic diagram (Fig. 5.11)

Thus, we need to have single 2:1 mux to get the NOT output.

5.7.2 NAND Using Mux

Now let us use the strategy explained in the above section to implement the 2-input NAND using minimum number of 2:1 multiplexers. Table 5.8 has four entries, and it is divided into two groups. For first two entries that is $a_{\text{in}} = 0$ if we compare b_{in} with y_{out} then we are getting an output $y_{\text{out}} = 1$. For the next two entries if we compare b_{in} with y_{out} of NAND then we get $y_{\text{out}} = \overline{b_{\text{in}}}$ (Table 5.9).

Now most of the time the beginners conclude that to implement the 2-input NAND gate we need to have single 2:1 mux (Fig. 5.12) but that is not correct. As shown for the select input $a_{\text{in}} = 1$ an output $y_{\text{out}} = \overline{b_{\text{in}}}$. So, to implement the NOT of b_{in} we need to have one more multiplexer.

The implementation is shown in Fig. 5.13.

5.7.3 NOR Using 2:1 Mux

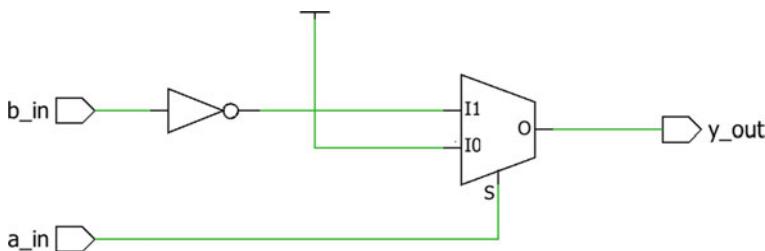
Now let us use the strategy explained in the above section to implement the 2-input NOR using minimum number of 2:1 multiplexers. Table 5.10 has four entries, and it

Table 5.8 Truth table of 2-input NAND gate

a_in	b_in	y_out
0 0	0	1
1	1	0

Table 5.9 The 2-input NAND using multiplexer truth table

sel_in = a_in	y_out
0	1
1	\bar{b}_{in}

**Fig. 5.12** NAND using MUX and NOT

is divided into two groups. For first two entries that is $a_{in} = 0$ if we compare b_{in} with y_{out} then we are getting an output $y_{out} = \bar{b}_{in}$. For the next two entries if we compare b_{in} with y_{out} of NAND then we get $y_{out} = 0$.

To implement the 2-input NOR gate we need to have single 2:1 mux and NOT gate (Fig. 5.14) but that is not correct approach as our objective is to implement the 2-input NOR using minimum number of 2:1 mux. As shown in Table 5.11 for the select input $a_{in} = 0$ an output $y_{out} = \bar{b}_{in}$. So, to implement the NOT of b_{in} we need to have one more multiplexers.

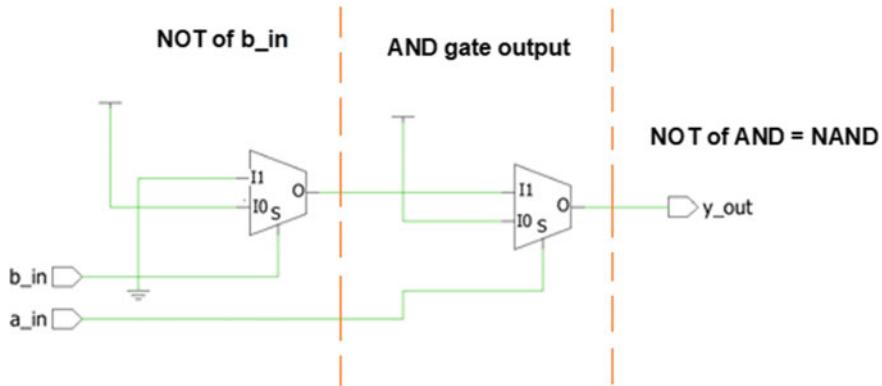


Fig. 5.13 NAND using only multiplexers

Table 5.10 Truth table of 2-input NOR gate

a_{in}	b_{in}	y_{out}
0 0	0	1
1 1	1	0

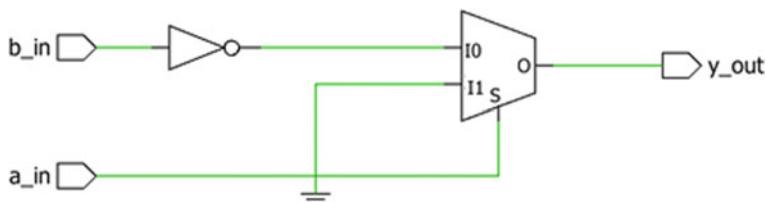


Fig. 5.14 NOR using NOT and 2:1 MUX

Table 5.11 The 2-input NOR table entries

sel_in = a_in	y_out
0	$\overline{b_in}$
1	0

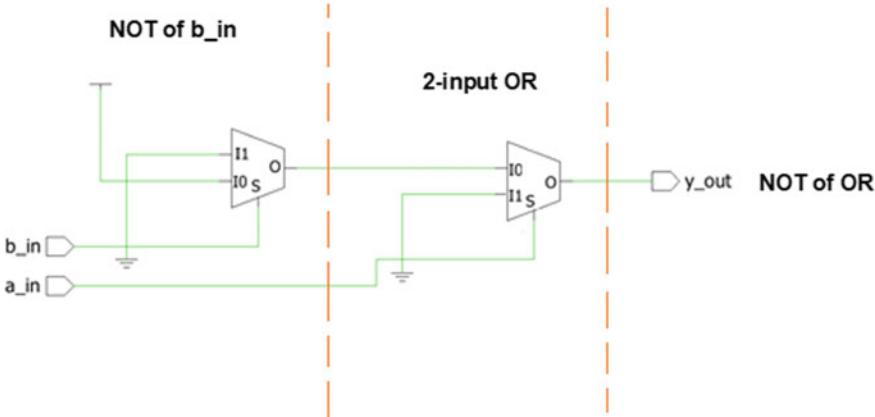


Fig. 5.15 NOR using multiplexers

The implementation of 2-input NOR using only minimum number of 2:1 mux is shown in Fig. 5.15.

As we can design the NAND and NOR universal logic gates using the 2:1 multiplexers we can conclude that any logic function we can implement using minimum number of 2:1 multiplexers. Hence multiplexers are treated as universal logic.

5.7.4 Design of XOR Gate to Get the SUM Output Using Mux

Let us design the SUM function of the half adder using minimum number of 2:1 mux. Now the better strategy is to use the truth table of XOR gate (Table 5.12). As shown the number of entries is 4, and as $a_{in} = 0$ for first two entries and $a_{in} = 1$ for the next two entries to realize XOR using the minimum number of 2:1 mux let us compare b_{in} with y_{out} of XOR gate.

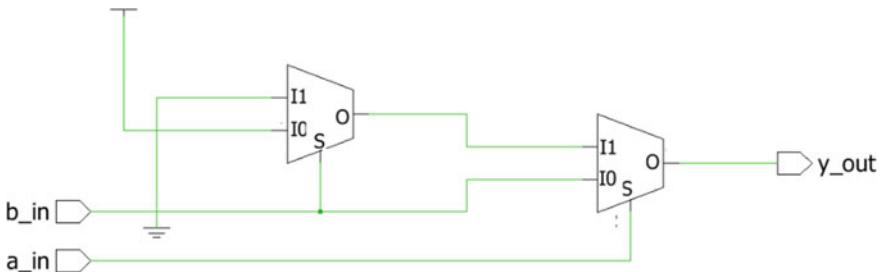
As shown (Table 5.13) for $a_{in} = 0$, $y_{out} = b_{in}$ and for $a_{in} = 1$, $y_{out} = \overline{b_{in}}$. The logic using two 2:1 multiplexers is shown in Fig. 5.16.

Table 5.12 The XOR gate truth table

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

Table 5.13 The truth table for realization of the XOR gate

a_in	y_out
0	b_{in}
1	\bar{b}_{in}

**Fig. 5.16** XOR gate using 2:1 MUX

5.7.5 Design of XNOR Gate to Get the Even Parity

Let us design the parity detector to detect the parity of 2-bit binary number. Output 1 indicates even number of 1's, and output 0 indicates odd number of 1's. This indicates the logic function as XNOR of the data inputs.

Now the better strategy is to use the truth table of XNOR gate (Table 5.14). As shown the number of entries is 4, and as $a_{in} = 0$ for first two entries and $a_{in} = 1$ for next two entries to realize XNOR using the minimum number of 2:1 mux, let us compare b_{in} with y_{out} of XNOR gate.

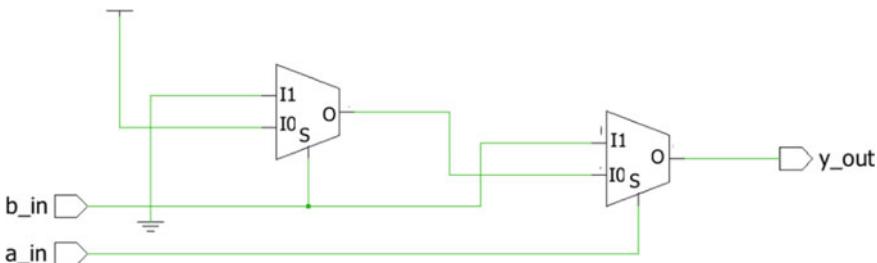
As shown (Table 5.15) for $a_{in} = 0$, $y_{out} = \bar{b}_{in}$ and for $a_{in} = 1$, $y_{out} = b_{in}$. The logic using two 2:1 multiplexers is shown in Fig. 5.17.

Table 5.14 The 2-input XNOR gate

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

Table 5.15 The XNOR entries to realize using multiplexers

a_in	y_out
0	\bar{b}_{in}
1	b_{in}

**Fig. 5.17** XNOR gate using 2:1 multiplexers

5.8 Boolean Functions and Implementation Using Mux

Now let us use the above understanding and let us try to implement the Boolean function using the suitable multiplexer.

1. XOR using 4:1 mux

To implement the XOR using single 4:1 mux let us use the truth table of XOR gate (Table 5.16).

Table 5.16 Truth table of 2-input XOR gate

A	B	Y_{out}
0	0	0
0	1	1
1	0	1
1	1	0

Now as we know that 4:1 mux has four inputs and 2 select lines, let us use A and B as select inputs and use the required logic levels at inputs to get the XOR function. We need to have $y_{\text{out}} = 0$ for $A = 0, B = 0$ so let us connect $T \times 0$ to 0. Similarly, to get $y_{\text{out}} = 0$ for $A = 1, B = 1$ let us connect $T \times 3$ to 0 (Fig. 5.18).

2. Let us find the logic function

Now let us consider the design shown in Fig. 5.19. Let us find the logic function.

From Fig. 5.19 the select lines are A and B. MSB that is $S1 = A$ and $S0 = B$. So let us document the entries in Table 5.17 as shown below.

From the truth table the output Y_{out} is equal to logic 0 for only $A = 1, B = 1$. So, the Boolean function is $\overline{A} + \overline{B}$ which is equal to $\overline{A} \cdot \overline{B}$.

Fig. 5.18 The XOR using 4:1 mux

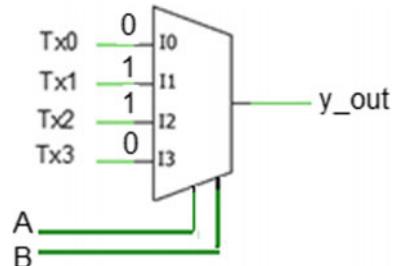


Fig. 5.19 The combinational logic

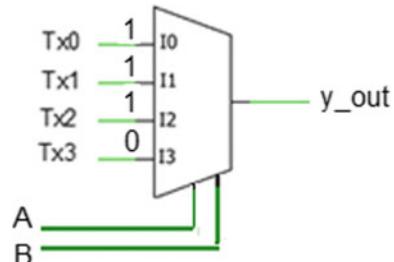


Table 5.17 The truth table from Fig. 5.13

$S1 = A$	$S0 = B$	Y_{out}
0	0	1
0	1	1
1	0	1
1	1	0

5.9 Mux as Universal Logic

Using the minimum number of 2:1 mux we can design any combinational function, and hence, multiplexers are treated as universal logic.

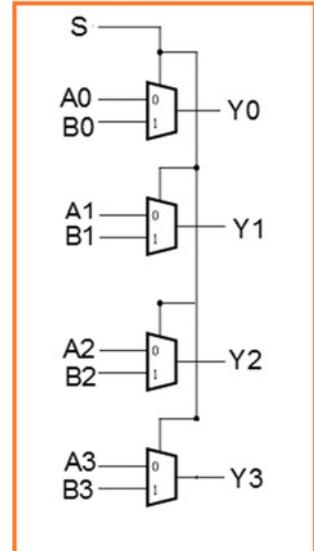
Consider the design requirement to pass 4-bit binary input A ($A_3 A_2 A_1 A_0$) for $S = 0$ and for $S = 1$ output should be binary input B ($B_3 B_2 B_1 B_0$). In such scenario we can use the chain of parallel multiplexers. Table 5.18 shows the equivalent output for the select input.

For each bit we need to have single 2:1 mux. Thus, we need to have four 2:1 multiplexers. The design using multiplexer is shown in Fig. 5.20.

Table 5.18 The table entries for selection logic

S	Y_3	Y_2	Y_1	Y_0
0	A_0	A_1	A_2	A_3
1	B_0	B_1	B_2	B_3

Fig. 5.20 The multiplexer as selection logic



5.10 Summary

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

1. The multiplexers are universal logic and used to implement any kind of combinational design.
2. The multiplexers are used in the designs such as bus, clock, and pin multiplexing.
3. The multiplexers are universal logic and used in many applications as selection network.
4. To implement the NOT gate single 2:1 mux is needed.
5. The 4:1 mux can be realized using the minimum 3, 2:1 mux.

Chapter 6

Decoders and Encoders



The various data select and control elements their understanding and applications are useful during the architecture design phase.

In the previous few chapters, we have discussed about the multiplexers and their use in the design. The chapter focuses on the demultiplexers, decoders, encoders, and priority encoders and their role in the design.

6.1 Demultiplexers and Use in the Design

Demultiplexers are reverse of the multiplexers and used to demultiplex the data lines. Consider the transmitter which has four channels and transmits the 4-bit information at a time. Due to parallel transmission the design uses more bandwidth and more power and many lines. To improve the design performance, the better strategy is to use the multiplexed channel at the transmitter side and demultiplex at receiver side to reconstruct the transmitted channels.

In such applications we need to have the multiplexers and demultiplexers. The strategy is shown in the following Fig. 6.1 (Table 6.1).

So instead of transmitting the four different channels we can multiplex the data and just transmit the single channel at a time using multiplexer with the select lines. The same select lines transmitted we can use to reconstruct the channel using demultiplexer at the receiver side. This improves the design power and bandwidth. Only the care should be taken by the design team to select the channel transmission frequency.

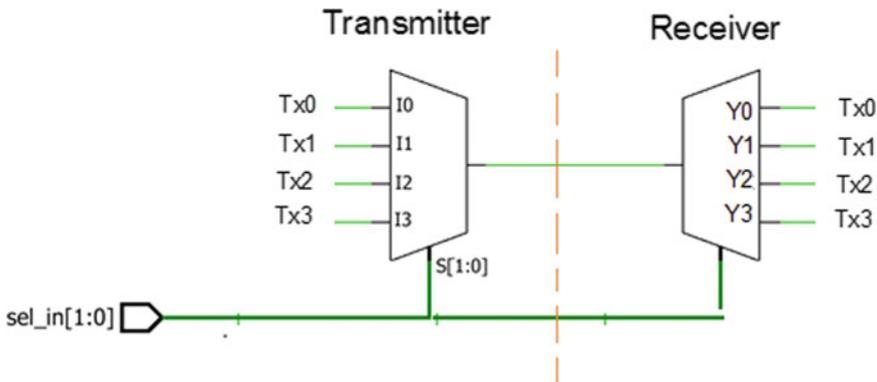


Fig. 6.1 Digital transmitter and receiver

Table 6.1 The channel reconstruction at receiver

sel_in [1]	sel_in [0]	Channel selected and transmitted
0	0	$T \times 0$
0	1	$T \times 1$
1	0	$T \times 2$
1	1	$T \times 3$

6.2 Decoder 2 to 4 Having Active High Output

In most of the interfacing applications we use the decoders to select one of the memory or IO devices. The goal is to enable the desired chip depending on the address and operation for the data transfer.

The 2:4 decoder which has active high enable input (EN) and active high outputs (only one line is active at a time during enable condition) is described in the table. It has select inputs S1 and S0, where S1 is MSB and S0 is LSB. Depending on the status of the select inputs during EN = 1 one of the output lines among the Y3 to Y0 is active high. When EN = 0 all output lines Y3, Y2, Y1, and Y0 are pulled down to logic 0.

How to design the decoding logic?

To design the decoder let us deduce the product term for every combination of the inputs and outputs shown in Table 6.2.

Table 6.2 Truth table of 2:4 decoder having active high output

Enable (EN)	S1	S0	Y3	Y2	Y1	Y0
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

For EN = 1, S1 = 0, S0 = 0 the product term is EN · $\overline{S1} \cdot \overline{S0}$.

For EN = 1, S1 = 0, S0 = 1 the product term is EN · $\overline{S1} \cdot S0$.

For EN = 1, S1 = 1, S0 = 0 the product term is EN · $S1 \cdot \overline{S0}$.

For EN = 1, S1 = 1, S0 = 1 the product term is EN · $S1 \cdot S0$.

For EN = 0 all the outputs are logic 0 as decoder is disabled. So, the design of 2:4 decoder has Boolean expressions for outputs

$$Y0 = EN \cdot \overline{S1} \cdot \overline{S0}$$

$$Y1 = EN \cdot \overline{S1} \cdot S0$$

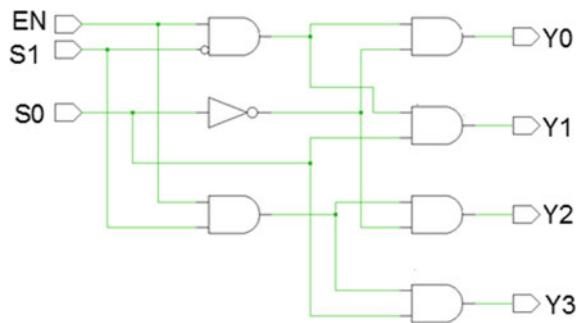
$$Y2 = EN \cdot S1 \cdot \overline{S0}$$

$$Y3 = EN \cdot S1 \cdot S0$$

The decoder design using minimum number of AND and NOT gates is shown in Fig. 6.2.

The timing waveform for the various combinational of EN, S1, and S0 is shown in Fig. 6.3. As shown one of the outputs of decoder is high during EN = 1 condition. For EN = 0 all decoder outputs are logic 0.

Fig. 6.2 The 2:4 decoder



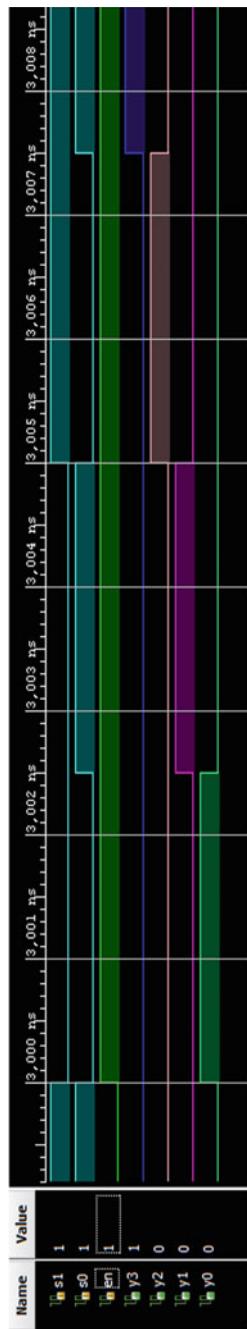


Fig. 6.3 The 2:4 decoder waveform

6.3 Decoder 2 to 4 Having Active Low Output

Now let us discuss the design of the 2:4 decoder having active high enable input (EN) and active low outputs. The truth table is shown in Table 6.3.

Let us design the decoding logic to have active low output

To design the decoder let us deduce the product term for every combination of the inputs and outputs shown in Table 6.3.

For EN = 1, S1 = 0, S0 = 0 the product term is $\overline{EN \cdot \overline{S1} \cdot \overline{S0}}$.

For EN = 1, S1 = 0, S0 = 1 the product term is $\overline{EN \cdot \overline{S1} \cdot S0}$.

For EN = 1, S1 = 1, S0 = 0 the product term is $\overline{EN \cdot S1 \cdot \overline{S0}}$.

For EN = 1, S1 = 1, S0 = 1 the product term is $\overline{EN \cdot S1 \cdot S0}$.

For EN = 0 all the outputs are logic 1 as decoder is disabled. So, the design of 2:4 decoder has Boolean expressions for outputs

$$Y0 = \overline{EN \cdot \overline{S1} \cdot \overline{S0}}$$

$$Y1 = \overline{EN \cdot \overline{S1} \cdot S0}$$

$$Y2 = EN \cdot S1 \cdot \overline{S0}$$

$$Y3 = \overline{EN \cdot S1 \cdot S0}$$

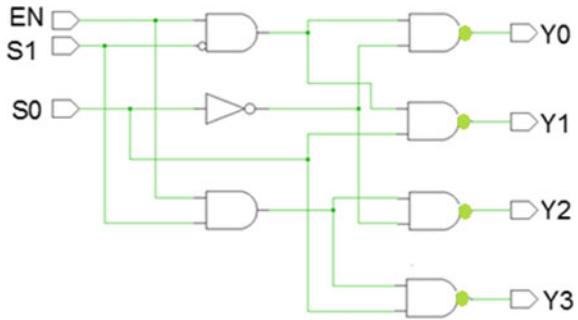
The decoder design using minimum number of AND, NOT, and NAND gates is shown in Fig. 6.4.

Using the above techniques, you can design any other decoder depending on the design requirements.

Table 6.3 The 2:4 decoder having active low outputs

Enable (EN)	S1	S0	Y3	Y2	Y1	Y0
1	0	0	1	1	1	0
1	0	1	1	1	0	1
1	1	0	1	0	1	1
1	1	1	0	1	1	1
0	X	X	1	1	1	1

Fig. 6.4 The 2:4 decoder having active low outputs



6.4 Design for the Given Specifications

Now let us design the decoding logic to have the active low outputs and active low enable input (EN).

How can we start? We can describe the above specifications to have the desired decoder (Table 6.4).

Let us design the decoding logic to have active low output and active low enable

To design the decoder let us deduce the product term for every combination of the inputs and outputs shown in Table 6.2.

For $EN = 0, S1 = 0, S0 = 0$ the product term is $\overline{\overline{EN} \cdot \overline{S1} \cdot \overline{S0}}$.

For $EN = 0, S1 = 0, S0 = 1$ the product term is $\overline{\overline{EN} \cdot \overline{S1} \cdot S0}$.

For $EN = 0, S1 = 1, S0 = 0$ the product term is $\overline{\overline{EN} \cdot S1 \cdot \overline{S0}}$.

For $EN = 0, S1 = 1, S0 = 1$ the product term is $\overline{\overline{EN} \cdot S1 \cdot S0}$.

For $EN = 1$ all the outputs are logic 1 as decoder is disabled. So, the design of 2:4 decoder has Boolean expressions for outputs

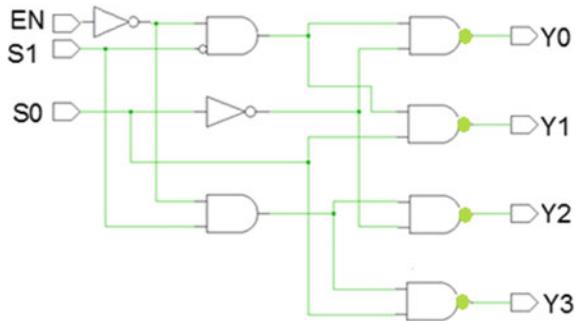
$$Y0 = \overline{\overline{EN} \cdot \overline{S1} \cdot \overline{S0}}$$

$$Y1 = \overline{\overline{EN} \cdot \overline{S1} \cdot S0}$$

Table 6.4 The 2:4 decoder having active low outputs and active low enable

Enable (EN)	S1	S0	Y3	Y2	Y1	Y0
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

Fig. 6.5 The 2:4 decoder having active low outputs and active low enable



$$Y2 = \overline{\overline{EN}} \cdot S1 \overline{S0}$$

$$Y3 = \overline{\text{EN}} \cdot S1 \cdot S0$$

The decoder design using minimum number of AND, NOT, and NAND gates is shown in Fig. 6.5.

Using the above techniques, you can design any other decoder depending on the design requirements.

6.5 Design of 3:8 Encoder Using 2:4 Decoders

Let us design now the 3:8 decoder having active high output and active high enable using minimum number of 2:4 decoders.

To start this let us document the entries in Table 6.5.

Now to design the 3:8 decoder we need two 2:4 decoders. Why? Because we need to have 8 outputs.

Table 6.5 The truth table of 3:8 decoder

Table 6.6 The truth table of 3:8 decoder using 2:4 decoder

enabl e (EN)	S 2	S 1	S 0	y 7	y 6	y 5	y 4	Y 3	Y 2	Y 1	Y 0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0
0	x	x	x	0	0	0	0	0	0	0	0

Decoder 2:4

Decoder 2:4

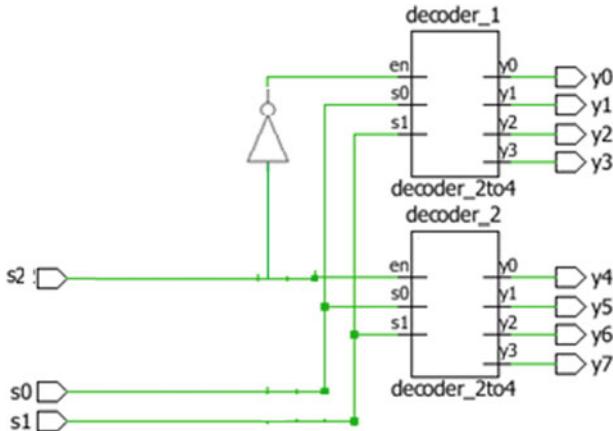
From Table 6.6 it is clear that the first 2:4 decoder is active for EN = 1 and S2 = 0 and generates outputs y3, y2, y1, and y0. The second 2:4 decoder is active for EN = 1 and S2 = 1 and generates outputs y7, y6, y5, and y4.

Both decoders use the select lines as S1 and S0 but the first decoder is enabled for S2 = 0 and second decoder is enabled for S2 = 1 (Table 6.7).

6.6 Encoders and Their Applications

As discussed in the previous section the decoders are used to generate one of the outputs as active at a time during enabled condition. The encoders are reverse of the decoder and used to encode the data inputs. For example, for $i_0 = 1$ we need $y_1 = 0$ and $y_0 = 0$ and for $i_3 = 1$ we need $y_1 = 1$ and $y_0 = 1$ then we can think of using 4:2 encoder.

The relationship between the inputs and outputs of encoder is given by $n = \log_2 m$ where m = number of inputs and n are number of outputs. For the $m = 4$ the output

Table 6.7 The design of 3:8 decoder using 2:4 decoders

lines are $n = \log_2 4 = 2$ that is y_1 and y_0 . Table 6.8: The truth table of 4:2 encoder describes the relationship between inputs i_3, i_2, i_1 , and i_0 and outputs y_1 and y_0 .

How to design the 4:2 encoder?

To design the encoder let us use the strategy of the maximum input conditions. As having four inputs, one of the inputs is active high at a time which is our assumption due to that we can consider other 12 conditions as x (pronounced as do not care). Now why? As we know that the number of conditions for the 4 inputs is $2^4 = 16$. But in the truth table of encoder only outputs are specified for the 1000, 0100, 0010, and 0001 so we need to specify outputs as x for remaining conditions.

So, to deduce the expression for y_1 and y_0 let us use the 4-variable K-map (Fig. 6.6).

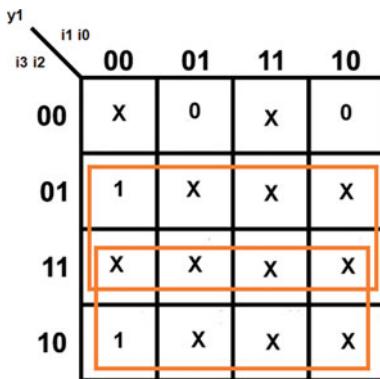
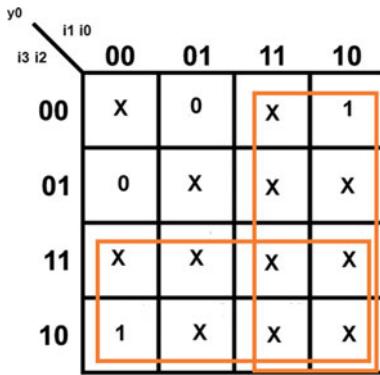
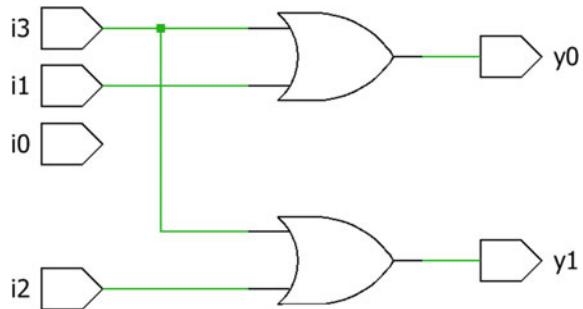
From the K-map entries we can get two terms as we have group of eight 1's $y_1 = i_3 + i_2$ that is OR of (i_3, i_2) (Fig. 6.7).

From the K-map entries we can get two terms as we have group of eight 1's $y_0 = i_3 + i_1$ that is OR of (i_3, i_1).

The data input i_0 is not connected, and hence, the design has the issues. The logic realized for the 4:2 encoder is shown in Fig. 6.8.

Table 6.8 The truth table of 4:2 encoder

i_3	i_2	i_1	i_0	y_1	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

Fig. 6.6 K-map for y_1 **Fig. 6.7** K-map for y_0 **Fig. 6.8** Encoder and the issue of dangling input

Practical issue in the 4:2 encoder designed are listed below:

- It is assumed that one of the inputs is logic 1, but practically more than one input can be logic 1.*

- b. *For all the inputs as logic 0, it is not specified what should be output. Output should be invalid.*

6.7 Practical Encoder Design

As discussed in the above section, we have not made any provision to implement the encoder if all inputs are logic 0. The encoder should generate invalid output as logic 1 when all inputs are logic 0. Table 6.9: Truth table of 4:2 practical encoder describes the 4:2 encoder which can be used in the practical system design.

So, for the invalid output logic we can have the NOR gate that is

$$\text{invalid_output} = \overline{i3} \cdot \overline{i2} \cdot \overline{i1} \cdot \overline{i0}$$

$$\text{invalid_output} = \overline{i3 + i2 + i1 + i0}$$

When all inputs are logic 0, an invalid_output is active high. Let us use the K-map to deduce the expression for $y1$ (Fig. 6.9).

Thus, $y1 = i3 + i2$ **that is OR of** ($i3, i2$). Now let us use K-map shown in Fig. 6.10 to deduce the expression of $y0$.

Thus, $y0 = i3 + i1$ **that is OR of** ($i3, i1$).

Table 6.9 Truth table of 4:2 practical encoder

i3	i2	i1	i0	y1	y0	Invalid_output
1	0	0	0	1	1	0
0	1	0	0	1	0	0
0	0	1	0	0	1	0
0	0	0	1	0	0	0
0	0	0	0	0	0	1

Fig. 6.9 K-map for y_1 of encoder

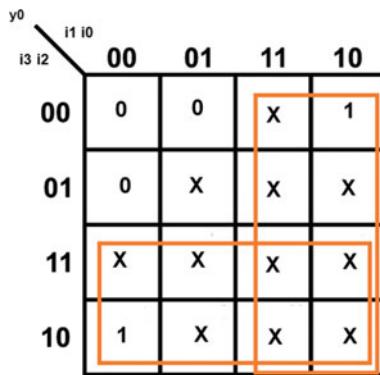
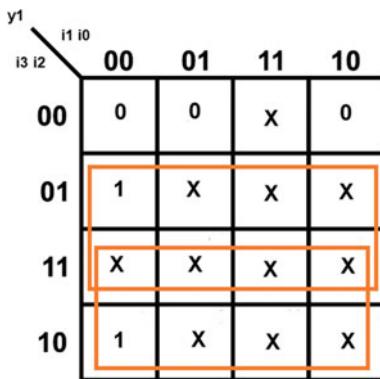


Fig. 6.10 K-map for y_0 of encoder

The 4:2 encoder having inputs i_3, i_2, i_1 , and i_0 and outputs as y_1 and y_0 and the invalid_output flag logic is shown in Fig. 6.11.

Still the encoder design shown in Fig. 6.11 has issues, as we have assumed that only one input is 1 at a time, and hence, we need to design the priority encoder design.

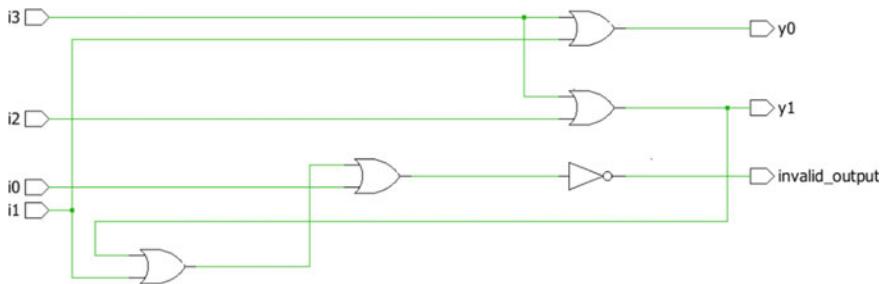


Fig. 6.11 Encoder with the invalid output detection

During the exercises let us use the understanding of the decoders and encoders and let us try to implement the designs which can be used in few system design applications.

6.8 Priority Encoders

Let us design the 4:2 priority encoder using minimum number of logic gates. Consider i_3 has highest priority and i_0 has lowest priority.

As it is specified that i_3 has highest priority and i_0 has lowest priority. Let us create Table 6.10 to indicate the relation between the inputs and outputs. Let us use the invalid_output to indicate the output invalid status when all the inputs i_3 to i_0 are logic 0.

Now by using the entries specified let us deduce the expression for the y_1 and y_0 using 4-variable K-map.

From the K-map (Fig. 6.12) $y_1 = i_3 + i_2$ **that is OR of** (i_3, i_2).

Thus, for the K-map entries shown in Fig. 6.13 $y_0 = i_3 + \overline{i_2} \cdot i_1$.

For the invalid output logic, we can have the NOR gate that is

$$\text{invalid_output} = \overline{i_3} \cdot \overline{i_2} \cdot \overline{i_1} \cdot \overline{i_0} = \overline{i_3 + i_2 + i_1 + i_0}$$

When all inputs are logic 0, an invalid_output is active high. The priority encoder design is shown in the Fig. 6.14.

Table 6.10 The 4:2 priority encoder truth table

i_3	i_2	i_1	i_0	y_1	y_0	invalid_output
1	X	X	X	1	1	0
0	1	X	X	1	0	0
0	0	1	X	0	1	0
0	0	0	1	0	0	0
0	0	0	0	0	0	1

Fig. 6.12 The K-map for the y_1 of priority encoder

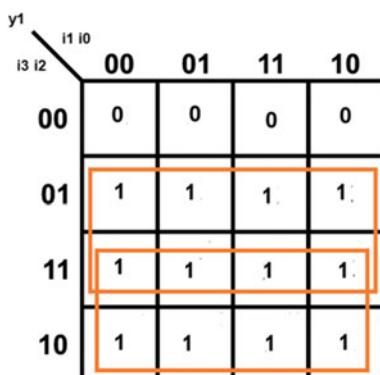


Fig. 6.13 The K-map for the y_0 of the priority encoder

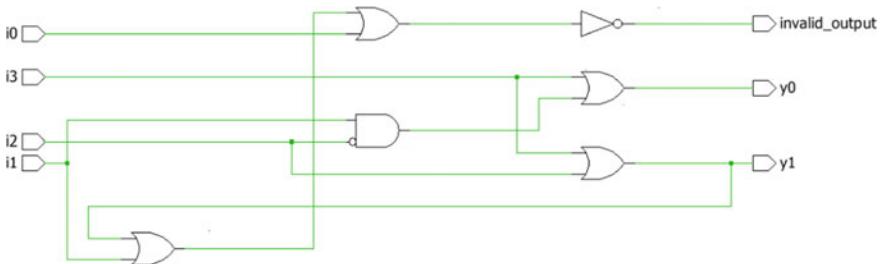
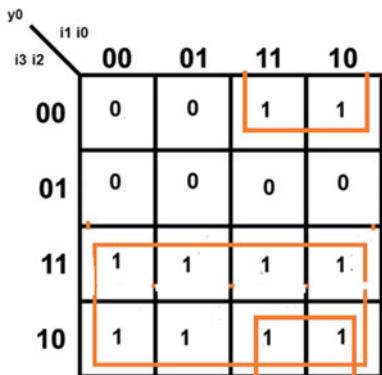


Fig. 6.14 The 4:2 priority encoder

6.9 Practical Design Scenario

Consider the processor which has two level sensitive interrupts. We need to sample the interrupts and schedule the interrupt service routine depending on the priority. INT0 has highest priority and INT1 has lowest priority. So how we can design?

So let us try to design this logic. What we need to do is that we need to have priority interrupt control logic. To design this let us document the entries in the table. INT0 INT1 y_{out} (Fig. 6.15; Table 6.11).

From the K-map it is clear that $y_{out} = INT0$. So following are the expressions

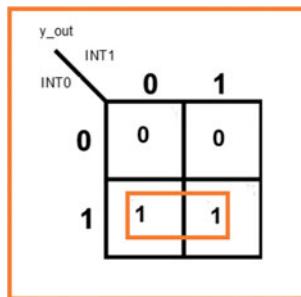
$$\text{invalid_output} = \overline{\text{INT0}} \cdot \overline{\text{INT1}} = \overline{\text{INT0}} + \overline{\text{INT1}}$$

Thus $y_{out} = 1$ and $\text{invalid_output} = 0$ indicates process the INT0 interrupt.

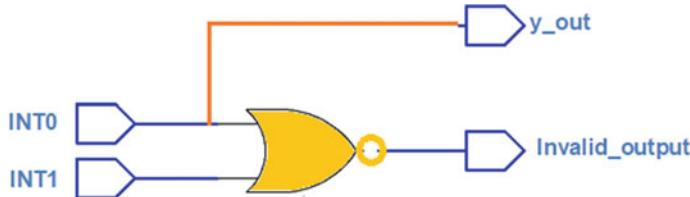
If $y_{out} = 0$ and $\text{invalid_output} = 0$ it indicates that process the INT1 interrupt.

If $\text{invalid_output} = 1$ it indicates none of the interrupt has arrived. The logic uses the single NOR gate having two inputs INT0 and INT1 to generate the invalid_output and $y_{out} = INT0$ (Fig. 6.16).

In this chapter we have discussed about the demultiplexers, decoders, encoders, and priority logic. The next chapter focuses on the combinational design and the various design scenarios.

**Fig. 6.15** K-map to deduce output y_{out} **Table 6.11** The truth table of 2:1 priority encoder

INT0	INT1	y_{out}	invalid_output
1	X	1	0
0	1	0	0
0	0	0	1

**Fig. 6.16** The priority encoder 2:1

6.10 Summary

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

1. The demultiplexers are reverse of the multiplexers and used to demultiplex the data at the receiver end.
2. Encoders are used to encode the digital data.
3. The practical designs should have the better control and data path optimization.
4. The priority encoders should have the valid data indicators.
5. The priority encoders are used to detect the highest priority inputs and to generate the outputs.

Chapter 7

Combinational Design Scenarios



The various combinational design and optimization techniques are useful during the design phase.

In the previous few chapters, we have discussed about the combinational design using various combinational elements. In this chapter let us discuss about few of the combinational design scenarios and how to design the efficient glue logic. The chapter is useful to understand the applications of the multiplexers and design using the combinational elements.

7.1 Mux-Based Designs and Optimization

We have discussed in the previous chapter about the multiplexers as universal logic. They are called as universal logic because by using the multiplexers we can implement any logic function.

Now consider the design requirement to have output as A for opcode = 0 and output as complement of A for opcode = 1. In such scenario we can think of using the multiplexers. The operations are listed in Table 7.1.

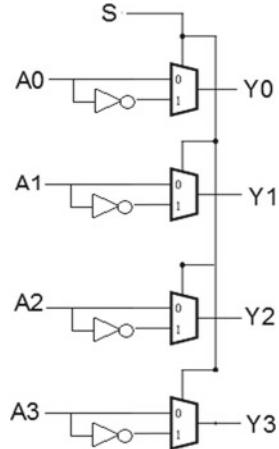
Now let us design the logic! What we need is the 2:1 mux and Not gate. If size of A is 8 bits then we need eight, 2:1 mux and 8 NOT gates. Figure 7.1 is the design using mux for 4-bit input A .

For $S = 0$ the output $Y = A$, and for the $S = 1$ the output Y is equal to complement of A .

Table 7.1 Operational table of mux-based design

opcode	Output
0	A
1	$\sim A$

Fig. 7.1 Mux-based design-1



7.2 Right and Left Shift Using Multiplexers

We can think of using the multiplexer chain to perform the right and left shift. As multiplexer is a combinational logic we can get the combinational shifter which we can think of using as barrel shifter.

Now consider the design requirement to have output as right shift of A for opcode = 0 and output as left shift of A for opcode = 1. In such scenario we can think of using the multiplexers. The operations are listed in Table 7.2.

Now let us design the logic! What we need is the chain of 2:1 mux. If size of A is 8 bits then we need eight, 2:1 mux.

Let us modify the truth table to have more detail understanding of the requirement (Table 7.3).

Figure 7.2 is the design using mux for 4-bit input A .

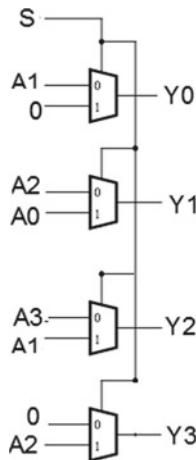
For $S = 0$ the output Y is right shift of A , and for the $S = 1$ the output Y is equal to left shift of A .

Table 7.2 Operational table of shifter

opcode	Output
0	Right shift A by 1 bit
1	Left shift A by 1 bit

Table 7.3 Entries to have right and left shift logic

opcode	Y_3	Y_2	Y_1	Y_0
0	0	A_3	A_2	A_1
1	A_2	A_1	A_0	0

**Fig. 7.2** Mux-based design-2

7.3 Design of 8:1 Mux Using 4:1 Mux

Now let us consider we need to have the 8:1 mux using only 4:1 mux. To design the 8:1 mux let us document the entries in Table 7.4.

Now to design the 8:1 mux using minimum number of 4:1 mux divide the table entries into two groups of four each and create the input logic.

From Table 7.5 it is clear that we need to have two 4:1 mux and single 2:1 mux to implement the 8:1 mux.

Table 7.4 8:1 mux truth table

S_2	S_1	S_0	Y
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3
1	0	0	I4
1	0	1	I5
1	1	0	I6
1	1	1	I7

Table 7.5 8:1 mux using 4:1 mux

S2	S1	S0	Y
0 0 0 0	0 0 1 1	0 1 0 1	I0 I1 I2 I3
1 1 1 1	0 0 1 1	0 1 0 1	I4 I5 I6 I7
			4:1 mux
			4:1 mux
			2:1 mux

The logic diagram is shown in Fig. 7.3. As shown the output mux uses the select input S2 and input multiplexers use select inputs S1 S0.

7.4 Design of 8:1 Mux Using 2:1 Mux

Now let us consider we need to have the 8:1 mux using only 2:1 mux. To design the 8:1 mux let us document the entries in Table 7.6.

Now to design the 8:1 mux using minimum number of 4:1 mux divide the table entries into four groups of two each and create the input logic.

From Table 7.7 it is clear that we need to have seven 4:1 mux to implement the 8:1 mux.

Fig. 7.3 Design of 8:1 mux using 4:1 multiplexers

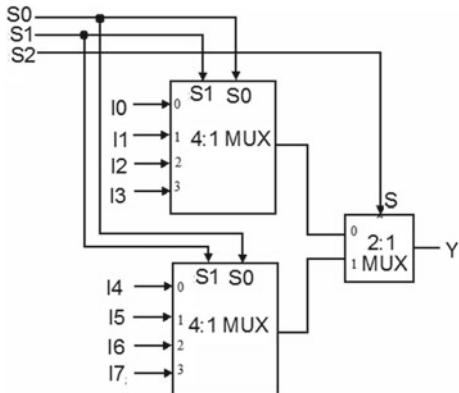


Table 7.6 8:1 mux truth table

S2	S1	S0	Y
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3
1	0	0	I4
1	0	1	I5
1	1	0	I6
1	1	1	I7

The logic diagram is shown in Fig. 7.4. As shown the output mux uses the select input S2 and input multiplexers uses select inputs S1 S0.

7.5 Boolean Expression from the Logic

Consider the following design and let us find the Boolean expression (Fig. 7.5).

Step 1: Let us document the Y_1 output of input 4:1 mux (Table 7.8)

Thus, output Y_1 is equal to

$$Y_1 = A \oplus B$$

$$Y_1 = A \cdot \overline{B} + \overline{A} \cdot B$$

Step 2: Let us document the Y output from output 4:1 mux (Table 7.9)

Thus, output Y is equal to

$$Y = Y_1 \oplus C$$

$$Y = A \oplus B \oplus C$$

Table 7.7 8:1 mux using 2:1 mux

2:1 mux			
S2	S1	S0	Y
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3
1	0	0	I4
1	0	1	I5
1	1	0	I6
1	1	1	I7

2:1 mux 2:1 mux 2:1 mux 2:1 mux

7.6 Boolean Expression for Mux-Based Design

Consider the following design and let us find the Boolean expression (Fig. 7.6).

Step 1: Let us document the Y output of 4:1 mux (Table 7.10).

Step 2: Let us deduce the output expression

Thus, output Y is equal to

$$Y1 = A \oplus B$$

$$Y1 = A \cdot \overline{B} + \overline{A} \cdot B$$

Fig. 7.4 Design of 8:1 mux using 2:1 mux

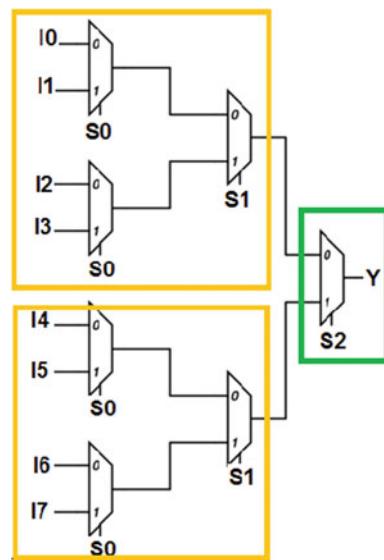


Fig. 7.5 Mux -based design-3



Table 7.8 Entries for output Y_1

A	B	Y_1
0	0	0
0	1	1
1	0	1
1	1	0

Table 7.9 Entries for output Y

Y_1	C	$Y_2 = Y$
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 7.6 Mux-based design-4

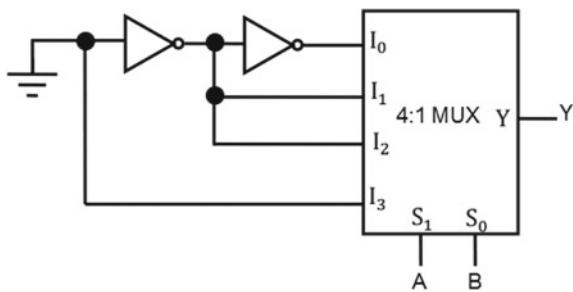
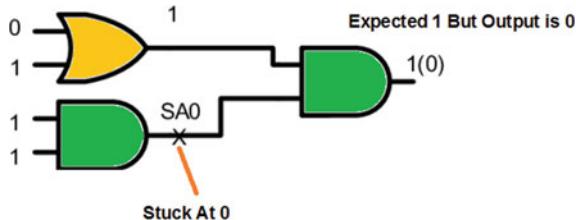


Table 7.10 Entries for output Y of mux

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 7.7 Single stuck at faults



7.7 Stuck at Faults

Consider the following design and let us find the Boolean expression (Fig. 7.7).

The design has the single stuck at fault and indicated as SA0. The stuck at fault is an issue in the design may be the net is shorted to ground.

Due to net stuck at 0 output of AND gate is always at logic 0 irrespective of the OR gate output 1.

7.8 Design Using Decoders

Implement the 2-input XOR and 2-input XNOR using the decoder having active high outputs and active high enable input. Use minimum logic gates.

Let us use the understanding of the decoder. During $\text{en} = 1$ the decoder has only one output as active high. When $\text{en} = 0$ the decoder all outputs are logic 0 and decoder is disabled.

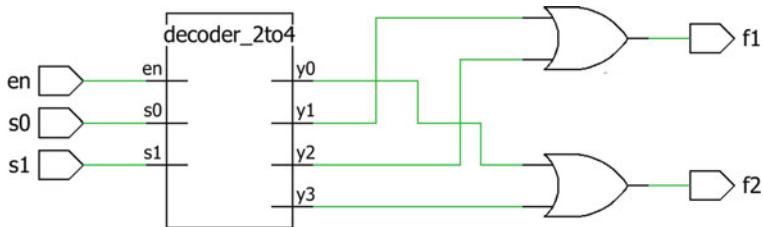


Fig. 7.8 XOR and XNOR logic function realization using decoder and logic gates

Table 7.11 2-input XOR, XNOR logic using decoder

Enable	s_1	s_0	f_1	f_2
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1
0	X	X	0	0

Now to realize the logic gate 2-input XOR let us have the Boolean function $f_1(s_1, s_0) = \sum m(1, 2)$ use the OR gate at output as function is SOP. The inputs of OR gate are y_1, y_2 . If one of the outputs y_1 or y_2 is logic 1 then f_1 is logic 1.

To realize the logic gate 2-input XNOR let us have the Boolean function $f_2(s_1, s_0) = \sum m(0, 3)$ use the OR gate at output as function is SOP. The inputs of OR gate are y_0, y_3 . If one of the outputs y_0 or y_3 is logic 1 then f_2 is logic 1.

The logic realized using the decoder and the OR gates is shown in Fig. 7.8 (Table 7.11).

7.9 Design Using Decoder and NAND Gates

Implement the 2-input XOR and 2-input XNOR using the decoder having active low outputs and active high enable input. Use minimum logic gates.

Let us use the understanding of the decoder. During $en = 1$ the decoder has only one output as active low. When $en = 0$ the decoder all outputs are logic 1 and decoder is disabled.

Now to realize the logic gate 2-input XOR let us have the Boolean function $f_1(s_1, s_0) = \sum m(1, 2)$ use the NAND gate at output as function is SOP. The inputs of NAND gate are y_1, y_2 . If one of the outputs y_1 or y_2 is logic 0 then f_1 is logic 1.

To realize the logic gate 2-input XNOR let us have the Boolean function $f_2(s_1, s_0) = \sum m(0, 3)$ use the NAND gate at output as function is SOP. The inputs of NAND gate are y_0, y_3 . If one of the outputs y_0 or y_3 is logic 0 then f_2 is logic 1.

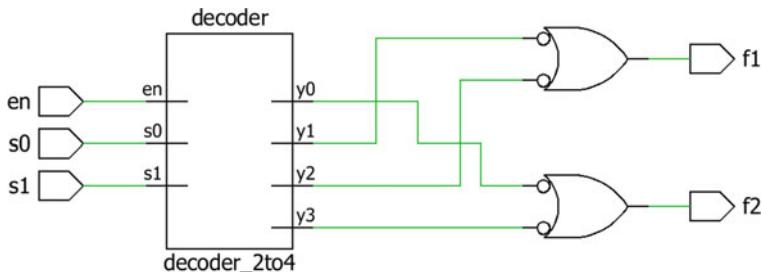


Fig. 7.9 Logic function realization using decoder and bubbled OR gates

Table 7.12 Logic function realization using decoder having active low outputs

Enable	s1	s0	f1	f2
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1
0	X	X	0	0

The logic realized using the decoder and the NAND gates (Bubbled OR) is shown in Fig. 7.9 (Table 7.12).

In this chapter we have discussed about the design scenarios and mux-based design. The next chapter is useful to understand the sequential design concepts.

7.10 Summary

The following are few of the important points to conclude this chapter.

1. The multiplexers are universal logic and used to implement any logic function.
2. The barrel shifters are combinational shifters and used in the DSP-based designs.
3. The combinational shifters are designed using chain of multiplexers.
4. To implement the 8:1 mux we need to have seven 2:1 mux.
5. The stuck at faults is due to net connected directly to VCC or GND.
6. Multiple SOP functions can be implemented using single decoder with additional logic gates.

Chapter 8

Synchronous Sequential Design



The understanding of the synchronous sequential circuits plays important role during the architecture design phase.

In the previous chapters we have discussed about the combinational design. In the sequential design an output is function of the present inputs and past outputs. The main important sequential design techniques are discussed in this chapter. The chapter focuses on the synchronous counters, shift registers, and design using D flip-flops.

We have two types of sequential designs, and they are

- **Synchronous design:** Clock is derived from common clock source.
- **Asynchronous design:** Clock is derived from different clock sources.

This chapter focuses on the synchronous sequential circuits and the various design techniques.

8.1 Sequential Design Elements

To design the synchronous sequential circuits or asynchronous circuits, we can use the elements as flip-flops. The important flip-flops which we can use are

1. SR flip-flop
2. JK flip-flop
3. D flip-flop
4. Toggle or T flip-flop.

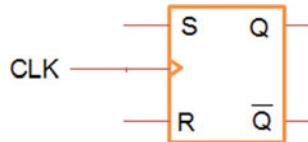
This section is useful to understand the various flip-flops used to design the sequential circuits.

1. **SR Flip-Flop:** The SR flip-flop has set and reset inputs and generates an output Q on the active edge of the clock. Table 8.1 describes the relationship between the inputs and outputs on the rising edge of the clock.

Table 8.1 Truth table of SR flip-flop

S	R	Q	\bar{Q}
0	0	No change	No change
0	1	0	1
1	0	1	0
1	1	Indeterminate	Indeterminate

Fig. 8.1 SR flip-flop symbolic representation



As described in the truth Table 8.1 the flip-flop output sets to 1 for $S = 1, R = 0$. The flip-flop output resets to logic 0 for the $S = 0$ and $R = 1$. The flip-flop output is equal to the previous output for $S = 0$ and $R = 0$. The main issue for the SR flip-flop is the indeterminate output for the $S = 1$ and $R = 1$. The output of flip-flop change state on the rising edge of the clock. The flip-flop symbolic representation is shown in Fig. 8.1.

In the VLSI context, as we need to have the design with minimum area the SR flip-flops are not recommended to design the sequential circuits.

The issue while using the SR flip-flop is due to next state logic needed to control the S and R inputs. Many inputs means more area to design the next state logic (Fig. 8.2).

2. **JK Flip-Flop:** The JK flip-flop has J and K inputs and generates an output Q on the active edge of the clock. Table 8.2 describes the relationship between the inputs and outputs on the rising edge of the clock. JK flip-flop is first integrated circuit in the world which was demonstrated by Jack Kilby during year 1958.

As described in the truth Table 8.2 the flip-flop output sets to 1 for $J = 1, K = 0$. The flip-flop output resets to logic 0 for the $J = 0$ and $K = 1$. The flip-flop output is equal to the previous output for $J = 0$ and $K = 0$. The advantage of the JK flip-flop

Fig. 8.2 Next state logic at S and R input

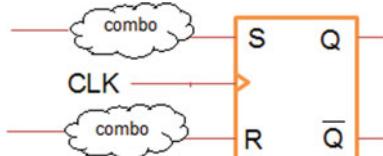
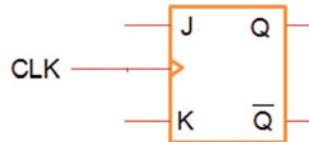


Table 8.2 Truth table of JK flip-flop

J	K	Q	\bar{Q}
0	0	No change	No change
0	1	0	1
1	0	1	0
1	1	Toggle	Toggle

Fig. 8.3 JK flip-flop



over SR flip-flop is it eliminates the indeterminate output for the $J = 1$ and $K = 1$. The output of flip-flop toggles for the $J = 1$ and $K = 1$. The flip-flop changes state on the rising edge of the clock. The flip-flop symbolic representation is shown in Fig. 8.3.

In the VLSI context, as we need to have the design with minimum area the JK flip-flops are not recommended to design the sequential circuits.

The issue while using the JK flip-flop is due to next state logic needed to control the J and K inputs. Many inputs means more area to design the next state logic (Fig. 8.4).

3. **D Flip-Flop:** The D flip-flop has data in input D and generates an output Q on the active edge of the clock. Table 8.3 describes the relationship between the input and outputs on the rising edge of the clock.

Fig. 8.4 JK flip-flop having next state logic at J and K inputs

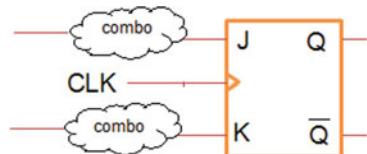
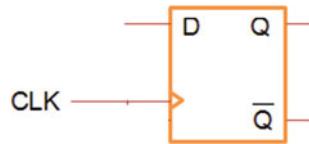


Table 8.3 Truth table of D flip-flop

D	Q	\bar{Q}
0	0	1
1	1	0

Fig. 8.5 D flip-flop

As described in the truth Table 8.3 the flip-flop output sets to 1 for $D = 1$. The flip-flop output resets to logic 0 for the $D = 0$. The flip-flop output is equal to the previous output for the inactive edge of the clock that is negative edge or level for rising edge sensitive D flip-flop. The advantage of the D flip-flop over JK flip-flop and SR flip-flop is it needs the single next state logic circuit at D input. The flip-flop symbolic representation is shown in Fig. 8.5.

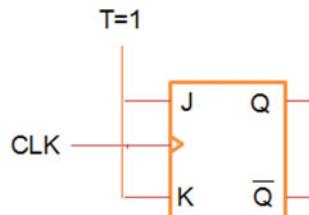
In the VLSI context, we always recommended to use D flip-flop to design the sequential circuits.

The main advantage of D flip-flop is the minimum combinational logic in the data path to get the next state. Designers need to control single D input to get the desired output.

4. Toggle or T flip-flop: The toggle flip-flop toggles on the active edge of the clock and can be designed using the JK flip-flop or D flip-flop.

Using JK flip-flop we can tie J and K inputs to logic 1 to get the toggle output (Fig. 8.6).

The truth table of toggle flip-flop designed using JK is shown in Table 8.4.

Fig. 8.6 Toggle flip-flop using JK flip-flop**Table 8.4** Truth table of T flip-flop

T	J	K	Q	\bar{Q}
0	0	0	No change	No change
1	1	1	Toggle	Toggle

Fig. 8.7 Toggle flip-flop using D flip-flop

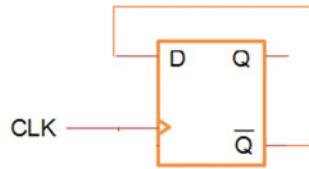
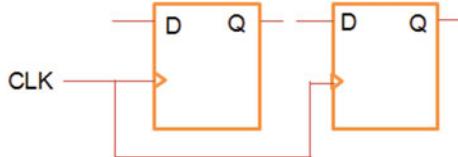


Fig. 8.8 Synchronous design



As described in Table 8.4 for $T = 0$ the flip-flop output maintains the previous output that is no change in the output Q and complement of Q . For $T = 1$ the flip-flop toggles on the rising edge of the clock.

Using D flip-flop, we can design the toggle flip-flop by feeding back the complement of Q to D input.

As shown in Fig. 8.7, let us assume the output Q of flip-flop is 0 and complement of Q is $Q = 1$. As $D = 1$ on the rising edge of the clock the flip-flop samples logic 1. As output $Q = 1$, the complement of Q is 0 and fed back to D . On the next rising edge of the clock 0 is sampled. Thus, output Q of flip-flop toggles on every rising edge of the clock.

Throughout this book we will use the D flip-flops to design the synchronous and asynchronous sequential circuits.

8.2 Synchronous Design

In the synchronous design the clock is common to all the flip-flops. That is clock is derived from the common clock source PLL. PLL is phase locked loop and used to generate the clock. Examples are counters, shift registers where all the elements are using the common clock. The synchronous design having common clock source is shown in Fig. 8.8.

8.3 Why to Use Synchronous Design?

As discussed earlier in the asynchronous design the clock of the MSB flip-flop is derived from the output of the previous stage. Depending on the operation requirement the clock of the flip-flop is derived and the LSB flip-flop receives the clock as PLL output.

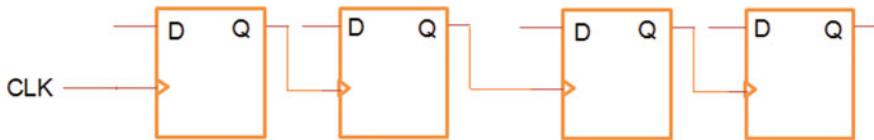


Fig. 8.9 Asynchronous design having D flip-flop stages

Now let us try to understand what is the issue in the asynchronous design? In the asynchronous design the overall delay to generate the output is due to propagation of the clock. For the n stages the delay to get the output from the MSB flip-flop is $n * tpff$. Where $tpff$ is flip-flop propagation delay or clock to q delay. The delays and advanced concepts like timing parameters of flip-flop are discussed in the next few chapters.

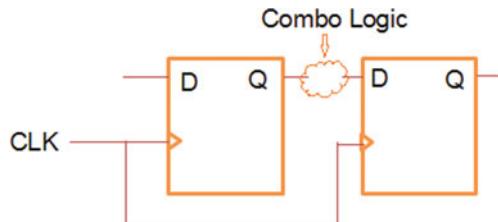
For the asynchronous design shown in Fig. 8.9. The number of stages $n = 4$ and hence the maximum propagation delay of the design is $4 * tpff$. If $tpff$ is 1 ns (pronounced as 1 ns) then the delay to get the output from MSB of flip-flop is 4 ns.

As discussed earlier the asynchronous designs have the more delay and they are slower as compared to the synchronous design. So, in most of the applications and in the system design we need to have the high speed hence the asynchronous design use is ruled out. Even asynchronous clocking has the issues and hence treated as poor clocking scheme.

Figure 8.10 shows the synchronous design where clock of both the flip-flops is derived from the common clock source. In between the flip-flops the combinational logic is used, and most of the time we call this as reg-to-reg path. Here the maximum frequency of the design is dependent on the timing parameters of the flip-flops that is setup time (tsu), hold time(th), clock to q delay ($tctoq$ or $tpff$), and the combinational logic delay.

The maximum frequency calculation and the timing parameters of the flip-flops are discussed in the subsequent chapter. The following section discusses the design techniques to implement the sequential design.

Fig. 8.10 Synchronous design (reg-to-reg path)



8.4 Asynchronous Design

In the asynchronous design the clock of the various sequential elements is driven by the different clock source. As shown in Fig. 8.11 the LSB flip-flop receives the clock from the PLL and the MSB flip-flop receives the clock as one of the output of the LSB flip-flop.

8.4.1 D Flip-Flop and Use in the Design

As D flip-flop has the single input, the D flip-flops are popular to design the sequential logic. It is easy to control the single D input as compared to the JK or SR flip-flops two inputs. Due to the less logic in the data path the area requirement for the sequential design using the D flip-flops is lesser as compared to SR or JK flip-flop.

What we need to understand about the D flip-flop?

For the sequential design using the D flip-flop we need to have the good understanding about the excitation inputs and excitation table. So let us try to discuss!

Consider the present state of the D flip-flop is Q , and the next state of the D flip-flop is Q^+ (Table 8.5). The excitation table gives information about what should be the data input of the D flip-flop to get the next state output on the active edge of the clock. The excitation table of the D flip-flop is documented below.

As shown in Table 8.6 the D input of the flip-flop is equal to the next state Q^+ . That means, if the present state $Q = 0$ and the next state $Q^+ = 1$ then to get the next state as output of the flip-flop the data input of flip-flop $D = 1$. That is $D = Q^+$. During the sequential circuit designs and the FSM designs we use the excitation table to deduce the combinational logic required at the data input of D flip-flop.

Fig. 8.11 Asynchronous design

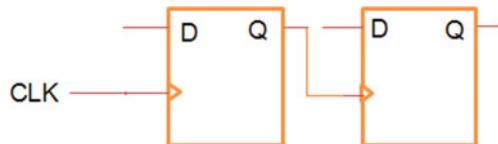


Table 8.5 State table of D flip-flop

Present state (Q)	Next state (Q^+)
0	0
0	1
1	0
1	1

Table 8.6 Excitation input of the D flip-flop

Present state (Q)	Next state (Q^+)	Excitation input (D)
0	0	0
0	1	1
1	0	0
1	1	1

Now let us consider the design scenario

Consider the design requirement is to get the toggle output using the single D flip-flop. What we need to do is that let us document the entries in the excitation table and let us deduce the logic.

As shown in Table 8.7, the excitation input $D = Q^+$ which is the complement of the present state Q . So, $D = \overline{Q}$. So, the toggle flip-flop is designed by using the single D flip-flop and NOT gate. If the flip-flop uses the asynchronous active low reset, then in the reset path the clear input of the flip-flop is controlled by NOT gate.

The design is shown in Fig. 8.12.

Table 8.7 Excitation table of the D flip-flop

Present State (Q)	Next State (Q^+)	Excitation Input (D)
0	1	1
1	0	0

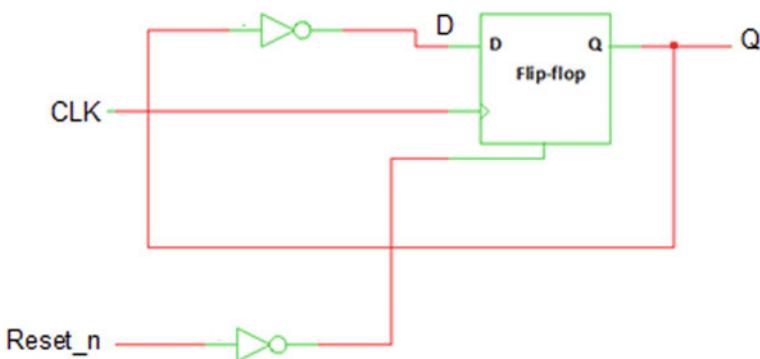


Fig. 8.12 Toggle D flip-flop

8.5 Design of the Synchronous Counters

Now let us try to use the design foundation discussed in the previous sections to design the synchronous modulo-8 or MOD-8 binary up-counter. As name itself indicates the counter has the 8 states. $s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7$ and for the binary up-counter design using the following steps.

1. Find the number of states to get the counter

Number of states = 8.

2. Find number of flip-flops

Number of flip-flops = $n = \log_2 8 = 3$. We will use the positive edge sensitive D flip-flops.

3. Reset strategy

Let us use active low asynchronous reset input $\text{reset_}n$. For $\text{reset_}n = 0$ the counter holds the previous output. For the $\text{reset_}n = 1$ the counter output increments on the rising edge of the clock.

4. Let us document the entries in the state table

The state table of the MOD-8 synchronous binary up-counter is shown in Table 8.8.

5. Let us document the entries in the excitation table

The excitation table consists of the information about the present state, next state, and excitation input (Table 8.9).

6. Let us deduce the logic at the input of the $D\ 2, D\ 1, D\ 0$

Let us use the present state Q_2, Q_1, Q_0 as the inputs to get the logic at the D_2, D_1, D_0 . Let us use the 3-varibale K-map (Figs. 8.13, 8.14, and 8.15).

$$D_2 = \overline{Q_2}.Q_1.Q_0 + \overline{Q_1}.Q_2 + \overline{Q_0}.Q_2$$

Table 8.8 State table of the 3-bit binary up-counter

Present state ($Q_2\ Q_1\ Q_0$)	Next state ($Q_2^+\ Q_1^+\ Q_0^+$)
000	001
001	010
010	011
011	100
100	101
101	110
110	111
111	000

Table 8.9 Excitation table of the MOD-8 counter

Present state ($Q_2 Q_1 Q_0$)	Next state ($Q_2^+ Q_1^+ Q_0^+$)	Excitation inputs ($D_2 D_1 D_0$)
000	001	001
001	010	010
010	011	011
011	100	100
100	101	101
101	110	110
110	111	111
111	000	000

Fig. 8.13 K-map for D_2

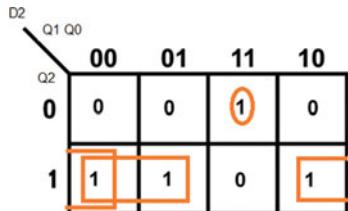


Fig. 8.14 K-map to get the logic at D_1

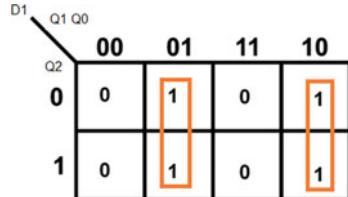
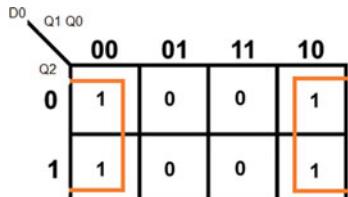


Fig. 8.15 K-map to get the logic at D_0



$$D_2 = \overline{Q_2} \cdot Q_1 \cdot Q_0 + Q_2 (\overline{Q_1} + \overline{Q_0})$$

$$D_2 = \overline{Q_2} \cdot Q_1 \cdot Q_0 + Q_2 (Q_1 \cdot Q_0)$$

$$D_2 = Q_2 \oplus (Q_1 \cdot Q_0)$$

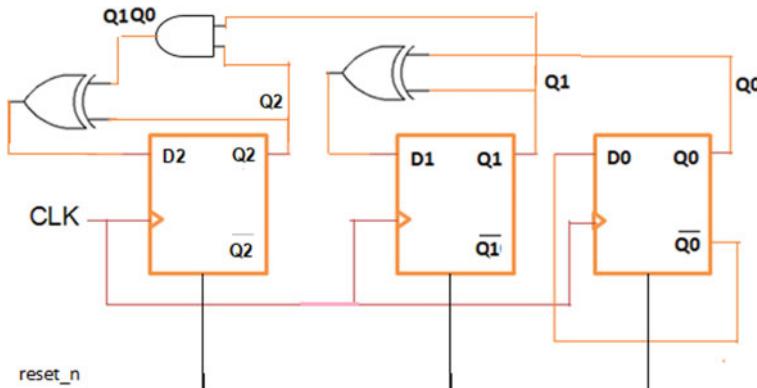


Fig. 8.16 3-bit synchronous binary up-counter

$$D1 = \overline{Q1}.Q0 + \overline{Q0}.Q1$$

$$D1 = Q1 \oplus Q0$$

$$D0 = \overline{Q0}$$

7. Let us sketch the logic

As discussed in the previous steps we need to have two flip-flops and combinational logic to implement the MOD-8 synchronous binary up-counter(Fig. 8.16).

8.6 Design of the Synchronous Down-Counters

Now let us try to use the design foundation discussed in the previous sections to design the synchronous modulo-8 or MOD-8 binary down-counter. As name itself indicates the counter has the 4 states. $s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7$ and for the binary down-counter design using the following steps.

1. Find the number of states to get the counter

Number of states = 8.

2. Find number of flip-flops

Number of flip-flops = $n = \log_2 8 = 3$. We will use the positive edge sensitive D flip-flops.

Table 8.10 State table of the MOD-8 binary up-counter

Present state ($Q_2\ Q_1\ Q_0$)	Next state ($Q_2^+\ Q_1^+\ Q_0^+$)
111	110
110	101
101	100
100	011
011	010
010	001
001	000
000	111

3. Reset strategy

Let us use active low asynchronous reset input reset__n . For $\text{reset_}_n = 0$ the counter holds the previous output. For the $\text{reset_}_n = 1$ the counter output decrements on the rising edge of the clock.

4. Let us document the entries in the state table

The state table of the MOD-8 synchronous binary down-counter is shown in Table 8.10.

5. Let us document the entries in the excitation table

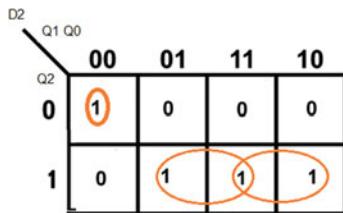
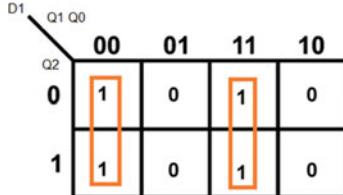
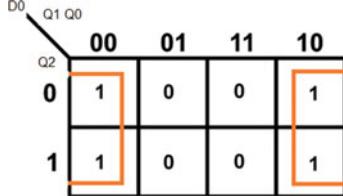
The excitation table consists of the information about the present state, next state, and excitation input (Table 8.11).

6. Let us deduce the logic at the input of the $D\ 2, D\ 1, D\ 0$.

Let us use the present state Q_2, Q_1, Q_0 as the inputs to get the logic at the D_2, D_1, D_0 . Let us use the 3-varibale K-map Figs. 8.17, 8.18 and 8.19.

Table 8.11 Excitation table of the MOD-8 down-counter

Present state ($Q_2\ Q_1\ Q_0$)	Next state ($Q_2^+\ Q_1^+\ Q_0^+$)	Excitation inputs ($D_2\ D_1\ D_0$)
111	110	110
110	101	101
101	100	100
100	011	011
011	010	010
010	001	001
001	000	000
000	111	111

Fig. 8.17 K-map for $D2$ **Fig. 8.18** K-map to get the logic at $D1$ **Fig. 8.19** K-map to get the logic at $D0$ 

$$D2 = \overline{Q2} \cdot \overline{Q1} \cdot \overline{Q0} + Q2 \cdot Q0 + Q2 \cdot Q1$$

$$D2 = \overline{Q2} \cdot \overline{Q1} \cdot \overline{Q0} + Q2 \cdot Q0 + Q2 \cdot Q1$$

$$D2 = \overline{Q2} \cdot \overline{Q1} \cdot \overline{Q0} + Q2(Q1 + Q0)$$

$$D2 = \overline{Q2} \cdot \overline{Q1 + Q0} + Q2(Q1 + Q0)$$

$$D2 = \overline{Q2 \oplus (Q1 + Q0)}$$

$$D1 = \overline{Q1} \cdot \overline{Q0} + Q1 \cdot Q0$$

$$D1 = \overline{Q1 \oplus Q0}$$

$$D0 = \overline{Q0}$$

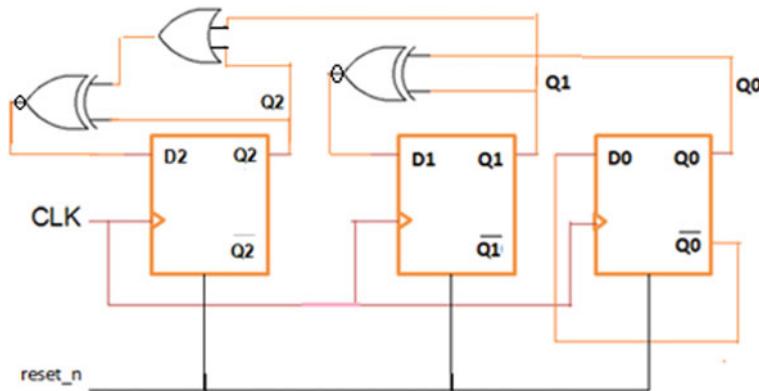


Fig. 8.20 3-bit synchronous binary down-counter

7. Let us sketch the logic

As discussed in the previous steps we need to have two flip-flops and combinational logic to implement the MOD-8 synchronous binary down-counter (Fig. 8.20).

8.7 Design of the Synchronous Gray Counter

Design the synchronous gray counter having active low asynchronous reset (reset_n), rising edge sensitive D flip-flops and the minimum logic gates.

Solution: Now let us try to use the design foundation discussed in the previous sections to design the synchronous gray counter. As name itself indicates the counter has the 4 states. s_0, s_1, s_2, s_3 and for the gray counter design using the following steps.

1. Find the number of states to get the counter

Number of states = 4.

2. Find number of flip-flops

Number of flip-flops = $n = \log_2 4 = 2$ We will use the positive edge sensitive D flip-flops.

3. Reset strategy

Let us use active low asynchronous reset input reset_n. For $\text{reset_n} = 0$ the counter holds the previous output. For the $\text{reset_n} = 1$ the counter output decrements on the rising edge of the clock.

4. Let us document the entries in the state table

The state table of the synchronous gray counter is shown in Table 8.12.

Table 8.12 State table of the 2-bit gray counter

Present state ($Q1\ Q0$)	Next state ($Q1^+ \ Q0^+$)
00	01
01	11
11	10
10	00

Table 8.13 Excitation table of the 2-bit gray counter

Present state ($Q1\ Q0$)	Next state ($Q1^+ \ Q0^+$)	Excitation input ($D1\ D0$)
00	01	01
01	11	11
11	10	10
10	00	00

5. Let us document the entries in the excitation table

The excitation table consists of the information about the present state, next state, and excitation input (Table 8.13).

6. Let us deduce the logic at the input of the $D\ 1, D\ 0$.

Let us use the present state $q1, q0$ as the inputs to get the logic at the $D1, D0$. Let us use the 2-varibale K-map (Figs. 8.21 and 8.22).

$$D1 = Q0$$

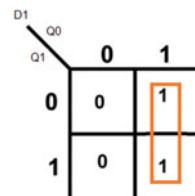
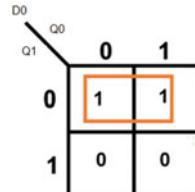
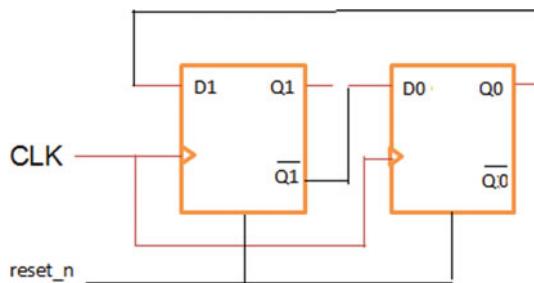
Fig. 8.21 K-map to get the logic at $D1$ **Fig. 8.22** K-map to get the logic at $D0$ 

Fig. 8.23 2-bit gray counter

$$D0 = \overline{Q1}$$

7. Let us sketch the logic

As discussed in the previous steps we need to have two flip-flops to implement the synchronous gray counter (Fig. 8.23).

8.8 Few Important Guidelines

In the VLSI design context if you are FPGA engineer or the system design engineer, then use the following guidelines:

1. Use the synchronous design as they are faster as compared to the asynchronous designs.
2. Don't use the derived clocks as they have the clock skews.
3. Asynchronous clocking is prone to glitches so avoid use of the asynchronous counter use and even asynchronous clocking.
4. Use the single PLL in the system for the single clock domain designs to derive the clock.
5. If you are using the asynchronous reset, then use the synchronizers to synchronize the asynchronous reset with the master reset.
6. Use the gray counter if the area and power requirement are minimum.

In this chapter we have discussed about the synchronous sequential circuits and design. The next chapter focuses on the sequential circuit design scenarios and asynchronous circuit design.

8.9 Summary

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

1. The SR and JK flip-flops are not recommended in the design as designer needs to control two inputs using combinational elements.

2. The D flip-flops are recommended in the digital design as designer needs to control the single input.
3. The toggle flip-flops are designed by using D or JK flip-flops.
4. The synchronous counter uses the common clock for all the flip-flops.
5. The asynchronous counters are not recommended to use in the design as they have large propagation delays.
6. Synchronous circuits are faster as compared to the asynchronous circuits.
7. MOD-4 counter indicates the 4 states and 2 flip-flops to implement the design.
8. MOD-8 counter indicates the 8 states and 3 flip-flops to implement the design.
9. In the gray counters the combinational logic is not required and hence used to minimize the power and area.
10. The Johnson counter is also called as twisted ring counter.
11. The ring counters are used to repeat the desired sequence at output.

Chapter 9

Logic Design Scenarios and Objectives

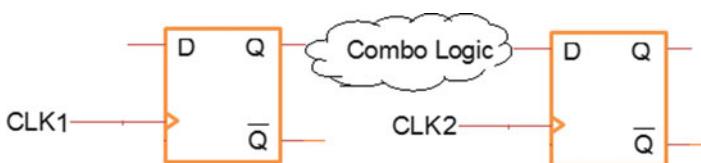


The objectives of logic design engineers are to have design with minimum area, maximum speed and low power.

In the previous chapter we have discussed about the synchronous counters and synchronous sequential circuits. It is not recommended to use the asynchronous designs due to issue of the delays and the data convergence. But the understanding of the asynchronous design can play important role during the design phase. The chapter is useful to understand the various asynchronous design techniques and practical scenarios and why asynchronous designs are not recommended in the design of SOCs? Even this chapter is useful to understand the important logic design scenarios and objective of the designer to have design with minimum area, high speed, and low power.

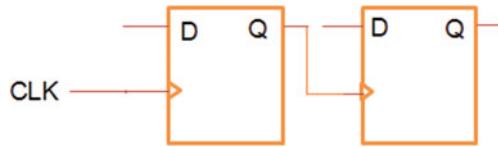
9.1 What is Asynchronous Design?

We know that in the synchronous design the clock of all the flip-flops is derived from common clock source that is Phase Lock Loop (PLL). In the asynchronous design the clock of all the flip-flops is not derived from the same clocking source, and hence there is phase shift in the clock of various flip-flops.



Let us consider the design shown in the figure and has clocking sources CLK1 and CLK2. The CLK1 is derived from the PLL1, and CLK2 is derived from PLL2.

Fig. 9.1 Asynchronous design



If the CLK1 and CLK2 frequency is also same, then the design is considered as asynchronous design or multiple clock domain design. The issue in such type of design is due to the phase shift between the CLK1 and CLK2.

Consider another example of the asynchronous design the clock of the various sequential elements is driven by the different clock source. As shown in Fig. 9.1 the LSB flip-flop receives the clock from the PLL and the MSB flip-flop receives the clock as one of the outputs of the LSB flip-flop.

9.2 Synchronous Versus Asynchronous Reset

The reset and clocking strategies play important role, and we need to have the reset strategies to initialize the design. We can consider the reset as the default state. The reset is of two types: asynchronous reset and synchronous reset. During the architecture design we need to choose the type of the reset and even the better understanding of the reset removal time and reset recovery time. If the asynchronous reset is the type, then we need to deploy the synchronizer in the reset path. The following section discusses about the asynchronous and synchronous reset.

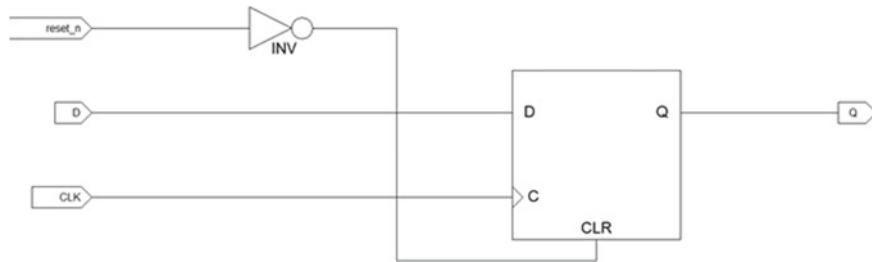
9.2.1 *D Flip-Flop Having Asynchronous Reset*

The asynchronous reset is an issue in ASIC or SOC designs as sampling of the reset is independent of active edge of the clock. The reset signal is used to clear the sequential logic at any instance of time irrespective of active edge of the clock.

In the asynchronous reset type, reset logic is not part of data path, and even as discussed previously the internally generated resets or asynchronous resets are not recommended in the ASIC design as they are prone to glitches. Even reset recovery and the reset removal are 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 synchronizers. This is also called as the reset synchronization. The active

Table 9.1 The truth table of the asynchronous reset logic

reset_n	clk	D	Q
0	X	X	0
1	Active edge	0	0
1	Active edge	1	1

**Fig. 9.2** Logic diagram of D flip-flop with asynchronous reset

edge can be positive or negative depending on the use of the positive or negative edge sensitive flip-flop, respectively.

The truth table of flip-flop having active low asynchronous reset is shown in Table 9.1.

If reset_n is equal to logic 0, then irrespective of active edge of the clock the flip-flop output is forced to logic 0. The flip-flop output is $Q = D$ when $\text{reset_n} = 1$ on rising edge of the clock.

The logic diagram of the D flip-flop having asynchronous reset is shown in Fig. 9.2.

9.2.2 *Synchronous Reset D Flip-flop*

The synchronous reset is a better strategy in the ASIC or SOC design as sampling of the reset is dependent on the active edge of the clock. The reset signal is used to clear or to initialize the sequential logic on the rising edge of the clock.

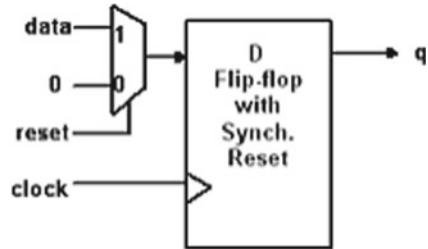
The reset logic is part of data path and not prone to glitches. Even reset recovery is not an issue, and if synchronous reset inputs are used, then the design does not need the use of level synchronizer!

The truth table of flip-flop having synchronous reset is shown in Table 9.2.

Table 9.2 The truth table of the synchronous reset logic

reset	clk	D	Q
0	Active edge	X	0
1	Active edge	0	0
1	Active edge	1	1

Fig. 9.3 Logic diagram of D flip-flop with synchronous reset



If reset_n is equal to logic 0, then on the active edge of the clock the flip-flop output is forced to logic 0. The flip-flop output is Q which is equal to data input D when $\text{reset_n} = 1$ on rising edge of the clock.

The logic diagram is shown in Fig. 9.3 where reset logic is included in the data path.

9.3 Asynchronous MOD Counters

In the asynchronous counters the clock signal of all the flip-flops is not driven by the common clock source. If the output of LSB flip-flop is used as a clock input to the subsequent flip-flop, then the design is said to be asynchronous. The issue with the asynchronous design is the addition of clock to q delay of flip-flop due to the cascading of the number of flip-flop stages. Asynchronous counters are not recommended in the ASIC or FPGA designs due to the issue of glitches or spikes, and even the timing analysis for such kind of design is extraordinarily complex.

Consider the four-bit ripple counter. The timing diagram is shown in Fig. 9.4. During design we have used the toggle that is T flip-flops, and when $T = 1$ that is toggle input($\text{toggle_in} = 1$) flip-flop changes state. When toggle_in and reset_n both are at logic level 1 the counter counts from 0 to F . The output format is hexadecimal.

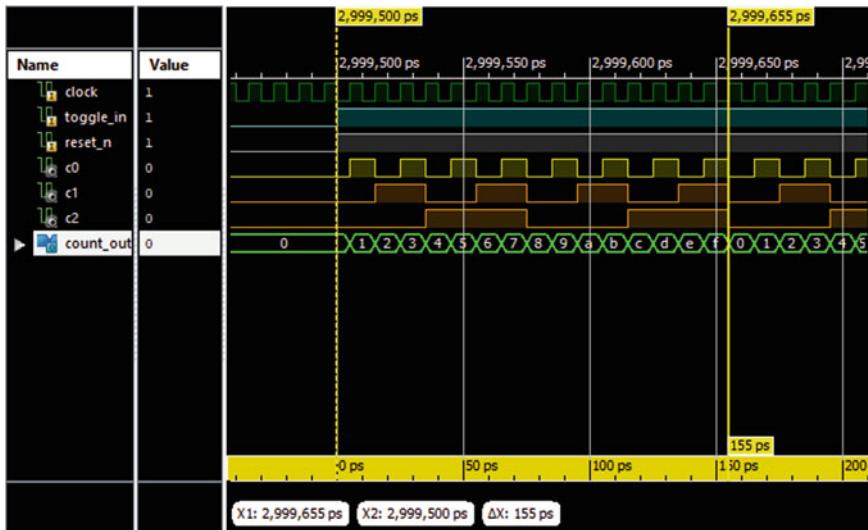


Fig. 9.4 Timing diagram of 4-bit ripple counter

9.3.1 Frequency Divider Network

Now let us discuss the frequency divider. As we know that the toggle flip-flop using JK or D acts as divide by 2 counter, the output frequency from the toggle flip-flop is input clock divided by 2. The divide by 2 using the JK flip-flop is shown in Fig. 9.5.

The timing diagram is shown in Fig. 9.6. As shown the frequency at Q is input clock frequency divided by 2.

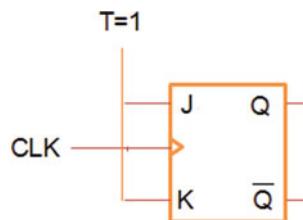


Fig. 9.5 Divide by 2 network

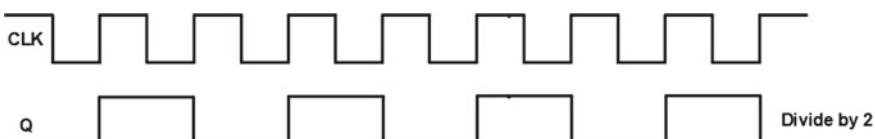
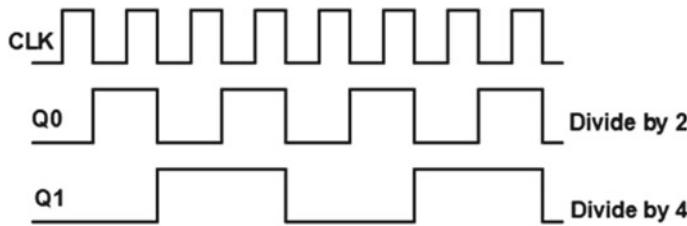
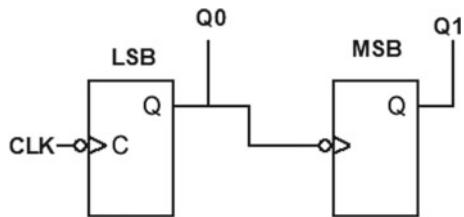


Fig. 9.6 The timing diagram of divide by 2

Fig. 9.7 Clock divider**Fig. 9.8** Timing diagram of frequency divider network

Consider the asynchronous design to get the output clock frequency as divide by 2 and divide by 4.

As shown in Fig. 9.7 the LSB flip-flop is negative edge triggered and uses the main clock (CLK). The MSB flip-flop clock is generated from the output of the LSB flip-flop. The timing diagram is shown in Fig. 9.8.

9.3.2 Ripple Counter Design

The ripple counter is an asynchronous counter, and following are important steps used to design the asynchronous counters.

- Number of flip-flops:* Identify the number of T flip-flops: Here for 3-bit counter we need to have 3 toggle flip-flops.
- State Table:* Let us describe the present state and next state

Present state ($Q_2\ Q_1\ Q_0$)	Next state ($Q_2^+\ Q_1^+\ Q_0^+$)
000	001
001	010
010	011
011	100
100	101
101	110
110	111

(continued)

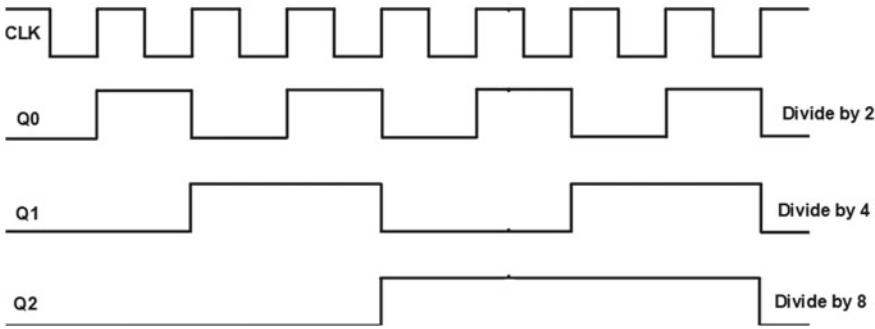


Fig. 9.9 Timing diagram of three-bit ripple counter

(continued)

Present state ($Q_2\ Q_1\ Q_0$)	Next state ($Q_2^+\ Q_1^+\ Q_0^+$)
111	000

3. *Derive the clock for flip-flop:* Let us derive the clock for the flip-flops.
 - a. LSB flip-flop receives clock CLK from PLL that is clock source. The flip-flop output is Q_0 , and frequency at Q_0 is f_{clk} divided by 2.
 - b. LSB + 1 flip-flop clock should be generated from the LSB flip-flop output. As flip-flop is positive edge sensitive the complement of Q_0 is used as clock input of the flip-flop. The output Q_1 is f_{clk} divided by 4.
 - c. MSB flip-flop clock should be generated from the LSB + 1 flip-flop output. As flip-flop is positive edge sensitive the complement of Q_1 is used as clock input of the flip-flop. The output Q_2 is f_{clk} divided by 8.
4. *Sketch timing and logic diagram:* Sketch the timing diagram of output sequence. If you observe the output sequence, then we can conclude that the state of the output Q_1 and Q_2 changes on the low to high transition of Q_0 and Q_1 , respectively.
5. *Logic Diagram:* As shown in Fig. 9.9 all the toggle flip-flops are positive edge triggered and the LSB flip-flop receives the clock from the master clock source. The output of LSB flip-flop is used as clock input to the next subsequent stage flip-flop. The logic digarm is shown in the Fig. 9.10.

9.4 Design Scenario

Consider that the design requirement is to use negative edge sensitive flip-flops. Then what we can do? The best strategy is to use the existing design of asynchronous counter (Fig. 9.10) and modify it using the negative edge triggered flip-flops.

1. **Step 1:** Sketch the timing sequence of 3-bit asynchronous counter (Fig. 9.11).

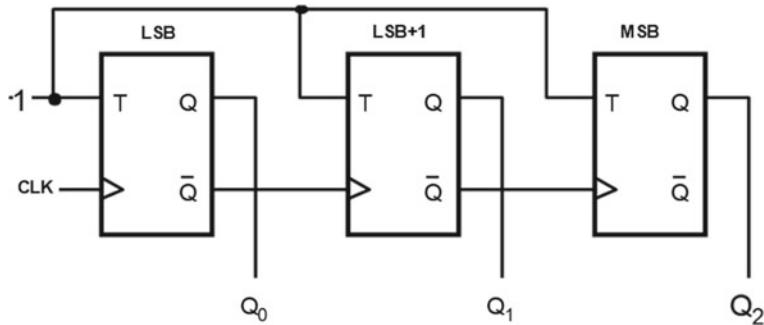


Fig. 9.10 Logic diagram of three-bit ripple up-counter

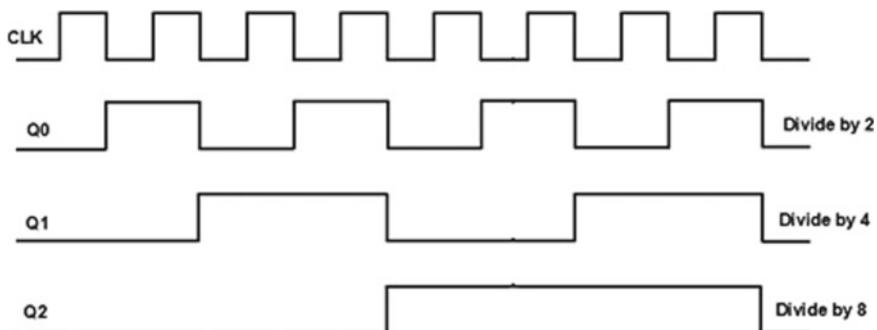


Fig. 9.11 The timing sequence of 3-bit ripple counter (negative level sensitive flip-flops)

- Step 2: As shown in the figure the frequency at Q₀ is divide by 2. The frequency at Q₁ is divide by 4, and the frequency at Q₂ is divide by 8. The logic diagram is shown in the figure. The LSB + 1 flip-flop output changes on the high to low transition of Q₀, and the MSB flip-flop output changes on the high to low transition of Q₂ (Fig. 9.12).

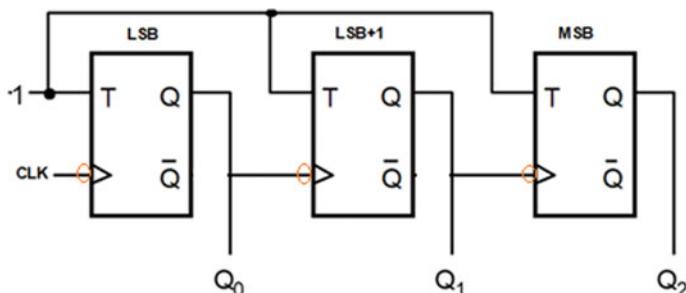


Fig. 9.12 Logic diagram of 3-bit asynchronous counter

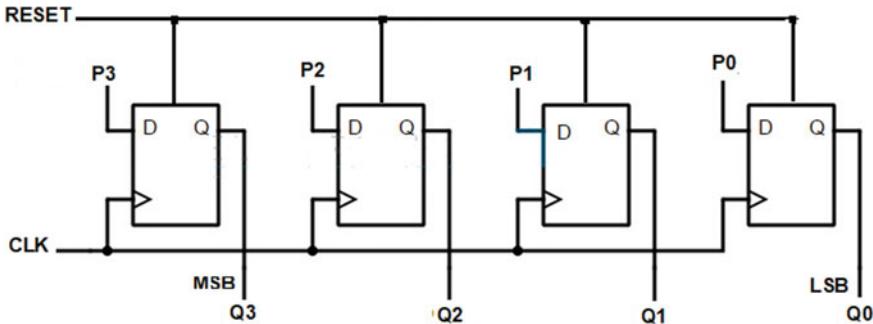


Fig. 9.13 The logic diagram of 4-bit PIPO register

9.5 PIPO Register

As we know that, to store the data internally the processor uses the register bank. The Parallel Input and Parallel Output register is called as PIPO register. The registers are popular as internal memory storage elements. The register receives the parallel inputs from data bus, and during write operation the outputs of register are equal to the parallel data inputs.

The design of the PIPO register is shown in Fig. 9.13: The logic diagram of 4-bit PIPO register and has 4 flip-flops. The design is synchronous, the 4-bit data inputs are P_3 , P_2 , P_1 , and P_0 , and 4-bit data outputs are Q_3 , Q_2 , Q_1 , and Q_0 .

Consider the design of the 4-bit PIPO register. The design has active high asynchronous reset input RESET and the rising edge of the clock. The design has the parallel data inputs D_3 , D_2 , D_1 , and D_0 and parallel outputs Q_3 , Q_2 , Q_1 , and Q_0 .

For the $\text{RESET} = 1$ the 4-bit register outputs are cleared. On the rising edge of the clock the 4-bit data input is sampled to get the outputs. The design is shown in Fig. 9.13.

9.6 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 right shift timing sequence which is sensitive to positive edge of the clock is shown in Fig. 9.14: Timing sequence of shift register. As shown in the timing sequence for every positive edge of the clock the data shifts by one bit, and hence for the four-bit shift register it requires four clock latency to get the valid output data.

As shown in Fig. 9.14: Timing sequence of shift register, the data is shifted on every active clock edge to get the serial output. During normal operation reset input reset_n is set to logic 1. To get the valid serial output the shift register needs four clock cycles, and hence the design has four clock latency to get the valid output.

The logic diagram of the 4-bit shift register is shown in Fig. 9.15.

9.6.1 Shift Operation and Clock Cycles

Let us discuss the right shift operation of 4-bit shift register. As shown in Fig. 9.16 the 4-bit shift register performs the right shift of the data on rising edge of clock.

To get the serial data output the logic needs four clock latency. We can use the right and left shift registers or bidirectional shift registers.

9.7 Bidirectional Shift Register

Most of the practical application demands the use of right or left shift of the data. Consider the RTL while implementing the protocol, in few serial protocols the requirement is to shift the data to the right side or to the left side by one bit or by multiple bits. In such scenario the bidirectional (right/left) shift registers can be used.

Fig. 9.14 Timing sequence of shift register

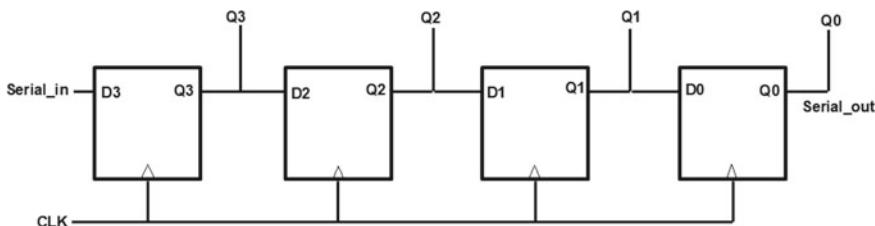
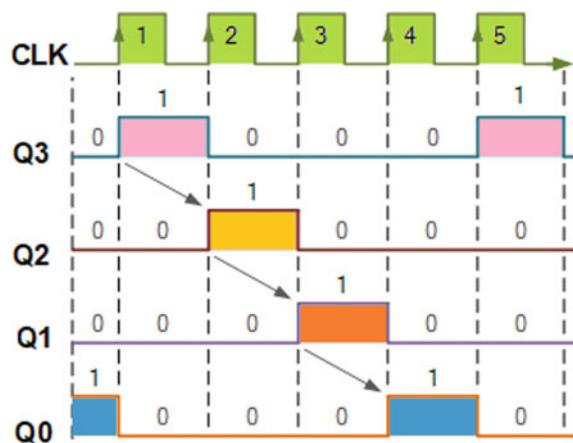
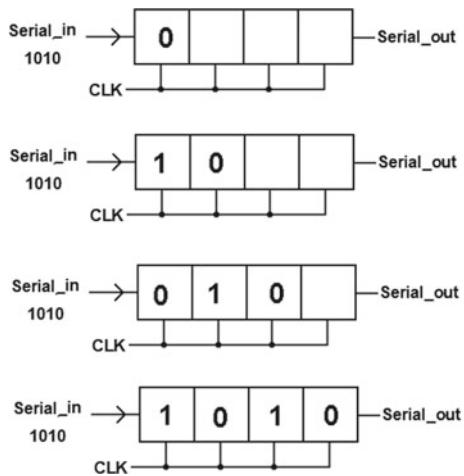
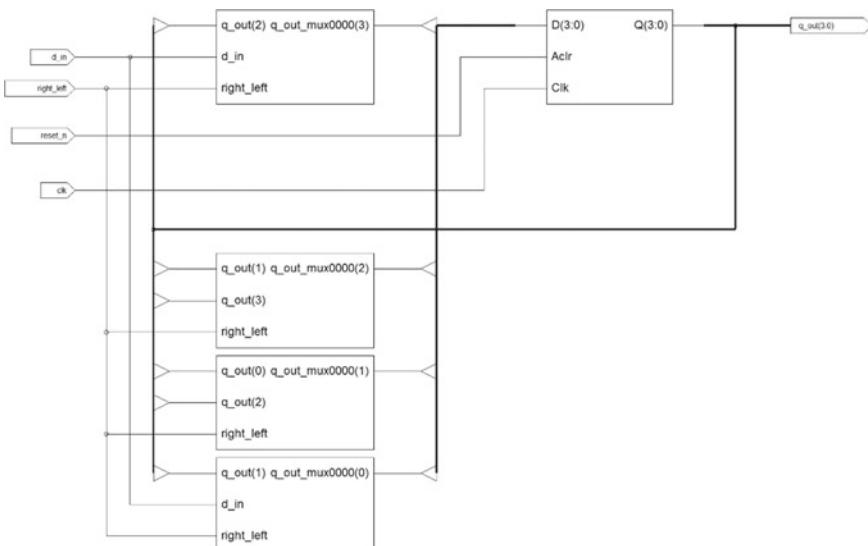


Fig. 9.15 The four-bit shift register

Fig. 9.16 The right shift

For bidirectional shift register the direction of data is controlled by `right_left` input. For `right_left` = 1 the data is shifted toward right side, and for the `right_left` = 0 the data is shifted toward left side.

The logic diagram is shown in Fig. 9.17, and the direction of data transfer is controlled by `right_left` input. The synthesized logic consists of four flip-flops with additional combinational logic to control the data flow either toward right or left side.

**Fig. 9.17** Logic of bidirectional shift register

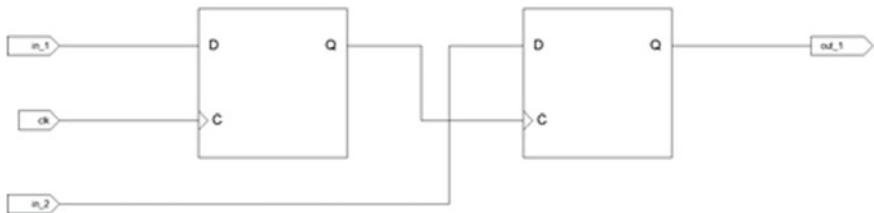


Fig. 9.18 The asynchronous design

9.8 Important Design Guidelines

Following are few of the important guidelines we can use from the VLSI context.

1. Instead of using the asynchronous designs it is better to have the synchronous designs.
2. While using the generated clocks the logic design team should take care of the cumulative delay. Consider the asynchronous design as the clock of the MSB flip-flop is derived from the LSB flip-flop output and the overall delay is $2 \cdot tpff$. The $tpff$ is flip-flop propagation delay or clock to q delay (Fig. 9.18).
3. It is recommended to avoid the generated clock as design becomes slower.
4. For the multiple clock domain designs use the synchronizers while passing data or control signals between the clock domains.
5. Avoid mix of the positive and edge sensitive flip-flops in the same clock path.
6. Use the low power design techniques to improve the power. More details are discussed in Chap. 13.
7. While passing the control signals between the multiple clock domains use the suitable synchronizers.
8. While passing the data between the multiple clock domains use the FIFO synchronizers.

In this chapter we have discussed about the asynchronous and synchronous designs and shift registers. In the next chapter we will discuss about the various sequential design scenarios.

9.9 Summary

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

1. In the synchronous design the clock of all the flip-flops is derived from common clock source that is Phase Lock Loop (PLL).
2. In the asynchronous design the clock of all the flip-flops is not derived from the same clocking source, and hence there is phase shift in the clock of various flip-flops.

3. The asynchronous reset is used to clear or initialize the sequential logic at any instance of time irrespective of active edge of the clock.
4. The synchronous reset signal is used to clear or initialize the sequential logic on the rising edge of the clock. Reset logic is part of data path and not prone to glitches.
5. In the asynchronous counters the clock signal of all the flip-slops is not driven by the common clock source. The output of LSB flip-flop is used as a clock input to the subsequent flip-flop.
6. In most of the ASIC and SOC-based designs memories are used to hold the binary data. Memories can be of type ROM, RAM, single port, or dual port.
7. For the better design performance, it is recommended to use the register inputs and register outputs.
8. It is recommended to avoid the generated clock as design becomes slower.
9. For the multiple clock domain designs use the synchronizers while passing data or control signals between the clock domains.
10. Avoid mix of the positive and edge sensitive flip-flops in the same clock path.
11. Asynchronous FIFO is the data path element and used to pass the data between the multiple clock domains.

Chapter 10

Sequential Design Scenarios



The design should have the higher speed and lesser power.

In the previous chapters we have discussed about the synchronous counters and asynchronous sequential circuits. Even we have discussed about few of the design examples using sequential logic elements to have higher speed and lesser power. In this chapter let us discuss about few design scenarios which are useful during the sequential designs.

10.1 Design Scenario I

Consider the JK flip-flop output as logic 0 at the start and find the sequence at output Q (Fig. 10.1).

To get the sequence let us create the truth table (Table 10.1).

For the first clk as $J = 1$ and $K = 1$ the flip-flop toggles. But from the second clk the $J = 1$ and $k = 0$ always so flip-flop output is set to 1. Thus, the output sequence at output of JK flip-flop is 011111....

10.2 Four-Bit Latch

As discussed in the previous chapters the latches are level sensitive and they are transparent during active level of latch enable or clk. Consider that we need to have the 4-bit D latch to demultiplex the four-bit address and data bus. To design the demultiplexing of the pins we need to use the active high enable D latches.

The design is shown in Fig. 10.2, and for $LE = 1$ the output $Q = D$, and for $LE = 0$ it holds the previous output.

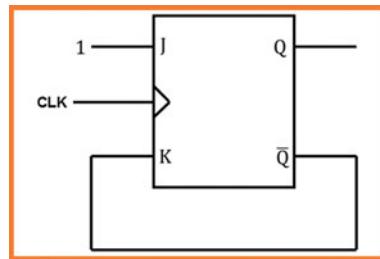


Fig. 10.1 Design using JK flip-flop

Table 10.1 The truth table of JK flip-flop

Clk	J	K	Q	$\sim Q$
0	1	1	0	1
1	1	1	1	0
2	1	0	1	0
3	1	0	1	0
4	1	0	1	0

Fig. 10.2 The multibit latch

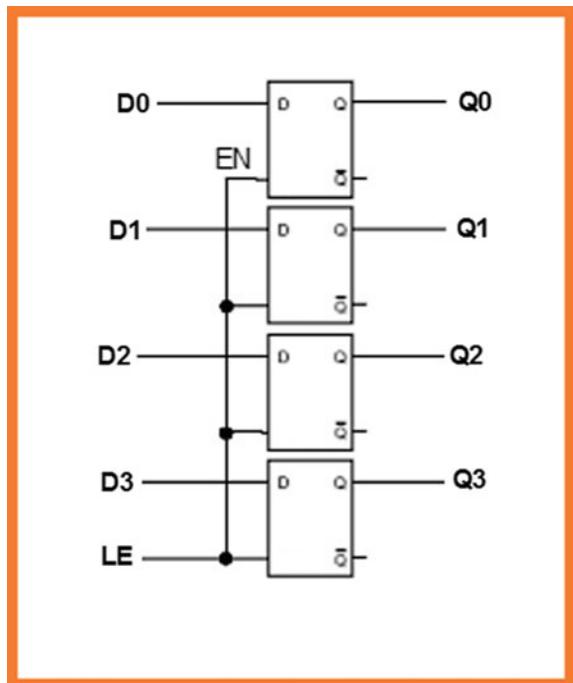
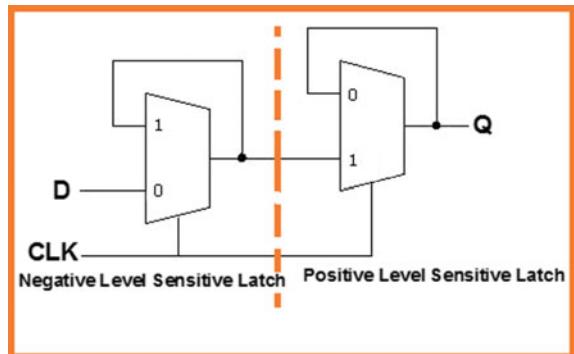


Fig. 10.3 Positive edge triggered D flip-flop



In the system design or VLSI context we can think of using the latches for demultiplexing of the shared buses.

10.3 Positive Edge Sensitive Flip-Flop Using Multiplexers

Now consider the design scenario to have the positive edge sensitive flip-flop using the multiplexers. As we know that the flip-flop is designed using cascaded latches we can design the positive level and negative level sensitive latch using 2:1 mux.

Positive Level Sensitive Latch: Using mux we can design the latch by using select line as latch enable input and data input connected to I1 input of mux. To hold the previous output, we can feedback the output of mux to I0 input.

Negative Level Sensitive Latch: Using mux we can design the latch by using select line as latch enable input and data input connected to I0 input of mux. To hold the previous output, we can feedback the output of mux to I1 input.

To design the positive edge sensitive flip-flop use input latch as negative level sensitive and output latch as positive level sensitive as shown in Fig. 10.3.

10.4 Flip-Flop Negative Edge Sensitive

Now consider the design scenario to have the negative edge sensitive flip-flop using the multiplexers. As we know that the flip-flop is designed using cascaded latches we can design the positive level and negative level sensitive latch using 2:1 mux.

Positive Level Sensitive Latch: Using mux we can design the latch by using select line as latch enable input and data input connected to I1 input of mux. To hold the previous output, we can feedback the output of mux to I0 input.

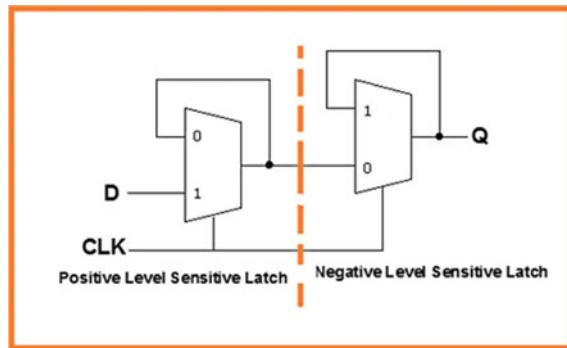


Fig. 10.4 Negative edge triggered D flip-flop

Negative Level Sensitive Latch: Using mux we can design the latch by using select line as latch enable input and data input connected to I0 input of mux. To hold the previous output, we can feedback the output of mux to I1 input.

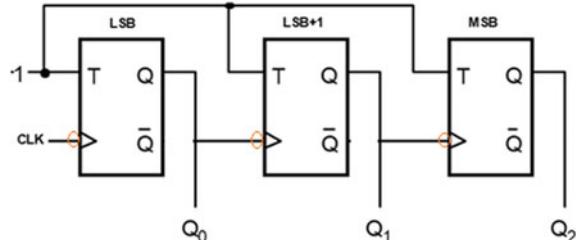
To design the negative edge sensitive flip-flop use input latch as positive level sensitive and output latch as negative level sensitive as shown in Fig. 10.4.

10.5 Timing Sequence of Design

Consider the following design (Fig. 10.5) using the T flip-flops and all the flip-flops are sensitive to falling edge of the clk. Sketch the timing sequence for Q_0 , Q_1 , and Q_2 .

From the design it is clear that the design is 3-bit up-counter and all the flip-flops are sensitive to falling edge of the clock. The first flip-flop output is divide by 2, second flip-flop output is divide by 4, and last flip-flop output is divide by 8. The timing sequence is shown in Fig. 10.6.

Fig. 10.5 Asynchronous counter



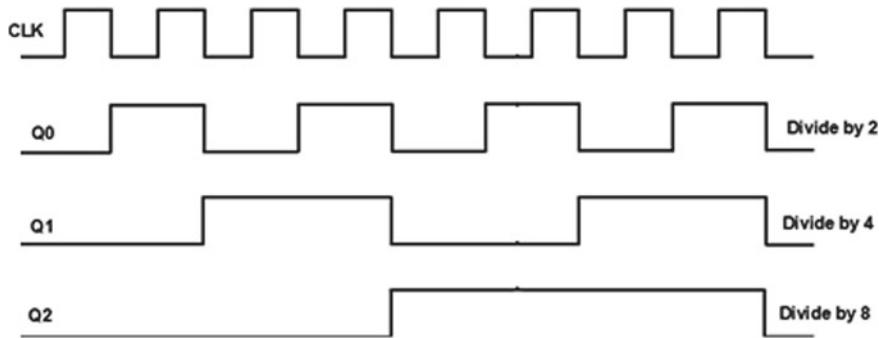


Fig. 10.6 The timing sequence of asynchronous counter

10.6 Load and Shift Register

Consider the design scenario to have the load and shift register. Following are design specifications

1. The shift register is 4-bit wide
2. The register has rising edge of the clock
3. The active high synchronous load to load the parallel data
4. Serial input and parallel output design
5. Every clock 1-bit shift
6. Active high asynchronous reset.

By considering all above what we need to do is, use the 4 D flip-flops which are sensitive to rising edge of the clk and having active high asynchronous reset.

Use multiplexer chain at the input to load the data when Load = 1 on the active edge of the clock or to shift the serial data for Load = 0.

The design is shown in Fig. 10.7.

10.7 Design Scenario II

For the circuit shown in Fig. 10.8 find the sequence at output Q .

To get the sequence at the output let us document the entries in the truth table (Table 10.2). From the table entries it is clear that we will get the output sequence at Q as 011010....

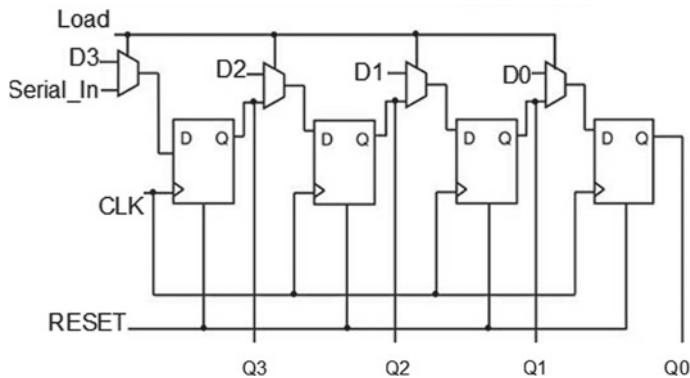


Fig. 10.7 The logic diagram of load and shift register

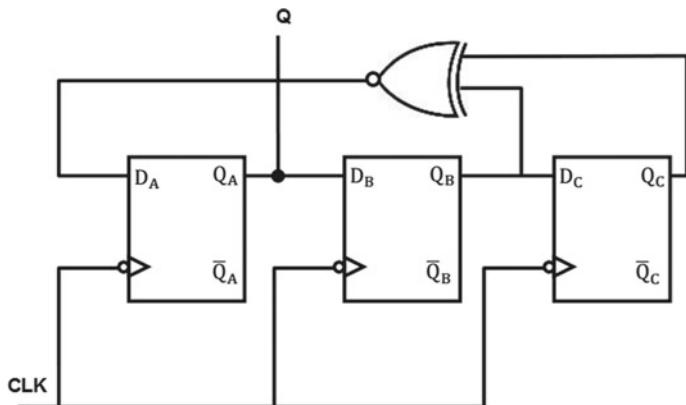
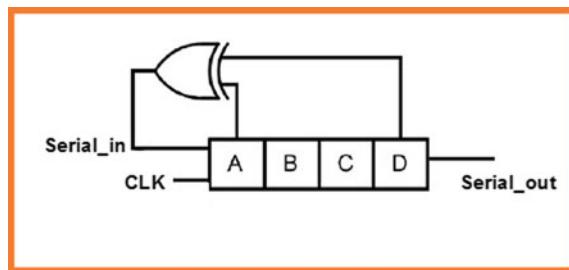


Fig. 10.8 Random number generator

Table 10.2 Table entries for design shown in Fig. 10.8

Clk	DA	DB	DC	Q
0	0	0	0	0
1	1	0	0	1
2	1	1	0	1
3	0	1	1	0
4	1	0	1	1
5	0	1	0	0

Fig. 10.9 The shift register**Table 10.3** Table entries for design shown in Fig. 10.9

Clk	A	B	C	D	Serial_out
0	1	0	0	0	0
1	1	1	0	0	0
2	1	1	1	0	0
3	1	1	1	1	1
4	0	1	1	1	1
5	1	0	1	1	1

10.8 Design Scenario III

For the shift register shown in Fig. 10.9 consider initial state as 1000. Find the sequence at output Serial_out.

To get the sequence at the Serial_out let us document the entries in the truth table (Table 10.3).

So, we will get the output sequence at Serial_out as 000111....

10.9 Design Scenario III

Consider the design requirement to have the AND gate in the design with registered inputs. Sketch the logic design. To sketch the logic design, use the following steps.

1. Use the flip-flops either negative edge sensitive or positive edge sensitive.
2. Use registered inputs.
3. Sketch the logic.

The AND gate having registered inputs is shown in Fig. 10.10.

Fig. 10.10 Registered inputs

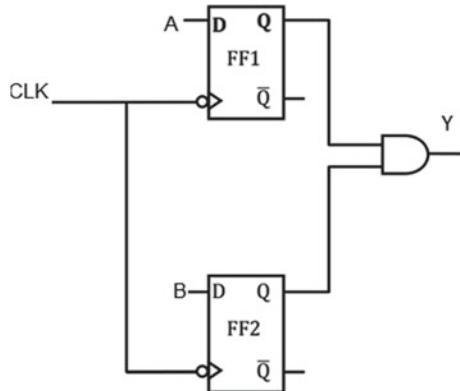


Table 10.4 Ring counter truth table

Counter output ($Q_1\ Q_0$)	Decoder Output (Y0 Y1 Y2 Y3)
00	1000
01	0100
10	0010
11	0001

10.10 Design Scenario IV

Consider the design requirement to design the ring counter using the 2:4 decoder and 2-bit binary counter.

What we can do is we can have 2-bit synchronous binary up-counter and 2:4 decoder. We can use the output of 2-bit synchronous binary up-counter as select inputs to 2:4 decoder. The output of 2:4 decoder gives us the output as 1000, 0100, 0010, 0001, 1000.... Table 10.4 describes the entries.

The logic design is shown in Fig. 10.11.

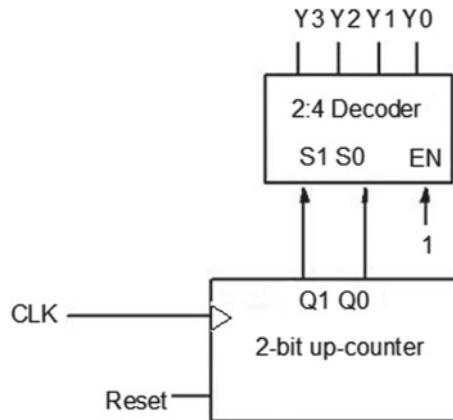
10.11 Design of 4-bit Ring Counter

As name indicates the ring counter generates an output sequence as 8, 4, 2, 1, 8, 4.... The counter can be designed very quickly using the state table. So let us discuss the steps to design the ring counter.

1. Find the maximum count

For the counter to have output as 8, 4, 2, 1 the maximum count value is 8.

2. Use the maximum count to find the number of flip-flops required

**Fig. 10.11** Ring counter**Table 10.5** The 4-bit ring counter state table

Present state ($Q_3\ Q_2\ Q_1\ Q_0$)	Next state ($Q_3^+\ Q_2^+\ Q_1^+\ Q_0^+$)
1000	0100
0100	0010
0010	0001
0001	1000

In the binary 8 is represented as 1000. Hence the number of flip-flops required is equal to 4.

3. Document the state table entries

The counter has four states, and the entries are documented in Table 10.5.

4. Document the excitation input of the flip-flops

Let us document the excitation input to get the next state count value (Table 10.6).

Table 10.6 The 4-bit ring counter excitation table

Present state ($Q_3\ Q_2\ Q_1\ Q_0$)	Next state ($Q_3^+\ Q_2^+\ Q_1^+\ Q_0^+$)	Excitation input ($D_3\ D_2\ D_1\ D_0$)
1000	0100	0100
0100	0010	0010
0010	0001	0001
0001	1000	1000

Table 10.7 Excitation table to deduce the expressions

Present State $(q_3 \ q_2 \ q_1 \ q_0)$	Next State $(q_3^+ \ q_2^+ \ q_1^+ \ q_0^+)$	Excitation Input $(D_3 \ D_2 \ D_1 \ D_0)$
1000	0100	0100
0100	0010	0010
0010	0001	0001
0001	1000	1000

5. Observe the present state and next state to get the desired value as excitation to data input of flip-flops (Table 10.7).

As shown in Table 10.7 if we compare the present state and next state then we can conclude that, on the rising edge of the clock the output of the ring counter shifts by 1 bit. So, using this understanding let us get the Boolean equations to have logic at the D_3 , D_2 , D_1 , and D_0 inputs.

6. Boolean equations

Let us get the Boolean equations

$$D_3 = Q_0$$

$$D_2 = Q_3$$

$$D_1 = Q_2$$

$$D_0 = Q_1$$

7. Let us sketch the logic

The design of the 4-bit ring counter is shown in Fig. 10.12.

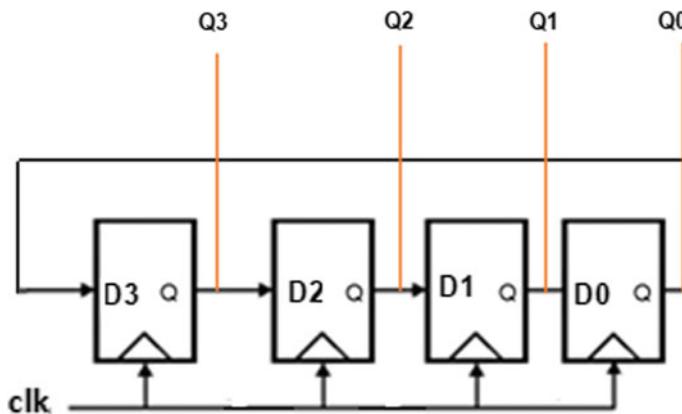


Fig. 10.12 The 4-bit ring counter design

10.12 Design of 4-bit Johnson Counter

As Johnson counter is twisted ring counter and used to generates an output sequence 0, 8, 12, 14, 15, 7, 3, 1, 0.... The counter can be designed very quickly using the state table. So let us discuss the steps to design the ring counter.

1. Find the maximum count

For the counter to have output as 0, 8, 12, 14, 15, 7, 3, 1, 0.... let us consider the maximum count value as 15. Now let us find the number of flip-flops needed.

2. Use the maximum count to find the number of flip-flops required

In the binary 15 is represented as 1111. Hence the number of flip-flops required is equal to 4.

3. Document the state table entries

The counter has four states, and the entries are documented in Table 10.8.

4. Document the excitation input of the flip-flops

Table 10.8 The 4-bit twisted ring counter state table

Present state ($Q_3\ Q_2\ Q_1\ Q_0$)	Next state ($Q_3^+\ Q_2^+\ Q_1^+\ Q_0^+$)
0000	1000
1000	1100
1100	1110
1110	1111
1111	0111
0111	0011
0011	0001
0001	0000

Table 10.9 Excitation table of twisted ring counter

Present state ($Q_3\ Q_2\ Q_1\ Q_0$)	Next state ($Q_3^+\ Q_2^+\ Q_1^+\ Q_0^+$)	Next state ($D_3\ D_2\ D_1\ D_0$)
0000	1000	1000
1000	1100	1100
1100	1110	1110
1110	1111	1111
1111	0111	0111
0111	0011	0011
0011	0001	0001
0001	0000	0000

Let us document the excitation input to get the next state count value (Table 10.9).

- Observe the present state and next state to get the desired value as excitation to data input of flip-flops (Table 10.10).

As shown in the Table 10.10 if we compare the present state and next state then we can conclude that, on the rising edge of the clock the output of the twisted ring counter shifts by 1 bit and the MSB of the counter is complement of the LSB output. So, using this let us get the Boolean equations to have D_3 , D_2 , D_1 , and D_0 inputs.

6. Boolean equations

Let us get the Boolean equations

$$D_3 = \overline{Q_0}$$

$$D_2 = Q_3$$

$$D_1 = Q_2$$

$$D_0 = Q_1$$

7. Let us sketch the logic

The design uses the four flip-flops, and the counter is called as twisted ring counter Fig. 10.13.

10.13 Duty Cycle Control

We need to have the efficient counters and shift registers. The MOD-3 counter has three states 00, 01, and 10 and is designed using minimum resources and shown in Fig. 10.14. For more details of counter design please refer Chap. 8. Due to three states the ON duty cycle is 33.33%, where duty cycle is $(T_{on}/(T_{on} + T_{off}))$, where T_{on} is ON time and T_{off} is OFF time.

Table 10.10 Excitation table to deduce the expressions

Present State $(q_3 \ q_2 \ q_1 \ q_0)$	Next State $(q_3^+ \ q_2^+ \ q_1^+ \ q_0^+)$	Next State $(D_3 \ D_2 \ D_1 \ D_0)$
0000	1000	1000
1000	1100	1100
1100	1110	1110
1110	1111	1111
1111	0111	0111
0111	0011	0011
0011	0001	0001
0001	0000	0000

The synthesis result of the MOD-3 counter is shown in Fig. 10.14 and has the 2 flip-flops and next state logic as the adder and multiplexer. It generates the sequence as 00, 01, 10, 00, 01, 10....

The simulation result of MOD-3 synchronous up-counter is shown in Fig. 10.15 and generates an output sequence as 00, 01, 10, 00, 01, 10. During $\text{reset_n} = 0$ an output is equal to 00. The counter is sensitive to rising edge of the clock.

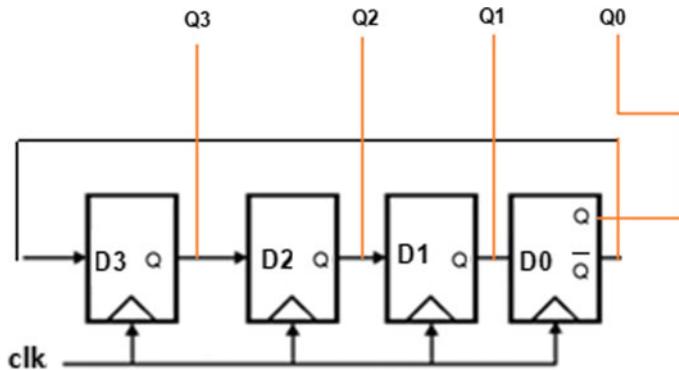


Fig. 10.13 The 4-bit Johnson counter design

10.13.1 Counter Design with 50% Duty Cycle

The MOD-3 counter which has three states 00, 01, and 10 is designed using the positive and negative edge sensitive flip-flops and other combinational elements. As shown in Fig. 10.16 the ON duty cycle is 50.00%, where duty cycle is $(T_{on}/(T_{on} + T_{off}))$, where T_{on} is ON time and T_{off} is OFF time. Both time durations are same. That is for three half cycles counter MSB output is high, and for three half cycles the MSB of counter output is low.

The strategy used is using the positive edge sensitive flip-flops and negative edge sensitive flip-flops in the parallel path to have the output with 50% duty cycle.

The logic design of the MOD-3 counter having 50% duty cycle is shown in Fig. 10.16 and has the 2 positive edge sensitive flip-flops, single negative level sensitive flip-flop, and next state logic as the adder, OR gate, and multiplexer. It generates the sequence with 50% duty cycle and counter output as 00, 01, 10, 00, 01, 10....

The simulation result of MOD-3 synchronous up-counter is shown in Fig. 10.17 and generates an output sequence as 00, 01, 10, 00, 01, 10.... During $\text{reset_n} = 0$ an output is equal to 00. The counter is sensitive to rising edge of the clock. Check during the period highlighted using two vertical yellow markers the output of MOD-3 counter is having 50% duty cycle. That is ON time is equal to OFF time.

As ON time is equal to OFF time the design has better timing performance, and if used as clock, then the equal time is provided during positive level and negative level.

In this chapter we have discussed about the sequential design scenarios and duty cycle control circuits. The next chapter is useful to understand about the timing parameters and frequency calculations.

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

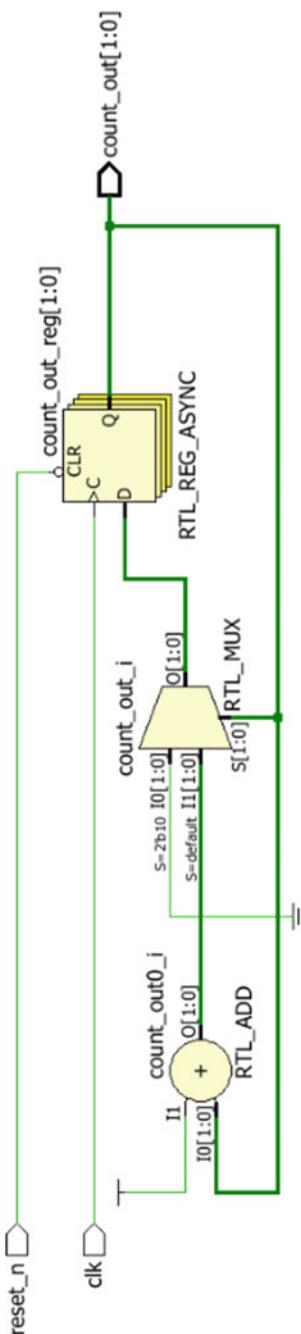


Fig. 10.14 Logic design of MOD-3 up-counter

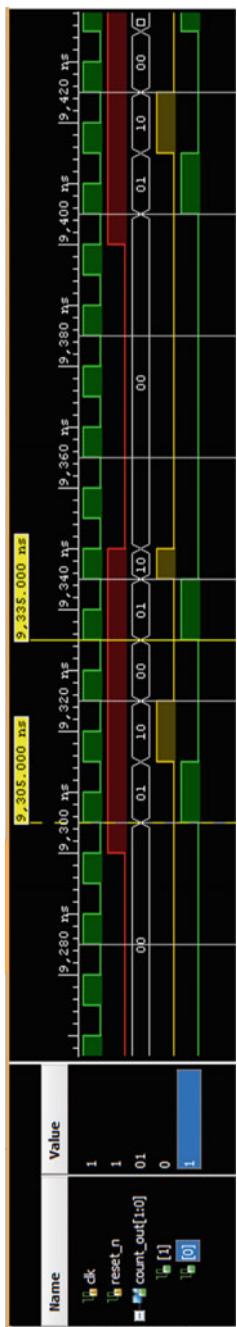


Fig. 10.15 MOD-3 up-counter waveform without having 50% duty cycle

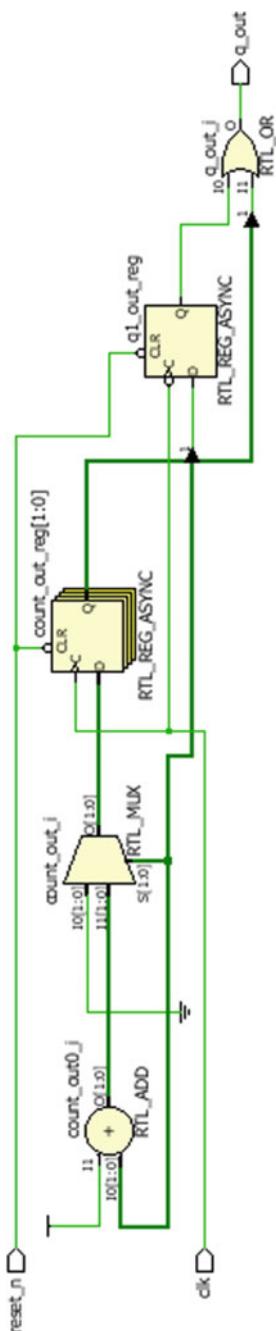


Fig. 10.16 Logic design of MOD-3 counter having 50% duty cycle



Fig. 10.17 Waveform of MOD-3 counter having 50% duty cycle output

10.14 Summary

1. Do not mix the positive edge and negative edge sensitive flip-flops.
2. The multibit latches are used in the demultiplexing logic.
3. The flip-flops are basically two latches connected in cascade.
4. The 4-bit ring counter pattern is 1000, 0100, 0010, 0001, 1000....
5. The Johnson counter is also called as twisted ring counter.
6. The duty cycle control circuit is needed to get 50% duty cycle.

Chapter 11

Timing Parameters and Maximum Frequency Calculations



The understanding of the timing and frequency calculations plays important role during various design phases.

Most of the time the design fails due to timing issues. For the logic design engineers it is essential to understand the delays and sequential circuit timings. To understand much more about the timing and frequency calculations and to design the architectures the first step is the conceptual understanding of the sequential circuit parameters. The chapter discusses about the delays, sequential circuit parameters, and frequency calculations.

11.1 What is Delay in the System?

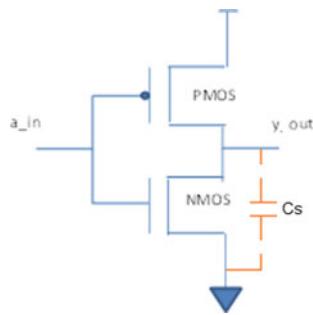
The propagation delay of any combinational logic element is the amount of time required to get the valid output after change in the input. The propagation delay is due to the charging and discharging of the stray capacitance formed at output. As we know that the voltage across capacitor cannot change instant immediately and needs some time hence we experience the delays in the digital circuits.

Consider the NOT gate shown in Fig. 11.1. As shown the stray capacitance is formed between the output pin and ground, and for logic 1 output the capacitor charges, and for logic 0 output capacitor discharges. The output of NOT gate is logic 1 which means PMOS is ON and NMOS is OFF. The output of NOT gate is logic 0 which means PMOS is OFF and NMOS is ON.

The stray capacitance is shown by C_s , and the energy stored in capacitor is given by the equation.

$$E = \frac{1}{2} * C_s * V^2$$

The power dissipation is given by

Fig. 11.1 The NOT gate

$$P = E \cdot f$$

$$p = \frac{1}{2} * C_s * V^2 * f$$

where C_s = output stray capacitance; V = maximum supply voltage that is V_{dd} ; and f = the frequency.

So, from this expression we can conclude that there is tradeoff between the speed and power. If we try to have the low power aware architecture, then we may need to compromise on the speed.

So, the goal and objective of the logic design engineer are to have the design which has lesser area that is lower gate count, maximum speed, and low power. So, we consider the optimization constraints as area, speed, and power.

If we have the cascaded stages of the combinational logic, then the design has more propagation delay due to cumulative effect. If we have the parallel logic or parallelism, then we always experience more area but less delay.

11.1.1 Cascade Logic Elements in Design

Consider the propagation delay of each XOR gate is 1 ns and the 15 XOR gates connected in cascade. Let us find the propagation delay of cascade logic.

As shown in Fig. 11.2, the number of cascade stages is $n = 15$. Each gate has the propagation delay (tpd) of 1.0 n.

So, the cascade logic has propagation delay of $n * tpd = 15 * 1.0$ ns = 15 ns.

It is recommended to avoid the cascade stages in the design as they add significant amount of the delay.

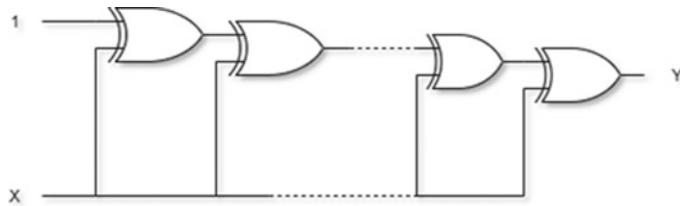


Fig. 11.2 Odd gates in cascade

11.1.2 Parallel Logic Elements in Design

If we have the parallel logic in the design, then we can have less delay; again it is design specific which is depending on the gate count of the design.

Consider the design of the half adder which is shown in Fig. 11.3.

For the half-adder design if we consider the delay of each logic gate as 1 ns, then the overall propagation delay is just 1 ns. Both the XOR and AND receive parallel inputs a_{in} and b_{in} .

It is recommended to use the parallel logic if area is not a major constraint in the design. *In the parallel logic the overall delay is minimized due to concurrent operations.* The basic concept of the parallelism in the processors is also evolved from the parallel logic.

11.2 How Delays Affect the Performance of the Design?

The delays affect the overall performance of the design. Consider the design of the 4:1 mux shown in Fig. 11.4. As shown the 4:1 mux is implemented using the cascaded

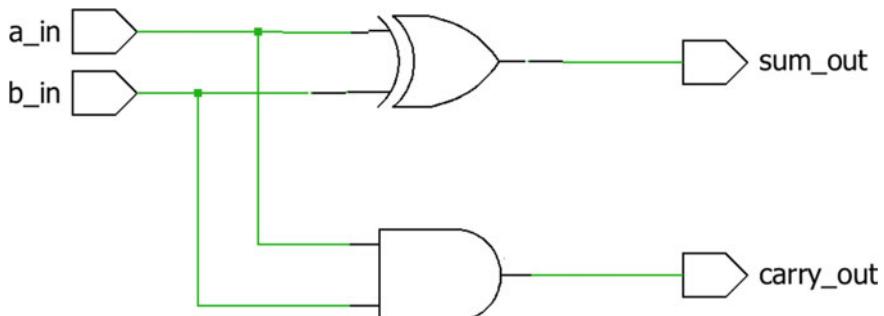
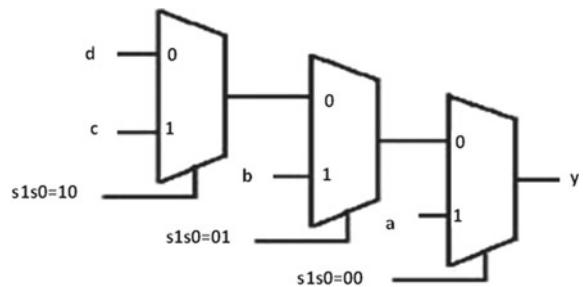


Fig. 11.3 The half adder using logic gates

Fig. 11.4 Priority mux

2:1 mux. If each mux has the propagation delay of 0.5 ns, then the overall propagation delay is 1.5 ns for the c or d input to propagate to output.

Hence avoid the cascade of the multiplexers and even the design behaves as priority multiplexer where a has highest priority and d has lowest priority.

In the design, for the $S1S0 = 00$ an output $y = a$, for the $S1S0 = 01$ an output $y = b$, for the $S1S0 = 10$ an output $y = c$, and for $S1S0 = 11$ an output $y = d$.

To avoid the priority multiplexing in the design, let us think about use of the parallel logic to get the 4:1 mux. The strategy is use two 2:1 multiplexers at inputs and single 2:1 mux at output. For more details, please refer Chap. 4.

As shown in Fig. 11.5, the 4:1 mux is designed using the minimum number of 2:1 mux, and if each mux has delay of 0.5 ns, the overall delay is $2 \times 0.5 \text{ ns} = 1 \text{ ns}$. This is better as compared to priority mux as the delay is less and the area is less.

Now let us think, for the design shown in Fig. 11.5, how area is minimum as compared to design shown in Fig. 11.4?

For the priority mux design (Fig. 11.4) we need additional decoding logic to pass the single select input depending on the status of $S1$, $S0$. But for the parallel logic mux (Fig. 11.5) the additional decoding logic is not required, hence reduction in the area.

11.3 Sequential Circuit and Timing Parameters

As we know that, the timing of digital circuits is very important, and if the timing is not met, then it is really nightmare for the logic design engineers. So, let us discuss important timing parameters of the flip-flop. The important timing parameters of flip-flop are shown in Fig. 11.6, and they are as follows:

1. Setup time (t_{su})
2. Hold time (t_h)
3. Propagation delay of flip-flop (t_{pff}).

Fig. 11.5 The 4:1 mux design using 2:1 multiplexers

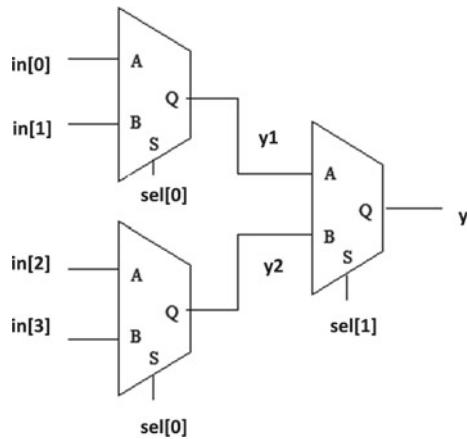
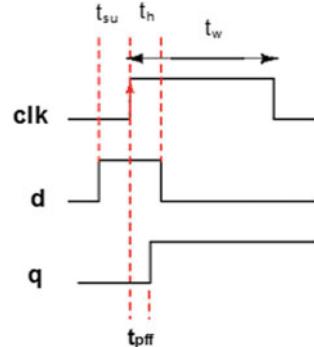


Fig. 11.6 The timing parameters for D flip-flop



- **Setup Time (t_{su}):** The minimum amount of time for which the data input of the flip-flop should maintain the stable value prior to arrival of the active edge of the clock is called as setup time.

To meet the setup time, it is required that the data should arrive at the input of D flip-flop before arrival of the active clock edge. For example, if we consider design operated with 250 MHz clock frequency (clock cycle time = 4 ns) and has setup time requirement of 1 ns, then it is required that data should arrive at least at 3 ns or prior to that. This ensures that the setup time is met and design will not have the setup violation.

During the specified setup time window if the data input changes, then the flip-flop output will be metastable and it indicates the setup violation.

- **Hold Time (t_h):** The minimum amount of time for which the data input of the flip-flop should maintain the stable value after the arrival of the active edge of the clock is called as hold time.

- Consider that the design is constrained at 250 MHz, that is clock cycle time is 4nsec. If hold time requirement is 0.5 ns and data arrived at D input of flip-flop changes during the 0.5 ns window after arrival of active clock edge, then there is hold violation in the design.

During the hold time window if the data input changes, then the flip-flop output will be metastable which indicates the hold violation.

- **Propagation Delay of flip-flop ($t_{pff} = t_{cq}$):** The amount of time required for the flip-flop to get the valid data at output after the arrival of the active edge of the clock is called as propagation delay of flip-flop.

The propagation delay is also called as the **clock to q delay**, and it is also referred as t_{ctoq} .

11.4 Timing Paths in Design

As discussed above, if the hold time or setup time is violated, then the design output is metastable. So, for the logic design engineers it is better to know about the various timing paths in the design.

Timing paths in design start at start point. The clock port or clock pin of the flip-flop or data input port of the design is called as start point. Timing path terminates or ends at the end point. The data input of D flip-flop or an output port is called as end point.

For any sequential 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).

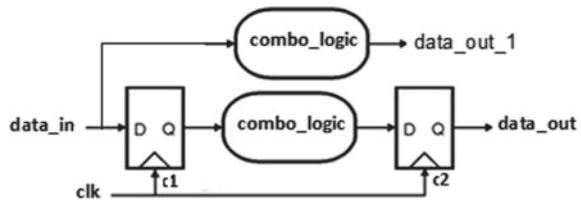
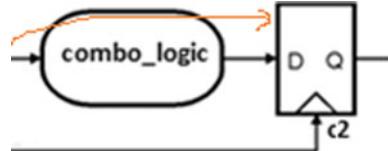
We check for the functional correctness of the design during the design simulation or verification stage. During timing analysis we need to check for the timing correctness of the design. The Static Timing Analysis (STA) is non-vectorized approach, and during the STA phase we always use the timing analysis tools. But to understand the timing analysis and timing reports for the design we need to have detail understanding of the setup slack, hold slack, and maximum frequency calculations.

So let us consider the design shown in Fig. 11.7, and let us find the timing paths.

Let us find the number of timing paths in the design shown in Fig. 11.7.

To find the number of timing paths for the design let us identify the start and stop points.

For the design, consider the start point as clock pin of the flip-flop or input port `data_in` and end point as data input D of sequential element, `data_out` and `data_out_1` that is output port.

Fig. 11.7 Sequential design**Fig. 11.8** Input to register path

- Input to reg path:** From input port `data_in` to data input of the first flip-flop
- Reg to output path:** From the clock pin `c2` to output port `data_out`
- Reg-to-reg path:** From clock pin `c1` to data input `D` of the second flip-flop
- Input to output path:** From input port `data_in` to output port `data_out_1`. This path is unconstrained path and also called as combo path.

So, the design shown in Fig. 11.7 has four timing paths.

To have more detail understanding of the timing analysis, the following section is useful to understand the different timing paths in much more detail.

11.4.1 *Input to Register Path*

Input to register path or input to reg path has start point input port and end point as data input `D` of the flip-flop. Figure 11.8 has the input port and combinational logic (Combo logic), and the path from input to `D` through Combo logic is called as input to reg path.

11.4.2 *Register to Output Path*

Register to output path or reg to output path has start point as clock input pin and end point as output port `q_out` of the register element. Figure 11.9 has the start point as `clk c1`, and data travels through the register through combinational logic and hence named as register to output path.

Fig. 11.9 Register to output path

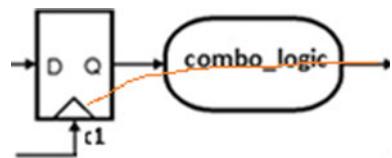
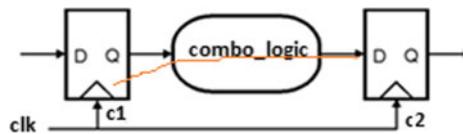


Fig. 11.10 Register to register path



11.4.3 Register to Register Path

Register to register path or reg-to-reg path has start point clock input pin c_1 , and first flip-flop is launch flip-flop. The end point of the reg-to-reg path is data input D of the second flip-flop. Figure 11.10 has the clock port clk , and data is launched by c_1 on the rising edge of the c_1 . The data output of flip-flop1 passes through the combinational logic (Combo logic) and arrives at the data input D of the second flip-flop. The time required to arrive the data at the D input of second flip-flop is called as arrival time.

The clock c_1 is launch clock, and the clock c_2 is capture clock.

11.4.4 Input to Output Path

Input to output path has start point as input port $data_in$ and end point as data output $data_out_1$. This path is also called as combinational path. Figure 11.11 has the input port $data_in$, and the data passes through the combinational logic (Combo logic) to generate an output $data_output_1$.

11.5 Maximum Frequency Calculations

To have more understanding about the use of the timing paths let us try to complete the maximum frequency calculations for following few designs.

Fig. 11.11 The combinational path



11.5.1 Design 1: Toggle Flip-Flop

Let us now find the maximum frequency for the design shown in Fig. 11.12.

To find the maximum operating frequency for the design let us identify the timing paths. The design has single timing path that is reg-to-reg path.

The start point is clk, and end point is D of flip-flop.

1. Let us find out the data arrival time (AT). $AT = t_{pdff1} + t_{not}$.
2. The data required time is RT. $RT = T_{clk} - t_{su}$ because the data at D input should be stable by t_{su} margin prior to arrival of rising edge of the clock.
3. Now find setup slack. $Slack = RT - AT$ and should be greater than or equal to 0.
4. $Slack = RT - AT$

$$\text{Setup Slack} = (T_{clk} - t_{su}) - (t_{pdff1} + t_{not})$$

5. If we equate the slack to zero, then we get

$$0 = (T_{clk} - t_{su}) - (t_{pdff1} + t_{not})$$

$$T_{clk} = t_{pdff1} + t_{not} + t_{su}$$

6. The maximum operating frequency of the design is f_{max}

$$f_{max} = \frac{1}{T_{clk}}$$

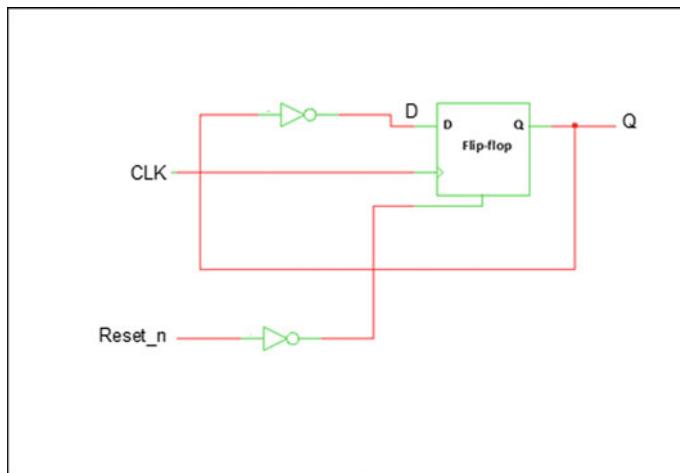


Fig. 11.12 The toggle flip-flop

$$= \frac{1}{t_{\text{pdff1}} + t_{\text{not}} + t_{\text{su}}}$$

11.5.2 Design II: The 2-bit Synchronous Up-Counter

Let us find the maximum operating frequency for the design shown in Fig. 11.13.

We need to identify the timing paths in the design, and the design has two reg-to-reg paths.

1. The reg-to reg path I

- The start point is clk, and end point is $D0$ of flip-flop.
- Let us find out the data arrival time (AT). $AT = t_{\text{pdff1}}$

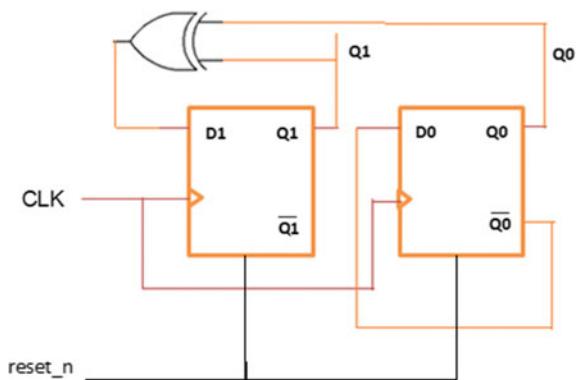
2. The reg-to reg path II

- The start point is clk of flip-flop 0, and end point is $D1$ of flip-flop.
- Let us find out the data arrival time (AT). $AT = t_{\text{pdff0}} + t_{\text{xor}}$
- For the design the arrival time is the maximum time hence $AT = t_{\text{pdff0}} + t_{\text{xor}}$. The propagation delay of both the flip-flops is of same value.
- The data required time is RT. $RT = T_{\text{clk}} - t_{\text{su}}$ because the data at D input should be stable by t_{su} margin prior to arrival of rising edge of the clock.
- Now find setup slack. $\text{Slack} = RT - AT$ and should be greater than or equal to 0.
- Setup Slack = $RT - AT$

$$= (T_{\text{clk}} - t_{\text{su}}) - (t_{\text{pdff1}} + t_{\text{xor}})$$

- If we equate the slack to zero, then we get

Fig. 11.13 The synchronous 2-bit binary up-counter



$$0 = (T_{\text{clk}} - t_{\text{su}}) - (t_{\text{pdff1}} + t_{\text{xor}})$$

$$T_{\text{clk}} = t_{\text{pdff1}} + t_{\text{xor}} + t_{\text{su}}$$

8. The maximum operating frequency of the design is f_{\max}

$$f_{\max} = \frac{1}{T_{\text{clk}}}$$

$$= \frac{1}{t_{\text{pdff1}} + t_{\text{not}} + t_{\text{su}}}$$

If we consider the $t_{\text{su}} = 1$ ns, $t_{\text{pdff0}} = 1$ ns, and $t_{\text{xor}} = 1$ ns, then the maximum operating frequency of the design is equal to

$$= \frac{1}{1 + 1 + 1}$$

$$f_{\max} = \frac{1}{3 \text{ ns}}$$

$$f_{\max} = 333.33 \text{ MHz}$$

11.6 Maximum Operating Frequency

Now let us find the maximum operating frequency for the design shown in the Fig. 11.14.

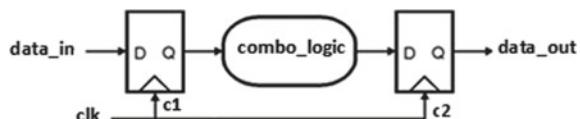
Let us identify the timing paths for the design. The design has three timing paths. But we will use the reg-to-reg timing path as it is critical path of the longest time duration.

To find the maximum operating frequency for the design let us use the reg-to-reg timing path. In the design the start point is c1 and end point is D of second flip-flop. The first flip-flop is launch flip-flop, and second flip-flop is capture flip-flop.

Use the following steps to find the maximum frequency calculations.

1. Let us find out the data arrival time (AT). $AT = t_{\text{pdff1}} + t_{\text{combo}}$
2. The data required time is RT. $RT = T_{\text{clk}} - t_{\text{su}}$ because the data at D input should be stable by t_{su} margin prior to arrival of rising edge of the clock.
3. Now find setup slack. $\text{Slack} = RT - AT$ and should be greater than or equal to 0.

Fig. 11.14 The reg-to-reg path



4. Setup Slack = RT – AT
5. Setup slack = $(T_{\text{clk}} - t_{\text{su}}) - (t_{\text{pdff1}} + t_{\text{combo}})$
6. If we equate the slack to zero, then we get

$$0 = (T_{\text{clk}} - t_{\text{su}}) - (t_{\text{pdff1}} + t_{\text{combo}})$$

$$T_{\text{clk}} = t_{\text{pdff1}} + t_{\text{combo}} + t_{\text{su}}$$

7. The maximum operating frequency of the design is f_{\max}

$$f_{\max} = \frac{1}{T_{\text{clk}}}$$

$$= \frac{1}{t_{\text{pdff1}} + t_{\text{combo}} + t_{\text{su}}}$$

11.6.1 Maximum Operating Frequency for Synchronous Designs

Now let us consider the synchronous design shown in Fig. 11.15 and let us try to find the maximum operating frequency for the design (Fig. 11.15).

The design has four flip-flops and has the four timing paths. To find the maximum operating frequency for the design use any of the reg-to-reg timing path. In the design the start point is clk and end point is D of the flip-flop.

1. Let us find out the data arrival time (AT). $AT = t_{\text{pdff1}}$
2. The data required time is RT. $RT = T_{\text{clk}} - t_{\text{su}}$ because the data at D input should be stable by t_{su} margin prior to arrival of rising edge of the clock.
3. Now find setup slack. $\text{Slack} = RT - AT$ and should be greater than or equal to 0.
4. Setup Slack = RT – AT

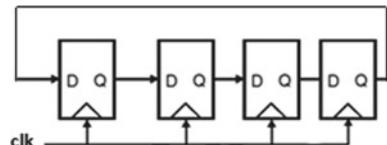
$$\text{Setup Slack} = (T_{\text{clk}} - t_{\text{su}}) - (t_{\text{pdff1}})$$

5. If we equate the slack to zero, then we get

$$0 = (T_{\text{clk}} - t_{\text{su}}) - (t_{\text{pdff1}})$$

$$T_{\text{clk}} = t_{\text{pdff1}} + t_{\text{su}}$$

Fig. 11.15 The ring counter



6. The maximum operating frequency of the design is f_{\max}

$$\begin{aligned} f_{\max} &= \frac{1}{T_{\text{clk}}} \\ &= \frac{1}{t_{\text{pdff1}} + t_{\text{su}}} \end{aligned}$$

11.7 Clock Skew

Practically if we consider any of the system design or VLSI design, then we experience the clock skew. The skew is due to arrival of the clock edge at different time instance. That is for the synchronous design the clock skew is due to clock buffers included in the clock path.

The skew are of two types:

1. *Positive clock skew*: If the source or launch flip-flop is triggered first and the capture or destination flip-flop is triggered last, then the design has positive clock skew.

In the positive clock skew designs the data and clock travel in the same direction.

2. *Negative clock skew*: If the source or launch flip-flop is triggered last and the capture or destination flip-flop is triggered first, then the design has negative clock skew.

In the negative clock skew designs the data and clock travel in the opposite direction.

Now let us understand the maximum operating frequency for the design having clock skew either positive or negative.

11.7.1 Positive Clock Skew and Maximum Operating Frequency

Let us find the maximum operating frequency for the design shown in Fig. 11.16.

To find the maximum operating frequency for the design use the reg-to-reg timing path. In the design the start point is c1 of launch flip-flop and end point is D of capture flip-flop.

Fig. 11.16 The positive clock skew



- Let us find out the data arrival time (AT). $AT = t_{pdff1} + t_{combo}$
- The data required time is RT. $RT = T_{clk} - t_{su} + t_{buf}$ because the data at D input should be stable by t_{su} margin prior to arrival of rising edge of the clock and clock is delayed by the buffer delay.
- Now find setup slack. $Slack = RT - AT$ and should be greater than or equal to 0.
- $Slack = RT - AT$

$$\text{Setup Slack} = (T_{clk} - t_{su} + t_{buf}) - (t_{pdff1} + t_{combo})$$

- If we equate the slack to zero, then we get

$$0 = (T_{clk} - t_{su} + t_{buf}) - (t_{pdff1} + t_{combo})$$

$$T_{clk} = t_{pdff1} + t_{combo} + t_{su} - t_{buf}$$

- The maximum operating frequency of the design is f_{max}

$$f_{max} = \frac{1}{T_{clk}}$$

$$= \frac{1}{t_{pdff1} + t_{combo} + t_{su} - t_{buf}}$$

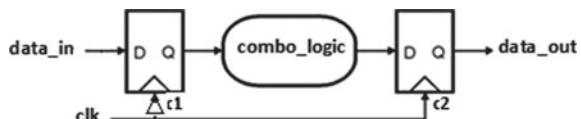
11.7.2 Negative Clock Skew and Maximum Operating Frequency for the Design

Now let us find the maximum operating frequency for the design shown in Fig. 11.17.

To find the maximum operating frequency for the design use the reg-to-reg timing path. In the design the start point is $c1$ and end point is D of second flip-flop.

- Let us find out the data arrival time (AT). $AT = t_{pdff1} + t_{combo}$.
- The data required time is RT. $RT = T_{clk} - t_{su} - t_{buf}$ because the data at D input should be stable by t_{su} margin prior to arrival of rising edge of the clock and clock is delayed by the buffer at flip-flop 1.
- Now find setup slack. $Slack = RT - AT$ and should be greater than or equal to 0.
- $Slack = RT - AT$

Fig. 11.17 The negative clock skew



$$\text{Setup Slack} = (T_{\text{clk}} - t_{\text{su}} - t_{\text{buf}}) - (t_{\text{pdff1}} + t_{\text{combo}})$$

5. If we equate the slack to zero, then we get

$$0 = (T_{\text{clk}} - t_{\text{su}} - t_{\text{buf}}) - (t_{\text{pdff1}} + t_{\text{combo}})$$

$$T_{\text{clk}} = t_{\text{pdff1}} + t_{\text{combo}} + t_{\text{su}} + t_{\text{buf}}$$

6. The maximum operating frequency of the design is f_{\max}

$$f_{\max} = \frac{1}{T_{\text{clk}}}$$

$$= \frac{1}{t_{\text{pdff1}} + t_{\text{combo}} + t_{\text{su}} + t_{\text{buf}}}$$

11.8 VLSI Specific Scenarios

Now let us consider the VLSI specific scenarios and let us try to understand the maximum frequency for the design. Practically the clock paths have the buffers and they affect on the timing of the design.

Now let us consider few VLSI scenarios and let us find the maximum operating frequency for the design.

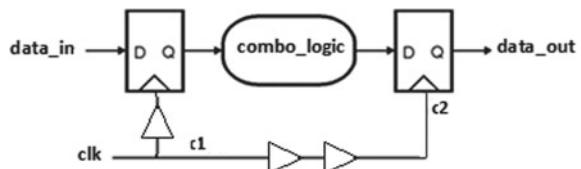
11.8.1 VLSI Specific Design Scenario I

Now let us consider few VLSI scenarios and let us find the maximum operating frequency for the design shown in Fig. 11.18.

To find the maximum operating frequency for the design use the reg-to-reg timing path. In the design the start point is $c1$ and end point is D of second flip-flop.

Here the launch flip-flop clock is delayed by single buffer delay and the capture flip-flop clock is delayed by two buffer delay time. Hence the design has positive clock skew. The single buffer delay in both the launch clock and capture path clock cancels effect of each other.

Fig. 11.18 The clock path and clock buffers



1. Let us find out the data arrival time (AT). $AT = t_{pdff1} + t_{combo}$
2. The data required time is RT. $RT = T_{clk} - t_{su} + t_{buf}$ because the data at D input should be stable by t_{su} margin prior to arrival of rising edge of the clock and clock is delayed by the buffer delay.
3. Now find setup slack. Slack = RT – AT and should be greater than or equal to 0.
4. Slack = RT – AT

$$\text{Setup Slack} = (T_{clk} - t_{su} + t_{buf}) - (t_{pdff1} + t_{combo})$$

5. If we equate the slack to zero, then we get

$$0 = (T_{clk} - t_{su} + t_{buf}) - (t_{pdff1} + t_{combo})$$

$$T_{clk} = t_{pdff1} + t_{combo} + t_{su} - t_{buf}$$

6. The maximum operating frequency of the design is f_{max}

$$f_{max} = \frac{1}{T_{clk}}$$

$$= \frac{1}{t_{pdff1} + t_{combo} + t_{su} - t_{buf}}$$

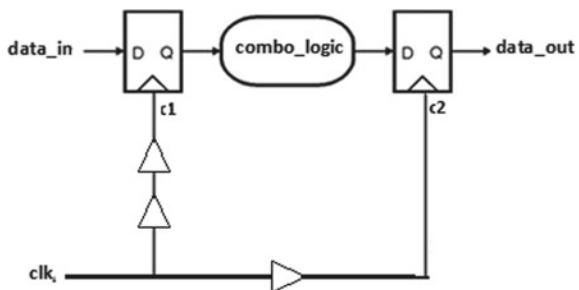
11.8.2 VLSI Specific Design Scenario II

Now let us find the maximum operating frequency for the design shown in the Fig. 11.19.

To find the maximum operating frequency for the design use the reg-to-reg timing path. In the design the start point is $c1$ and end point is D of second flip-flop.

Here the launch flip-flop clock is delayed by two buffer delay and the capture flip-flop clock is delayed by single buffer delay time. Hence the design has negative

Fig. 11.19 The clock buffers in clock path



clock skew. The single buffer delay in both the launch clock and capture path clock cancels effect of each other.

1. Let us find out the data arrival time (AT). $AT = t_{pdff1} + t_{combo}$
2. The data required time is RT. $RT = T_{clk} - t_{su} - t_{buf}$ because the data at D input should be stable by t_{su} margin prior to arrival of rising edge of the clock and clock is delayed by the buffer at flip-flop 1.
3. Now find setup slack. $Slack = RT - AT$ and should be greater than or equal to 0.
4. $Slack = RT - AT$

$$\text{Setup Slack} = (T_{clk} - t_{su} - t_{buf}) - (t_{pdff1} + t_{combo})$$

5. If we equate the slack to zero, then we get

$$0 = (T_{clk} - t_{su} - t_{buf}) - (t_{pdff1} + t_{combo})$$

$$T_{clk} = t_{pdff1} + t_{combo} + t_{su} + t_{buf}$$

6. The maximum operating frequency of the design is f_{\max}

$$f_{\max} = \frac{1}{T_{clk}}$$

$$= \frac{1}{t_{pdff1} + t_{combo} + t_{su} + t_{buf}}$$

11.9 Hold Slack

In the previous section we have discussed about the setup slack, and it should be positive or zero; then we can say that design does not have setup violation. Now let us discuss about hold slack!

The hold slack for the design is the difference between the data arrival time and data required time.

Consider the design shown in Fig. 11.20.

We can find out hold slack using following steps.

1. Let us find out the data arrival time (AT). $AT = t_{pdff1}$

Fig. 11.20 Synchronous design

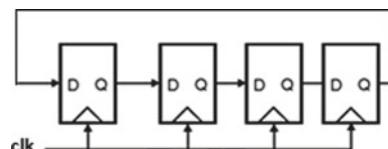
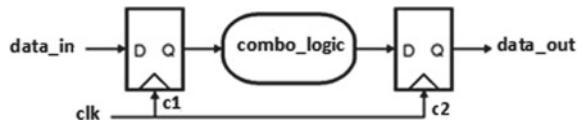


Fig. 11.21 Synchronous design (reg-to-reg path)



2. The data required time is RT. $RT = t_h$ because the data at D input should be stable by t_h margin after arrival of rising edge of the clock.
3. Now find hold slack. $Slack = AT - RT$ and should be greater than or equal to 0.
4. Hold Slack = $AT - RT$

$$\text{Hold Slack} = t_{\text{pdff1}} - t_h$$

5. To avoid the hold violations in the design the hold slack should be greater than or equal to zero.

11.9.1 VLSI Specific Design Scenario III

Consider the design shown in Fig. 11.21.

We can find out hold slack using following steps.

1. Let us find out the data arrival time (AT). $AT = t_{\text{pdff1}} + t_{\text{combo}}$
2. The data required time is RT. $RT = t_h$ because the data at D input should be stable by t_h margin after arrival of rising edge of the clock.
3. Now find hold slack. $Slack = AT - RT$ and should be greater than or equal to 0.
4. Hold Slack = $AT - RT$

$$\text{Hold Slack} = (t_{\text{pdff1}} + t_{\text{combo}}) - t_h$$

5. To avoid the hold violations in the design the hold slack should be greater than or equal to zero. This indicates that the $(t_{\text{pdff1}} + t_{\text{combo}})$ should be greater than or equal to t_h .

In this chapter we have discussed the timing parameters, timing paths, and the maximum frequency calculations for the design from VLSI perspective. In the next chapter let us design the FSM using lesser area for maximum speed.

11.10 Summary

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

1. The propagation delay of any combinational logic element is the amount of time required for the element to get the valid output after change in the input.

2. So, the goal and objective of the logic design engineer are to have the design which has lesser area that is lower gate count, maximum speed, and low power. So, we consider the optimization constraints as area, speed, and power.
3. It is recommended to avoid the cascade stages in the design as they add significant amount of the delay.
4. In the parallel logic the overall delay is minimized due to concurrent operations.
5. The minimum amount of time for which the data input of the flip-flop should maintain the stable value prior to arrival of the active edge of the clock is called as setup time.
6. The minimum amount of time for which the data input of the flip-flop should maintain the stable value after the arrival of the active edge of the clock is called as hold time.
7. The amount of time required for the flip-flop to get the valid data at output after the arrival of the active edge of the clock is called as propagation delay of flip-flop.
8. The propagation delay is also called as the **clock to q delay**, and it is also referred as t_{ctoq} .
9. For any sequential 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).
10. If the source or launch flip-flop is triggered first and the capture or destination flip-flop is triggered last, then the design has positive clock skew.
11. If the source or launch flip-flop is triggered last and the capture or destination flip-flop is triggered first, then the design has negative clock skew.
12. The setup slack for the design is the difference between the data required time and data arrival time.
13. The hold slack for the design is the difference between the data arrival time and data required time.
14. Practically the design experiences timing violation due to inefficient clock trees and clock buffer delays.
15. Design meets timing requirements which means both the setup and hold slack values are positive or equal to zero.

Chapter 12

FSM Designs



The FSM understanding and their application to design controller plays important role in design.

The finite state machines (FSMs) are used to design the arbitrary counters, sequence detectors. There are various FSM encoding techniques, and depending on the area, speed, and power these encoding techniques are used during the design of FSM. In the VLSI context the chapter is useful to discuss the various FSM encoding methods for Moore and Mealy FSMs.

12.1 Introduction to FSM

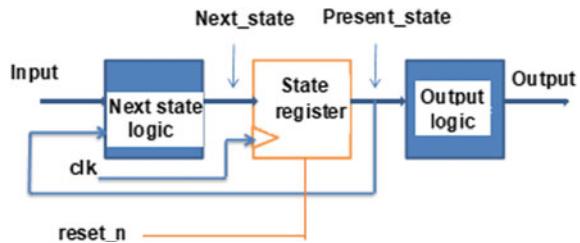
As we know that FSM is finite state machine and is used to design the FSM-based controllers, arbitrary counters, and sequence detectors. The objectives of the logic designers are to design the FSMs to have better data and control path logic. Even we need to consider the area, speed, and power optimizations of the FSM designs and FSM-based controllers.

The FSMs are classified mainly as

1. Moore FSM
2. Mealy FSM

So now in this section let us discuss the basic concepts of Moore and Mealy machines and their use in the design.

Fig. 12.1 Block diagram of Moore machine



12.1.1 Moore FSM

In the Moore FSM designs, an output is function of the present state and output is stable for one clock cycle. The state transition happens on the active edge of the clock.

The Moore FSM has three functional blocks:

1. State register
2. Next state logic
3. Output logic.

The state register is sequential logic. The next state and output logic blocks are of type combinational logic.

The block diagram of the Moore FSM is shown in Fig. 12.1.

As shown in Fig. 12.1, the Moore machine has three blocks and our goal is to design the digital logic to have minimum area, maximum speed, and minimum power dissipation.

12.1.2 Mealy FSM

In the Mealy FSM, an output is function of the present state and the inputs hence output may or may not be stable for one clock cycle. The state transition happens on the active edge of the clock. As compared to the Moore FSM the Mealy FSM is prone to glitches but uses a lesser number of states.

The Mealy FSM has three functional blocks:

1. State register
2. Next state logic
3. Output logic.

The state register is sequential logic. The next state and output logic are of combinational logic type.

The block diagram of the Mealy FSM is shown in Fig. 12.2.

As shown in Fig. 12.2, the Mealy machine has three blocks and our goal is to design the digital logic of these blocks to have maximum performance.

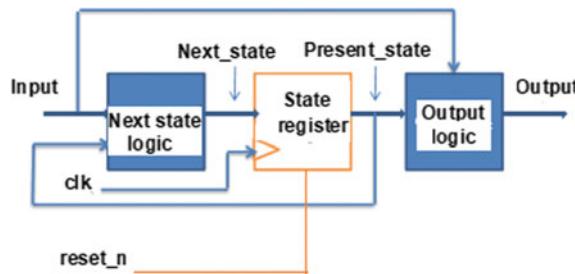


Fig. 12.2 Block diagram of Mealy machine

12.1.3 Moore Versus Mealy FSM

The main differences between the Moore and Mealy FSM are documented in Table 12.1.

12.2 State Encoding Methods

Our goal is to design the FSM for lesser area, maximum speed, and lesser power; hence for the FSM designs, we should understand the various state encoding methods. There are three types of the state encoding for FSM-based designs, and they are

1. Binary encoding

In the binary encoding if the number of states are m then the number of flip-flops required is computed using $n = \log_2 m$.

Where n = number of flip-flops and m are number of states.

Consider the number of states as $m = 8$ then number of flip-flops needed to design FSM are $n = \log_2 8 = 3$.

Table 12.1 Differences between Moore and Mealy machines

Moore machine	Mealy machine
Outputs are function of present state only	Outputs are function of the present state and inputs also
Output is stable for one clock cycle, and Moore FSMs are not prone to glitches or spikes	Output may change multiple times depending on changes in the input and may or may not be stable for one clock cycle; hence, Mealy FSMs are prone to glitches or hazards
It requires a greater number of states as compared to Mealy machine	Mealy machine needs lesser states as compared to Moore machine
Moore FSM has the higher operating frequency as compared to Mealy machine	Mealy FSM has the lower operating frequency as compared to Moore machine

The eight states are represented as

$$S_0 = 000$$

$$S_1 = 001$$

$$S_2 = 010$$

$$S_3 = 011$$

$$S_4 = 100$$

$$S_5 = 101$$

$$S_6 = 110$$

$$S_7 = 111$$

2. Gray encoding

In the gray encoding if the number of states are m , then the number of flip-flops required is computed using $n = \log_2 m$.

Where n = number of flip-flops and m are number of states.

Consider the number of states as $m = 8$ then number of flip-flops needed to design FSM are $n = \log_2 8 = 3$.

The eight states are represented as

$$S_0 = 000$$

$$S_1 = 001$$

$$S_2 = 011$$

$$S_3 = 010$$

$$S_4 = 110$$

$$S_5 = 111$$

$$S_6 = 101$$

$$S_7 = 100$$

The gray encoding is useful in the FSM design to save the power as in two successive gray numbers only one bit changes.

3. One-hot encoding

In the one-hot encoding if the number of states are m then the number of flip-flops required is computed using $n = m$. In this encoding only one bit is active high or hot at a time.

Where n = number of flip-flops and m are number of states.

Consider the number of states as $m = 8$ then number of flip-flops needed to design FSM are $m = n = 8$.

The eight states using the one-hot encoding techniques are represented as

$$S_0 = 00000001$$

$$S_1 = 00000010$$

$$S_2 = 00000100$$

$$S_3 = 00001000$$

$$S_4 = 00010000$$

$$S_5 = 00100000$$

$$S_6 = 01000000$$

$$S_7 = 10000000$$

The one-hot encoding is useful in the FSM design to have better timing if area is not the constraint.

12.3 Moore FSM Design

Now let us design the Moore FSM for the given specifications. What we need to do is that we need to design the digital logic to have better performance for the

1. State register
2. Next state logic
3. Output logic

Let us consider the design of the sequential circuit to get the output `data_out` which is input clock frequency divided by 8 when `data_in` = 1. We can use the following design steps to design the Moore FSM.

1. Find the number of states to get the divide by 8 output

Number of states = 8. The states are $s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7$.

2. State Transition

The state transition happens depending on `Data_in` logic values on the active edge of the clock.

3. Find number of flip-flops

We will use the binary encoding and the number of flip-flops = $n = \log_2 8 = 3$. We will use the positive edge sensitive D flip-flops.

4. Reset strategy

Let us use active low asynchronous reset input `reset_n`. For `reset_n` = 0 the counter holds the previous output. For the `reset_n` = 1 the output increments on the rising edge of the clock.

5. Let us document the entries in the state table to get state register logic

The state table of the MOD-8 synchronous counter is shown in Table 12.2.

To have the MOD-4 binary up-counter we need to have two D flip-flops having asynchronous active low reset input `reset_n` and active high `data_in`.

6. Use of the excitation table to design the next state logic

The excitation table consists of the information about the present state, next state, and excitation input (Table 12.3).

Now let us use the K -map to deduce the Boolean equation for the next state logic to get the logic at $D2, D1, D0$.

1. Next state logic for $D2$

Now let us deduce the expression for the next state logic for input $D2$ (Fig. 12.3).

Table 12.2 State table of the MOD-8 binary up-counter

Enable (data_in)	Present state ($Q_2\ Q_0\ Q_1$)	Next state ($Q_2^+\ Q_1^+\ Q_0^+$)
0	s_0	s_0
1	s_0	s_1
0	s_1	s_0
1	s_1	s_2
0	s_2	s_0
1	s_2	s_3
0	s_3	s_0
1	s_3	s_4
0	s_4	s_0
1	s_4	s_5
0	s_5	s_0
1	s_5	s_6
0	s_6	s_0
1	s_6	s_7
0	s_7	s_0
1	s_7	s_0

Table 12.3 Excitation table of the MOD-8 counter

Enable (data_in)	Present state ($Q_2\ Q_0\ Q_1$)	Next state ($Q_2^+\ Q_1^+\ Q_0^+$)	Excitation input ($D_2\ D_1\ D_0$)
0	$s_0 = 000$	$s_0 = 000$	000
1	$s_0 = 000$	$s_1 = 001$	001
0	$s_1 = 001$	$s_0 = 000$	000
1	$s_1 = 001$	$s_2 = 010$	010
0	$s_2 = 010$	$s_0 = 000$	000
1	$s_2 = 010$	$s_3 = 011$	011
0	$s_3 = 011$	$s_0 = 000$	000
1	$s_3 = 011$	$s_4 = 100$	100
0	$s_4 = 100$	$s_0 = 000$	000
1	$s_4 = 100$	$s_5 = 101$	101
0	$s_5 = 101$	$s_0 = 000$	000
1	$s_5 = 101$	$s_6 = 110$	110
0	$s_6 = 110$	$s_0 = 000$	000
1	$s_6 = 110$	$s_7 = 111$	111
0	$s_7 = 111$	$s_0 = 000$	000
1	$s_7 = 111$	$s_0 = 000$	000

Fig. 12.3 Next state logic for $D2$

		Q1 Q0	00	01	11	10
		00	0	0	0	0
		01	0	0	1	0
		11	1	1	0	1
		10	0	0	0	0

$$\begin{aligned} Q2^+ = D2 &= \text{Data_in}.Q2.\overline{Q1} + \text{Data_in}.Q2.\overline{Q0} \\ &\quad + \overline{\text{Data_in}}.Q2.Q1.Q0 \end{aligned}$$

2. Next state logic for $D1$

Now let us deduce the expression for the next state logic for input $D1$ (Fig. 12.4).

$$Q1^+ = D1 = \text{Data_in}.(Q1.\overline{Q0} + \overline{Q1}.Q0.)$$

$$Q1^+ = D1 = \text{Data_in}.(Q1 \oplus Q0.)$$

3. Next state logic for $D0$

Now let us deduce the expression for the next state logic for input $D0$ (Fig. 12.5).

$$Q0^+ = D0 = \text{Data_in}.(\overline{Q1} \oplus \overline{Q0})$$

7. Output logic

Fig. 12.4 Next state logic for $D1$

		Q1 Q0	00	01	11	10
		00	0	0	0	0
		01	0	0	0	0
		11	0	1	0	1
		10	0	1	0	1

Fig. 12.5 K-map for the next state logic at D0

		Q1 Q0	00	01	11	10
		D0	00	01	11	10
Data_in Q2			00	01	11	10
	00	0	0	0	0	0
01	0	0	0	0	0	0
11	0	1	0	0	1	1
10	1	0	0	0	1	1

In the Moore FSM output is function of the present state only. Let us deduce the combinational logic at output (Table 12.4 and Fig. 12.6).

Let us deduce the logic expression for the output combinational logic

$$\text{Data_out} = Q2.Q1.Q0$$

8. Let us sketch the FSM Design

Table 12.4 Truth table of the Moore FSM output logic

Enable (data_in)	Present state ($Q2\ Q0\ Q1$)	Data_out
0	$s0 = 000$	0
1	$s0 = 000$	0
0	$s1 = 001$	0
1	$s1 = 001$	0
0	$s2 = 010$	0
1	$s2 = 010$	0
0	$s3 = 011$	0
1	$s3 = 011$	0
0	$s4 = 100$	0
1	$s4 = 100$	0
0	$s5 = 101$	0
1	$s5 = 101$	0
0	$s6 = 110$	0
1	$s6 = 110$	0
0	$s7 = 111$	1
1	$s7 = 111$	1

Fig. 12.6 K-map for the output logic of Moore machine

		Q1 Q0			
		00	01	11	10
Data_in Q2	00	0	0	0	0
	01	0	0	1	0
11	0	0	1	0	
10	0	0	0	0	

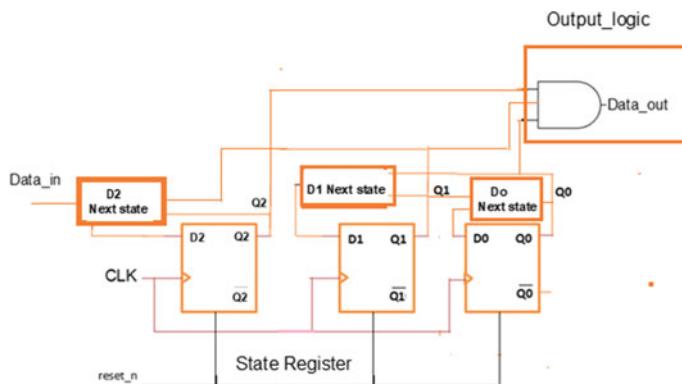


Fig. 12.7 MOD-8 synchronous up-counter using FSM design approach

As discussed in the previous steps we need to have two flip-flops and logic gates to implement the MOD-4 synchronous up-counter which is divide by 4. For four clock cycles output is single clock pulse. For MOD-8 counter we should have 3 flip-flops and output is divide by 8. As shown in Fig. 12.7 the state register uses $\text{Data_in} = 1$ as active high enable. During the $\text{Data_in} = 0$ the output of the register is same as the default state s_0 .

As shown in Fig. 12.7 the next state logic gates are denoted using elliptical boxes.

12.4 Mealy FSM Design

Now let us design the Mealy FSM for the given specifications. What we need to do is that we need to design the digital logic to have better performance for the

1. State register
2. Next state logic

3. Output logic

Let us consider the design of the sequential circuit to get the output Data_out which is input clock frequency divided by 4 when Data_in = 1. We can use the following design steps to design the Mealy FSM.

1. Find the number of states to get the divide by 4 output

Number of states = 4. The states are s_0, s_1, s_2, s_3 .

2. State Transition

The state transition happens depending on Data_in logic values on the active edge of the clock.

3. Find number of flip-flops

We will use the binary encoding and the number of flip-flops = $n = \log_2 4 = 2$.
We will use the positive edge sensitive D flip-flops.

4. Reset strategy

Let us use active low asynchronous reset input reset_n. For reset_n = 0 the counter holds the previous output. For the reset_n = 1 the output increments on the rising edge of the clock.

5. Let us document the entries in the state table to get state register logic

The state table of the MOD-4 synchronous counter is shown in Table 12.5.

To have the MOD-4 binary up-counter we need to have two D flip-flops having asynchronous active low reset input reset_n and active high data_in.

6. Use of the excitation table to design the next state logic

The excitation table consists of the information about the present state, next state, and excitation input (Table 12.6).

Now let us use the K-map to deduce the Boolean equation for the next state logic to get the logic at $D1, D0$.

Table 12.5 State table of the MOD-4 binary up-counter

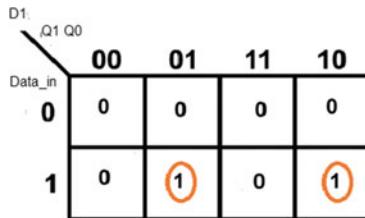
Enable (data_in)	Present state ($Q0\ Q1$)	Next state ($Q0^+ Q1^+$)
0	s_0	s_0
1	s_0	s_1
0	s_1	s_0
1	s_1	s_2
0	s_2	s_0
1	s_2	s_3
0	s_3	s_0
1	s_3	s_0

Table 12.6 Excitation table of the MOD-4 counter

Enable (data_in)	Present state (Q0 Q1)	Next state (Q0 ⁺ Q1 ⁺)	Excitation input (D1 D0)
0	s0 = 00	s0 = 00	00
1	s0 = 00	s1 = 01	01
0	s1 = 01	s0 = 00	00
1	s1 = 01	s2 = 10	10
0	s2 = 10	s0 = 00	00
1	s2 = 10	s3 = 11	11
0	s3 = 11	s0 = 00	00
1	s3 = 11	s0 = 00	00

1. Next state logic for D1

Now let us deduce the expression for the next state logic for input D1.



$$Q1^+ = D1 = \text{Data_in}.(Q1.\overline{Q0} + \overline{Q1}.Q0.)$$

$$Q1^+ = D1 = \text{Data_in}.(Q1 \oplus Q0.)$$

2. Next state logic for D0

Now let us deduce the expression for the next state logic for input D0 (Fig. 12.8).

$$Q0^+ = D0 = \text{Data_in}.\overline{Q0}$$

7. Output Logic

In the Mealy FSM output is function of the present state only. Let us deduce the combinational logic at output (Table 12.7 and Fig. 12.9).

Let us deduce the logic expression for the output combinational logic

$$\text{Data}_{\text{out}} = \text{Data_in}.Q1.Q0$$

Fig. 12.8 K-map for the next stage logic

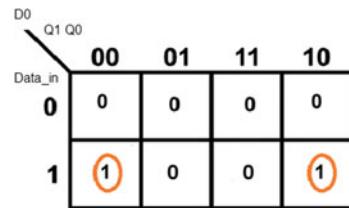
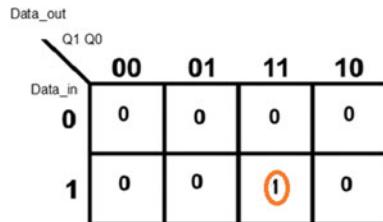


Table 12.7 Truth table of the output logic

Enable (data_in)	Present state ($Q1\ Q0$)	Output (data_out)
0	$s0 = 00$	0
1	$s0 = 00$	0
0	$s1 = 01$	0
1	$s1 = 01$	0
0	$s2 = 10$	0
1	$s2 = 10$	0
0	$s3 = 11$	0
1	$s3 = 11$	1

Fig. 12.9 K-map for the output logic



8. Let Us Sketch the FSM Design

As discussed in the previous steps we need to have two flip-flops and logic gates to implement the MOD-4 synchronous up-counter which is divided by 4. For four clock cycles output is single clock pulse. In the Mealy FSM an output Data_out is function of the present state and input Data_in. As shown in Fig. 12.10 the state register uses Data_in = 1 as active high enable. During the Data_in = 0 the output of the register is same as the default state $s0$ (Fig. 12.10).

As shown in Fig. 12.7 the next state logic gates are denoted using elliptical boxes.

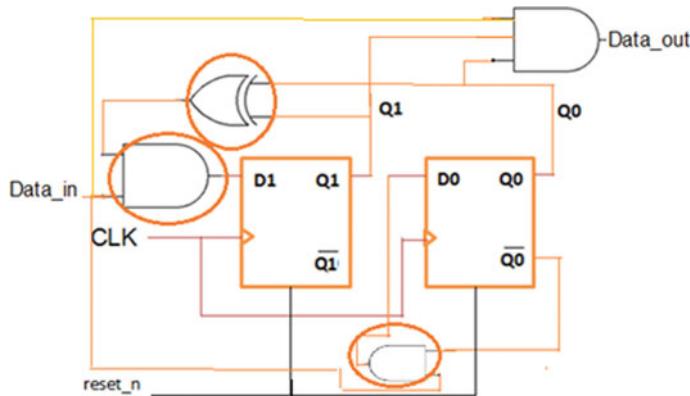


Fig. 12.10 MOD-4 synchronous up-counter using FSM design approach

12.5 Applications and Design Strategies

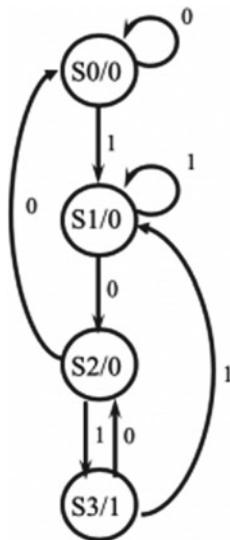
The FSM designs are useful to design the sequential logic with the objective to have better data and control path optimization. For more details about the data and control paths refer Chap. 9. Following are few of the applications and the strategies while designing the FSM-based designs

1. FSM-based design approach with better partitioning for the data and control path is useful to design FSM-based controllers.
2. FSMs are extensively useful to design the large density counting circuit with better partitioning and better area, timing.
3. FSMs are used to detect the sequence from the input string.
4. The objective of the logic designer is to design the glitch free FSMs.
5. The FSM-based controllers should have separate data and control path for better area and speed.
6. While designing the FSM to optimize for the area try to use the strategy to eliminate unwanted states.
7. Use the gray encoding for the power optimization.
8. If the area is not a constraint then for the better timing use the one-hot encoding.

12.6 State Diagrams

Now let us focus on the state diagrams of the Moore and Mealy FSM. We can use the state diagrams during the design to understand about the transitions on inputs and to understand about the outputs.

Fig. 12.11 Moore state diagram of sequence 101



12.6.1 Moore Machine State Diagram

Let us sketch the Moore state diagram to detect the 101-overlapping sequence.

To detect the sequence 101 use the understanding of the Moore machine. Output is function of the present state only and output is 1 when the sequence 101 is detected. We need to use 4 states for the binary encoding. State $s_0 = 00$, $s_1 = 01$, $s_2 = 10$, and $s_3 = 11$.

Consider the default state is S_0 and output is 0. So, the state transition should be

1. If input is 1 then S_0 to S_1 . Be there in state S_0 for input is equal to 0.
2. If next input bit is 0, then S_1 – S_2 . Be there in state S_1 when input is 1.
3. If next input bit is 1, then S_2 – S_3 and output is 1. If input is 0 then S_2 – S_0 .
4. To detect the overlapping sequence 101, if again next input is 0 then S_3 – S_2 . If input is 1, then state transition from S_3 – S_1 .

The state diagram is shown in Fig. 12.11.

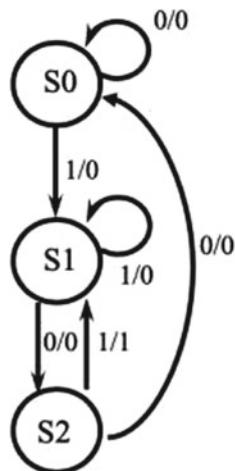
12.6.2 Mealy Machine State Diagram

Let us sketch the Mealy state diagram to detect the 101-overlapping sequence.

To detect the sequence 101 use the understanding of the Mealy machine. Output is function of the present state and input also and output is 1 when the sequence 101 is detected. We need to use 3 states for the binary encoding. State $s_0 = 00$, $s_1 = 01$, $s_2 = 10$.

Consider the default state is S_0 and output is 0. So, the state transition should be

Fig. 12.12 Mealy state diagram of sequence 101



1. If input is 1, then $S_0 \rightarrow S_1$. Be there in state S_0 for input is equal to 0.
2. If next input bit is 0, then $S_1 \rightarrow S_2$. Be there in state S_1 when input is 1.
3. To detect the overlapping sequence 101, if next input bit is 1 then $S_2 \rightarrow S_1$ and output is 1. If input is 0, then $S_2 \rightarrow S_0$.

The state diagram is shown in Fig. 12.12.

12.7 Summary

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

1. FSM are finite state Machines and classified as Moore and Mealy FSM.
2. In the Moore FSM, an output is function of the present state.
3. In the Mealy FSM, an output is function of the present state and inputs.
4. Moore FSM needs a greater number of states as compared to Mealy FSM.
5. The FSM-based design approach is useful to design the sequence detectors and FSM based controllers.
6. The FSM can use one of the state encoding method, binary, gray, or one-hot.
7. Consider the number of states as $m = 8$ then number of flip-flops needed to design binary, gray encoding FSM are $n = \log_2 8 = 3$.
8. Consider the number of states as $m = 8$ then number of flip-flops needed to design one-hot encoding FSM are 8.
9. Using the gray encoding the power can be optimized.
10. Using the one-hot encoding method the timing can be improved but it increases the area.

Chapter 13

Design of Sequence Detectors



The sequence detector design techniques are useful to design the FSM based controller and timing and control units.

In the previous chapter we have discussed about the FSM design basics and various encoding methods. In this chapter let us design the sequence detectors to have minimum area, maximum speed, and minimum power. From the VLSI perspective the chapter is useful to understand about the state diagrams of Moore and Mealy sequence detectors and the design of the sequence detectors.

13.1 Moore Machine Non-overlapping 101 Sequence Detector

Let us sketch the Moore state diagram to detect the 101 non-overlapping sequence.

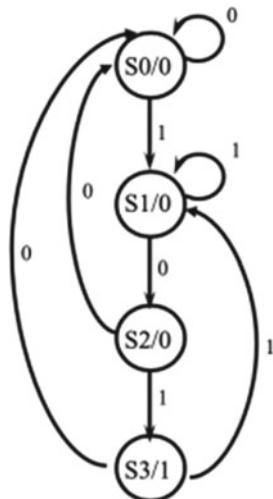
To detect the sequence 101, use the understanding of the Moore machine. Output is function of the present state only, and output is 1 when the sequence 101 is detected. We need to use 4 states for the binary encoding. State $S_0 = 00$, $s_1 = 01$, $s_2 = 10$, and $S_3 = 11$.

Consider the default state is S_0 and output is 0. So, the state transition should be

1. If input is 1, then $S_0 \rightarrow S_1$. Be there in state S_0 for input is equal to 0.
2. If next input bit is 0, then $S_1 \rightarrow S_2$. Be there in state S_1 when input is 1.
3. If next input bit is 1, then $S_2 \rightarrow S_3$ and output is 1. If input is 0, then $S_2 \rightarrow S_0$.
4. To detect the non-overlapping sequence 101, if again next input is 0, then $S_3 \rightarrow S_0$. If input is 1, then state transition from S_3 to S_1 .

The state diagram is shown in Fig. 13.1

Fig. 13.1 Moore state diagram to detect the non-overlapping 101 sequence



13.2 Mealy Machine Non-overlapping 101 Sequence Detector

Let us sketch the Mealy state diagram to detect the 101 non-overlapping sequence.

To detect the sequence 101, use the understanding of the Mealy machine. Output is function of the present state and input also and output is 1 when the sequence 101 is detected. We need to use 3 states for the binary encoding. State $S0 = 00$, $s1 = 01$, and $s2 = 10$.

Consider the default state is $S0$ and output is 0. So, the state transition should be

1. If input is 1, then $S0 \rightarrow S1$ and output is 0. Be there in state $S0$ for input is equal to 0 and output is 0.
2. If next input bit is 0, then $S1 \rightarrow S2$ and output is 0. Be there in state $S1$ when input is 1 and output is zero.

To detect the non-overlapping sequence 101, if next input bit is 1, then $S2 \rightarrow S0$ and output is 1. If input is 0, then $S2 \rightarrow S0$ and output is 0.

The state diagram is shown in Fig. 13.2.

13.3 One-Hot Encoding

For the following state diagram to detect the 101 overlapping sequence document the states using one-hot encoding let us find the number of flip-flops needed to implement the one-hot encoding state machine (Fig. 13.3).

Fig. 13.2 Mealy state diagram to detect the 101 non-overlapping sequence

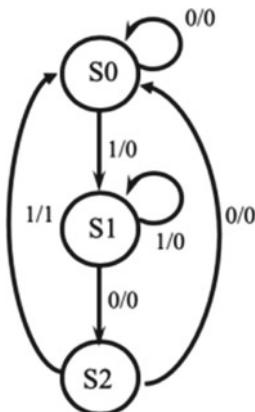
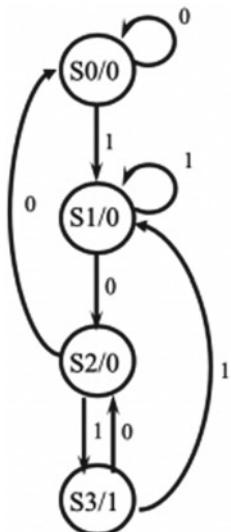


Fig. 13.3 Moore state diagram of sequence 101



For the state machine design using one-hot encoding the number of flip-flops is equal to number of states. For the given Moore sequence detector, the number of flip-flops = 4.

We can represent the states using one-hot encoding as

$$S0 = 0001$$

$$S1 = 0010$$

$$S2 = 0100$$

$$S3 = 1000$$

13.4 FSM Area and Power Optimization

For the following state diagram to detect the 101 overlapping sequence how we can think about the area and power optimization? (Fig. 13.4).

For the given Mealy sequence detector, the number of flip-flops = 2 if we use the binary or gray encoding. As there is sing bit change in the two consecutive gray codes use the gray encoding to optimize for the power.

We can represent the states using gray encoding as

$$S0 = 00$$

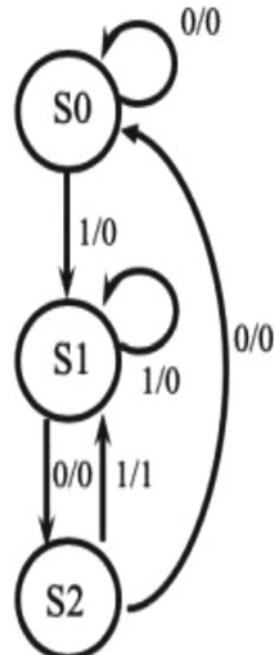
$$S1 = 01$$

$$S2 = 11$$

As compared to the one-hot encoding the gray encoding optimizes the area as number of flip-flops for gray encoding are \log_2 States.

If states are 4, then the number of flip-flops is 2. For three states we need two flip-flops.

Fig. 13.4 Mealy state diagram of sequence 101



13.5 Moore Sequence Detector for 101 Overlapping Sequence

Now let us design the Moore FSM for the given specifications. What we need to do is that we need to design the digital logic to have better performance for the

1. State register
2. Next state logic
3. Output logic.

Let us consider the design of the sequence detector to detect the overlapping sequence 101. We can use the following design steps to design the Moore FSM.

1. Find the number of states to detect the 101 overlapping sequence

Number of states = 4. The states are s_0, s_1, s_2, s_3 .

2. State Diagram

Let us sketch the Moore state diagram to detect the 101 overlapping sequence.

To detect the sequence 101 use the understanding of the Moore machine. Output is function of the present state only and output is 1 when the sequence 101 is detected. We need to use 4 states for the binary encoding. State $S_0 = 00$, $s_1 = 01$, $s_2 = 10$, and $s_3 = 11$.

Fig. 13.5 Moore state diagram of sequence 101

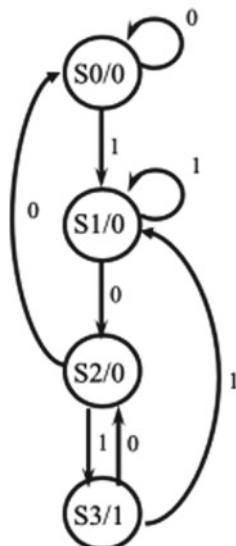


Table 13.1 The state table of the 101 overlapping Moore machine

Input (data_in)	Present state ($Q_1 Q_0$)	Next state ($Q_1^+ Q_0^+$)
0	s_0	s_0
1	s_0	s_1
0	s_1	s_2
1	s_1	s_1
0	s_2	s_0
1	s_2	s_3
0	s_3	s_2
1	s_3	s_1

Consider the default state is S_0 and output is 0. So, the state transition should be

1. If input is 1, then $S_0 \rightarrow S_1$. Be there in state S_0 for input is equal to 0.
2. If next input bit is 0, then $S_1 \rightarrow S_2$. Be there in state S_1 when input is 1.
3. If next input bit is 1, then $S_2 \rightarrow S_3$ and output is 1. If input is 0, then $S_2 \rightarrow S_0$.
4. To detect the overlapping sequence 101, if again next input is 0, then $S_3 \rightarrow S_2$. If input is 1, then state transition from S_3 to S_1 .

The state diagram is shown in Fig. 13.5.

3. Find number of flip-flops

We will use the binary encoding and the number of flip-flops = $n = \log_2 4 = 2$. We will use the positive edge sensitive D flip-flops.

4. Reset strategy

Let us use active low asynchronous reset input reset__n . For $\text{reset_}_n = 0$ the counter holds the previous output. For the $\text{reset_}_n = 1$ the output increments on the rising edge of the clock.

5. Let us document the entries in the state table to get state register logic

The state table of the 101 Moore overlapping sequence detector is shown in Table 13.1.

To have the 101 Moore overlapping sequence detector we need to have two D flip-flops having asynchronous active low reset input reset__n and active high data_in .

6. Use of the excitation table to design the next state logic

The excitation table consists of the information about the present state, next state, and excitation input (Table 13.2).

Table 13.2 The excitation table of the 101 Moore overlapping sequence detector

Enable (data_in)	Present state (Q1 Q0)	Next state (Q1 ⁺ Q0 ⁺)	Excitation input (D1 D0)
0	s0 = 00	s0 = 00	00
1	s0 = 00	s1 = 01	01
0	s1 = 01	s2 = 10	10
1	s1 = 01	s1 = 01	01
0	s2 = 10	s0 = 00	00
1	s2 = 10	s3 = 11	11
0	s3 = 11	s2 = 10	10
1	s3 = 11	s1 = 01	01

Now let us use the *K*-map to deduce the Boolean equation for the next state logic to get the logic at *D*1 and *D*0.

1. Next state logic for *D*1

Now let us deduce the expression for the next state logic for input *D*1 (Fig. 13.6).

$$Q1^+ = D1 = \text{Data_in} \cdot Q0$$

2. Next state logic for *D*0

Now let us deduce the expression for the next state logic for input *D*0 (Fig. 13.7).

$$Q0^+ = D0 = \text{Data_in}$$

Fig. 13.6 The *K*-map for next state logic *D*1

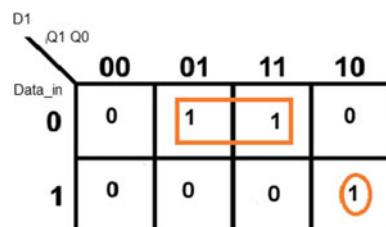


Fig. 13.7 The *K*-map for the next state logic

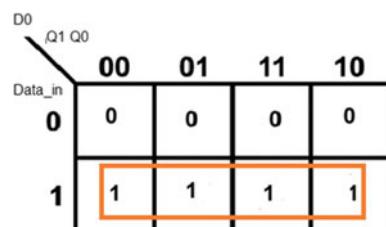
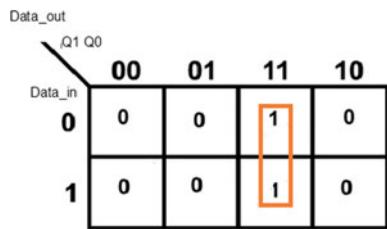


Table 13.3 The truth table of the output logic

Enable (data_in)	Present state ($Q_1 Q_0$)	Output (data_out)
0	$s_0 = 00$	0
1	$s_0 = 00$	0
0	$s_1 = 01$	0
1	$s_1 = 01$	0
0	$s_2 = 10$	0
1	$s_2 = 10$	0
0	$s_3 = 11$	1
1	$s_3 = 11$	1

Fig. 13.8 K-map for the output logic of Moore machine



7. Output logic

In the Moore FSM output is function of the present state only. Let us deduce the combinational logic at output (Table 13.3; Fig. 13.8).

Let us deduce the logic expression for the output combinational logic

$$\text{Data_out} = Q_1 \cdot Q_0$$

8. Let us sketch the FSM Design

As discussed in the previous steps we need to have two flip-flops and logic gates to implement the 101 overlapping Moore sequence detector. As shown in Fig. 13.9 the state register uses next state logic to get the next state. During the reset_n = 0 the output of the register is same as the default state s0.

As shown in Fig. 13.9 the next state logic gates are denoted using elliptical boxes.

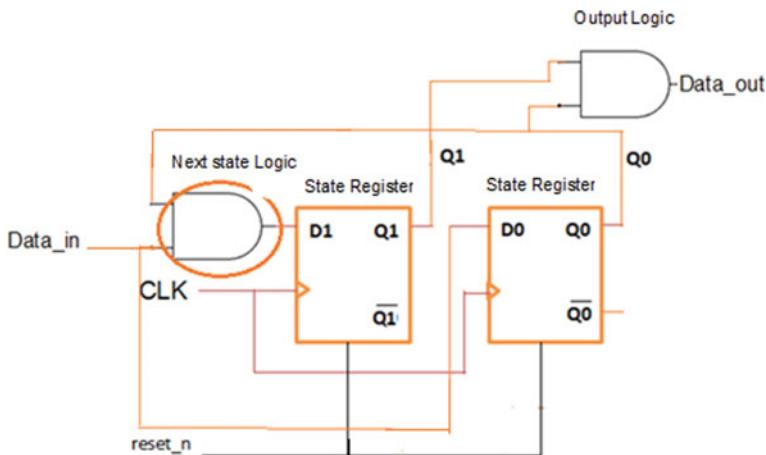


Fig. 13.9 The Moore 101 overlapping sequence detector

13.6 Mealy Sequence Detector for 101 Overlapping Sequence

Now let us design the Mealy FSM for the given specifications. What we need to do is that we need to design the digital logic to have better performance for the

1. State register
2. Next state logic
3. Output logic.

Let us consider the design of the sequence detector to detect the overlapping sequence 101. We can use the following design steps to design the Mealy FSM.

1. Find the number of states to detect the 101 overlapping sequence

Number of states = 3. The states are s_0 , s_1 , and s_2 .

2. State Diagram

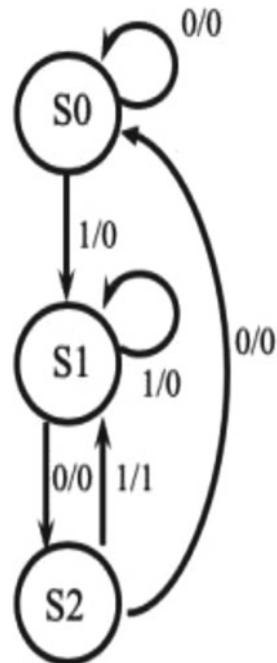
Let us sketch the Mealy state diagram to detect the 101 overlapping sequence.

To detect the sequence 101 use the understanding of the Mealy machine. Output is function of the present state and input also and output is 1 when the sequence 101 is detected. We need to use 3 states for the binary encoding. State $S_0 = 00$, $s_1 = 01$, and $s_2 = 10$.

Consider the default state is S_0 and output is 0. So, the state transition should be

1. If input is 1, then $S_0 \rightarrow S_1$. Be there in state S_0 for input is equal to 0.
2. If next input bit is 0, then $S_1 \rightarrow S_2$. Be there in state S_1 when input is 1.
3. To detect the overlapping sequence 101, if next input bit is 1, then $S_2 \rightarrow S_1$ and output is 1. If input is 0, then $S_2 \rightarrow S_0$.

Fig. 13.10 Mealy state diagram of sequence 101



The state diagram is shown in Fig. 13.10.

3. Find number of flip-flops

We will use the binary encoding and the number of flip-flops = $n = \log_2 4 = 2$. We will use the positive edge sensitive D flip-flops.

4. Reset strategy

Let us use active low asynchronous reset input reset__n . For $\text{reset_}_n = 0$ the counter holds the previous output. For the $\text{reset_}_n = 1$ the output increments on the rising edge of the clock.

5. Let us document the entries in the state table to get state register logic

The state table of the 101 Mealy overlapping sequence detector is shown in Table 13.4.

To have the 101 Mealy overlapping sequence detector we need to have two D flip-flops having asynchronous active low reset input reset__n and active high data_in.

6. Use of the excitation table to design the next state logic

The excitation table consists of the information about the present state, next state, and excitation input (Table 13.5).

Table 13.4 The state table of the Mealy 101 overlapping sequence detector

Input (data_in)	Present state ($Q1\ Q0$)	Next state ($Q1^+Q0^+$)
0	$s0$	$s0$
1	$s0$	$s1$
0	$s1$	$s2$
1	$s1$	$s1$
0	$s2$	$s0$
1	$s2$	$s1$

Table 13.5 The excitation table of the 101 Mealy overlapping sequence detector

Enable (data_in)	Present state ($Q1\ Q0$)	Next state ($Q1^+Q0^+$)	Excitation input ($D1\ D0$)
0	$s0 = 00$	$s0 = 00$	00
1	$s0 = 00$	$s1 = 01$	01
0	$s1 = 01$	$s2 = 10$	10
1	$s1 = 01$	$s1 = 01$	01
0	$s2 = 10$	$s0 = 00$	00
1	$s2 = 10$	$s1 = 10$	10

Now let us use the K -map to deduce the Boolean equation for the next state logic to get the logic at $D1$ and $D0$.

1. Next state logic for $D1$

Now let us deduce the expression for the next state logic for input $D1$ (Fig. 13.11).

$$Q1^+ = D1 = \overline{\text{Data_in}} \cdot Q0$$

2. Next state logic for $D0$

Now let us deduce the expression for the next state logic for input $D0$ (Fig. 13.12).

$$Q0^+ = D0 = \text{Data_in}$$

Fig. 13.11 The K -map for the next state logic at $D1$

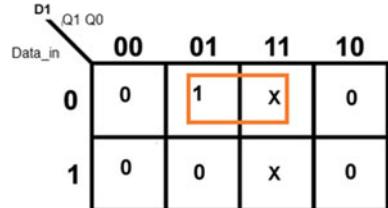


Fig. 13.12 The K-map for the next state logic

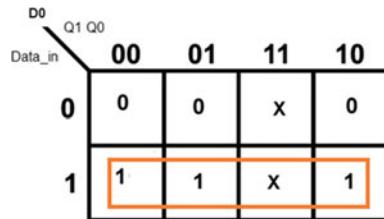


Fig. 13.13 K-map for the output logic of Mealy machine

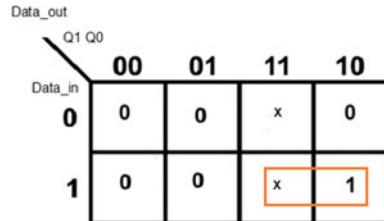


Table 13.6 The truth table of the output logic

Enable (data_in)	Present state ($Q_1 Q_0$)	Output (data_out)
0	$s_0 = 00$	0
1	$s_0 = 00$	0
0	$s_1 = 01$	0
1	$s_1 = 01$	0
0	$s_2 = 10$	0
1	$s_2 = 10$	1

7. Output logic

In the Mealy FSM output is function of the present state and inputs. Let us deduce the combinational logic at output (Fig. 13.13; Table 13.6).

Let us deduce the logic expression for the output combinational logic

$$\text{Data_out} = \text{Data_in} \cdot Q_1$$

8. Let us sketch the FSM Design

As discussed in the previous steps we need to have two flip-flops and logic gates to implement the 101 overlapping Mealy sequence detector. As shown in Fig. 13.9 the state register uses next state logic to get the next state. During the reset_n = 0 the output of the register is same as the default state s0.

As shown in Fig. 13.14 next state logic gates are denoted using elliptical boxes.

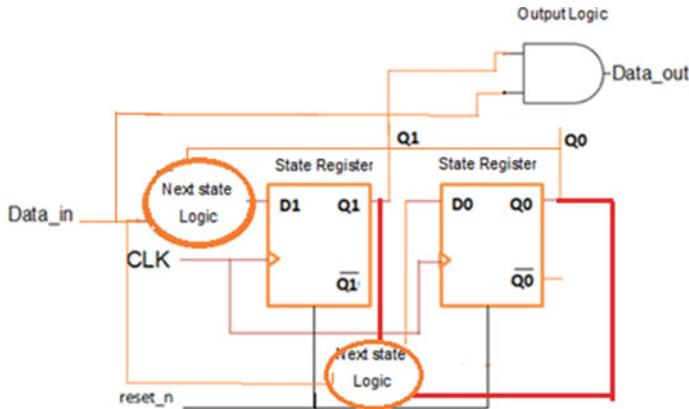


Fig. 13.14 The Mealy 101 overlapping sequence detector

13.7 Mealy Sequence Detector for 1010 Overlapping Sequence

Now let us design the Mealy FSM for the given specifications. What we need to do is that we need to design the digital logic to have better performance for the

1. State register
2. Next state logic
3. Output logic.

Let us consider the design of the sequence detector to detect the overlapping sequence 1010. We can use the following design steps to design the Mealy FSM.

1. Find the number of states to detect the 1010 overlapping sequence

Number of states = 4. The states are s_0 , s_1 , s_2 , and s_3 .

2. State Diagram

Let us sketch the Mealy state diagram to detect the 1010 overlapping sequence.

To detect the sequence 1010 use the understanding of the Mealy machine. Output is function of the present state and input also and output is 1 when the sequence 1010 is detected. We need to use 4 states for the binary encoding. State $S_0 = 00$, $s_1 = 01$, $s_2 = 10$, and $s_3 = 11$.

Consider the default state is S_0 and output is 0. So, the state transition should be

1. If input is 1, then $S_0 \rightarrow S_1$. Be there in state S_0 for input is equal to 0.
2. If next input bit is 0, then $S_1 \rightarrow S_2$. Be there in state S_1 when input is 1.
3. If next input bit is 1, then $S_2 \rightarrow S_3$ and for input 0 transition to state S_0 .
4. To detect the overlapping sequence 1010, if next input bit is 0, then $S_3 \rightarrow S_2$ and output is 1. If input is 1, then $S_3 \rightarrow S_1$.

The state diagram is shown in Fig. 13.15.

1. Find number of flip-flops

We will use the binary encoding and the number of flip-flops = $n = \log_2 4 = 2$. We will use the positive edge sensitive D flip-flops.

2. Reset strategy

Let us use active low asynchronous reset input reset__n . For $\text{reset_}_n = 0$ the counter holds the previous output. For the $\text{reset_}_n = 1$ the output increments on the rising edge of the clock.

3. Let us document the entries in the state table to get state register logic

The state table of the 1010 Mealy overlapping sequence detector is shown in Table 13.7.

To have the 1010 Mealy overlapping sequence detector we need to have two D flip-flops having asynchronous active low reset input reset__n and active high data_in.

4. Use of the excitation table to design the next state logic

The excitation table consists of the information about the present state, next state, and excitation input (Table 13.8).

Now let us use the K -map to deduce the Boolean equation for the next state logic to get the logic at $D1$ and $D0$.

Fig. 13.15 Mealy state diagram of sequence 1010

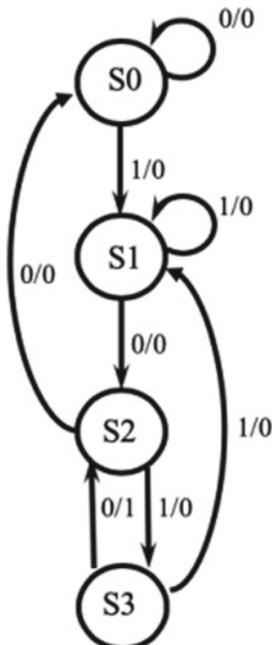


Table 13.7 The state table of the Mealy 1010 overlapping sequence detector

Input (data_in)	Present state ($Q1\ Q0$)	Next state ($Q1^+Q0^+$)
0	$s0$	$s0$
1	$s0$	$s1$
0	$s1$	$s2$
1	$s1$	$s1$
0	$s2$	$s0$
1	$s2$	$s3$
0	$s3$	$s2$
1	$s3$	$s1$

Table 13.8 The excitation table of the 1010 Mealy overlapping sequence detector

Enable (data_in)	Present state ($Q1\ Q0$)	Next state ($Q1^+Q0^+$)	Excitation input ($D1\ D0$)
0	$s0$	$s0$	00
1	$s0$	$s1$	01
0	$s1$	$s2$	10
1	$s1$	$s1$	01
0	$s2$	$s0$	00
1	$s2$	$s3$	11
0	$s3$	$s2$	10
1	$s3$	$s1$	01

5. Next state logic for D1

Now let us deduce the expression for the next state logic for input $D1$ (Fig. 13.16).

$$Q1^+ = D1 = \overline{\text{Data_in}} \cdot Q0$$

6. Next state logic for D0

Now let us deduce the expression for the next state logic for input $D0$ (Fig. 13.17).

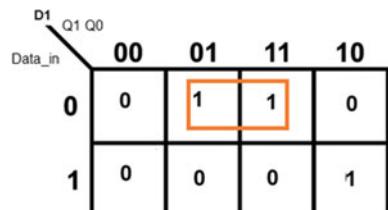
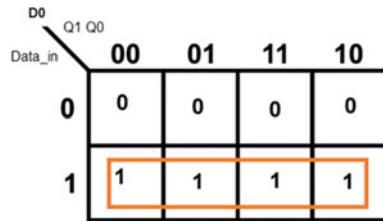
Fig. 13.16 The K-map for the next state logic at $D1$ 

Fig. 13.17 The K-map for the next state logic



$$Q0^+ = D0 = \text{Data_in}$$

7. Output logic

In the Mealy FSM output is function of the present state and inputs. Let us deduce the combinational logic at output (Fig. 13.18; Table 13.9).

Let us deduce the logic expression for the output combinational logic

$$\text{Data_out} = \text{Data_in} \cdot Q1Q0$$

8. Let us sketch the FSM Design

As discussed in the previous steps we need to have two flip-flops and logic gates to implement the 101 overlapping Mealy sequence detector. As shown in Fig. 13.9 the

Fig. 13.18 K-map for the output logic of Mealy machine

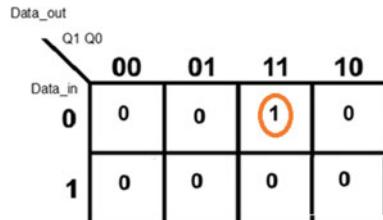


Table 13.9 The truth table of the output logic

Enable (data_in)	Present state ($Q1Q0$)	Output (data_out)
0	$s0$	0
1	$s0$	0
0	$s1$	0
1	$s1$	0
0	$s2$	0
1	$s2$	0
0	$s3$	1
1	$s3$	0

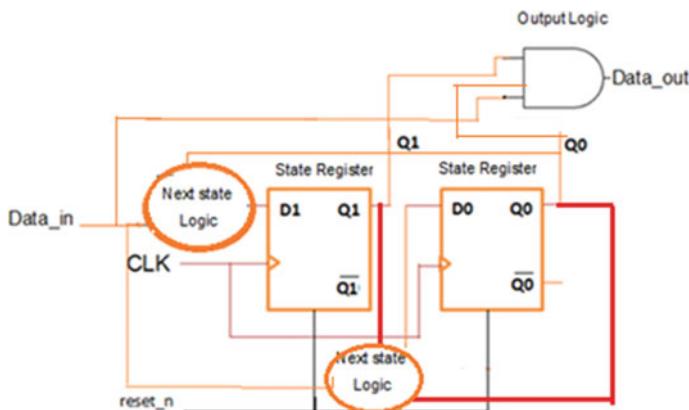


Fig. 13.19 The Mealy 1010 overlapping sequence detector

state register uses next state logic to get the next state. During the $\text{reset_n} = 0$ the output of the register is same as the default state s_0 .

As shown in Fig. 13.19 next state logic gates are denoted using elliptical boxes.

13.8 Various Paths in the Design

Practically the design has the

1. Clock paths
2. Reset paths
3. Data paths and control paths.

Few of the recommendations during the design are

- The design should have the better clock and reset management schemes. For example, if the asynchronous reset is used, then there should be the reset synchronizers to synchronize the internal asynchronous reset with the master reset.
- The clock distribution schemes should be such that there is uniform skew across the clock path.
- Do not generate the clocks using multiple sources as there is issue due to multiple clock domain and data convergence.
- If multiple clock is requirement for the design, then use the synchronizers in the data and control paths.
- Use the level synchronizers to pass the control signals from one of the clock domains to another clock domain.
- Use the FIFO synchronizers to pass the data between the clock domains.
- Have the better data and control path management..

For the complex design the data and control path is discussed in the following sections. For more details about multiple clock domain designs please refer Chap. 14.

13.9 Data and Control Path Design Techniques

Now let us use the FSM designed in the previous section to optimize the data and control paths. Most of the time we do not pay attention to have separate data and control paths and this significantly affects the area, speed, and power of the design. As discussed in the previous section if we have the better strategies to design the separate logic for the data and control path then we can have significant improvement in the timing and even in the area.

Consider now the design scenario, the design requirement is to enable the data path logic after detecting the sequence 101 to transfer the 16-bit of the data from register A to output. In such scenario we can use the following strategy.

1. *Design the control path logic:* Design the sequence detector to detect the sequence 101. The control logic uses the input as a clk, reset_n, and data_in and generates a pulse at data_out if 101 sequence is detected.
2. *Design the data path logic:* In the data path let us have the register A which can hold the 16-bit data and when it is enabled it transfers the contents of A to output Y_out.

The design is shown in Fig. 13.20.

In this chapter we have discussed about the design of the sequence detectors. Now let us discuss about the various performance improvement techniques in the next chapter.

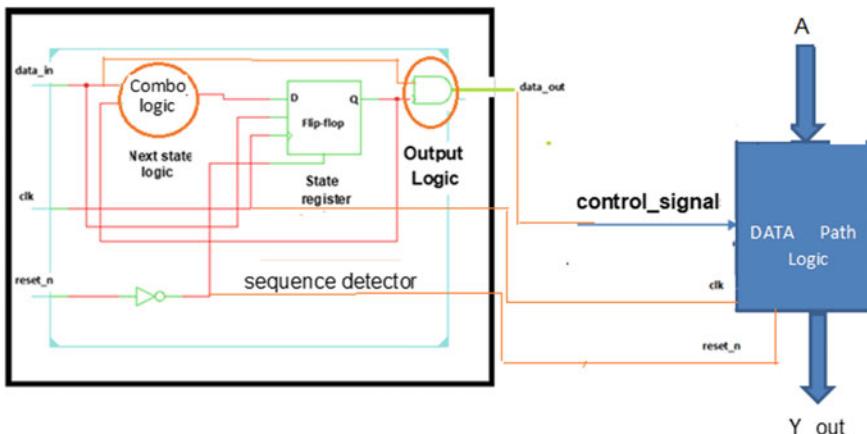


Fig. 13.20 Data and control path design strategy and use

13.10 Summary

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

1. Sequence detectors we can design by considering the overlapping or non-overlapping sequence.
2. As compared to the one-hot encoding the gray encoding optimizes the area as number of flip-flops for gray encoding are $\log_2 \text{States}$.
3. For the state machine design using one-hot encoding the number of flip-flops is equal to number of states.
4. If we have the better strategies to design the separate logic for the data and control path then we can have significant improvement in the timing and even in the area.
5. Practically the design has the
 - Clock paths
 - Reset paths
 - Data paths and control paths.

Chapter 14

Performance Improvement for the Design



At the various design phases, we need to use the performance improvement techniques.

During the design it is essential to understand and use the various performance improvement techniques. The performance of the design can be improved by using area, power, and speed improvement techniques. In this context the chapter discusses the basics of the performance improvement for the design and how we can tweak the architectures and micro-architectures to improve the design performance.

14.1 What Is Design Performance?

The design using digital logic elements has combinational and sequential logic, and the density of the logic is measured in the form of number of logic gates. The number of logic cells or logic gates in the design we can consider as the area of the design. For example, consider the design of the processor which has 16 instructions, and the design has pipelined architecture. The overall area of the design is addition of the combinational elements and sequential elements.

The speed of the design is another important factor, and the demand is of the high-speed logic. The timing parameters of the sequential elements and propagation delays of the combinational elements limit the speed of the design. During design or architecture phase we need to find the maximum frequency for the design and we need to use the speed improvement techniques for the design.

The power dissipation for the design is summation of the static and dynamic power, and we need to have the low-power-aware architectures to improve the power of the design.

The goals of the architecture and logic design engineers are to have design which has lesser area, maximum speed, and less power, and hence we will try to use various area, speed, and power improvement techniques during various design phases.

In the simple words the design has optimization constraints such as area, speed, and power, and these constraints should be met during each and every milestone.

This following sections are useful to understand the area, speed, and power improvement need and few classic scenarios in VLSI context.

14.2 How to Use the Minimum Arithmetic Resources

During the logic design phase if we wish to perform the arithmetic operations and logical operations, then always we consider about use of the combinational elements such as adders, subtractors, multiplexers, and logic gates. If we don't have the strategy to optimize the design, then the area of the design can lead to maximum number of the logic elements. The design can become bulky.

Consider the design scenario, we wish to perform the addition and subtraction on 4-bit binary numbers. What will we consider? We will try to identify the logic elements such as 4-bit adders and 4-bit subtractors to perform these operations.

For 4-bit pipelined processor to perform the addition we need to have the 4-bit adder and to perform the subtraction we need to have 4-bit subtractor. Hence double the resources and more the area. The following section discusses about the multibit adders and subtractors for your quick reference.

14.3 Multibit Adders and Subtractors

Most of the time, the multibit adders and subtractors are used during the design of Arithmetic Logic Unit (ALU). The logic density of ALU depends on the number of adders, subtractors, and other combinational elements used in the design.

We will use the component as full adder and full subtractor (Fig. 14.1) to design the multibit adder-subtractor.

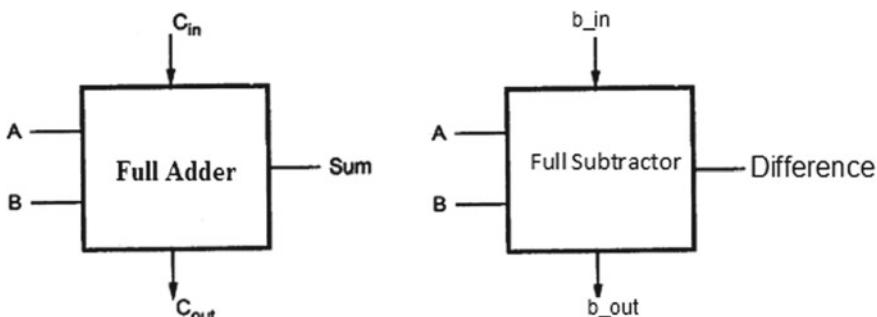


Fig. 14.1 Full adder and full subtractor

14.4 Four-Bit Full Adder

ALU designs we can use the multibit adder. Consider the 4-bit processor, and the processor performs arithmetic operations as addition, subtraction, increment, and decrement. To perform these operations, we use adder as combinational element.

Consider addition of A, B , we have four-bit binary input as A_3, A_2, A_1, A_0 and B_3, B_2, B_1, B_0 . To perform the addition, we can use the 4-bit binary adder which consists of 4 full adders. The logic design is shown in Fig. 14.2.

As shown in the design, the LSB adder uses A_0, B_0 inputs and carry input C_{in} . The carry output from the LSB propagates to the next adder and from MSB adder we can get the carry output C_{out} . The adder outputs are S_0, S_1, S_2, S_3 , and they are parallel outputs to get the sum of A, B binary inputs.

14.4.1 4-Bit Full Subtractor

Consider subtraction of A, B , we have four-bit binary input as A_3, A_2, A_1, A_0 and B_3, B_2, B_1, B_0 . To perform the subtraction, we can use the 4-bit binary subtractor which consists of 4 full subtractors. The logic design is shown in Fig. 14.3.

As shown in the design, the LSB full subtractor uses A_0, B_0 inputs and borrow input b_{in} . The borrow output from the LSB propagates to the next subtractor and from MSB adder subtractor we can get the borrow output b_{out} . The subtractor outputs are D_0, D_1, D_2, D_3 , and they are parallel outputs to get the difference of A, B binary inputs.

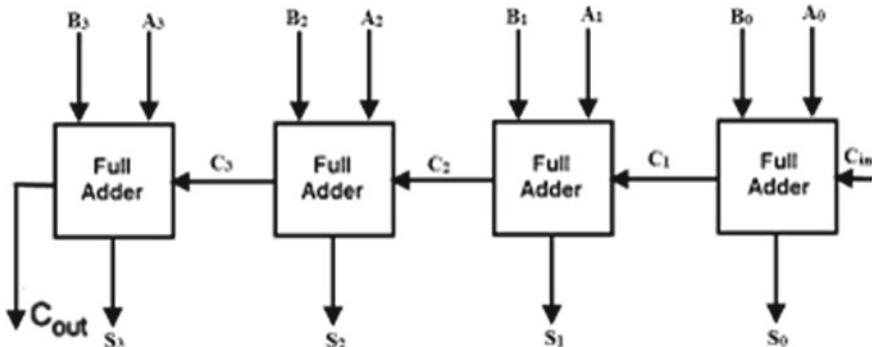


Fig. 14.2 Logic design of 4-bit binary adder

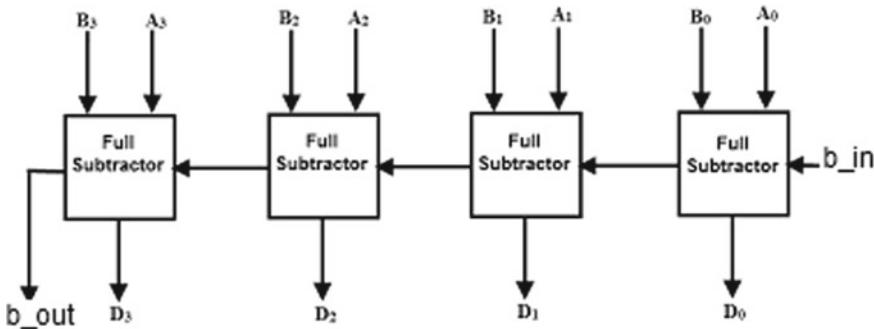


Fig. 14.3 Logic design of 4-bit binary subtractor

14.4.2 4-Bit Adder and Subtractor

Design of logic to perform the 4-bit addition and subtraction can be performed by using the adders only. This strategy is useful to optimize for the area. Subtraction can be performed using 2's complement addition. For example, consider the scenario shown in Table 14.1.

The design performs the addition when $C_{in} = 0$ and performs the subtraction when $C_{in} = 1$. The C_{in} acts as control input and can be treated as single-bit opcode.

The design of 4-bit adder and subtractor is shown in Fig. 14.4 and as shown the design has two data paths. The data path to perform the addition and data path to perform the subtraction. Depending on the status of control input control_in the result is generated.

The chain of multiplexer is used at output to select the result_out as addition or subtraction for control_in = 0 and control_in = 1 respectively.

The logic design shown has limitations. The main issue is more area and more power due to use of the adders and subtractors in two separate data paths. Both the data paths are active at a time, and design performs both the operations concurrently.

The output multiplexer chain selects either result of addition or result of subtraction.

Table 14.1 Operational table for adder subtractor

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

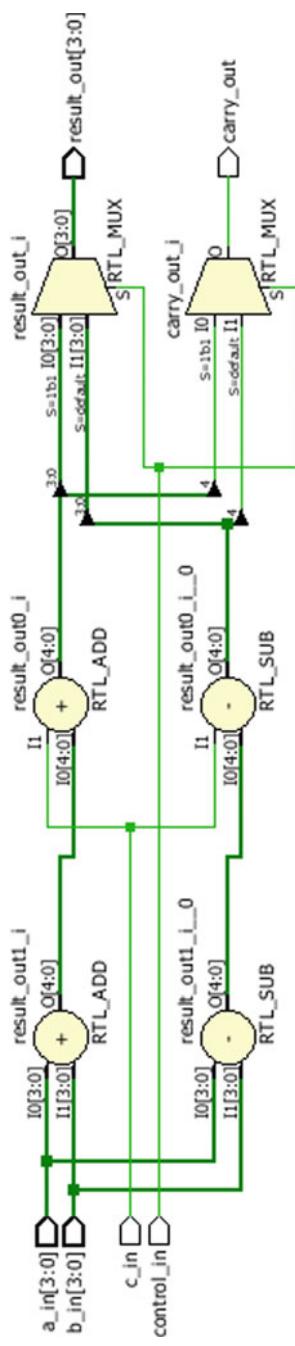


Fig. 14.4 Logic design of 4-bit adder/subtractor

Table 14.2 Operational table with optimization goal

Operation	control_in	Expression
Addition	0	$A + B + 0$
Subtraction	1	$A - B = A + \sim B + 1$

14.4.3 Area Optimization of 4-Bit Adder and Subtractor

As discussed in the previous section the design uses more resources or double the resources. The logic performs both operations at a time and at output multiplexer as selection logic is used to select from one of the operation depending on the multiplexer select input (control_in). The design is inefficient and needs optimization. The following section discusses few of the logic design techniques to optimize the design for better area.

14.4.4 Optimization of Design Using Only Adders

To optimize for the resources, let us try to implement the subtraction using 2's complement addition. The strategy is documented in Table 14.2.

The strategy used to implement the subtraction using 2's complement addition minimizes area as 2 full subtractors are not needed. The design uses three adders and chain of multiplexers at output (Fig. 14.5).

The design has issue as it performs both the operations at a time and needs to go through the optimization phase.

14.4.5 Optimization by Tweaking the Logic to Have Least Area and Least Power

To perform the single operation at a time and for the better data path and control path optimization let us use the common resource as adder. This technique is called as resource sharing. To perform the 4-bit addition and subtraction the common resource 4-bit adder is used and chain of multiplexer is pushed at the input side. The strategy is explained in Table 14.3.

The logic design is shown in Fig. 14.6, as shown the logic tweaked uses only two adders and multiplexer. At the output stage it infers the adder as common resource and the design has the optimized data and control path. Now the logic performs only one operation at a time.

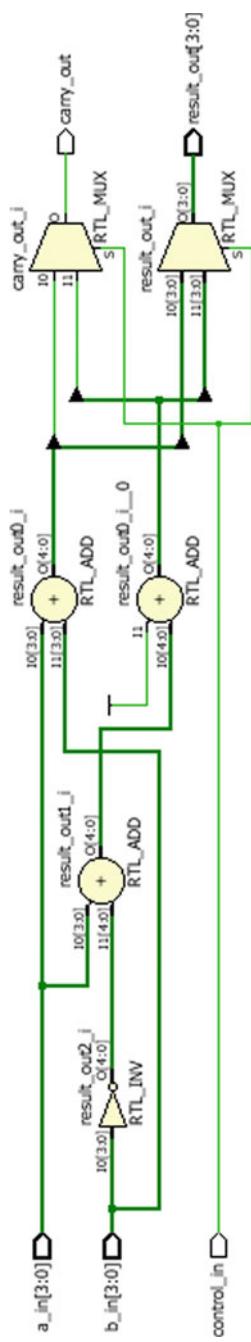


Fig. 14.5 Logic design after tweak

Table 14.3 Use of the common expression to have resource sharing

Operation	control_in	Variable input	Common expression
Addition	0	b_{in}	$A + B + control_in$
Subtraction	1	$\sim b_{in}$	$A + \sim B + control_in$

14.5 Optimization of the Design for Least Area and Power

The design can be further optimized by considering the area as main important constraints by using adders and XOR gates. One of the inputs of adder is binary input A , and other input of adder can be controlled by XOR gate.

For the $C_{in} = 0$ the XOR gate output is binary input B and for the $C_{in} = 1$ the output of XOR gate is complement of binary input B . Thus, it performs both the operations $A = B + 0$ and $A + \sim B + 1$ to perform adder and subtraction respectively at a time.

The design is shown in the figure, and the design has least area and lesser power as only one operation is performed at a time. The design uses only four full adders and four XOR gates. For more information about the resource sharing please refer Chap. 15. Logic design after optimization is shown in Fig. 14.7.

14.6 Comparators and Parity Detectors with Lesser Area

In most of the practical scenarios, comparators are used to compare the 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. The goal of the designer is to design the logic using least logic elements.

14.6.1 Binary Comparator Design with Least Area

These are used to compare the two binary numbers. For example, consider the design of comparator to get three outputs (Table 14.4). As shown in the table, for $a_{in} = b_{in}$ $a_{equal_b} = 1$, $a_{in} > b_{in}$ $a_{greater_b} = 1$, $a_{in} < b_{in}$ $a_{less_b} = 1$. But it should generate the parallel output to reflect the result. The design has parallel inputs and parallel outputs.

The logic of the comparator can be designed using the multiplexers and logic gates. The logic design is shown Fig. 14.8. The logic elements $>$ and $=$ can be designed using logic gates. The outputs $a_{greater_b}$, a_{equal_b} , a_{less_b} are generated from the tree of multiplexers at the output, and it is inefficient design.

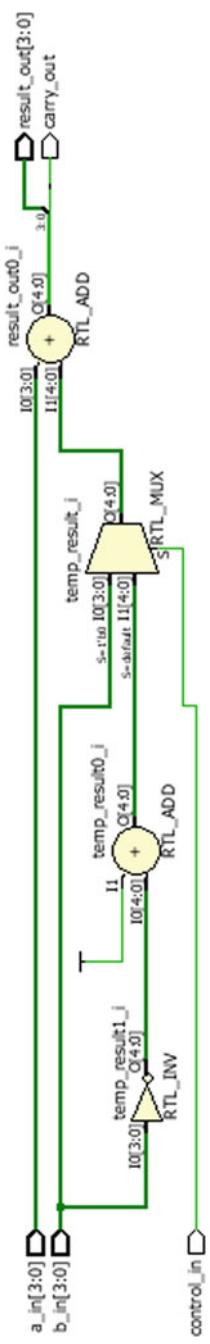


Fig. 14.6 Logic schematic after optimization

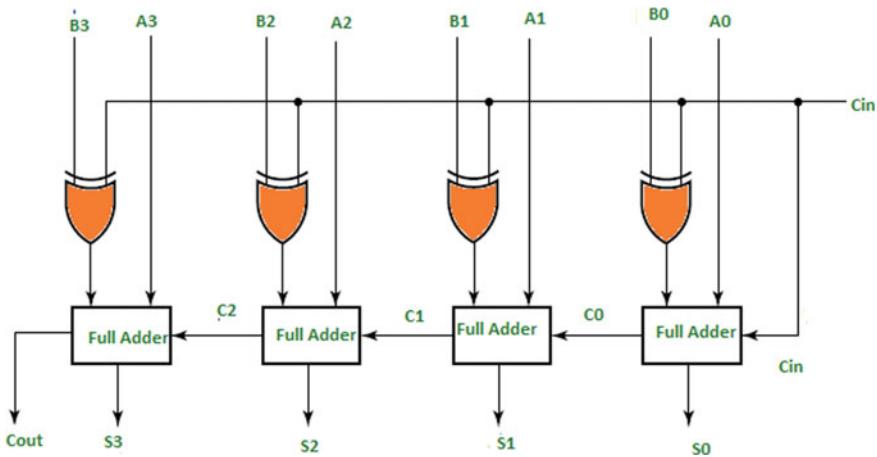


Fig. 14.7 Logic design after optimization

Table 14.4 Operational table for 1-bit comparator

a_in	b_in	a_greater_b	a_equal_b	a_less_b
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

The design can be optimized to improve the area using the logic gates. The logic design of the 1-bit comparator is shown in Fig. 14.9, and the design uses minimum number of logic gates that is 2 AND gates, XOR gate, and 2 NOT gates.

14.6.2 Parity Detector Design with Least Area

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 logic 0 and for odd number of 1's the output required is logic 1 then the logic can be designed using the XOR gates.

The operational table for the parity detector is shown in Table 14.5. For odd number of 1's the output is logic 1 and for even number of 1's output is assigned as logic 0.

The logic design is shown in Fig. 14.10. As shown the design uses the cascaded XOR gates to generate the parity output as logic 1 for odd number of 1's in the 8-bit binary number string. The design has the propagation delay of $n * tpd$ where $n = \text{number of XOR gates}$. Here $n = 7$ and if $tpd = 0.5 \text{ ns}$, then the design has the propagation delay of 3.5 ns.

Better design strategy is to use the 8-input XOR gate having delay of around 1 ns and maximum fanout.

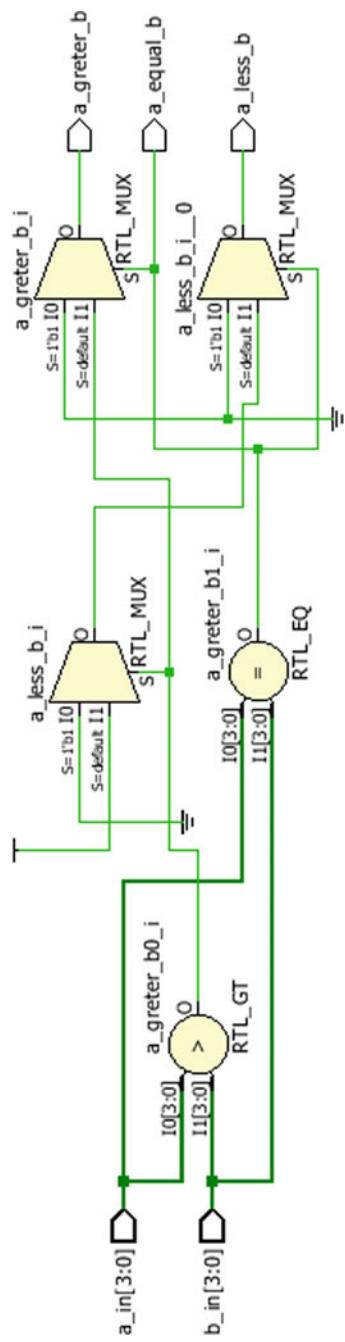


Fig. 14.8 Logic design of 1-bit comparator

Fig. 14.9 1-bit comparator with least area

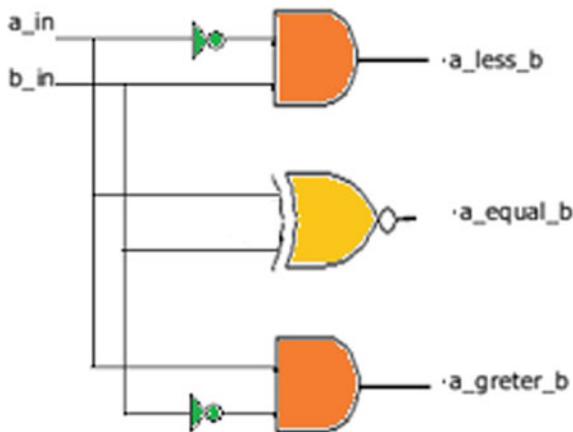


Table 14.5 Operational table for parity detector

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

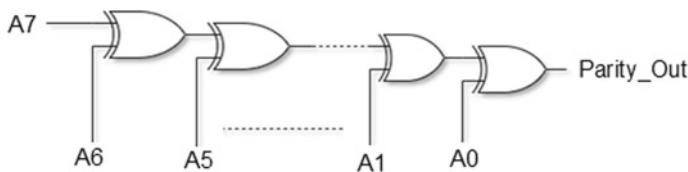


Fig. 14.10 Logic design of parity detector

14.7 Processor Designs and Speed Improvement Techniques (Source: www.onerupeest.com)

Consider the design of 4-bit processor. The design is single clock domain design and processor performs the arithmetic operations such as addition, subtraction, increment, and decrement. The logical operation processor performs as OR, AND, XOR, and NOT. The processor has following main important functional blocks

1. 4-bit ALU
2. Serial IO control logic
3. Hardware interrupt control logic
4. Instruction decoding unit
5. Internal memory
6. IO interfaces
7. Status and control logic

8. High-speed bus interface logic (Fig. 14.11).

By using the top-level understanding we need to work on the top-level interface signals such as clk, reset, serial IO pins, interrupt pins, address, data, and control bus.

According to the requirements we can think about the design of the instruction control mechanism and 4-bit ALU. The IO signals are shown in Fig. 14.12 for 4-bit processor which has 8 instructions that is 3-bit opcode and the 4-bit operands.

The design of ALU is discussed in Chap. 11, and to improve the speed of the design and to have the better and clean timing we can use the registered inputs and registered outputs. For the area optimization please refer Chap. 15. The ALU design strategy is shown in the Fig. 14.13.

The ALU uses two operands X , Y and each operand is 4-bit. The output from ALU is 4-bit, and ALU result is used to generate flags as carry, parity, and sign. To

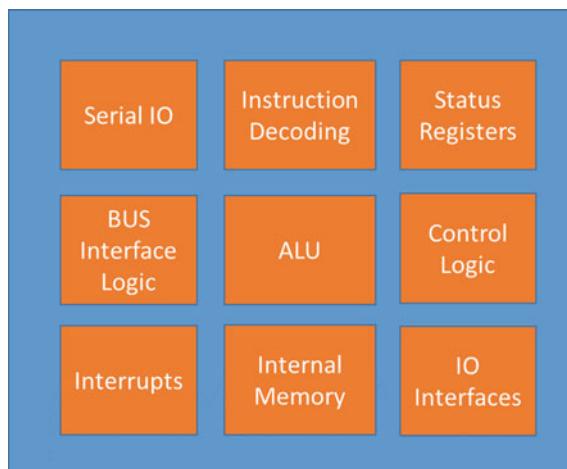


Fig. 14.11 Processor top-level architecture

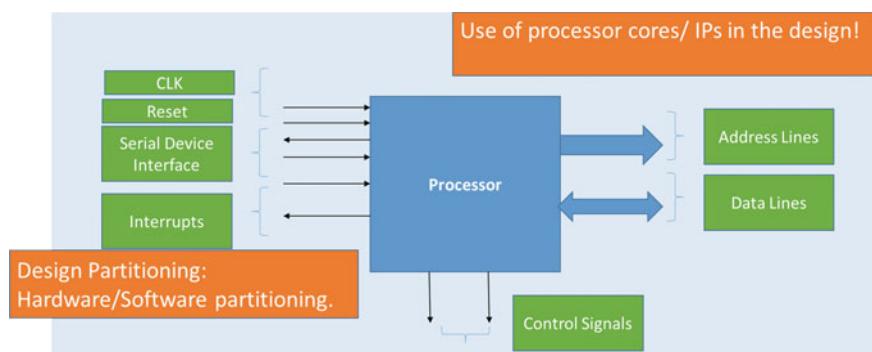


Fig. 14.12 Processor IO signals

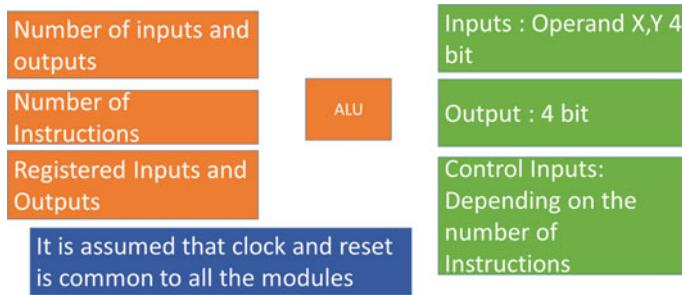


Fig. 14.13 ALU design strategy

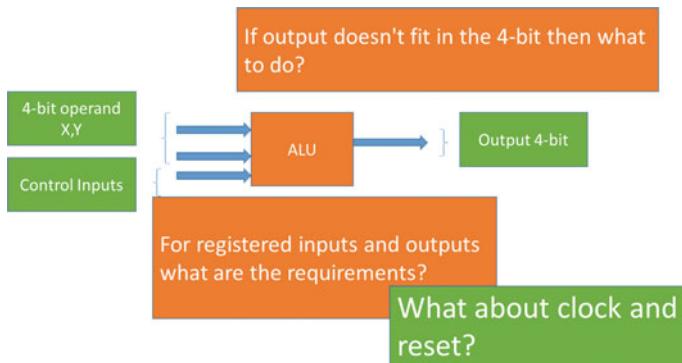


Fig. 14.14 ALU inputs and outputs

perform the 8 instructions, the ALU uses 3-bit control input and it can be also treated as 3-bit opcode. The ALU inputs and outputs are shown in Fig. 14.14.

Now let us think how we can improve the speed of the design. To improve the speed of the design we can use following

1. ALU having registered inputs and registered outputs
2. Pipelining of operations using pipelined stages (For more information refer Chap. 15)
3. Use the register balancing

The pipelining can be achieved by using separate functional blocks to perform the fetch. Decode, execute, and store. For more details about the processor architecture design using pipelined stages please refer Chap. 15. The Figs. 14 and 15 shows the pipeline register stages.

The ALU speed can also be improved by using the registered inputs and registered outputs design. This avoids the impact of the input and output insertion delay in the design. The ALU design top-level schematic is shown in Fig. 14.16.

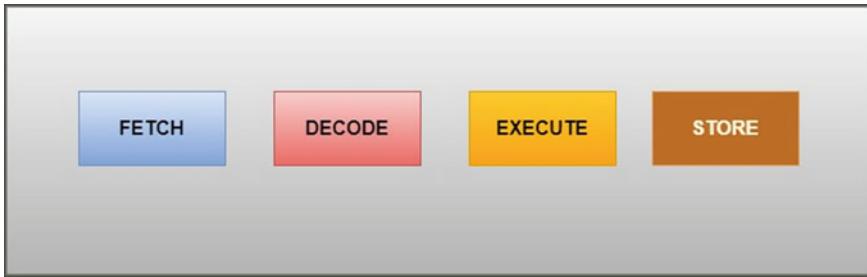
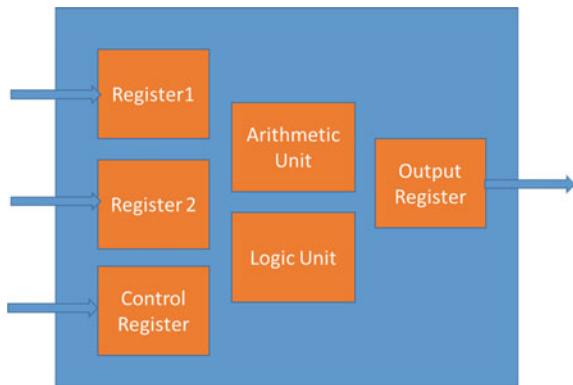


Fig. 14.15 Pipelined register blocks

Fig. 14.16 ALU having registered inputs and registered outputs



14.8 Avoid Asynchronous Designs to Improve the Speed

Most of the designs we have discussed till now are single clock domain designs and during logic and architecture design phase we should take care to avoid the asynchronous counters.

As shown in the figure, if all the divide by 2 counters doesn't have the common clock then the design is asynchronous. The issue with the asynchronous design is cumulative delay to get the output due to cascading of the counters. This type of the design is having issues due to uneven clock skew.

Consider the design of divide by 16 asynchronous counter shown in Fig. 14.17. The design has four outputs QA, QB, QC, QD and assumption is that LSB flip-flop receives clock input and the clock input of subsequent stages is generated from the output of previous stages.

Asynchronous clocking is major issue as the design has non-uniform clock skew and hence the triggering is an issue. Avoid use of the asynchronous designs and hence it is recommended to use synchronous design for better timing performance.

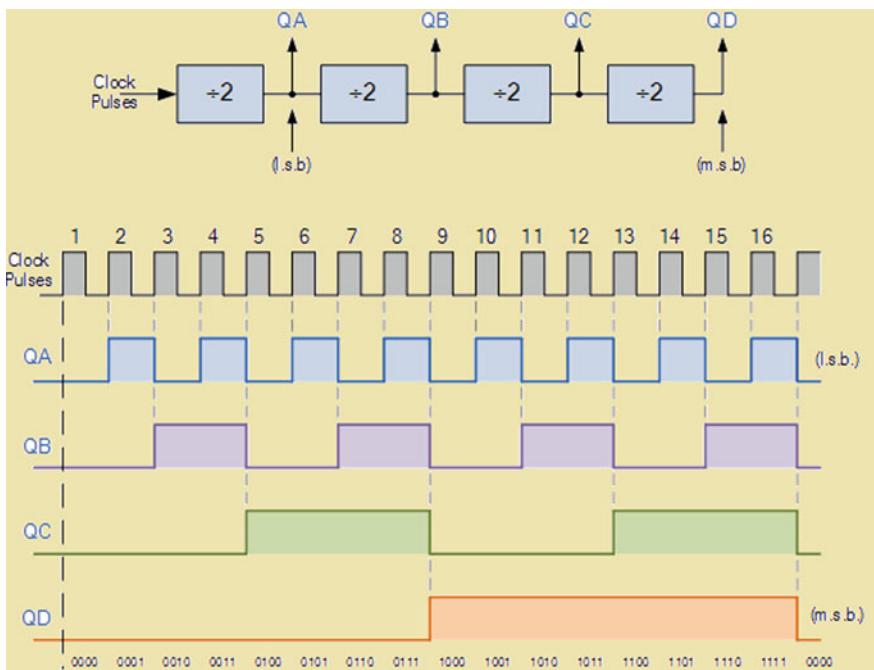


Fig. 14.17 Asynchronous clocking

14.9 Power Improvement

Power is one of the important criteria and can be improved by using the following strategy

1. Use of the low voltage and low power logic elements
2. Using multiple Vdd designs
3. Using the clock gating.

For more details about the power please refer Chap. 15. The clock gating is discussed in the following section, and this technique is useful in the dynamic power reduction.

14.9.1 Gated Clocks and Dynamic Power Reduction

Gated clock signals are used to turn on or turn off the switching at the clock net for the multiple clock domain designs and use the clock enable inputs. When enable input is high the clock switching is on and when enable input is low the clock switching is off. The clock gating logic is used to control the clock turn on or turn off. Clock

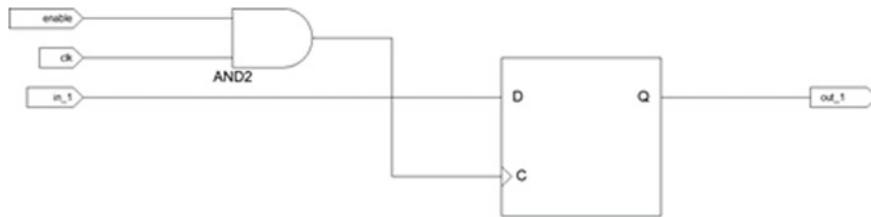


Fig. 14.18 Clock gating logic

gating is an efficient technique used in the ASIC and FPGA designs to reduce the switching power at the clock input of register or flip-flop. By using the clock gating cell, the clock switching can be controlled as and when required according to the design functional requirements. The clock gating is useful to reduce the dynamic power for the design.

But the issue with the clock gating is that it can't 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 needs to be taken by ASIC or FPGA design engineer.

The design uses enable input to control the clock switching activity. For ‘enable = 1’ the clock input ‘clk’ toggles and for ‘enable = 0’ clock input is active low so no switching at clock input. Thus, it reduces the dynamic power.

The logic is shown in Fig. 14.18 where clock is gated by using AND logic. The logic is prone to glitches, and it is recommended to use the dedicated clock gating cell. Please refer Chap. 15 to have more understanding about the low power design.

In this chapter we have discussed the basics of the area, speed, and power. In the next subsequent chapters, our goal is to understand in much more detail about the area, speed, and power improvement techniques and design scenarios.

14.10 Summary

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

1. The goals of the architecture and logic design engineers are to have design which has lesser area, maximum speed, and less power, and hence, we will try to use various area, speed, and power improvement techniques during various design phases.
2. For each design we should understand the optimization constraints such as area, speed, and power, and these constraints should be met during each and every milestone.
3. The resource sharing is one of the powerful area optimization techniques.
4. The speed of the design is improved by using register balancing and pipelining.
5. Don't use asynchronous clocking as the skew is issue.

6. Use the power reduction techniques such as clock gating to reduce the dynamic power.

Chapter 15

Optimization Techniques



The logic design team should use the various optimization techniques during the design.

As discussed in the previous chapters we sketch the design architecture to have minimum area, maximum speed, and less power. In the VLSI context we need to understand the various area, speed and power optimization techniques. The chapter discusses about the various techniques used during the logic and architecture design to have the lesser area for the design.

15.1 Let Us Understand About the Area Optimization

Consider the design of two instructions

1. Addition: $A + B$
2. Subtraction: $A - B$.

Most of the time if we try to design the logic for the addition and subtraction we land up with more resources. The better way is to understand the common resources and utilize them to get the lesser area.

The following section is useful to understand the resource sharing for arithmetic operations.

15.2 Arithmetic Resource Sharing

Most of the time we experience the need of the arithmetic resource sharing. During the logic design of the arithmetic unit we can improve the performance of the design by sharing the common arithmetic resources such as adders, multipliers to have the better data and control path optimization of the design!

Now let us consider the operations documented in Table 15.1. Using the entries from table let us identify the component to perform 4-bit addition. The components we need are four full adders and three multiplexers or XOR gate. The design requirement is that it should perform the single operation depending on the status of op_code.

The design without resource sharing we need the four full adders, four full subtractors and chain of multiplexers at output. But without resource sharing the logic density is higher.

So let us have strategy to share the common resource as adder. As we know that subtraction is 2's complement addition we can use the adder to perform the subtraction. The entries are shown in Table 15.2.

Thus, the design needs three full adders and three XOR gates. The MSB of result we can treat as carry output. The logic design is shown in Fig. 15.1.

Table 15.1 Functional table description

op_code	Operation
0	Addition of A, B
1	Subtraction of A, B

Table 15.2 Resource sharing entries

op_code	Operation	Boolean expression
0	Addition of A, B	$A + B$
1	Subtraction of A, B	$A + \sim B + 1$

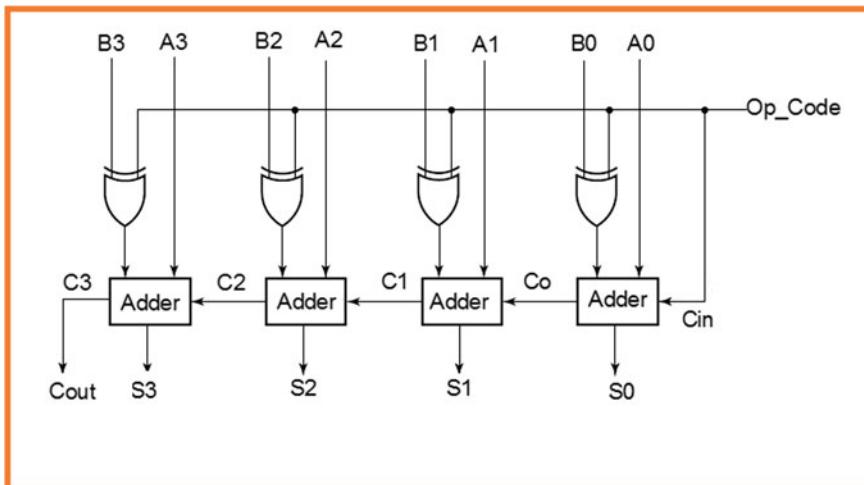
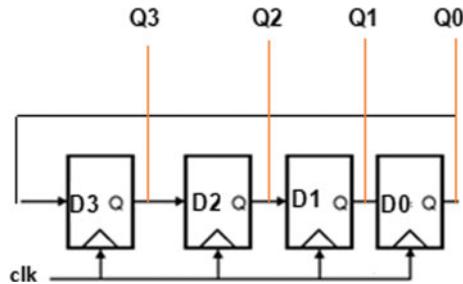
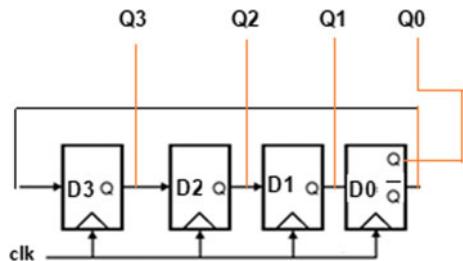


Fig. 15.1 Logic with resource sharing

Fig. 15.2 4-bit ring counter**Fig. 15.3** 4-bit Johnson counter

15.3 Resource Sharing for Sequential Circuits

Consider the sequential circuit to design the ring and Johnson counter. If we consider the design of both these counters we need four flip-flops. So, we can use the common resources as flip-flops.

The 4-bit ring counter is shown in Fig. 15.2. As shown the output Q is feedback to data input of MSB flip-flop.

The 4-bit Johnson counter is shown in Fig. 15.3. As shown the output complement of Q is feedback to data input of MSB flip-flop.

Design Using Resource Sharing:

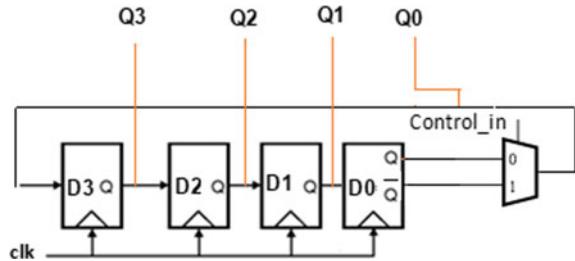
We can use the common resources as flip-flops and 2:1 multiplexer at the output to get the Q or complement of Q.

For $\text{control_in} = 0$ the design acts as ring counter, and for $\text{control_in} = 1$ the design acts as Johnson counter. This strategy is useful to get the design with minimum resources (Fig. 15.4).

15.4 Logic Duplications

Let us consider the design of the 4:16 decoder using minimum number of 2:4 decoders. The design needs five 2:4 decoders. Now why?

Fig. 15.4 Sequential design using resource sharing



As 2:4 decoder need to be used, let us have the truth table (Table 15.3) of the 4:16 decoder having active high output during $en = 1$. For $en = 0$ the decoder all outputs are active low.

Now to get the 16 outputs y_0 to y_{15} let us use the four 2:4 decoders. To select the one of the decoder let us use the 2:4 decoder at input. Table 15.4 gives information about the selection strategy for one of the output decoder.

The 4:16 decoder design using the minimum number of 2:4 decoders is shown in Fig. 15.5. As shown the 4:16 decoder is implemented by using five 2:4 decoders.

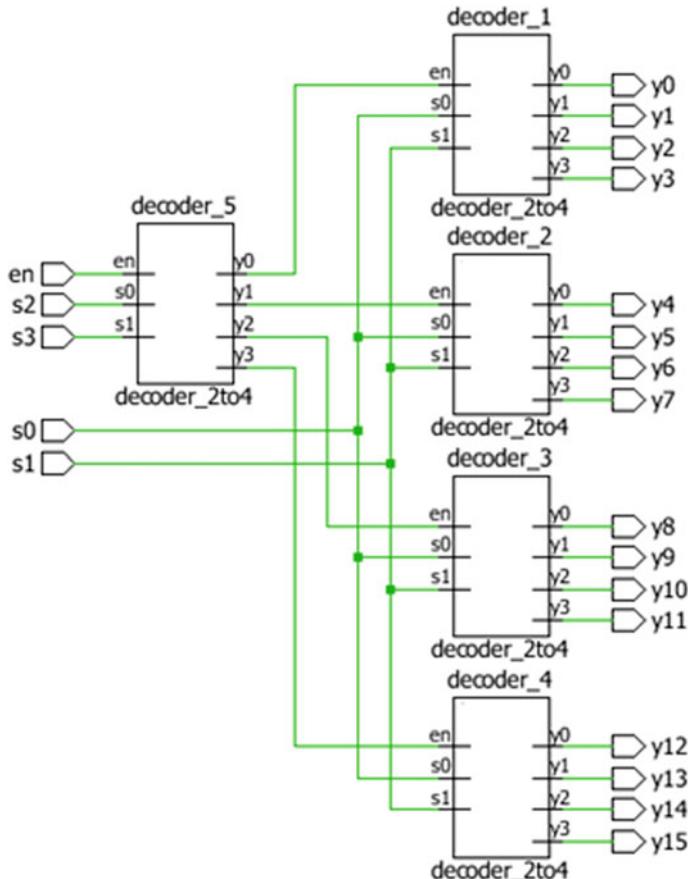
Now let us use the logic duplication and try to optimize the resources. Using the logic duplications, we need to use the two, 2:4 decoders. As shown in the table s3, s2 inputs are same for four combinations we can use the 2:4 decoder having select

Table 15.3 Truth table of 4:16 decoder

Enable (en)	s3s2s1s0	Decoder outputs ($y_{15}-y_0$)
1	0000	0000_0000_0000_0001
1	0001	0000_0000_0000_0010
1	0010	0000_0000_0000_0100
1	0011	0000_0000_0000_1000
1	0100	0000_0000_0001_0000
1	0101	0000_0000_0010_0000
1	0110	0000_0000_0100_0000
1	0111	0000_0000_1000_0000
1	1000	0000_0001_0000_0000
1	1001	0000_0010_0000_0000
1	1010	0000_0100_0000_0000
1	1011	0000_1000_0000_0000
1	1100	0001_0000_0000_0000
1	1101	0010_0000_0000_0000
1	1110	0100_0000_0000_0000
1	1111	1000_0000_0000_0000
0	XXXX	0000_0000_0000_0000

Table 15.4 Decoder selection strategy

Enable (en)	s3	s2	Decoder selected
1	0	0	Decoder #1
1	0	1	Decoder #2
1	1	0	Decoder #3
1	1	1	Decoder #4
0	X	X	All outputs are zero

**Fig. 15.5** 4:16 decoder using minimum number of 2:4 decoders

inputs as s3, s2 and another 2:4 decoder to have inputs as s1, s0. For each combination of s3, s2 we need four AND gates. So, these AND gates are duplicated at the output.

So, using the logic duplication technique we need two 2:4 decoders and 16 AND gates. This optimizes area, and it is design specific.

For this design each decoder logic has 11 gates (eight, 2 input AND gates and three NOT gates). Hence without logic duplication we need 55 logic gates. Using logic duplications, we need only 22 plus 16 that is 38 logic gates. Hence reduction in the overall area.

The design using logic duplications is shown in Fig. 15.6 and uses two, 2:4 decoders and 16 AND gates.

15.5 Design Scenario: Performance Improvement

To improve the speed of the design, consider the design scenario. The objective is to improve the design timing.

To find the maximum operating frequency for the design (Fig. 15.7) use the reg-to-reg timing path that is longest path in the design. In the design the start point is clk pin of MSB flip-flop and endpoint is D0 of the LSB flip-flop.

Let us consider $t_{\text{pff}} = 1 \text{ ns}$, $t_{\text{combo}} = t_{\text{not}} = \text{propagation delay of Not gate} = 1 \text{ ns}$ and $t_{\text{su}} = \text{setup time} = 1 \text{ ns}$ and $t_h = \text{hold time} = 0.5 \text{ ns}$.

1. Let us find out the data arrival time (AT). $\text{AT} = t_{\text{pdff1}} + t_{\text{not}}$
2. The data required time is RT. $\text{RT} = T_{\text{clk}} - t_{\text{su}}$ because the data at D input should be stable by t_{su} margin prior to arrival of rising edge of the clock.
3. Now find setup slack. $\text{Slack} = \text{RT} - \text{AT}$ and should be greater than or equal to 0.
4. $\text{Slack} = \text{RT} - \text{AT}$

$$\text{Setup Slack} = (T_{\text{clk}} - t_{\text{su}}) - (t_{\text{pdff1}} + t_{\text{not}})$$

5. If we equate the slack to zero then we get

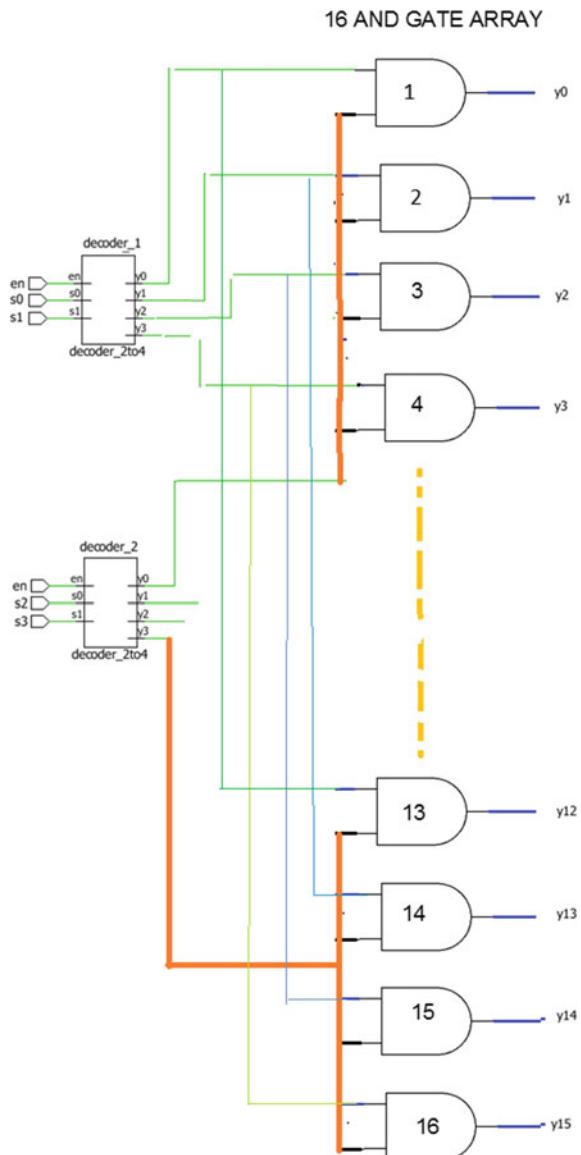
$$0 = (T_{\text{clk}} - t_{\text{su}}) - (t_{\text{pdff1}} + t_{\text{not}})$$

$$T_{\text{clk}} = t_{\text{pdff1}} + t_{\text{not}} + t_{\text{su}}$$

6. The maximum operating frequency of the design is f_{\max}

$$f_{\max} = \frac{1}{T_{\text{clk}}}$$

Fig. 15.6 4:16 decoder using logic duplication technique



$$= \frac{1}{t_{pdff1} + t_{not} + t_{su}}$$

$$= \frac{1}{1\text{ ns} + 1\text{ ns} + 1\text{ ns}}$$

$$= \frac{1}{3\text{ ns}}$$

$$= 333.33 \text{ MHz}$$

Now let us improve the design performance!

The logic in the design (Fig. 15.7) can be tweaked by removing NOT gate from the data path. Directly the complement of Q is given as data input to D flip-flop.

To find the operating frequency for the design (Fig. 15.8) use the following steps.

1. Let us find out the data arrival time (AT). $AT = t_{pdff1}$
2. The data required time is RT. $RT = T_{clk} - t_{su}$ because the data at D input should be stable by t_{su} margin prior to arrival of rising edge of the clock.
3. Now find setup slack. $Slack = RT - AT$ and should be greater than or equal to 0.
4. $Slack = RT - AT$

$$\text{Setup Slack} = (T_{clk} - t_{su}) - (t_{pdff1})$$

5. If we equate the slack to zero then we get

$$0 = (T_{clk} - t_{su}) - (t_{pdff1})$$

Fig. 15.7 Design before logic tweaks

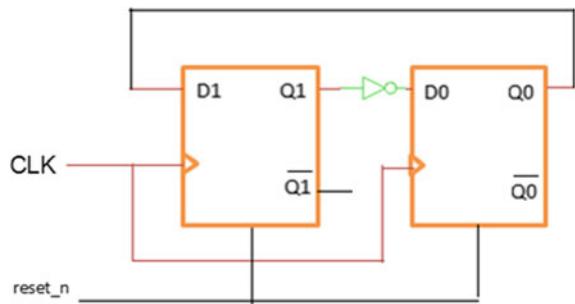
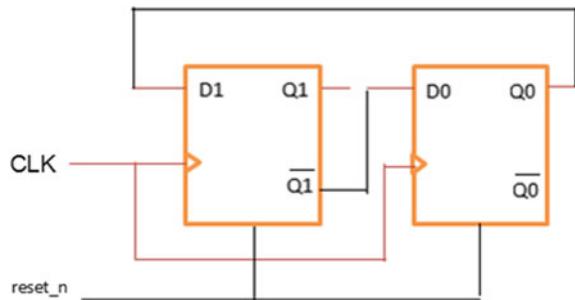


Fig. 15.8 Design after logic tweak



$$T_{\text{clk}} = t_{\text{pdff1}} + t_{\text{su}}$$

6. The maximum operating frequency of the design is f_{\max}

$$\begin{aligned} f_{\max} &= \frac{1}{T_{\text{clk}}} \\ &= \frac{1}{t_{\text{pdff1}} + t_{\text{su}}} \\ &= \frac{1}{1 \text{ ns} + 1 \text{ ns}} \\ &= \frac{1}{2 \text{ ns}} \\ &= 500 \text{ MHz} \end{aligned}$$

Thus, at logic level we can improve the design performance using logic tweaks.

15.6 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 processor designs, and FPGA and ASIC designs perform multiple tasks at a time. This section discusses about the design without pipelining and design with pipelining.

15.6.1 Design Without Pipelining

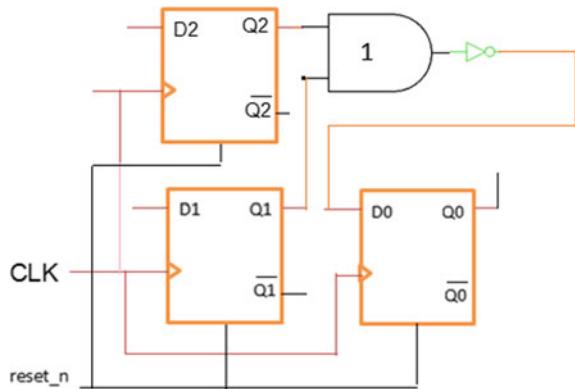
Consider the design shown in Fig. 15.9 and does not have any pipelined logic. Now let us try to find the performance of the design.

To find the maximum operating frequency for the design (Fig. 15.9) use the reg-to-reg timing path that is longest path in the design. In the design the start point is clk pin of two parallel flip-flops and endpoint is D0 of the LSB flip-flop.

Let us consider $t_{\text{pff}} = 1 \text{ ns}$, $t_{\text{combo}} = t_{\text{not}} = \text{propagation delay of Not gate} = 1 \text{ ns}$, $t_{\text{and}} = \text{propagation delay of AND gate} = 1 \text{ ns}$ and $t_{\text{su}} = \text{setup time} = 1 \text{ ns}$ and $t_h = \text{hold time} = 0.5 \text{ ns}$.

1. Let us find out the data arrival time (AT). $AT = t_{\text{pdff1}} + t_{\text{and}} + t_{\text{not}}$
2. The data required time is RT. $RT = T_{\text{clk}} - t_{\text{su}}$ because the data at D input should be stable by t_{su} margin prior to arrival of rising edge of the clock.

Fig. 15.9 Logic without pipelined stage



3. Now find setup slack. Slack = RT – AT and should be greater than or equal to 0.
4. Slack = RT – AT

$$\text{Setup Slack} = (T_{\text{clk}} - t_{\text{su}}) - (t_{\text{pdf}f1} + t_{\text{and}} + t_{\text{not}})$$

5. If we equate the slack to zero then we get

$$0 = (T_{\text{clk}} - t_{\text{su}}) - (t_{\text{pdf}f1} + t_{\text{and}} + t_{\text{not}})$$

$$T_{\text{clk}} = t_{\text{pdf}f1} + t_{\text{and}} + t_{\text{not}} + t_{\text{su}}$$

6. The maximum operating frequency of the design is f_{\max}

$$\begin{aligned} f_{\max} &= \frac{1}{T_{\text{clk}}} \\ &= \frac{1}{t_{\text{pdf}f1} + t_{\text{and}} + t_{\text{not}} + t_{\text{su}}} \\ &= \frac{1}{1 \text{ ns} + 1 \text{ ns} + 1 \text{ ns} + 1 \text{ ns}} \\ &= \frac{1}{4 \text{ ns}} \\ &= 250.00 \text{ MHz} \end{aligned}$$

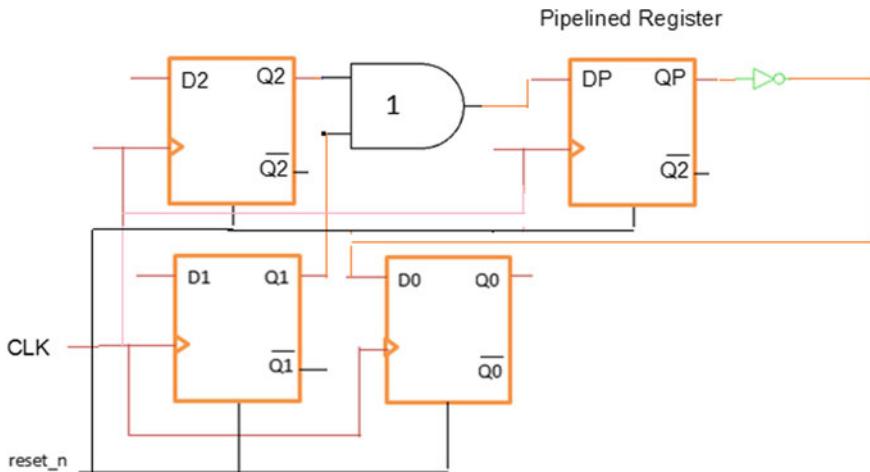


Fig. 15.10 Synthesized logic with pipelined stage

15.6.2 Speed Improvement Using Register Balancing or Pipelining

To improve the design performance the combinational logic AND output can be passed through the pipelined register (input of register D_p and output Q_p) and the output of the pipelined register can drive NOT gate (Fig. 15.10).

This technique will improve the overall performance of the design at the cost of one clock latency. The improvement in the design performance is due to the reduction in the combinational delay in the register-to-register path. This improves the data arrival time for the reg-to-reg path! The reg-reg path will have balanced delay; this is also treated as register balancing.

To find the maximum operating frequency for the design (Fig. 15.9) use the reg-to-reg timing path that is longest delay path that is critical path in the design. In the design the start point is clk pin of two parallel flip-flops and endpoint is D0 of the LSB flip-flop.

Let us consider $t_{pff} = 1$ ns, $t_{combo} = t_{not} =$ propagation delay of Not gate = 1 ns, t_{and} = propagation delay of AND gate = 1 ns and t_{su} = setup time = 1 ns and t_h = hold time = 0.5 ns.

1. Let us find out the data arrival time (AT). $AT_1 = t_{pdff1} + t_{and}$ and $AT_2 = t_{pdff1} + t_{not}$. As AND, NOT gate delays are equal consider AT1 as arrival time.
2. The data required time is RT. $RT = T_{clk} - t_{su}$ because the data at D input should be stable by t_{su} margin prior to arrival of rising edge of the clock.
3. Now find setup slack. $Slack = RT - AT$ and should be greater than or equal to 0.
4. $Slack = RT - AT$

$$\text{Setup Stack} = (T_{\text{clk}} - t_{\text{su}}) - (t_{\text{pdff1}} + t_{\text{and}})$$

5. If we equate the slack to zero then we get

$$0 = (T_{\text{clk}} - t_{\text{su}}) - (t_{\text{pdff1}} + t_{\text{and}})$$

$$T_{\text{clk}} = t_{\text{pdff1}} + t_{\text{and}} + t_{\text{su}}$$

6. The maximum operating frequency of the design is f_{\max}

$$\begin{aligned} f_{\max} &= \frac{1}{T_{\text{clk}}} \\ &= \frac{1}{t_{\text{pdff1}} + t_{\text{and}} + t_{\text{not}} + t_{\text{su}}} \\ &= \frac{1}{1 \text{ ns} + 1 \text{ ns} + 1 \text{ ns} + 1 \text{ ns}} \\ &= \frac{1}{3 \text{ ns}} \\ &= 333.33 \text{ MHz} \end{aligned}$$

Thus, due to use of the pipelined register the delays in the reg-to-reg path delay got balanced and the overall performance has improved.

15.7 Power Improvement of Design

Now let us understand the power optimization at the logic level. Consider the design of the Moore FSM for the given specifications to have the lesser power and lesser area. As discussed in the previous chapters what we need to do is that we need to design the digital logic to have better area and power performance for the

1. State register
2. Next state logic
3. Output logic

Let us consider the design of the sequential circuit to get the output data_out using the gray counter. We can use the following design steps to design the Moore FSM using gray encoding method.

Table 15.5 State table of the 2-bit gray counter

Present state (q0)	Next state (q0 ⁺)
s0	s1
s1	s2
s2	s3
s3	s0

1. Find the number of states to get the 2-bit gray output

Number of states = 4. The states are s0, s1, s2, s3, and s1.

2. State Transition

The state transition happens depending on the active edge of the clock.

3. Find number of flip-flops

We will use the gray encoding and the number of flip-flops = $n = \log_2 4 = 2$. We will use the positive edge sensitive D flip-flops.

4. Reset strategy

Let us use active low asynchronous reset input `reset_n`. For `reset_n` = 0 the counter holds the previous output. For the `reset_n` = 1 the output increments on the rising edge of the clock.

5. Let us document the entries in the state table to get state register logic

The state table of the synchronous gray counter is shown in Table 15.5.

To have the 2-bit gray counter we need to have two D flip-flops having asynchronous active low reset input `reset_n` and active high `data_in`.

6. Use of the excitation table to design the next state logic

The excitation table consists of the information about the present state, next state, and excitation input (Table 15.6).

Now let us use the K-map to deduce the Boolean equation for the next state logic to get the logic at D1, D0.

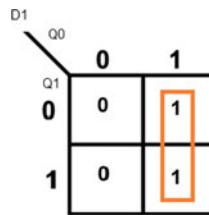
1. Next state logic for D1

Let us deduce the logic expression for the next state logic for D1.

Table 15.6 Excitation table of the 2-bit gray counter

Present state (Q1 Q0)	Next state (Q1 ⁺ Q0 ⁺)	Excitation inputs (D1 D0)
s0 = 00	s1 = 01	01
s1 = 01	s2 = 11	11
s2 = 11	s3 = 10	10
s3 = 10	s0 = 00	00

$$Q1^+ = D1 = Q0$$



$$Q1^+ = D1 = Q0$$

2. Next state logic for D0

Let us deduce the logic expression for the next state logic for D0 (Fig. 15.11).

$$Q0^+ = D0 = \overline{Q1}$$

7. Output logic

In the Moore FSM output is function of the present state only. Let us deduce the combinational logic at output (Fig. 15.12 and Table 15.7).

Let us deduce the logic expression for the output combinational logic

Fig. 15.11 K-map for the next state logic

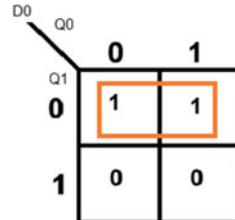


Fig. 15.12 K-map for the output logic of 2-bit gray counter

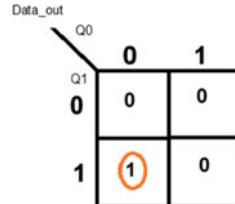
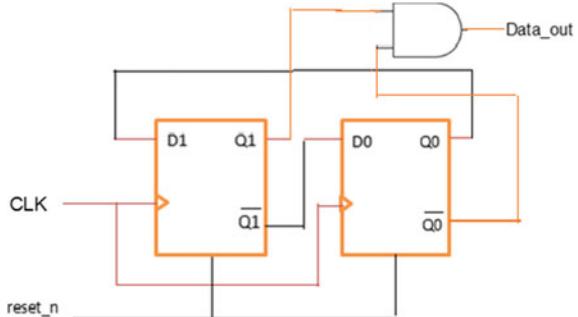


Table 15.7 Truth table of the output logic

Present state (Q1 Q0)	Next state (Q1 ⁺ Q0 ⁺)	Output (Data_out)
s0 = 00	s1 = 01	0
s1 = 01	s2 = 11	0
s2 = 11	s3 = 10	0
s3 = 10	s0 = 00	1

Fig. 15.13 2-bit gray counter using FSM design approach

$$\text{Data}_{\text{out}} = \text{Q}1.\overline{\text{Q}0}$$

8. Let us sketch the FSM Design

As discussed in the previous steps we need to have two flip-flops and logic gates to implement the synchronous gray counter.

As shown in Fig. 15.13, in the gray counters only 1-bit changes in two successive count, hence reduces switching power. This reduces significant amount of power at the logic level as next state logic is not needed.

15.8 Dynamic Power Reduction

Use the clock gating technique using the clock gating cells to minimize the power for the design. Clock gating can be implemented by identifying the synchronous load enable register banks. Clock gating can be implemented by using the gating of clock with the enables instead of recirculating of the data when enable is inactive

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

The design without clock gating is shown in Fig. 15.14.

The above 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, it is useful to avoid

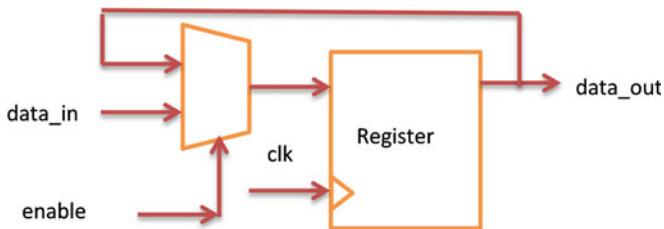


Fig. 15.14 Logic without clock gating

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. 15.15. The timing sequence is shown in Fig. 15.16.

In this chapter we have discussed about the few of the design performance improvement techniques at the architecture and at logic level. Next subsequent chapter is useful to understand about the case study of the speed improvement and pipelined controller design.

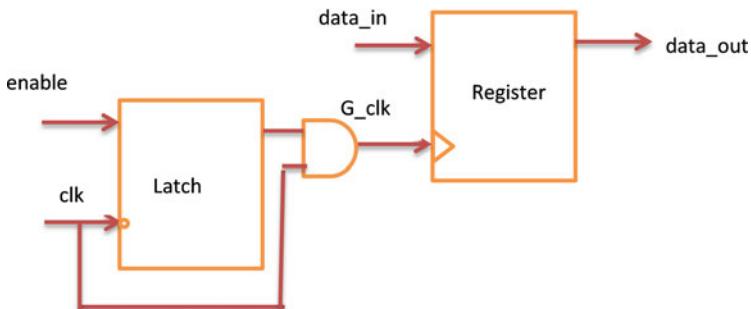


Fig. 15.15 Logic with clock gating

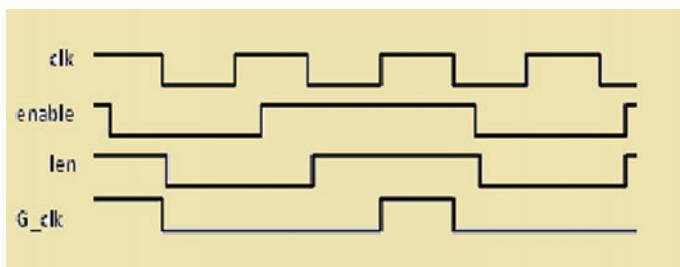


Fig. 15.16 Timing sequence for the clock gating

15.9 Summary

The following are few of the important points to conclude this chapter.

1. The arithmetic resource sharing is powerful technique to optimize the area.
2. Sometime the logic duplication is also used to optimize for the area but this technique is design and scenario specific
3. The subtraction can be performed as 2's complement addition to use common resource as adder.
4. All logic operations can be performed by using adders, multiplexers, and minimum number of logic gates.
5. The area can be optimized at the logic and architecture level.
6. The design should have the minimum area and minimum power.
7. The design area and power can be improved using the data and control path separation.
8. To have the lesser area in design avoid use of the parallel blocks of same type to perform the multiple operations concurrently.
9. Speed can be improved using pipelined stages.
10. The power can be optimized at logic level using the gray counters.
11. The clock gating is useful to reduce the dynamic power.

Chapter 16

Case Study: Speed Improvement for the Design



The speed of the design is one of the major optimization constraints and logic design team should consider various speed improvement techniques during the design.

In the previous chapters we have discussed about the performance improvement basics and area, speed and power improvement techniques. This chapter is useful to understand about the speed improvement case study and optimization techniques. The chapter is useful to understand the pipelining, register balancing and other techniques to improve the speed of design. The chapter also covers the case study of the pipelined processor and useful during the system design and product design using FPGA as a programmable logic.

16.1 Case Study: Speed Improvement at Logic-Level Case Study

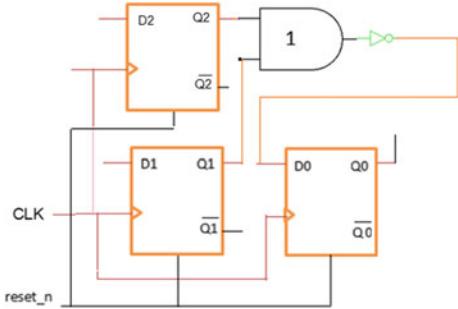
Let us discuss the speed improvement at the logic level. To find the maximum operating frequency for the design (Fig. 16.1) use the reg-to-reg timing path that is longest path in the design. In the design the start point is clk pin of two parallel flip-flops and endpoint is D0 of the LSB flip-flop.

Let us consider $t_{\text{pff}} = 1 \text{ ns}$, $t_{\text{combo}} = t_{\text{not}} = \text{propagation delay of NoT gate} = 1 \text{ ns}$, $t_{\text{and}} = \text{propagation delay of AND gate} = 1 \text{ ns}$ and $t_{\text{su}} = \text{setup time} = 1 \text{ ns}$ and $t_h = \text{hold time} = 0.5 \text{ ns}$.

1. Let us find out the data arrival time (AT). $\text{AT} = t_{\text{pdff1}} + t_{\text{and}} + t_{\text{not}}$.
2. The data required time is RT. $\text{RT} = T_{\text{clk}} - t_{\text{su}}$ because the data at D input should be stable by t_{su} margin prior to arrival of rising edge of the clock.
3. Now find setup slack. $\text{Slack} = \text{RT} - \text{AT}$ and should be greater than or equal to 0.
4. $\text{Slack} = \text{RT} - \text{AT}$

$$\text{Setup Slack} = (T_{\text{clk}} - t_{\text{su}}) - (t_{\text{pdff1}} + t_{\text{and}} + t_{\text{not}}).$$

Fig. 16.1 Design having speed issues



5. If we equate the slack to zero then we get

$$0 = (T_{\text{clk}} - t_{\text{su}}) - (t_{\text{pdff1}} + t_{\text{and}} + t_{\text{not}})$$

$$T_{\text{clk}} = t_{\text{pdff1}} + t_{\text{and}} + t_{\text{not}} + t_{\text{su}}$$

6. The maximum operating frequency of the design is f_{max} .

$$\begin{aligned} f_{\text{max}} &= \frac{1}{T_{\text{clk}}} \\ &= \frac{1}{t_{\text{pdff1}} + t_{\text{and}} + t_{\text{not}} + t_{\text{su}}} \\ &= \frac{1}{1\text{ns} + 1\text{ns} + 1\text{ns} + 1\text{ns}} \\ &= \frac{1}{4\text{ns}} \\ &= 250.00\text{MHz} \end{aligned}$$

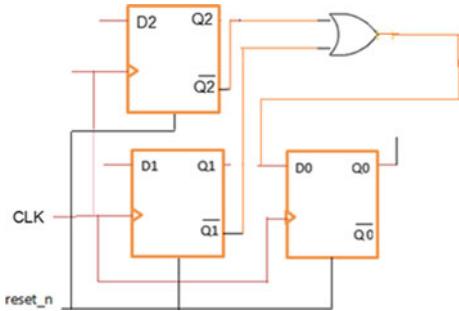
Design with improved performance after logic tweaks.

Now let us use De'morgens law to replace AND followed by NOT (NAND gate) using bubbled OR gate. The design after logic tweaks is shown in Fig. 16.2.

Let us consider $t_{\text{pff}} = 1$ ns, $t_{\text{or}} = \text{propagation delay of OR gate} = 1$ ns and $t_{\text{su}} = \text{setup time} = 1$ ns and $t_h = \text{hold time} = 0.5$ ns.

1. Let us find out the data arrival time (AT). $\text{AT} = t_{\text{pdff1}} + t_{\text{or}}$.
2. The data required time is RT. $\text{RT} = T_{\text{clk}} - t_{\text{su}}$ because the data at D input should be stable by t_{su} margin prior to arrival of rising edge of the clock.
3. Now find setup slack. $\text{Slack} = \text{RT} - \text{AT}$ and should be greater than or equal to 0.
4. $\text{Slack} = \text{RT} - \text{AT}$
Setup Slack = $(T_{\text{clk}} - t_{\text{su}}) - (t_{\text{pdff1}} + t_{\text{or}})$.
5. If we equate the slack to zero then we get

Fig. 16.2 Design with improved speed



$$0 = (T_{\text{clk}} - t_{\text{su}}) - (t_{\text{pdff1}} + t_{\text{or}})$$

$$T_{\text{clk}} = t_{\text{pdff1}} + t_{\text{or}} + t_{\text{su}}$$

6. The maximum operating frequency of the design is f_{\max}

$$\begin{aligned} f_{\max} &= \frac{1}{T_{\text{clk}}} \\ &= \frac{1}{t_{\text{pdff1}} + t_{\text{or}} + t_{\text{su}}} \\ &= \frac{1}{1\text{ns} + 1\text{ns} + 1\text{ns}} \\ &= \frac{1}{3\text{ns}} \\ &= 333.33\text{MHz} \end{aligned}$$

16.2 Speed Improvement at Architecture Level (Source: www.onerupeest.com)

Let us discuss the processor design with multistage pipelining architecture. Consider the processor design to perform the arithmetic and logical operations and has two operands. The processor is of RISC type, and the following are the specifications.

1. Processor performs 16 arithmetic and logical instructions.
2. Processor has 4-bit opcode.
3. The processor has the data bus = 16 bit or 32-bit depending on the configuration parameters.
4. Processor uses two operands.
5. Processor has the internal register array of 4 registers.
6. For improved performance processor should use the 4-stage pipelined architecture.
7. Processor has 8-bit flag register.
8. Processor should store the result in external memory of 256 bytes.

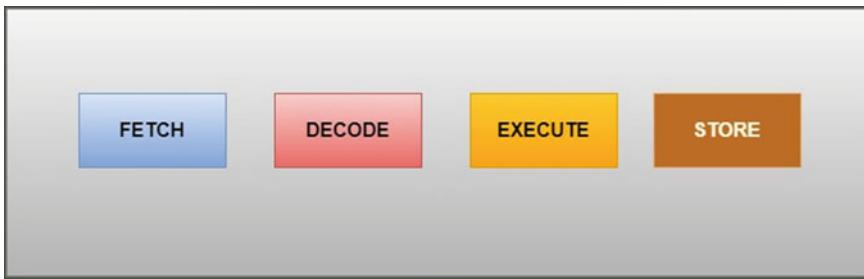


Fig. 16.3 Pipelining units

9. The processor is single clock domain design and operates at clock frequency of 250 MHz.
10. The processor uses synchronous active low reset.

By considering the details of arithmetic and logical operations we can design the execution unit that is ALU of the pipelined processor.

The top-level pipelined stages are shown in Fig. 16.3.

Consider the execution of four consecutive arithmetic or logical instructions. If each instruction needs four clock cycles then without pipelined architecture processor needs 16 clock cycles.

If the pipelined stages are used then the multitasking can be done and Table 16.1 gives information of the fetch, decode, execute and store of the four consecutive instructions.

As shown in table, while fetching the instruction 2 the first instruction is decoded, and so on. Thus, it reduces the number of cycles, and hence, the processor architecture has improved performance.

Table 16.1 Pipelining of instructions

Instruction	Fetch	Decode	Execute	Store
1	1	x	x	x
2	2	1	x	x
3	3	2	1	x
4	4	3	2	1

16.2.1 Top-Level Pin Interface

Using the specifications, we can think about the top-level architecture having following units.

1. Fetch block
2. Decode block
3. Execute block
4. Store block
5. Flag register logic.

The top-level interfaces needed for the four-stage pipelined controller are shown in Fig. 16.4.

16.2.2 Pin Description

The pin description is documented in Table 16.2, and the DATA_SIZE parameter we can configure depending on the requirement to 8,16,32, etc. The ADDR_SIZE we can configure to 8,9,10, and so on depending on the need of the external memory.

16.2.3 Case Study: Micro-architecture Design

The micro-architecture of the processor is sub-block-level representation and has main important blocks with their interfaces. The main important blocks of the architecture with their interfaces are discussed in the section below.

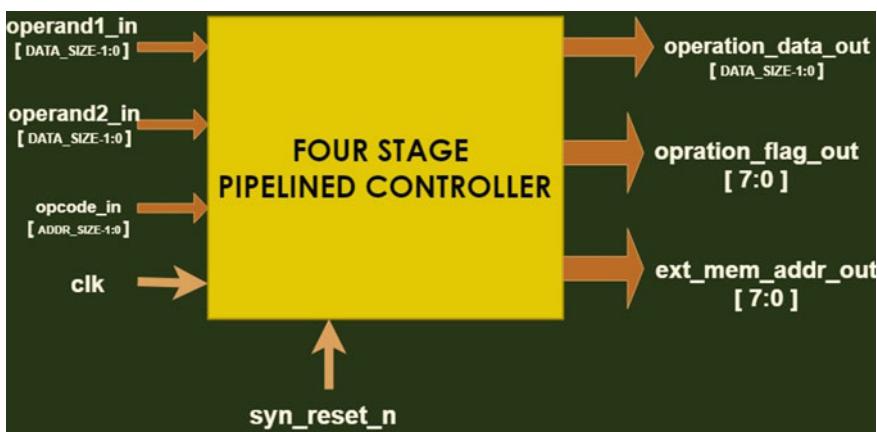
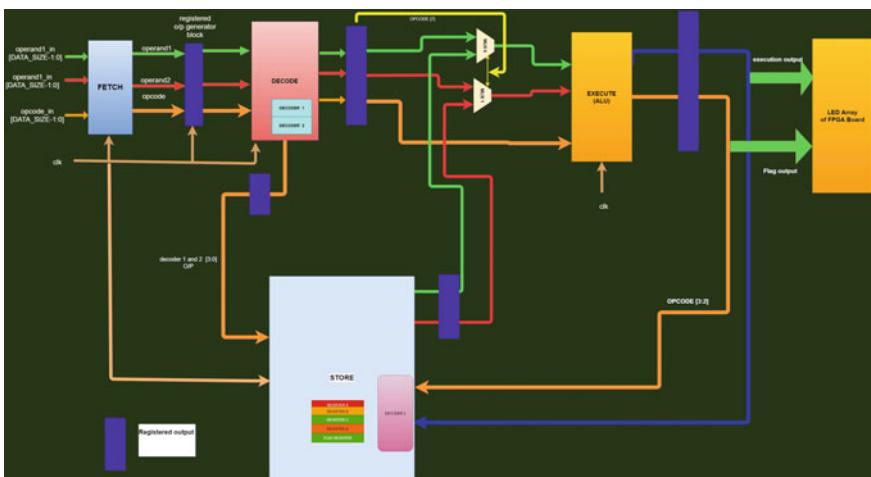


Fig. 16.4 Four-stage pipelined controller top-level interfaces

Table 16.2 Pin description of processor

Sr. no	Signal name	I/O	Width	PIN description
1	Operand I	1	DATA_SIZE	First operand input data
2	Operand 2	1	DATA_SIZE	Second operand input data
3	Opcode	1	ADDR_SIZE	Opcode of instruction
4	Clock	1	1	Clock input
5	Reset	1	1	Reset input
6	Operation output	0	DATA_SIZE	Output data after execution
7	Flag output	0	8	Flag output to indicate overflow, etc.
8	Address output for External memory	0	ADDR_SIZE	Address line to address external memory (“It is currently not included in RTL design”)

**Fig. 16.5** Micro-architecture of the pipelined processor

The four-stage pipelined processor micro-architecture is shown in Fig. 16.5. The pipelined registers are denoted by using violet boxes.

16.2.3.1 Fetch Block

The fetch block is used to fetch the opcode and operands needed for the instruction execution. The pin description of the interface signals is documented in Table 16.3.

As discussed earlier, the `ADDR_SIZE` and `DATA_SIZE` are parameterized to have the design according to required data and address size.

Table 16.3 Interface signals of fetch block

Sr. no	Signal name	I/O	Width	PIN description
1	Operand one input	1	DATA_SIZE	First operand input data
2	Operand two input	1	DATA_SIZE	Second operand input data
3	Opcode	1	ADDR_SIZE	Opcode of Instruction
4	Clock	1	1	Clock input
5	Reset	1	1	Synchronous active low reset input
6	Operand one fetch output	0	DATA_SIZE	First operand fetch output data
7	Operand two fetch output	0	DATAJSIZE	Second operand fetch output data
8	Opcode fetch output	0	ADDR_SIZE	Fetch opcode output data

16.2.3.2 Decode Block

The decode block is used to decode the opcode to identify the type of instruction and this block uses the opcode fetched by fetch logic. The decode block interface signals are documented in Table 16.4. The output from decode is used by the register array and the execute block.

Table 16.4 Decode block interface signals

Sr. no	Signal name	I/O	Width	PIN description
1	Operand one input from fetch	1	DATA_SIZE	First operand input data from fetch output
2	Operand two input from fetch	1	DATA_SIZE	Second operand input data from fetch output
3	Opcode from fetch	1	ADDR_SIZE	Opcode of Instruction from fetch output
4	Clock	1	1	Clock input
5	Reset	1	1	Synchronous active low reset input
6	Opcode for ALU/ execution unit	0	ADDR_SIZE	First operand fetch output data
7	Operand one output decoder	0	DATA_SIZE	Operand one output from decoder
8	Operand two output decoder	0	DATA_SIZE	Operand two output from decoder
9	Decoder output to select the Reg. A, B, C, and D	0	8	Decoder output is used to select the register

Table 16.5 Interface signals of execute block

Sr. no	Signal name	I/O	Width	PIN description
1	Operand one input	1	DATA_SIZE	First operand input data from mux logic
2	Operand two input	1	DATA_SIZE	Second operand input data from mux logic
3	Opcode	1	ADDR_SIZE	Opcode of instruction from decode output
4	Clock	1	1	Clock input
5	Reset	1	1	Synchronous active low reset input
6	ALU/execution block output	0	DATA_SIZE	Output data after execution
7	Flag output	0	8	Flag output to indicate overflow
8	Opcode out form ALU/execution unit	0	ADDR_SIZE	Opcode out form ALU/execution unit for register selection

16.2.3.3 Execution Block

The execution block uses operands depending on the type of instruction and performs the execution of either arithmetic or logical operations and generates an output. The interface signals of the execute block are documented in Table 16.5.

16.2.3.4 Store Block

The store block is register array and has the four registers and used to store the results computed by the execute block. The read and write operations are permitted with the store block. The interface signals of the store block are documented in Table 16.6.

16.2.3.5 Flag Register

The processor architecture has support to update the five flags. The pin description and flag description are documented in Table 16.7.

Depending on the design requirements the readers are encouraged to tweak the architecture to design the pipelined processor logic using VHDL, Verilog or SystemVerilog synthesizable constructs. For the basics of the FPGA architecture and Verilog HDL please refer Chaps. 15 and 16.

In this chapter we have discussed the speed improvement techniques and the four-stage pipelined controller top-level architecture and micro-architecture design. The next subsequent chapter focuses on the multiple clock domain designs.

Table 16.6 Interface signals of the store block

Sr. no	Signal name	I/O	Width	PIN description
1	Register select signal from decoder	1	REG_SIZE	Signal to select the register A, B, C, D for read operation
2	Register select signal from execution block	1	DATA_SIZE	Signal to select the register A, B, C, D for write operation
3	Clock	1	1	Clock input
4	Reset	1	1	Synchronous active low reset input
5	Operand one from store block	0	DATA_SIZE	Operand one output Example: ADD A,B; Reg A value out
6	Operand two from store block	0	DATA_SIZE	Operand two output Example: ADD A,B; Reg B value out

Table 16.7 Flag register

Sr. no	Signal name	I/O	Width	PIN description
1	Sign flag (S)	0	1	1: Negative 0: Positive
2	Zero flag (Z)	0	1	1: Zero result 0: Nonzero result
3	Auxiliary carry flag (AC) (Not included in current RTL design)	0	1	This flag is used in BCD number system (0–9). If after any arithmetic or logical operation D(3) generates any carry and passes on to B(4) this flag becomes set 1; otherwise, it becomes reset 0
4	Parity flag (P)	0	1	1: Even 0: Odd
5	Carry flag (CY)	0	1	1-carry out from MSB bit on addition or borrow into MSB bit on subtraction 0-no carry out or borrow into MSB bit

16.3 Summary

The following are few of the important points to conclude this chapter.

1. The speed is one of the important optimization constraints.
2. The speed can be improved using register balancing technique at the cost of additional register or registers.
3. The pipelining is used to improve the speed of the design.

4. The pipelined processors can be designed by deploying the fetch, decode, execute, and store stages.
5. The performance can be improved using logic tweaks.
6. The performance can be improved using the tweaks at architecture level.

Chapter 17

Case Study: Multiple Clock Domains and FIFO Architecture Design



The multiple clock domain designs and understanding of the synchronizers plays important role during architecture and system design.

In the previous chapters we have discussed about the performance improvement techniques and optimization techniques. From VLSI perspective let us discuss about the multiple clock domain designs. The chapter focuses on the data and control path synchronizers and also discusses about the FIFO design case study.

17.1 Single Clock Domain Designs

In the single clock domain design, each functional block uses the clock generated using the same clock source PLL. At the system level design if the speed of each design component is same, then each functional block is clocked by the same clock signal. Hence the design has single clock domain.

Consider the 16-bit processor top-level architecture. For this each functional block uses the single clock. Hence there is no issue of valid data and data convergence. The single clock domain design of the pipelined processor is shown in Fig. 17.1.

17.2 Multiple Clock Domain Designs

Most of the time we have the multiple clock domain designs. Consider the design which has the general-purpose processor, video encoder/decoder, and memory controllers. The general-purpose processor operates on $\text{clk1} = 500 \text{ MHz}$, the video encoder/decoder operates on the $\text{clk2} = 250 \text{ MHz}$, and the memory controller operates at the frequency of $\text{clk3} = 333.33 \text{ MHz}$. So, we need to have three different clock sources, and such type of the design is called as the multiple clock domain design (Fig. 17.2).

Fig. 17.1 Single clock domain processor top-level architecture

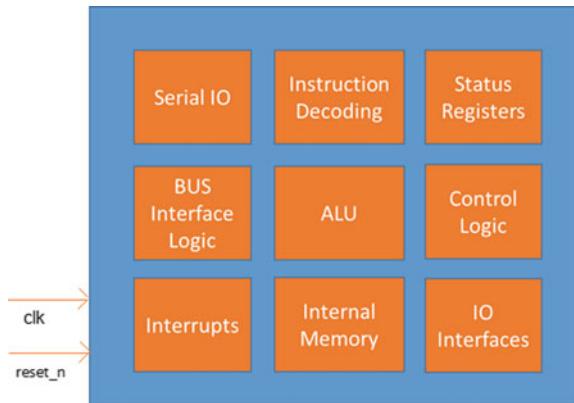
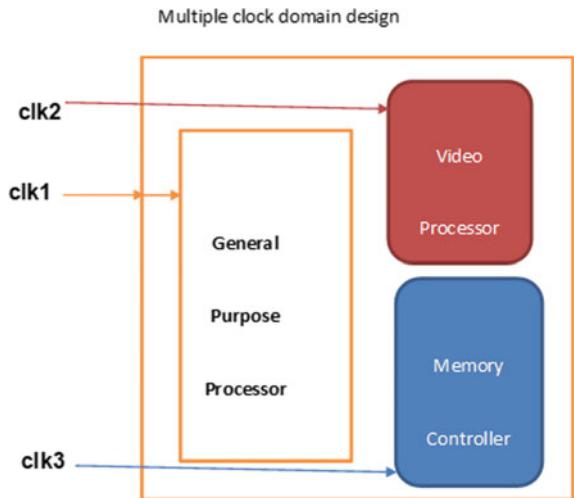


Fig. 17.2 Multiple clock domain design



What are issues in such designs?

In the multiple clock domain designs the major issue is exchange of the data between the clock domains. There is issue of the data convergence and can be overcome by using the control and data path synchronizers.

We can think of using the level synchronizers in the control path and FIFO synchronizers in the data path. The following section discusses about the issues in the multiple clock domain designs!

Fig. 17.3 Level synchronization concept

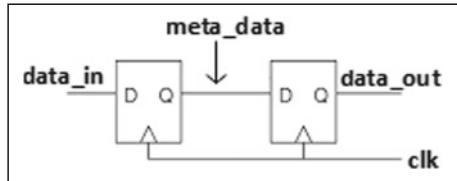
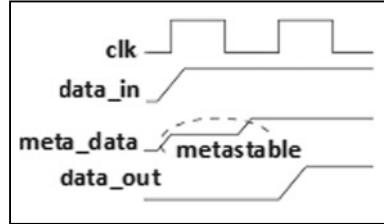


Fig. 17.4 The timing sequence for Fig. 17.3



17.3 Metastability

Consider the design shown in Fig. 17.3. The data_in is input of the design and sampled on the rising edge of the clock. If the other design which works at different clock frequency drives the data_in, then the design has multiple clock boundaries. In such scenario due to phase difference between the multiple clocks the first flip-flop (Fig. 17.3) goes into the metastable state. The meta_data indicates the flip-flop output is metastable, and hence there is timing violation for the first flip-flop.

The metastability indicates the data output is not valid and to get the valid data output the design need to use the multiflop level synchronizers.

The timing sequence is shown in Fig. 17.4. As shown the output of the first flip-flop is in the metastable state and the data_out output from the output flip-flop is having the valid legal state.

17.4 Control Path Synchronizer

As discussed in the previous section we can think of using the level synchronizers in the control path. For the level synchronizer the first flip-flop output is metastable and should be ignored by setting the false path in the timing analysis. The control path synchronizer which uses two flip-flops in cascade is shown in Fig. 17.5.

There are various other synchronization techniques to pass the signal between the multiple clock domains. Few of them are mux synchronizers and pulse synchronizers.

Consider the use of the level synchronizer in the control path. Consider the clock domain 1 operates at 100 MHz and clock domain 2 at 50 MHz. To pass the control signals from one of the clock domains to another clock domain we can think of use of the level synchronizers (Fig. 17.6).



Fig. 17.5 The two-flop level synchronizer

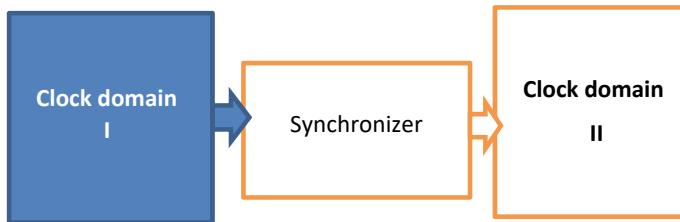


Fig. 17.6 Passing the control signal between the clock domains

17.5 Data Path Synchronizers

As discussed in the above section to pass the multibit 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 but due to skew between the multiple clock signals the synchronization cannot be achieved. So, for the multibit 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 multibit data, and these are used in the practical ASIC designs. These techniques are as follows:

- (a) Handshaking mechanism
- (b) FIFO memory buffers.

For the ASIC and FPGA-based designs you can use these mechanisms to carry data between the clock domains. The objective of this manuscript is to have understanding of the data path synchronizer that is FIFO and architecture of FIFO in VLSI perspective.

17.5.1 Why We Need FIFO?

In the multiple clock domain designs as a data path synchronizer, we need asynchronous FIFO. If the sender clock domain is faster and receiver clock domain is

slower then to avoid loss of data, we always deploy the FIFO of finite depth. Now let us discuss how to find the depth of FIFO?

Consider the design scenario where design having write clk frequency $f_1 = 100$ MHz and read clock frequency $f_2 = 50$ MHz and the burst length is 100 byte. Let us find the depth of FIFO.

Let us use the following steps to find the depth of the FIFO.

1. **Time required to write single data byte (T_w)**

$$T_w = 1/100 \text{ MHz} = 10 \text{nsec}$$

2. **Time required to write burst of the data that is 100 bytes (T_{b_w})**

$$\begin{aligned} T_{b_w} &= T_w * \text{Burst length} \\ &= 10 \text{nsec} * 100 \\ &= 1000 \text{ nsec} \end{aligned}$$

3. **Time required to read one data (T_r)**

$$\begin{aligned} T_r &= 1/50 \text{ MHz} \\ &= 20 \text{ nsec} \end{aligned}$$

4. **The number of data reads in duration of T_{b_w}**

$$\begin{aligned} \text{No of reads} &= 1000 \text{ nsec}/20 \text{ nsec} \\ &= 50 \end{aligned}$$

5. **The depth of FIFO.**

$$\begin{aligned} \text{Depth of FIFO} &= \text{Burst length} - \text{No of reads} \\ &= 100 - 50 \\ &= 50 \end{aligned}$$

17.5.2 FIFO Depth Calculation

Let us consider few examples on the FIFO depth calculation.

17.5.2.1 Scenario 1: Idle Cycles Between Writes and Reads

Let us consider the design having write clk frequency $f_1 = 100$ MHz and read clock frequency $f_2 = 50$ MHz and the burst length is 100 bytes. The number of idle cycles between two writes is 1, and number of idle cycle between two reads is 3. Let us find the depth of the FIFO.

To find the depth of the FIFO use the following steps.

1. Time required to write single data byte (T_w)

One idle cycle between two writes so for two cycles one data is written.

$$\begin{aligned} T_w &= 2 * (1/100 \text{ MHz}) \\ &= 20 \text{ nsec} \end{aligned}$$

2. Time required to write burst of the data that is 100 bytes (T_{b_w})

$$\begin{aligned} T_{b_w} &= T_w * \text{Burst length} \\ &= 20 \text{ nsec} * 100 \\ &= 2000 \text{ nsec} \end{aligned}$$

3. Time required to read one data (T_r)

For two successive reads three cycles (3 idle + 1 = 4 cycles)

$$\begin{aligned} T_r &= 4 * (1/50 \text{ MHz}) \\ &= 80 \text{ nsec} \end{aligned}$$

4. The number of data reads in duration of T_{b_w}

$$\begin{aligned} \text{No of reads} &= 2000 \text{ nsec}/80 \text{ nsec} \\ &= 25 \end{aligned}$$

5. The depth of FIFO.

$$\begin{aligned} \text{Depth of FIFO} &= \text{Burst length} - \text{No of reads} \\ &= 100 - 25 \\ &= 75 \end{aligned}$$

17.5.2.2 Scenario 2: Idle Cycles Between Writes and Reads

Consider the design having write clk frequency $f_1 = 50$ MHz and read clock frequency $f_2 = 100$ MHz and the burst length is 50 bytes. The number of idle

cycles between two writes is 1, and number of idle cycle between two reads is 3. Let us find the depth of FIFO.

To find the depth of the FIFO use the following steps.

1. Time required to write single data byte (T_w)

One idle cycle between two writes so for two cycles one data is written.

$$\begin{aligned} T_w &= 2 * (1/50 \text{ MHz}) \\ &= 40 \text{ nsec} \end{aligned}$$

2. Time required to write burst of the data that is 50 bytes (T_{b_w})

$$\begin{aligned} T_{b_w} &= T_w * \text{Burst length} \\ &= 40 \text{ nsec} * 50 \\ &= 2000 \text{ nsec} \end{aligned}$$

3. Time required to read one data (T_r)

For two successive reads three cycles (3 idle + 1 = 3 cycles)

$$\begin{aligned} T_r &= 4 * (1/100 \text{ MHz}) \\ &= 40 \text{ nsec} \end{aligned}$$

4. The number of data reads in duration of T_{b_w}

$$\begin{aligned} \text{No of reads} &= 2000 \text{ nsec}/40 \text{ nsec} \\ &= 50 \end{aligned}$$

5. The depth of FIFO.

$$\begin{aligned} \text{Depth of FIFO} &= \text{Burst length} - \text{No of reads} \\ &= 50 - 50 \\ &= 0 \text{ NO FIFO required} \end{aligned}$$

17.5.2.3 Scenario 3: Idle Cycles Between Writes and Reads

Consider the design having write clk frequency $f1 = 50 \text{ MHz}$ and read clock frequency $f2 = 50 \text{ MHz}$ and the burst length is 100 bytes. The number of idle cycles between two writes is 1, and number of idle cycle between two reads is 3. Let us find the depth of FIFO?

To find the depth of the FIFO use the following steps.

1. Time required to write single data byte (T_w)

One idle cycle between two writes so for two cycles one data is written.

$$\begin{aligned} T_w &= 2 * (1/50 \text{ MHz}) \\ &= 40 \text{ nsec} \end{aligned}$$

2. Time required to write burst of the data that is 80 bytes (T_{b_w})

$$\begin{aligned} T_{b_w} &= T_w * \text{Burst length} \\ &= 40 \text{ nsec} * 100 \\ &= 4000 \text{ nsec} \end{aligned}$$

3. Time required to read one data (T_r)

For two successive reads three cycles (3 idle + 1 = 4 cycles)

$$\begin{aligned} T_r &= 4 * (1/50 \text{ MHz}) \\ &= 80 \text{ nsec} \end{aligned}$$

4. The number of data reads in duration of T_{b_w}

$$\begin{aligned} \text{No of reads} &= 4000 \text{ nsec}/80 \text{ nsec} \\ &= 50 \end{aligned}$$

5. The depth of FIFO.

$$\begin{aligned} \text{Depth of FIFO} &= \text{Burst length} - \text{No of reads} \\ &= 100 - 50 \\ &= 50 \end{aligned}$$

17.5.3 Case Study: FIFO as a Data Path Synchronizer

In the system designs or FPGA/ASIC designs, the FIFO memory buffers are used as data path synchronizer to pass the data between multiple clock domains. The sender 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 by using the read_clk.

So basically, FIFO consists of the memory buffer that is dual port RAM, write control logic, read control logic, and the synchronizers with the empty and full flag generation logic. The FIFO having various functional blocks is shown in Fig. 17.7.

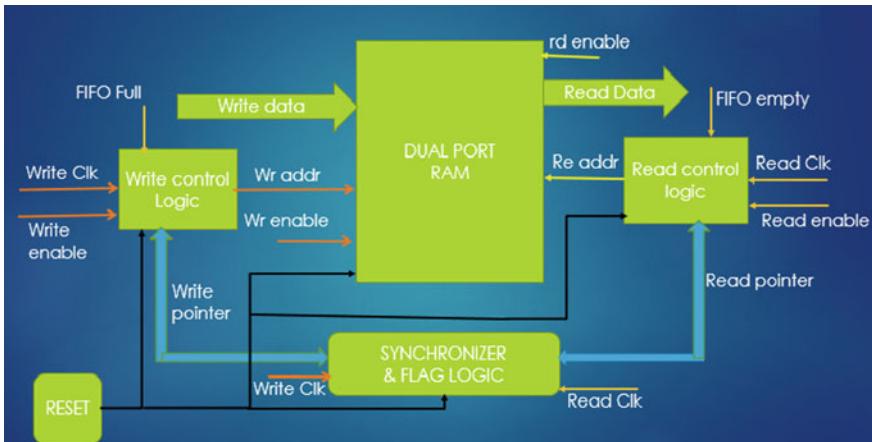


Fig. 17.7 Architecture of FIFO

17.5.4 Micro-architecture of FIFO

The top-level pin interface of FIFO is shown in Fig. 17.8.

The pin description is documented in Table 17.1.

The top-level signal names are documented in Table 17.2.

The important blocks of FIFO are as follows:

1. Dual port RAM
2. Write control logic
3. Read control logic
4. Level synchronizers
5. Gray pointers
6. Empty flag logic
7. Full flag logic.

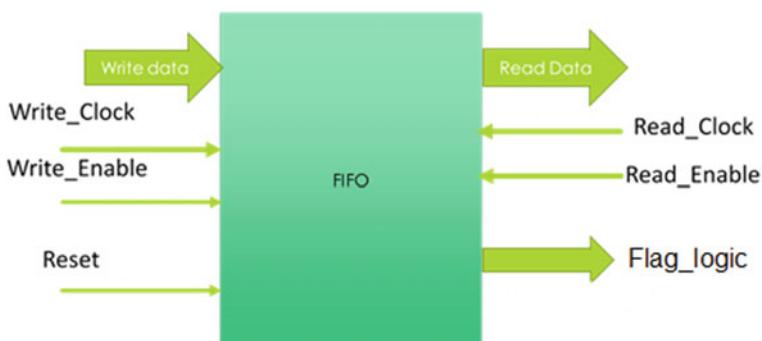


Fig. 17.8 FIFO top-level pin diagram

Table 17.1 Top-level pin description of FIFO signals

Sr. no	Signal name	I/O	Width	PIN description
1	Write data	1	DATA_SIZE-1	Input data to be written to FIFO
2	Read data	0	DATA_SIZE-1	Data out from FIFO
3	Write clock	1	1	Clock to write the data into FIFO
4	Read clock	1	1	Clock to read data from FIFO
5	Reset	1	1	Reset the value of FIFO to zero
6	Write enable	1	1	Enable the write operation to FIFO
7	Read enable	1	1	Enable read from the FIFO
8	FIFO Full	0	1	Indicate FIFO is full
9	FIFO empty	0	1	Indicate FIFO is empty
10	FIFO almost full	0	1	Indicate FIFO is almost full
11	FIFO almost empty	0	1	Indicate FIFO is almost empty

Table 17.2 Top-level FIFO signal names

Sr. no	Signal name	I/O	Width	Top-level signal
1	Write data	1	DATA_SIZE-1	wr_data
2	Read data	0	DATA_SIZE-1	rd_data
3	Write clock	1	1	wr_clk
4	Read clock	1	1	rd_clk
5	Reset	1	1	reset
6	Write enable	1	1	wr_en
7	Read enable	1	1	rd_en
8	FIFO Full	0	1	full
9	FIFO empty	0	1	empty
10	FIFO almost full	0	1	almost_full
11	FIFO almost empty	0	1	almost_empty

The micro-architecture of the FIFO is sketched by considering the depth of FIFO as 50 and for the detail Verilog or SystemVerilog designs please refer books related to the hardware description languages.

Here the micro-architecture is designed by considering the sender clock domain is faster as compared to receiver clock domain.

The micro-architecture is shown in Fig. 17.9.

The pin description of each signal is documented in Table 17.3.

Readers can use the micro-architecture sketched and the top-level pin description to design the FIFO sub-blocks. This can give the good understanding of the multiple clock domain designs and system level designs. For FPGA and ASIS designs you can refer the relevant manuscripts as the discussion on ASIC/FPGA design and implementation is out of scope in this chapter.

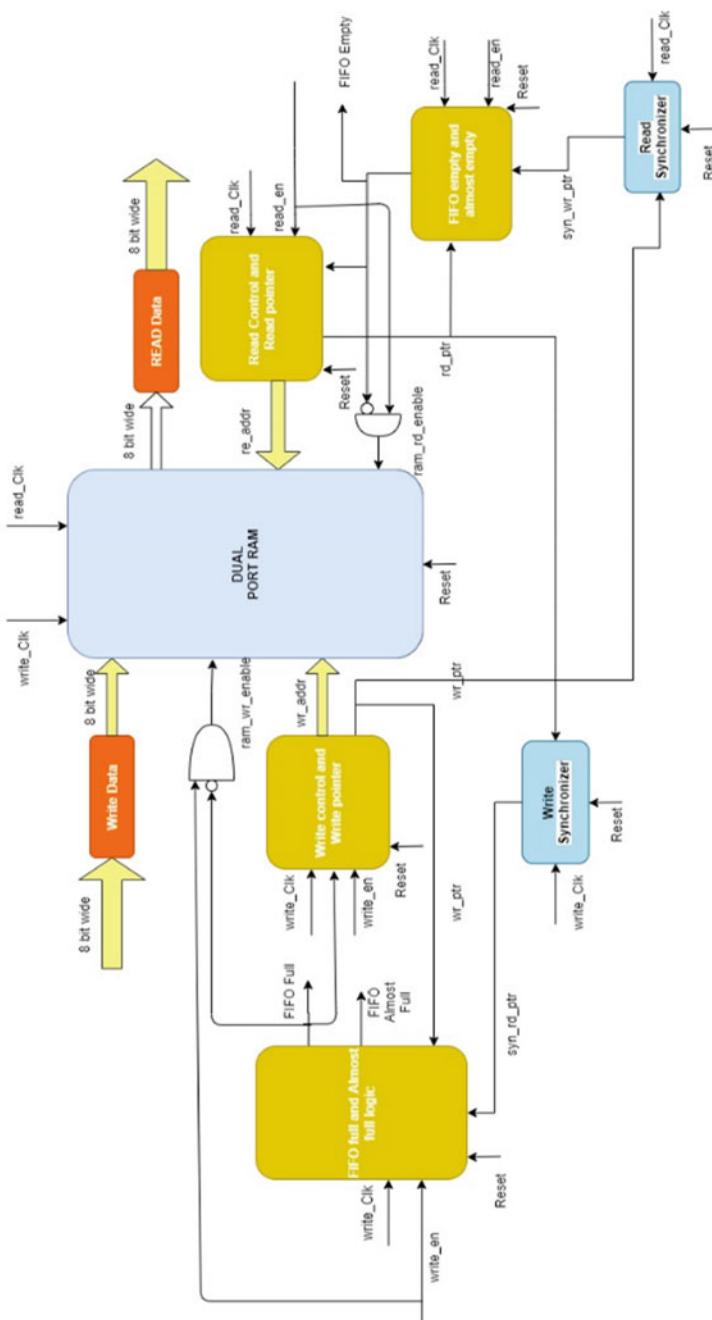


Fig. 17.9 FIFO micro-architecture. Source www.onerupeest.com

Table 17.3 Pin description

Sr. no	Signal name	I/O	Width	PIN Description
1	Write data	1	DATA_SIZE-1	Input data to be written to FIFO
2	Read data	0	DATA_SIZE-1	Data out from FIFO
3	Write clock	1	1	Clock to write the data into FIFO
4	Read clock	1	1	Clock to read data from FIFO
5	Reset	1	1	Reset the value of FIFO to zero
6	Write address	1	ADDR SIZE-1:0	Address in which data will be written
7	Read address	1	ADDR SIZE-1:0	Address from which data will be read
8	RAM write enable	1	1	Enable RAM write
9	RAM read enable	1	1	Enable RAM read
10	FIFO full	0	1	indicate FIFO is full
11	FIFO empty	0	1	Indicate FIFO is empty

17.6 Design Guidelines

During the system design use the following guidelines if the design has multiple clocks.

- Level synchronizers:** Avoid metastability by using the level synchronizer while passing the control signals between the clock domains.
- Data path synchronizers:** Use the FIFO as data path synchronizers while passing the data between the multiple clock domains.
- Partitioning of design:** Partition the design depending on the clock groups and have the registered inputs and registered outputs at each clock boundary.
- Use gray code counters:** In most of the ASIC designs having CDC boundaries, 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.
- Clock naming conventions:** It is recommended to use the clock naming conventions to identify the clock domains and the clock sources. 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.
- Reset synchronization:** 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 states of flip-flops.

7. **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 as follows:
 - a. Multiple bits on the bus
 - b. Multiple handshake signals
 - c. Unrelated signals.

To avoid this, use the clock intent verification technique this technique will ensure the passing of multibit signal across the clock boundaries.

In this chapter we have discussed about the control path and data path synchronizers which need to be deployed in the multiple clock domain designs. The next chapter is useful to understand about the basics of the RTL design using Verilog.

17.7 Summary

Following are important points to conclude this chapter.

1. 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.
2. Use the multistage synchronizers in the receiving clock domain while sampling the single-bit control signals at the receiver clock domain.
3. While passing the multiple control or data signals from one of the clock domain to another clock domain, use the consolidated control signal that is one-bit representation of the multiple signals in the sending clock domain.
4. Use the multistage synchronizer in the receiver clock domain while sampling the consolidated control signals.
5. To pass multiple control signals across the clock domains use the MCP formulation.
6. Use the gray code counters instead of binary counters while passing the data across multiple clock domains.
7. Use FIFO in the data or control path while passing the multiple data bits or control bits.

Chapter 18

Hardware Description for Design



The role and use of the synthesizable constructs from Verilog and SystemVerilog is for the design.

In the previous chapters we have discussed about the logic design techniques, optimization, and performance improvements for the design. As the design architecture phase is completed the design should be coded using synthesizable Verilog or SystemVerilog constructs. In this context the chapter discusses about the basics of the hardware description and the role of the hardware description languages to have the efficient VLSI-based designs.

18.1 Verilog HDL

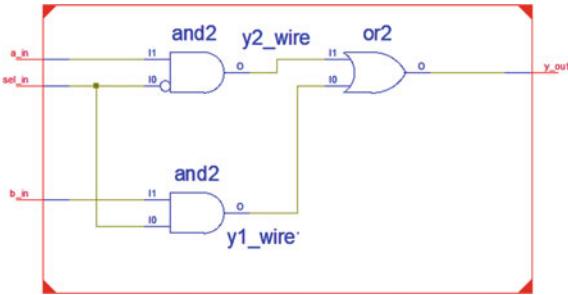
The Verilog-2005 has powerful synthesizable and non-synthesizable constructs, and these constructs can be used for the design and verifications, respectively. The hardware description using few of the synthesizable constructs is discussed in the subsequent sessions.

The powerful Verilog constructs are as follows:

1. The assign that is continuous assignments to model the combinational logic
2. always@* to model the combinational logic
3. The procedural block always @(posedge clk) to model the sequential logic which is sensitive to rising edge of the clock
4. The procedural block always @(negedge clk) to model the sequential logic which is sensitive to falling edge of the clock.

Verilog has other powerful synthesizable and non-synthesizable constructs, and readers are requested to refer the books on Verilog and SystemVerilog to have detail understanding and use of them.

Fig. 18.1 Schematic of 2:1 mux



18.2 Use of Continuous Assignments

The continuous assignments using **assign** are neither blocking nor non-blocking and used to model the combinational logic. The continuous assignments are updated in the active event queue. The event on the right-hand side activates the assignment, and the RHS side expression is executed and assigned to LHS.

Multiple continuous assignments execute concurrently. Example 18.1 describes the 2:1 multiplexer design using continuous assignments.

Example 18.1: The 2:1 Mux Using Multiple Concurrent Assignments

```
///////////////////////////////
//      Multiple assign construct and 2:1 mux for sel_in=1
//      y_out=b_in, sel_in=0 y_out=0
module mux_2to1 (
    input a_in, b_in, sel_in,
    output y_out
);
wire y1_wire, y2_wire;
assign y1_wire = sel_in & b_in;
assign y2_wire = ~sel_in & a_in;
assign y_out = y1_wire | y2_wire;
endmodule
/////////////////////////////
```

By default, the inputs and outputs are of **wire** type. The multiple continuous assignments execute concurrently to infer the combinational logic. The inferred logic RTL schematic is shown in Fig. 18.1.

18.3 The always Procedural Block

Verilog has powerful procedural block **always @()**. In the parenthesis (), we need to include the inputs to invoke the procedural block. This procedural block executes

always for any change in the event on the inputs. It is compulsory to include all the desired inputs to invoke the ***always*** procedural block.

Example 18.2 describes the combinational logic using procedural block ***always***.

Example 18.2: Combinational Design Using Always Procedural Block

```
///////////
// The always procedural block sensitive to a_in, b_in
module combo_logic (
    input a_in, b_in,
    output reg y_out
);
always @ (a_in, b_in)
    y_out = a_in & b_in ;
endmodule
///////////
```

To model the combinational logic, it is recommended to use the blocking assignments. The data type of output *y_out* should be declared as **reg** type as the assignment is within the procedural block. The ***always*** procedural block is sensitive to changes on *a_in* and *b_in*, and any event on these inputs invokes ***always*** procedural block. Example 18.2 infers the AND gate.

18.4 The Procedural Block always@*

The issue with the ***always*** procedural block with the sensitivity list is due to missing variable or inputs. If we have hundreds of variables or inputs and the combinational logic is function of them, then it is mandatory to include all these inputs or variables in the sensitivity list. If any of the variable or input is missing from the sensitivity list, then we will get simulation and synthesis mismatch.

So, it is recommended to use the ***always@**** procedural block where * indicates include all inputs or variables in the sensitivity list.

Example 18.3 describes the 4:1 mux using the ***always@**** procedural block.

Example 18.3: The RTL Design of 4:1 Mux

```
///////////
module mux_4to1 (
    input clk_1, clk_2, clk_3, clk_4,
    input [1:0] sel_in,
    output reg y_out
);
always @(*)
if (sel_in == 2'b00)
    y_out = clk_1 ;
else if (sel_in == 2'b01)
```

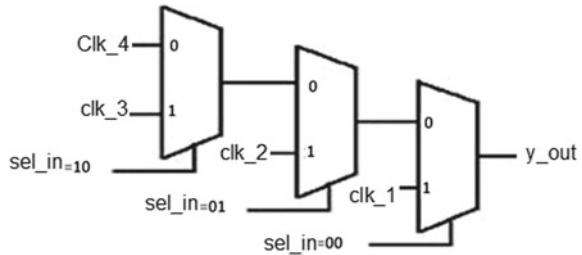
```

    y_out = clk_2 ;
else if (sel_in == 2'b10)
    y_out = clk_3 ;
else
    y_out = clk_4 ;
endmodule
///////////

```

The nested *if...else* construct infers the priority logic, and the RTL schematic is shown in Fig. 18.2.

Fig. 18.2 The RTL schematic of 4:1 mux



18.5 Use of the case Construct

As discussed earlier the nested *if...else* construct infers the priority logic and due to cascade stages the logic has larger delay. The Verilog has another powerful construct *case...endcase* and is recommended to code the parallel logic.

Example 18.4 describes the RTL design of 4:1 mux using *case* construct.

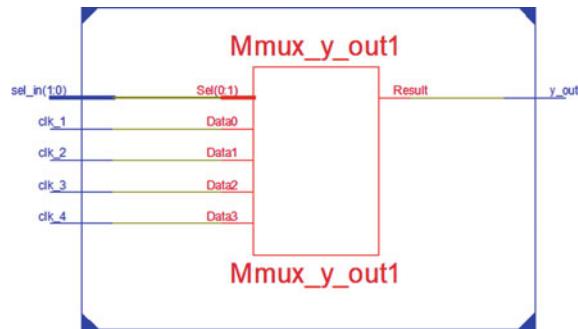
Example 18.4: The 4:1 Mux Parallel Logic

```

///////////
// The 4:1 mux using case construct
module combo_logic (
    input clk_1, clk_2, clk_3, clk_4,
    input [1:0] sel_in,
    output reg y_out
);
always @(*)
case (sel_in)
    2'b00 : y_out = clk_1;
    2'b01 : y_out = clk_2;
    2'b10 : y_out = clk_3;
    2'b11 : y_out = clk_4;
endcase
endmodule
/////////

```

Fig. 18.3 The 4:1 mux RTL schematic



The *case* construct infers the parallel logic, and the RTL schematic is shown in Fig. 18.3.

18.6 Continuous Versus Procedural Assignments

The Verilog is powerful HDL and has the continuous assignments and procedural assignments. The continuous assignments using `assign` are used to model the combinational logic.

The `always@*` procedural block is used to model the combinational logic.

The `always@(posedge clk)` is used to model the sequential logic which is sensitive to rising edge of the clk.

The `always@(negedge clk)` is used to model the sequential logic which is sensitive to falling edge of the clk.

To model the combinational logic using `always@*` and use the blocking assignments.

To model the sequential logic using `always@(posedge clk)` or by using `always@(negedge clk)` it is recommended to use the non-blocking assignments.

Continuous assignments: As discussed earlier the continuous assignments are used to assign the value to the net. These are used to code 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.

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

Procedural assignments: Procedural assignments are used to assign value to the `reg`. These are used to code both the combinational and sequential logic. These

assignments are used in the procedural blocks *always* and *initial* according to the requirements.

Example 18.5: Use of Assignments in the RTL Design

```
//////////  
always @(* posedge clk) // Sequential design description  
begin  
    q_out<= data_in;  
end  
always @(*) // Combinational design description  
begin  
    y_out = sel_in ? a_in : b_in;  
end  
//////////
```

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

18.7 Multiple Blocking Assignments Within the always Block

It is recommended to use the blocking (=) assignments to code the combinational logic. But if we use the multiple blocking assignments to code the sequential design then let us check what happens?

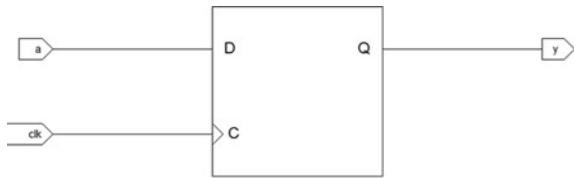
Consider the design scenario to infer the shift register, and we have described the RTL using the blocking assignment as shown in the Example 18.6. As blocking assignment blocks all the future assignments the logic infers the single flip-flop.

Example 18.6: Blocking Assignments in Always Block

```
//////////  
module blocking_assignment(  
    input a,  
    input clk,  
    output reg y );  
reg b,c;  
always @(* posedge clk)  
begin  
    b=a;  
    c=b;  
    y=c;  
end  
endmodule  
//////////
```

The RTL schematic is shown in Fig. 18.4, and as shown it infers the single flip-flop due to use of blocking assignments. Therefore it is recommended to use the non-blocking assignments while coding the RTL for the sequential logic.

Fig. 18.4 RTL schematic of Example 18.6



18.8 Design Scenario I: Blocking Assignments

Consider the reordering of the blocking assignments from Example 18.6. The RTL is described as shown in the Example 18.7, and designer intent is to create the serial input and serial output shift register. As the assignments are reordered and used within the *begin...end* it infers the serial input serial output shift register.

Example 18.7: Blocking Assignments in the Always Block (Ordering)

```
///////////////////////////////
module blocking_assignment(
    input a,
    input clk,
    output reg y );
reg b,c;
always @(posedge clk)
begin
    y=c;
    c=b;
    b=a;
end
endmodule
/////////////////////////////
```

RTL schematic is shown in Fig. 18.5, and it infers the serial input serial output shift register. So the important point to conclude is that order of the blocking assignments within the procedural *always* block is important factor and decides the logic inferred.

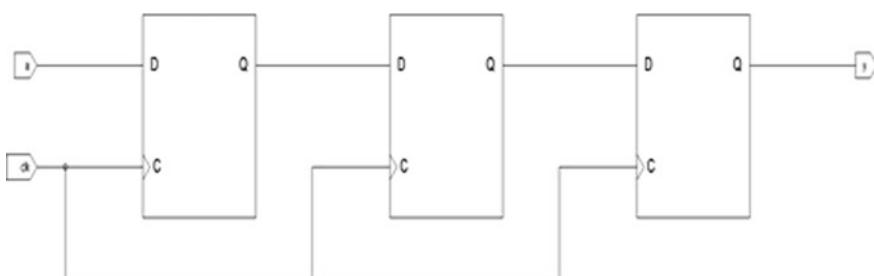


Fig. 18.5 RTL schematic of Example 18.7

18.9 Non-blocking Assignments

As discussed previously the blocking assignments(BA) are updated in the active queue and all the non-blocking assignments(NBA) are updated in the NBA queue. The blocking assignments within the begin–end executes sequentially, and all the non-blocking assignments executes in parallel within the *always* procedural block.

Let us consider the design to infer the shift register. The RTL is shown in the Example 18.8 and uses the NBA assignment. All the NBA assignments execute concurrently.

Example 18.8: Non-blocking Assignment in the Always Block

```
///////////////////////////////
module non_blocking_assignment (
    input a,
    input clk,
    output reg y );
    reg b, c;
    always @(posedge clk)
    begin
        y<=c;
        c<=b;
        b<=a;
    end
endmodule
/////////////////////////////
```

The RTL schematic is shown in Fig. 18.6, and it infers the serial input serial output shift register.

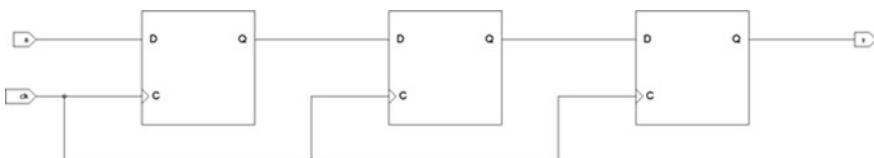


Fig. 18.6 RTL schematic of Example 18.8

18.10 Design Scenario II: Example Using Non-blocking Assignments

Let us consider reordering the assignments from Example 18.8. The RTL with reordered NBA is described in the Example 18.9, and intention is to create the serial input and serial output shift register.

Example 18.9: Non-blocking Assignment with Order Change in the Always Block

```
//////////  
module non_blocking_assignment(  
    input a,  
    input clk,  
    output reg y );  
  
reg b,c;  
always @(posedge clk)  
begin  
    b<=a;  
    c<=b;  
    y<=c;  
end  
endmodule  
//////////
```

The RTL schematic is shown in Fig. 18.6, and it infers the serial input serial output shift register. So the important point to conclude is that order of the non-blocking assignments within the procedural *always* block does not affect the synthesis result for these kind of designs!

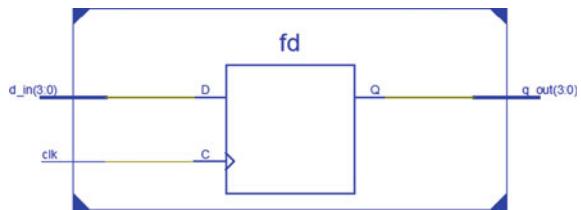
18.11 The 4-bit Register

Now let us discuss to code the RTL for 4-bit parallel in parallel out (PIPO) register. The RTL is shown in the Example 18.10.

Example 18.10 The RTL Design of 4-Bit PIPO Register

```
//////////  
//the 4-bit PIPO register  
module sequential_design (  
    input [3:0] d_in,  
    input clk,  
    output reg [3:0] q_out  
);  
always @(posedge clk)  
begin
```

Fig. 18.7 The RTL schematic of PIPO



```

q_out <= d_in;
end
endmodule
///////////////////////////////

```

The RTL schematic is shown in Fig. 18.7, and the inferred logic has four flip-flops.

18.12 Asynchronous Reset

Most of the time during the design we need to have the RTL design which has asynchronous reset. Refer Chap. 9 for more information about the asynchronous reset. The RTL shown in the Example 18.11 describes the 4-bit register sensitive to positive edge of the clock and having active low asynchronous reset.

Example 18.11: The RTL Design Having Asynchronous Reset

```

/////////////////////////////
// The 4-bit PIPO having asynchronous reset
module sequential_design (
    input [3:0] d_in,
    input clk, reset_n,
    output reg [3:0] q_out
);
always @(posedge clk, negedge reset_n)
begin
    if (~reset_n)
        q_out <= 4'b0000;
    else
        q_out <= d_in;
end
endmodule
/////////////////////////////

```

The RTL schematic is shown in Fig. 18.8, and the asynchronous reset logic have clean data path.

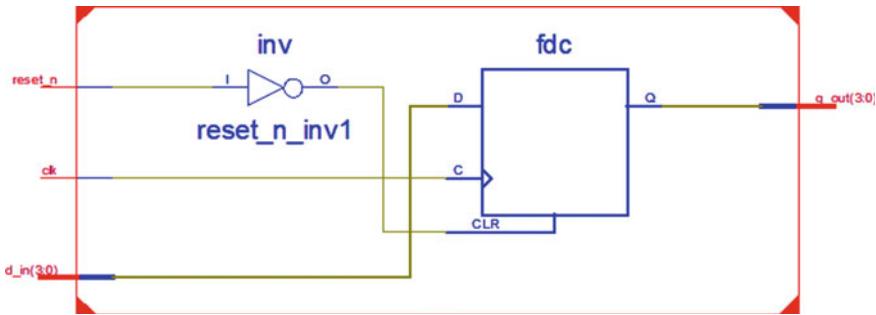


Fig. 18.8 The RTL schematic of Example 18.11

18.13 Synchronous Reset

The synchronous reset is sampled on the active edge of the clock, and the RTL shown in the Example 18.12 describes the 4-bit PIPO register which is sensitive to rising edge of the clock and having active low synchronous reset.

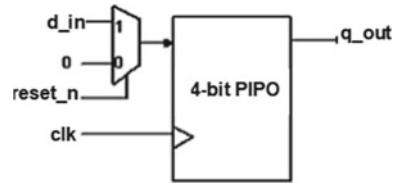
Example 18.12: RTL Design of PIPO Having Synchronous Reset

```
///////////
// The PIPO having synchronous reset.
module sequential_design (
    input [3:0] d_in,
    input clk, reset_n,
    output reg [3:0] q_out
);
always @ (posedge clk)
begin
    if (~reset_n)
        q_out <= 4'b0000;
    else
        q_out <= d_in;
end
endmodule
/////////
```

The logic inferred for the Example 18.12 is shown in Fig. 18.9, and as shown the design has extra reset logic in the data path.

In this chapter we have discussed about how to code the RTL using Verilog and important Verilog constructs. The next chapter focuses on the FPGA design flow and architecture. The design techniques discussed in the manuscript are useful during RTL design and FPGA-based design implementation. For more details about the Verilog you can refer the books on Verilog.

Fig. 18.9 RTL schematic of Example 18.12



18.14 Design Guidelines and Summary

Following are few of the guidelines to conclude the chapter.

1. Use blocking assignments to model the 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. Cover all the case conditions or include default while using the ***case-endcase*** to avoid unintentional latches.
5. Use all the required inputs or signals in the sensitivity list while using 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.
9. For decoders and multiplexers code the RTL using ***case-endcase*** to infer parallel logic.
10. For priority encoders use the nested ***if-else*** while coding the RTL to infer the priority logic.
11. To include all required inputs in the sensitivity list use ***always@****.

Chapter 19

FPGA Architecture and Design Flow



The understanding of the FPGA architecture and the FPGA flow is useful during the proof-of-concept phase that is during prototyping.

As a proof of concept, we use the FPGA as programmable logic. Depending on the need of design we need to identify the FPGA which has suitable architecture. The chapter is useful to understand about the FPGA architecture and FPGA design flow.

19.1 Basics of Programmable Logic

The programmable logic evolution started in the early 70s, and we had following small-density programmable logic devices.

1. Programmable array logic (PAL)
2. Programmable logic array (PLA)
3. Simple programmable logic devices (SPLD)
4. Complex programmable logic devices (CPLD)

These programmable devices are used to design and test the small density logic and useful in the product design.

As logic complexity has grown we have witnessed high-density programmable logic devices such as field programmable gate arrays (FPGAs), and they are useful in the field due to their programmable features.

In the market there are many popular FPGAs from the AMD and Intel, and depending on the designer's need, they can be procured and used in the product design and prototyping.

So, before proceeding further let us understand the important differences between the CPLD and FPGA.

19.2 CPLD Versus FPGA

The CPLD stands for the complex programmable logic device, and it is combination of the many SPLDs with the interconnect resources. They are mainly useful in the design of small-density state machines and state machine controllers, and they are also treated as gate rich logic. They exhibit the clean timing but having limitations due to gate count.

CPLD is also considered as mega PAL, super PLA, or enhanced PLD (EPLD). The main advantage of CPLD is their improved timing performance as compared to the SPLDs. The CPLD architecture is shown in Fig. 19.1.

FPGAs are field programmable gate arrays, and they have a greater number of sequential elements such as flip-flops as compared to CPLD. The FPGAs from AMD and Intel are useful to realize the larger-density FSM controllers, processors, and larger-density logic. Few of the FPGAs are Spartan, Virtex, Kintex, Pynq, and Stratix. Modern FPGAs have the resources like DDR controller cores, USB connectivity, Ethernet, and PCI and are useful during prototyping.

As FPGAs support the programmable features and can be programmed at field using the vendor-specific EDA tools, they are called as FPGA.

FPGA is also called as programmable ASIC and consists of the basic resources as configurable logic blocks (CLBs), IO blocks (IOBs), clocking resources, and programmable interconnects. Modern FPGAs even consist of the multipliers, block RAMs, DSP blocks, high-speed interfaces, and processor cores. The FPGA basic architecture is shown in Fig. 19.2.

Following are few of the important parameters of the FPGA.

- Configurable Logic Block (CLB):** The programmable logic block which can be configured to realize the desired combinational and sequential logic functionality is called as CLB. While implementing the logic on the programmable array the logic is decomposed into small-density logic blocks and mapped using the

Fig. 19.1 Block diagram of CPLD. Source www.onerup.com

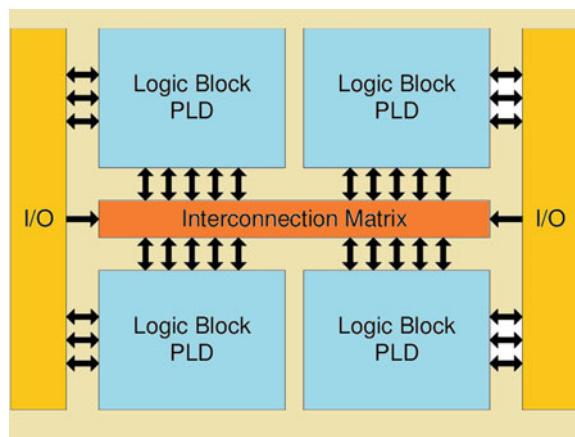
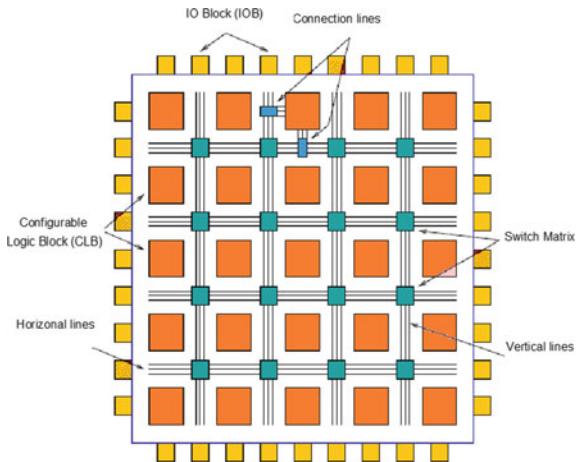


Fig. 19.2 Basic FPGA architecture. Source www.xilinx.com



multiple CLBs. To perform the placement, routing, and to program the FPGA we need to use the vendor-dependent EDA tools.

2. **Programmable Interconnects:** The routing resources used in the FPGA are called as a programmable interconnect.
3. **Programmable Switches:** The switches or switch matrixes are used to route the connections between CLBs.
4. **Logic Density:** The number of logic gate or the number of logic cells in the FPGA per unit area is called as logic density. The number of logic cells is combination of the combination and sequential logic resources.
5. **Logic Capacity:** The amount of logic that is mapped into the single FPGA device is called as logic capacity. The logic capacity is given in the form of the number of logic gates in the programmable gate array. The logic capacity can be thought in the form of number of universal logic gates.
6. **FPGA Performance:** The maximum operating frequency of the FPGA is measure of the performance of the sequential design. For the combinational logic, the longest path in the design decides the performance.

19.3 ASIC Versus FPGA

The ASIC is application-specific integrated circuit. The comparison of the ASIC and FPGA design is listed in the following Table 19.1.

Table 19.1 Comparison of ASIC with FPGA

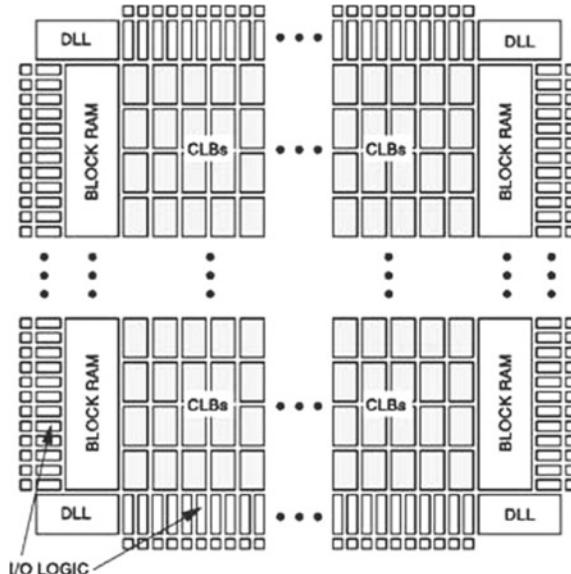
Selection criteria	ASIC	FPGA
Design cycle	Long	Short
Time to prototype samples	High	Low
Time for volume Production	High	Low
design cost	High	Low
Risk	High	Low
Cost per unit	Low	High

19.4 FPGA Architecture

The important functional blocks of the FPGA are discussed in this section. For the logic design and prototype team the understanding of FPGA plays important role during product design. The FPGA architecture is shown in Fig. 19.3.

- Configurable Logic Block (CLB):** CLB consists of the look-up tables (LUTs), multiplexers, and flip-flops. 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. The CLB structure is vendor specific. Most of the time we experience the LUTs having 4,6,9 inputs for various FPGA families.
- Input Output Block (IOB):** This block is useful to control the data flow between the internal logic and IO pins of the device. Each IO supports the bidirectional data flow and has the tri-state control. There are almost 24 different IO standards

Fig. 19.3 FPGA architecture. Source www.xilinx.com



- which include 7 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 larger amount of the data and available in the form of dual port RAM, for example, 18 Kbit dual-port block RAM. FPGA can consist of such multiple BRAM blocks depending on the device architecture.
 4. **Digital Clock Managers (DCMs):** They are used for clock management and provide fully calibrated digital clock solution. They are used to distribute the clock with uniform clock skew. Even they are useful to delay the 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 FPGA device family 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 and data processing. These blocks are used to improve the overall performance of the FPGA while processing the larger amount of data during the DSP applications.

Modern FPGAs have the processor cores, DDR controllers, USB, Ethernet, high-speed interfaces to realize and to prototype the SOC.

The high-density Spartan series FPGA architecture is shown in the figure.

The architecture of FPGA consists of the array of CLBs, block RAMs, multipliers, DSPs, IOBs, and digital clock managers. Delay Locked Loop (DLL). The floor plan for the XILINX SPARTAN Series FPGA is shown in Fig. 19.4.

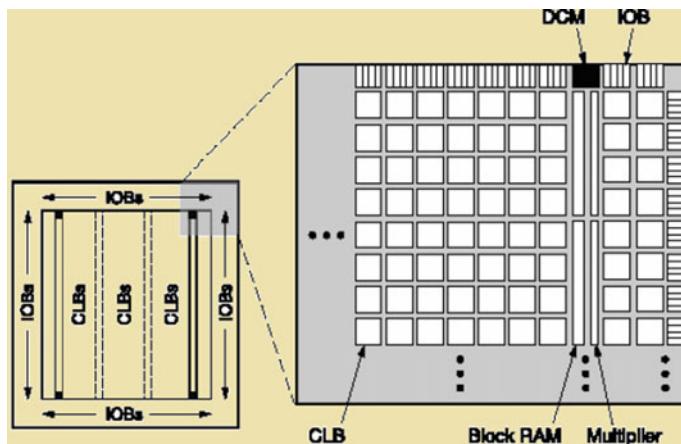


Fig. 19.4 XILINX SPARTAN Series device

19.5 FPGA Design Flow

The product idea starts with the market survey and can be conceptualized to have better product features by understanding the specifications. The documentation to capture the product specifications should be created during the initial stages. At the high level the FPGA design flow includes following important steps and shown in the flow Fig. 19.5.

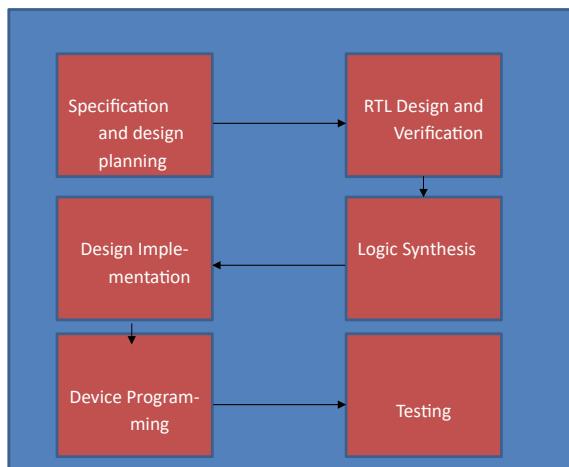
1. Design specifications understanding and requirement capture
2. RTL design that is design entry
3. Design verification and logic synthesis
4. Design implementation
5. Device programming
6. Testing

19.5.1 Design Planning

During this phase the logic design team understands the overall product or design specifications. The functional specifications are captured in the requirement capture documents. By using the functional specifications, the architect team creates the top-level design architecture and microarchitecture. This phase is one of the important phases during the design as the results of all future phases are dependent on the architecture designed.

The architecture team should consider the area, speed, and power requirements to design the architecture and microarchitecture. The multiple architectures for the

Fig. 19.5 FPGA design flow



same design can be useful to decide the best suitable architecture for the desired functional specifications.

19.5.2 RTL Design

Before the design entry the design planning needs to be completed by understanding the design requirements and the design specifications. The design specifications need to be documented in the form of block and sub-block level design and called as architecture and microarchitecture document.

During the architecture design phase, the requirement of memory, resources required, area, speed, and power requirements for the design need to be estimated. Depending on the requirement the suitable FPGA device need to be identified for the implementation.

RTL can be coded by using either Verilog (.v) or VHDL (.vhd) or by using SystemVerilog (.sv). The main objectives of the RTL design team are.

1. Code the RTL to meet the functional requirements.
2. RTL design should use the synthesizable constructs.
3. RTL design team should use the optimization and performance improvement techniques to have efficient RTL
4. RTL design team should perform the sanity level tests.

After the successful RTL design phase, the design need to be simulated to check for the functional correctness of the design. This is called as functional simulation.

19.5.3 Design Verification and Synthesis

During functional verification the set of inputs are applied to the design with intention to check for the functional correctness of the design. Although the timing or area, power issues can crop up during the later design cycle but design team is at least sure about the functional correctness of the design. The verification can be automated using the verification architecture, and the design can be verified using the various test cases and corner cases with the goal of the higher coverage.

The major goal of the FPGA design engineer is to infer the intended design logic using FPGA! The synthesis is the process of getting the lower level of the design abstraction from the higher level. During the logical synthesis the RTL is used as one of the input to get the lower level of abstraction as the gate level netlist. The netlist is device independent and can be in the standard format like electronic design interchangeable format (EDIF).

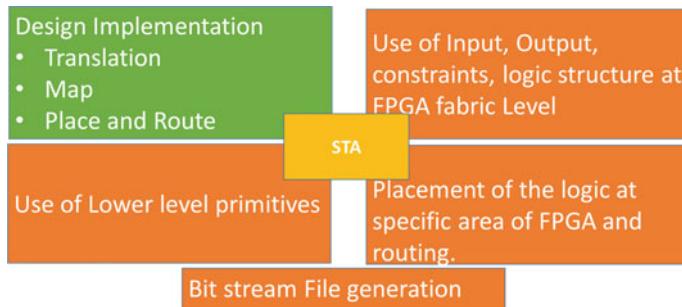


Fig. 19.6 Design implementation and bit-file generation. Source www.onerupeest.com

19.5.4 *Design Implementation*

The design passes through the various steps during implementation. These steps are translate, map and place, and route. During the design implementation the EDA tool translates the design into the desired format and maps it on the FPGA fabric by considering the overall area requirements. The mapping is performed by the EDA tool and functionality uses the 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 to get the intended functionality. This 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. The intention is to find out how many timing paths have setup or hold violations? Extracting the RC delays and using them by timing analyzer is called as back annotation. Figure 19.6 shows the design implementation various steps including the role of the timing analyzer.

19.5.5 *Device Programming and Testing*

The FPGA is programmed by using the vendor-specific or proprietary bit-stream file. Bit stream is binary data file need to be loaded into the FPGA to program the device.

Test the design using the suitable test setup and document the results.

19.6 FPGA-Based Product Design

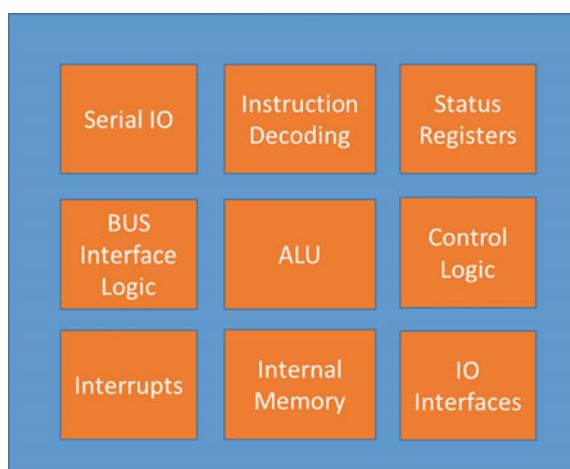
Consider the design of the single clock domain processor (Fig. 19.7).

What we need to do is that we need to understand the functional and timing requirements of the design, and we need to plan for the following milestones.

1. Feasibility understanding and specification extraction
2. Top-level architecture design
3. Microarchitecture design
4. Resource requirements
5. Design partitioning
6. Block-level RTL design
7. Block-level RTL verification
8. Top-level RTL design
9. Top-level RTL verification
10. Constraints and synthesis
11. Design implementation
12. Timing analysis
13. FPGA programming
14. Testing
15. Product documentation.

To have more understanding about the FPGA-based product design, prototyping using single and multiple FPGAs you can refer the SOC prototyping books.

Fig. 19.7 Top-level architecture of processor



19.7 Summary

Following are few important points to conclude this chapter.

1. PLDs are classified into three main categories and are SPLD, CPLD, and FPGAs.
2. PAL and PLA are also called as SPLDs and used to realize small gate count designs.
3. CPLDs are moderate-density FPDs and are used to design small gate count FSMs due to good timing performance.
4. FPGAs are used to design the complex gate count FSMs and are called as flip-flop rich logic.
5. The design specifications need to be documented in the form of block and sub-block level design and called as architecture and microarchitecture document.
6. During functional simulation, the set of inputs are applied to the design with intention to check for the functional correctness of the design.
7. The synthesis is the process of getting the lower level of the design abstraction from the higher level.
8. The design passes through the various steps during implementation. These steps are translate, map and place, and route.
9. The FPGA is programmed by using the vendor-specific or proprietary bit-stream file.
10. The architecture of FPGA consists of the array of CLBs, block RAMs, multipliers, DSPs, IOBs, and digital clock managers. Delay Locked Loop (DLL).
11. Modern FPGA consists of the dedicated multipliers, DSP blocks, high-speed interfaces with the processor cores.
12. FPGA designer needs to use the design guidelines while using the FPGAs.

19.8 Further Reading

Hope you have enjoyed reading this book, and I hope that the domain knowledge shared is useful to you to design the digital systems, architectures, and even to improve the design performance. Now what is next?

You can use the domain knowledge shared in this manuscript to design.

1. Complex VLSI systems
2. VLSI architectures
3. Product ideas

Even during the RTL design using Verilog, SystemVerilog you can use the conceptual understanding of design, performance improvements.

For further readings, exercises, and complex designs you can read books on Verilog, SystemVerilog, VHDL, Chisels. Few of my manuscripts in these areas are listed below.

1. Digital Logic Design Using Verilog: <https://link.springer.com/book/10.1007/978-981-16-3199-3>
2. SystemVerilog for Hardware Description: <https://link.springer.com/book/10.1007/978-981-15-4405-7>
3. PLD Based Design with VHDL: <https://link.springer.com/book/10.1007/978-981-10-3296-7>

Let us hope that you all have benefited with the domain knowledge shared in this manuscript! I hope that you should develop products and you should contribute in semiconductor design and product innovation.

All the very best!

Bibliography

Architecture design case studies of processors and FIFO. VLSI, SOC and AI Research Centre Pune,
www.onerupeest.com

Digital logic design using verilog: Coding and RTL synthesis, www.springer.com; <https://link.springer.com/book/10.1007/978-981-16-3199-3>

Digital design techniques and exercises, www.springer.com; <https://link.springer.com/book/10.1007/978-981-16-5955-3>

Index

0–9

- 1010 Mealy overlapping sequence detector, 206
- 101 Mealy overlapping sequence detector, 202
- 101 overlapping Moore sequence detector, 200
- 16-bit processor top-level architecture, 259
- 1-bit comparator, 222
- 2:1 mux, 52, 95
- 2:4 decoder, 144
- 2:4 decoders, 80, 85
- 2-XOR gates, 57
- 2's complement addition, 53, 216, 232
- 2-bit Binary to Gray code, 33
- 2-bit gray counter, 243
- 2-bit gray to binary code, 34
- 2-bit synchronous binary up-counter, 144
- 2-input NAND, 64, 69
- 2-input NOR, 69
- 2-input XNOR, 102
- 2-input XOR, 102
- 2-varibale K-map, 119
- 3:8 decoder, 85
- 3-bit binary to gray code converter, 57
- 3-bit gray to binary code converter, 59
- 3-bit up-counter, 140
- 3-varibale K-map, 56, 113, 116
- 4:1 mux, 66, 97, 98, 100
- 4:2 encoder, 87, 89
- 4:2 priority encoder, 91
- 4-bit ALU, 225
- 4-bit BCD or 8421 Code, 2
- 4-bit Johnson counter, 233
- 4-bit PIPO register, 131
- 4-bit adder and subtractor, 53, 216

4-bit addition and subtraction, 216

- 4-bit binary subtractor, 215
- 4-bit processor, 224
- 4-bit register, 282
- 4-bit ring counter, 146, 233
- 4-stage pipelined architecture ., 251
- 50% duty cycle, 150
- 8421 code, 2
- 8:1 mux, 97, 98

A

- Active event queue, 274, 278
- Active high asynchronous reset, 131, 141
- Adder chain, 52
- Adders and subtractors, 214
- Addition, 231
- Address, 6
- Address bus, 30
- ALU, 225
- Always@*, 273
- Always @(negedge clk), 273
- Always @(posedge clk), 273
- Analog systems, 4
- AND, 37
- AND Gate, 15, 37, 42
- Application Specific Integrated Circuit (ASIC), 287
- Architecture, 27, 291
- Architecture and micro-architecture, 27
- Architecture design, 27
- Area, 7, 8, 53
- Area and power optimization, 53
- Area optimization, 53, 218, 225, 231
- Arithmetic and logical operations, 252
- Arithmetic resources, 46

Arithmetic resource sharing, 231
 Assign, 273
 Asynchronous counters, 126, 135
 Asynchronous design, 105, 109, 111, 124, 134, 227
 Asynchronous FIFO, 262
 Asynchronous reset, 124, 209, 282
 Asynchronous reset logic, 282
 Asynchronous sequential circuits, 109

B

Back annotation, 292
 BCD codes, 2
 Bidirectional shift register, 133
 Binary coded decimals, 2
 Binary encoding, 179
 Binary number, 1
 Binary to gray code, 32
 Bit-stream, 292
 Blocking (=) Assignments (BA), 278–280
 Block RAM (BRAM), 289
 Blocks of FIFO, 267
 Boolean equation, 36, 39, 43, 45, 46, 146, 148, 203, 206
 Boolean expression, 41, 100
 Boolean function, 46, 47
 Bubbled AND, 21
 Bubbled OR, 21, 64
 Burst length, 263
 Bus multiplexing, 68

C

Capture flip-flop, 171
 Cascaded 2:1 mux, 160
 Cascaded stages, 158
 Cascaded XOR gates, 222
 Cascade logic, 158
 Cascade multiplexer, 68
 Cascade stages, 276
 Case Study, 249, 266
 Case Study : Microarchitecture Design, 253
 Chain of multiplexers, 52
 Channel transmission frequency, 79
 Clock buffers, 169
 Clock distribution schemes, 209
 Clock gating, 229, 245
 Clock gating cell, 229
 Clock gating logic, 228, 245
 Clock multiplexing, 68
 Clock network, 246
 Clock path, 169, 209
 Clock skew, 169, 227

Clock source, 110
 Clock switching, 229
 Clock to q delay (tctoq or tpff), 110
 Combinational Design elements, 11
 Combinational logic, 11, 108, 273, 278
 Combinational logic delay, 110
 Combinational path, 162, 164
 Combinational shifter, 96
 Common resources, 53
 Comparator, 220
 Compatibility, 8
 Complex Programmable Logic Devices (CPLD), 285
 Configurable Logic Block (CLB), 286, 288
 Continuous assignments, 273, 274, 277
 Control, 6
 Control and data path synchronizers, 260
 Control path, 61
 Control path logic, 210
 Control signal, 61, 134
 Counters, 109
 Critical path, 241

D

Data, 6
 Data and control path, 61, 209
 Data and control path management, 209
 Data and control path optimization, 189, 231
 Data arrival time, 173, 236, 238, 239, 241, 249, 250
 Data bus, 30
 Data control elements, 52
 Data or control signals, 134
 Data path, 52, 61, 108, 125, 135, 210, 216, 238
 Data path and control path optimization, 218
 Data paths and control paths, 209
 Data path logic, 210
 Data path synchronizer, 262, 266
 Data required time, 173, 236, 238, 239, 241, 249, 250
 Data selection elements, 52
 De 'morgens law, 250
 Decimal number, 1
 Decode Block, 255
 Delay Locked Loop (DLL), 289, 294
 Delays, 157
 De Morgan's theorem, 20, 36, 39, 44, 45, 64
 Demultiplexers, 79
 Density of the logic, 213

- Depth of FIFO, 263, 268
Derived clocks, 120
Design considerations, 7
Design implementation, 292
Design performance, 135, 238
Design planning, 290
Design scenario, 129
Design verification and synthesis, 291
Design without resource sharing, 232
Device programming, 292
D flip-flop, 105, 107, 111
Digital Clock Managers (DCMs), 289
Digital systems, 4
DSP blocks, 289
Dual port RAM, 267
Dynamic power, 229
Dynamic power reduction, 245
- E**
EDA, 292
Edge, 22
Electronic Design Interchangeable Format (EDIF), 291
Empty and full flag, 266
Empty flag logic, 267
Encoder, 86, 87
End point, 162, 165
Energy stored, 157
Even or odd parity, 222
Excess 3 code, 3
Excitation input, 112, 145, 148
Excitation table, 111, 113, 116, 119, 181, 186, 198, 202, 206, 243
Exclusive OR, 18
Execution block, 256
External IO, 7
External memory, 7, 251
- F**
Fanout, 8
Fetch block, 254
Field Programmable Gate Array (FPGA), 286
FIFO depth calculation, 263
FIFO memory buffers, 262
FIFO synchronizers, 134, 209, 260
Finite State Machine (FSM), 177
First integrated circuit, 106
Flag Register, 256
Flip-flop negative edge sensitive, 139
Flip-flop propagation delay, 110
Flip-flops, 24
- Four-bit binary, 1
Four-bit Latch, 137
Four-bit shift register, 131
Four clock latency, 131
FPGA Performance, 287
FSM based controllers, 189
FSM Design, 185, 188, 200, 204, 208, 245
Full adder, 49, 51, 220
Full flag logic, 267
Functional simulation, 291
Functional specifications, 28
- G**
Gate level design of 2:1 mux, 63
Generated clocks, 134
Glitches, 124, 125, 135, 178, 229
Glitch free FSMs, 189
Glue logic, 32
Gray code, 3
Gray counter, 118, 242
Gray encoding, 180, 189, 243
Gray pointers, 267
Gray to binary code converter, 57
- H**
Half-adders, 46, 47, 51, 72
Half-subtractor, 47, 49
Handshaking, 262
Hexadecimal, 1
Hold slack, 162, 173
Hold Time (t_h), 110, 160, 161
- I**
Idle cycles, 264, 265
Idle cycles between writes and reads, 264
Indeterminate output, 106
Input Output Block (IOB), 288
Input to reg path, 162, 163
Internal memory, 131
Internal register array, 251
Invalid output, 89
IO blocks, 292
IO devices, 8, 29
IO high performance standards, 289
- J**
JK flip-flop, 105, 106, 137
Johnson counter, 147

K

Karnaugh maps, 33
K-map, 33, 46, 56, 87

L

Latches, 22
Launch flip-flop, 171
Left shift, 96
Level, 22
Level synchronizers, 124, 209, 260, 261, 267
Load and shift register, 141
Logic capacity, 287
Logic cells, 292
Logic density, 287
Logic duplications, 234
Logic gates, 12
Longest delay path, 241
Loss of correlation, 271
Lowest order mux, 63

M

Macrocells, 292
Many to one switch, 52
Maximum frequency calculations, 110, 162, 164
Maximum operating frequency, 165, 167–169, 171, 172, 236, 239–241, 249, 250
Mealy FSM, 177, 178, 187, 188, 201, 204, 205, 208
Mealy sequence detector, 196
Mealy state diagram, 190, 194, 201, 205
Mechanical assembly, 7
Mega PAL, 286
Memories, 7, 135
Memory devices, 29
Memory or IO devices, 80
Metastability, 261
Metastable state, 261
Microarchitecture, 29, 253, 268
Microarchitecture of the FIFO, 267, 268
Microprocessor, 7, 29
Minimum number of 2:1 mux, 72
MOD-3 counter, 148
MOD-3 counter having 50% duty cycle, 150
MOD-3 synchronous up-counter, 150
MOD-4, 185, 188
MOD-8 binary down-counter, 115
MOD-8 binary up-counter, 113

MOD-8 synchronous binary down-counter,

118

MOD-8 synchronous binary up-counter,
115

Moore FSM, 177, 178, 184, 200, 244

Moore state diagram, 190, 193, 197

Multi-bit adder, 53, 215

Multi-bit adders and subtractors, 52

Multi-bit signals, 262

Multiple clock boundaries, 261

Multiple clock domain, 36, 209, 260

Multiple clock domain designs, 59, 134, 259

Multiple power domain strategies, 7

Multiplexer, 52, 62, 63, 218, 245

Multiplexer chain, 96, 141

Multiplier, 289

Multistage pipelining architecture, 251

Mux Based Designs, 95

N

NAND, 64
NAND Gate, 16, 38, 39, 42, 43
Negative clock Skew, 169, 170
Negative edge sensitive D flip-flop, 25
Nested *if...else*, 276
Netlist, 291
Next state logic, 107, 181, 185, 197, 199, 201, 205, 242
NMOS, 157
Noise margin, 8
Non-blocking assignments (NBA), 278, 280, 281
NOR gate, 14, 39, 41
NOT gate, 12, 157
Number representation, 1

O

Octal, 1
Octal numbers, 1
ON duty cycle, 148
One-hot encoding, 180, 189, 194
Operations results, 52
Optimization, 218
Optimization constraints, 158, 214
Optimization phase, 218
OR gate, 13, 38, 41
Output logic, 181, 186, 197, 201, 205, 242
Output to reg path, 162
Overlapping Mealy sequence detector, 204, 208

P

Parallel In Parallel Out (PIPO) register, 281
 Parallel Input and Parallel Output, 131
 Parallel logic, 159
 Parallel paths, 61
 Parity detector, 73, 222
 Partitioning, 189
 Performance improvement, 236
 Peripheral devices, 7
 Phase Lock Loop (PLL), 134
 Phase shift, 123, 134
 Pin multiplexing, 68
 Pipelined processor, 27
 Pipelined register, 241, 242
 Pipelined stages, 252
 Pipelining, 226, 239, 249
 PIPO Register, 131
 PMOS, 157
 POS expression, 37
 Positive clock Skew, 169
 Positive edge sensitive D flip-flop, 25
 Positive edge sensitive Flip-flop, 139
 Power, 7, 8, 180
 Power consumption, 245
 Power dissipation, 157, 213
 Power optimization, 189
 Priority encoder, 90
 Priority interrupt control logic, 92
 Priority logic, 276
 Priority multiplexer, 160
 Priority multiplexing, 160
 Procedural blocks *always*, 278
 Processors, 5
 Product of Sum (POS), 33, 37
 Product term, 63, 80, 83, 84
 Programmable Array Logic (PAL), 285
 Programmable ASIC, 286
 Programmable Interconnects, 287, 292
 Programmable Logic Array (PLA), 285
 Programmable Switches, 287
 Propagation delay, 68, 157, 158
 Propagation delay of flip-flop (t_{pff}), 160
 Propagation Delay of flip-flop ($t_{pff}=t_{cq}$), 162

R

Read and write operations, 30
 Read control logic, 267
 Receiver, 79
 Register, 131
 Register balancing, 226, 241, 249
 Registered inputs, 143

Registered inputs and registered outputs, 226
 Register-to-register path, 241
 Reg to output path, 163
 Reg-to-reg path, 162, 164, 166, 241, 242
 Reg-to-reg timing path, 172, 236, 241
 Reset de-assertion, 270
 Reset path, 112, 209
 Reset synchronizers, 209, 270
 Resources, 218
 Resource sharing, 53, 218, 220, 231, 233
 Right or left shift, 132
 Right shift, 96
 Ring counter, 144
 Ripple counter, 128
 RISC, 251
 RISC processor, 28
 Rising edge, 25
 RTL Design, 291

S

Sensitivity list, 275
 Sequential circuit, 233
 Sequential design, 111, 162
 Sequential logic, 11
 Serial input serial output shift register, 281
 Setup Slack, 162, 165, 238, 240, 241, 249, 250
 Setup Time, 161
 Setup Time (t_{su}), 110, 161
 Shift registers, 109, 131, 278
 Simple Programmable Logic Devices (SPLD), 285
 Simulation and synthesis mismatch., 275
 Single clock domain, 252
 Single clock domain design, 224, 259
 Single stuck at fault, 102
 SOP expression, 36, 37, 47
 Speed, 8
 Speed and power, 158
 Speed improvement, 249
 Speed of the design, 213
 SR Flip-flop, 105
 STA, 292
 Start point, 162, 165
 State diagrams, 189
 State encoding, 179
 State encoding methods, 179
 State register, 181, 185, 197, 201, 205, 242
 State table, 113, 116, 118, 145, 147, 181, 186, 198, 202, 206, 243
 State transition, 178

Status, 6
 Store Block, 256
 Stray capacitance, 157
 Stuck at 0, 102
 Subtraction, 231
 SUM function, 72
 Sum of Product (SOP), 33, 63

Switching, 228
 Switching power, 229, 245
 Synchronous, 109
 Synchronous active low reset, 252
 Synchronous design, 105, 109, 110
 Synchronous gray counter, 118, 120, 245
 Synchronous modulo-8, 113
 Synchronous reset, 125, 283
 Synthesis, 291, 294
 Synthesizable, 273
 System components, 6
 System design, 29
 System design architecture, 29
 System level design, 259
 System specifications, 6

T

Testing, 292
 Three-bit binary to gray code, 55
 Three-bit gray to binary code converter, 57
 Timing, 22, 23
 Timing analysis, 162
 Timing or area, power, 291
 Timing paths, 162, 166
 Timing performance, 292
 Timing sequence, 140, 261
 Timing violation, 261

Timing waveform, 81
 Toggle flip-flop (T flip-flop), 105, 108, 109, 165
 Transmitter, 79
 Twisted ring counter, 148
 Two variable function, 34

U

Universal gate, 40, 41, 45, 64
 Unweighted numbers, 1

V

Veitch chart, 33
 Verilog constructs, 273
 VLSI, 1, 27
 VLSI based system, 27
 VLSI perspective, 259
 VLSI scenarios, 171
 VLSI specific design scenario, 174
 VLSI specific scenarios, 171

W

Weighted numbers, 1
 Write control logic, 267

X

XNOR, 40, 45
 XNOR Gate, 19, 44, 73
 XOR, 44, 53, 220
 XOR function, 75
 XOR Gate, 18, 43, 50, 222