Shirshendu Roy

Advanced Digital System Design

A Practical Guide to Verilog Based FPGA and ASIC Implementation

Ane Books Pvt. Ltd.

Springer

Advanced Digital System Design

Shirshendu Roy

Advanced Digital System Design

A Practical Guide to Verilog Based FPGA and ASIC Implementation

Shirshendu Roy
Department of Electronics
and Communication Engineering
Dayananda Sagar University
Bengaluru, India

ISBN 978-3-031-41084-0 ISBN 978-3-031-41085-7 (eBook)
https://doi.org/10.1007/978-3-031-41085-7

Jointly published with Ane Books Pvt. Ltd.
In addition to this printed edition, there is a local printed edition of this work available via Ane Books in South Asia (India, Pakistan, Sri Lanka, Bangladesh, Nepal and Bhutan) and Africa (all countries in the African subcontinent).
ISBN of the Co-Publisher's edition: 978-81-94891-88-8

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

To My Adorable Daughter
Trijayee.

Preface

Objective of the Book

In today's world where technology is applied at every application, there has been a huge demand of implementation of signal-, image- or video-processing algorithms. These real-time systems consist of both analog and digital sub-systems. The analog part is mainly responsible for signal acquisition step and the processing part is majorly achieved by digital sub-systems. An optimized implementation of a digital system is very crucial to improve the performance of the overall integrated circuit (IC).

Digital system design is not a new thing to the researchers or to the engineers in the field of VLSI system design. The field of digital system design is divided into two zones, viz., transistor-level design and gate-level architecture design. Over the past few decades many research works, books or online tutorials on both the topics of digital system design are published. In this book, gate-level design of digital systems using Verilog HDL is discussed. The major objective of this book is to cover all the topics which are very important for a gate-level digital system designer.

This book covers some basic topics from digital logic design like basic combinational circuits and sequential circuits. Also covers some advanced topics from digital arithmetic like fast circuit design for addition, multiplication, division and square root operation. Realization of circuits using Verilog HDL is also discussed in this book. Overview on the digital system implementation on Field Programmable Gate Array (FPGA) platform and for Application-Specific Integrated Circuit (ASIC) is covered in this book. Timing and power consumption analysis are two most important things that must be performed to make successful implementation. Thus this book covered these two areas to give readers an overview on timing and power analyses. At the end, few design examples are given in this book which can help readers directly or indirectly. Thus this book can be a perfect manual to the researchers in the field of digital system design.

Organization of the Book

Chapter 1 focusses on the representation of binary numbers. This chapter discusses the representation of binary numbers in One's complement, Two's complement and Signed magnitude number system. Basics of floating point data representation and fixed point data representation is discussed in this chapter. Signed binary number system which is frequently used for performing fast arithmetic operations is also discussed.

Chapter 2 discusses the Verilog HDL which is a very powerful programming language to model the digital systems. In this chapter, concepts about the Verilog HDL are discussed with suitable examples. All the different programming styles are discussed with the help of simple Multiplexer design. The test bench writing technique is also discussed in this chapter.

Basic concepts of combinational circuits are discussed in Chap. 3. All the major combinational circuits are covered in this chapter. Some of the basic circuits are Adder/Subtractor, Multiplexer, De-multiplexer, Encoder and Decoders. In addition to these circuits, design of 16-bit comparator, constant multipliers and code converters is also discussed.

Basic concepts of sequential circuits are discussed in Chap. 4. This chapter initially covers the concepts of different clocked flip-flops and then discusses about the various shift registers. Counter is a very important sequential circuit and this chapter discusses design of a simple synchronous up counter. Then this up counter is converted to a loadable up counter. In addition to the counter design, design of pseudonoise sequence generator and clock division circuits is also discussed.

In Chap. 5, memory design problem is discussed. This chapter mainly focusses on realization of memory elements using Verilog HDL. Behavioural HDL coding style is used to model the memory elements. Verilog codes for ROM and RAM are provided in this chapter. In addition to the single port memory elements, dual port ROM and dual port RAM are also modelled in this chapter.

Design of Finite State Machines is very important in designing digital systems. Thus a detailed discussion on the FSM design is given in Chap. 6. Design of Mealy and Moore machine is explained with the help of '1010' sequence detector. Then some of the applications are discussed where FSM design style is used. Various FSM state minimization techniques are also discussed in this chapter using a design problem.

Various architectures for addition operation are discussed in Chap. 7. This chapter mainly focusses on fast addition techniques but also discusses some other addition techniques. The different techniques which are discussed here are Carry Look-Ahead, Carry Skip, Conditional Sum, Carry Increment and Carry Bypass. Multi-operand addition techniques like Carry Save Adders are also discussed here.

Chapter 8 focusses on various architectures for multiplication operation and these architectures can be sequential or parallel. The array multipliers for both signed and unsigned operands are discussed. Like previous chapter, this chapter also focusses mainly on fast multiplication techniques like Booth multiplier. But, other important

multiplier design aspects like VEDIC multiplication techniques are also discussed here. Along with the multiplication, techniques to efficiently compute square of a number are also discussed in this chapter.

Chapter 9 discusses various division algorithms like restoring and non-restoring algorithm with proper example. Implementation of these algorithms is discussed here. Basic principle of SRT division algorithm is also given here with some examples. Some iterative algorithms for division operation are also explained here. Along with the division operation, computation of modulus operation without division operation is discussed in this chapter.

Square root and square root reciprocal are also very important arithmetic operations in implementing digital systems. Thus in Chap. 10, various algorithms and architectures to compute square root and square root reciprocal are discussed. Sequential algorithms, restoring and non-restoring algorithm also can be applied to compute square root. Likewise SRT algorithm is also applicable for square root with minor modifications. Some iterative algorithms are also explained to compute square root and square root reciprocal.

CORDIC algorithm is a very promising algorithm to compute various arithmetic operations and some other functions. Thus in Chap. 11, CORDIC theory and its architectures are explained. Two architectures for CORDIC are possible, serial and parallel. Both the architectures are discussed in detail. This chapter also provides a brief survey on different CORDIC architectures which are reported in recent publications.

Till this chapter fixed point data point is used to implement the digital systems. But floating point representation is another technique to represent the real numbers. Floating point data format is useful if high accuracy is desired. Thus in Chap. 12 floating point architectures are discussed to compute addition/subtraction, multiplication, division and square root with proper examples.

Timing analysis or more specifically static timing analysis is an important step to verify that a digital IC will work satisfactorily after fabrication or not. Thus Chap. 13 focusses on explaining different timing definitions and important concepts of static timing analysis. These topics are discussed here so that readers can carefully plan their design for desired maximum frequency at strict area constraint.

Digital systems can be implemented on FPGA platform or can be designed for ASIC as an IC. Chapter 14 covers a detailed discussion on the FPGA and ASIC implementation steps. First a detailed theory on the FPGA device is discussed and then the FPGA implementation steps are explained using XILINX EDA tool. A brief theory on the ASIC implementation using the standard cells with help of CADENCE EDA tool is covered.

Power consumption is a very important design metric to analyse the design performance. Thus Chap. 15 focusses on various techniques to achieve low power consumption. Dynamic power consumption can be reduced at every level of abstraction. Dynamic power consumption reduction using both algorithmic and architectural techniques is discussed here.

Example of some digital systems is given in Chap. 16 to give the readers idea about designing their own systems. First, implementation of digital filters (FIR and IIR) is

described using various topologies. Comparative study of the performances of the different FIR and IIR filter structures is also given. Two algorithms are implemented on FPGA which are K-means algorithm and spatial Median filtering algorithm. In addition to this, various sorting structures and architectures for matrix multiplication are discussed. At last, Verilog codes are provided to interface SPI protocol-based external ICs (DAC, ADC) or computers and micro-controllers using UART protocol with the FPGA device.

Verilog HDL is very popular in modelling the digital systems but has some limitations when verification of such systems comes into the picture. Thus system Verilog develops. Nowadays, system Verilog is mostly used and industry standard, which combines the features of C++ and Verilog. Basics of system Verilog is discussed in Chap. 17. This chapter highlights the major features of system Verilog and the differences from Verilog HDL.

Many advanced technologies are established to program the FPGAs. One such advancement is the idea to integrate the whole system on a single chip. In order to do this, many modern FPGAs are accommodating a dedicated processor. Partial re-configuration is another advanced feature of modern FPGAs. Thus in Chap. 18, these modern techniques of FPGA implementation are discussed.

Bengaluru, India Shirshendu Roy

Acknowledgements

I would like to place on record my gratitude and deep obligation to the professors of IIEST Shibpur and NIT Rourkela as this book is a result of their teachings and guidance. Specifically, I like to thank Dr. Ayan Banerjee, IIEST Shibpur (Department of ETC) and Dr. Debiprasad P. Acharya, NIT Rourkela (Department of ECE) to inspire me to pursue research in the field of digital system design. Also, I like to thank Prof. Ayas K. Swain whose teachings helped me writing this book.

I am indebted to my fellow researches and friends who have helped me by giving inspiration, moral support and encouragement to complete the book. Specifically I would like to thank S. Aloka Patra, Ardhendu Sarkar, Sandeep Gajendra and Jayanta Panigrahi to help me in finalizing the contents and preparing the manuscript.

I give immeasurable thanks to my family members for their patience, understanding and encouragement during the preparation of this book. They all kept me going and this book would not have been possible without their support.

Contents

About the Author

Dr. Shirshendu Roy has completed Bachelor of Engineering (B.E.) degree in Electronics and Tele-Communication Engineering in 2010 and Master of Engineering (M.E.) in Digital Systems and Instrumentation in 2016 from Indian Institute of Engineering Science and Technology (IIEST), Shibpur, Howrah, India. He has four years of valuable industrial experience as Control and Instrumentation engineer at MAHAN Captive Power Plant for Hindalco Industries Limited (Aditya Birla Group). He has completed Ph.D. degree from National Institute of Technology (NIT), Rourkela, Odisha, India in VLSI signal processing. Previously he worked in Gandhi Institute of GIET University, Odisha as assistant professor. Currently he is working in Dayananda Sagar University, Bengaluru as assistant professor.

He has published many international journals with the publishing houses like IEEE and IET. He has also authored and published many tutorials on the online platform in the field of Digital System Design. His current research interest includes compressed sensing, FPGA based implementation of algorithms (signal, image or video processing algorithms, machine learning algorithms, artificial neural networks, etc.), low-power architecture design, ASIC, application of FPGA for IoT applications.

Abbreviations

AB	Average Block
ADC	Analog-to-Digital Converter
ALU	Arithmetic Logic Unit
AMBA	Arm Advanced Micro-controller Bus Architecture
ASIC	Application-Specific Integrated Circuit
ATPG	Automatic Test Pattern Generator
AXI	Advanced Extensible Interface
BCD	Binary Coded Decimal
BIC	Bus Inversion Coding
BN	Basic Network
BNS	Binary Number System
BRAM	Block RAM
CB	Cluster Block
CIA	Carry Increment Adder
CLA	Carry Look-Ahead
CLB	Configurable Logic Block
CMOS	Complementary Metal-Oxide Semiconductor
CORDIC	Co-Ordinate Rotation DIgital Computer
CPA	Carry Propagate Adder
CPF	Common Power Format
CPU	Central Processing Unit
CSA	Carry Save Adder
CTS	Clock Tree Synthesis
DAC	Digital-to-Analog Converter
DCO	Digital Controlled Oscillator
DDR	Double Data Rate
DFT	Design For Testability
DIT	Decimation In Time
DPR	Dynamic Partial Re-configuration
DPRAM	Dual Port Random Access Memory
DPROM	Dual Port Read-Only Memory

DRC	Design Rule Checks
DSP	Digital Signal Processing
DTS	Dynamic Timing Simulation
EDC	Euclidean Distance Calculator
ERC	Electrical Rule Checks
FA	Full Adder
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FS	Full Subtractor
GE	Gaussian Elimination
GPIO	General-Purpose Input/Output
GPU	General Processing Unit
HA	Half-Adder
HDL	Hardware Description Language
HS	Half-Subtractor
IC	Integrated Circuit
ICAP	Internal Configuration Access Port
IIR	Infinite Impulse Response
ILA	Inline Logic Analyzer
IOB	Input/Output Block
IP	Intellectual Property
LEC	Logic Equivalence Check
LEF	Library Exchange Format
LFSR	Linear Feedback Shift Register
LIB	LIBerty timing models
LPF	Low-Pass Filter
LSB	Least Significant Bit
LUT	Look-Up Table
LVS	Layout Vs. Schematic
LZC	Leading Zero Counter
MAC	Multiply-ACcumulate
MCAP	Media Configuration Access Port
MCC	Manchester Carry Chain
MCF	Modified Cholesky Factorization
MFB	Minimum Finder Block
MMMC	Multi-mode Multi-corner
MSB	Most Significant Bit
NAN	Not A Number
NCD	Native Circuit Description
NGC	Native Generic Circuit
NGD	Native Generic Database
NRE	Non-recurring Engineering
OS	Occupied Slices
OTP	One Time Programmable

PAR	Placement And Routing
PCAP	Processor Configuration Access Port
PCF	Physical Constraints File
PG	Phase Generation
PIPO	Parallel Input Parallel Output
PISO	Parallel Input Serial Output
PL	Programmable Logic
PLL	Phase Locked Loop
PN	Pseudonoise
PR	Partial Re-configuration
PRM	Partial Re-configuration Modules
PS	Processing System
PSM	Programmable Switching Block
QRD	QR Decomposition
RCA	Ripple Carry Adder
RMSE	Root Mean Squared Error
RTL	Register Transfer Logic
SAIF	Switching Activity Interchange Format
SB	Sub-block
SD	Signed Digit
SDC	Synopsys Design Constraints
SDF	Standard Delay Format
SEU	Single Event Upsets
SI	Signal Integrity
SIPO	Serial Input Parallel Output
SISO	Serial Input Serial Output
SoC	System-on-Chip
SPI	Serial-to-Parallel Interface
SPR	Static Partial Re-configuration
SPRAM	Single Port Random Access Memory
SPROM	Single Port Read-Only Memory
SRAM	Static RAM
STA	Static Timing Analysis
TDP	Time-Driven Placement
TNS	Total Negative Slack
UART	Universal Asynchronous Receiver and Transmitter
UCF	User Constraints File
UUT	Unit Under Test
VCD	Value Change Dump
WNS	Worst Negative Slack
XST	Xilinx Synthesis Technology

Chapter 1
Binary Number System

1.1 Introduction

Representation of numbers is very important in digital systems for efficient performance. Binary number system (BNS) is a common way to represent any number in digital systems. In this conventional system, number representation should be valid for both positive and negative numbers. Also, representation technique should produce maximum accuracy in representing a real number. In addition to this conventional BNS, some computers adopted unconventional number systems to achieve faster speed for performing addition, multiplication or division. These unconventional number systems may have higher base compared to base 2 of binary system. This chapter will discuss basics of BNS and its representation techniques.

1.2 Binary Number System

Integers are represented using BNS in the digital systems implemented on computers, micro-controllers or on FPGAs. Any number is represented by two symbols which are '0' and '1' in BNS. A number X of length n is represented in BNS as

$$X = \{x_{n-1}, x_{n-2}, ..., x_1, x_0\} \tag{1.1}$$

Here, each digit or bit (x_i) takes value from the set {0, 1} and an integer is represented using n-bits. The value of n is important in correct representation of an integer. This decides the accuracy of a digital system. The value of the integer can be evaluated from the binary representation as

$$X = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + ... + x_1 2^1 + x_0 2^0 = \sum_{i=0}^{n-1} x_i 2^i \tag{1.2}$$

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_1

1.3 Representation of Numbers

In the implementation of a digital system, an integer can be positive or negative. The representation of integers should be done in a way that it can represent both negative and positive numbers. There are three major ways to represent the integers which are

1. Signed Magnitude Representation.
2. One's Complement Representation.
3. Two's Complement Representation.

1.3.1 Signed Magnitude Representation

In the signed magnitude representation, sign and magnitude of a number are represented separately. Sign is represented by a sign bit. For an n-bit binary number, 1-bit is reserved for sign and $(n-1)$ bits are reserved for magnitude. In general BNS, MSB is used for sign bit and logic 1 on this bit represents the negative numbers. This format is shown in Fig. 1.1. The maximum number that can be represented in this number system is

$$X_{max} = 2^{n-1} - 1 \tag{1.3}$$

This number can be both negative and positive depending upon the MSB bit. If $n = 8$ then $X_{max} = 127$. In this representation, zero does not have unique representation. Signed magnitude representation has a symmetric range of numbers. This means that every positive number has its negative counterpart. The integer $X = -9$ can be represented in signed magnitude representation as $X = 10001001$. Here, 7 bits are used to represent the integer and the MSB is 1 as the integer is negative.

Example

Addition of two numbers $X1 = -9$ and $X2 = 8$ can be done in the following way:

	1	0	0	0	1	0	0	1	$X1$
–	0	0	0	0	1	0	0	0	$X2$
	0	0	0	0	0	0	0	1	Y

In the above example, $X1$ is a negative number and $X2$ is a positive number. Thus the $X2$ is subtracted from $X1$ that results Y. In signed magnitude representation, sign

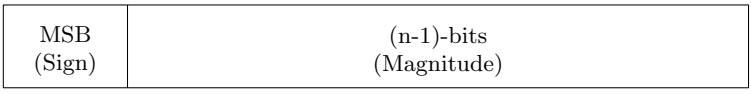


Fig. 1.1 Signed magnitude representation

Table 1.1 Addition of two signed magnitude numbers

MSB ($X1$)	MSB ($X2$)	Operation	MSB (Y)	
			$X1 < X2$	$X1 > X2$
0	0	Addition	0	
0	1	Subtraction	1	0
1	0	Subtraction	0	1
1	1	Addition	1	

bits of the operands decide the operation to be performed on the operands. Table 1.1 shows the addition/subtraction operation depending on the MSB bit of the operands.

If the sign of the two operands are same then the sign of the output is also same. In other case, comparison of the two operands is required.

1.3.2 One's Complement Representation

In one's complement representation, positive numbers are represented in the same way as they are represented in the signed magnitude representation. The negative numbers are represented by performing the bit-wise inversion on the number as shown below:

1. Obtain binary value of the magnitude of the number. For example, $X = 9$ binary of the magnitude is $9 = 01001$.
2. If the number is negative then invert the number bit wise. For example, -9 in one's complement representation is $X = 10110$.
3. If the number is positive then the binary value is the one's complement representation of that number.

The range of the one's complement representation is

$$-2^{n-1} + 1 \geq X \geq 2^{n-1} - 1 \tag{1.4}$$

In this representation, zero does not have unique representation as in case of signed magnitude representation. This representation also has the symmetric range. The MSB differentiates the positive and negative numbers. The range of one's complement representation for $n = 8$ is $-127 \geq X \geq 127$. When adding a positive number X and a negative number $-Y$ represented in one's complement. The result is

$$X + (2^n - ulp) - Y = (2^n - ulp) + (X - Y) \tag{1.5}$$

where $ulp = 2^0 = 1$. This can be explained below with the value of $n = 8$

$$X + (256 - 1) - Y = 255 + (X - Y) \tag{1.6}$$

Example

Addition of two numbers $X1 = -9$ and $X2 = 8$ represented in one's complement can be done in the following way:

	1	1	1	1	0	1	1	0	$X1$
+	0	0	0	0	1	0	0	0	$X2$
	1	1	1	1	1	1	1	0	Y

One's complement equivalent of the output is -1. In one's complement number system, a carry out is indication of a correction step. This is shown in the following example:

	1	1	1	1	0	1	1	0	$X1$
+	0	0	0	0	1	0	1	0	$X2$
$c_{out} = 1$	0	0	0	0	0	0	0	0	Y

Here, the result must be 1 but we get 0 instead. The correction is done by adding the c_{out} with the result.

1.3.3 Two's Complement Representation

Two's complement representation is very popular in implementing practical digital systems. Two's complement representation of a number can be obtained by first taking the one's complement representation and then by adding ulp. The steps for obtaining two's complement representation are

1. Obtain binary value of the magnitude of the number. For $X = -9$ binary of the magnitude is $9 = 01001$.
2. If the number is negative then take one's complement and then add ulp to it. After complementing $X = 10110$ and the final value is $10110 + 00001 = 10111 = X$.
3. If the number is positive then the binary value is the two's complement representation of that number.

In two's complement representation, positive and the negative numbers are differentiated by the status of the MSB bit. If the MSB is 1 then the number is treated as negative. Here the zero has unique representation. The range of numbers in two's complement number system is

$$-2^{n-1} \geq X \geq 2^{n-1} - 1 \tag{1.7}$$

The range is asymmetric as the number -2^{n-1} (100...000) does not have the positive counterpart. If a complement operation is attempted on 2^{n-1} (100...000) then the

result will be same. Thus in a design with fixed word length, this value is ignored and symmetric range is used. The usable range of two's complement representation for $n = 8$ is same as that of one's complement representation. The subtraction operation in two's complement representation can be expressed as

$$X + 2^n - Y = 2^n + (X - Y) \tag{1.8}$$

Example

Addition of two numbers $X1 = -9$ and $X2 = 8$ represented in two's complement can be done in the following way:

	1	1	1	1	0	1	1	1	$X1$
+	0	0	0	0	1	0	0	0	$X2$
	1	1	1	1	1	1	1	1	Y

Here two's complement equivalent of the Y is -1. In two's complement number system all the digits participate in the addition or subtraction process. In addition process, there may be chance of generating carry out and the overflow. If the sign of the operands are opposite, then overflow will not occur but carry out can occur. If the result is positive then the carry out occurs. Another example where carry is generated is shown below:

	1	1	1	1	0	1	1	1	$X1$
+	0	0	0	0	1	0	1	0	$X2$
$c_{out} = 1$	0	0	0	0	0	0	0	1	Y

In this example, result is $+1$ and carry out (c_{out}) is generated. The addition and subtraction process should be done in such a way that the overflow never occur otherwise wrong result will be produced. Consider the following example:

	1	1	0	0	1	-7
+	1	0	1	1	0	-10
	0	1	1	1	1	15

The actual result is -17 but here 15 is produced. Thus it should be taken care that the final result should not exceed the maximum number that can be represented in two's complement representation.

1.4 Binary Representation of Real Numbers

In the above section, how to represent positive or negative numbers is discussed considering that the integers do not have the fractional part. But practical integers can be fractional also. The digital platforms use a specific data format to represent such fractional integers. There are two types of data formats which are used in any digital design and these are

1. Fixed Point Data Format.
2. Floating Point Data Format.

1.4.1 Fixed Point Data Format

In the fixed point data format base architectures, the data width to represent the numbers is fixed. Thus the number of bits that are reserved to represent fractional part and integer part are also fixed. The fixed point data format is shown in Fig. 1.2. The decimal equivalent of a binary number in this format is computed as

$$\begin{aligned} X &= x_{m-1}2^{m-1} + \ldots + x_1 2^1 + x_0 2^0 \cdot x_{-1}2^{-1} + x_{-2}2^{-2} + \ldots + x_{-(n-m)}2^{-(n-m)} \\ &= \sum_{i=0}^{m-1} x_i 2^i + \sum_{i=1}^{(n-m)} x_{-i}2^{-i} \end{aligned} \tag{1.9}$$

Here, m-bits are reserved to represent the integer part and $(n-m)$-bits are reserved for fractional part. For example, if the data length is 16 bit and value of m is 6 then 6 bits are reserved for the integer part and rest of the bits are reserved for the fractional part.

Majority of practical digital systems implemented on any digital platform use this data format to represent fractional numbers. Now, to represent signed or unsigned numbers, any of the above techniques (signed magnitude representation, one's complement representation, and two's complement representation) can be used. But mostly the two's complement representation and signed magnitude representation are used. In the integer field, $(m-1)$-bits are usable as the sign of the number is represented using the MSB bit. An example to represent a fractional number in fixed point data format using two's complement representation is shown below:

Fig. 1.2 Fixed point representation of real numbers

1. Let the number be $X = -9.875$, $n = 16$ and $m = 6$.
2. Binary representation of the integer part using 6 bits is $9 = 001001$.
3. Binary representation of the fractional part using 10 bits is $0.875 = 2^{-1} + 2^{-2} + 2^{-3} = 1110000000$.
4. The magnitude of the number in fixed point format is 001001_1110000000.
5. Perform two's complement as the number is negative. Thus $X = 1\_10110\_0010000000$.

If 6 bits are reserved for integer part for $n = 16$, then maximum value of positive integer that can be represented is $2^{m-1} - 1 = 31$. Maximum representable negative number is $2^{m-1} = 32$. But to avoid confusion, the range of negative and positive numbers must be kept same. The maximum representable real number with this format is

$$011111\_1111111111 = 31.999023438 \tag{1.10}$$

Any number beyond this maximum number cannot be represented with $n = 16$ and $m = 6$. If the value of m decreases then the resolution of the number increases and range of representable numbers reduces. This means the gap between two consecutive numbers decreases. But if the value of m increases then range increases but resolution decreases. Designers have to carefully select the value of n and m. All the architectures discussed in the book are based on the fixed point data format as it provides an easy means to represent the fractional numbers. But this representation has some limitations due to its lower range.

1.5 Floating Point Data Format

Floating point data format is another technique to represent the fractional numbers. Floating point data format increases the range of the numbers that can be represented. Many dedicated processors or micro-controllers use this format to represent the fractional numbers. Floating point data format covers a wide range of numbers to achieve better accuracy compared to the accuracy achieved in case of fixed point representation. The concept of floating point data format comes from the representation of real fractional numbers. For example, the fractional number -9.875 can also be represented as

$$-9.875 = -1 \times 9875 \times 10^{-3} \tag{1.11}$$

Other representations are also possible. Thus floating point representation is not unique. The general format of the floating point representation is

$$X = S.M.r^{E_b} \tag{1.12}$$

So, a floating point number has three fields, viz., Sign (S), Mantissa (M) and Exponent (E). Here, r represents the radix and its value is 10 for decimal numbers. Similarly,

Fig. 1.3 Floating point representation of real numbers

the binary numbers also can be represented in this format where $r = 2$. Sign of a number is identified by a single digit. If the Sign bit is 1 then the number is negative else it is a positive number. In the mantissa part all the digits of the number are present. Number of bits reserved for mantissa part defines the accuracy.

The floating point data format according to IEEE 754 standard is shown in Fig. 1.3 for 16-bit word length. Here, 11 bits are reserved for mantissa part and 4 bits are reserved for exponent field. Mantissa part is represented as unsigned version. In the exponent field, unique representation must be adopted to differentiate positive and negative exponents. If the two's complement representation is used, then the negative exponent will be greater than the positive exponent. Thus a bias is added to the exponent to generate biased exponent (E_b). For example, if 4 bits are allocated for exponent field then the bias value is 7 ($2^3 - 1$). In general, for p bits the bias is $2^{(p-1)} - 1$. The value of E_b is obtained as $E_b = bias + E$. In this way, the value of $E = 0$ is represented as $E_b = bias$. Exponent value of $E = -1$ is represented as $E_b = 6$ and the exponent value of $E = 1$ is represented as $E_b = 8$. In this way the negative and positive exponents are distinguished.

Example

Represent the fractional number $X = -9.875$ in floating point data format for 16-bit word length.

1. Represent the magnitude of the fractional number in binary. $abs(x) = 9.875 =$ 1001_111.
2. First decide the sign bit. The sign bit is 1 as X is negative.
3. In the mantissa part 11 bits are reserved. The first step to represent the mantissa part is to normalize the binary representation. After normalization and adding zeros the result is 1_00111100000. The normalization is done to restrict the mantissa part between 1 and 2. Here, the number is shifted 3 bits in the left side. The MSB may not be included in the final version of the mantissa. The MSB is called as hidden bit and this bit is ignored in the mantissa representation according to the IEEE 754 standard.
4. As the mantissa part is 3 bits left shifted then the value of exponent is $E = +3$ and the value of the biased exponent is $E_b = 10_{10} = 1010_2$.
5. The floating point representation is 1_1010_00111100000.

A detailed discussion on the floating point numbers and floating point architectures is given in Chap. 12.

1.6 Signed Number System

In the above sections, the conventional methods of BNS are discussed. There exists some unconventional number systems which are very useful in developing very fast algorithms for addition, square root or division. One such number system is signed digit number system. In all the number systems discussed above, the digit set has been restricted to {0, 1}. However, the following digit set also can be allowed

$$x_i \in \{\overline{r-1}, \overline{r-2}, ..., \overline{1}, 0, 1, ..., (r-2), (r-1)\} \tag{1.13}$$

Here, $\overline{r-1} = -(r-1)$ and thus each bit is either positive or negative. There is no need of using separate sign bit. This establishes the concept of signed digit (SD) number system. Here the binary SD representation will be discussed where the value of r is 2.

1.6.1 Binary SD Number System

In case of binary SD number system, the digits can take the values from the set

$$x_i \in \{\overline{1}, 0, 1\} \tag{1.14}$$

Integers are represented in SD number system in similar way as in the BNS. Only thing is that in SD number system there are three states compared to two states in BNS. The number $X = 9$ is represented in SD number system as

$$X = 9 = 0000101\overline{1} = 2^3 + 2^1 - 2^0 \tag{1.15}$$

Representation of an integer in binary SD number system is not unique. There can be other representations also. The range of representable numbers (R) in binary SD number system is

$$\overline{1}\,\overline{1}\,\overline{1}...\overline{1}\,\overline{1} \text{ to } 111...11 \tag{1.16}$$

If there are n bits used to represent a SD number then $R = 3^n$ number of combinations are possible. Among these combinations, many are equivalent. Thus actual representable integers (R_{act}) are $R_{act} = 2^{n+1} - 1$. Thus total redundancy in binary SD number system is

$$\%\text{Redundancy} = \frac{R - R_{act}}{R} \times 100 \tag{1.17}$$

The redundancy value is for $n = 8$ is almost 93%. Binary SD numbers are different from the conventional binary numbers due to the presence of negative bit $\overline{1}$. In order

Table 1.2 Encoding binary SD numbers

x	Coding 1	Coding 2
0	00	00
1	01	01
$\bar{1}$	10	11

Table 1.3 Negative numbers in binary SD number system

SD form	After coding
$\bar{1}111$	11010101
$0\bar{1}11$	00110101
$00\bar{1}1$	00001101

to simplify the circuits, encoding of the SD digits is very useful. Two such coding methods are shown in Table 1.2.

Representation of negative numbers in binary SD number system is critical as the MSB no longer decides the sign of an integer. Here, more number of bits from the MSB are to be analysed to detect that the number is negative. Representation of -1 using 4 bits is shown in Table 1.3 using second coding technique. Avoiding the leading zeros if two consecutive $1's$ are found from MSB then the number is negative.

Binary SD number system is used to develop fast algorithms for complex arithmetic operations. In addition or subtraction operation, this number system eliminates carry propagation chains and thus results in a faster implementation. Consider the addition operation between two numbers X and Y. Both these numbers are represented with n-bits. This addition operation generates the sum s and a carry. The carry chain is eliminated by making s_i depends only on x_i, y_i , x_{i-1} and y_{i-1}. This way the addition time becomes independent of operand length. The addition algorithm that can achieve this independence is

1. Compute the intermediate sum (u_i) and carry (c_i) digits as

$$u_i = x_i + y_i - 2c_i \tag{1.18}$$

where

$$c_i = \begin{cases} 1, & \text{if } x_i + y_i \geq 1 \\ \bar{1}, & \text{if } x_i + y_i \leq \bar{1} \\ 0, & \text{if } x_i + y_i = 0 \end{cases} \tag{1.19}$$

2. Calculate the final sum as

$$s_i = u_i + c_{i-1} \tag{1.20}$$

The rules for addition of binary SD numbers are shown in Table 1.4. This table does not include the combinations $x_i y_i = 10$ $x_i y_i = \bar{1}0$ and $x_i y_i = \bar{1}1$. This is because the addition $x_i + y_i$ is a commutative operation.

Table 1.4 Rules for adding binary SD numbers

$x_i\, y_i$	00	01	$0\bar{1}$	11	$\bar{1}\,\bar{1}$	$1\bar{1}$
c_i	0	1	$\bar{1}$	1	$\bar{1}$	0
u_i	0	$\bar{1}$	1	0	0	0

Here, two cases are tested. In the first case, consider that the operands do not have the digit $\bar{1}$. It is shown in the following example of addition. In the conventional binary representation, there will be carry chain that will propagate from the LSB to the MSB. But in this case there is no carry chain exists.

		1	1	...	1	1	y_i
+		0	0	...	0	1	x_i
	1	1	1	...	1		c_i
		$\bar{1}$	$\bar{1}$	...	$\bar{1}$	0	u_i
	1	0	0	...	0	0	s_i

In the second case, if the operands have the digit $\bar{1}$ then the carry chain may exist. For example, if $x_{i-1}y_{i-1} = 01$, then $c_{i-1} = 1$ and if $x_i\, y_i = 0\bar{1}$, then $u_i = 1$ resulting $s_i = u_i + c_{i-1} = 1 + 1$. Thus a new carry is generated. This is explained in the following example:

		0	$\bar{1}$	1	$\bar{1}$	1	1	y_i
+		1	0	0	$\bar{1}$	0	1	x_i
	1	$\bar{1}$	1	$\bar{1}$	1	1		c_i
		$\bar{1}$	1	$\bar{1}$	0	$\bar{1}$	0	u_i
		*	*	*	1	0	0	s_i

At the $*$ marked positions carry signals are generated and propagated.

In Table 1.4, the combination $c_{i-1} = u_i = 1$ occurs when $x_i\, y_i = 0\bar{1}$ and $x_{i-1}\, y_{i-1}$ equals to either 11 or 01. The setting $u_i = 1$ can be avoided in this case by selecting $c_i = 0$ and therefore making u_i equal to $\bar{1}$. We should not, however, change the entry for $x_i\, y_i = 0\bar{1}$ in Table 1.4 to read $c_i = \bar{1}$ and $u_i = 1$. This is because if $x_{i-1}y_{i-1} = \bar{1}\,\bar{1}$ then $c_{i-1} = \bar{1}$. But we still have to set $c_i = \bar{1}$ and $u_i = 1$. Similarly, the combination $c_{i-1} = u_i = \bar{1}$ occurs when $x_i\, y_i = 01$ and $x_{i-1}\, y_{i-1}$ equals to either $\bar{1}\,\bar{1}$ or $0\bar{1}$. We can avoid setting $u_i = \bar{1}$ by selecting in these cases (and in these cases only) $c_i = 0$ and therefore $u_i = 1$. In summary, we can ensure that no new carries will be generated by examining the 2 bits to the right $x_{i-1}\, y_{i-1}$ when determining u_i and c_i, arriving at the rules shown in Table 1.5. Observe that we can still calculate c_i and u_i for all bit positions in parallel. The above example of adding two numbers in binary SD number system is repeated in the following example. Here the carry chain from the

Table 1.5 Modified rules for adding binary SD numbers

$x_i\, y_i$	00	01	01	$0\bar{1}$	$0\bar{1}$	11	$\bar{1}\bar{1}$
$x_{i-1}\, y_{i-1}$	–	Neither is $\bar{1}$	At least one is $\bar{1}$	Neither is $\bar{1}$	At least one is $\bar{1}$	–	–
c_i	0	1	0	0	$\bar{1}$	1	$\bar{1}$
u_i	0	$\bar{1}$	1	$\bar{1}$	1	0	0

Table 1.6 Different representations of X = 7 in binary SD number representation

8	4	2	1
0	1	1	1
1	$\bar{1}$	1	1
1	0	$\bar{1}$	1
1	0	0	$\bar{1}$

LSB to the MSB is avoided. The result of the summation of the two operands is $1\bar{1}1\bar{1}00$. In both cases, the result is same which is $010100 = 20_{10}$.

		0	$\bar{1}$	1	$\bar{1}$	1	1	y_i
+		1	0	0	$\bar{1}$	0	1	x_i
	0	0	0	$\bar{1}$	1	1		c_i
		1	$\bar{1}$	1	0	$\bar{1}$	0	u_i
		1	$\bar{1}$	0	1	0	0	s_i

The binary SD number system is useful in developing the fast algorithms for multiplication, square root or division. As discussed earlier that the SD representation has some redundancy so that an integer has multiple representations. But to develop an efficient implementation of the fast algorithms, it is required that the representation should have the minimum number of nonzero digits. This representation is called minimal SD representation. This minimal representation will result in less number of addition and subtraction operation. For example, the number X = 7 can be represented by different ways as shown in Table 1.6. Here, the SD representation $100\bar{1}$ is minimal representation as it has minimum number of nonzero digits.

1.6.2 SD Representation to Two's Complement Representation

If an architecture is implemented using the binary SD representation, then it is also required to convert the result to the conventional binary representations. Two's complement is the mostly used representation in the practical implementation of the digital systems. Thus conversion from the binary SD representation to two's complement representation is important. One way to achieve this conversion is by encoding the digits of SD representations by binary equivalent. A simple way to achieve this con-

Table 1.7 Rules for converting a binary SD number to two's complement number

y_i	c_i	z_i	c_{i+1}
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0
$\bar{1}$	0	1	1
$\bar{1}$	1	0	1

version is shown here. Binary SD representation to two's complement representation conversion is achieved by satisfying the following relation. Here, y_i is the ith digit in the binary SD representation, z_i is the ith bit corresponding to the two's complement representation, c_i is the previous borrow and the c_{i+1} is the next borrow.

$$z_i + c_i = y_i + 2c_{i+1} \tag{1.21}$$

The rules for converting a number in SD form to its two's complement equivalent are shown in Table 1.7. An example of this conversion is shown below to convert -10_{10} represented in SD representation to its two's complement form.

y_i		0	$\bar{1}$	0	$\bar{1}$	0
c_i	1	1	1	1	0	0
z_i		1	0	1	1	0

The range of representable numbers in the SD method is almost double that of the two's complement representation. $(n + 1)$ bits are required to represent an n-digit SD number in two's complement representation as illustrated below:

y_i		0	1	0	1	0	$\bar{1}$
c_i	0	0	0	0	1	1	0
z_i		0	1	0	0	1	1

The number 19 would be converted to -13 without the use of extra bit.

1.7 Conclusion

In this chapter, we have discussed about BNS. BNS has base of 2 and it has three ways to represent numbers which are Signed number system, One's complement and Two's complement number system. This chapter also discusses how to perform basic addition/subtraction operation using these representation techniques. The real numbers are represented using either fixed point number system or floating point number

system. Floating point number system can produce better accuracy but implementation using floating point numbers is costly. Fixed point number representation is mostly used in system design and its accuracy depends on data width. Some computers use binary SD numbers to achieve faster execution speed. Throughout the book we will discuss architectures using fixed point representations and floating point architectures will be discussed in a separate chapter.

Chapter 2
Basics of Verilog HDL

2.1 Introduction

Verilog is a very popular Hardware Description language (HDL) that is used to model the digital systems. The Verilog HDL can be used at different abstraction levels like from gate level to system design level. This means that a digital circuit can be described by the hardware connectivity or by writing only the behaviour of the circuit. It may be the timing model or the behaviour of the circuit, Verilog provides an easy platform to realize the digital circuits. Thus Verilog HDL provides an easy platform for rapid modelling of the digital circuits.

There are many online tutorials or books available to learn this language. Here, a brief overview of the Verilog HDL is given so that the reader finds a smooth reading process. The major features of this language are discussed in this chapter. In Verilog HDL, a module is written for a digital circuit and in that model the hardware description of that circuit is mentioned. The basic syntax of Verilog module is

```
'timescale 1ns/100ps
module(port_list)
    Declarations:
        Inputs, Outputs,
        Wire, Reg, Parameter

    Statements:
        Initial Statement
        Always Statement
        Continuous Statement
        Module Instantiation
endmodule
```

The Verilog description is written under a **module**. First stage of the **module** is declaration where all the inputs, outputs, intermediate nets or constants are defined. Then the statements or expressions are written. The module is ended with an **endmodule** statement. All delays are specified in terms of time units. The *'timescale*

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_2

directive specified before the module defines the time unit and time precision. Here time unit is 1 ns and precision is 100 ps. Thus all delays must be rounded to multiple of 0.1 ns.

2.2 Verilog Expressions

Before proceeding towards learning Verilog HDL, it is better to know about the different operators and different operands. In these sections, all different types of operators and operands are summarized in Tables 2.1 and 2.2. These expressions will help the readers to understand complex Verilog codes.

2.2.1 Verilog Operands

The Operands in Verilog can be categorized in the following categories:

1. Constants.
2. Parameters.
3. Nets.
4. Registers.
5. Bit-select.
6. Part-select.
7. Memory Element.

These various operands are summarized in Table 2.1.

2.2.2 Verilog Operators

The various operators in the Verilog HDL are classified in the following categories:

1. Arithmetic operators.
2. Bit-wise operators.
3. Reduction operators.
4. Shift operators.
5. Relational operators.
6. Logical operators.
7. Conditional operators.
8. Equality operators.

These operators are summarized in Table 2.2.

2.2.3 Concatenation and Replication

Concatenation is an important operation in Verilog HDL. Concatenation operation can be used to form a bigger expression from smaller expressions. The syntax of this operation is

Table 2.1 Various operands in Verilog HDL

Operands	Syntax

Constants	`256 // Unsized Decimal Constants`
	`4'b0101 // Integer Constants`
	`'b0, 'd8, 'hFB // Unsized Integer Constants`
	`50.5 // Real Constants`
	`"STRONG" // String Constants`

Parameter	`parameter a1 = 4'b1010;//a1 is 4-bit constant`

Net	`wire a1; // a1 is a scalar net.`
	`wire [3:0] a1; // a1 is a 4-bit vector net`

Reg	`reg a1; // a1 is 1-bit register.`
	`reg [3:0] a1; // a1 is 4-bit register.`

Bit-select	`wire[3:0] a1;` `a1[1]; //One bit is selected from a1`
Part-select	`wire[3:0] a1;` `a1[2:1]; //Two bits are selected from a1`
Memory element	`reg [3:0] mem [7:0]//4-bit data, 3-bit address`

```
wire [2:0] a;
wire b;
wire [3:0] y;
assign {y[3],y[2:0]} = {a[2:0], b};
```

Replication is performed by specifying a repetition number. An example of repetition operation is

```
assign y = {3(1'b1)}; // This is equivalent to 3'b111.
```

In Verilog HDL, there are four type of modelling styles to realize the logic circuits and they are

Table 2.2 Various operators in Verilog HDL

Operators	Operation
+	Addition
−	Subtraction
*	Multiplication
/	Division
+	Addition
%	Modulus
|	Bit-wise or reduction OR
&	Bit-wise or reduction AND
∼	Bit-wise negation
∼ |	Bit-wise or reduction NOR
<<	Left shift and fill with zeros
>>	Right shift and fill with zeros
<<<	Left shift but keep sign
>>>	Right shift and fill with MSB
? :	Conditional
<	Greater than
>	Less than
< =	Less than or equal to
> =	Greater than or equal to
||	Logical OR
& &	Logical AND
!	Logical negation
==	Logical equality
! =	Logical inequality
===	Case equality
!==	Case inequality

1. Data Flow Modelling.
2. Behavioural Modelling.
3. Structural Modelling.
4. Mixed Modelling.

2.3 Data Flow Modelling

In data flow modelling, a design is described using continuous assignment. In continuous assignment a value is assigned to a net. The syntax for continuous assignment is

```
assign [delay] net = expression;
```

Here the delay is optional. Data Flow Modelling can be understood using design of a simple 2:1 MUX. Boolean expression of a simple 2:1 MUX is

$$y = a\bar{s} + bs \tag{2.1}$$

Here, if the control signal s is zero then a is passed to y otherwise b is passed. In data flow modelling, the Boolean expression can be directly modelled. A simple 2:1 MUX designed using the data flow modelling is shown below:

```
module mux_df(
    input  a,b,s,
    output y
    );
wire sbar;
assign y = (a & sbar)|(s & b);
assign sbar = ~s;
endmodule
```

Here in the design of the MUX, *sbar* is an intermediate net and thus it is called as wire. A wire can be a single bit or can be a vector also. An 8-bit wire is defined as

```
wire [7:0] a;
```

Many software tools consider a wire as 1 bit when undefined. The same code of the MUX can be written in the following format also: .

```
module mux_df(a,b,s,y);
input  a,b,s;
output y;
wire sbar;
assign y = (a & sbar)|(s & b);
assign sbar = ~s;
endmodule
```

In defining a module, it must be taken care of that all the inputs, outputs and the wires are defined. Another way of designing the 2:1 MUX using the continuous assignment is using the conditional assignment operator.

```
module mux_cs(
    input  a,b,s,
    output y
    );
wire s;
assign y = (s == 0)? a : b;
endmodule
```

Multiple assignments can be written in a single continuous assignment as

```
assign y = (s == 0)? a : 'bz,
       y = (s == 1)? c : 'bz;
```

This is equivalent to the assignment of multiple assignments as

```
assign y = (s == 0)? a : 'bz;
assign y = (s == 0)? c : 'bz;
```

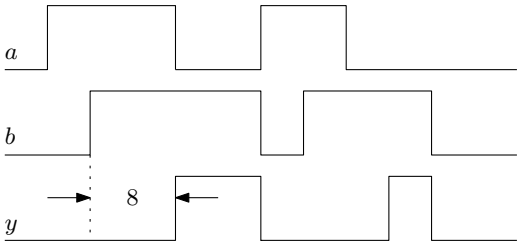


Fig. 2.1 Example of net delay

Assignment can be done during the net declaration also. This is shown below:

```
wire reset = 1'b0;
```

This is equivalent to the following:

```
wire reset;
assign reset = 1'b0;
```

The delay also can be introduced during net declaration. But net delay and assignment delay are different. It is explained by the following example:

```
wire #5 y; \\ net delay
assign #3 y = a & b; \\ assignment delay
```

The result is assigned to y after total delay of net delay plus assignment delay. This delay is of 8 units. This example is shown in Fig. 2.1.

2.4 Behavioural Modelling

In behavioural modelling, a digital circuit is modelled by knowing the behaviour. The behavioural model is described using the procedural constructs and these are

1. Initial Statement.
2. Always Statement.

2.4.1 Initial Statement

The initial statement is used to initialize a variable. This statement is executed only once. The syntax for Initial statement is

```
initial
[Timing Control] Procedural Statements
```

An example of Initial statement is

```
initial begin x = 1'b0; y = 1'b0; end
```

Here, the variables x and y are initialized by 0.

2.4.2 Always Statement

The always statement executes in a loop repeatedly. Its syntax is

```
always
 [Timing Control] procedural statements
```

An example of the always statement is

```
always #5 clk = ~clk;
```

Here, the always statement is used to create the clock. Repeatedly value of the right-side net is updated to the left-side expression after each 5 units delay.

Note that only a register data type can be assigned in the above two statements. Initial statements are used to initialize input vectors or scalars whereas the always statement is used to execute statements repeatedly. In a module, many initial or always statements can be used. These statements are executed concurrently. But the expressions inside the initial or always statement are executed sequentially.

2.4.3 Timing Control

Timing control is of two types which are

1. Delay Control.
2. Event Control.

Delay Control

The delay control specifies the delay after which the statement is executed. An example of this delay control is

```
always
begin
#5
clk = 0;
#5
clk = 1;
end
```

In the above example, the signal clk gets value 0 after 5 unit delay and again the clk signal get value 1 after another 5 unit of delay.

Event Control

There are two types of event controls which are

1. Edge-Triggered Event Control: In case of edge triggering-based event control, the procedural statement is executed when a low-to-high or high-to-low event has occurred. An example of this control is

```
always @(posedge clk)
q = d;
```

   In the above example, the value of d is assigned to q whenever there is a low-to-high transition in the clock signal. Similarly, negedge specifies high-to-low transition.
2. Level-Triggered Event Control: In level triggering-based event control, execution of a statement depends on change of value or change of level of the control signal. An example of this level control is

```
wait (en)
q = d;
```

   Here, the value of d is assigned to q whenever the value of en is equal to logic 1.

An example of behavioural coding to implement the same 2:1 MUX is shown below:

```
module mux_bh(input  a,b,s,output y);
reg y;
always @( s or a or b )
begin
    if( s == 0)
     y = a;
    else
     y = b;
end
endmodule
```

Here, an if-else loop is used to describe the behaviour of the MUX. Notice that the output is a register data type. Similarly, other type of loops can also be used to describe the behaviour of circuit. Their format is shown in Table 2.3 with examples.

If combinational circuit is to be implemented using the behavioural modelling then sensitivity list can be ignored by placing the operator after **always** statement. The **always** statement automatically considers the inputs as the sensitivity list. This is written as

```
module mux_bh1(input  a,b,s,output y);
reg y;
always @( *)
begin
     if( s == 0) y = a;
     else y = b;
end
endmodule
```

Table 2.3 Various loops in behavioural modelling

Loop	Operation	Example

Forever	In a Forever-loop statements are exeucuted continously	`initial begin`
		`clock = 0;`
		`#5 forever`
		`#10 clock = $\sim$clock;`
		`end`

Repeat	In a Repeat-loop statements are executed repeatedly for specified number of times	`repeat (count)` `sum = sum + 5;`

While	In While-loop statements are executed untill a condition is satified	`while (count > 5)`
		`begin`
		`sum = sum + 5;`
		`end`

For	In a For-loop statements are executed for a certain number of times	`integer k;`
		`for (k = 0; k < 5; k = k + 1)`
		`begin`
		`sum = sum + 5;`
		`end`

Another way of describing the behaviour of a circuit is using the case statement. The case statement covers several output combinations of the output depending on the inputs. The 2:1 MUX using the case statement is shown below:

```
module mux_case(
    input  a,b,s,
    output y
    );
reg y;
always @(s or a or b)
begin
   case(s)
       0 : y = a;
       1 : y = b;
   endcase
end
endmodule
```

2.4.4 Procedural Assignment

2.4.4.1 Blocking Procedural Statement

In the procedural assignment statement where the operator '=' is used is called blocking procedural statement. In case of the blocking procedural statements, a blocking procedural statement is executed completely before the next statement is executed. An example is shown below:

```
begin
x = #5 1'b1;
x = #6 1'b0;
x = #7 1'b1;
end
```

Here, three blocking procedural statements are defined inside the always loop. Here, the first statement is executed completely before the second statement and 1 is assigned to the x after 5 time units. Then after 6 time units x is again updated with value 0. This way the statements are evaluated sequentially. Simulation diagram is shown in Fig. 2.2 for this example.

2.4.4.2 Non-blocking Procedural Statement

In the procedural assignment statement where the operator '$<=$' is used is called non-blocking procedural statement. A procedural statement has two steps, viz., execution and assignment. The execution of the non-blocking procedural statements is started at the same time but assignment to the targets is not blocked due to delays. Assignment to the targets is scheduled based on the delay associate to the execution of respective statements plus the intra-statement delay. The same example is taken here to illustrate the non-blocking assignment.

```
begin
x <= #5 1'b1;
x <= #6 1'b0;
x <= #7 1'b1;
end
```

Consider that execution of the three statements is started at time 0. Assignment of first statement took place after 5 unit time, assignment of the second statement took place after 6 unit time and the assignment of the third statement took place after 7

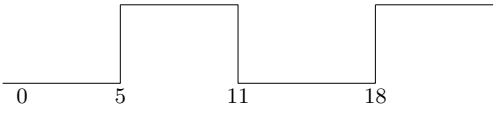


Fig. 2.2 Output for the blocking procedural statement

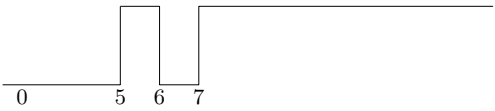


Fig. 2.3 Output for the non-blocking procedural statement

time unit. This situation is shown in Fig. 2.3. Understanding the difference between these two assignment techniques is important. Here for the above example, if the intra-statement delays are not mentioned then output for both the cases is same. In this case, the assignment for the non-blocking statements depends on the execution delays.

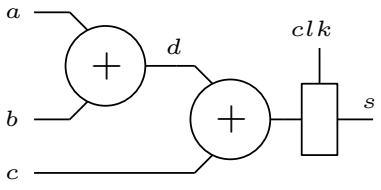
The difference between the blocking and non-blocking statement can be understood by another important example where a three-input adder is modelled using behavioural statement. The Verilog code for three-input adder using non-blocking statement is shown below:

```
module mac(
input clk,input [3:0]a,b,c, output reg [3:0] s);
 reg [3:0] d;
 always @(posedge clk)
 begin
 d <= a + b;
 s <= c + d;
 end
endmodule
```

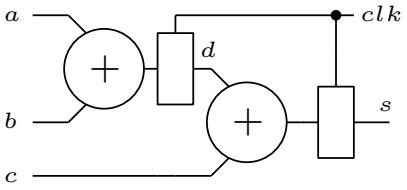
The hardware which is modelled by this code is shown in Fig. 2.4a. Here, one register is placed after addition of a and b and one register is placed at the last. The same three-input adder is modelled using the blocking statement in the following code:

```
module mac(
input clk,input [3:0]a,b,c, output reg [3:0] s);
 reg [3:0] d;
 always @(posedge clk)
 begin
 d = a + b;
 s = c + d;
 end
endmodule
```

The hardware which is modelled by this code is shown in Fig. 2.4b. Here, register is placed only at the output. Thus blocking and non-blocking statements result in different hardware. This is why Verilog modelling of hardware is sometimes called as register transfer logic (RTL) as placing of registers can be controlled using blocking and non-blocking statements.



(a) 3-input adder modelled using non-blocking statement.



(b) 3-input adder modelled using blocking statement.

Fig. 2.4 Three-input adder modelled using non-blocking and blocking statement

2.5 Structural Modelling

In the structural modelling, a digital circuit is realized using smaller sub-blocks. These sub-blocks can be pre-defined gates or any smaller sub-blocks. Structural modelling is of two types which are

1. Gate-Level modelling.
2. Hierarchical modelling.

2.5.1 Gate-Level Modelling

In the gate-level modelling, a design is realized using pre-defined gates which are already known to compiler. These basic gates can be a NOT gate, an OR gate or can be an AND gate. A two input OR gate defined in Verilog as

```
or o1( op, I1, I2);
```

there output is defined first and then the inputs. Similarly the other gates are defined. The gate-level schematic of the 2:1 MUX is shown in Fig. 2.5. Here two AND gates, one NOT gate and one OR gate are used. The 2:1 MUX can be designed using the gate-level modelling as

```
module mux_gl(
    input  a,b,s,
    output y
    );
wire  q1,q2,sbar;
not n1( sbar,s);
and a1( q1, sbar, a);
and a2( q2, s, b);
or o1( y, q1, q2);
endmodule
```

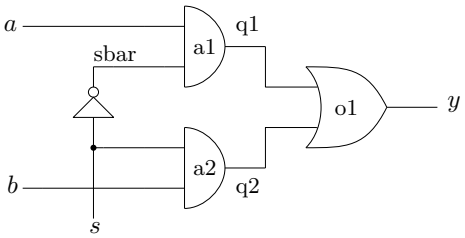


Fig. 2.5 Gate-level schematic of the 2:1 MUX

2.5.2 Hierarchical Modelling

In hierarchical modelling, a digital circuit is realized by exploring the hierarchy of that design. Initially the smaller sub-blocks present in the hierarchy are designed and then the main design is realized using the smaller sub-blocks. This modelling technique is called module instantiation. Here the main design is called as top module. An example of hierarchical modelling is given below for a 4:1 MUX. The 4:1 MUX using the 2:1 MUX is shown in Fig. 2.6. A 4:1 MUX for 1-bit data uses three 2:1 MUXes. A 2:1 MUX is designed in the previous sections. Any of the designs can be used here to realize a 4:1 MUX.

```
module mux_4_1(
     input a1,a2,a3,a4,
      input [1:0]s,
      output y
     );
wire t1,t2;
mux_df m1(a1,a2,s[0],t1);
mux_df m2(a3,a4,s[0],t2);
mux_df m3(t1,t2,s[1],y);
endmodule
```

In the above Verilog code three 2:1 MUXes are used which are previously modelled in data flow modelling. There are two techniques of calling the sub-blocks which are

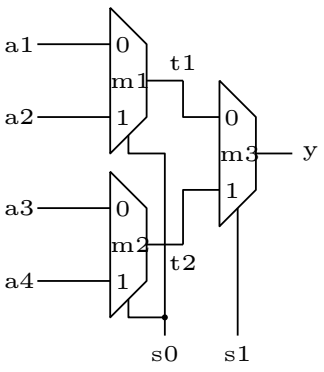


Fig. 2.6 Gate-level schematic of the 4:1 MUX using 2:1 MUX

1. Module instantiation using position of the input/output ports.
2. Module instantiation by mentioning the input/output ports.

The above code of the 4:1 MUX is written using the position of the input and outputs. In this technique, it is required to maintain the proper sequence of the input and output ports while calling. On the other hand, if the modules are instantiated by mentioning the ports then the sequence need not be maintained. The above code can be rewritten using this technique as

```
module mux_4_1(
    input a1,a2,a3,a4,
     input [1:0]s,
     output y
    );
wire t1,t2;
mux_df m1(.a(a1),.b(a2),.s(s[0]),.y(t1));
mux_df m2(.a(a3),.b(a4),.s(s[0]),.y(t2));
mux_df m3(.a(t1),.b(t2),.s(s[1]),.y(y));
endmodule
```

In the case of module instantiation, there may be a situation where some of the output ports of the sub-modules are left unconnected. The syntax for representing that is

```
mux_df m1(.a(a1),.b(a2),.s(s[0]),.y());//by writing ports
mux_df m2(a,,s,y); // by position
```

The expression for the unconnected ports is left blank. In the first MUX, output port is unconnected and in the second MUX input port is left unconnected. Unconnected output ports are unused but the unconnected input ports are connected to Z.

2.6 Mixed Modelling

In the mixed modelling style, the design styles mentioned above can be mixed to realize a digital circuit. An example of mixed design style is shown below to implement the same 4:1 MUX.

```
module mux_4_1_mix(
    input a1,a2,a3,a4,
     input [1:0] s,
     output y
    );
reg y;
wire t1,t2;
mux_df m1(a1,a2,s[0],t1);
mux_gl m2(a3,a4,s[0],t2);
always @( s[1] or t1 or t2 )
begin
   if( s[1] == 0)
      y = t1;
    else
```

```
        y = t2;
end
endmodule
```

Here, one MUX is designed by data flow modelling, one MUX is designed using the gate-level modelling and the third MUX is designed using the behavioural modelling. The overall top module is modelled using the module instantiation.

2.7 Verilog Function

In writing Verilog code of a complex design, one can write specific expressions in a Verilog function and can call that function multiple times. Verilog functions helps designers to comfortably write their code. Verilog functions have the following characteristics:

- A Verilog function cannot drive more than one output but can have multiple inputs.
- Functions are defined in the module in which they are used.
- It is possible to define functions in separate files and use compile directive **'include** to include the function.
- Functions cannot include timing delays, like **posedge**, **negedge**, # **delay**, which means that functions should be executed in 'zero' time delay.
- The variables declared within the function are local to that function. The order of declaration within the function defines how the variables are passed to the function.
- Functions can be used for modelling combinational logic. Functions can call other functions, but cannot call tasks (described below).

An example of a Verilog function is shown below:

```
module function_test(input [3:0] a1,a2,a3,a4, output reg [3:0] c)
    ;
reg [3:0] b1,b2;

function [3:0] sum(input [3:0] a,b);
begin sum = a + b; end
endfunction

always @*
begin
b1 = sum(a1,a2);
b2 = sum(a3,a4);
c = sum(b1,b2);
end
endmodule
```

2.8 Verilog Task

Just like other programming languages, Verilog also has tasks. Tasks are also called as sub-routines. In a task, part of programme is written and the task is called many times in the main code. This way, modelling a complex design is easy. A Verilog task has following characteristics:

- Verilog task can have multiple inputs and multiple outputs.
- Tasks are defined in the module in which they are used.
- It is possible to define a task in a separate file and use the compile directive **'include** to include the task in the file which instantiates the task.
- Task can include timing delays, like **posedge**, **negedge**, **#delay** or **wait**.
- The variables declared within the task are local to that task. The order of declaration within the task defines how the variables passed to the task by the caller are used.
- Tasks can call another task or function.
- tasks can be used for modelling both combinational and sequential logic.
- A task must be specifically called with a statement, it cannot be used within an expression as a function can.

A simple example of a Verilog task is shown below:

```
module task_test(input [3:0] a1,a2,a3,a4, output reg [3:0] c);
reg [3:0] b1,b2;

task sum(input [3:0] a,b, output [3:0] s);
begin s = a + b; end
endtask

always @*
begin
sum(a1,a2,b1);
sum(a3,a4,b2);
sum(b1,b2,c);
end
endmodule
```

2.9 File Handling

In order to verify a complex design, we may need huge set of data elements. In such cases, reading data elements from an external file or writing into an external file can be very useful. Here, we will discuss how data elements can be read from or written to a text file using Verilog HDL.

2.9.1 Reading from a Text File

The command $fopen()$ opens a file $input\_vector.txt$ in read mode. In the loop of j, jth element of $input\_vector.txt$ ($of0$) stored in register A1. Then the content of A1 is stored in jth location of b which is an array of registers. The text file should be in the project directory.

```
module mem_read(C1,ada,clk);
    input [9:0] ada;
     input clk;
    integer of0;
    output reg [17:0] C1;
    reg [17:0] A1;
     reg [17:0] b1 [1023:0];
     integer j;
     initial begin
     of0=$fopen("input_vector.txt","r");
     for (j=0;j<=1023;j=j+1)
     begin
        $fscanf(of0,"%d\n",A1);
         #1;
        b1[j] = A1;
    end
     $fclose(of0);
     end
    always @ (posedge clk)
    begin
     C1 = b1[ada];
     end
endmodule
```

2.9.2 Writing into a Text File

The value of the register A is written to the text file $output\_vector.txt$ only if en signal is high and with positive edge of clock. The clock is considered to be of period 10 ns. The output file will be written in the project directory.

```
module mem_write(A,en,clk);
    integer of0;
    input [17:0] A ;
     input en,clk;
     integer j;
     initial begin j = 0; end
     always @(posedge clk)
     begin
     if (en)
          begin
     of0=$fopen("output_vector.txt","w");
       for (j=0;j<=4;j=j+1)
```

```
            begin
        $fdisplay(of0,"%d\n",A);
        #10;
            end
             end
        else
        $fclose(of0);
        end
endmodule
```

2.10 Test Bench Writing

In the previous sections, various modelling styles are discussed. A simple design of 2:1 MUX is taken to illustrate the design styles. Test benches are written to verify the Verilog codes. Test benches contain the input test vectors and clock information to verify the design code. Here, the main Verilog file is Unit Under Test (UUT) and the test bench is used to give test vectors to verify the UUT. This is shown in Fig. 2.7. An example of a test bench for the 2:1 MUX is shown below:

```
module mux_tb;
    // Inputs
    reg a; reg b; reg s;
    // Outputs
    wire y;
    // Instantiate the Unit Under Test (UUT)
    mux_gl uut (
        .a(a),
        .b(b),
        .s(s),
        .y(y));
    initial begin
        // Initialize Inputs
        a = 0; b = 0; s = 0;
        // Wait 100 ns for global reset to finish
        #100;
        a = 1; b = 0; s = 0;
        #20;
        a = 1; b = 0; s = 1;
        #20;
        a = 1; b = 1; s = 0;
        // Add stimulus here
    end
endmodule
```

Here the inputs are given under initial statement. The input variables are register type as they are mentioned under the initial statement and the outputs are wire type.

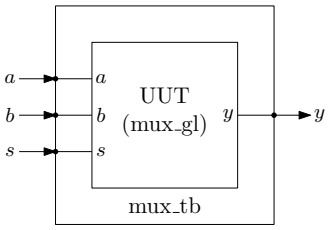


Fig. 2.7 Verification environment for Verilog files

2.11 Frequently Asked Questions

Q1. **How to modify a parameter defined within a module (within a hierarchy)**?
A1. In a structural modelling of a complex digital system, if it is required to change the size of the whole design then it is very difficult to change size of the sub-modules individually. Thus it is better to use the *parameter* operand to define the size. It is shown below for a simple 2:1 MUX.

```
module muxM(A,B,S,Y);
parameter M = 4;
    input [M:0] A,B;
    output [M:0] Y;
    input S;
assign Y = (S)? B : A;
endmodule
```

Here, the size of the MUX is M. A 4:1 MUX is designed using these 2:1 MUXes. Now if we want to change the size of the 4:1 MUX then size of all the three 2:1 MUXes should be changed. The parameters of the sub-module muxM can be override by the *defparam* operand. This is shown below:

```
module Mux4_1(A,B,S,Y);
parameter M1 = 7;
input [M1:0] A,B;
input S;
output [M1:0] Y;
wire [M1:0] Y1,Y2;
defparam mux1.M = M1, mux2.M = M1, mux3.M = M1;
muxM mux1(A,B,S,Y1);
muxM mux2(A,B,S,Y2);
muxM mux3(Y1,Y2,S,Y);
endmodule
```

In order to override the parameter (M) or multiple parameters, the path of the parameter is given in the *defparam* operand. Here the path is *mux1.M*.

Q2. **What is the difference between asynchronous reset and synchronous reset?**
A2. The sequential designs are generally cleared using a *reset* signal. This *reset* signal can be asserted in two ways, viz., in synchronization with clock signal or without syncing with clock signal. The behavioural model of D flip-flop shown in Sect. 4.2.3

is an example of synchronous reset. Example of asynchronous reset is shown below for the same D flip-flop:

```
module DFF(q,clk,reset,d);
    input d,clk,reset;
    output reg q;
initial begin q=0; end
always @ (posedge clk, posedge reset) begin
 if (reset)
  q <= 0;
 else
 q<= d ;
end
endmodule
```

Q3. **What is the importance of a default clause in a case construct?**
A3: The *case* statement is frequently used to model any combinational and sequential circuit. In the Verilog code using case statement, if all the combinations of the input variable are not mentioned then a latch is inferred. This is explained with the following example code of simple LUT.

```
module LUT(
   input [1:0] s,
   output reg y);
always @ (s )
case(s)
2'b00 : y = 0;
2'b01 : y = 1;
2'b10 : y = 1;
endcase
endmodule
```

Here, the output is not defined for $s = 11$ and at this condition output is equal to the previous value due to the inferred latch. The corresponding hardware model realized by the XILINX tool is shown in Fig. 2.8. The inferring of latch can be avoided by mentioning default for the unused conditions. Realization of the same code using default statement is shown in Fig. 2.9. Note that, without default statement two LUTs are used instead of one.

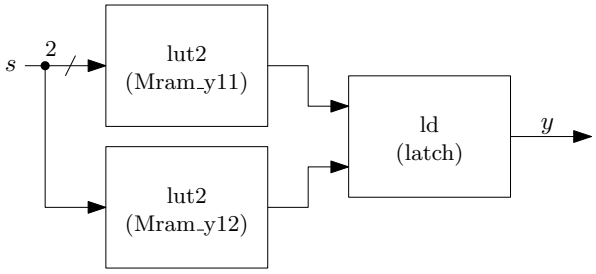


Fig. 2.8 Realization of the LUT without default statement

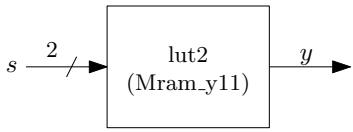


Fig. 2.9 Realization of the LUT with default statement

```
module LUT(
   input [1:0] s,
   output reg y);
always @ (s )
case(s)
2'b00 : y = 0;
2'b01 : y = 1;
2'b10 : y = 1;
endcase
endmodule
```

The latch also can be inferred in case of incomplete specification of loop statements. One example is shown below for incomplete *if-else* statements.

```
module LUT(
   input [1:0] s,
    output reg y
    );
always @ (s )
begin
   if( s == 2'b00)
      y = 0;
    else if( s == 2'b01)
        y = 1;
    else if( s == 2'b10)
    y = 1;
end
```

Q4. **What are the different CASE statements and their use?**
A4. There are two types of case statements which are casex and casez. In the control expression, the statements casez and casex allow use of x and z in the don't care bits. An example of casex is given below:

```
module encoder(
   input [2:0] s,
    output reg y2,y1,y0
    );
always @(s)
  casex (s)
    3'b1?? : y2 = 1'b1;
    3'b01? : y1 = 1'b1;
    3'b001 : y0 = 1'b1;
    default: {y2,y1,y0} = 3'b000;
  endcase
```

Here, $y2$ is 1 when the s_2 bit is 1. Other don't care bits can be undefined or unknown. The use of these case statements can be very useful when dealing with don't care bits as any bit can be undefined or in high impedance due to designer's fault or fabrication fault.

Q5. **How to instantiate same module multiple times in Verilog?**
A5. Sometimes we have to instantiate same module multiple times. For example, in designing a 4-bit 2:1 MUX we need four 1-bit 2:1 MUXes. In that case, we have to instantiate the 2:1 MUX module four times. It is difficult in case of very complex design. An easiest way is to use the *generate* statement. Use of this statement is shown below. The module *mux_cs* is explained in Sect. 2.3.

```
module Mux2_1_4bit(a,b,s,y);
input [3:0] a,b;
input s;
output [3:0] y;
genvar i;
generate for(i=0; i<4; i = i+1)
begin : Mux2_block
mux_cs m1(a[i],b[i],s,y[i]);
end
endgenerate
endmodule
```

Q6. **How to execute statements in parallel in behavioural coding style inside always or initial statement?**
A6. A parallel block can be inserted in a behavioural style using statements fork and join. An example of this is shown below:

```
fork
begin
x = #5 1'b1;
x = #6 1'b0;
x = #7 1'b1;
end
join
```

Here, all the statements under begin and end are executed concurrently. Bit 1 is assigned to x after 5 unit time and after 1 unit time bit 0 is assigned to x.

Q7. **Write a Verilog code for 8-bit Arithmetic Logic Unit (ALU) to perform different arithmetic functions?**
A7. The design of ALU using Verilog code is introduced here to show how functionally various arithmetic functions can be performed using simple Verilog operators.

```
module alu(
            input [7:0] A,B,   // ALU 8-bit Inputs
            input [3:0] ALU_Sel,// ALU Selection
            output [7:0] ALU_Out, // ALU 8-bit Output
            output CarryOut // Carry Out Flag
    );
    reg [7:0] ALU_Result;
    wire [8:0] tmp;
    assign ALU_Out = ALU_Result; // ALU out
    assign tmp = {1'b0,A} + {1'b0,B};
    assign CarryOut = tmp[8]; // Carryout flag
    always @(*)
    begin
        case(ALU_Sel)
        4'b0000: // Addition
           ALU_Result = A + B ;
        4'b0001: // Subtraction
           ALU_Result = A - B ;
        4'b0010: // Multiplication
           ALU_Result = A * B;
        4'b0011: // Division
           ALU_Result = A/B;
        4'b0100: // Logical shift left
           ALU_Result = A<<1;
         4'b0101: // Logical shift right
           ALU_Result = A>>1;
         4'b0110: // Rotate left
           ALU_Result = {A[6:0],A[7]};
         4'b0111: // Rotate right
           ALU_Result = {A[0],A[7:1]};
          4'b1000: //  Logical and
           ALU_Result = A & B;
          4'b1001: //  Logical or
           ALU_Result = A | B;
          4'b1010: //  Logical xor
           ALU_Result = A ^ B;
          4'b1011: //  Logical nor
           ALU_Result = ~(A | B);
          4'b1100: // Logical nand
           ALU_Result = ~(A & B); b,m ,v9kmv9v
          4'b1101: // Logical xnor
           ALU_Result = ~(A ^ B);
          4'b1110: // Greater comparison
           ALU_Result = (A>B)?8'd1:8'd0 ;
          4'b1111: // Equal comparison
            ALU_Result = (A==B)?8'd1:8'd0 ;
          default: ALU_Result = A + B ;
        endcase
    end
endmodule
```

2.12 Conclusion

In this chapter, a brief discussion on Verilog HDL is given as Verilog HDL will be used in the next chapters in modelling of digital systems. There is always a debate that which programming style is better to model a system. In designing a complex digital system, all the four programming styles are required. In the subsequent chapters, we have followed the structural modelling style. This is because the structural modelling helps to gain more knowledge of designing efficient circuits. But the behavioural model is also useful for rapid prototyping of a system. Also it is easier to design some circuits like FSM using behavioural style. Thus a designer can use behavioural or data flow modelling to design the basic blocks. But the sub-blocks must be integrated using structural modelling. This way the circuit can be optimized for power, area and speed.

Chapter 3
Basic Combinational Circuits

3.1 Introduction

In a combinational circuit, the output is a pure function of inputs only whereas in sequential circuits output is not only a function of the present inputs but also a function of previous output status. This means combinational circuits do not have memory. Combinational circuits implement a particular Boolean expression and are also known as time-independent circuits.

Complex digital systems cannot be a pure combinational circuit. Both sequential and combinational circuits are required for a digital system implementation. Examples of combinational circuits are Adder, Subtractor, Multiplexer, De-Multiplexer, Encoder, Decoder, etc. In this chapter, various combinational circuits will be discussed. This will be a very brief discussion as there are many books and online tutorials available on this topic.

3.2 Addition

Addition operation is the most important basic arithmetic operation which is mostly used in implementing the digital systems. A two-input adder circuit receives input operands a and b and generates two outputs s and c_{out}. Here, s is the summation output, and c_{out} is the carry out representing that the overflow is occurred. The truth table for two-input addition is shown in Table 3.1. The two-input addition circuit is commonly known as Half Adder (HA). The logical expression for the HA derived from the truth table is

$$s = a \oplus b \tag{3.1}$$

$$c_{out} = a.b \tag{3.2}$$

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_3

Table 3.1 Characteristic table of a HA

a	b	s	c_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

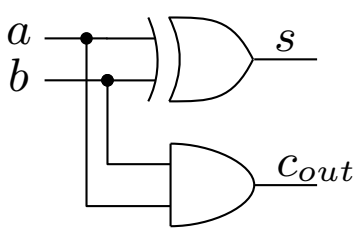


Fig. 3.1 Gate level logic diagram for the HA

Table 3.2 Characteristic table of a FA

a	b	c_{in}	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The circuit diagram for the HA is shown in Fig. 3.1. A Full Adder (FA) is an arithmetic circuit that receives three inputs and generates two outputs. In addition operation of two-input operands, the HA circuit do not consider the carry input. But in the FA circuit, a third input c_{in} is considered and thus FA is called the complete adder. The truth table of the FA is shown in Table 3.2. The logical expressions for the FA derived from the truth table are shown below

$$s = a \oplus b \oplus c_{in} \tag{3.3}$$

$$c_{out} = a.b + a.c_{in} + b.c_{in} \tag{3.4}$$

The gate level logic diagram for the FA is shown in the Fig. 3.2. Here the FA is implemented using two HA circuits.

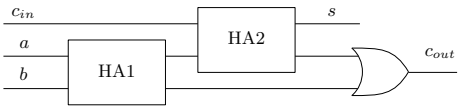


Fig. 3.2 Gate level realization of FA using HA

3.3 Subtraction

Subtraction operation is also very important operation in implementing digital systems like addition operation. A two-input subtraction circuit receives input operands (a and b) and generates two outputs (d and b_{out}). Here, d is the difference output and b_{out} is the borrow out representing that the higher number is subtracted from lower number. The truth table for two-input subtraction is shown in Table 3.3. The two-input subtraction circuit is commonly known as Half Subtractor (HS). The logical expression for the HS derived from the truth table is

$$d = a \oplus b \tag{3.5}$$

$$b_{out} = \bar{a}.b \tag{3.6}$$

The circuit diagram for the HS is shown in Fig. 3.3. A Full Substractor (FS) is an arithmetic circuit that receives three inputs and generates two outputs. In the subtraction operation of two-input operands, the HS circuit does not consider the borrow in (b_{in}) input. But in the FS circuit, a third input b_{in} is considered and thus FS is called as complete subtractor. The truth table of the FS is shown in Table 3.4. The logical expressions for the FS derived from the truth table are shown below

$$s = a \oplus b \oplus c_{in} \tag{3.7}$$

$$c_{out} = \bar{a}.b_{in} + \bar{a}.b + b.b_{in} \tag{3.8}$$

The gate level logic diagram for the FS is shown in Fig. 3.4. Here FS is implemented using two HS circuits.

Table 3.3 Characteristic table of a HS

a	b	d	b_{out}
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

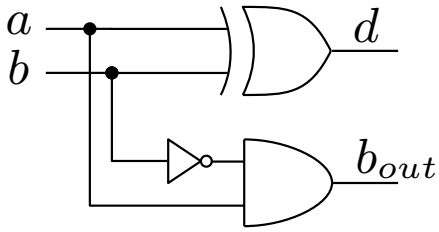


Fig. 3.3 Gate level logic diagram for the HS

Table 3.4 Characteristic table of a FS

a	b	b_{in}	d	b_{out}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

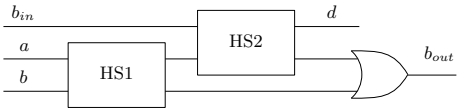


Fig. 3.4 Gate level realization of FS using HS

3.4 Parallel Binary Adder

Previously we have discussed the addition operation between two 1-bit operands. The addition operation between two n-bit operands can be performed using *n* FA blocks. Here, a 4-bit adder which adds two 4-bit operands is discussed and it generates a 4-bit sum output and a *carry out* output. Each bit from the two operands is added in parallel by 4 FA blocks. The architecture of this parallel binary adder is shown in Fig. 3.5.

The first FA block receives the initial carry input which can be kept as 0. The first FA block generates a *carry out* signal which is passed to the second FA block. The second FA block then computes its sum and *carry out* signal. This way *carry out* signal propagates to the last FA block. Thus this structure is known as Ripple Carry Adder (RCA). This block has a delay of $n.t_{FA}$ for n bits where t_{FA} is the delay of one FA block. More about the fast adders are discussed in Chap. 7.

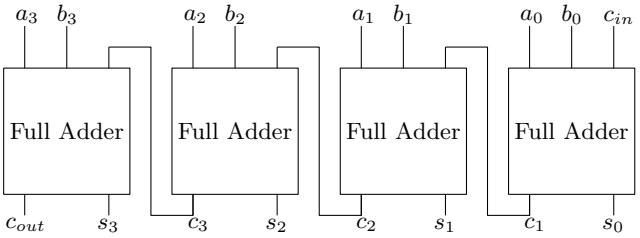


Fig. 3.5 Architecture of 4-bit ripple carry adder

3.5 Controlled Adder/Subtractor

Controlled adder/subtractor block is one of the most important combinational circuits in designing digital systems. In many applications, it is required to perform addition and subtraction operation by a single block. The controlled adder/subtractor block performs addition and subtraction operation depending on a control signal.

The architecture for 4-bit adder/subtractor is shown in Fig. 3.6. In two's complement representation, two operands a and b are added as $a + b$. In order to subtract b from the operand a, the first two's complement of b is taken and then added to a. This can be written as $a - b = a -$ one's complement of $b + ulp$ where $ulp = 2^0$. The addition operation is performed when the $ctrl$ input is low and the subtraction operation is performed when the $ctrl$ input is high. The XOR gates are used to take the one's complement of b and the $ctrl$ input is connected to the first full adder as input carry to add ulp.

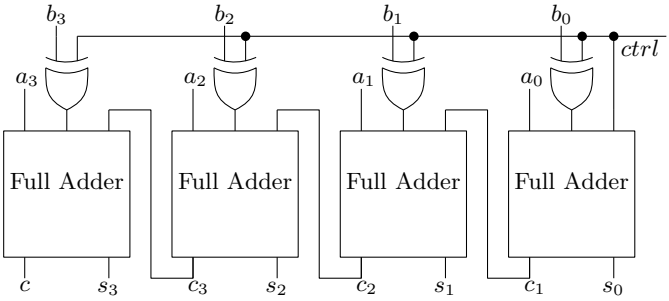


Fig. 3.6 Architecture of 4-bit adder/subtractor

3.6 Multiplexers

Multiplexer is a circuit which selects one signal between two or many signals based on a control signal. A 2:1 Multiplexer circuit selects from two inputs. Multiplexer circuits have many use in digital circuits for data sharing, address bus sharing, control signal selection, etc. The Boolean expression for a simple 2:1 Multiplexer circuit is shown below

$$y = \overline{s}.x0 + s.x1 \tag{3.9}$$

The Multiplexer circuit chooses input $x0$ when the control signal s is low and selects input $x1$ when the control signal is high. The details of Multiplexer circuits are discussed in Chap. 2.

3.7 De-Multiplexers

A De-Multiplexer sends a input signal to different output ports depending on a control signal. A De-Multiplexer can do the reverse operation that a multiplexer does. The basic Boolean expression for 1:2 De-Multiplexer is given below

$$y0 = \overline{s}.x \tag{3.10}$$

$$y1 = s.x \tag{3.11}$$

Here, s is the control signal. The input signal x is passed to the output line $y0$ when s is logic zero and x is connected to the output line $y1$ when s is logic one. The architecture of 1:4 De-Multiplexer using 1:2 De-Multiplexer is shown in Fig. 3.7.

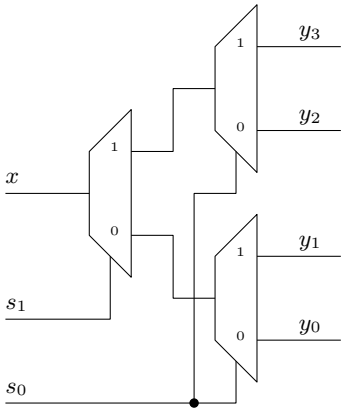


Fig. 3.7 A schematic for a 1:4 DeMUX

Table 3.5 Characteristic table of a 2-4 decoder

en	s_1	s_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

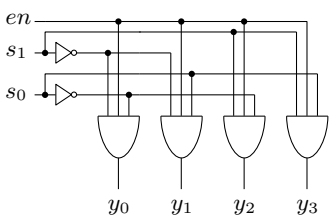


Fig. 3.8 Circuit diagram of a 2-4 decoder

3.8 Decoders

A decoder circuit is used to change or decode a code into a set of signals. The decoder receives n signals and produces 2^n output signals. Only one output signal is at logic level 1 at a time. There may be 2-4 decoder, 3-8 decoder or 4-16 decoder circuit. The truth table for a 2-4 decoder is shown in Table 3.5.

The Boolean expressions for each output signal are

$$y_0 = \overline{s_0}.\overline{s_1}.en \tag{3.12}$$

$$y_1 = s_0.\overline{s_1}.en \tag{3.13}$$

$$y_2 = \overline{s_0}.s_1.en \tag{3.14}$$

$$y_3 = s_0.s_1.en \tag{3.15}$$

The circuit diagram of 2-4 decoder is shown in Fig. 3.8. The circuit diagram is very similar to that of De-Multiplexer. But in case of De-Multiplexer one input is passed to different output lines but on the other hand decoder decodes a n input code. Higher order decoder circuits can be easily implemented using the smaller decoder circuits.

3.9 Encoders

Encoders does the opposite function that a decoder does. An encoder receives 2^n input signals and converted them into a code of n output lines. Encoders are very useful in sending coded messages in the field of communication. The encoders can

Table 3.6 Characteristic table of a 4-2 Binary Encoder

a	b	c	d	y_1	y_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

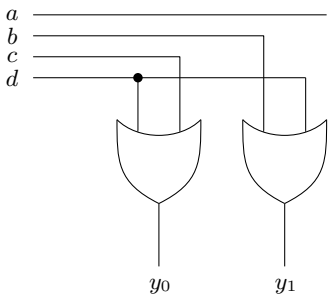


Fig. 3.9 Circuit for a 4-2 encoder

be available as 4-2 encoder, 8-3 encoder or 16-4 encoder. The truth table for a 4-2 encoder is shown in Table 3.6.

The Boolean expressions for each output signal are

$$y_0 = b + d \tag{3.16}$$

$$y_1 = c + d \tag{3.17}$$

The circuit diagram for 4-2 encoder is shown in Fig. 3.9. Encoders are used to reduce the number of bits to represent the input information. Also encoder reduces the number of bits to store the coded information.

3.10 Majority Voter Circuit

Majority voter circuit is useful in fault-tolerant computing and in other applications. Output of a majority voter circuit becomes true when above 50% inputs to it are true. For example in a four-input majority voter circuit, if more than two inputs are logic 1 then output will become 1. This means majority of inputs are true. Here, a 4-input majority voter circuit is designed and its Boolean expression is

$$y = ab(c + d) + cd(a + b) \tag{3.18}$$

Corresponding logic diagram of this majority voter circuit is shown in Fig. 3.10.

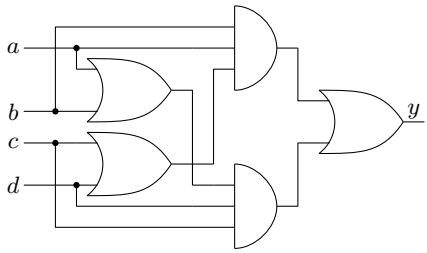


Fig. 3.10 A four-input majority voter circuit

3.11 Data Conversion Between Binary and Gray Code

In Gray code, only one bit changes in the code while going from one state to another. In Gray code no weight is assigned to a bit position and thus Gray code is an unweighted code. Gray code finds application where low switching rate is required. The comparison of Gray code compared to the decimal numbers and binary code is given in Table 3.7.

It may be noted that the Gray code can be regarded as reflected code. It can be clear by seeing the first 3-bit positions below and above the horizontal line. The conversion between binary and Gray code is very important to interface two kind of systems. The Boolean expression to convert a binary code to Gray code is

Table 3.7 Binary representation versus Gray representation

Decimal number	Binary representation	Gray representation
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

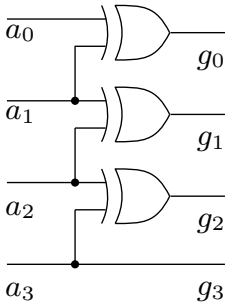


Fig. 3.11 Binary data to Gray code conversion for 4 bits

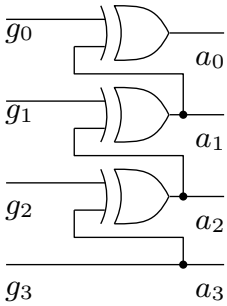


Fig. 3.12 Gray code to binary data conversion for 4 bits

$$g_i = \begin{cases} a_n, & \text{if } i = n \\ a_{i+1} \oplus a_i, & \text{if } i = 0, 1, 2, ..., (n-1) \end{cases} \tag{3.19}$$

The architecture for converting a 4-bit binary code to a 4-bit Gray code is shown Fig. 3.11. Similarly the Boolean expressions for converting the Gray code to equivalent binary code can be generated using K-map. The expressions are

$$a_i = \begin{cases} g_n, & \text{if } i = n \\ a_{i+1} \oplus g_i, & \text{if } i = 0, 1, 2, ..., (n-1) \end{cases} \tag{3.20}$$

The corresponding scheme for converting a 4-bit Gray code to binary code is given in Fig. 3.12.

3.12 Conversion Between Binary and BCD Code

The binary number system versus the Binary Coded Decimal (BCD) code is shown in Table 3.8. In BCD code, 10 digits are used to represent decimal numbers. These digits vary from 0 to 9. Equivalently binary representation of these numbers can be found. Thus up to digit 9 in decimal, both BCD and binary number system are the

Table 3.8 Binary representation and their BCD equivalents

Decimal number	Binary codes	BCD codes
0	0000	00000
1	0001	00001
2	0010	00010
3	0011	00011
4	0100	00100
5	0101	00101
6	0110	00110
7	0111	00111
8	1000	01000
9	1001	01001
10	1010	10000
11	1011	10001
12	1100	10010
13	1101	10011
14	1110	10100
15	1111	10101

same. Often BCD codes are used to display the results of a digital system on LCD display. In such case, conversion between binary and BCD codes is very important. These conversion techniques are described below

3.12.1 Binary to BCD Conversion

After observing Table 3.8, it can be said that after digit 9, 6 is added to the decimal number to get the digit in the BCD code. For example, 10 in the decimal system is represented as 16 in the BCD code. Here, we will discuss well known double-dabble [1] algorithm for binary to BCD conversion. This algorithm is also known as shift and Add 3. A binary number is left shifted and if the shifted part under weightage of One's, Ten's or Hundred's is equal to or greater then '101_2' then 3 is added. Then again apply a left shift and repeats the steps. This way the number is converted to the BCD code. An example of this conversion technique is shown in Table 3.9.

The architecture for binary to BCD conversion is shown in Fig. 3.13. The Add-3 block adds 3 to a 4-bit number whenever the number is equal to or greater than 5. The structure of this ADD-3 block is shown in Fig. 3.14. There are seven Add-3 blocks used in Fig. 3.13. A selection logic is there to select the number after addition with 3.

Table 3.9 Example of binary to BCD code conversion

Operations	Tens	Ones	Binary
Start			00111111
Shift		0	0111111
Shift		00	111111
Shift		0001	11111
Shift	0	0011	1111
Shift	00	0111	111
Add 3	00	1010	111
Shift	001	0101	11
Add 3	001	1000	11
Shift	0011	0001	1
Shift	0110	0011	

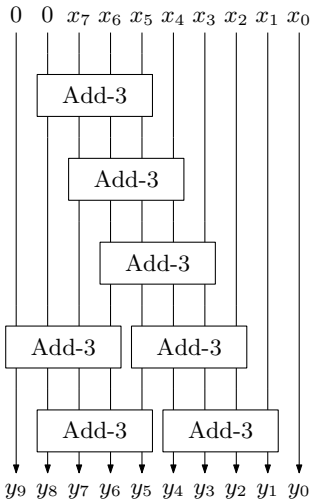


Fig. 3.13 Binary number system to BCD code conversion circuit

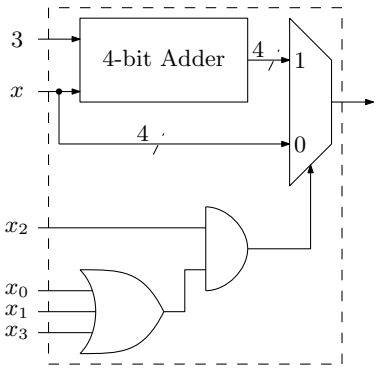


Fig. 3.14 The structure of the Add-3 circuit

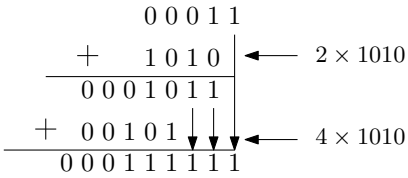


Fig. 3.15 An example of BCD to binary conversion

3.12.2 BCD to Binary Conversion

The conversion of BCD codes to Binary numbers is also important in digital circuits. A BCD number $x = 123$, can be expressed as $x = 123 = 1 \times 100 + 2 \times 10 + 3 \times 1$. The conversion from a BCD number to binary number follows this philosophy. For example, in the number BCD number $x = 0001\_1001$ first four bits from the LSB have the weightage of one and the next four bits have the weightage of Ten (1010). Thus this number can be converted to binary as $y = 1001 + (0001 \times 1010) = 0001\_0011$. Another example of BCD to binary conversion is shown in Fig. 3.15. Here, BCD number 63 (0110_0011) is converted to binary (00111111).

The BCD to Binary conversion circuit is shown in Fig. 3.16. The LSB is equal for Binary and BCD codes. Here, two 4-bit adders are used. In the first adder, two bits under the weightage of TEN are dissolved and in the second adder next two bits

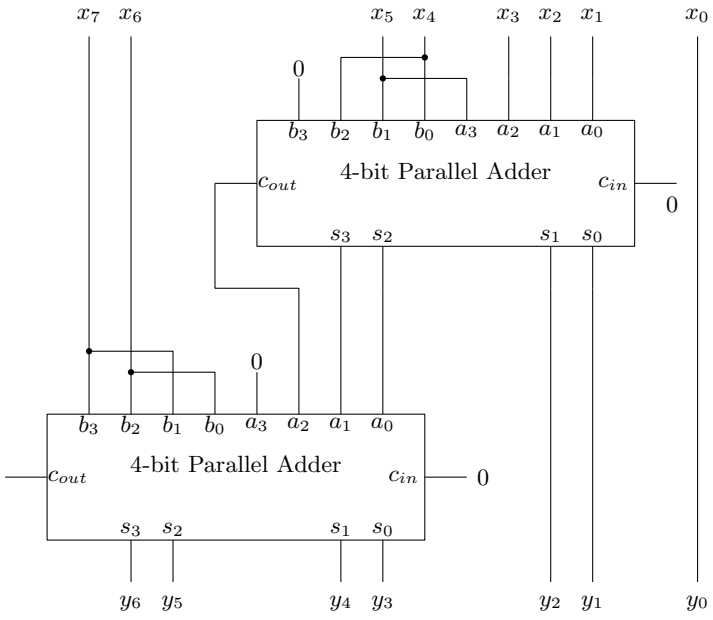


Fig. 3.16 BCD code to binary conversion circuit

Table 3.10 Condition for adding multiple of TENs

x_5	x_4	b_3	b_2	b_1	b_0
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	1	0
1	1	1	1	1	1

are considered. Through the b input of the adder, multiple of weightages are added. Condition for inputs to the b input is based on Table 3.10. Here for the first adder $b_3 = b_1 = x_5$ and $b_2 = b_0 = x_4$. Similarly, the conditions for the second 4-bit adder are derived.

3.13 Parity Generators/Checkers

Parity check and generation is an error detection technique in the digital transmission of bits. A parity bit is added to the data to make the number of ones either even or odd. In even parity bit scheme, the parity bit is '0' for even number of ones present in the data and the parity bit is '1' for odd number of ones in the data. If there are even number of ones in the data then the parity bit becomes '1' in odd parity bit scheme and similarly the parity bit becomes '0' for odd number of ones. Parity bit is generally added in the MSB.

Realization of both types of parity for 4-bit data is described in Figs. 3.17 and 3.18. If evn_parity is '1' then there are odd number of ones in the data. Three XOR gates are used to calculate evn_parity. odd_parity is just invert of evn_parity but a separate architecture is shown in Fig. 3.18. This is a balanced architecture and has some advantages over the structure shown in Fig. 3.17. This balanced structure will be discussed in detail in Chap. 15.

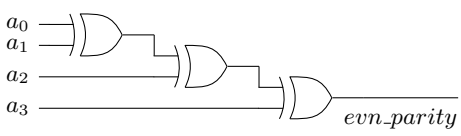


Fig. 3.17 Even parity generation for-bit data

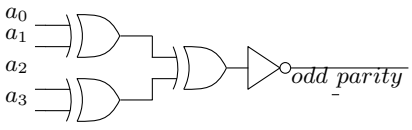


Fig. 3.18 Odd parity Generation for 4-bit data

3.14 Comparators

Comparison is a very important operation in implementation of any algorithm where sorting operation is carried out or where operands are compared to find the maximum or minimum. A comparator compares an operand (a) with another operand (b) to check whether the first operand is equal to, less than or greater than the second operand. In this section, design of a comparator block using hierarchical modelling style is discussed. First the comparator will be designed for smaller bits and then design of the higher order comparator will be discussed.

In a comparator, the operands are compared bit by bit. For example, for 16-bit comparator comparison starts from the MSB and then all the bits are compared. Thus first 1-bit comparator is discussed and then design of a 16-bit comparator is discussed. The truth table for a 1-bit comparator is shown in Table 3.11. Here, from the truth table of the 1-bit comparator it can be said that when the two bits are same, the equal (=) output is high. This equality check can be done simply by a XNOR gate. Similarly Boolean expressions for other output signals can be easily derived and these are shown below

$$eq = \overline{(a \oplus b)} \tag{3.21}$$

$$lt = \overline{a}.b \tag{3.22}$$

$$gt = a.\overline{b} \tag{3.23}$$

The architecture for 1-bit comparator is shown in Fig. 3.19. This is the optimized block diagram of the 1-bit comparator where the XNOR gate is replaced with the

Table 3.11 Truth table of a 1-bit comparator

a	b	$=$	$<$	$>$
0	0	1	0	0
0	1	0	1	0
1	0	0	0	1
1	1	1	0	0

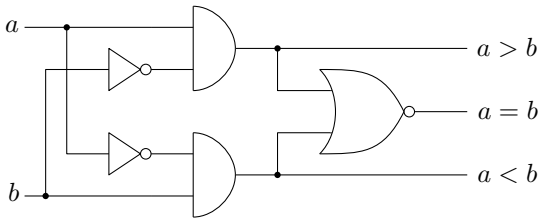


Fig. 3.19 Schematic of a 1-bit comparator

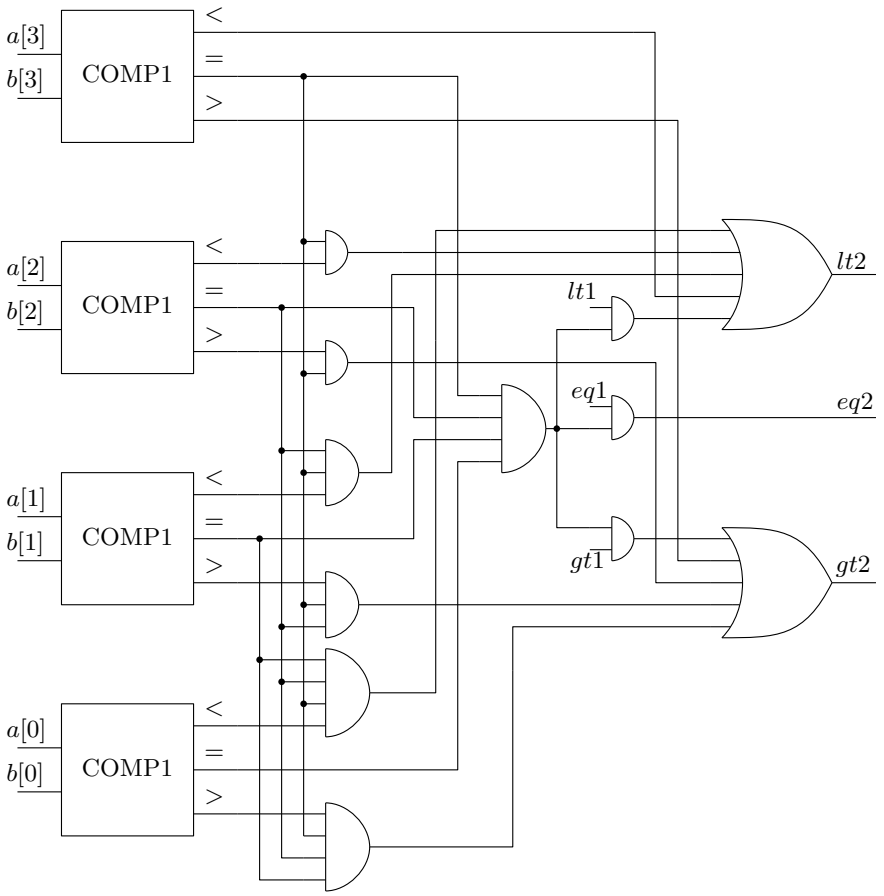


Fig. 3.20 Architecture of a 4-bit comparator using 1-bit comparator

NOR gate. Now we can proceed to design of higher order comparators using the 1-bit comparators. Lets discuss design of a 4-bit comparator. Four 1-bit comparators are required to design a 4-bit comparator. If all the bits are equal then only operands a and b are equal. Thus to check equality, equal (=) outputs from all the comparators are checked and ANDed. The output signals *less than* (<) or *greater than* (>) are checked by comparing the operands a and b from the MSB side. If $a_3 > b_3$ then it can be said that $a > b$. If $a_3 = b_3$ then the next bit is checked. If $a_3 = b_3$ and $a_2 > b_2$ then also $a > b$. This way the next bit is checked until LSB is faced. Similarly, the less than operation is carried out.

The architecture of a 4-bit comparator is shown in Fig. 3.20. Here, four 1-bit comparators are used. The comparator has three outputs $lt2$, $eq2$ and $gt2$. Three more inputs are included to this block to support the hierarchical design. These

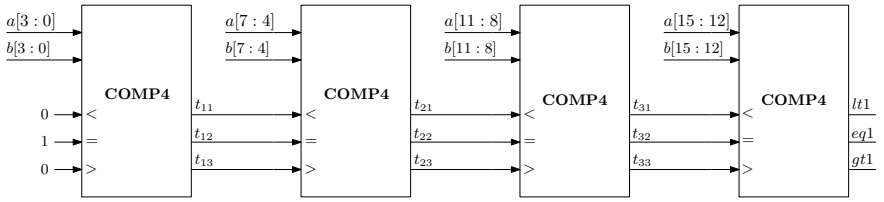


Fig. 3.21 Architecture of a 16-bit comparator using 4-bit comparator

inputs are $lt1$, $eq1$ and $gt1$. The 4-bit comparator block receives these inputs from another block.

An architecture of a 16-bit comparator using 4-bit comparators is shown in Fig. 3.21. Here, four 4-bit comparators are used. The 16-bit word length is partitioned in four nibbles. Each nibble has four bits. The nibble from the MSB side is compared first and then the next nibble is compared. The initial inputs for the 4-bit comparator that receives the lower 4 bits are set as $lt1 = 0$, $eq1 = 1$ and $gt1 = 0$. The input $eq1$ is facing an AND gate and thus it must be set to logic 1.

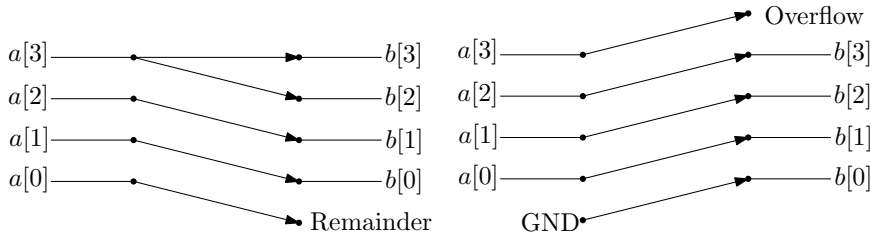
3.15 Constant Multipliers

In many signal processing or image processing applications constant parameters are multiplied. These constants are fixed for a particular design. Then the complete multipliers are not required in those applications to multiply the constants. Basically, the complete multipliers are avoided to multiply constants as they consume more logic gates. An alternative is to use constant multipliers or scale blocks to realize the multiplication with the fixed constants.

In evaluation of the equation $b = ca$, where b is the output, a is the input operand and c is the constant parameter, constant multipliers are very useful. The constant multipliers are constant specific means that the hardware specification varies from constant to constant. Let's consider an example where the input data a is divided by the constant 3. In other words, input a is multiplied by $1/3 = 0.3333$. This multiplication process can be written as

$$b = a/c = a(2^{-2} + 2^{-4} + 2^{-6} + 2^{-8}) \tag{3.24}$$

Here, data width of 18-bit is considered and 9-bit is taken to represent the fractional part. Here, when a operand a is multiplied by 2^{-i}, then the operand is right shifted by ith bit. This shifting operation is realized by wired shifting technique which does not consume any logic element. This shifting operation is realized by directly connecting the wires thus very fast. A schematic for hardware wire shifting for 1-bit right and 1-bit left is shown in Fig. 3.22.



(a) Schematic for 1-bit hardware right shift for 4-bit data width.

(b) Schematic for 1-bit hardware left shift for 4-bit data width.

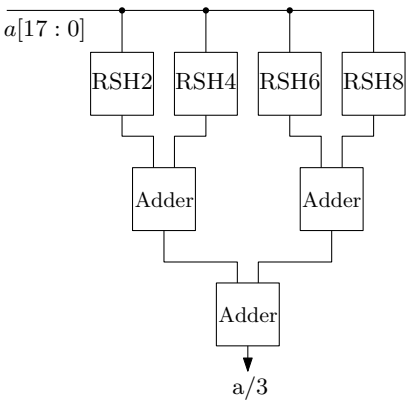
Fig. 3.22 Schematic for wired shifting by 1-bit right and left for 4-bit data width

This wired shifting methodology supports both signed and unsigned numbers represented in two's complement representation. For example, for $0.5/2$ the result should be 0.25 and for $-0.5/2$ the result should be -0.25. This is why the MSB bit is kept same for input operand and the output. The wired shift block for 1-bit right shift is called here as RSH1 and for 1-bit left shift block is called here as LSH1.

The constant multiplication block that multiplies the input operand a by 0.3333 is shown in Fig. 3.23. Here, four right shift blocks are designed which are RSH2, RSH4, RSH6 and RSH8. The input operand is shifted and added to obtain the final result according to Eq. (3.24). Here, only three adders are used thus it can be said that this constant multiplier block is hardware efficient than a complete multiplier.

The shift blocks shown in Fig. 3.23 are for fixed number of bits. But in many applications variable shifting operations are required. These variable shift blocks shift an input operand by a variable count. Before discussing the variable shift blocks, first consider a block that shifts an input operand by 1-bit in the right side depending on a control signal. If the control bit is high then the input operand is right shifted by 1-bit otherwise the block passes the same input to the output. This type of block is called Controlled RSH (CRSH) block for the right shift and called as Controlled

Fig. 3.23 Scheme for constant multiplier



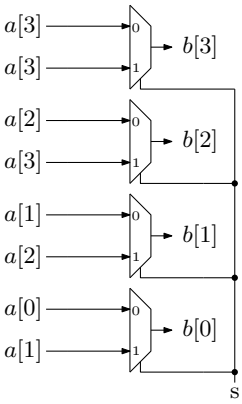


Fig. 3.24 Scheme for controlled 1-bit right shift

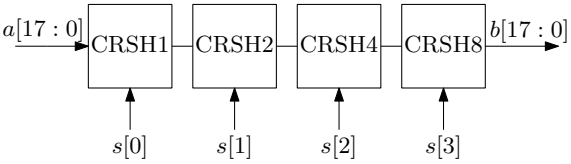


Fig. 3.25 Scheme for variable right shift

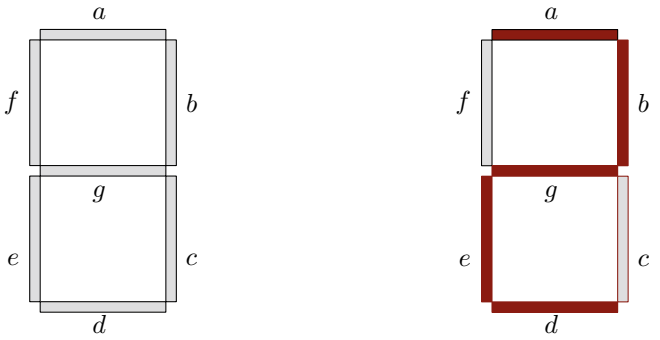
LSH (CLSH) for left shift. Here, CRSH1 block is shown in Fig. 3.24 which shifts the input operand by 1-bit if s is equal to 1. The CRSH block has a delay of a MUX in the path.

Now, the variable shift blocks can be designed using the controlled shift blocks. The variable shift block for right shift is called as Variable RSH (VRSH) and the variable shift block for left shift is called as Variable LSH (VLSH). A diagram for VRSH block is shown in Fig. 3.25. Here, this block is configured using CRSH1, CRSH2, CRSH4 and CRSH8 blocks. This block is capable of shifting an operand by any number from 0 to 15. Shifting by $s = 0$ means all the controlled shift blocks are disabled and they pass the input data as it is. Shifting by $s = 15$ is achieved by enabling all the blocks. The VRSH or VLSH block has a delay of a maximum of 4 MUXes connected in series. Thus has a speed limitation.

3.16 Frequently Asked Questions

Q1. **Write a Verilog code in behavioural style for an 18-bit comparator?**
A1. Realization of a comparator using behavioural model is very straightforward forward as shown below

```
module comp18(A1,B1,LT1,GT1,EQ1);
    input [17:0] A1,B1;
    output reg LT1,GT1,EQ1;
     always @ (A1,B1)
```



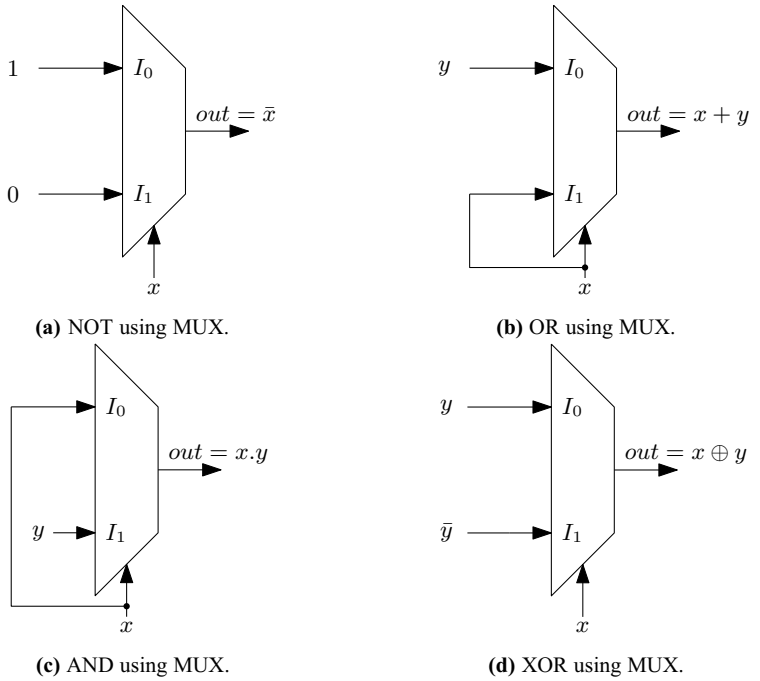
(a) Seven Segments of a digit. (b) Number 2 is represented.

Fig. 3.26 Seven segment display and it representation

```
        begin
        if (A1>B1)
        begin
        LT1 <= 0; GT1 <= 1; EQ1 <= 0;
        end
        else if (A1<B1)
        begin
        LT1 <= 1; GT1 <= 0; EQ1 <= 0;
        end
        else
        begin
        LT1 <= 0; GT1 <= 0; EQ1 <= 1;
        end
        end
endmodule
```

Q2. **Write Verilog Code to display BCD numbers on Seven Segment display?**
A2. In order to display the BCD numbers, Seven Segment Display (SSD) is used. SSD is an inbuilt feature of many FPGA kits. Seven segments together display any number in the SSD. These seven segments can be used to display $2^7 = 128$ number of combinations but few combinations are used. Representation of a BCD number using seven segments is shown in Fig. 3.26.

```
module segment7(BCD,SEG);
      input [3:0] BCD;
      output reg [6:0] SEG;
     always @(BCD)
     begin
          case (BCD)
               0 : SEG = 7'b1111110;
               1 : SEG = 7'b0110000;
               2 : SEG = 7'b1101101;
               3 : SEG = 7'b1111001;
               4 : SEG = 7'b0110011;
```



(a) NOT using MUX. **(b)** OR using MUX.

(c) AND using MUX. **(d)** XOR using MUX.

Fig. 3.27 Realization of different gates using 2:1 MUX

```
            5 : SEG = 7'b1011011;
            6 : SEG = 7'b1011111;
            7 : SEG = 7'b1110000;
            8 : SEG = 7'b1111111;
            9 : SEG = 7'b1111011;
            default : SEG = 7'b0000000;
        endcase
    end
endmodule
```

Q3. **Realize different logic gates using 2:1 MUX?**

A3. Different logic gates can be realized using 2:1 MUX and it is very helpful in FPGA implementation. The implementation of different logic functions using 2:1 MUX is shown in Fig. 3.27. For a 2:1 MUX, output is equal to I_0 when the select signal is 0 and output is equal to I_1 when the select signal is 1.

3.17 Conclusion

Various combinational circuits are discussed in this chapter. Though this is a brief discussion, we have covered all the major combinational circuits required to implement a complex digital system. Among the arithmetic circuits, basic full adders, subtractors and control adder/subtractor blocks are discussed. The multiplication, division or other complex circuits are discussed in separate chapters.

Other combinational circuits like Multiplexers, De-Multiplexers, Encoder, Decoder, data coders are also discussed here. Design of a 16-bit comparator using smaller comparators is shown in this chapter. The data shifters are very important in designing digital systems. Here, wired shifting methods for signed data is demonstrated. The design of constant multipliers is also shown in this chapter using these wired shift blocks. These combinational blocks can be modelled using Verilog easily.

Chapter 4
Basic Sequential Circuits

4.1 Introduction

In the last chapter, we have discussed different combinational circuits and their implementation. But no digital system purely contains the combinational circuits. In case of sequential circuits, the output is not only a function of the present input but also depends on output past history. Sequential circuits are sometimes called as time dependent circuits as outputs are updated according to a time event.

In this chapter, a brief theory on the sequential circuits is discussed. The objective of this chapter is to discuss the major sequential blocks which are needed to implement a complex digital system. Some of the basic concepts are avoided here as these are already discussed in many textbooks or in many online tutorials. Operation of major sequential blocks such as flip-flops, shift register, counters and frequency dividers is explained below in the following sections.

4.2 Different Flip-Flops

The basic element of a sequential circuit is a flip-flop. Flip-flops are sometimes referred as memory elements as they can store 1-bit of information. Flip-flops are also sometimes regarded as Bistable multivibrators as flip-flop has two stable states. There are mainly four types of flip-flops studied in literature which are

1. SR Flip-Flop
2. JK Flip-Flop
3. D Flip-Flop
4. T Flip-Flop

All these clocked flip-flops change their state depending on the triggering by the clock signal. A flip-flop can be either level triggered or edge triggered. The edge triggering can be of two types which are positive edge triggering or negative edge triggering. The transition of clock signal from the active high state to low state is

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_4

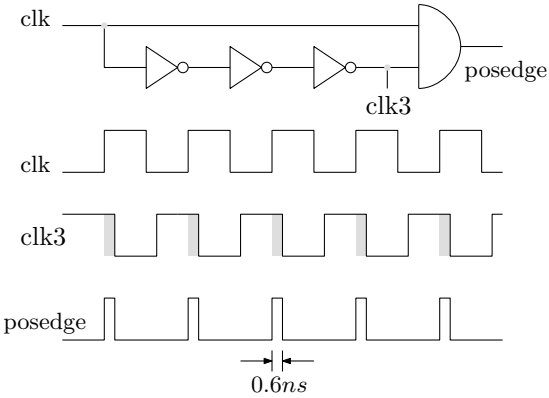


Fig. 4.1 Concept of positive edge triggering

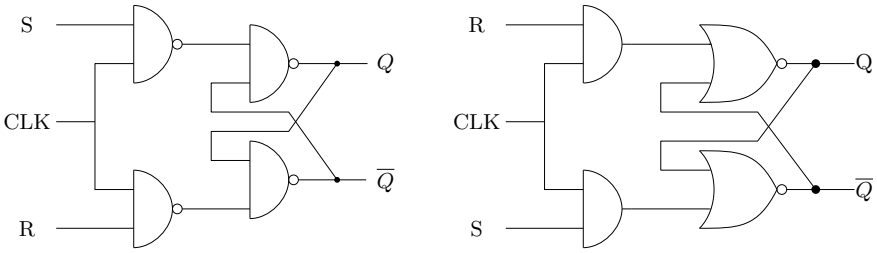
the positive edge. Similarly in the negative edge, clock signal goes to low state from the high state. The edge triggering can be generated either by an analog circuit (R-C network) or by a digital circuit. The conception of positive edge triggering is shown in Fig. 4.1.

4.2.1 SR Flip-Flop

SR flip-flop is the most basic flip-flop. The truth table of the SR flip-flop is shown in Table 4.1 where Q is the present state and Q^* represents the next state. SR flip-flop has two inputs, one is set (S) and another is reset (R). The set input sets the output (Q) and the reset input resets the output. When the S and R input is '10', Q is set. The output is reset when input is '01'. The SR flip-flop retains its previous state when

Table 4.1 SR flip-flop truth table

Inputs		Output	
S	R	Q	Q^*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	X
1	1	1	X



(a) NAND Based clocked SR flip-flop. **(b)** NOR Based clocked RS flip-flop.

Fig. 4.2 Schematic for SR flip-flop

input is '00' due to the feedback connection of the SR latch. The output goes to an undefined state when the input is '11' because of NOR or NAND gate functionality.

The implementation of SR flip-flop using NAND and NOR SR latch is shown in Fig. 4.2. The realization of the SR flip-flop using Verilog is shown below. Here, SR flip-flop is realized using behavioural coding using the truth table of SR flip-flop.

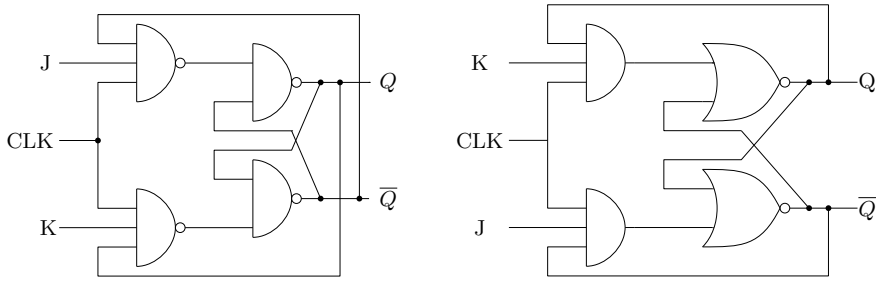
```
module srff(S,R,clk,reset,q,qb);
output reg q,qb ;
input S,R,clk,reset;
initial begin q=1'b0; qb =1'b1; end

always @ (posedge clk)
 if (reset) begin
  q <= 0;   qb <= 1;
 end
 else begin
  if (S!=R) begin
   q <= S;   qb <= R;
  end
  else if (S==1 && R==1) begin
   q <= 1'bZ;  qb <= 1'bZ;
  end
 end
endmodule
```

4.2.2 JK Flip-Flop

The basic SR flip-flop has an undefined state when $S = 1$ and $R = 1$. Thus application of SR flip-flop is limited. JK flip-flop is modified version of SR flip-flop and works exactly same way as the SR flip-flop does but eliminates the limitation of SR flip-flop. JK flip-flop has two inputs called J and K instead of S and R in case of SR flip-flop. The input combination $J = 1$ and $K = 1$ is permitted in case of JK flip-flop. The truth table of JK flip-flop is shown in Table 4.2.

Table 4.2 JK flip-flop truth table

Inputs		Output	
J	K	Q	Q^*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0



(a) JK flip-flop by NAND SR latch. **(b)** JK flip-flop using NOR SR latch.

Fig. 4.3 Schematic for JK flip-flop

The output of the JK flip-flop toggles when $J = K = 1$. That means, if the present state of Q is '0' then Q is switched to '1'. Also, if the present state of Q is '1' then Q is switched to '0'. The schematic of the JK flip-flop using the NAND-based SR latch and NOR-based SR latch is shown in Fig. 4.3. The realization of JK flip-flop using Verilog HDL is given below.

```
module jk(q,qb,j,k,reset,clk);
output reg q,qb;
input j,k,clk,reset;
initial begin q = 1'b0; qb = 1'b1; end

always @ (posedge clk)
if(reset)
 begin
  q = 1'b0;  qb = 1'b1;
 end
else
 case({j,k})
```

```
  {1'b0,1'b0}: begin q=q; qb=qb; end
  {1'b0,1'b1}: begin q=1'b0; qb=1'b1; end
  {1'b1,1'b0}: begin q=1'b1; qb=1'b0; end
  {1'b1,1'b1}: begin q=~q; qb=~qb; end
 endcase
endmodule
```

4.2.3 D Flip-Flop

The D flip-flop is the most used clocked flip-flop in designing the digital systems. Compared to the SR flip-flop, D flip-flop ensures that both S and R input do not get same value. Thus all the states of D flip-flop is valid. The D flip-flop is used for delay insertion as it is capable of delaying a signal by one clock period. The truth table of the D flip-flop is shown in Table 4.3.

Here, the output of the D flip-flop does not change its state when the value of the clock is low and output follows the input when the clock signal is high. Thus clock signal works as enable signal. The timing diagram for the D flip-flop is shown in Fig. 4.4. Here, clock signal is positive edge triggered. The duration of one sample of data is equal to clock period. The output signal Q is a delayed version of the input data signal D.

The realization D flip-flop using SR flip-flop or JK flip-flop is shown in Fig. 4.5. In every sequential circuits, D flip-flop is used as the basic building block. Thus it is important to know the Verilog modelling of D flip-flop. Below is a Verilog code of a D flip-flop in structural modelling style. Here, the positive edge triggering signal is generated within the code. Additional signals like control enable (ce) and $reset$

Table 4.3 Truth Table for D flip-flop

Clock	D	Q	$\bar{Q}$
Low	X	Q	$\bar{Q}$
High	0	0	1
High	1	1	0

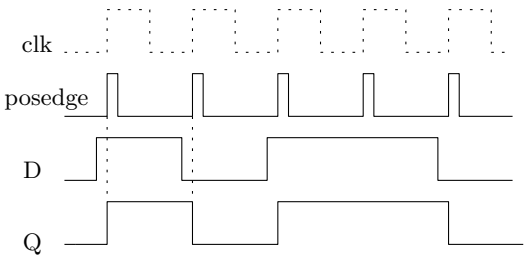
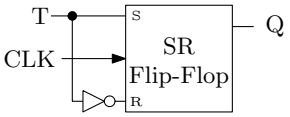
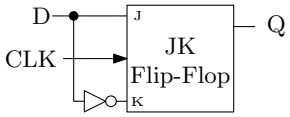


Fig. 4.4 Timing diagram for the D flip-flop



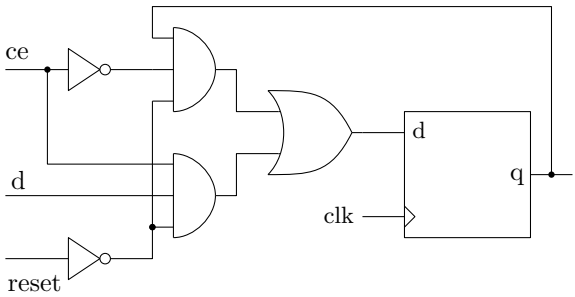
(a) D flip-flop using SR flip-flop.



(b) D flip-flop using JK flip-flop.

Fig. 4.5 Realization of D flip-flop using SR and JK flip-flop

Fig. 4.6 A typical modelling style of D flip-flop



are also used. An extra enable signal *ce* is used to control the writing of data signals without interfering the clock signal. The *reset* signal clears the output.

```
module dff_struct(q,qb,d,reset,ce,clk);
output q,qb;
input d,reset,ce,clk;
wire t1,t2,d1,d2,d3;
wire clk1,clk2,clk3,posedge;
assign #0.2 clk1 = ~clk;
assign #0.2 clk2 = ~clk1;
assign #0.2 clk3 = ~clk2;
assign d1 = ~ce & q & ~reset;
assign d2 = ce & d & ~reset;
assign d3 = d1 |d2 ;
assign posedge = clk3 & clk;
assign t1 = ~(d3 & posedge) ;
assign t2 = ~(~d3 & posedge) ;
assign q = ~(t1 & qb);
assign qb = ~(t2 & q );
endmodule
```

The D flip-flop according to the structural Verilog code is shown in Fig. 4.6. Here, if the *ce* is active then only input is passed to the output and if *ce* is low then output retains its previous value. This configuration has many use in designing complex digital systems. In the following Verilog code, D flip-flop is also modelled using behavioural coding Style. Here, we do not need to generate the positive or negative edge. This is the mostly used model for D flip-flop.

```
module dff(q,reset,clk,d);
output reg q;
input reset,d,clk;
initial begin q=1'b0; end
always @ (posedge clk)
if (reset)
q <= 1'b0;
else
q<=d;
endmodule
```

4.2.4 T Flip-Flop

T flip-flop is also known as 'Toggle flip-flop'. This is because its output toggles between a state and its inverted state depending on a single input. The truth table for this type of flip-flop is shown in Table 4.4. The behaviour shown in Table 4.4 can be modelled in Verilog as

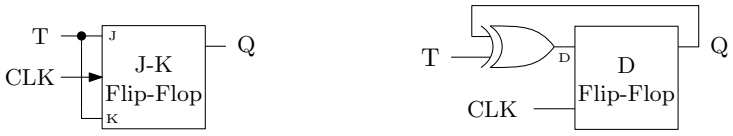
```
module tff(q,reset,clk,t);
output reg q;
input t,reset,clk;
initial begin q=1'b0; end
always @ (posedge clk)
if (reset)
q <= 1'b0;
else if (t)
q= ~q;
else
q = q;
endmodule
```

T flip-flop can be designed using either D or JK flip-flop. T flip-flop can be realized using D flip-flop as

$$D = T \oplus Q \tag{4.1}$$

Table 4.4 Truth table for T flip-flop

Input	Output	
	Present state	Next state
T	Q	Q^*
0	0	0
0	1	1
1	0	1
1	1	0



(a) T flip-flop using JK flip-flop. **(b)** T flip-flop using D flip-flop.

Fig. 4.7 Schematic of T flip-flop using D flip-flop and JK flip-flop

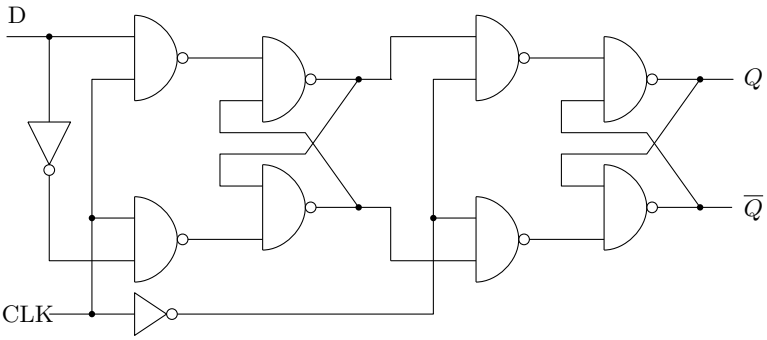


Fig. 4.8 Schematic of Master-Slave D flip-flop

Similarly, JK flip-flop can also be used to realize T flip-flop by setting $J = K = T$. The realization T flip-flop using JK flip-flop and D flip-flop is shown in Fig. 4.7.

4.2.5 Master-Slave D Flip-Flop

A master-slave flip-flop is constructed using two separate but same type of flip-flops. One flip-flop serves as master while the other serves as the slave. It is used to convert level triggered to edge triggered flip-flop. Thus the master-slave D flip-flop has the advantage that it always works on clock edge. When the clock is high, the first latch store the data and state of second latch does not change. When the clock is low, second latch gets data that is stored by the first latch, and the first latch do not change its state. This is for positive edge triggering and similar operation is followed for negative edge triggering. A master-slave D flip-flop is shown in Fig. 4.8.

4.3 Shift Registers

Shift registers load the data present on its inputs and then moves or shifts it to its output once in every clock cycle. A shift register basically consists of several single bit D-Type flip-flops, one for each data bit. Shift registers can be used to shift a input

data, serial to parallel conversion, parallel to serial conversion or for data storage. There are mainly four types of shift registers which are

1. Serial Input Serial Output (SISO)
2. Serial Input Parallel Output (SIPO)
3. Parallel Input Serial Output (PISO)
4. Parallel Input Parallel Output (PIPO)

All these shift registers discussed here are synchronous, which means the same clock input is connected to all the flip-flops.

4.3.1 Serial In Serial Out

SISO is a type of shift register which receives a serial stream of bits or bytes and shifts the stream serially. In other words, the data stream is delayed by the SISO. If there are n flip-flops then the data stream will be delayed by n clock cycles. The SISO for $n = 4$ is shown in Fig. 4.9. Here four D flip-flops are connected in series. The output signal is the same as the output of the 4th flip-flop. The clockwise shifting of the single bit data stream for $n = 4$ is shown in Fig. 4.10. Each flip-flip introduces a delay of one clock cycle. Here all the flip-flops are positive edge triggered.

4.3.2 Serial In Parallel Out

SIPO is a type of shift register which is used to convert a serial data stream into a parallel data stream. The main use of this type of shift register is in the interfacing circuits (ADC, DAC interfacing) or in the circuits where we need to convert serial data to equivalent parallel one. The SIPO is shown in Fig. 4.11 for $n = 4$. Here four flip-flops are serially connected but the output is taken in parallel. In converting a serial data to parallel data stream, the care should be taken in selecting the clock frequency for input sampling and output sampling. If $clk1$ is used to sample the outputs then the relation $clk1 = n.clk$ should be maintained. The time instant when the parallel outputs $q_{3:0}$ is taken is shown in Fig. 4.10 by dotted line.

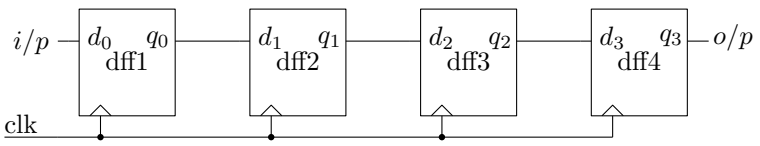


Fig. 4.9 A 4-bit serial in serial out shift register

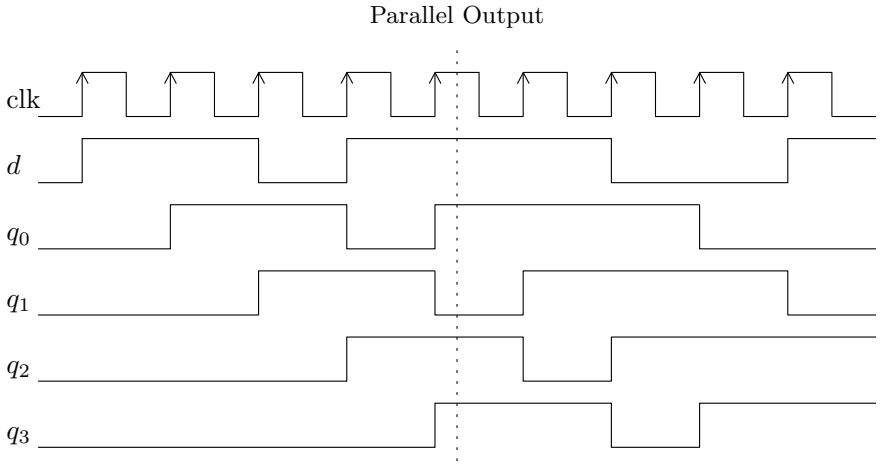


Fig. 4.10 A 4-bit serial in serial out shift register

4.3.3 Parallel In Serial Out

PISO is a type of shift register which is opposite of SIPO type of shift register. PISO takes a parallel data stream as input and outputs a serial stream of data. The PISO is used to convert a parallel stream in a serial one. In integrated circuits, due to the limitation of input and output ports parallel data stream is converted to the serial data stream.

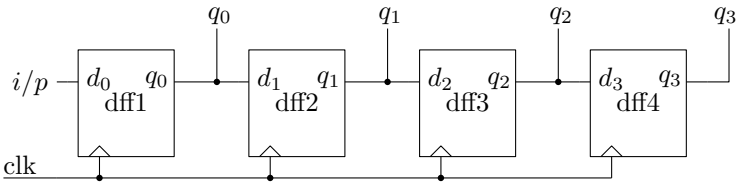


Fig. 4.11 A 4-bit serial in parallel out shift register

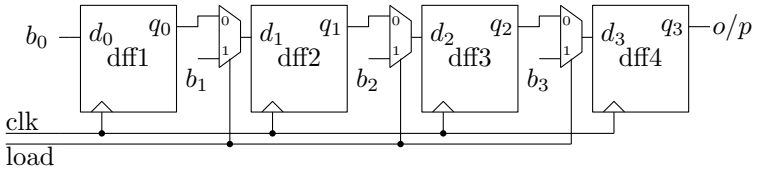


Fig. 4.12 A 4-bit parallel in serial out shift register

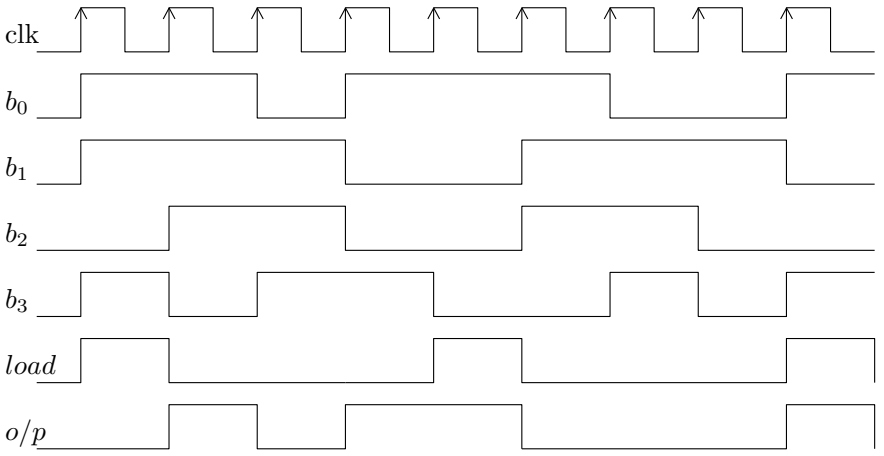


Fig. 4.13 A 4-bit Parallel in serial out shift register

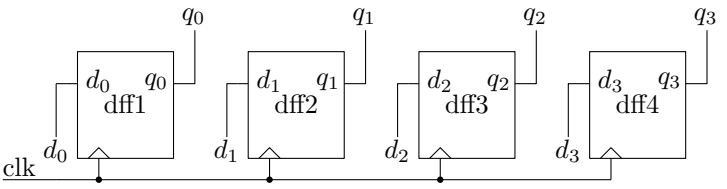


Fig. 4.14 A 4-bit parallel in parallel out shift register

The schematic of PISO for $n = 4$ is shown in Fig. 4.12. Here, $b_{3:0}$ is the parallel input data and q_3 is the serial output. The parallel data is loaded to the flip-flops by a control signal $load$. The PISO starts working as a SISO after the data is loaded. The timing diagram for the PISO is shown in Fig. 4.13. After four clock cycle of delay, the $load$ signal is again high. The clock frequency at which the $load$ signal should be asserted is $clk1 = clk/n$ for n number of flip-flops.

4.3.4 Parallel In Parallel Out

PIPO is type of shift register which is mostly used in digital systems than the other type of registers. In PIPO a parallel data stream is input and a parallel data stream is output. PIPO is most popularly known as pipeline register or simply as register. PIPO is used for storing a data for one clock cycle or delaying a data by one clock cycle. The schematic of the PIPO is shown in Fig. 4.14 for $n = 4$. The D flip-flops are not connected to each other. Each flip-flop independently receives an input and produces an output synchronously.

Table 4.5 Truth table for sequence generator using three flip-flops

q_2	q_1	q_0	d_2
1	0	1	0
0	1	0	1
1	0	1	1
1	1	0	0
0	1	1	1

Table 4.6 Truth table for sequence generator using four flip-flops

q_3	q_2	q_1	q_0	d_3
1	0	1	1	0
0	1	0	1	1
1	0	1	0	1
1	1	0	1	0
0	1	1	0	1

4.4 Sequence Generator

Sequence generator is a sequential block which generates a particular sequence and then repeats. First step in designing a sequence generator is to find number of flip-flops. Lets say the length of the sequence is L. Then number of flip-flops can be find from the relation $L \leq (2^n - 1)$. Here, n indicates the minimum number of flip-flops. Lets consider the sequence is '10110'. Here, $L = 5$ thus minimum three flip-flops are enough. The truth table for this sequence using three flip-flops is shown in Table 4.5.

Here output is taken from the q_2 port. The expression of input d_2 will be some combination of q_0, q_1 and q_2. Here, two states are same which is equal to '101'. Thus this is not possible to generate the sequence with three flip-flops. Lets rewrite the truth table with four flip-flops which is shown in Table 4.6.

Here, no states are same and four flip-flops are to be used. The Boolean expression of the d_3 can be written using K-map. The K-map is shown in Fig. 4.15. The optimized Boolean expression for d_3 is

$$d_3 = \bar{q}_3 + \bar{q}_0 \tag{4.2}$$

The circuit diagram for the sequence generator using four flip-flops is shown in Fig. 4.16. In designing a sequence generator, first step is to find the number of flip-flops. The second important thing is to make the correct truth table. In the truth table, from the left side first column is the sequence and the second column is the circular shifted version of the sequence. Third step is the K-map optimization problem.

q_3q_2 \ q_1q_0	00	01	11	10
00	×	×	×	×
01	×	1	×	1
11	×	0	×	×
10	×	×	0	1

Fig. 4.15 K-map optimization for d_3

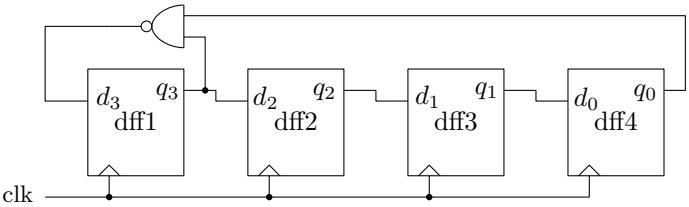


Fig. 4.16 Architecture for the '10110' sequence generator

4.5 Pseudo Noise Sequence Generator

Pseudo Noise (PN) sequence or sometimes called as Pseudo Random (PR) sequence has many application in signal processing or in the field of communication. The voltage level of the PN sequence can vary either from 0 to 1 or from −1 to 1. PN sequence is not totally random but has randomness in a certain period. Most common way to generate a PN sequence is using Linear Feedback Shift Register (LFSR). Some of the techniques to generate PN sequence are mentioned below.

Fibonacci LFSR is a famous technique to generate PN sequences. An n bit LFSR generates a PN sequence of period $(2^n - 1)$ using n flip-flops. LFSR is a register whose input is a linear function of its previous state. This linear function is most commonly XOR. The PN sequence to be generated is controlled by a feedback polynomial. The value of co-efficients of the polynomial can be either 1 or 0. This polynomial says the XOR function will take input from which flip-flops. An example of such polynomial is shown below

$$x^4 + x^3 + 1 = x^4 + x^3 + x^0 \tag{4.3}$$

This polynomial says that output of 4th and 3rd flip-flops are taken to XOR network. The '1' does not represent any flip-flop, it represents that the input is taken to the first flip-flop. The structure of the LFSR according to the above equation is shown in Fig. 4.17. Here, total $n = 4$ number of flip-flops are used and the period of the PN sequence is 15. This means that after 15 samples the PN sequence will repeat. Different PN sequences can be generated by using different values of n. Generation

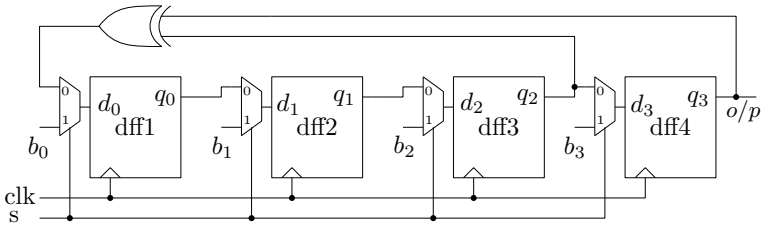


Fig. 4.17 PN sequence generation using Fibonacci LFSR

Table 4.7 Different feedback polynomials and their period

Bits	Feedback polynomial	Period $(2^n - 1)$
2	$x^2 + x + 1$	3
3	$x^3 + x^2 + 1$	7
4	$x^4 + x^3 + 1$	15
5	$x^5 + x^4 + 1$	31
6	$x^6 + x^5 + 1$	63
7	$x^7 + x^6 + 1$	127

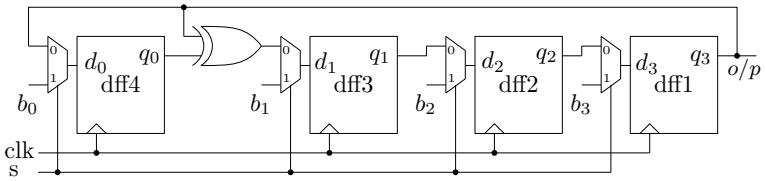


Fig. 4.18 PN sequence generation using Galois LFSR

of PN sequences using different feedback polynomials and their period is shown in Table 4.7.

Another way of generating PN sequence is to use Galois LFSR. The Galois LFSR follows the same feedback polynomial but in a reverse order. The Galois LFSR technique is shown in Fig. 4.18. Here the XOR gate is placed between the flip-flops and Galois LFSR uses always two-input XOR gate. Thus Galois LFSR is faster than Fibonacci LFSR.

Another way of generating PN sequence is using Gold codes. Gold codes have many application in the field of communication. PN sequence generated using Gold has small cross-correlations within a set. Gold codes are function of two PN sequences and most common function is XOR. Generation of simple Gold code is shown in Fig. 4.19. Here, two n-bit PN sequence are XORed and the resultant Gold code has same period of $(2^n - 1)$.

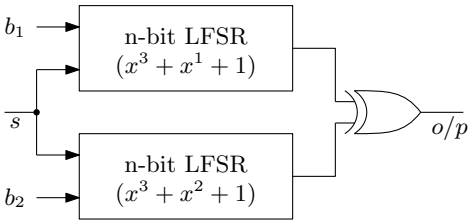


Fig. 4.19 PN sequence generation using Gold code

4.6 Synchronous Counter Design

Counters can be broadly classified as follows:

1. Asynchronous and Synchronous counters
2. Single and multi-mode counters
3. Modulus counters
4. Shift Register counters.

Design of asynchronous counters (Ripple counter) is easy as flip/flops are not under control of a single clock. Its speed of operation is limited and also there may be glitches at the outputs. These drawbacks are eliminated by giving single clock reference to every flip/flops. A counter may be either an up counter or a down counter. Multimode counters are also possible which have a control input to switch between up and down. Modulus counters are defined based on the number of states they are capable of counting. Shift registers are also can be arranged to form a counter. There are two types of shift register-based counter which are Ring counter and Johnson counter. A counter to count arbitrary sequence can also be designed using basic flip-flops.

In this section, design of synchronous counter is discussed. A synchronous counter may count in increasing order or in decreasing order. The truth table of a 4-bit synchronous up counter using D flip-flop is shown below in Table 4.8. The present state is $q_{3:0}$ and $q^*_{3:0}$ represents the next state. Corresponding inputs to the different flip-flops are also shown. Input to the D flip-flop is same as next state as D flip-flop just delays the input signal by one clock period.

The up counter is to be realized using the D flip-flops. The Boolean expressions for the input of the D flip-flops can be evaluated using the K-map shown in Fig. 4.20.

The Boolean expression for d_0 derived from the K-map is

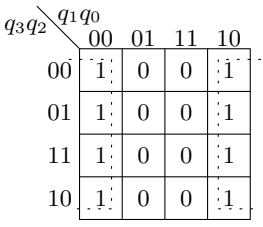
$$d_0 = \overline{q_0} \tag{4.4}$$

The Boolean expression for the d_1 is

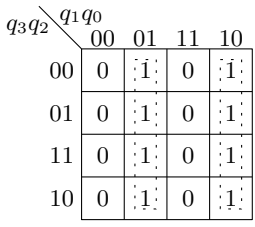
$$d_1 = q_1.\overline{q_0} + \overline{q_1}.q_0 = q_1 \oplus q_0 \tag{4.5}$$

Table 4.8 Truth table synchronous up counter

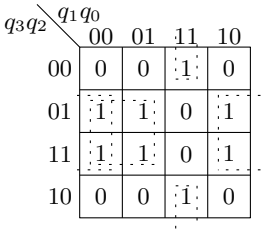
Decimal	$q_3q_2q_1q_0$	$q_3^*q_2^*q_1^*q_0^*$	d_3	d_2	d_1	d_0
0	0000	0001	0	0	0	1
1	0001	0010	0	0	1	0
2	0010	0011	0	0	1	1
3	0011	0100	0	1	0	0
4	0100	0101	0	1	0	1
5	0101	0110	0	1	1	0
6	0110	0111	0	1	1	1
7	0111	1000	1	0	0	0
8	1000	1001	1	0	0	1
9	1001	1010	1	0	1	0
10	1010	1011	1	0	1	1
11	1011	1100	1	1	0	0
12	1100	1101	1	1	0	1
13	1101	1110	1	1	1	0
14	1110	1111	1	1	1	1
15	1111	0000	0	0	0	0



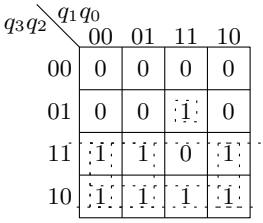
(a) K-map for d_0.



(b) K-map for d_1.



(c) K-map for d_2.



(d) K-map for d_3.

Fig. 4.20 Deriving the logical equation for up counter using K-map

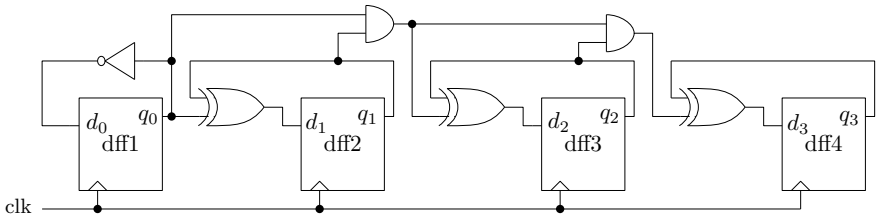


Fig. 4.21 A 4-bit synchronous up counter

The Boolean expression for d_2 can be derived as

$$d_2 = q_2.\overline{q_1} + q_2.\overline{q_0} + \overline{q_2}.q_1.q_0 = q_2 \oplus (q_1.q_0) \tag{4.6}$$

The Boolean expression for d_3 can be derived as

$$d_3 = q_3.\overline{q_2} + q_3.\overline{q_1} + q_3.\overline{q_0} + \overline{q_3}.q_2.q_1.q_0 = q_3 \oplus (q_2.q_1.q_0) \tag{4.7}$$

The schematic for the 4-bit synchronous up counter is shown in Fig. 4.21. Here, four D flip-flops are used and they are connected to a common clock. The counter counts from 0 to 15 and again reset to the 0.

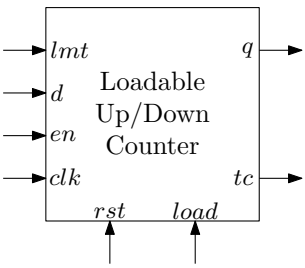
4.7 Loadable Counter

Loadable counter is a very popular counter used in many applications. The loadable counter can be treated as a general counter which can be used as BCD counter or Modulus counter. The loadable counter can be used to generate any arbitrary or semi-arbitrary sequence. The loadable counter can be of two types

1. Loadable Up Counter
2. Loadable Down Counter

The basic block diagram of the loadable up/down counter is shown in Fig. 4.22.

Fig. 4.22 A 4-bit synchronous up counter



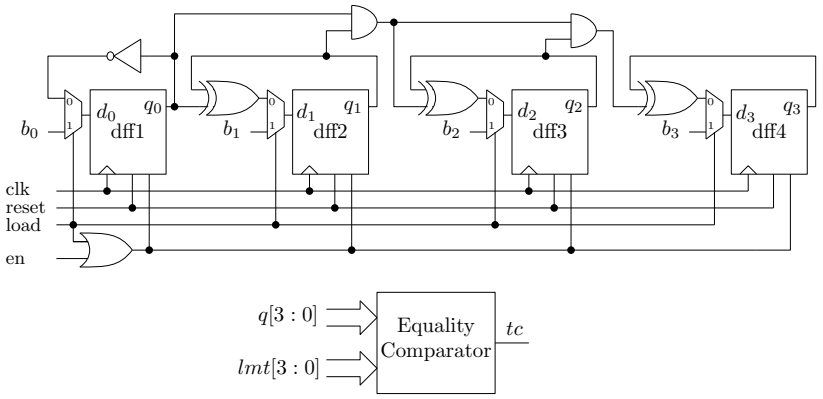


Fig. 4.23 A 4-bit loadable up counter

4.7.1 Loadable Up Counter

In the previous section, design of synchronous up counter is discussed. The loadable up counter is simply a synchronous up counter with load facility. A 4-bit loadable up counter is shown in Fig. 4.23. 2:1 MUXes are placed before each D flip-flop to load parallel data to the flip-flops. A control signal *load* selects the data $b_{3:0}$ when its value is logic 1. The loadable counter can start the sequence from the load value (b). The loadable up counter starts counting when the control signal enable (en) is high and also it has control sequence $reset$ which can reset the counter. The loadable up counter has two outputs, viz., $q_{3:0}$ and terminal count (tc). The terminal count signal is generated when the count is equal to another limit value (lmt). The tc signal can be used for many purposes. The tc signal can be used to stop the counter, load the counter or to start another counter. This control signal can be generated using a equality comparator which is a combination of XNOR gates and AND gate.

An example is shown in Fig. 4.24 to demonstrate the use of loadable up counter to generate a semi-arbitrary sequence. In this example, the value of data is taken as $b = 5$ and the value of limit is $lmt = 7$. As the enable signal goes high, the loadable up counter starts counting. The value of 5 is loaded to the counter when the *load* signal is high. As the *en* signal is high, the counter starts counting from the 5. The terminal count signal (tc) is generated when the count reaches the value 7.

4.7.2 Loadable Down Counter

In the previous section, design of loadable up counter is discussed. The loadable down counter is simply a synchronous down counter with load facility. A 4-bit loadable down counter is shown in Fig. 4.25. The operation of a loadable down counter is

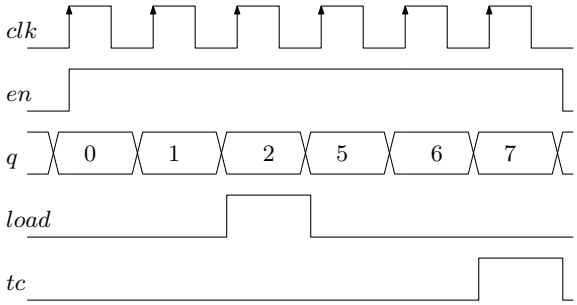


Fig. 4.24 Timing diagram for the 4-bit loadable up counter

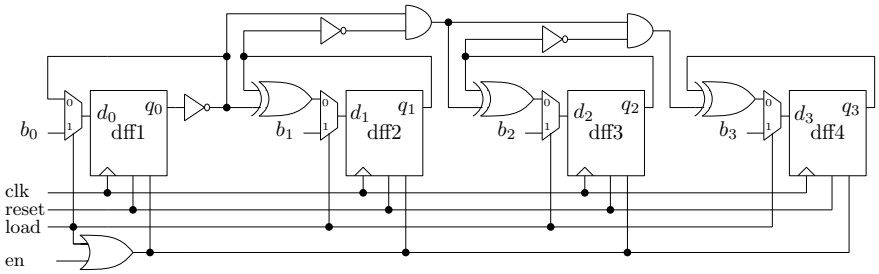


Fig. 4.25 A 4-bit loadable down counter

similar to that of a loadable up counter. The Boolean expressions for the loadable down counter are shown below

$$d_0 = \overline{q_0} \tag{4.8}$$

$$d_1 = q_1 \oplus \overline{q_0} \tag{4.9}$$

$$d_2 = q_2 \oplus (\overline{q_1}.\overline{q_0}) \tag{4.10}$$

$$d_3 = q_3 \oplus (\overline{q_2}.\overline{q_1}.\overline{q_0}) \tag{4.11}$$

The operation of both loadable up and down counter is similar. But count for a loadable up counter is incremented but for a loadable down counter the count is decremented. Initially the loadable up counter starts from 0 and then incremented. But for down counter a valid count should be loaded first and then should be decremented.

4.8 Even and Odd Counter

In implementing signal processing algorithms, it is sometimes required to generate even and odd address locations using counters. Even and odd counter can be easily designed using the simple up counters as shown in Fig. 4.26. In this design, even and

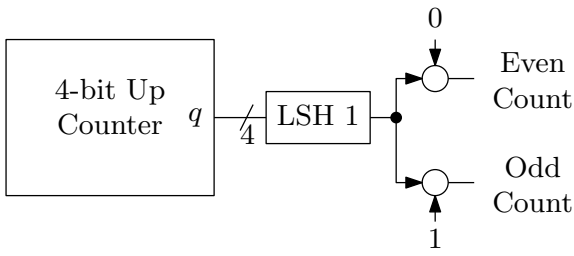


Fig. 4.26 A 4-bit even and odd counter using a simple up counter

odd counter are generated from a single up counter. In generating the even counter, up counter output (q) is first left shifted by 1-bit using wired shift block LSH1. Then using concatenation even and odd count are generated. The circle indicates the concatenation function. In even count generation, 0 is concatenated and for even count, 1 is concatenated at LSB position.

4.9 Shift Register Counters

Shift register counters are another type of counters which do not actually count consecutive numbers but generate special sequences. The sequence {8, 12, 14} can be generated using normal counters but need extra circuitry but shift register-based counters will easily generate this sequence. Thus shift register-based counters have many use in the implementation of signal processing algorithms. Two types of shift register-based counters are popular which are Ring counter and Johnson counter.

A Ring counter is basically a SISO register-based counter. Here, output of the last flip-flop is connected to the input of the first flip-flop. This counter is also known as one-hot counter. It circulates logic one around the ring. The truth table for this Ring counter is shown in Table 4.9. This Ring counter has n states for n number of

Table 4.9 Truth table for Ring counter

State	q_0	q_1	q_2	q_3
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	1	0	0	0
5	0	1	0	0
6	0	0	1	0
7	0	0	0	1
0	1	0	0	0

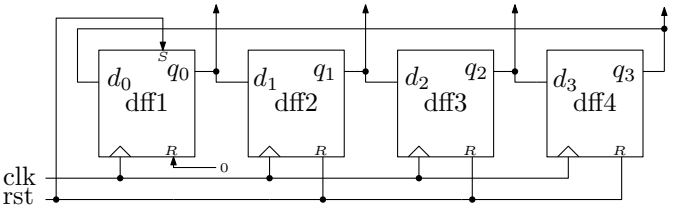


Fig. 4.27 A 4-bit Ring counter

Table 4.10 Truth table for Johnson counter

State	q_0	q_1	q_2	q_3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1
0	0	0	0	0

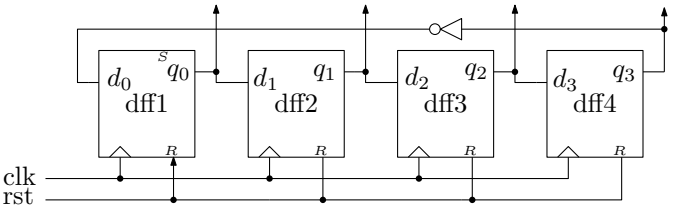


Fig. 4.28 A 4-bit Johnson counter

flip-flops used in the circular chain. The schematic of the Ring counter is shown in Fig. 4.27. Here, the flip-flop has one input for reset and one input for set. Initially the first flip-flop is set and all other flip-flops are cleared.

The above Ring counter is called as straight Ring counter. Another type of Ring counter is Johnson counter where the inverted output of the last flip-flop is the input of the first flip-flop. The truth table of the Johnson counter is shown in Table 4.10. The Johnson counter has $2n$ states instead of n states for n number of flip-flops. The schematic of the Johnson counter is shown in Fig. 4.28. Here, only the reset input is used and there is no need of setting the first flip-flop.

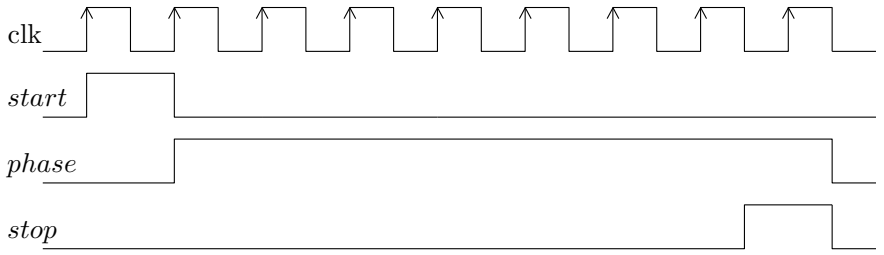


Fig. 4.29 Timing diagram of the PG block

4.10 Phase Generation Block

Phase Generation (PG) block is a very important and frequently used sequential circuit to control the selective operation of memory blocks, counters and other important computing units. This PG block along with counter circuit control the operation of a complex digital system. The PG block generates a *phase* signal when it receives a *start* pulse and the *phase* signal is deactivated when another *stop* pulse is asserted. The timing diagram of this PG block is shown in Fig. 4.29.

This kind of timing diagram can be generated in many ways. Here we have given a simple Verilog code to achieve the same timing diagram. Typically PG block is used to run a counter selectively and this block is stopped by the terminal count signal (tc) of the counter.

```
module pg(start,stop,phase,clk,reset);
input start,stop,reset,clk;
output reg phase;
initial begin phase <=0; end
always @(posedge clk)
if (reset)
  phase <= 1'b0;
else if(start)
  phase <= 1'b1;
else if(stop)
  phase <= 1'b0;
endmodule
```

4.11 Clock Divider Circuits

Clock division circuits are very important in digital systems to provide clock signal of different frequencies. For example, two types of clock frequencies are required for serial to parallel conversion and for parallel to serial conversion. In any digital platform where the digital systems are implemented, slower clocks are generated

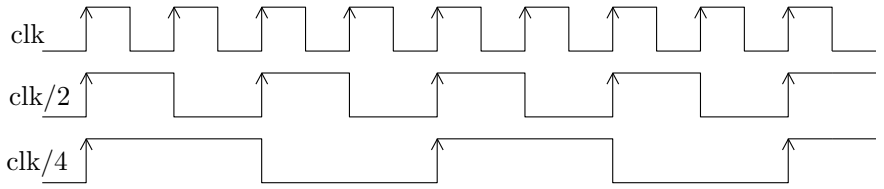


Fig. 4.30 Concept of clock division

from a high speed clock using the clock divider circuits. In Phase Locked Loops (PLL) clock divider circuits are an integral part to generate high frequency of clock signal.

In Fig. 4.30 the clock signal and the concept of clock division is shown. Here, clock division is shown for factor of 2 and 4. Note that here 50% duty cycle is considered for all signals. Frequency is defined as the inverse of clock period. Thus as the clock period increases the frequency decreases.

4.11.1 Clock Division by Power of 2

The clock division by power of 2 can be achieved using D flip-flops connected in a fashion shown in Fig. 4.31. The total number of flip-flops required to divide a clock by 2^N is N. This is an asynchronous circuit where a flip-flop is triggered by output of the previous flip-flop. The above approach of clock division can also be realized using T flip-flop as shown in Fig. 4.32. Here same number of flip-flop is required as earlier.

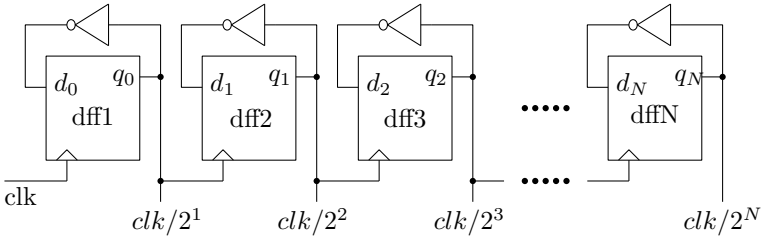


Fig. 4.31 First approach for clock division by power of two

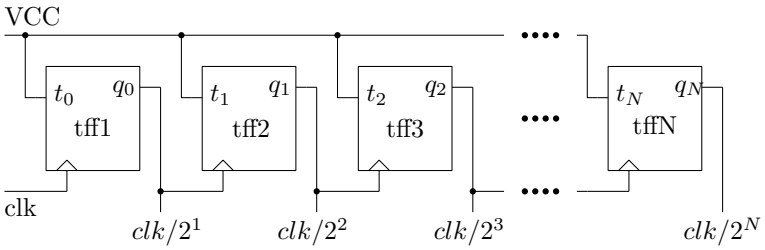


Fig. 4.32 Second approach for clock division by power of two

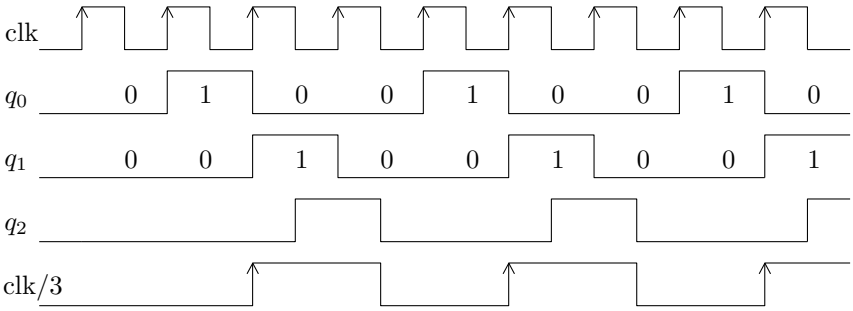


Fig. 4.33 Timing diagram for the clock division by 3

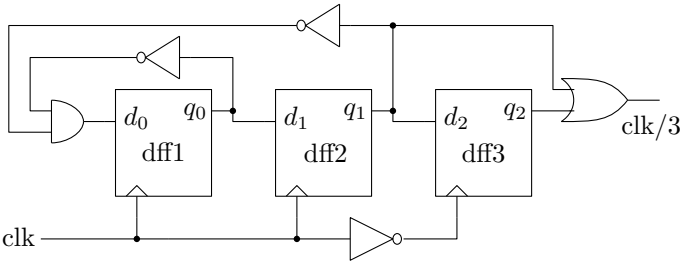


Fig. 4.34 Schematic for the clock division by 3

4.11.2 Clock Division by 3

Clock division by power of 2 is comparatively easier than clock division by any other number. To achieve clock division by any other number, lets discuss clock division circuit by 3 first. Clock division by 3 can be achieved by a mod-3 counter. A mod-3 counter can count up to 2. Clock division by 3 is explained in Fig. 4.33. Output $q1$ is shifted half clock cycle to generate $q2$. The final output is logical OR between $q1$ and $q2$. To divide by three, along with 2 flip-flops an extra flip-flop is required. This extra flip-flop is negative edge triggered. The schematic for clock divider by 3 is shown in Fig. 4.34.

4.11.3 Clock Division by 6

Previously we have discussed clock signal division by odd number which is 3. Now we will discuss clock division by a even number. This even number is 6 for example. The concept of clock division by 6 is shown in Fig. 4.35. Here, a mod counter counts up to 5 and then again starts from zero. The q_1 signal is delayed by one clock cycle and then ORed with q_3 to generate clock signal divided by 6. Thus major blocks needed here are a mod counter, a positive edge triggered flip-flop and an OR gate. The architecture is shown in Fig. 4.36.

The frequency synthesizer circuits or clock multipliers need a circuit that can divide clock signal by any integers within a range. An integer can be even or odd. Clock division by odd number can be achieved by first designing a mod counter and then using a negative edge triggered flip-flop. On the other hand, the clock division by even number needs an extra positive edge triggered flip-flop in place of negative edge triggered flip-flop. Thus a general clock divider circuit can be designed by combining clock division circuits for even and odd numbers.

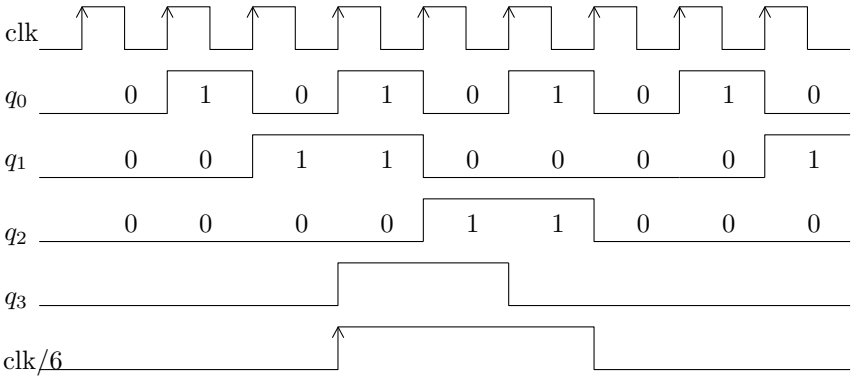


Fig. 4.35 Timing diagram for the clock division by 6

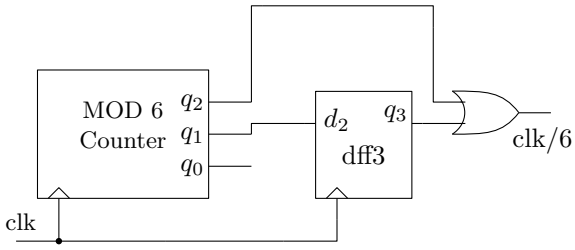


Fig. 4.36 Schematic for the clock division by 6

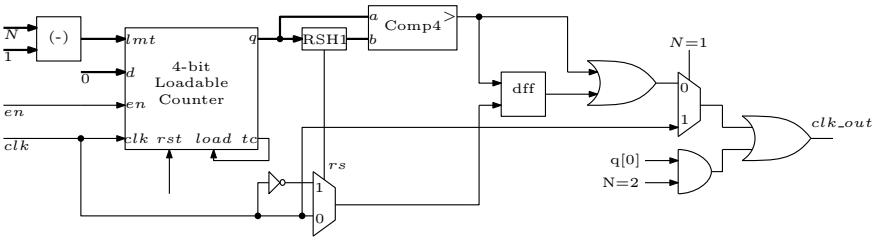


Fig. 4.37 Schematic for a programmable clock divider

4.11.4 Programmable Clock Divider Circuit

In this section, a basic architecture of a programmable clock divider is presented. Clock divider circuits have many use in frequency synthesizers. A basic circuit is presented below in Fig. 4.37. The circuit is capable of dividing the input clock by N where N can take values from 1 to 15. The circuit passes the same input to the output when N = 1. The circuit is based on a 4-bit loadable counter and a 4-bit comparator. The value of N is decremented and passed to the *lmt* input of the loadable counter. This increment is done by a simple 4-bit subtractor. The *tc* output of the loadable counter is connected to the *load* input and thus the counter acts as mod counter. The RSH1 block is a simple 1-bit right shift block. This block has another output *rs* which is the residual bit after shifting. The *rs* bit selects the negative or positive edge triggering for the D flip-flop. The above-mentioned programmable clock divider can be scaled to increase the range of clock division. This can be done by increasing the width of the loadable counter and comparator.

4.12 Frequently Asked Questions

Q1. **Write a Verilog code in behavioural style for a Mod-N counter?**
A1. Realization of a Mod-N counter using behavioural model is very straightforward forward as shown below

```
module modN_counter
  # (parameter N = 10,WIDTH = 4)
  ( input    clk,reset,
  output  reg[WIDTH-1:0] out);
  always @(posedge clk) begin
    if (reset) begin
      out <= 0;
    end else begin
      if (out == N-1)
```

```
        out <= 0;
      else
        out <= out + 1;
    end
  end
endmodule
```

Q2. **Design D flip-flip using MUX?**
A2. D flip-flop can be realized using 1-bit 2:1 MUX. First we will discuss the realization of a latch using 2:1 MUX. The realization of a latch using MUX is shown in Fig. 4.38. Here, output of the MUX is connected back to the I_0 input of the MUX. The clk signal is connected to the select signal of the MUX.

Now a Master-slave D flip-flop can be realized using one MUX as master and the other MUX as slave. The realization of MSD flip-flop is shown in Fig. 4.39.

Q3. **If an IC has maximum clock frequency of 100 MHz then how to operate a design at double data rate?**
A3. If a design is to be operated at double data rate on an IC which has maximum clock frequency of 100 MHz then the design should be operated at 200 MHz. In this case, clock frequency is multiplied by factor of 2. Frequency multiplication by any factor is achieved by Phase Locked Loop (PLL) but frequency multiplication by 2 can be achieved by considering both the clock edges instead of any one edge (either positive edge or negative edge). An example of operating a counter at double data rate is shown below.

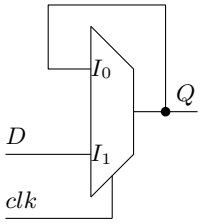


Fig. 4.38 Realization of Latch using 2:1 MUX

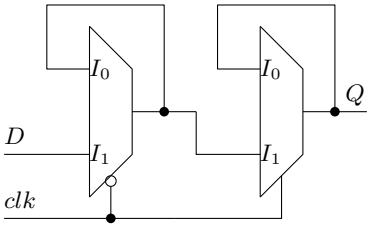


Fig. 4.39 Realization D flip-flop using MUX

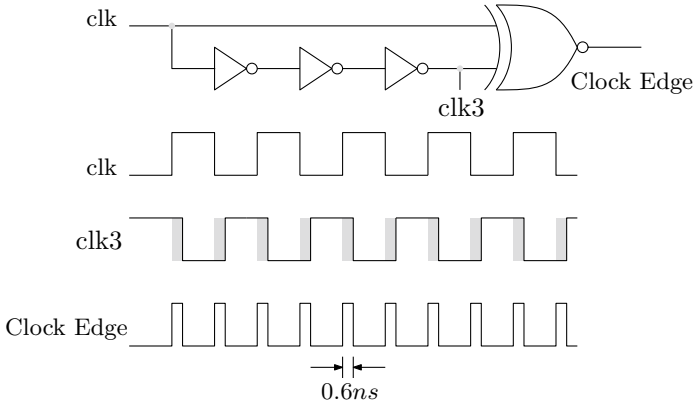


Fig. 4.40 Scheme of generation of both the clock edges

```
module counter_ddr(out,clk,en,reset);
output reg[2:0] out;
input clk,en,reset;
initial begin out=3'b000; end
always @(posedge clk or negedge clk)
if (reset) begin
  out <= 3'b000;
end else if (en)
  out <= out + 3'b0001;
else out <= out;
endmodule
```

The scheme of achieving both the clock edges is shown in Fig. 4.40. This is similar to the scheme shown in Fig. 4.1 but here an XNOR gate is used instead of an AND gate.

4.13 Conclusion

Many major sequential blocks are discussed in this chapter with their Verilog modelling. Any of these blocks can be required in design of complex digital systems. Firstly flip-flops are discussed and among all the flip-flops D Flip-Flop is mostly used in most of the designs reported in this chapter or subsequent chapters. All types of shift registers are very important in system design applications. Counter design is a very important topic in sequential circuits and we have focused on design of synchronous loadable up/down counter as it is broadly used in complex designs. Lastly, various frequency divider circuits are discussed and then a programmable clock divider is explained. The programmable clock divider has the most use in frequency multiplier circuits.

Chapter 5
Memory Design

5.1 Introduction

In implementation of digital system, we may require to store data vectors or matrices to initialize or we may require to store intermediate results. The storing of initial or intermediate data can be done using memory elements. A design can support real-time execution using memory elements. On the other hand, a design can achieve better execution time by insertion of more parallelism using memory elements. Thus memory elements play an very important role in efficient implementation of a digital system.

In this chapter, realization of basic memory elements is discussed. Different types of memory elements are realized using Verilog HDL along with their different operating modes. All the memory elements are explained using suitable example. In the digital systems, there are three types of memory elements used which are

1. Controlled Register
2. Read Only Memory
3. Random Access Memory.

In the following sections, all the memory elements are discussed one by one using their Verilog model and timing diagram.

5.2 Controlled Register

In many applications, we may require to store a single bit or a byte instead of an array. In these cases, we do not require a complete memory element. In such cases, a single register or a flip-flop will serve the purpose. This type of registers is called as controlled registers. The block diagram of a controlled register is shown in the Fig. 5.1. This register is simply a parallel in parallel out register as discussed in this chapter. Only difference is that this register has a control enable input (*en*).

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_5

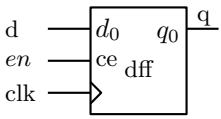


Fig. 5.1 The block diagram of a controlled register

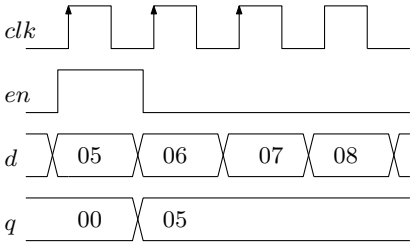


Fig. 5.2 The timing diagram for controlled register

The timing diagram for the controlled registers is shown in Fig. 5.2. The positive edge triggered clock event is considered here. When the control input is high, the byte $d = 05$ is loaded in the register. The register holds the output data whenever the control input becomes low. The output of the register can be accessed until the enable signal is again become high.

5.3 Read Only Memory

Read Only Memory (ROM) elements are used to store a vector, matrix of data or control data words in a digital system to start the execution. As the name suggests, the data stored in ROMs are only can be read and we cannot write data into it. ROM stores the essential data to start a process or specific control words. These types of memory element can be of two types which are

1. Single Port ROM (SPROM)
2. Dual Port ROM (DPROM).

5.3.1 Single Port ROM

The block diagram of a SPROM is shown in Fig. 5.3. This type of ROM element has single port ($dout$) through which data are read. The address of the memory locations is provided by the $addr$ input. Here the $addr$ input is of 3-bit width, this means there are a total of 8 memory locations and at each location 8-bit data can be stored. It has two other inputs which are clk and en. The SPROM is synchronously read that means in each clock cycle one data byte is read. The en signal is used to enable the memory element and whenever the en signal is high the data bytes can be read. Synchronous read is also possible when data reading does not depend on clock event.

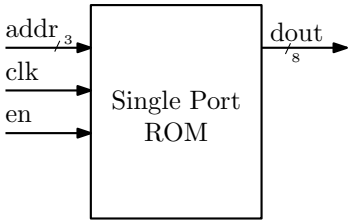


Fig. 5.3 Block diagram of a SPROM

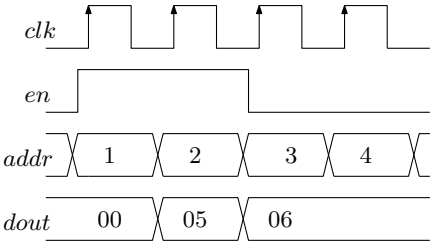


Fig. 5.4 Timing diagram for ROM

The timing diagram for the SPROM is shown in Fig. 5.4. Here, initially the data set {3, 5, 6, 7, 8, 9, 10, 11} is stored in the address locations from 0 to 7. The enable signal (*en*) is high for two clock cycle and during these two clock cycles SPROM is active. The data are read through the output port corresponding to the address locations provided on the address line during this period.

The SPROM is realized here using the Behavioural modelling style and the Verilog code is shown below. In this Verilog code there are two sections, viz., initialization and the data accessing part. Here, the memory locations are initially loaded with the case statement under one always statement. The data accessing part is written under another always statement.

```
module rom(clk,addres,data_out,en);
input clk,en;
input [2:0] addres;
output reg [7:0] data_out;
reg [7:0] mem [0:7];
//initial begin data_out = 8'b00000000; end
always @ (addres)
case (addres)
3'b000 : mem[addres] = 8'b00000001;
3'b001 : mem[addres] = 8'b00000010;
3'b010 : mem[addres] = 8'b00000011;
3'b011 : mem[addres] = 8'b00000100;
3'b100 : mem[addres] = 8'b00000101;
3'b101 : mem[addres] = 8'b00000110;
3'b110 : mem[addres] = 8'b00000111;
3'b111 : mem[addres] = 8'b00001000;
default : mem[addres] = 8'b0000000;
endcase
always@(posedge clk)
begin
```

```
if(en)begin
data_out <= mem[addres];
end else
data_out <= data_out;
end
endmodule
```

5.3.2 Dual Port ROM (DPROM)

The block diagram of a DPROM is shown in Fig. 5.5. The DPROM element has two ports which are Port A and Port B. The output port *douta* is for port A and *doutb* is for port B. All the memory locations are shared to both the ports. It is possible to read from any location through both the ports in parallel. Thus both the ports have separate address lines and enable inputs. Here, address bus *addra* is for port A and *addrb* is for port B. Both the address bus have same width. Similarly, *ena* is for port A and *enb* is for port B. The truth table for the DPROM is given in the Table 5.1. Both the ports of DPROM can be selectively used using the enable control signals.

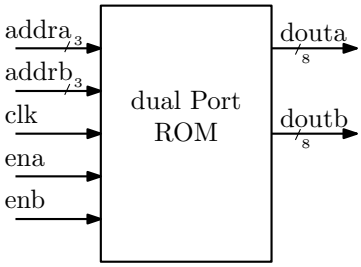


Fig. 5.5 Block diagram of DPROM

Table 5.1 Reading data from the DPROM

ena	*enb*	Operation	
		Port A	Port B
0	0	N.A.	N.A.
0	1	N.A.	Read
1	0	Read	N.A.
1	1	Read	Read

5.4 Random Access Memory (RAM)

Random Access Memory (RAM) is used to store the intermediate results in a digital system. Compared to the ROMs discussed in the previous section, data can be written into or read from the RAM blocks. RAMs have a special feature to write the data in the address locations and thus consume more hardware than the ROMs. There are two types of RAM blocks similar to the ROMs which are

1. Single Port RAM (SPRAM)
2. Dual Port RAM (DPRAM).

5.4.1 Single Port RAM (SPRAM)

The block diagram of the SPRAM is shown in Fig. 5.6. This block has two control inputs which are *en* and *we*. The *en* signal enables the SPRAM block and *we* signal enables the writing operation. The data on the data bus *din* are written when the SPRAM block is active (*en* signal is high) and the *we* signal is high. The address locations are indicated by the *addr* bus. The reading operation is taken place when the SPRAM block is active and the *we* signal is low. The status of the control signal and the corresponding operations are shown in Table 5.2.

An example of write and read operation in the SPRAM block is shown Fig. 5.7. The data samples 5 and 6 are written to the addresses 1 and 2 respectively when both the control signals are high. The addresses 1 and 2 are again inserted in the address bus to read the data samples which are written previously. Thus the latency in the

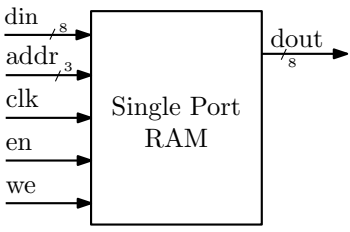


Fig. 5.6 Block diagram of a SPRAM

Table 5.2 Reading and writing operation of SPRAM

en	*we*	Operation
0	0	N.A.
0	1	N.A.
1	0	Read
1	1	Write

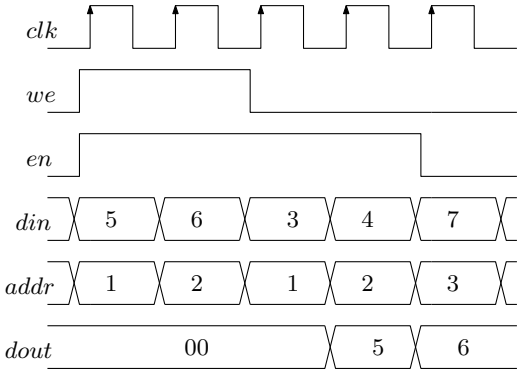


Fig. 5.7 Timing diagram for data writing and reading in SPRAM

reading operation is three clock cycles. The SPRAM memory element is realized using the behavioural coding style and the Verilog code is given below.

```
module ram(clk,addres,data_in,en,we,data_out);
input clk,en,we;
input[2:0] addres;
input[7:0] data_in;
output reg [7:0] data_out;
reg [7:0] mem [0:7];
initial begin data_out = 8'b00000000; end
always@(posedge clk)
if(en)begin
if(we)
mem[addres]=data_in;
else
data_out=mem[addres];
end
else
data_out = data_out;
endmodule
```

5.4.2 Dual Port RAM (DPRAM)

The DPRAM blocks are most promising memory block in current memory technology. The DPRAM blocks revolutionized the implementation of digital systems with its concurrent read and write facility. In implementation of many signal processing algorithms DPRAM blocks provide an easy platform for real-time operation. In real-time operation, the DPRAM block help to achieve concurrent data acquisition and data processing.

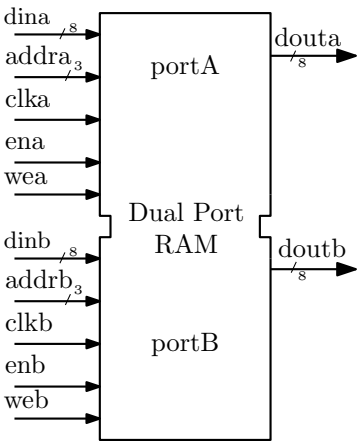


Fig. 5.8 The block diagram of DPRAM

Table 5.3 Reading and writing operation of DPRAM

ena	*wea*	*enb*	*web*	Operation	
				Port A	Port B
1	0	1	0	Read	Read
1	0	1	1	Read	Write
1	1	1	0	Write	Read
1	1	1	1	Write	Write

Similar to the DPROM block, the DPRAM block has two ports which are port A and port B. Both the ports share the common memory locations. In the previous section, we have seen that a SPRAM block has two control signals. In case of DPRAM block there are two sets of control signals, one for port A and one for port B. Port A and Port B has separate address lines. Also, DPRAM has two output ports, viz., *douta* and *doutb*. The block diagram of the DPRAM is shown in the Fig. 5.8.

The DPRAM block can be used in different modes, depending on the control inputs. The different operating modes of the DPRAM are shown in Table 5.3. Either of the port can be used for writing or reading. Also, both the ports can be used for writing or reading. The mode write-read/read-write is mostly used in which writing operation is carried out through one port and reading operation is carried out through the other port. In the write-write mode of operation, both the ports are used for writing. In this mode, it should be noted that the address on the *addra* and *addrb* bus should be different. As both the port share the same address locations and writing at the same address cannot be possible.

An example of read-write operation for the DPRAM is shown in Fig. 5.9. Here, port A is used for writing and port B is used for reading. Writing operation is carried out for two clock cycles and reading operation is started after one clock cycle. The control signals *ena* and *wea* are high during writing. During this period the input

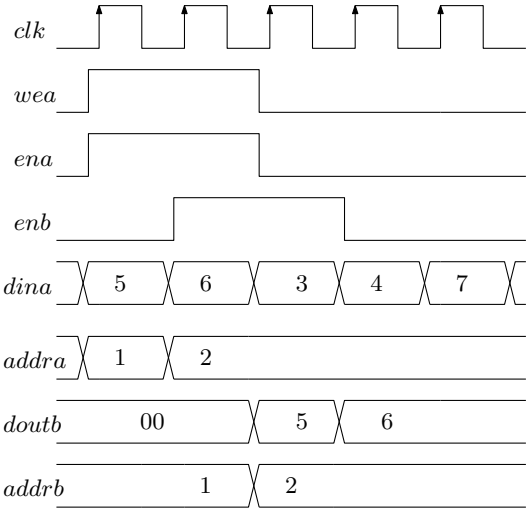


Fig. 5.9 The timing diagram for write-read operation using DPRAM

data on *dina* bus and address on *addra* bus is given. After one clock cycle, the control signal *enb* becomes high and the *web* signal is kept low. After two clock cycle of latency, data is read out through the port B. In this mode, the *douta* port is not connected. The Verilog code for the DPRAM is shown below.

```
module dp_ram(clka,clkb,ada,adb,ina,inb,
                ena,enb,wea,web,outa,outb);
input clka,clkb,ena,wea,enb,web;
input[2:0] ada,adb;
input[7:0] ina,inb;
output reg [7:0] outa,outb;
reg [7:0] mem [0:7];
initial begin
outa = 8'b00000000;
outb = 8'b00000000;
end
always@(posedge clka)
if(ena)begin
if(wea)
mem[ada]=ina;
else
outa = mem[ada];
end
else
outa = outa;

always@(posedge clkb)
if(enb)begin
if(web)
mem[adb]=inb;
else
outb = mem[adb];
```

```
end
else
outb = outb;
endmodule
```

5.5 Memory Initialization

One method of memory initialization is shown previously for ROM, which is by assign statement. But for bigger memory size it is very difficult to initialize this way. Another way is

```
initial begin
 $readmemb("c1.txt", mem1); //here, mem1 is the memory array.
 ///$readmemb("c1.mem", mem1); ///for newer Xilinx version
 ///Xilinx vivado now allow only .mem file for memory
     initialization.
 end
```

Here the *readmemb* command loads the memory array from a text file $c1.txt$ or $c1.mem$ which must be in the project directory. Here in the text file data are written in binary. To load the data in hexadecimal format the command *readmemh* can be used. Note that these commands are very useful for verifying the digital systems. But user has to check whether they are synthesizable or not. Thus if it is required to fill a ROM by pre-defined data elements then the assignment procedure using *case* statement must be opted. The intellectual property-based memory block provides easy initialization by *.coe* files. The data written in *.coe* file can be loaded to the memory arrays easily.

5.6 Implementing Bigger Memory Element Using Smaller Memory Elements

In many cases we may have smaller memory blocks and we need a memory block with bigger size. The bigger memory block can be easily realized using the smaller memory blocks. Implementation of a 32×8 ROM using four 8×8 ROMs is shown in Fig. 5.10. Here, the data width is 8-bit and the address width is 5-bit for the bigger ROM block. The lower three bits are for providing the address to the smaller blocks and the two bits from the MSB side are used to select the outputs from the ROM blocks. This way any bigger memory block can be realized by smaller blocks.

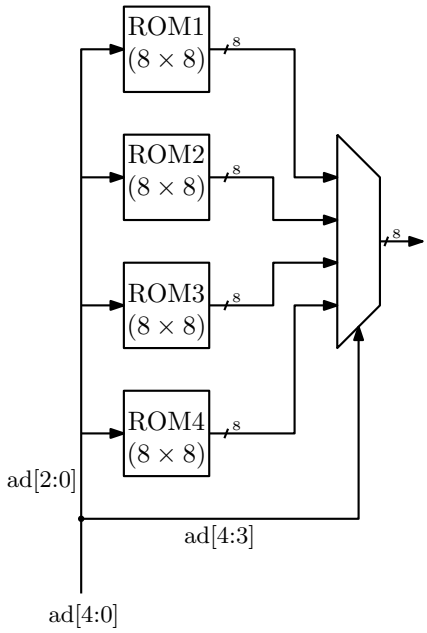


Fig. 5.10 Realization of 32×8 ROM using 8×8 ROM blocks

5.7 Implementation of Memory Elements

In this chapter, we have focused on gate level modelling and HDL realization of the memory elements whereas the actual transistor level discussion is avoided here for simplicity. In case of ASIC design either the ready made macro blocks are available or memory elements are realized using the registers. In FPGA, the memory elements are realized in two ways which are

1. Block RAMs: Block RAMs (BRAMs) are inbuilt to the FPGA devices. The size of these BRAMs is either 18 Kb or 36 Kb. These BRAMs can be configured as ROM or RAM and also as single port or dual port. These BRAMs provide many other features for high performance in terms of low power, less area and high speed.
2. Distributed RAMS: Distributed RAMs on the other hand realized using the LUTs present in the FPGA device. Thus a memory block realized using distributed RAMs consumes LUTs. Distributed memory blocks are not flexible as the Bock RAMs. Distributed RAMs can be registered or non-registered.

In case of FPGA implementation, it is a common question that which type of memory implementation should be used. If the storage requirement is high then it is better to use the Block RAMs as they do not consume LUTs and thus can be very faster. Smaller memories can be realized using the distributed RAMs. A simple dual port RAM realized using the distributed RAMs is shown in Fig. 5.11. Four D flip-flops are

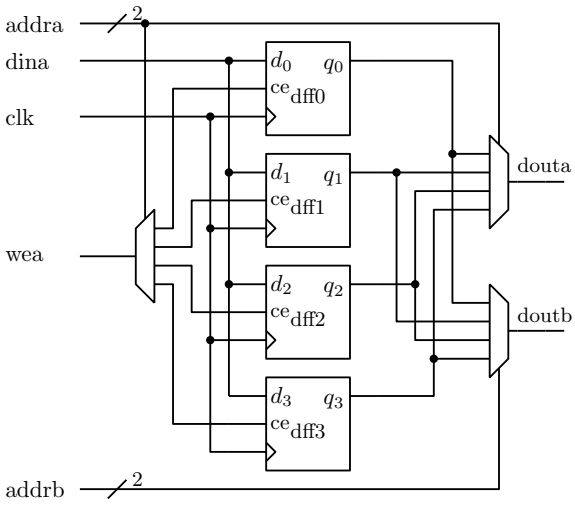


Fig. 5.11 Concept of simple dual port ram

Table 5.4 Simultaneous reading and writing using two SPRAMs

ena	*enb*	*ada*	Operation		Output
			RAM A	RAM B	
0	0	0–15	Read	Read	Douta and doutb
0	1	0–15	Read	Write	Douta
1	0	0–15	Write	Read	Doutb
1	1	0–15	Write	Write	Not applicable

used to store four bits. Two 4:1 MUXes are used to take two outputs. The DEMUX is used to provide the *wea* signal to all the flip-flops.

Sometimes it may be also be required to realize the DPRAM-like memories using the SPRAMs. The realization of the DPRAM-like memories using the two SPRAMs is shown in Fig. 5.12. Here, the data input is the same to both the SPRAMs and both the SPRAMs are connected to the same clock and same address bus. Both the RAMs have separate enable pins *ena* and *enb*. Here data acquisition is accomplished in two phases. In the phase 1, the RAM A is enabled and data is written to RAM A. In the second phase, RAM A is in reading mode and data is written into the RAM B. Output of the both the RAMs are multiplexed and *enb*1 signal is the delayed version of the *enb* signal. This configuration is not truly a DPRAM but resembles the simultaneous read and write feature and is very useful in signal acquisition of huge stream of serial data. Different modes of this type of usage of memory elements are shown in Table 5.4 for 4-bit address width.

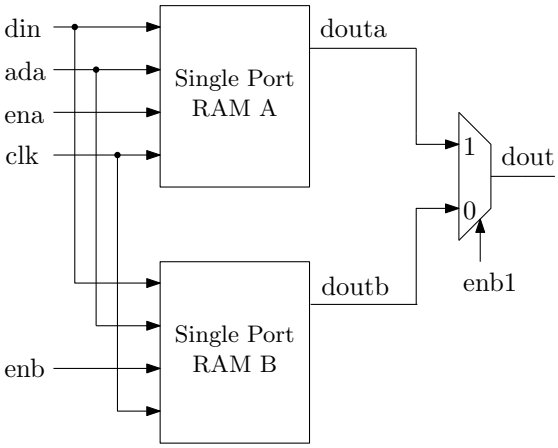


Fig. 5.12 Realization of dual port ram using single port ram

5.8 Conclusion

In this chapter, different memory elements are discussed using their Verilog HDL model, truth table and timing diagram. This chapter will help the readers to use the memory elements where necessary. Memory elements must be used carefully. If it is required to store constant data elements then ROM must be used. Designers should always try to use single port memory elements wherever applicable. Dual port memory is mostly used to implement algorithms as it enables concurrent data acquisition and processing. In FPGA implementation, BRAMs must be preferred first and then if all the BRAMs are utilized then distributed memories can be used.

Chapter 6
Finite State Machines

6.1 Introduction

Finite State Machine (FSM), also known as finite state automation, is a style of modelling a system which can be represented by finite number of states. In those systems, transition occurs from initial state to final state through some intermediate states. This transition of states is modelled by FSM and system is implemented in terms of combination of sequential and combinational circuits.

FSM design style is very significant in implementing control systems for practical automation problems. Any control system such as process parameters variation control in an industry, event control in an integrated chip, controlling of any electronic equipment, interfacing a slave IC to a controller, etc. can be designed using FSM design style. Verilog HDL is a powerful language to model a system by following its behaviour. FSM design style along with Verilog HDL is a very powerful method for rapid prototyping of critical system. In this chapter, we have presented a very brief theory on FSM design style along with some examples of system implementation using FSM. Also, we have presented Verilog codes for modelling FSMs.

6.2 FSM Types

Any digital system can be represented using finite number of states. The transition from one state to another reflects the behaviour of the system. Consider an example of a simple 2-bit synchronous up counter with enable input. The counter counts from 0 to 3 when an enable (*en*) signal is high and retains the previous output when *en* signal becomes low. The count value of this counter at a particular clock cycle is a state. A state transition occurs when the count value is updated.

State transition diagram or simply the state diagram for a system is very important to characterize the system behaviour. The state diagram for the up counter is shown in Fig. 6.1. There are four states and a state transition occurs in every clock cycle.

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_6

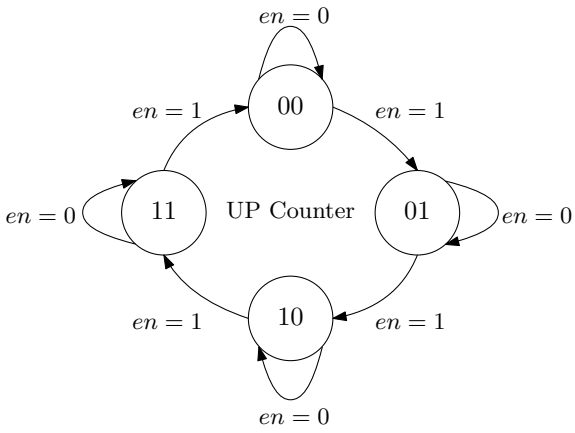


Fig. 6.1 State diagram of 2-bit synchronous up counter

Note that transition occurs only when the *en* signal is high. Otherwise the same state is retained. The behaviour of the counter is explained by the state diagram.

An FSM has three sections which are output state decoder, state register and output decoder. FSMs are generally of two types based on the dependency of the output signal on present state and input signal. These two types are

1. Mealy Machine: Mealy circuits are named after G. H, Mealy, one of the leading personalities in designing digital systems. The basic property of Mealy circuits is that the output is a function of the present input conditions and the present state (PS) of the circuit. The concept of the Mealy type of FSMs is shown in Fig. 6.2.
2. Moore Machine: Moore circuits are named after E. F. Moore, another leading personality in designing digital systems. The basic property of Moore circuits is that the output is strictly a function of the present state (PS) of the circuit. The concept of the Moore type of FSMs is shown in Fig. 6.3.

Most of the digital systems use either Moore or Mealy machine but both machines also can be used together. In the initial days of digital system design when HDL languages are not discovered, Mealy or Moore machines are realized using K-Map optimization technique. The K-map optimization technique provides an optimized solution but it is a rigorous and lengthy process. On the contrary, HDL provides

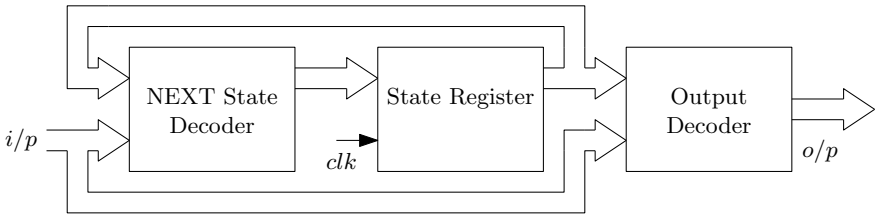


Fig. 6.2 Concept of the Mealy type FSMs

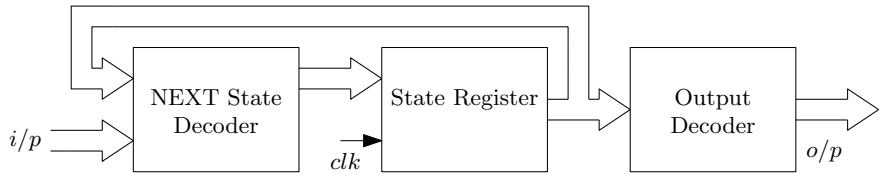


Fig. 6.3 Concept of the Moore type FSMs

an easy solution to the design of FSMs by saving design time. In this chapter, we will discuss the design of some of the digital systems using both Mealy and Moore machine. A comparison between these two machines is given in this chapter. We will end up our discussion by explaining some of the popular FSM state optimization techniques.

6.3 Sequence Detector Using Mealy Machine

Sequence detector is a good example to describe FSMs. It produces a pulse output whenever it detects a pre-defined sequence. In this chapter, we have considered a 4-bit sequence '1010'. The first step of an FSM design is to draw the state diagram. The sequence detectors can be of two types: non-overlapping and overlapping. For example, consider the input sequence as '1101010101'. Then in non-overlapping style, the output y will be '0000100010' and the output y in with overlapping style will be '0000101010'. This situation for Mealy machine is shown in Fig. 6.4.

In Fig. 6.4, it is observed that the overlapping style also considers the non-overlapping sequences. The Design of both types of sequence detectors will be discussed in this chapter. Lets consider the non-overlapping case first. The state diagram of the '1010' sequence detector using the Mealy machine in non-overlapping style is shown in Fig. 6.5. The Mealy machine has four states and these states are defined by parameter S. Each state corresponds to each bit in the sequence '1010'. For example, state S_0 corresponds to 1 and state S_1 corresponds to 0. The states are written inside a circle and along the branch it is written as 'input/output'.

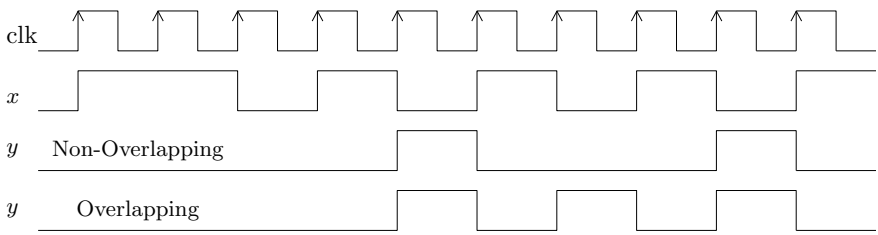


Fig. 6.4 '1010' sequence detector output for overlapping and non-overlapping case

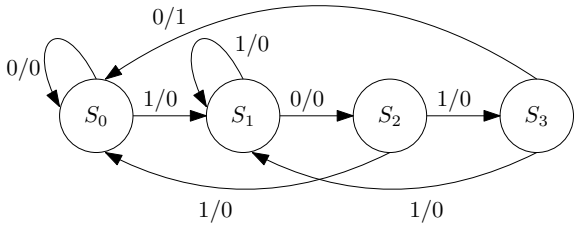


Fig. 6.5 State diagram for '1010' sequence detector using Mealy machine in non-overlapping style

The drawing of the correct state diagram is very crucial in designing FSMs. In PS of S_0, if the input is $x = 1$ then a state transition occurs from S_0 to S_1 but if $x = 0$ then next state (NS) is equal to PS (S_0). Similarly, in the PS of S_1, state transition occurs only when x is equal to 0. Similarly the other state transitions can be assigned. Here, S_0 is the starting state and S_3 is the end state when the output is 1. If there is a false input, the next state will be the nearest similar state or there will be no state transition. It is to remember that for any combinations we have to reach the branch where the output is '1'. For example, if there is a false state at $PS = S_1$ then end state can be reached as $S_1S_1S_2S_3S_0$. But if there is a false state at $PS = S_2$ then the path from S_2 to S_3 does not complete the sequence. Thus we have to start again from S_0. Consider an input sequence as '011010' then the sequence of next states will be $S_0S_1S_1S_2S_3S_0$.

Once the state diagram is drawn, now we can proceed to realize the sequence detector in terms of hardware using K-map optimization method. The steps to be followed to realize an FSM are the formation of state table, assignment of the states, and then formation of the excitation table according to a flip-flop by which the FSM will be realized. The state table is shown in Table 6.1 for sequence 1010.

The next step in designing an FSM is assignment of the states or representation of the states using their binary equivalent or any other coding techniques. The assignment of the states is shown in Table 6.2. Here, S_0 is represented as 00, S_1 is represented as 01, S_2 is represented as 10 and S_3 is represented as 11.

The next step is to form the excitation table. Here, we have chosen D flip-flop to implement the FSM and in a D flip-flop input is reflected on output after a clock period. The excitation table is shown in Table 6.3. Two D flip-flops will be required as we need only two bits to represent the states. The Boolean expressions for the

Table 6.1 State table for the state diagram shown in Fig. 6.5

Present state	Next state		Output	
	X = 0	X = 1	X = 0	X = 1
S_0	S_0	S_1	0	0
S_1	S_2	S_1	0	0
S_2	S_0	S_3	0	0
S_3	S_0	S_1	1	0

Table 6.2 State assignments for the state table shown in Table 6.1

Present state	Next state		Next state	
	X = 0	X = 1	X = 0	X = 1
00	00	01	0	0
01	10	01	0	0
10	00	11	0	0
11	00	01	1	0

Table 6.3 Excitation table for 1010 sequence detector

Present state		Input	Next state		F/F inputs		Output
q_1	q_0	x	q_1^*	q_0^*	d_1	d_0	y
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	1	0	1	0	0
0	1	1	0	1	0	1	0
1	0	0	0	0	0	0	0
1	0	1	1	1	1	1	0
1	1	0	0	0	0	0	1
1	1	1	0	1	0	1	0

inputs of the D flip-flops can be obtained by K-map optimization technique using this excitation table. The K-maps for determining the expressions for d_0, d_1 and y are shown in Fig. 6.6. The Boolean expressions are shown below

$$d_0 = x \tag{6.1}$$

$$d_1 = q_1\bar{q_0}x + \bar{q_1}q_0\bar{x} \tag{6.2}$$

$$y = q_1q_0\bar{x} \tag{6.3}$$

The circuit for 1010 sequence detector using Mealy machine can be drawn using the above equations. The sequence detector is shown in Fig. 6.7. Here output of the sequence detector (y) is a function of input signal x and also function of the present states (q_1, q_0). This reflects that the sequence detector is strictly a Mealy machine.

The above discussion was for 1010 sequence detector using Mealy machine in non-overlapping style. Earlier in Fig. 6.4, we have shown the nature of the output signal in case of overlapping style. The state diagram for the 1010 sequence detector using Mealy machine for overlapping style is shown in Fig. 6.8. In the PS of S_3 as soon as $x = 0$ is received, state transition occurs from S_3 to nearest state for 1 which is S_2. Previously, in non-overlapping case, sequence is started from S_0 but here searching for new sequence can be started from the intermediate states also.

q_1 \ q_0x	00	01	11	10
0	0	1	1	0
1	0	1	1	0

(a) K-Map for d_0.

q_1 \ q_0x	00	01	11	10
0	0	0	0	1
1	0	1	0	0

(b) K-Map for d_1.

q_1 \ q_0x	00	01	11	10
0	0	0	0	0
1	0	0	0	1

(c) K-Map for Output.

Fig. 6.6 K-Map for 1010 sequence detector using Mealy machine

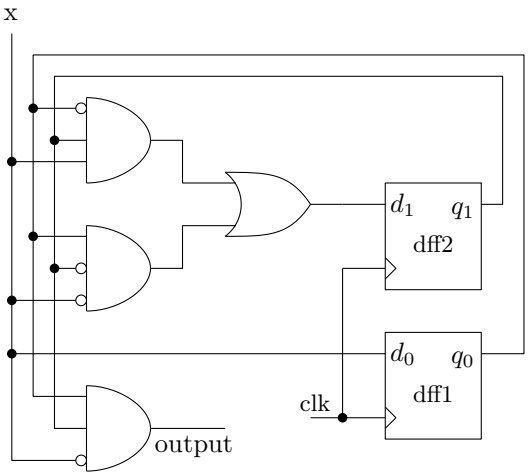


Fig. 6.7 Hardware realization of the 1010 sequence detector using Mealy machine using non-overlapping style

Table 6.4 State table for the state diagram of sequence '1010' sequence detection using Moore Machine in non-overlapping style

Present state	Next state		Output	
	X = 0	X = 1	X = 0	X = 1
S_0	S_0	S_1	0	0
S_1	S_2	S_1	0	0
S_2	S_0	S_3	0	0
S_3	S_4	S_1	0	0
S_4	S_0	S_1	1	1

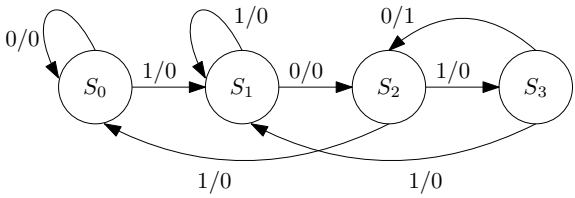


Fig. 6.8 1010 sequence detector using Mealy machine for overlapping style

6.4 Sequence Detector Using Moore Machine

In this section, we will discuss implementation of the same 1010 sequence detector but using Moore machine. In Moore type FSMs, we also can have two types of sequence detectors which are non-overlapping and overlapping. The variation of the output signal y in both the cases is shown in Fig. 6.9. It can be observed that the output is one clock cycle delayed compared to the output of Mealy machine.

The state diagram for 1010 sequence detector using Moore machine in non-overlapping style is shown in Fig. 6.10. Here inside the circle $S_0/0$ represents that in the PS of S_0 output is zero. The input signal (x) is written along the branches from one state to another. The Moore machine needs extra states compared to the Mealy machine and in these extra states (S_4) output is 1. Here one extra state is used and in this extra state, output is always 1. Thus the objective is to reach the output state from any state. Considering the same input sequence as '011010' then the sequence of next states will be $S_0S_1S_1S_2S_3S_4$.

The 1010 sequence detector using Moore machine is also designed here using K-map. The state table is shown in Table 6.4. The state table is formed according to the state diagram shown in Fig. 6.10. The next step is to assign the steps. Here, four states are used and thus minimum of three bits are needed to represent them. The state assignment table is shown in Table 6.5.

The excitation table for 1010 sequence detector using the Moore machine in non-overlapping style is shown in Table 6.6. Here, same D flip-flip is used. In comparison to the Mealy machine, three flip-flops are required. Thus using the K-map optimization, the Boolean expression for the input of the flip-flops and the output is derived.

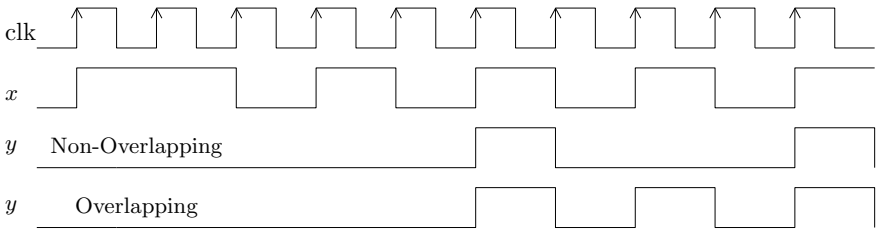


Fig. 6.9 '1010' sequence detector output for overlapping and non-overlapping case for Moore machine

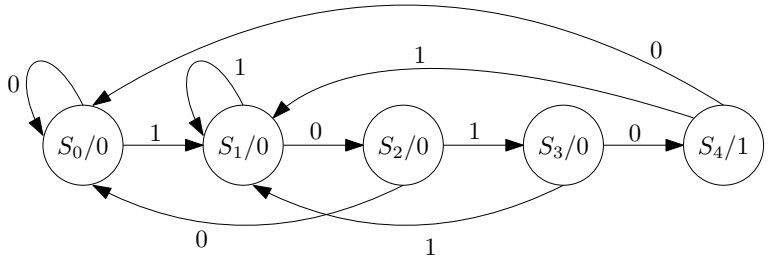


Fig. 6.10 State diagram for 1010 sequence detector using Moore machine in non-overlapping style

Table 6.5 State table with state assignments for sequence '1010' detector using Moore Machine in non-overlapping style

Present state	Next state		Next state	
	X = 0	X = 1	X = 0	X = 1
000	000	001	0	0
001	010	001	0	0
010	000	011	0	0
011	100	001	0	0
100	000	001	1	1

Table 6.6 Excitation table for 1010 sequence detector using Moore machine

Present state			Input	Next state			F/F inputs			Output
q_2	q_1	q_0	X	q_2^*	q_1^*	q_0^*	d_2	d_1	d_0	y
0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0	0	1	0
0	0	1	0	0	1	0	0	1	0	0
0	0	1	1	0	0	1	0	0	1	0
0	1	0	0	0	0	0	0	0	0	0
0	1	0	1	0	1	1	0	1	1	0
0	1	1	0	1	0	0	1	0	0	0
0	1	1	1	0	0	1	0	0	1	0
1	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	1	0	0	1	1

q_2q_1 \ q_0x	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	X	X	X	X
10	0	1	X	X

(a) K-Map for d_0.

q_2q_1 \ q_0x	00	01	11	10
00	0	0	0	1
01	0	1	0	0
11	X	X	X	X
10	0	0	X	X

(b) K-Map for d_1.

q_2q_1 \ q_0x	00	01	11	10
00	0	0	0	0
01	0	0	0	1
11	X	X	X	X
10	0	0	X	X

(c) K-Map for d_2.

q_2q_1 \ q_0x	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	X	X	X	X
10	1	1	X	X

(d) K-Map for Output.

Fig. 6.11 K-Map for 1010 sequence detector using Moore machine

The K-map optimization is shown in Fig. 6.11. The don't care conditions may be used to generate optimized expressions but here these conditions are not considered. The Boolean expressions without optimization are shown below.

$$d_o = \bar{q_2}x + \bar{q_1}\bar{q_0}x \tag{6.4}$$

$$d_1 = \bar{q_2}q_1\bar{q_0}x + \bar{q_2}\bar{q_1}q_0\bar{x} \tag{6.5}$$

$$d_2 = \bar{q_2}q_1q_0\bar{x} \tag{6.6}$$

$$y = q_2\bar{q_1}\bar{q_0} \tag{6.7}$$

The hardware realization of the 1010 sequence detector using Moore machine in non-overlapping style is shown in Fig. 6.12. Here, three D flip-flops are used. The output of this sequence detector is a function of only the present states. Thus this sequence detector is strictly a More machine.

The above discussion was for sequence detector using Moore machine in non-overlapping style. Similarly we can convert this detector to detect overlapping sequences. The nature of the output was previously shown in Fig. 6.9. The state diagram for overlapping style is shown in Fig. 6.13. In the PS of S_4 if the input is 1, then state transition occurs from S_4 to S_3 to search the nearest state for 0.

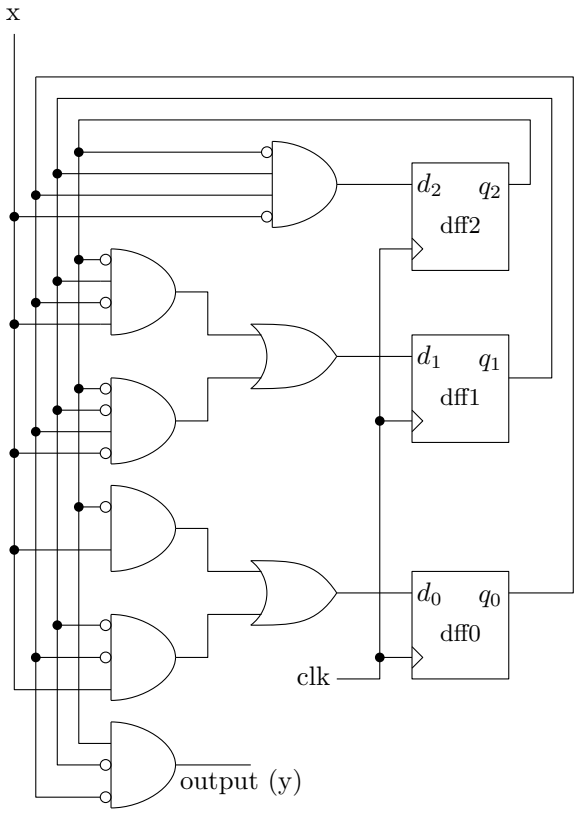


Fig. 6.12 Hardware implementation of 1010 sequence detector using Moore machine in non-overlapping style

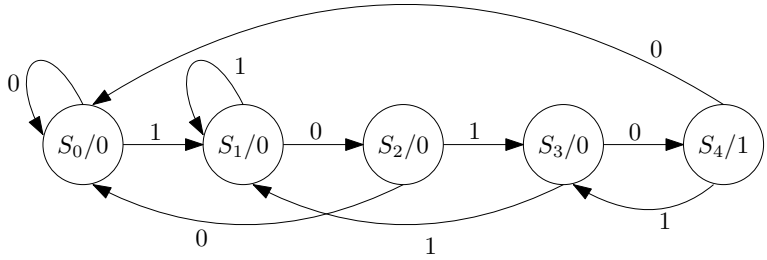


Fig. 6.13 State diagram of 1010 sequence detector using Moore machine in overlapping style

Table 6.7 Comparison between Mealy and Moore machine

Mealy machine	Moore machine
Output depends on present input and present state of the circuit	Output depends only on the present state of the circuit
Required less number of states	Required more number of states
Asynchronous output generation though the state changes synchronous to the clock	Both output and state change synchronous to the clock edge
Faster, the output is generated on the same clock cycle	The output is generally produced in the next clock cycle
Glitches can be generated as output change depends on input transition	Safer to use, because they change states on the clock edge

6.5 Comparison of Mealy and Moore Machine

In the above sections, we have studied Mealy and Moore machine with an example of detecting a sequence. But the question now arises that which machine is better to use and what are the differences between them. A comparison between these two FSM styles is shown in Table 6.7.

Thus it is clear from the above discussion is that the Mealy machine is faster, asynchronous and glitches can be occurred in Mealy machine. On the other hand, Moore machine is synchronous but needs more states. To avoid the glitches in the Mealy machine, registered Mealy machine or synchronous Mealy can be used. Synchronous Mealy machines are nothing but Moore machines without output state decoder.

6.6 FSM-Based Serial Adder Design

Serial adder design using FSM is a popular design which is frequently used in literature. Here in this chapter, we will design a serial adder using both Mealy and Moore machine. The serial adder adds two single bit stream a and b. The truth table for adding two inputs is shown in chapter 3. The adder results two outputs sum and c_{out}. In this addition, if carry is generated and then it is added with the next set of inputs.

The state diagram for the serial addition in case of Mealy machine is shown in Fig. 6.14. There are two states defined based on the carry output. The state S_0 is for carry equal to zero and S_1 is for carry equal to 1. The corresponding state table is shown in Table 6.8. This table can be used to design the hardware for serial adder. There is a provision of $reset$ signal. This $reset$ signal will initially set the NS as 0. Thus addition starts from the state S_0. The corresponding architecture of the serial adder using Mealy machine is shown in Fig. 6.15.

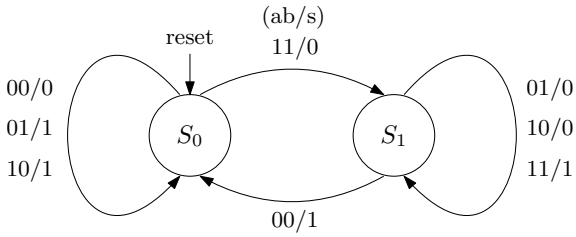


Fig. 6.14 State diagram for serial adder using Mealy machine

Table 6.8 State table for serial adder using Mealy machine

PS	a	b	NS	Sum	c_{out}
S_0	0	0	S_0	0	0
S_0	0	1	S_0	1	0
S_0	1	0	S_0	1	0
S_0	1	1	S_1	0	1
S_1	0	0	S_0	1	0
S_1	0	1	S_1	0	1
S_1	1	0	S_1	0	1
S_1	1	1	S_1	1	1

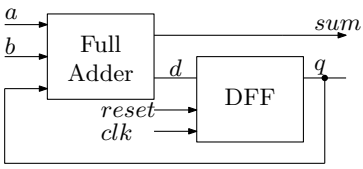


Fig. 6.15 Architecture of serial adder using Mealy machine

In case of Moore based serial adder design, we need more states. These are

1. S_0—Carry is 0 and sum is 0.
2. S_1—Carry is 1 and sum is 0.
3. S_2—Carry is 0 and sum is 1.
4. S_3—Carry is 1 and sum is 1.

The state diagram is shown in Fig. 6.16. The state table can be formed using the state diagram shown in Fig. 6.16. In Moore case, there are four states and thus we need two bits to represent the states. The state table is shown in Table 6.9. This means one extra D flip-flop will be used in the architecture of serial adder but the architecture will be fully synchronous. This structure is shown in Fig. 6.17.

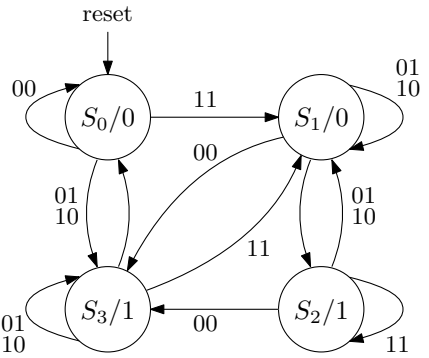


Fig. 6.16 State diagram of serial adder using Moore FSM

Table 6.9 State table for serial adder using Moore machine

Present state	Next state				Output (*sum*)
	$ab = 00$	$ab = 01$	$ab = 10$	$ab = 11$	
S_0	S_0	S_3	S_3	S_1	0
S_1	S_3	S_1	S_1	S_2	0
S_2	S_3	S_1	S_1	S_2	1
S_3	S_0	S_3	S_3	S_1	1

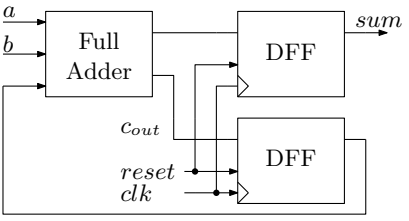


Fig. 6.17 Implementation of serial adder using Moore FSM

6.7 FSM-Based Vending Machine Design

Vending Machine is a practical example where FSM can be used. The ticket dispatcher unit at the stations, and the can drinks dispatcher at the shops are some examples of Vending machines. Here in this chapter we will try to understand a simple Vending machine which dispatches a can of coke after deposition of 15 rupees. The machine has only one hole to receive coins that means customers can deposit one coin at a time. Also the machine receives only 10 (T) or 5 (F) rupee coins and it doesn't give any change. So the input signal x can take values like

1. x = 00, no coin deposited.
2. x = 01, 5 rupee coin (F) deposited.
3. x = 10, 10 rupee coin (T) deposited.
4. x = 11 (forbidden) Both coins can't be deposited at the same time.

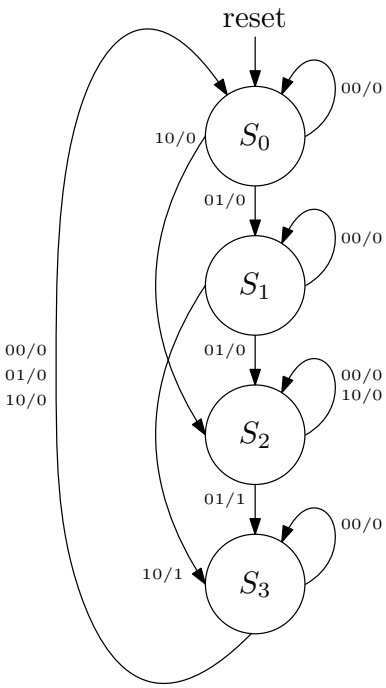


Fig. 6.18 State diagram for the simple vending machine problem using FSM

Table 6.10 State table for FSM-based vending machine

Present state	Next state				Output
	$ab = 00$	$ab = 01$	$ab = 10$	$ab = 11$	
S_0	S_0	S_3	S_3	S_1	0
S_1	S_3	S_1	S_1	S_2	0
S_2	S_3	S_1	S_1	S_2	1
S_3	S_0	S_0	S_0	S_1	1

Also a customer can deposit 15 rupees as $10 + 5 = 15, 5 + 10 = 15$ and $5 + 5 + 5 = 15$. If more money is deposited than 15 then the machine will be in the same state asking the customer to deposit right amount. The state diagram for the vending machine is shown in Fig. 6.18. In order to get a can of drinks the customer has to give 15 rupees. In terms of machine language, the objective is to reach the step S_3 from the step S_0. Once the step S_3 is reached the Vending machine dispatches a can and asks the customer if he wants another. The architecture of the Vending machine can be designed using the state table shown in Table 6.10.

6.8 State Minimization Techniques

State minimization is important if we want to reduce the number of states in a complex FSM which has many states. If the number of states is reduced then the reduced number of bits will be required for state assignment. Thus less number of flip-flops will be required and so there is a chance to reduce combinational or sequential blocks also. This is why it is important to reduce the number of states.

Before going to discuss the state minimization techniques some definitions must be known. If input $x = 0$ is applied to a state machine in $PS = S_1$ and NS is S_2 then S_2 is a 0-successor of S_1. Similarly, If input $x = 1$ is applied to a state machine in $PS = S_1$ and NS is S_3 then S_3 is a 1-successor of S_1. These two successors are generally called as k-successors. Two states S_1 and S_2 can be called equivalent if the following conditions are satisfied

1. The states S_1 and S_2 should have same output for all the input sequences.
2. Their k-successors also obey the first criteria.

Some of the popular state minimization techniques are

1. Row Equivalence Method
2. Implication Chart Method
3. State Partition Method
4. Some Heuristic Methods.

6.9 Row Equivalence Method

In the row equivalence method [26], it is checked that rows of a state table are equivalent or not. Here, a comparatively strict definition of state equivalence is used. The conditions for two states S_1 and S_2 to be equivalent are

1. The outputs must be same for both the states.
2. The k-successors must be same for all the input conditions.

The row equivalence method is explained with the help of the state table shown in Table 6.11. Here, states S_1 and S_5 can be said equivalent as their output is same and their k-successors are also same. Similarly, the states S_2, S_4 and S_6 are equivalent. Thus using this simple state minimization technique the state table is reduced to the state table as shown in Table 6.12. Here, S_1^* is written for states S_1 and S_5. Similarly, S_2^* is written for states S_2, S_4 and S_6.

Table 6.11 State table example for state minimization

Present state	Next state		Output
	$x = 0$	$x = 1$	
S_0	S_1	S_2	0
S_1	S_3	S_4	0
S_2	S_5	S_6	0
S_3	S_3	S_4	1
S_4	S_5	S_6	0
S_5	S_3	S_4	0
S_6	S_5	S_6	0

Table 6.12 State table example for state minimization using row equivalence

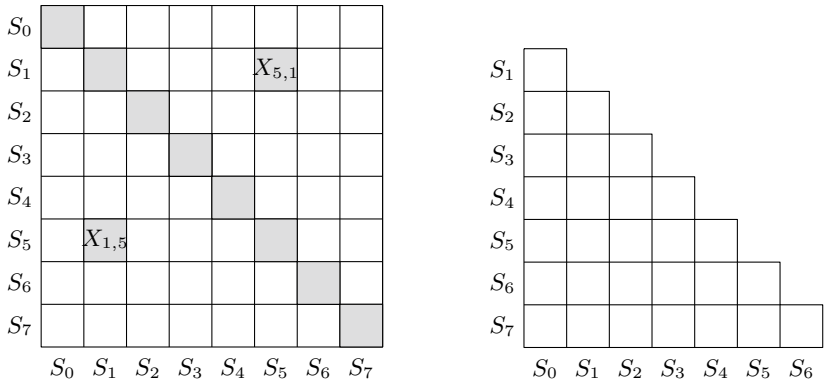
Present state	Next state		Output
	$x = 0$	$x = 1$	
S_0	S_1^*	S_2^*	0
S_1^*	S_3	S_2^*	0
S_2^*	S_1^*	S_2^*	0
S_3	S_3	S_2^*	1

6.10 Implication Chart Method

Implication Chart method [47] is a very popular method for reducing the steps of an FSM and it is more machine friendly method. In this method, a chart is prepared to find the equivalent steps. The implication chart is shown in Fig. 6.19a. States are written along the x-axis as S_0, S_1,...,S_n, and the states are written along the y-axis in the reverse order. A square X_{ij}, contains the equivalent states between S_i and S_j. The Implication Chart can be modified as $X_{ij} = X_{ji}$ and thus the triangle above the diagonal can be removed. Also, the diagonal can be removed as there is no sense to find equivalence between a state and itself. The reduced implication chart is shown in Fig. 6.19b.

First step of state minimization is to fill the squares of the chart correctly. The implication chart is shown in Fig. 6.20 according to the state table shown in Table 6.13. The squares are filled as by two rows. First row consists of 0-successors and the second row consists of 1-successors. In the topmost square from the left side, first row is $S_3 - S_1$ and the second row is $S_5 - S_2$. These pairs are called as implied state pairs. During the filling of squares, some boxes are crossed for states which cannot be combined. For example, S_2 and S_0 cannot be combined as their output is different.

State minimization by the implication chart method is accomplished by some passes until no further combination is possible. Searching for equivalent states is



(a) Implication Chart. (b) Reduced Implication Chart.

Fig. 6.19 Implication chart and reduced implication chart

Fig. 6.20 Implication chart after filling of squares and marking cross for not related states

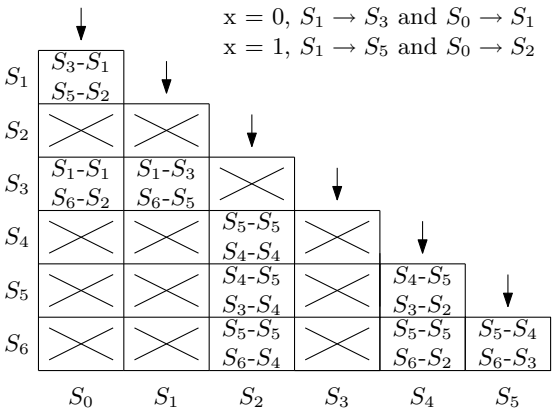


Table 6.13 State table example for state minimization

Present state	Next state		Output
	$x = 0$	$x = 1$	
S_0	S_1	S_2	1
S_1	S_3	S_5	1
S_2	S_5	S_4	0
S_3	S_1	S_6	1
S_4	S_5	S_2	0
S_5	S_4	S_3	0
S_6	S_5	S_6	0

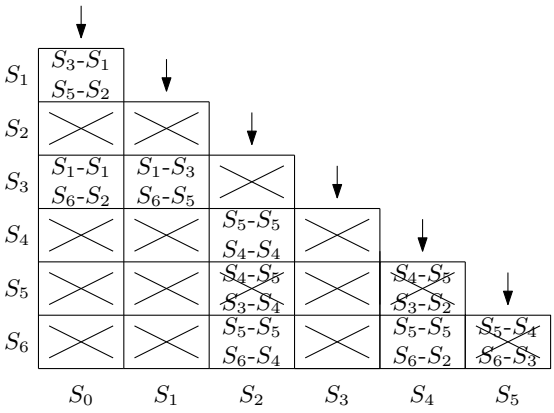


Fig. 6.21 Implication chart after pass 1

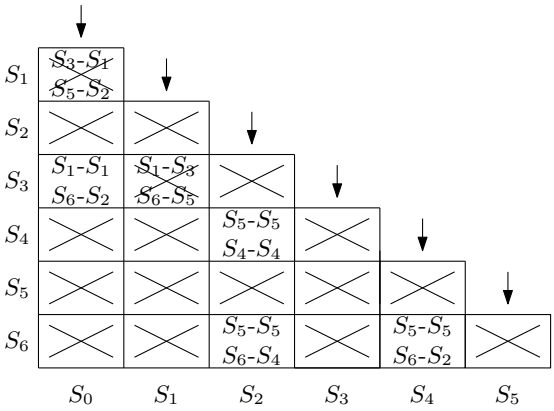


Fig. 6.22 Implication chart after second pass

done from the top to the bottom starting from the left side of the chart. The state S_1 can be combined with state S_0 or S_3 as shown in Fig. 6.21. But state S_2 and S_5 cannot be combined as the square corresponding to the states S_3 and S_4 is marked crossed. So, the square block corresponding to S_2 and S_5 is cross marked. Similarly for the square block X_{65} as state S_6 and S_3 cannot be combined. At the end of the first pass it can be concluded that S_0, S_1 and S_3 can be equivalent. Also, S_2, S_4 and S_6 can be combined. The state S_5 cannot be combined with any other states.

Second pass started similarly from the top to bottom starting from the left to right as shown in Fig. 6.22. Here, the square block X_{10} is cross marked as we have seen in the earlier pass that S_2 cannot be combined with S_5. Similarly, the square block X_{31} is cross marked because of state S_5. After the second pass it can be concluded that S_0 and S_3 are equivalent. Also, S_2, S_4 and S_6 are equivalent.

Another pass can be run but after the third pass there is no new state for reduction. Thus finally the modified state table can be formed. Here, S_0 and S_3 are combined as

Table 6.14 State table in Table after state minimization

Present state	Next state		Output
	$x = 0$	$x = 1$	
S_0^*	S_1	S_2	1
S_1	S_0^*	S_5	1
S_2^*	S_5	S_2^*	0
S_5	S_2^*	S_0^*	0

S_0^*. The states S_2, S_4 and S_6 are combined as S_2^*. The modified state table is shown below in Table 6.14. Here, initially there were seven states and after minimization there are now four states.

6.11 State Partition Method

State Partition method is another powerful method for state minimization. In this technique, the states are partitioned into groups based on the possibility that they can be combined. Lets consider the same state table as shown in Table 6.13 for state minimization using state partition method. This method also performs state minimization after some passes. These passes are described below.

1. Start: In the first pass there is only one group and this is $P = (S_0S_1S_2S_3S_4S_5S_6)$.
2. First Pass: In the first pass, the states which have different outputs are partitioned in separate groups. Here, S_0, S_1 and S_3 have same output and thus grouped in $P_1 = (S_0S_1S_3)$. The rest of the states have same output and thus they grouped as $P_2 = (S_2S_4S_5S_6)$.
3. Second Pass: In this pass, the states are partitioned based upon their k-successors. In order to combine two states, their k-successors should be in the same partition or group. Consider the first partition P_1 and their k-successors are

   (a) 0-successors—$S_0 \rightarrow S_1$, $S_1 \rightarrow S_3$ and $S_3 \rightarrow S_1$. Here, S_1 and S_3 belong to the same group P_1. States S_0, S_1 and S_3 can be combined.
   (b) 1-successors—$S_0 \rightarrow S_2$, $S_1 \rightarrow S_5$ and $S_3 \rightarrow S_6$. Here, S_2, S_5 and S_6 belong to the same group P_2. States S_0, S_1 and S_3 can be combined.

   Now consider the second partition P_2 and their k-successors are

   (a) 0-successors—$S_2 \rightarrow S_5$, $S_4 \rightarrow S_5$, $S_5 \rightarrow S_4$ and $S_6 \rightarrow S_5$. Here, S_4 and S_5 belong to the same group P_2. States S_2, S_4, S_5 and S_6 can be combined.
   (b) 1-successors—$S_2 \rightarrow S_4$, $S_4 \rightarrow S_2$, $S_5 \rightarrow S_3$ and $S_6 \rightarrow S_6$. Here, S_2, S_4 and S_6 belong to group P_2 but S_3 belong to the group P_1. Thus states S_2, S_4 and S_6 can be combined but S_5 is a different state and it is assigned to another partition P_3.

4. Third Pass: In the third pass we have three partitions $P_1 = S_0S_1S_3$, $P_2 = S_2S_4S_6$ and $P_3 = S_5$. Same steps are followed in this pass also. Consider the partition P_1 and their k-successors are

   (a) 0-successors—The 0-successors are S_1 and S_3 which belong to same group.
   (b) 1-successors—The 1-successors are S_2, S_5 and S_6. Here, S_2 and S_6 belong to P_2 but S_5 belong to P_3. Thus S_1 cannot be combined with S_0 and S_3. The state S_1 must be kept in another partition.

   Similar analysis can be run for partition P_2 and P_3.

After the third pass, the partitions are updated as $P_1 = S_0S_3$, $P_2 = S_1$, $P_3 = S_2S_4S_6$ and $P_4 = S_5$. Further passes can be run but after the third pass there is no change in the partitions. Thus final states are same as result of the third pass. The state minimization result is same as the Implication chart produces.

6.12 Performance of State Minimization Techniques

In this chapter, we have discussed three techniques for state minimization. The row equivalence method is a basic method for state minimization. This method does not always lead to an optimized number of states. The Implication chart method is a rigorous technique for state minimization. Though the chart preparation is difficult, it supports machine implementation. The partition-based technique is simple but another rigorous method for state minimization. It is also machine realizable. Some of the heuristic methods based on K-map also exist but they are more pen-and-paper methods.

State minimization is a wonderful way to minimize the states and thus to reduce hardware elements. But there is not always a need to optimize an FSM. This is because minimization of states can lead to complex circuit in terms of reduced flip-flops but increased complexity in combinational path. The state minimization is also difficult when there are don't care conditions. It is even more difficult when don't care conditions exist on output. Thus minimization algorithms must be carefully applied.

6.13 Verilog Modelling of FSM-Based Systems

```
module melfsm(din, reset, clk, y);
input din;
input clk;
input reset;
output reg y;
reg [1:0] cst, nst;
parameter S0 = 2'b00, //all state
          S1 = 2'b01,
```

```
          S2 = 2'b10,
          S3 = 2'b11;
always @(cst or din)
 begin
 case (cst)
   S0: if (din == 1'b1)
          begin
         nst = S1;
          y=1'b0;
          end
       else
          begin
          nst = cst;
          y=1'b0;
          end
   S1: if (din == 1'b0)
          begin
        nst = S2;
          y=1'b0;
          end
       else
          begin
          y=1'b0;
           nst = cst;
          end
   S2: if (din == 1'b1)
          begin
         nst = S3;
          y=1'b0;
          end
          else
          begin
          nst = S0;
          y=1'b0;
          end
   S3: if (din == 1'b0)
          begin
         nst = S0;
          y=1'b1;
          end
       else
          begin
          nst = S1;
          y=1'b0;
          end
   default: nst = S0;
  endcase
end
always@(posedge clk)
          begin
           if (reset)
             cst <= S0;
           else
```

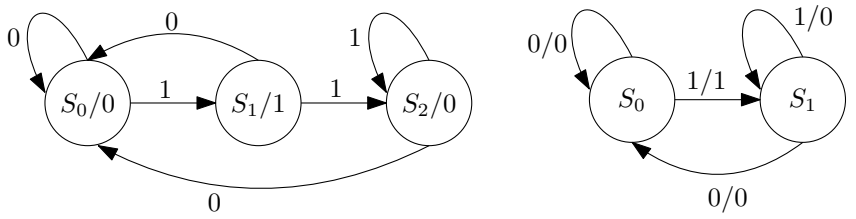
```
            cst <= nst;
          end
endmodule

module moore2(din, reset, clk, y);
input din;
input clk;
input reset;
output reg y;
reg [2:0] cst, nst;
parameter S0 = 3'b000,
          S1 = 3'b001,
          S2 = 3'b010,
          S3 = 3'b011,
            S4 = 3'b100;
always @(cst or din)
 begin
 case (cst)
   S0: if (din == 1'b1)
          begin
         nst = S1;
          y=1'b0;
          end
      else nst = cst;
   S1: if (din == 1'b0)
          begin
        nst = S2;
          y=1'b0;
          end
       else
          begin
          nst = cst;
          y=1'b0;
          end
   S2: if (din == 1'b1)
          begin
         nst = S3;
          y=1'b0;
          end
            else
          begin
          nst = S0;
          y=1'b0;
          end
   S3: if (din == 1'b0)
          begin
         nst = S4;
          y=1'b0;
          end
       else
          begin
          nst = S1;
          y=1'b0;
          end
```

```
S4: if (din == 1'b0)
          begin
         nst = S1;
          y=1'b1;
          end
          else
          begin
          nst = S3;
          y=1'b1;
          end
   default: nst = S0;
  endcase
end
always@(posedge clk)
begin
          if (reset)
          cst <= S0;
          else
            cst <= nst;
end
endmodule
```

6.14 Frequently Asked Questions

Q1. **Design a positive edge detector using both Mealy and Moore machine.**
A1. Positive edge of a signal is defined as the transition of the signal from the zero state to one state. The detection of positive edge is very important in many applications. This problem of edge detection can be easily solved using FSM design technique. In case of Mealy machine, two states can be defined as S_0 for signal value equal to 0 and S_1 for signal value equal to 1. In case of Moore machine one extra state is required. The state diagram for both types of edge detectors is shown in Fig. 6.23. The Verilog code for the edge detector can be written easily based on these state diagrams.

Q2. **Design a square wave generator with programmable duty cycle using FSM.**
A2. Square wave generator using FSM is another important application of FSM-based design. Here two states can be defined which are S_0 and S_1. The state S_0 stands



(a) Moore based Edge detector. (b) Mealy based Edge detector.

Fig. 6.23 Design of edge detector using Moore and Mealy machine

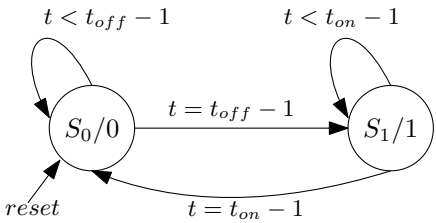


Fig. 6.24 State diagram of FSM-based square wave generator

for off state and state S_1 stands for on state. Here, t is a parameter to track the on and off time of the square wave. During t_{on} the value of the signal is high and during t_{off} the signal is low. The state diagram for square wave generator is shown in Fig. 6.24. The Verilog code for FSM-based square wave generator is shown below.

```
 module square_wave
#( parameter
        N = 4,
        on_time = 3'd5,
        off_time = 3'd3
)
(
    input wire clk, reset,
    output reg s_wave
);

localparam
    S0 = 0,
    S1 = 1;
reg PS, NS;
reg[N-1:0] t = 0;

always @(posedge clk) begin
    if (reset == 1'b1)
        PS <= S0;
    else
        PS <= NS;
end

always @(posedge clk) begin
    if (PS != NS)
        t <= 0;
    else
        t <= t + 1;
end

always @(PS, t) begin
    case (PS)
        S0 : begin
            s_wave = 1'b0;
            if (t == off_time - 1)
                NS = S1;
```

```
            else
                NS = S0;
        end
        S1 : begin
            s_wave = 1'b1;
            if (t == on_time - 1)
                NS = S0;
            else
                NS = S1;
        end
    endcase
end
endmodule
```

Q3. **Design a serial two's complementer circuit using FSM based design methodology?**
A3. Two's complement of a number is obtained by taking one's complement first and then adding 1 at the LSB position. In other words, starting from the LSB all the bits are retained til 1-bit is found and then retaining this bit, all the following bits are complemented. For example, if input $x = 0110$ then output is $y = 1010$. If this hardware is to be designed using FSM, then two states are to be assigned. State S_0 is for tracking the first 1-bit from the LSB and the state S_1 is for complementing the remaining bits which follow the 1-bit. The state diagram for the serial two's complementer circuit using Mealy Machine is shown in Fig. 6.25.

Q4. **Draw the state diagram for FSM-based serial odd parity indicator?**
A4. A serial odd parity indicator is a circuit which indicates that the number of 1's till a time instant is odd and is indicated by giving 1 at output. If there are even 1's then the output becomes 0. For example, if input is $x = 0010\_1100\_1101\_0100$ then the output is $y = 0011\_0111\_0110\_0111$. The design of this kind of circuit is straightforward and two states are required. One for output 0 and one for output 1. The state diagram is shown in Fig. 6.26.

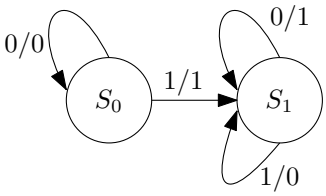


Fig. 6.25 State diagram of FSM-based 2's complementer

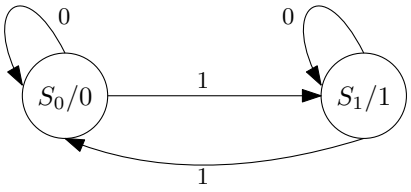


Fig. 6.26 State diagram of FSM-based serial odd parity checker

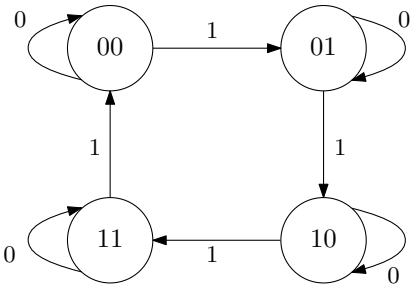


Fig. 6.27 State diagram of FSM-based Gray counter (Enable based)

Q5. **Identify the circuit realized using the state diagram shown in Fig. 6.27 and explain its working?**
A5. The state diagram shown in Fig. 6.27 is of enable-based 3-bit Gray counter. Here, the LSB of the counter works as the enable bit. In order to count in Gray sequence the proper sequence of enable input is required. If enable input is continuously 0 then present state is always the initial state and if it is contentiously 1 then state transition occurs in circular order.

6.15 Conclusion

In this chapter, we have learned about the FSMs. We first discussed the theoretical background of the FSM. The FSM is mainly of two types which are Mealy and Moore. Design of both the FSM types is discussed using the sequence detector problem in non-overlapping and overlapping style. The K-map optimization method is used here to implement the FSMs in terms of sequential and combinational circuits. Then the performance of both the FSM types is compared. It was found that Mealy machine is faster but has chances of producing glitches. The Moore machines need a higher number of states but synchronous in nature.

We have also focused on state minimization techniques. There are three popular techniques, viz., Row equivalence method, Implication chart and partition method. The row equivalence method is not so robust to produce fully optimized FSM. But the implication chart method is a robust technique to reduce number of states and also machine implementable. Partition method is simpler than the Implication chart method and also a robust technique. State minimization techniques are not always preferred as optimization can lead to a more complex combinational circuit.

We have also covered some of the example designs using the FSM like serial adder ad Vending machine. At the end of the chapter, we have discussed Verilog implementation of the FSM. Verilog is a very easy way to implement the FSM. Here, only state diagram is important to design a FSM. Thus Verilog is an easier method for implementing FSM. Some Verilog codes are also provided at the end of this chapter.

Chapter 7
Design of Adder Circuits

7.1 Introduction

Adders are the most important basic logic element in designing a digital system. Adders are used in almost every processing units. Addition operation is also used to compute multiplication, division or square root. Thus speed of an adder is very important parameter to decide performance of a design.

So, it is important to optimize the performance of this logic block. There are various techniques discovered to make faster adder, to design adders with low power consumption or to design such an adder which is comparatively faster but area efficient. Application-specific trade-offs are to be considered. The basic idea about various types of adders is discussed in this chapter.

7.2 Ripple Carry Adder

Basic theory of adder, subtractor or adder/subtractor is explained in Chap. 3. A parallel addition mechanism for 4 bits of data width was also shown. In that parallel adder, there are four FA blocks used. This parallel adder is known as ripple carry adder. The carry signal is propagated from the first block to the last block. The delay of this ripple carry adder is $n.t_{FA}$ where n indicates the number of bits. In case of higher data widths this carry chain is higher. This carry propagation must be completed before the final output is used in other operations. A fast adder may be designed by reducing this carry propagation time.

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_7

7.3 Carry Look-Ahead Adder

Carry Look-ahead Adder (CLA) is a very common fast adder which is used frequently in digital systems. CLA is based on computing all the carries in parallel. The truth table of a full adder is shown in Table 7.1. Two parameters are introduced in case of CLA. These are shown below:

$$G_i = A_i B_i C_{in} + A_i B_i \overline{C_{in}} = A_i B_i \tag{7.1}$$

$$P_i = A_i \oplus B_i \tag{7.2}$$

The variable G_i represents the generated carry and P_i is the propagation condition. An alternate equation of propagation condition is

$$P_i = A_i + B_i \tag{7.3}$$

Both the equations can be used. The first equation is hardware efficient but the second equation is more popular as only an OR gate is required. The use of the second equation will be discussed later. Until now we will stick to the first expression.

From the truth table of the full adder, some conclusions can be made. In the first two rows, output carry (C_{out}) is always zero. In the rows from 3 to 6, input carry (C_{in}) propagates to output carry and in the last two rows, C_{out} is one. It can be said that C_{out} is equal to the generated carry (G_i) in the rows $\{1, 2, 7, 8\}$ and equal to C_{in} in rows 3–6 when propagation condition (P_i) is satisfied. The general Boolean expression for the output carry can be written in the following style:

$$C_{i+1}(C_{out}) = G_i + P_i C_i \tag{7.4}$$

Initially C_0 is equal to C_{in} and $i = 0, 1...(n-1)$, where n is data width. The computation of the C_{i+1} is progressed as

Table 7.1 Truth table for a full adder

Row	a	b	c_{in}	s	c_{out}
1	0	0	0	0	0
2	0	0	1	1	0
3	0	1	0	1	0
4	0	1	1	0	1
5	1	0	0	1	0
6	1	0	1	0	1
7	1	1	0	0	1
8	1	1	1	1	1

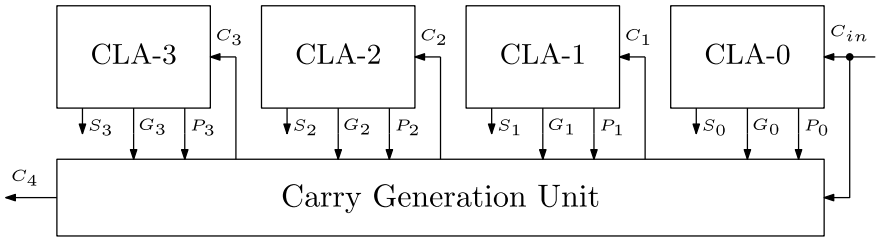
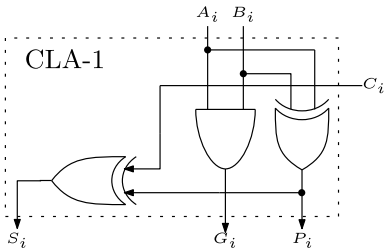


Fig. 7.1 Structure of the 4-bit CLA

Fig. 7.2 Structure of the basic sub-block (CLA-1) for 4-bit CLA architecture



$$C_1 = G_0 + P_0C_0 \tag{7.5}$$

$$C_2 = G_1 + G_0P_1 + C_0P_0P_1 \tag{7.6}$$

$$C_3 = G_2 + G_1P2 + G_0P_1P_2 + C_0P_0P_1P_2 \tag{7.7}$$

$$C_4 = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3 + C_0P_0P_1P_2P_3 \tag{7.8}$$

All the carry signals depend on C_{in} and the inputs instead depending on the other carry signals. Then the equation for sum (S) is

$$S_i = A_i \oplus B_i \oplus C_i \tag{7.9}$$

Based on the above expressions, the architecture for the 4-bit CLA is shown in Fig. 7.1. There are four basic sub-blocks used. Each sub-block is called here as CLA-i. A CLA-i block computes P_i, G_i and S_i. This block passes P_i and G_i to a carry generation unit. The structure of the CLA-i block is shown in Fig. 7.2. The carry generation unit is shown in Fig. 7.3.

CLA is a very fast adder in which timing complexity to compute the sums or carries is almost the same for every value of i irrespective of n. The maximum timing path to compute C_3 is XOR2-AND4-OR4 and that for C_4 is XOR2-AND4-OR5. Thus the timing path varies with respect to gate fan-in. Similarly, the maximum timing path for S_3 is (XOR2-AND4-OR4)-(XOR2-XOR2). It takes $5t_g$ delay to compute sum of numbers of any width where t_g is the delay of a single gate.

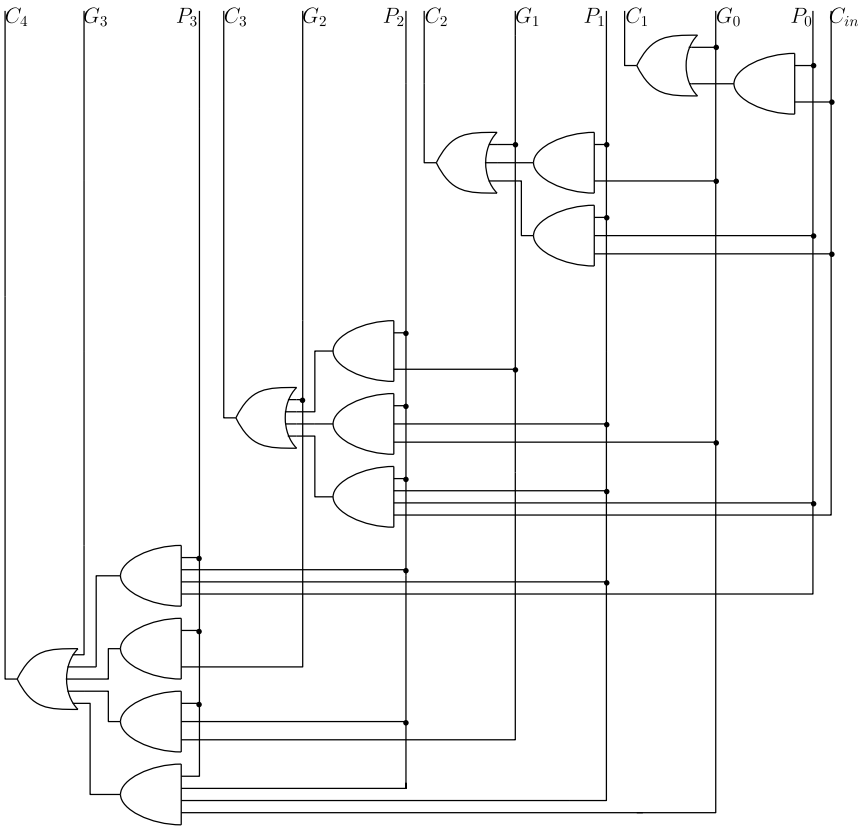


Fig. 7.3 Architecture of the carry propagation chain for the 4-bit CLA

7.3.1 Higher Bit Adders Using CLA

Carry look-ahead adders prove to be very fast regardless the data width. But for higher data width, the hardware complexity to generate carry is extremely high and thus area increases. To reduce the hardware overhead, several techniques are proposed to efficiently design the carry generation unit for higher order adders. Higher order adders can be implemented in terms of smaller adders. Like a 64-bit adder can be implemented using 16 numbers of 4-bit adders. Generally, 4-bit adders are taken as the basic building block.

In a 4-bit adder, 4 bits are grouped together and so the corresponding signals can also be grouped. The equations for block propagated carry, block generated carry and carryout are given below:

$$P_{i:j} = \begin{cases} P_i, & \text{for } i = j \\ P_i P_{i-1:j}, & \text{for } i > j \end{cases} \tag{7.10}$$

$$G_{i:j} = \begin{cases} G_i, & \text{for } i = j \\ G_i + P_i G_{i-1:j}, & \text{for } i > j \end{cases} \tag{7.11}$$

$$C_{i+1} = G_{i:j} + P_{i:j} C_j \tag{7.12}$$

First consider an 8-bit adder is to be implemented using CLA. It can be designed using two 4-bit CLAs as shown in Fig. 7.4. More levels of carry look-ahead generators can be added as the data width (n) increases. The required number of block is log_b^n where b is the blocking factor. The blocking factor is 4 in Fig. 7.4. The addition time for carry look-ahead adder is therefore proportional to log_b^n. Thus this type of adder is famously known as logarithmic adder.

The carry generation unit in the CLAs is the most critical. Several tree-based designs are available in literature for the carry generation unit. The tree-based designs are implemented using parallel prefix circuit. A prefix circuit receives inputs $x_1, x_2, ..., x_n$ and generates outputs like $x_1, x_2 \circ x_1, ..., x_n \circ x_{n-1} .... \circ x_1$ where $\circ$ is symbol for binary associative operation. CLAs with prefix circuit are sometimes called as prefix adders. To understand the prefix tree adders a new associative binary function is defined.

$$(P_{i:m}, G_{i:m}) \circ (P_{v:j}, G_{v:j}) = (P_{i:m} P_{v:j}, G_{i:m} + P_{i:m} G_{v:j}) \tag{7.13}$$

and the following equation is also valid:

$$(P_{i:j}, G_{i:j}) = (P_{i:m}, G_{i:m}) \circ (P_{v:j}, G_{v:j}) \tag{7.14}$$

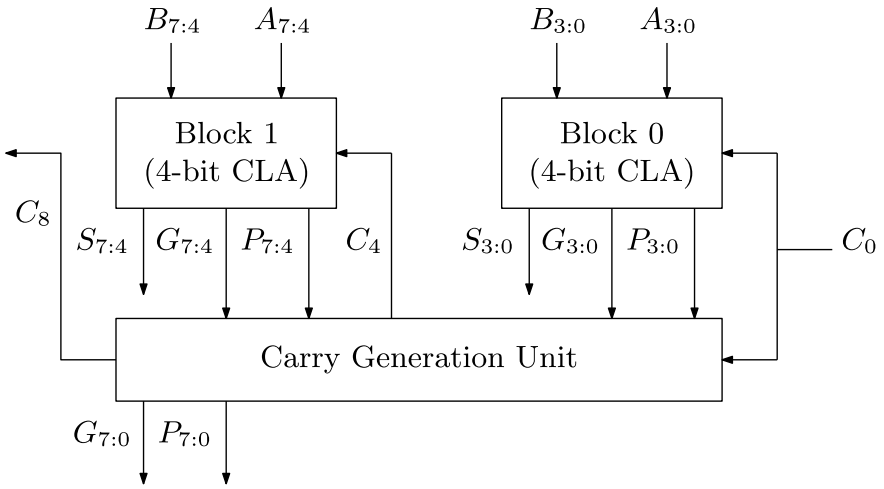


Fig. 7.4 Design of an 8-bit carry look-ahead adder using 4-bit CLAs

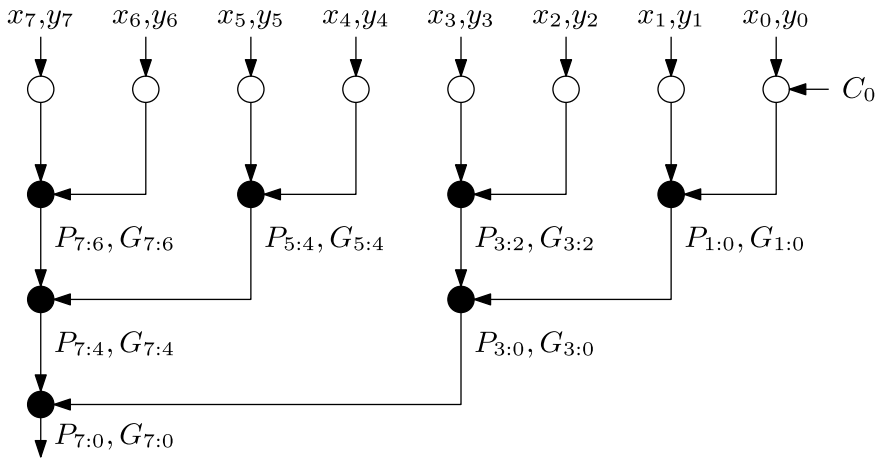


Fig. 7.5 Basic tree structure for the carry generation unit

where the range of m is $j < m < i$ and condition for v is $v \leq (m-1)$. For $n = 4$, associative operation will be

$$(P_{3:2}, G_{3:2}) \circ (P_{1:0}, G_{1:0}) = (P_{3:2}P_{1:0}, G_{3:2} + P_{3:2}G_{1:0}) \tag{7.15}$$

The operation of the tree structures can be understood by taking an example. The equations involved in calculating C_8 are

$$\begin{aligned}(P_{7:0}, G_{7:0}) &= (P_{7:4}, G_{7:4}) \circ (P_{3:0}, G_{3:0}) && (7.16)\\ &= \{(P_{7:6}, G_{7:6}) \circ (P_{5:4}, G_{5:4})\} \circ \{(P_{3:2}, G_{3:2}) \circ (P_{1:0}, G_{1:0})\} && (7.17)\\ &= \{[(P_7, G_7) \circ (P_6, G_6)] \circ [(P_5, G_5) \circ (P_4, G_4)]\} && (7.18)\\ &\circ \{[(P_3, G_3) \circ (P_2, G_2)] \circ [(P_1, G_1) \circ (P_0, G_0)]\} && (7.19)\end{aligned}$$

The tree structure to compute C_8 is shown in Fig. 7.5. This is a simple tree structure where each node has fan-in of two inputs but this may not be the case for other tree structures.

7.3.2 Prefix Tree Adders

Some of the popular prefix adders are discussed in this chapter. All the prefix adders have their own pros or cons. The performance of a prefix tree depends on number of used nodes, number of levels, fan-in/fan-out count or ease of implementation.

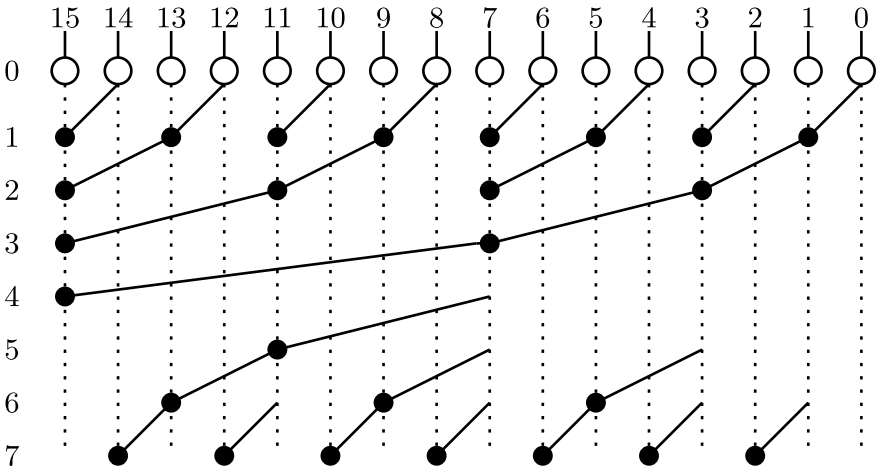


Fig. 7.6 Data flow graph for Brent–Kung prefix tree

7.3.2.1 Brent–Kung Parallel Prefix Tree

Brent–Kung parallel prefix tree adder [17] is a very popular prefix tree adder to implement the carry generation unit. The data flow path for Brent–Kung adder is shown in Fig. 7.6. Some of the important features of this configuration are

1. Total stages—The Brent–Kung tree has $(2\log_2 n - 1)$ number of stages in comparison to the minimum $\log_2 n$ number of stages. Thus for $n = 16$, there are seven stages in the Brent–Kung prefix tree. As the depth is higher, the delay is also higher.
2. Total nodes—$n/2\log_2 n$.
3. Fan-in/Fan-out—The Brent–Kung tree has the advantage of low fan-in and fan-out at each stage. This value is two at each stage. Thus Brent–Kung tree avoids explosion of wires and odd computation.

7.3.2.2 Ladner–Fischer Adder

Brent–Kung prefix adder tree has higher number of stages. Ladner–Fischer prefix tree adder [41] is an improvement over Brent–Kung tree in terms of number of stages. The data flow path for Ladner–Fischer adder is shown in Fig. 7.7. The features of this adder are

1. Stages—Ladner–Fischer prefix tree adder has $\log_2 n$ stages and thus it has lower depth and supposed to have less delay.
2. Total nodes—This prefix tree has the same number of nodes $(n/2\log_2 n)$ that of Brent–Kung tree.

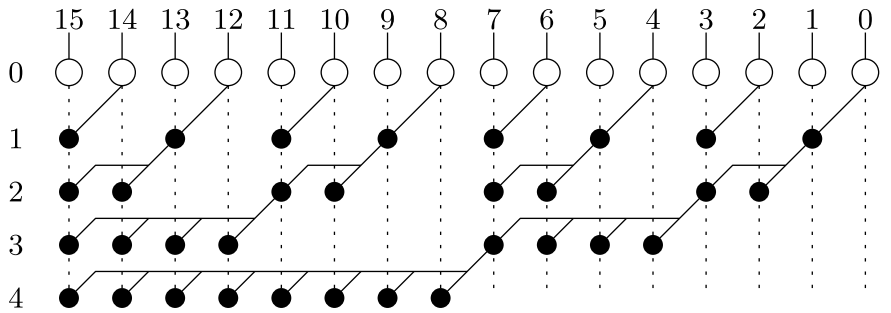


Fig. 7.7 Ladner–Fischer prefix tree adder data flow

3. Fan-in/Fan-out—Nodes are having high fan-out compared to Brent–Kung tree. Thus implementation has high congestion. The need of adding buffers and wiring delay can slow down the executing speed.

7.3.2.3 Kogge–Stone Adder

The Kogge–Stone prefix tree adder [35] improves the speed as well as improves the implementation ease by reducing the congestion due to high fan-in/fan-out. The data flow path for Kogge–Stone adder is shown in Fig. 7.8. The features of Kogge–Stone adder are

1. Stages—This adder has the same number of stages ($\log_2 n$) as that of Ladner–Fischer prefix tree adder.
2. Total Nodes—Total number of nodes used in this type of adder is $(n-1)+(n-2)+(n-4)+(n-8)+$..... This tree adder uses higher number of nodes and thus has higher area.
3. Fan-in/Fan-out—This tree adder has fan-in/fan-out of two at each stage. Performance is faster but still suffers from delay due to long wires.

7.3.2.4 Han–Carlson Adder

Han–Carlson prefix tree adder [30] is proposed by combining the advantages of Brent–Kung adder and Kogge–Stone adder. The data flow path for Han–Carlson adder is shown in Fig. 7.9.

1. Stages—This tree adder has $\log_2 n + 1$ number of stages. For $n = 16$, it has five stages. Thus it has moderate stages.
2. It is a Hybrid Adder, where middle stages resemble the Kogge–Stone adder.

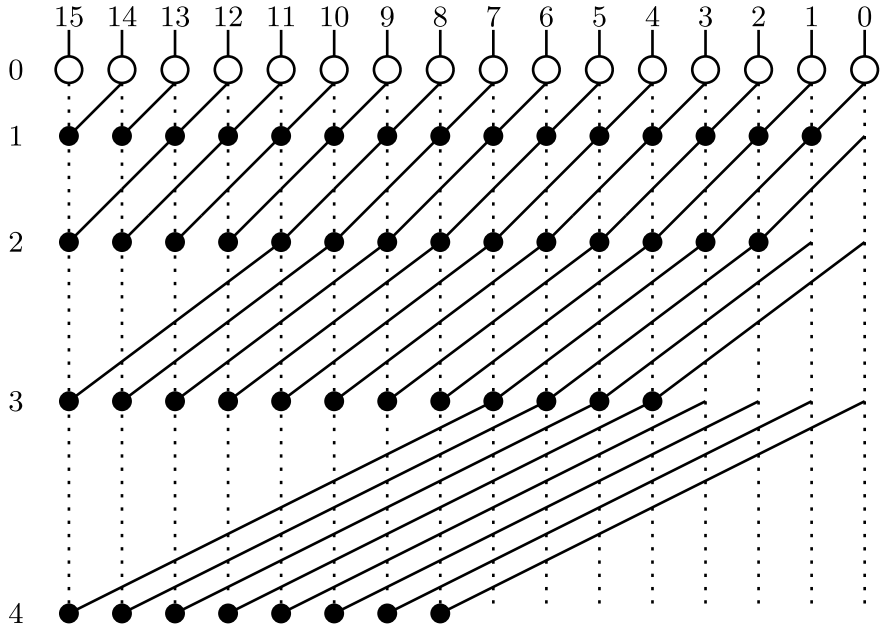


Fig. 7.8 Data path structure of Kogge–Stone prefix adder

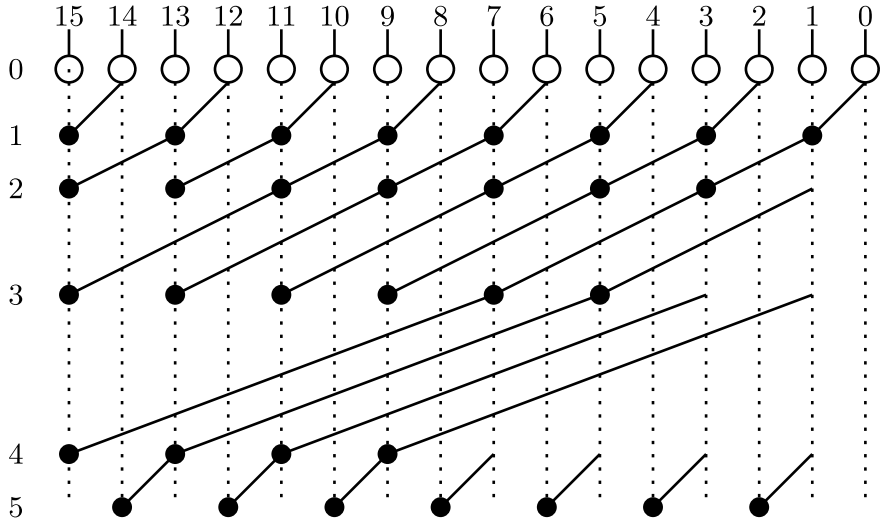


Fig. 7.9 Han–Carlson prefix tree adder data path

3. Fan-in/Fan-out—This tree adder has fan-in/fan-out value as that of Brent–Kung tree.
4. Total Nodes—This tree adder uses same number of nodes as that of Brent–Kung tree adder.

7.4 Manchester Carry Chain Module (MCC)

The Manchester carry chain is a variation of the carry look-ahead adder that uses shared logic to lower the transistor count. A Manchester carry chain generates the intermediate carries by tapping off nodes in the gate that calculates the most significant carry value. However, not all logic families have these internal nodes, CMOS being a major example. Dynamic logic can support shared logic, as can transmission gate logic. One of the major drawbacks of the Manchester carry chain is that the capacitive load of all of these outputs together with the resistance of the transistors causes the propagation delay to increase much more quickly than a regular carry look-ahead. A Manchester carry chain section generally doesn't exceed 4 bits. The Manchester carry scheme for the group of 4 is shown in Fig. 7.10.

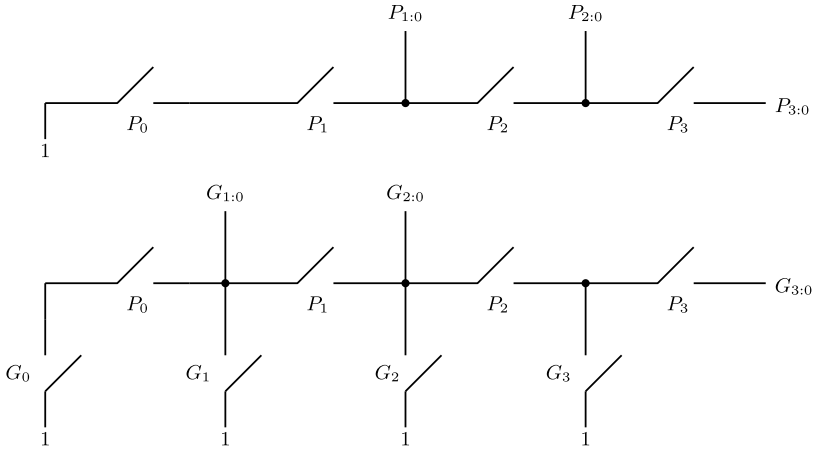


Fig. 7.10 Overall framework for CS-based RADAR signal reconstruction

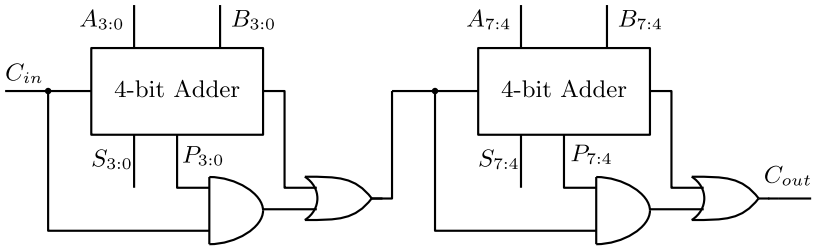


Fig. 7.11 The architecture of the carry skip adder

7.5 Carry Skip Adder

The Carry Skip Adder, also known as carry bypass adder, is based on the similar carry propagation criteria. If the propagation criterion is satisfied, the input carry is passed to the output. Thus the status of output carry is evaluated using propagation criteria. The simple carry skip adder for n = 8 is shown in Fig. 7.11. The performance of the carry skip adder scheme depends on the size of carry skip adder block size. Here 8-bit adder is implemented using the block size of 4 bit. The condition for the block size is $\sqrt{n/2}$. The variable block size is adopted to achieve fast addition.

7.6 Carry Increment Adder

The design of Carry Increment Adder (CIA) consists of an adder block (CLA, RCA) and an incremental circuitry. The incremental circuit can be designed using HA's in ripple carry chain with a sequential order. For example, the addition operation for 8-bit data can be done by dividing the total number of bits in two groups of 4 bits and addition operation is done using 4-bit RCA or CLA. This fast addition technique is not much popular since a carry chain still exists. The architecture of CIA is shown in Fig. 7.12.

7.7 Carry Select Adder

The carry select adder consists of two addition paths. One path calculates sum considering C_{in} is equal to zero and the other path calculates sum with C_{in} is equal to one. After the sums are calculated, correct sum and correct C_{out} are selected through a MUX. Thus it has two adder blocks and two MUX units. The adder block can be an RCA or a CLA. The size of the carry select block can be fixed or can be variable. For fixed size carry select adder, optimum delay occurs when block size is $\sqrt{n}$. For example, the block size is 4 for n = 16. So, there will be two 4-bit RCA/CLA in a

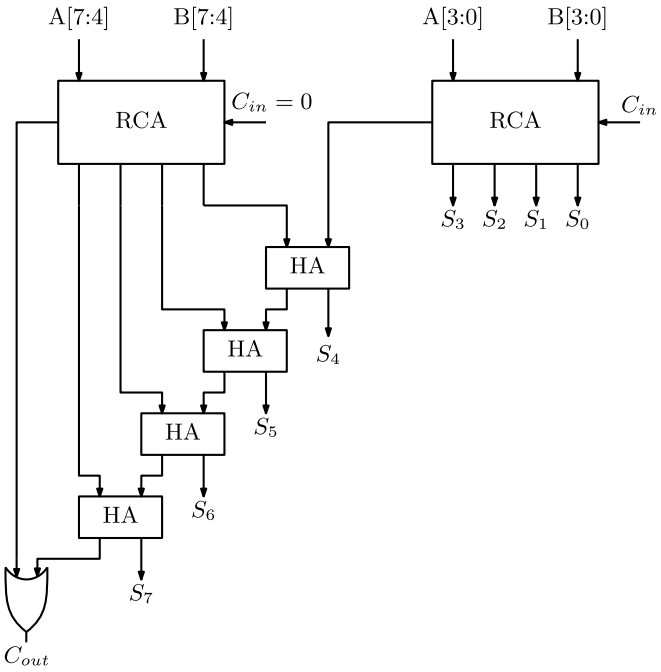


Fig. 7.12 The structure of the carry increment adder

carry select adder block. Thus carry select adders achieve fast addition by consuming more hardware. The simple block diagram of a carry select adder is shown in Fig. 7.13.

If the block size is fixed then carry select adder is called linear. For example, a 16-bit adder can be realized using block sizes 4-4-4-4. On the other hand to improve the performance of non-linear carry select adders, choose variable block sizes. For example, the same 16-bit adder can be implemented by block sizes 2-2-3-4-5.

7.8 Conditional Sum Adder

The idea of carry select adder is behind the idea of fast conditional sum adders [65]. An n-bit adder can be designed using smaller n/2 or n/4 bit adders using the same carry select concept. For example, a 4-bit adder can be built using seven 1-bit adders. This example is shown in Fig. 7.14. Conditional sum adders also provide logarithmic speedup.

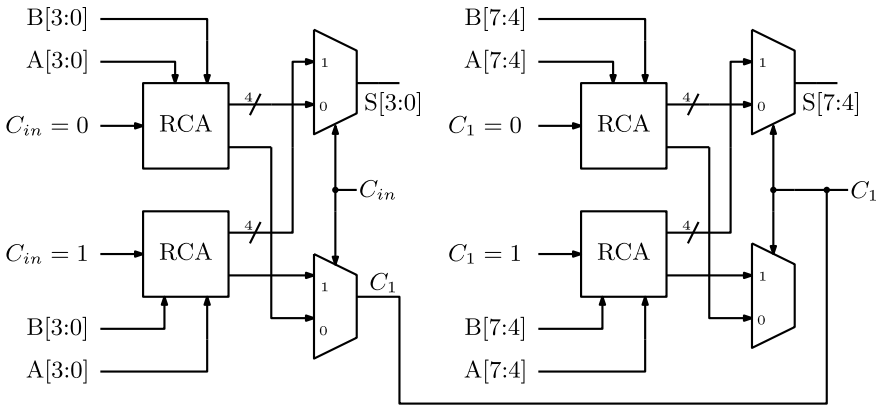


Fig. 7.13 A schematic of 4-bit carry select adder

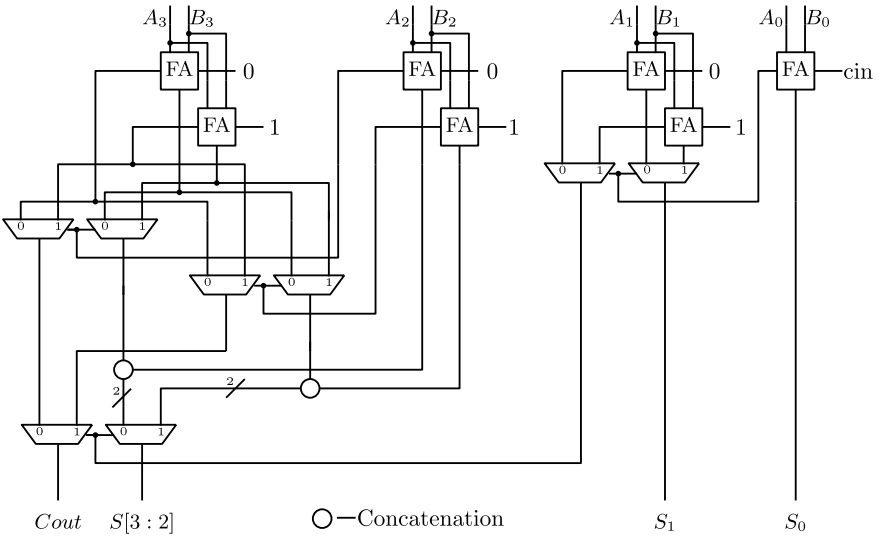


Fig. 7.14 Schematic of 4-bit conditional sum adder

7.9 Ling Adders

Ling adders [42] are variations of carry look-ahead adders. It uses the simpler equation of group generated carry and thus resulting in fast addition. The equation for the output carry of a 4-bit adder block can be expressed as

$$C_4 = G_{3:0} = G_{3:2} + P_{3:2}G_{1:0} \tag{7.20}$$

Assume, $C_0 = 0$ or $G_0 = A_0B_0 + P_0C_0$. The equation of group generated carry in a block of 4-bit adder is

$$G_{3:0} = G_3 + P_3G_2 + P_3P2(G_1 + P_1G_0) \tag{7.21}$$

Assume as $P_i = A_i + B_i$, $G_3P_3 = G3$. So, in the above equation P_3 can be taken as common factor and the equation be rewritten as

$$G_{3:0} = G_3H_{3:0} \tag{7.22}$$

where $H_{3:0}$ can be written as

$$H_{3:0} = H_{3:2} + P_{2:1}H_{1:0} \tag{7.23}$$

and

$$H_{3:2} = G_3 + G_2,\ H_{1:0} = G_1 + G_0 \tag{7.24}$$

The general expression is

$$H_{i:0} = G_1 + P_{i-1}H_{i-1:0} \tag{7.25}$$

The sum is calculated by the following equation:

$$S_i = C_i \oplus (A_i \oplus B_i) = P_{i-1}H_{i-1:0}(A_i \oplus B_i) \tag{7.26}$$

$$= \overline{H_{i-1:0}}A_iC_i \oplus (A_i \oplus B_i) + H_{i-1:0}(P_{i-1}(A_i \oplus B_i)) \tag{7.27}$$

The calculation of $H_{i-1:0}$ is faster than calculation of C_i which reduces the delay in calculating sum.

7.10 Hybrid Adders

A hybrid adder is the one which uses two or more above kind of addition techniques to implement higher order adder. Generally, a hybrid adder employs one kind of adder for generating the carry and another kind for computing sum. For example, Manchester Carry Chain (MCC) can be used for carry generation and carry select adder can be used for sum calculation.

7.11 Multi-operand Addition

Up to now, we have discussed fast addition with two operands. If several operands are to be added then these adders are needed to be used several times. Direct application of these adders for multi-operand addition can be inefficient because of multiple use of carry processing units. Many techniques are reported in literature to efficiently add multiple operands. Carry save addition is one such technique which reduces the complexity involved in multi-operand addition.

7.11.1 Carry Save Addition

Carry Save Adder (CSA) is actually a three-input adder which receives three operands and produces two outputs. For 1-bit data, CSA is actually a full adder. It is sometimes called as a 3:2 counter/compressor as it compresses three inputs to two outputs. The general equation for a CSA is given below:

$$x + y + z = 2c + s \tag{7.28}$$

where s and c are the sum and carry outputs from CSA. They are evaluated as

$$s = (x + y + z) mod 2 \tag{7.29}$$

and

$$c = \frac{(x + y + z) - s}{2} \tag{7.30}$$

A simple CSA-based addition of three operands is shown in Fig. 7.15. Here we need one stage of CSA and one Carry Propagated Adder (CPA).

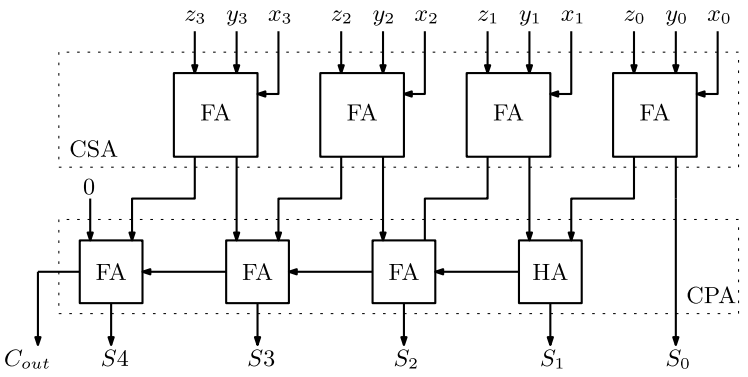


Fig. 7.15 A simple example of carry save addition for three operands

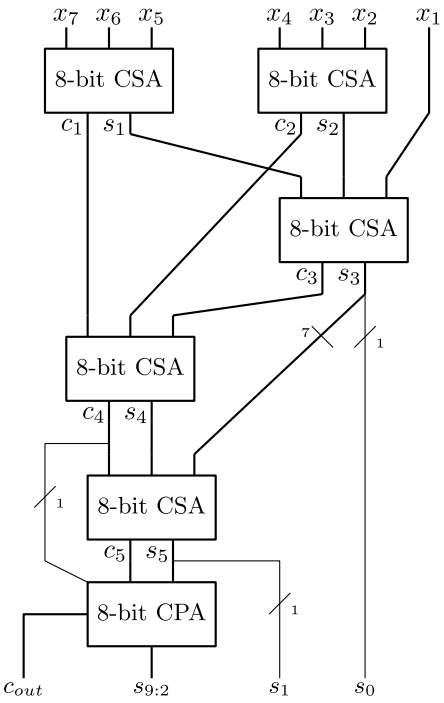


Fig. 7.16 Wallace tree of CSAs for seven-input addition

7.11.2 Tree of Carry Save Adders

For fast addition of multi-operands tree of CSAs can be formed. An example of the addition of seven 8-bit operands is shown in Fig. 7.16. This scheme to add seven operands is popularly known as Wallace tree. Five CSA modules and one CPA module are used here. The scheme of the fast addition of multiple operands is needed to be understood. A basic scheme of the addition of seven 4-bit numbers is shown in Fig. 7.17 using Wallace tree addition method. There are other types of CSA trees available for multi-operand addition. We will discuss them in more detail in the next chapter where multiplication operation is discussed.

7.12 BCD Addition

Though BCD addition does not belong to the category of fast addition, we discussed it here as it is also a type of adder. In BCD addition, inputs are in BCD format and the output is also in BCD format. In BCD format, counting is done from 0 to 9. If the summation result is greater than 9, then correction is needed. And that correction is done by adding 6 to the summation result. The equation for the correction logic is

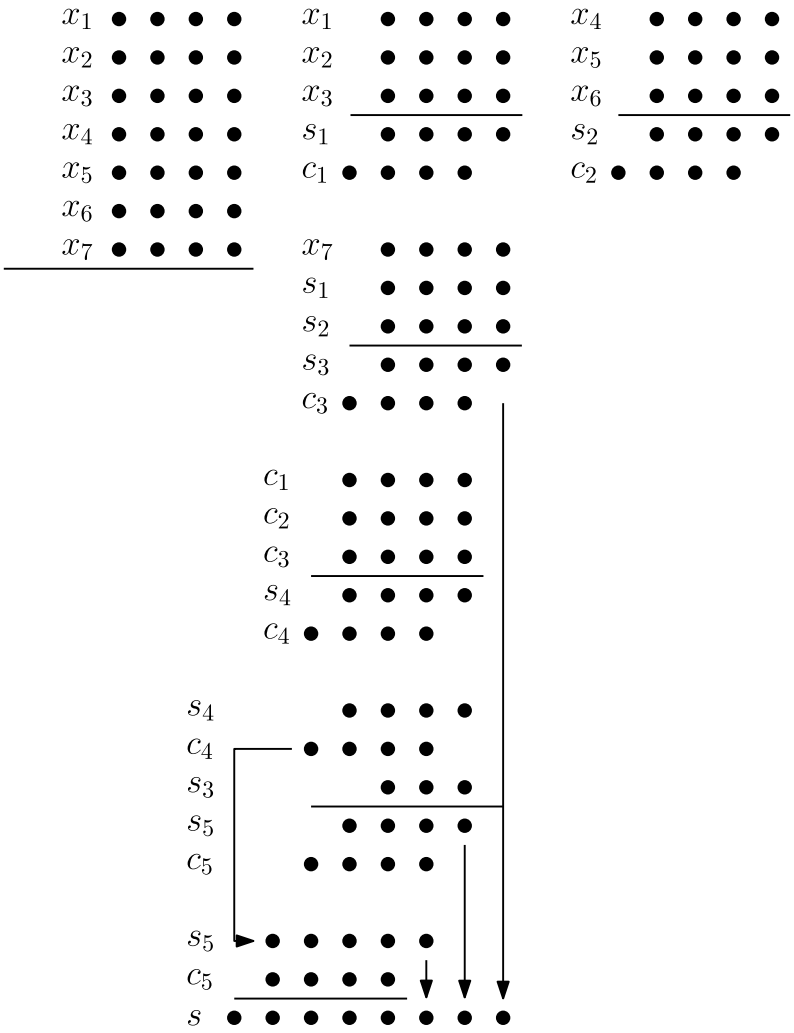


Fig. 7.17 An example of addition of seven operands using Wallace tree

$$C_c = C_3 + S_3(S_2 + S_1) \tag{7.31}$$

Here C_3 is the carryout of the last adder in the first stage. The logic diagram is shown in Fig. 7.18.

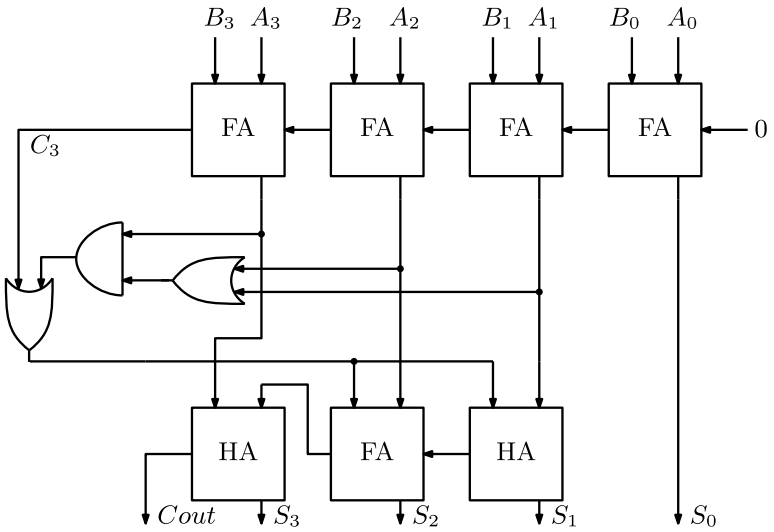


Fig. 7.18 Architecture of 4-bit BCD adder

7.13 Conclusion

We have discussed several techniques for fast addition in this chapter but a proper comparison is not shown. So that the general question arises that which adder is suitable for system design? For two operands, carry look-ahead adder performs well. For higher data width, CLA with prefix adders is a common choice. In signal processing, vector multiplication is a common operation to be performed and it involves multipliers and several multi-operand additions. Carry save adders are always suitable for these types of multi-operand addition. We will discuss further multi-operand addition techniques in Chap. 8.

Chapter 8
Design of Multiplier Circuits

8.1 Introduction

Over the past few decades, researchers have been trying to improve the architecture of a multiplier in terms of speed, power or area. This is because in almost every digital systems a multiplier is used. Also, multiplication operation is used to approximate other complex arithmetic operations like division, reciprocal, square root or square root reciprocal.

The multiplication can be achieved in three general ways. Firstly, the sequential multipliers sequentially generate the partial products and add them with the previously stored partial products. In the second method, parallel multipliers generate the partial products in parallel and add them by a multi-operand adder. The third method corresponds to use of array of identical blocks that generates and adds the partial products simultaneously.

In this chapter, various structures for multiplication operation will be discussed. A multiplier can be sequential or parallel depending on the system requirement. A multiplier can be signed or unsigned. The major objective of this chapter is to discuss about the fast multiplication techniques. But other techniques to evaluate multiplication operation are also discussed.

8.2 Sequential Multiplication

Sequential multiplier is an old method to multiply two binary numbers. But it is also relevant in many architectures and it is the base of many newly developed multiplication techniques. The multiplication between a and b is shown in Fig. 8.1. The multiplication between two operands a and b can be considered as addition of the operand a total b times. For example, $s = 5 \times 3 = 5 + 5 + 5 = 15$. Serially 5 is added total 3 times to compute the final result. Thus total one adder is sufficient. For

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_8

Fig. 8.1 Implementation of 8-bit multiplier using four 4-bit multipliers

$$
\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & & \times & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & p_{30} & p_{20} & p_{10} & p_{00} \\
 & & & p_{31} & p_{21} & p_{11} & p_{01} & \times \\
 & & p_{32} & p_{22} & p_{12} & p_{02} & \times & \times \\
 & p_{33} & p_{23} & p_{13} & p_{03} & \times & \times & \times \\
\hline
s_7 & s_6 & s_5 & s_4 & s_3 & s_2 & s_1 & s_0
\end{array}
$$

a word length of 4 bits, width of the multiplication result is 8 bit. So, an 8-bit adder is required.

An alternative method is shift and add method. If any bit in the multiplier (b) is 0 then the multiplicand (a) is added with zero. An adder is used which is of the same length as of the operands. Output of the adder and the multiplier is augmented in a register bank. After each addition, contents of the register bank are shifted right. A scheme of serial addition is shown in Fig. 8.2. The *start* signal starts the multiplication process. It loads a in a register and also loads b in another register. Each D flip-flop is controlled by a control signal. The DFFs shift data to the right only when the control signal is high. The counter tracks the latency of the multiplier. The PG block is there to generate the enable signal (en) for the counter and the bottom register. The *start* pulse generates the *en* signal.

8.3 Array Multipliers

In the previous section, a scheme of sequential multiplier is discussed. Sequential multipliers may be useful when we are concerned about hardware not the speed. But in majority of digital systems, parallel multipliers are used. The first and most popular parallel multiplication method is array multiplication. Array multiplication for both unsigned and signed numbers is discussed here.

Array multiplier resembles the pen-and-paper method of multiplication process which is shown in Fig. 8.1. An array of full adders is used for the unsigned multiplication process. For n-bit data width, total $n(n-1)$ full adders are used in this multiplier. Carry outputs of a stage are added in the next stage to form a systolic architecture. But in the last stage carry is used in the same stage to reduce hardware. The architecture of the array multiplier is shown in Fig. 8.3. At the first stage, the FAs can be replaced with HAs but here FA blocks are retained.

The signed multiplication is little bit complicated than the unsigned array multiplication. In case of signed multiplication where the operands are represented in two's complement representation, instead of adding the product bits $a_3.b_0$ or $a_0.b_3$ should be subtracted. This is because the MSB bit in 4-bit operand is signed bit. Array multipliers are designed to handle such bits and a Sub-block (SB) is designed. The basic operation of this block is

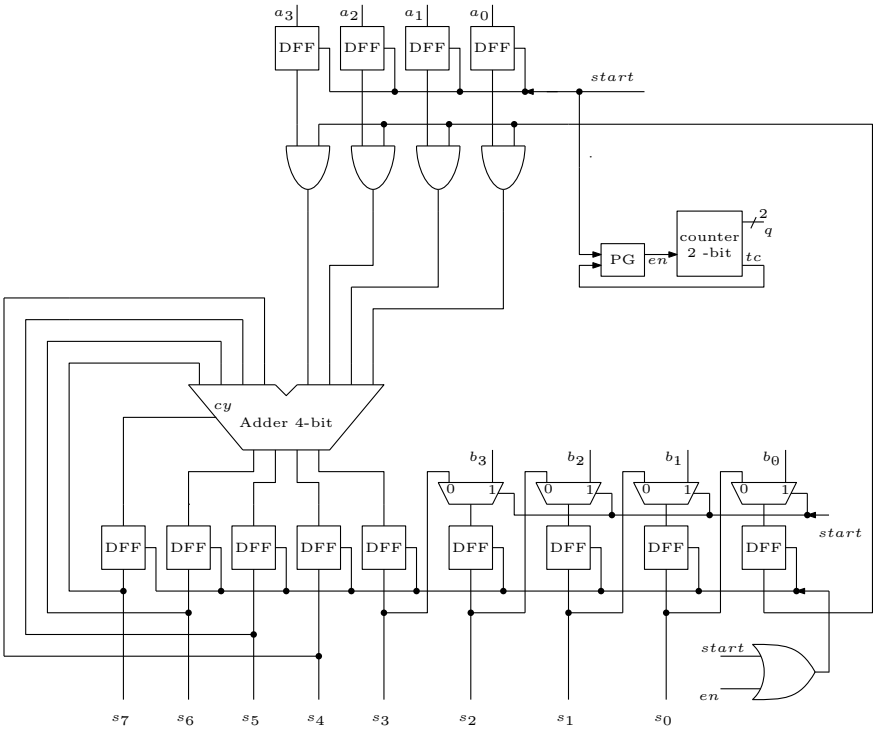


Fig. 8.2 4-bit sequential multiplier

$$s = (x + y - z) mod\ 2 \ \text{ and } \ c = ((x + y - z) + s)/2 \tag{8.1}$$

The Boolean expressions become upon simplifying using K-map as

$$s = x \oplus y \oplus z \ \text{ and } \ c = xy + x\bar{z} + y\bar{z} \tag{8.2}$$

The expressions are similar to that of the FA but negative bit is inverted in the computation of carry. The signed array structure is shown in Fig. 8.4. Here, three type of SB blocks are shown but their logic expressions are same. In case of SB, z is the negative bit. But in case of SB1 and SB2, x and y are the negative bits.

The fast multiplication can be achieved in three general ways.

1. Fast multiplication can be achieved by increasing the clock frequency for sequential or array multipliers.
2. The second method corresponds to use of fast multi-operand adder to add the partial products generated by high-speed parallel multipliers.
3. Other way is to reduce the partial products which are to be added to generate the final product.

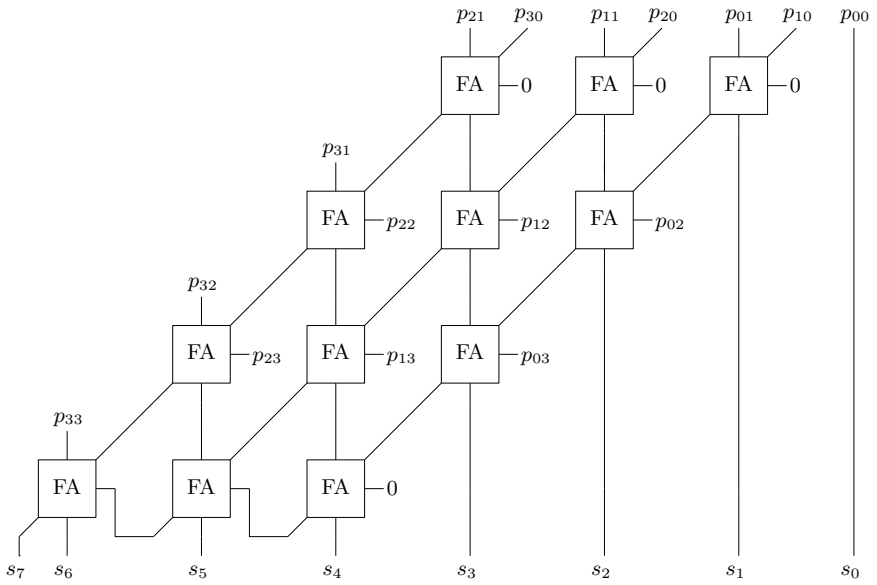


Fig. 8.3 Architecture of 4-bit unsigned array multiplier

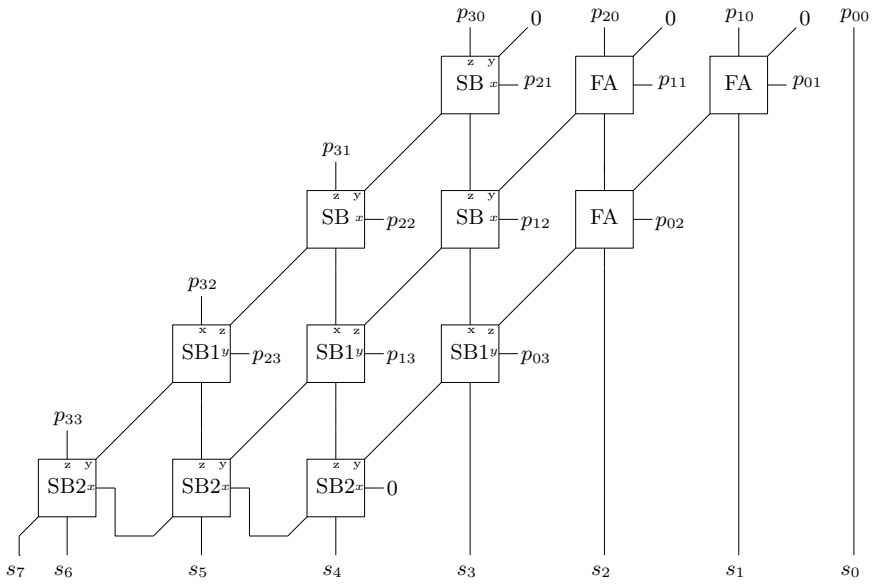


Fig. 8.4 Architecture of 4-bit signed array multiplier

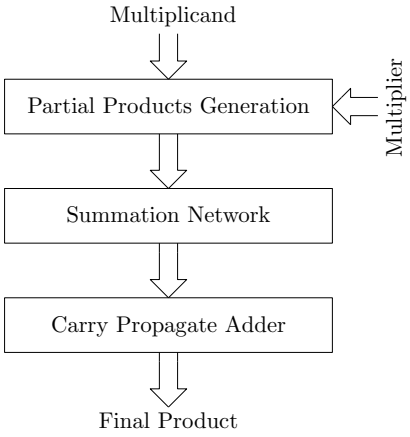


Fig. 8.5 Framework for fast multiplication techniques

The increase in frequency does not make sense as it will increase power consumption. The performance of the parallel multipliers depends on the performance of multi-operand adder and also on the optimal number of partial products. An obvious method to design a fast multiplier is to reduce the partial products and then apply fast addition methods. The framework for fast multiplication techniques is shown in Fig. 8.5. In this section, some techniques for fast multiplication are discussed.

8.4 Partial Product Generation and Reduction

8.4.1 Booth's Multiplication

For consecutive zeros, a multiplier only needs to shift the accumulated result to the right without generating any partial products. For example, the accumulated result is shifted 1 bit right for every '0' in the multiplier. This principle can be explained by the help of the following example.

Consider multiplication of two 8-bit numbers where A is the multiplicand and the multiplier X takes the value as 00111100. Here X has two repetitive zeros in the left and in the right. The multiplier X has also four repetitive ones in the middle. In the general multiplication scheme, a multiplier will need four partial products and each partial product has eight product bits. Totally eight shift operations are required. The repetitive zeros can be dealt with by only shifting the accumulated result. To deal with the repetitive ones, the above multiplication can be written as

$$A * (00\{1111\}00) = A * (01\{0000\}00 - 00\{0001\}00) = A * (0100\bar{0}\bar{1}00) \quad (8.3)$$

Table 8.1 Booth's Radix-2 recoding method

x_i	x_{i-1}	Operation	Comments	y_i
0	0	Shift only	String of zeros	0
1	1	Shift only	String of ones	0
1	0	Subtract and shift	Beginning of string of ones	$\bar{1}$
0	1	Add and shift	End of string of ones	1

The multiplicand X is written as $01000\bar{1}00$. This is the Signed Decimal (SD) representation where $\bar{1}$ represents the -1. The partial product $-A$ is added to the accumulated result due to the presence of $\bar{1}$ in the newly modified X. Thus in this case, instead of four partial products only two partial products are needed. The number of shifting operations remains same.

The above-mentioned technique is called Booth's recoding of the multiplier [15] in SD form. In this technique, current bit x_i and the previous bit x_{i-1} of the multiplier $x_{n-1}x_{n-2}..x_1x_0$ are checked to generate the current bit y_i of the recoded multiplier $y_{n-1}y_{n-2}..y_1y_0$. A simple way of recoding is by the equation $y_i = x_{i-1} - x_i$. This technique of recoding is also called as Booth's Radix-2 recoding method. Recoding need not to be done in any pre-defined manner and can be done in parallel from any bit positions. The simplest recoding scheme is shown in Table 8.1.

An example of multiplication using Booth's radix-2 algorithm is shown in Table 8.2 for two 4-bit signed operands. Here recoding is started from the LSB. The computation of Y is not necessary as it involves extra hardware. Instead the adder and subtractor blocks are controlled accordingly. There are two drawbacks of this Booth's algorithm. First, the number add/sub operations is not fixed and also the number of shift operations between two add/sub operations is not fixed. The second is that the algorithm is not efficient when there is isolated ones. For example, 001010101(0) is recoded as $01\bar{1}1\bar{1}1\bar{1}1\bar{1}$ which increases the add/sub operations instead of reducing it.

8.4.2 Radix-4 Booth's Algorithm

The disadvantages of the Radix-2 algorithm are addressed by the Radix-4 Booth's algorithm [45]. Here 3 bits are examined instead of 2 bits. The bits x_i and x_{i-1} are recoded into y_i and y_{i-1} while x_{i-2} act as reference bit. The variable i takes the value from the set $\{1, 3, 5....\}$. The recoding of the multiplier can be done easily by the following equation:

$$y_i y_{i-1} = x_{i-1} + x_{i-2} - 2x_i \tag{8.4}$$

Table 8.2 An example for Booth's Radix-2 algorithm

A		1	0	1	0				–6
X	×	1	1	0	1				–3
Y		0	$\bar{1}$	1	$\bar{1}$				Recoded multiplier
Add -A		0	1	1	0				
Shift		0	0	1	1	0			
Add A		1	0	1	0				
		1	1	0	1	0			
Shift		1	1	1	0	1	0		
Add -A		0	1	1	0				
		0	1	0	0	1	0		
Shift		0	0	1	0	0	1	0	

Table 8.3 Booth's Radix-4 algorithm multiplier recoding scheme

x_i	x_{i-1}	x_{i-2}	y_i	y_{i-1}	Operation	Comments
0	0	0	0	0	+0	String of zeros
0	1	0	0	1	+A	A single one
1	0	0	$\bar{1}$	0	–2A	Beginning of ones
1	1	0	0	$\bar{1}$	–A	Beginning of ones
0	0	1	0	1	+A	End of ones
0	1	1	1	0	+2A	End of ones
1	0	1	0	$\bar{1}$	–A	A single zero
1	1	1	0	0	+0	String of ones

The scheme of recoding of the multiplier in Booth's Radix-4 algorithm is shown in Table 8.3. The Radix-4 algorithm efficiently overcomes all the limitations of the Radix-2 recoding algorithm. An example of multiplication using Radix-4 recoding algorithm is shown in Table 8.4. In this multiplication process, totally three add/sub operations are performed. Hence the Radix-4 algorithm takes totally $n/2$ add/sub operations. In each operation, 2 bits are dealt with and shifting operation is of 2 bits.

Booth Array Multiplier

An array multiplier using the Booth radix-4 algorithm can be designed. Booth's Radix-4 algorithm works on the principle of selecting partial products from the set $\{A, 2A, 0, -A, -2A\}$. The partial product $2A$ can be easily obtained by wired

Table 8.4 An example for Booth's Radix-4 algorithm

A			01	00	01				17
X	×		11	01	11				–9
Y			$0\bar{1}$	10	$0\bar{1}$				Recoded multiplier
			–A	+2A	–A				Operation
Add –A	+		10	11	11				
2-bit shift		1	11	10	11	11			
Add 2A	+	0	10	00	10				
			01	11	01	11			
2-bit shift			00	01	11	01	11		
Add –A	+		10	11	11				
			11	01	10	01	11		–153

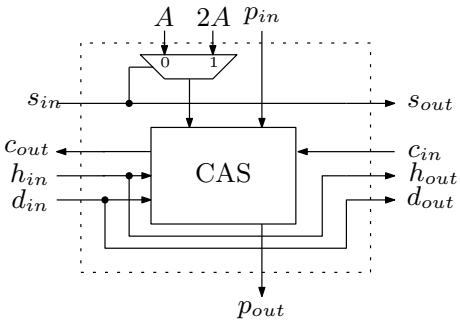


Fig. 8.6 The basic Sub-block (SB) structure

shifting method. Thus we need a selection procedure to select between A and $2A$. The block diagram for the basic Sub-block (SB) is shown in Fig. 8.6. Here the input s selects between A and $2A$. The major block of the SB is the controlled adder and subtractor (CAS) block [46]. The Boolean expression for the CAS block is

$$p_{out} = p_{in} \oplus (a.h) \oplus (c_{in}.h) \tag{8.5}$$

$$c_{out} = (p_{in} \oplus d).(a + c_{in}) + a.c_{in} \tag{8.6}$$

If $h = 1$, arithmetic operation is performed, otherwise input is passed to the output. If $d = 1$, subtraction operation is performed and input a is subtracted from p_{in}. Then c_{in} is the incoming borrow and c_{out} is the outgoing borrow. If $d = 0$, addition operation is performed and input a is added with p_{in}.

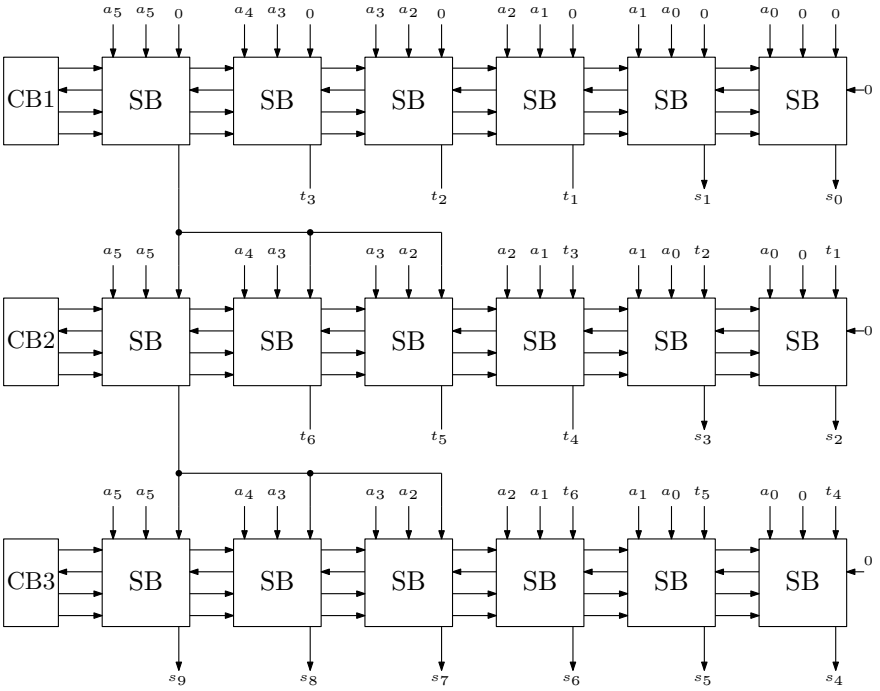


Fig. 8.7 Architecture of the Booth array multiplier

The overall Booth's array multiplier for 6 bits is shown in Fig. 8.7. This multiplier is capable of performing sign multiplication without any extra hardware. The control signals are generated from the Control Blocks (CB). Totally 18 SBs are used in this design. The generation of the control signals is important. Three control signals are used in this design which are s, h and d. The truth table for these control signals according to the input data x is shown in Table 8.5. Boolean expressions for the control signals can be obtained by applying K-map. The logical expressions are obtained as

$$h_i = x_i.\overline{x_{i-1}} + \overline{x_i}.x_{i-2} + x_{i-1}.\overline{x_{i-2}} \tag{8.7}$$

$$s_i = x_{i-1}.x_{i-2} + \overline{x_{i-1}}.\overline{x_{i-2}} \tag{8.8}$$

$$d_i = x_i \tag{8.9}$$

Here i took the value from the set {1, 3, 5, ...}.

Table 8.5 Truth table for control signals for Booth array multiplier

x_i	x_{i-1}	x_{i-2}	h	s	d	Operation
0	0	0	0	×	×	+0
0	0	1	1	0	0	+A
0	1	0	1	0	0	+A
0	1	1	1	1	0	+2A
1	0	0	1	1	1	–2A
1	0	1	1	0	1	–A
1	1	0	1	0	1	–A
1	1	1	0	×	×	+0

8.4.3 Canonical Recoding

The number of add/sub operations in a multiplier depends on the optimum SD representation of the multiplier (X). In other way, the number of nonzero elements in Y decides the number of add/sub operations. The SD representation of the multiplier in Booth's Radix-2 and Radix-4 algorithm is not optimum. Canonical recoding algorithm is a technique which obtains an optimum representation of a multiplier. Canonical recoding algorithm operates on a multiplier from right to left on 1 bit a time. Here x_{i+1} serves as a reference bit. A multiplier (X) represented in two's complement form is treated as $x_{n-1}x_{n-1}x_{n-2}....x_2x_1x_0$ to obtain optimum SD representation.

Unlike the Radix-2 and Radix-4 algorithms, Canonical recoding algorithm includes carry input to obtain SD representation. Here c_i is the carry input and c_{i+1} is the carry output. The different rules of obtaining optimum representation are shown in Table 8.6. The SD representation of the multiplier $X = 01101110$ in Radix-2 algorithm is $Y = 10\bar{1}10\bar{1}0$. The optimum SD representation of this multiplier in Canonical recoding algorithm is $Y = 100\bar{1}00\bar{1}0$.

There are mainly two disadvantages of Canonical recoding algorithm. Firstly the bits of the recoded multiplier are obtained sequentially as it involves carry generation and propagation. The second disadvantage is same as it is for Radix-2 algorithm, that is, optimum SD representation corresponds to variation in number of add/sub operations.

8.4.4 An Alternate 2-bit at-a-time Multiplication Algorithm

An alternative algorithm exists to reduce the partial products by involving fixed number of add/sub operations. This algorithm is similar to the Radix-4 algorithm and operates on 2 bits at a time. In this algorithm, the x_{i+1} is considered as reference bit and the recoded multiplier Y can be computed in parallel. Total number of add/sub

Table 8.6 Canonical recoding rules

x_{i+1}	x_i	c_i	y_i	c_{i+1}	Comments
0	0	0	0	0	String of zeros
0	1	0	1	0	A single one
1	0	0	0	0	String of zeros
1	1	0	$\bar{1}$	1	Beginning of ones
0	0	1	1	0	End of ones
0	1	1	0	1	String of ones
1	0	1	$\bar{1}$	1	A single zero
1	1	1	0	1	String of ones

Table 8.7 Rules for recoding in alternative 2-bit at a time multiplication algorithm

x_{i+1}	x_i	x_{i-1}	Operation	Comments
0	0	0	+0	String of zeros
0	0	1	2A	End of ones
0	1	0	+2A	A single one
0	1	1	+4A	End of ones
1	0	0	–4A	Beginning of ones
1	0	1	–2A	A single zero
1	1	0	–2A	Beginning of ones
1	1	1	+0	String of ones

Table 8.8 Correction step for the alternative to Radix-4 multiplication algorithm

x_2	x_1	x_0	Operation
0	0	1	+2A – A = A
0	1	1	+4A – A = 3A
1	0	1	–2A – A = –3A
1	1	1	0 – A = A

operations is always $n/2$. The rules for this algorithm are shown in Table 8.7. Here the multiple of A can be obtained easily by wired shifting.

In this algorithm as the reference bit is x_{i+1}, it ignores if there is a start of string of ones in the rightmost pair x_1x_0. A correction step is needed in this algorithm. This is described in Table 8.8. This alternative algorithm provides easy recoding rules compared to the original Radix-4 algorithm but it has a correction step. Initially, there is decision to make to select between A and 3A. The computation of 3A involves an extra add/sub operation.

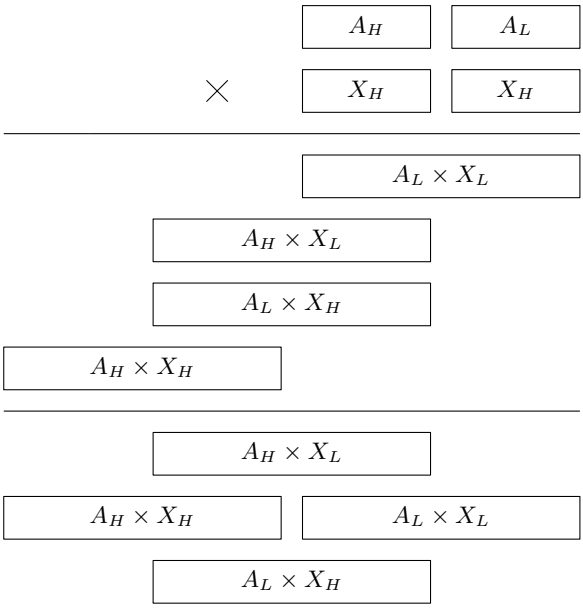


Fig. 8.8 Implementation of 8-bit multiplier using four 4-bit multipliers

8.4.5 Implementing Larger Multipliers Using Smaller Ones

The larger multiplier blocks can be realized using smaller multiplier blocks. A $2n \times 2n$ multiplier can be realized using four $n \times n$ multiplier blocks. This is based on the following equation:

$$A.X = (A_H.2^n + A_L).(X_H.2^n + X_L) = A_H.X_H.2^{2n} + (A_H.X_L + A_L.X_H).2^n + (A_L.X_L) \tag{8.10}$$

where A_H is the most significant half of A, X_H is the most significant half of X, A_L is the least significant half of A and X_L is the least significant half of X.

The partial products from the smaller multiplier blocks should be correctly arranged and accumulated by fast multi-operand adders. A scheme of implementing a 8-bit multiplier using four 4×4 multipliers is shown in Fig. 8.8.

8.5 Accumulation of Partial Products

The optimum number of partial products can be obtained by applying any of the algorithms mentioned in Sect. 8.4. The partial products for a larger multiplier can also be obtained from the smaller multiplier blocks. The next job in designing a fast multiplier is to accumulate these partial products by a fast accumulating circuit. In

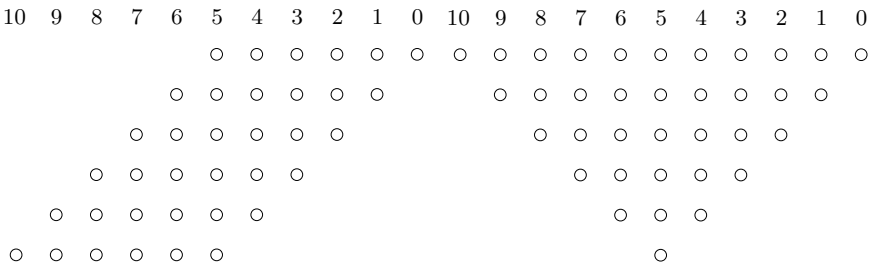
this section, all the techniques involved in fast accumulation of partial products are discussed.

8.5.1 *Accumulation of Partial Products for Unsigned Numbers*

A basic circuit to accumulate two single bits is a half-adder (HA) and a full adder (FA) accumulates 3 bits. Thus a HA is called as 2-2 counter and a FA is called a 3-2 counter. Again a HA is hardware efficient than a FA. Carry save adders (CSA) operates on multiple operands. A basic CSA is equivalent to an FA block. Similarly, several other counters also exist which can be applied to the design of fast accumulating circuit. The major objective is to reduce the number of basic counters to reduce hardware complexity. Thus suitable arrangement of the partial products is important. An example of the partial products for a 6×6 multiplier is shown in Fig. 8.9a. Figure 8.9b shows that the partial products can be reorganized to reduce the number of counters.

Once the partial products are reorganized, carry save operation can be performed. Basic 2-2 counters or 3-2 counters are applied wherever possible. It can be seen from Fig. 8.10a that at level 1, 3 HAs and 8 FAs are used. The results of level one are shown in Fig. 8.10b. At level 2, 3 HAs and 4 FAs are sufficient as shown in Fig. 8.10c. Similarly the last carry save addition is performed at level 3 as shown in Fig. 8.10d. A carry propagation adder (CPA) is needed at the final stage to obtain the final result. This technique reduces the level of addition from six to four compared to array multiplication scheme and also reduces the number of counters (HAs and FAs).

The number of counters can be further reduced by employing the idea of reducing number of bits in each column to closest element from the set {3, 4, 6, 9, 13, 19....}. This idea is illustrated for the same example in Fig. 8.11a–d. Total number of 5 HAs and 15 FAs are used in this technique whereas totally 9 HAs and 16 FAs are used



(a) Original form of the partial products (b) Rearrangement of the partial products

Fig. 8.9 Partial products for a 6-bit multiplier

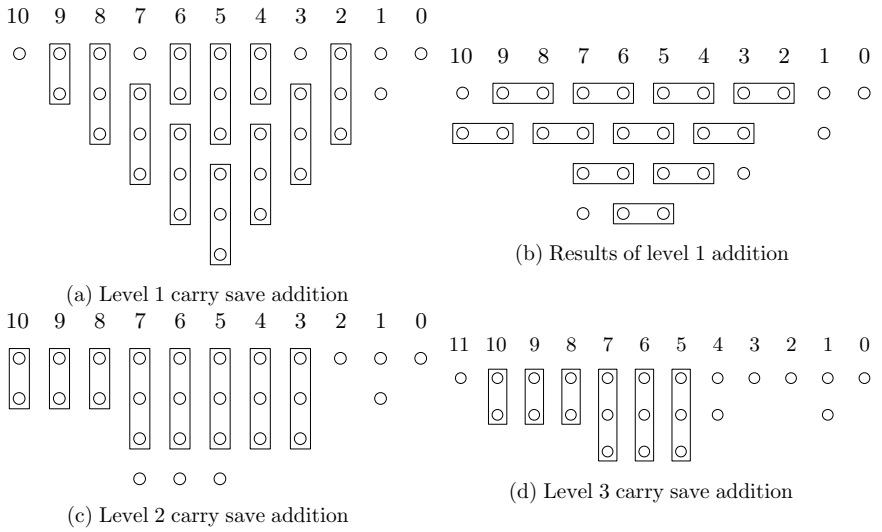


Fig. 8.10 Partial products for a 6-bit multiplier

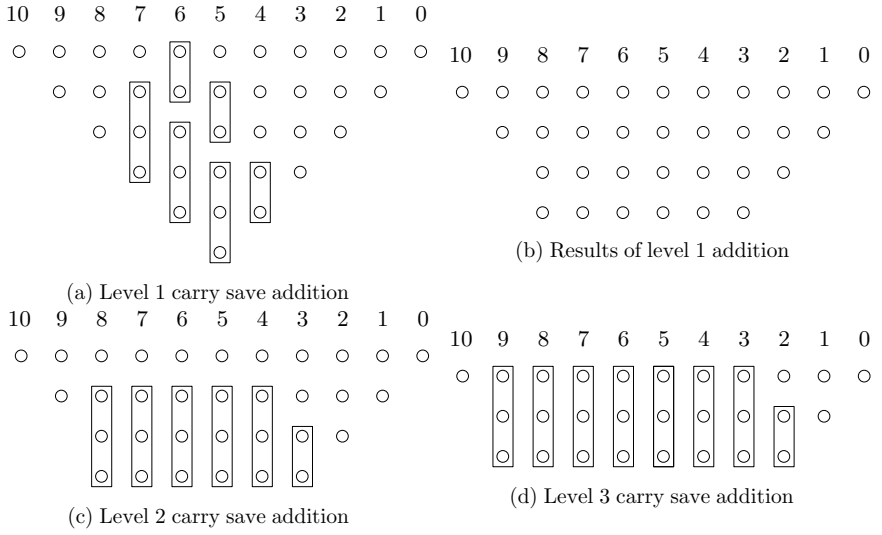


Fig. 8.11 Partial products for a 6-bit multiplier

in the previous scheme mentioned above. The savings of counter are substantial for higher bit multipliers.

8.5.2 Accumulation of Partial Products for Signed Numbers

Earlier we have discussed how the partial products for an unsigned multiplier can be accumulated using suitable organization and consuming minimum number of counters. In this section, accumulation will be done by considering negative partial products. If some of the partial products are negative numbers represented in two's complement number system, then matrix of bits needs to be changed. All the sign bits must be properly extended before addition. The extension of sign bit is shown in Fig. 8.12 for 6-bit partial products. Here the filled circles are representing sign bits. This modification increases the hardware complexity and also the number of stages. If the two's complement numbers are obtained by generating one's complement then a carry must be added in the least significant bit. This will again increase the hardware complexity.

A 6-bit partial product $z_5z_4z_3z_2z_1z_0$ represented in two's complement can be represented using 11 bit as

$$s\,s\,s\,s\,s\,s\,z_4z_3z_2z_1z_0$$

whose value is

$$-s.2^{10}+s.2^9+s.2^8+s.2^7+s.2^6+s.2^5+z_4.2^4+z_3.2^3+z_2.2^2+z_1.2^1+z_0.2^0 \tag{8.11}$$

can be replaced as

$$00000\,(-s)\,z_4z_3z_2z_1z_0 \tag{8.12}$$

since

$$-s.2^{10}+s.(2^9+2^8+2^7+2^6+2^5)=-s.2^{10}+s.(2^{10}-2^5)=-s.2^5 \tag{8.13}$$

To represent the value $-s$ in column 5, the original sign digit s is complemented to obtain $(1-s)$ and 1 is added. This way we get $-s$ in the column 6 along with a carry of 1. This carry serves as the extra 1 to deal with the sign of the second partial product. This way sign bit of all the partial products are dealt with. This solution is

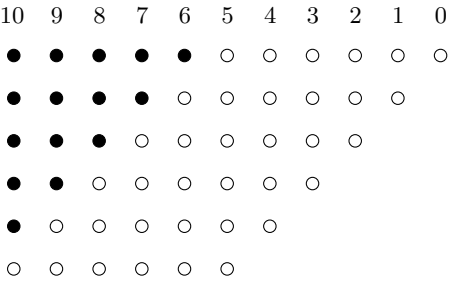


Fig. 8.12 Extension of sign bit in case of signed partial products

Fig. 8.13 The modified array of signed partial products

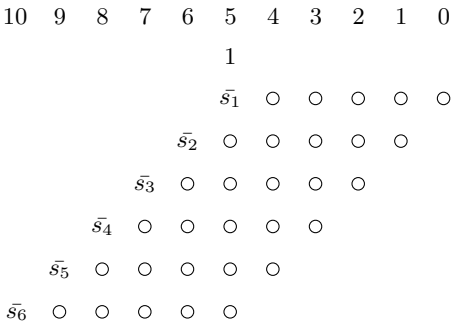
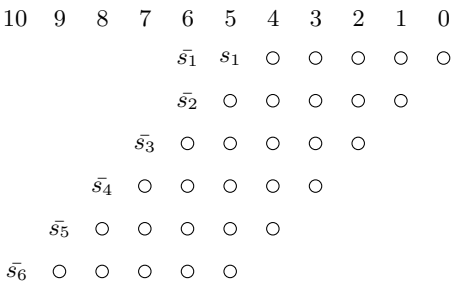


Fig. 8.14 Further modified array of signed partial products



shown in Fig. 8.13. Here number of bits compared to the array in Fig. 8.12 is reduced but the height is increased.

The disadvantage of the first solution is that its length is 7. Now, the 1 in the column 5 can be eliminated if the two sign bits s_1 and s_2 can be placed in the same column. This is possible as $(1 - s_1) + (1 - s_2) = 2 - s_1 - s_2$. This 2 is carried out to the next column leaving $-s_1$ and $-s_2$. The extra 1 in column 5 is no longer required. Placing the two sign bits in the same column is achieved by extending the sign bit s_1 bit in one position as shown in Fig. 8.14.

If the negative partial products are obtained by first generating the one's complement and then adding a carry at the least significant side then the arrangement can be made differently. The extra carry at the LSB side then must be added to the matrix. This solution is shown in Fig. 8.15 where the filled circles represent the complements of the bits whenever $s_i = 1$. Here in this solution the height of the matrix is again 7 but for the unsigned case the last carry at the LSB side can be omitted. The accumulation of signed partial products can be explained using an example. Let us consider the multiplication of two 6-bit numbers using Booth's Radix-4 algorithm as shown in Table 8.9. Here two partial products are negatively represented in two's complement format. The above techniques can be applied to decrease the number of operands. The general technique to reduce the operands in case of Radix-4 Booth algorithm for signed partial products is shown in Fig. 8.16. The matrix of the operand bits of Table 8.9 is modified by applying the second technique as shown in Table 8.10.

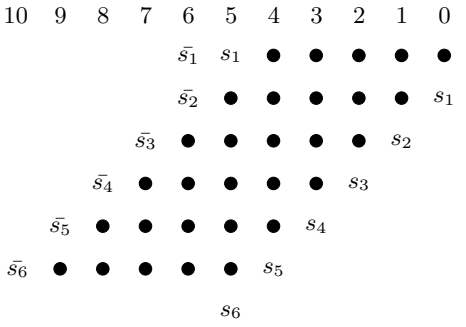


Fig. 8.15 Modified array of signed partial products represented in one's complement

Fig. 8.16 General operand reduction scheme for signed partial products for Radix-4 algorithm

Table 8.9 An example of sign extension for Booth's Radix-4 algorithm

A				00	01	11	7
X	×			11	01	11	–9
Y				$0\bar{1}$	10	$0\bar{1}$	recoded multiplier
				–A	+2A	–A	operation
Add –A	+	11	11	11	10	01	
Add 2A	+	00	00	11	10	××	
Add –A	+	11	10	01	××	××	
		11	11	00	00	01	–63

Table 8.10 An example of operand reduction for signed partial products for Booth's Radix-4 algorithm

A						00	01	11	7
X	×					11	01	11	–9
Y						$0\bar{1}$	10	$0\bar{1}$	recoded multiplier
						–A	+2A	–A	operation
Add –A	+			0	11	11	10	01	
Add 2A	+		1	1	00	11	10	××	
Add –A	+	0	1	1	10	01	××	××	
		1	1	1	11	00	00	01	–63

8.5.3 Alternative Techniques for Partial Product Accumulation

Several techniques are suggested for partial products accumulation. Some of them target to reduce the logic elements to reduce hardware complexity whereas some of them target to reduce numbers of levels in the tree of partial products to achieve high speed. As the number of levels increases irregularity in the design also increases. The irregularities in the design create problem in generating area-efficient layout. Thus some techniques tried to achieve more modular architectures for easy implementation.

Previously efficient accumulation of partial products is achieved by performing carry save addition using 2-2 or 3-2 counters. More reduction of levels is possible using compressors such as 4-2 compressor and 7-2 compressor. A basic 4-2 compressor operates on four operands and produces two results (c and s). The advantage of using a compressor is that c_{out} is not a function of c_{in} so that ripple carry effect is eliminated. Thus compressor has lower overall delay. A simple design of 4-2 compressor using 3-2 counters is shown in Fig. 8.17. A 4-2 compressor can be designed as a multilevel circuit as shown in Fig. 8.18. This type of design achieves lesser delay compared to the compressor circuit using 3-2 counters.

The delay of the compressor circuit using 3-2 counter is of maximum four XOR gates whereas there are three XOR gates in the critical path of the compressor circuit of Fig. 8.18. Many realizations of compressor are possible but all the compressor circuits should follow the following equation:

$$x_1 + x_2 + x_3 + x_4 + c_{in} = s + 2(c + c_{out}) \tag{8.14}$$

and c_{out} should not depend on c_{in} to avoid rippling of carry signal. The use of compressors reduces the number of levels in the accumulation process. Also, the delay of a 4-2 compressor is 1.5 times that of a 3-2 counter. The accumulation of partial

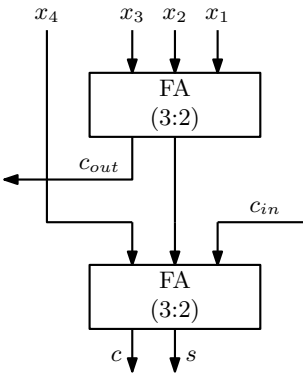


Fig. 8.17 A basic 4-2 compressor using 3-2 counters

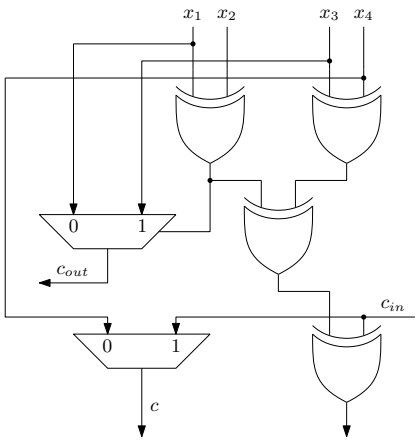


Fig. 8.18 A efficient implementation of 4-2 compressor

products using compressor circuits is supposed to be faster than the accumulation using 3-2 counters. But this is not always true in all cases.

Several techniques are suggested to improve the performance of the CSA-based accumulation of partial products using 3-2 counters. The objective is to either reduce the overall delay by reducing number of levels or to obtain a more regular structure. Three such techniques are shown in this chapter.

1. Firstly the Wallace tree structure is shown in Fig. 8.19 provides regular structure. It uses total 16 CSA blocks and it has six levels. In the layout perspective, the Wallace tree uses six wiring tracks between adjacent bit slices.
2. Researches suggested overturned-stair trees [53] shown in Fig. 8.20 to reduce the wiring tracks from 7 to 3. Overturned-stair trees are more regular, use same number of CSA modules and have same number of levels.
3. Further a balance tree structure[84] is suggested as shown in Fig. 8.21. The balance tree structures have highest delay due to the presence of seven levels but require only two wiring tracks.

Similarly a compressor tree is shown in Fig. 8.22. The compressor tree needs only 3 levels to operate on the 20 operands whereas CSA-based tree takes 6 levels. Thus it can be said that compressor tree provides low overall delay for higher number of operands.

8.6 Wallace and Dedda Multiplier Design

Chris Wallace in 1964 gave some suggestions on fast multiplication [76]. In order to achieve high speed without consuming extra hardware, he suggested some techniques. He proposes to use the basic HAs and FAs as basic elements for operand reduction. He applied operand reduction in parallel layers and achieves better speed. The steps are

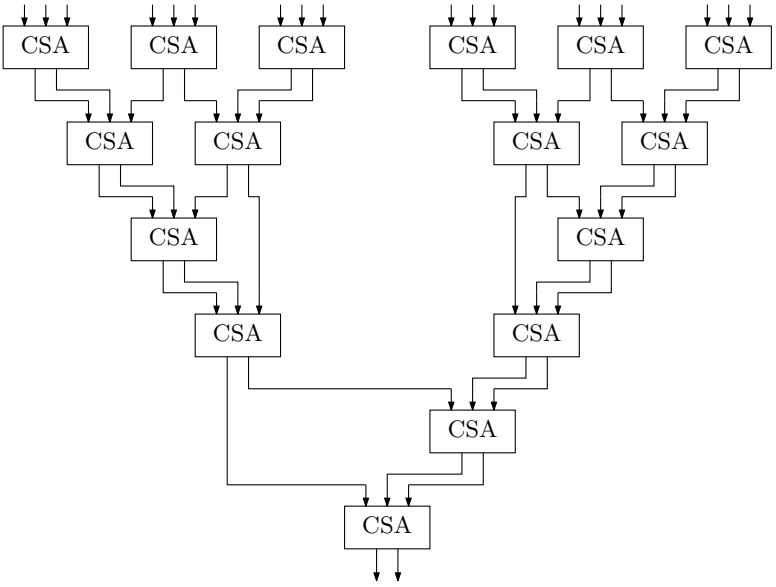


Fig. 8.19 Wallace tree structure

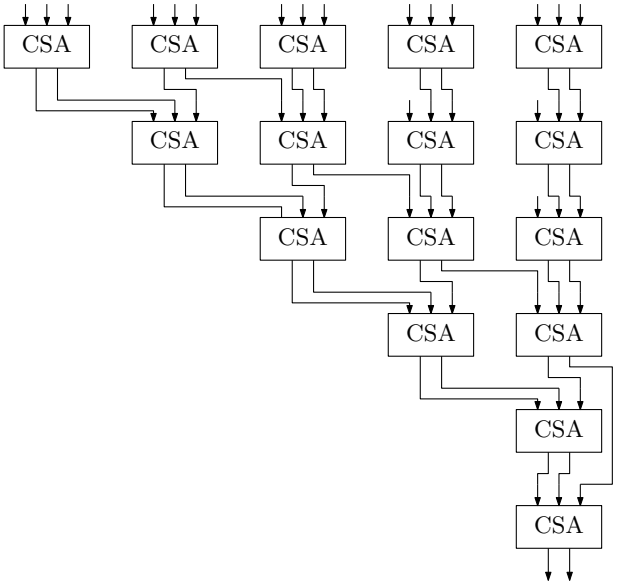


Fig. 8.20 Overturned-stair tree structure

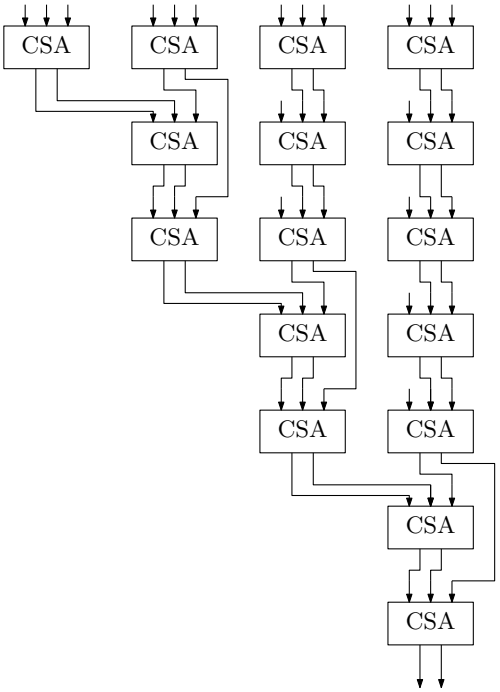


Fig. 8.21 Balance tree structure

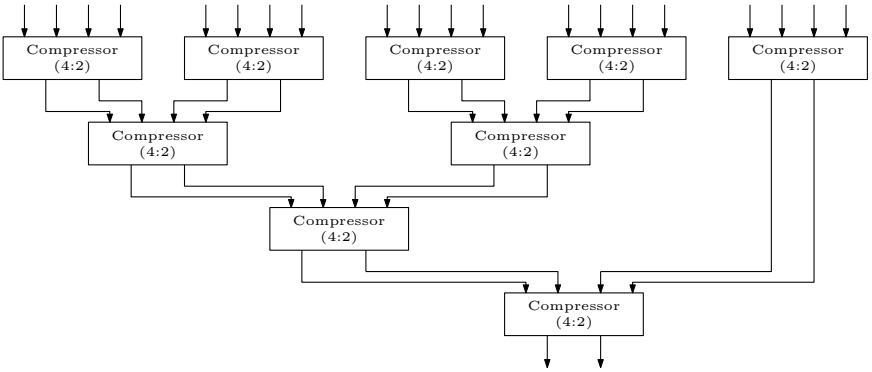


Fig. 8.22 A compressor tree for 20 operands

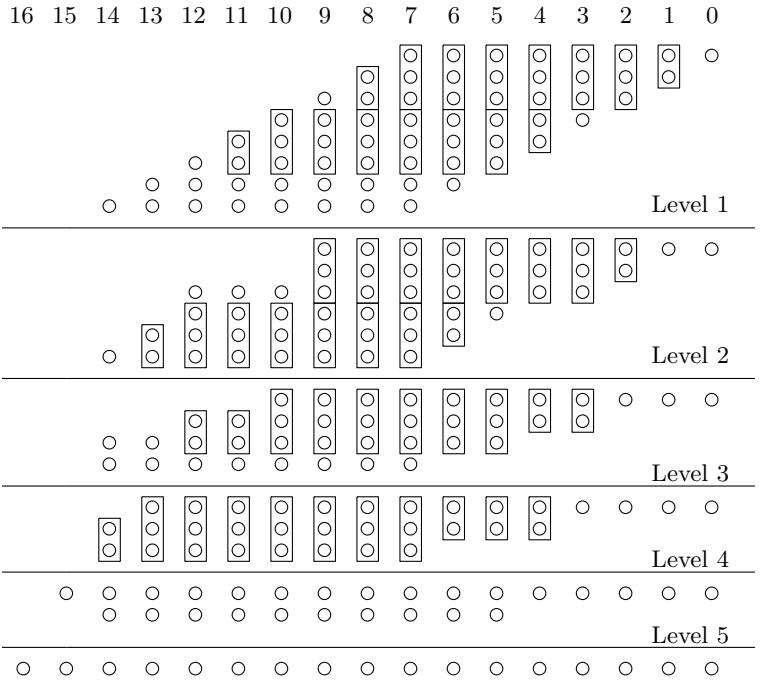


Fig. 8.23 Wallace tree multiplier scheme

1. Obtain the partial products and form the matrix.
2. Apply the HAs and FAs to reduce the operands. Try to reduce the operands as much as possible in parallel.
3. Apply carry propagating adder to generate the final product.

The flow of the Wallace tree multiplier is shown in Fig. 8.23 for 8-bit numbers. There are totally five layers of addition involved. Totally 16 number of HAs and 47 number of FAs are consumed.

Luigi Dadda in 1965 also gave some suggestions on fast multiplication [22]. He proposes some similar suggestions to achieve better speed. Dedda targets to reduce the operands in a particular manner so as to consume minimum number of basic elements. He achieves slightly better speed than the Wallace multiplier and consumes less hardware. The steps are

1. Obtain the partial products and form the matrix. Rearrange the matrix in the form of a tree.
2. Apply the HAs and FAs to reduce the operands. Try to reduce the operands as much as possible in parallel. Keep the height of length taking value from the set {2, 3, 4, 6, ..}.
3. Apply carry propagating adder to generate the final product.

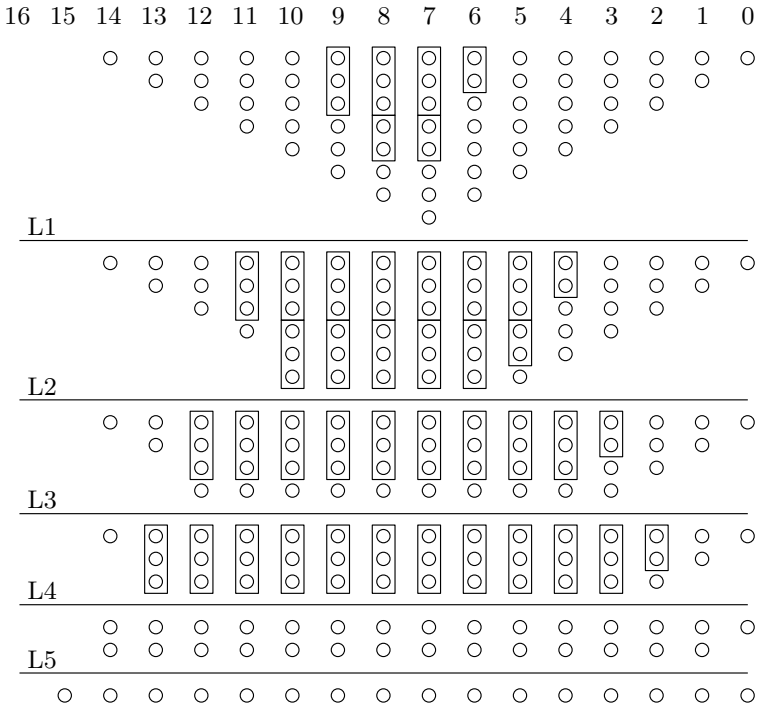


Fig. 8.24 Dedda tree multiplier scheme

The flow of the Dedda tree multiplier is shown in Fig. 8.24 for 8-bit numbers. There are totally five layers of addition involved. Totally 8 number of HAs and 48 number of FAs are consumed.

8.7 Multiplication Using Look-Up Tables

An alternative way of computing multiplication is using look-up tables. This technique can be useful where serial multiplication is needed or memory devices are available like in FPGA device. This technique is based on the following popular algebraic equation:

$$A \times B = \{(A+B)^2 - (A-B)^2\}/4 = 4 \times (A \times B)/4 \tag{8.15}$$

In the first step, two data elements A and B are added and subtracted. Then results of addition and subtraction are provided to two squaring tables as addresses. The squaring tables store the square of all the elements which are required to fetch. The output of the squaring tables is then subtracted to get the final multiplication result.

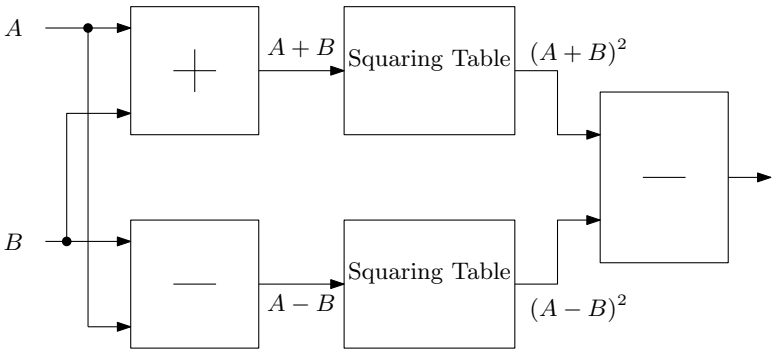


Fig. 8.25 A scheme for computing multiplication using tables

The simplest parallel architecture of multiplication using tables is shown in Fig. 8.25. Here, two n-bit adders and subtractors are placed in the first step. The results of these adder and subtractor blocks are of $(n + 1)$-bit. Thus size of the squaring tables will be $2^n \times (n + 1)$. At the final stage, a subtractor is placed which is of $2n$-bit size. The circuit is simpler and pipeline registers can be easily inserted.

One squaring table can also be used. The adder and subtractor blocks in the first stage also can be shared. In this case, the architecture will be serial and the circuit will be more hardware efficient. The size of the table can be reduced as they only store the square numbers. The square numbers will be always even and also the two LSB bits will be shifted out at the final stage. Thus the new size of the tables will be $2^n \times (n - 1)$.

8.8 Dedicated Square Block

In the previous sections, we have discussed techniques which can be used to achieve fast multiplication. But when the multiplicand and the multiplier are same, there must be some way to simplify the implementation. Thus squaring operation does not require the full length hardware of a multiplier. In applications where a squaring operation is required, a dedicated square block can be used.

In Fig. 8.26, the squaring operation for 2 bits is shown. Here the simplification of the result is shown. Further the squaring operation by a dedicated square block for 8 bits is shown in Fig. 8.27. The first array of partial products shows the original structure. The second array shows the rearrangement of the previous array. The logic for simplification is also shown in Fig. 8.27.

The fast multiplication techniques can be applied to the array of partial products as shown in the previous sections. It is sometimes required to arrange the partial products so the general techniques can be applied. Thus the array of partial products is arranged. Though it is not necessary. Figure 8.28 shows a possible architecture

$$
\begin{array}{cccc}
 & & a_1 & a_0 \\
 & \times & a_1 & a_0 \\
\hline
 & & a_0a_1 & a_0 \\
 & a_1 & a_0a_1 & \\
\hline
 & a_0a_1 & 0 & a_0 \\
 & a_1 & & \\
\hline
a_0a_1 & \overline{a_0}a_1 & 0 & a_0
\end{array}
$$

Fig. 8.26 Squaring operation for 2-bit data

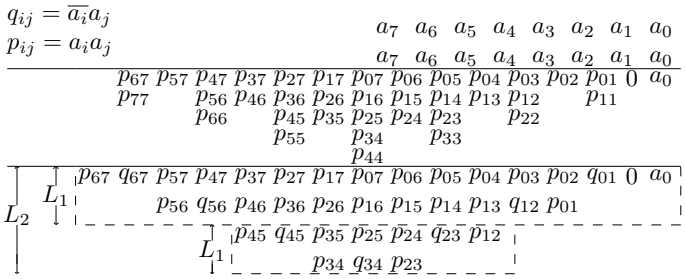


Fig. 8.27 Squaring operation for 8 bits

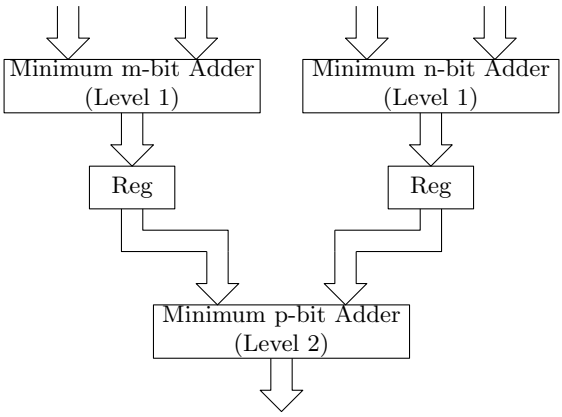


Fig. 8.28 Architecture of the dedicated square block

of the dedicated square block. Here the so-called CSA optimization techniques are not applied. The partial products are added in parallel to increase speed. The values of m, n, p are 13, 5, 11. Though this is not an optimized architecture but simple to implement.

8.9 Architectures Based on VEDIC Arithmetic

Ancient Indian VEDIC Mathematics consists of 16 mathematical formulae reconstructed from the Atharvaveda. It is recognized as an efficient technique for enhancing the mathematical skills of students. The arithmetic operations like multiplication, division, square root, cubing, squaring and finding cube root are time-consuming processes. VEDIC mathematics results in fast and easy solution for such type of time-consuming process. In this section, VEDIC mathematics is adopted to compute multiplication, square and cube of a number.

8.9.1 VEDIC Multiplier

VEDIC multiplication algorithm is another option to implement an efficient multiplier. This section discusses the VEDIC multiplier. There are three methods to implement multiplication in VEDIC mathematics. Out of three, only one method is generic method which can be applied to all cases whereas other two are for special cases. Main algorithm of Vedic multiplication is Urdhva Triyakbhyam. It is a general multiplication formula applicable to all cases of multiplication. It literally means Vertically and Crosswise.

The multiplication of two operands using VEDIC multiplier is achieved by multiplication by Vertically and Crosswise and then adding all the results. This multiplication algorithm can be understood using two operands 46 and 33. The operand 33 can be represented as $33 = (3 \times 10 + 3)$ and 46 can be represented as $46 = (4 \times 10 + 6)$. The multiplication (46×33) can be represented as $(3 \times 6 + 40 \times 3 + 30 \times 6 + 30 \times 40)$. This multiplication is shown in Fig. 8.29.

Similar way, this multiplication algorithm can be adopted to implement faster binary multiplier. A 4-bit binary multiplication is shown in Fig. 8.30. VEDIC multiplier is a good alternative to the other fast multiplicative algorithms. VEDIC multiplier reduces hardware as well as the delay compared to other algorithms. A 2-bit multiplier is shown in Fig. 8.31. This circuit uses just two HA blocks and four AND gates.

VEDIC multiplier for 4-bit data width is shown in Fig. 8.32. This structure is achieved using four 2-bit multipliers. Here three Add blocks are used. These blocks

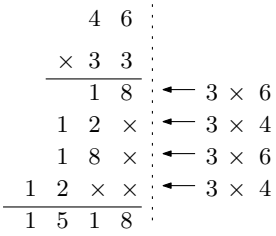


Fig. 8.29 VEDIC multiplication

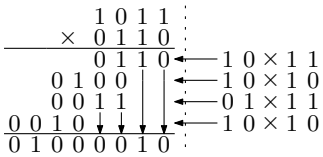


Fig. 8.30 VEDIC multiplication for 4-bit data width

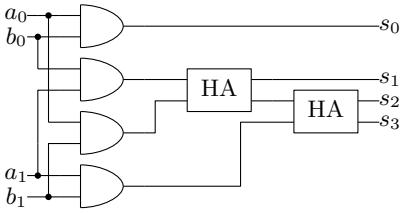


Fig. 8.31 Architecture of 2-bit VEDIC multiplier block

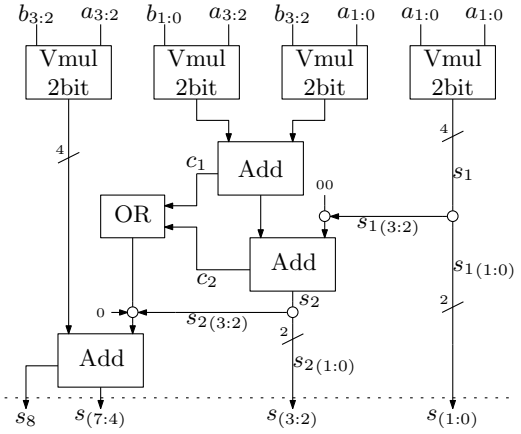


Fig. 8.32 4-bit VEDIC multiplier block

can be implemented using high-speed adders like conditional sum adder, carry look-ahead adder or carry select adder as shown in Chap. 7. In Fig. 8.32 the circles represent the concatenation block. For example, the 2 bits from the wire s_1 are connected to the output directly.

8.9.2 VEDIC Square Block

Square of a number can also be computed using VEDIC arithmetic formulas. Square computation is generally faster and hardware efficient than the complete multipliers. Similarly a VEDIC square block is hardware efficient than the multiplier block. Square of an operand is computed using the Dwandwa Yoga or Duplex method. Any number can be represented as (x + y) and in this method, square of this number can be computed as

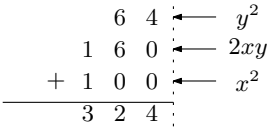


Fig. 8.33 VEDIC square example

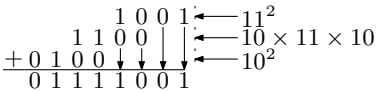


Fig. 8.34 VEDIC square for 4-bit binary data

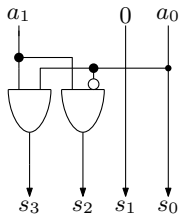


Fig. 8.35 Architecture of 2-bit square VEDIC square block

$$(x+y)^2 = x^2 + 2xy + y^2 \tag{8.16}$$

This can be shown with an example. Let the operand 18 and 18 can be represented as $(1 \times 10 + 8)$. Thus the square of this number using duplex square method is shown in Fig. 8.33.

Similarly, duplex square method can be applied to the binary numbers as well. This is shown in Fig. 8.34 by taking '1011' as an example. As discussed earlier that squaring operation is easier than multiplication. This will be clearer by seeing the architecture. The 2-bit duplex square block architecture is shown in Fig. 8.35.

This block is used to develop the 4-bit duplex square block. The architecture of 4-bit square block is shown in Fig. 8.36. Here, two 2-bit square block and one 2-bit multiplier block are used. The LSH1 block is used for left shifting for 1-bit. This shifting is wired shifting as discussed in Chap. 1. Here, two Add blocks are used in comparison to the three blocks in case of multiplier. These blocks can be implemented using high-speed adders like conditional sum adder, carry look-ahead adder or carry select adder as shown in Chap. 7. In Fig. 8.36 the circles represent the concatenation block.

8.9.3 VEDIC Cube Block

Apart from the multiplication and square, finding cube of a number is an another important operation in signal processing. The cube of a number can be computed using Anurupya Sutra which is the subsutra of Ekadhikena Purvena. VEDIC Cube architecture is efficient in terms of hardware and also in terms of delay. Cube of

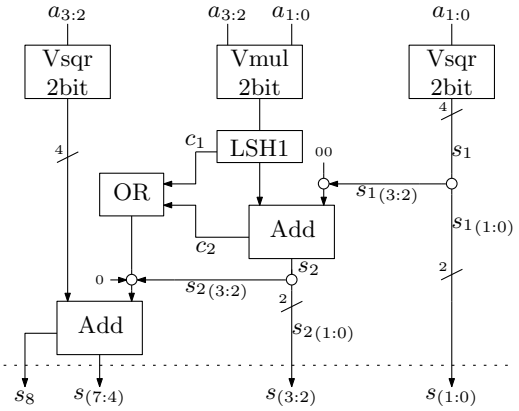


Fig. 8.36 Architecture of the 4-bit VEDIC square block

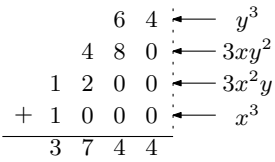


Fig. 8.37 Cube operation

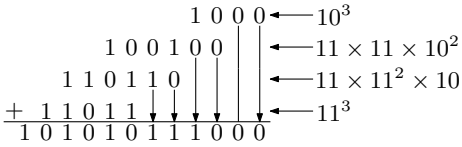


Fig. 8.38 Computation of cube for binary numbers

a number (z^3) can be computed easily using the VEDIC sutras. First express the numbers (z) as sum of two parts as ($z = x + y$). Then the cube of that number can be computed as

$$(x + y)^3 = x^3 + 3x^2y + 3xy^2 + y^2 \tag{8.17}$$

This operation is explained in Fig. 8.37 with an example. This method can be easily adopted for the binary numbers. VEDIC cube for 4-bit binary number is shown in Fig. 8.38 for z = '1110'.

In the computation of cube, several other operations are involved. These are square, multiplication and lower order cubes. The architecture of these blocks are discussed in earlier sections. Here the cube architectures are discussed. The 2-bit VEDIC cube architecture is shown in Fig. 8.39. This is optimized and simple block.

The architecture to compute cube of 4-bit number is shown in Fig. 8.40. In this figure, starting from the left side the operations computed are x^3, x^2, $3y$, $3x$, y^2 and y^3. All these operations are computed in parallel. This is the main advantage of this method. Multiplication by 3 is obtained by first multiplying by 2 and then adding the input number. The block LSH1 multiplies a number by 2 using wired left shifting. $3x$ and y^2 are multiplied to obtain $3xy^2$. Similarly, $3x^2y$ is computed. Then, $y^3 + 3xy^2$

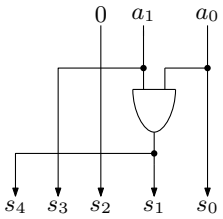


Fig. 8.39 2-bit cube architecture

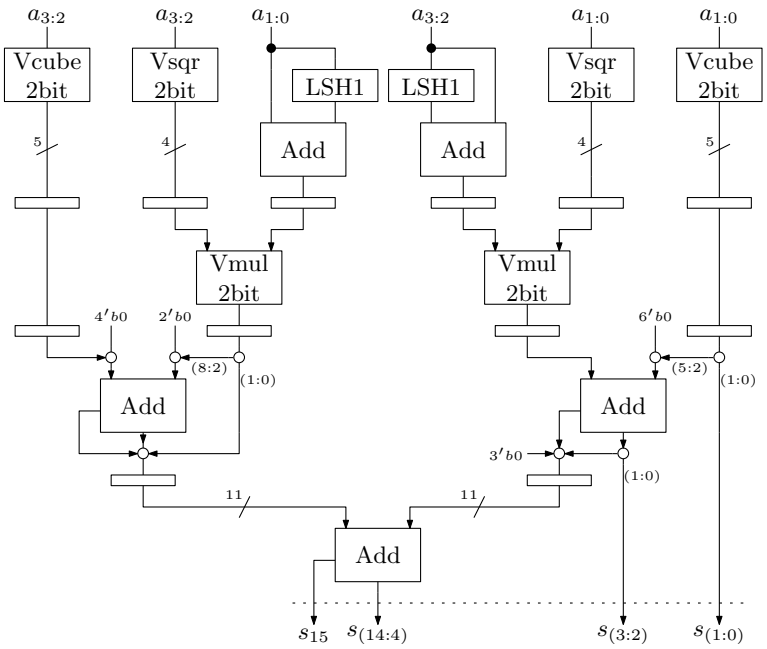


Fig. 8.40 VEDIC cube architecture for 4-bit data width

and $x^3 + 3x^2y$ are computed parallel. The results are added together at the last stage. The pipeline registers are also shown in suitable places. The circles are placed for concatenation operation. Add blocks can be implemented using high-speed adders like conditional sum adder, carry look-ahead adder or carry select adder as discussed in Chap. 7.

8.10 Conclusion

In this chapter, various multiplication schemes are discussed. Multiplication operation can be carried out by either sequential circuits or by parallel circuits. Sequential multipliers are used in systems where less area is desired. The array multipliers are basic parallel multipliers and available for both signed and unsigned operands.

This chapter mainly discusses fast multiplication algorithms. Fast step of the fast multiplication operation is partial products generation and reduction. Reduction of partial products can be achieved by Booth's radix-2 or radix-4 algorithm. In addition to Booth's algorithm, other methods of generation of partial products and reduction are also mentioned here. The generated partial products are added by fast multi-operand added. Various techniques of multi-operand addition are presented in this chapter. Two special multipliers, Wallace and Dedda tree multipliers, are also discussed.

This chapter also discusses how LUTs can be used to realize fast multiplier circuit. Along with the multiplication process, efficient architecture for computing square operation is also discussed here. VEDIC sutras based on ancient Indian arithmetic are very popular for developing efficient architecture for square, multiplication and cube. Thus this chapter also discusses the VEDIC arithmetic architectures.

Chapter 9
Division and Modulus Operation

9.1 Introduction

Many signal processing algorithms include the division operation which is more complex than the other arithmetic operations like addition/subtraction and multiplication. Both the timing complexity and the hardware complexity of a divider are higher than that of a multiplier. Thus it is always better to avoid the use of a divider in a digital system.

In this chapter, we will discuss various division algorithms and their architectures. The division algorithms can be classified as sequential algorithms, fast division algorithms and iterative approximation algorithms. Classification can also be done according to the base such as radix-2 dividers and higher radix dividers. Here, only radix-2 dividers are discussed in detail. Based on the sign of the operands, a divider can be signed and unsigned. Let's discuss some of the methods of division.

Another arithmetic operation which is discussed here is modulus operation. Evaluation of modulus operation is based on finding the residue. Thus repeated subtraction is the direct approach to find modulus. Computation of modulus using division operation is a general approach. But a divider with high complexity should be avoided to compute modulus as only the final residual is required. In this chapter, few such methods of modulus computation is discussed.

9.2 Sequential Division Methods

The basic equation of a division operation is

$$N = Q.D + R \tag{9.1}$$

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_9

where N is the dividend, D is the divisor, Q is the quotient and R is the remainder. The remainder R is less than D. Two types of sequential division algorithms are there which are restoring algorithm and non-restoring algorithm.

9.2.1 Restoring Division

The division operation is carried away by assuming that the dividend and the divisor are fractional. Also, the condition $N < D$ is true, so that overflow will not occur. In the division process, the residual obtained must always be less than the divisor ($R < D$). The quotient is generated as $Q = 0.q_1q_2q_3...q_m$, here $m = n - 1$. The restoring division algorithm is based on sequential addition and subtraction operation until the residual is zero or less than the divisor. The restoring division algorithm is shown below in Algorithm 9.1. Here n is the data width and R is set to N initially.

Algorithm 9. 1 Restoring division algorithm

Input: Dividend N, Divisor D and word length n.
Output: Quotient Q and Remainder R.
1: **Initialization** $r_0 = N$.
2: $r_i = 2 * r_{i-1} - D$
3: **for** $i \leftarrow 1$ to $(n - 1)$ **do**
4: **if** $r_i \geq 0$ **then**
5: $Q(i) = 1$
6: **else if** $r_i < 0$ **then**
7: $Q(i) = 0$
8: $r_i = r_i + D$
9: **end if**
10: **end for**

In Algorithm 9.1, the quotient bits are set based on the current residual and checked if it is greater than zero or not. If the r_i is greater than 0 then q_i is set to 1. If the current residual r_i is less than 0 then a restoring operation is carried out by adding D with the residual. An example of restoring algorithm is shown below for $N = 0.5(0.100)$ and $D = 0.75(0.110)$. Here positive values are considered and the result is $Q = 0.625$. The value of n is 4 here.

$r_0 = N$	0 0.1 0 0	
$2r_0$	0 1 0.0 0 0	
Add $(-D)$	1 1 0.0 1 0	
$r_1 = 2 * r_0 - D$	0 0 0.0 1 0	set $q_1 = 1$
$2r_1$	0 0 0.1 0 0	
Add $(-D)$	1 1 0.0 1 0	
$r_2 = 2 * r_1 - D$	1 1 0.1 1 0	set $q_2 = 0$
$r_2 = 2r_1$	0 0 0.1 0 0	Restoring
$2r_2$	0 1 0.0 0 0	
Add $(-D)$	1 1 0.0 1 0	
$r_3 = 2 * r_2 - D$	0 0 0.0 1 0	set $q_3 = 1$

A simplest architecture of the restoring division algorithm is shown in Fig. 9.1. The architecture is an array of sub-blocks (SB). The architecture of the SB block is also shown in that figure. The *sel* signal executes the restoring operation through a MUX. Totally n^2 number of SB blocks are used in the architecture. Intermediate pipeline registers can be used to increase the speed of operation. In such case, latency will be of three clock cycles.

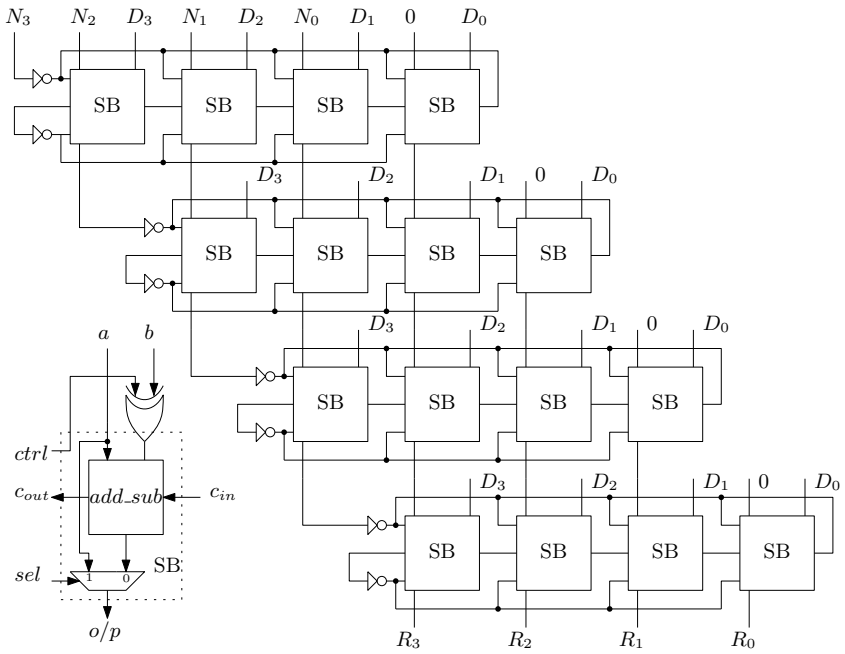


Fig. 9.1 Restoring division architecture

9.2.2 Unsigned Array Divider

Our objective is to divide N by D where both N and D can be positive or negative. The division operation can be expressed by the following equation by considering 4-bit data width as

$$N = Q_3.2^3.D + Q_2.2^2.D + Q_1.2^1.D + Q_0.2^0.D + R \tag{9.2}$$

The division operation is carried out by subsequent stages where each stage determines a quotient bit. Thus it is called as an array divider. In the first stage to find the bit q_3, D is left shifted by 3 bits and subtracted from N. The quotient bit q_3 is 1 if the difference is zero or positive. If the difference is negative, q_3 is 0 and the value of N is passed to the next stage. This technique is simply the restoring division discussed in the previous section. The division operation is explained by an example shown in Fig. 9.2 where $N = 1100$ and $D = 0010$.

The unsigned divider for 4-bit data width is shown in Fig. 9.3. From the example shown in Fig. 9.2, it is clear that to compute q_3, D_0 is subtracted from N_3 only when other bits of D are zero. This logic is implemented in the first stage. Similarly to compute q_2, bits D_1 and D_0 are needed to be subtracted and thus two sub-blocks are used. The block diagram of the sub-block is also given in Fig. 9.3. It consists of a full subtractor and a MUX.

The above architecture is shown for division by unsigned numbers. The signed division can be done easily by adopting this division method. The input signed operands (two's complement representation) are needed to be converted to unsigned operands and at the output the quotient and residual are also needed to be converted to signed operands depending on the sign of input operands.

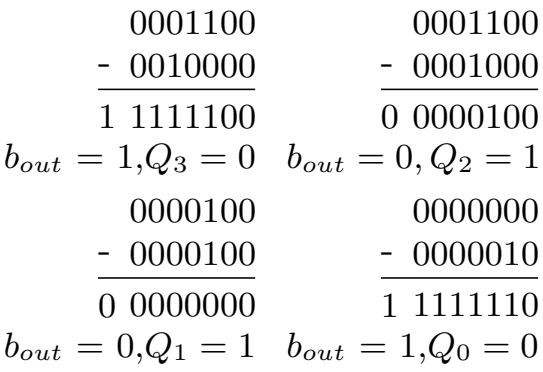


Fig. 9.2 An example of unsigned division for 4-bit binary numbers

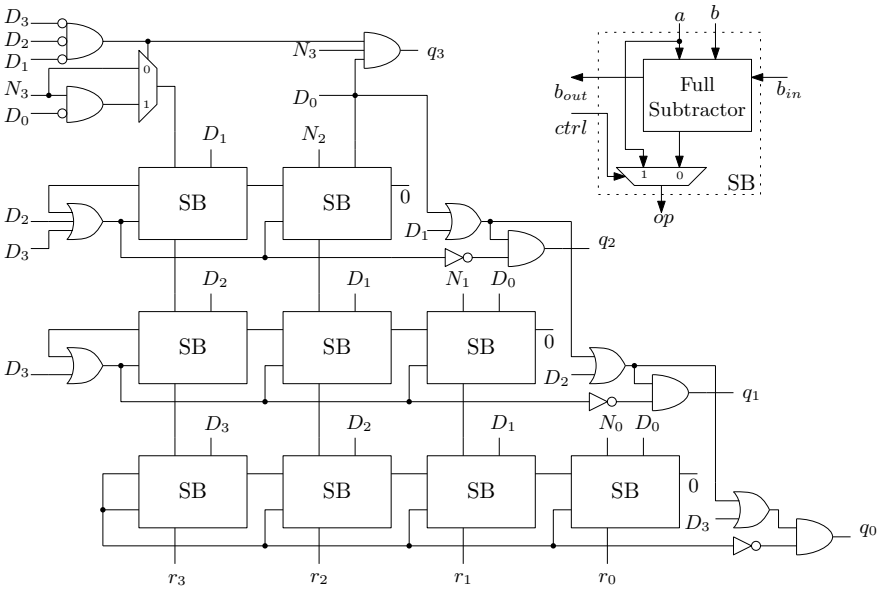


Fig. 9.3 A 4-bit unsigned divider

9.2.3 Non-restoring Division

The non-restoring algorithm is also a sequential division algorithm. It also works for fractional numbers. In non-restoring division algorithm, the quotient bit is not corrected and the restoration of the remainder also not done immediately if it is negative. The quotient bit in non-restoring algorithm is defined as

$$q_i = \begin{cases} 1, & \text{if } 2r_{i-1} \geq 0 \\ \overline{1}, & \text{if } 2r_{i-1} < 0 \end{cases} \tag{9.3}$$

The new remainder can be computed as

$$r_i = 2r_{i-1} - q_i.D \tag{9.4}$$

The generalized selection rule for quotient is

$$q_i = \begin{cases} 1, & \text{if } 2r_{i-1} \text{ and D have same sign} \\ \overline{1}, & \text{if } 2r_{i-1} \text{ and D have opposite sign} \end{cases} \tag{9.5}$$

The division operation is carried away by assuming fractional numbers. Initially R is set equal to N and n is the data width. The operands are in two's complement form

where MSB bit is the signed bit. The non-restoring division algorithm is shown in Algorithm 9.2.

Algorithm 9. 2 Non-restoring division algorithm

Input: Dividend N, Divisor D and word length n.
Output: Quotient Q and Remainder R.
1: **Initialization** $r_0 = N$.
2: **for** $i \leftarrow 1$ to $(n-1)$ **do**
3: **if** $2 * r_{i-1} \geq 0$ **then**
4: $Q(i) = 1$
5: $r_i = 2 * r_i - D$
6: **else if** $2 * r_i < 0$ **then**
7: $Q(i) = -1$
8: $r_i = 2 * r_i + D$
9: **end if**
10: **end for**

In non-restoring divider, quotient takes value from the digit set $\{-1, 1\}$. At the output, a conversion is needed to get the actual output in two's complement form. An example of non-restoring algorithm is shown below. This example is for N = 0.5 (0.100) and D = -0.75 (1.010). Here n is equal to 4.

$r_0 = N$	0 0.1 0 0	
$2r_0$	0 1 0.0 0 0	set $q_1 = \bar{1}$
Add (D)	1 1 0.0 1 0	
$r_1 = 2 * r_0 + D$	0 0 0.0 1 0	
$2r_1$	0 0 0.1 0 0	set $q_1 = \bar{1}$
Add (D)	1 1 0.0 1 0	
$r_2 = 2 * r_1 + D$	1 1 0.1 1 0	
$2r_2$	1 1 0.1 0 0	set $q_2 = 1$
Add $(-D)$	0 0 0.1 1 0	
$r_3 = 2 * r_2 - D$	0 0 0.0 1 0	

A simplest architecture of non-restoring algorithm-based division for $n = 4$ is shown in Fig. 9.4. This architecture is very similar to the restoring algorithm-based division architecture. This architecture also uses the same number of sub-blocks. Structure of the sub-blocks is also shown in Fig. 9.4. This SB block is different from the SB block mentioned in restoring case. Here, there is no provision of restoring the residual.

The architecture shown in Fig. 9.4 does not consider some aspects of non-restoring-based division. The sign of the remainder and the dividend should be same. If 10 is divided by –3 then the remainder should be 1 not –2 for quotient of –4. If the sign of the remainder is different from the dividend then a correction step is needed

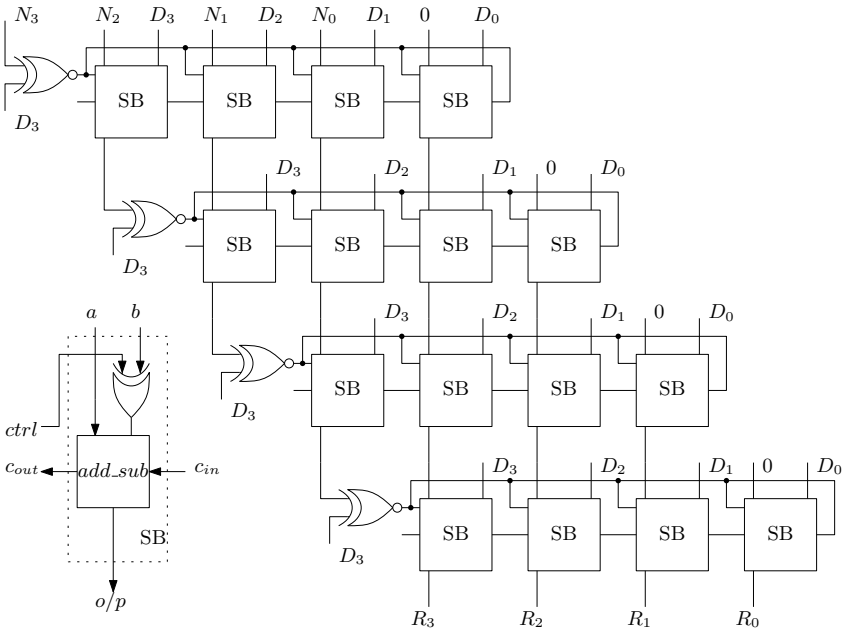


Fig. 9.4 Non-restoring division architecture

to generate exact remainder and quotient. It is to be noted that even quotient cannot be generated as the quotient bits are restricted $\{1, \overline{1}\}$. An example of this situation is shown below.

$r_0 = N$	0	0.1 0 1	
$2r_0$	0	1 0.0 1 0	set $q_1 = 1$
Add $(-D)$	+ 1	1 0.0 1 0	
$r_1 = 2 * r_0 - D$	0	0 0.1 0 0	
$2r_1$	0	1 0.0 0 0	set $q_2 = 1$
Add $(-D)$	+ 1	1 0.0 1 0	
$r_2 = 2 * r_1 - D$	0	0 0.0 1 0	
$2r_2$	0	0 0.1 0 0	set $q_3 = 1$
Add $(-D)$	+ 1	1 0.0 1 0	
$r_3 = 2 * r_2 - D$	1	1 0.1 1 0	

Here in this example, the sign of the remainder is negative while the sign of the dividend is positive thus a correction step is needed. The remainder is corrected by adding D to the final remainder r_3. This addition yields $1.110 + 0.110 = 0.100$. The quotient is also needed to be corrected as

$$Q_{corrected} = Q - ulp = 0.111 - 0.001 = 0.110 \tag{9.6}$$

Another situation in non-restoring division is that, the remainder cannot be zero in the intermediate steps. If the remainder becomes zero, then a correction step is needed. This situation is explained by the following example:

$r_0 = N$	1 0.1 0 1	
$2r_0$	1 1 0.0 1 0	set $q_1 = \bar{1}$
Add (D)	+ 0 0 0.1 1 0	
$r_1 = 2 * r_0 + D$	0 0 0.0 0 0	
$2r_1$	0 0 0.0 0 0	set $q_2 = 1$
Add $(-D)$	+ 1 1 0.0 1 0	
$r_2 = 2 * r_1 - D$	1 1 0.0 1 0	
$2r_2$	1 0 0.1 0 0	set $q_3 = \bar{1}$
Add (D)	+ 0 0 0.1 1 0	
$r_3 = 2 * r_2 + D$	1 1 0.0 1 0	

Here, sign of the final remainder and the dividend is same but the quotient is wrong. We get $Q = 0.\bar{1}1\bar{1} = 0.\bar{1}01_2 = 3/8$ in place of $-1/2$. Thus a correction step is needed according to the zero occurrence of the intermediate remainder.

$$r_3(corrected) = r_3 + D = 1.010 + 0.110 = 0.000 \tag{9.7}$$

The quotient is correct as

$$Q_{corrected} = 0.\bar{1}01 - 0.001 = 0.\bar{1}00 = -1/2 \tag{9.8}$$

Thus in general correction step is needed for

1. If the remainder and dividend have opposite signs.
   (a) If the dividend and divisor have the same sign, then the remainder is corrected by adding D and the quotient is corrected by subtracting ulp.
   (b) If the dividend and the divisor have opposite signs, then D is subtracted from remainder and quotient is corrected by adding ulp.
2. If there is a zero intermediate remainder. Correction is done according to the number of occurrence of zero remainder.

9.2.4 Conversion from Signed Binary to Two's Complement

The conversion from Signed Binary to Two's Complement is important in order to simplify the implementation. The quotient Q is represented in SD format as $0.q_1q_2q_3 = 0.\bar{1}\bar{1}1$. To get the actual quotient in two's complement format an on-the-fly conversion is needed. The basic scheme for conversion is

- First mask bits for $\bar{1}$. That is Q1 = 0.001.
- Then assign Q2 as Q2 = 0.110.
- Subtract Q2 from Q1 to get actual result. That is Q = 1.011 (–0.625).

In the hardware implementation, if $\bar{1}$ is represented as 0 then the quotient is $Q = 0.001$. This can be converted to two's complement form as

1. Shift the given number left by 1-bit position.
2. Complement the most significant bit.
3. Replace a 1 into the LSB position.

These steps result the same as $0.001 = (1 - 0).011$.

9.3 Fast Division Algorithms

9.3.1 SRT Division

The most well-known fast division algorithm is SRT division algorithm named after Sweeney, Robertson and Tocher [45, 58, 72]. The SRT algorithm basically improves the non-restoring algorithm. The non-restoring division algorithm needs n addition/-subtraction operations and allows bit 0 in the quotient for which no addition/subtraction operation needed. The rule for selecting quotient bit selection can be changed as

$$q_i = \begin{cases} 1, & \text{if } 2r_{i-1} \geq D \\ 0, & \text{if } -D \leq 2r_{i-1} < D \\ \bar{1}, & \text{if } 2r_{i-1} < -D \end{cases} \tag{9.9}$$

The difficulty with this new selection rule is that we need to compare $2r_{i-1}$ with D or $-D$. If the value of the normalized value of D is restricted as $1/2 \leq |D| < 1$ then the partial remainder $2r_{i-1}$ is to be compared with $1/2$ or $-1/2$. The new selection rule is now modified as

$$q_i = \begin{cases} 1, & \text{if } 2r_{i-1} \geq 1/2 \\ 0, & \text{if } -1/2 \leq 2r_{i-1} < 1/2 \\ \bar{1}, & \text{if } 2r_{i-1} < -1/2 \end{cases} \tag{9.10}$$

This algorithm is famously known as the SRT division algorithm. The SRT division can be extended to negative divisors in two's complement. The selection rule then becomes

$$q_i = \begin{cases} 0, & \text{if } |2r_{i-1}| < 1/2 \\ 1, & \text{if } |2r_{i-1}| \geq 1/2 \,\&\, r_{i-1} \text{ and } D \text{ have the same sign} \\ \bar{1}, & \text{if } |2r_{i-1}| \geq 1/2 \,\&\, r_{i-1} \text{ and } D \text{ have the opposite signs.} \end{cases} \tag{9.11}$$

An example of division operation using the SRT fast division algorithm is shown below for $N = 0.0101 = 5/16$ and $D = 0.1100 = 3/4$. The SRT algorithm is applied as follows:

$r_0 = N$		0 .0 1 0 1	
$2r_0$		0 .1 0 1 0	$\geq 1/2$ set $q_1 = 1$
Add $(-D)$	+	1 .0 1 0 0	
$r_1 = 2 * r_0 - D$		1 .1 1 1 0	
$2r_1 = r_2$		1 .1 1 0 0	$\geq -1/2$ set $q_2 = 0$
$2r_2 = r_3$	+	1 .1 0 0 0	$\geq -1/2$ set $q_3 = 0$
$2r_3$		1 .0 0 0 0	$< -1/2$ set $q_4 = \bar{1}$
Add (D)	+	0 .1 1 0 0	
r_4		1 .1 1 0 0	negative remainder and
Add D		0 .1 1 0 0	correction
r_4		0 .1 0 0 0	negative remainder and

The quotient generated before correction is $Q = 0.100\bar{1}$. This is a minimal representation of $Q = 0.0111$ in SD form. Means, minimum number of addition/subtract operations are performed. The correction of the quotient is made as $Q_{corrected} = 0.0111 - ulp = 0.0110 = 3/8$ and the final remainder is $1/2.2^{-4}$.

9.3.2 SRT Algorithm Properties

Based on the simulation and statistical analysis, SRT algorithm has the following properties:

1. The average number of shift operations in SRT division is $n/2.67$ where n is the length of the dividend. For example, for $n = 24$, approximately $24/2.67 \approx 9$ shift operations are needed.
2. The actual number of operations needed depends upon the divisor D. The smallest number is achieved when $17/28 \leq D \leq 3/4$ (approximately $3/5 \leq D \leq 3/4$) with an average shift of 3.

In order to reduce the number of add/subtract operations, the SRT method should be modified when the divisor happens to be out of range ($3/5 \leq D \leq 3/4$). Two ways to achieve this which are

1. In some of the steps of the division, multiple of D like $2D$ can be used if D is too small or $D/2$ can be used if D is too large. Subtracting $2D$ or $D/2$ instead of D is equivalent to performing subtraction one position earlier or later.
2. The comparison constant $K = 1/2$ can be changed if D is outside the optimal range. This change is allowed because the ratio of D/K matters since partial remainder is compared with K not with D.

9.4 Iterative Division Algorithms

9.4.1 Goldschmidt Division

The Goldschmidt division [29] is one of the popular fast division methods. It evaluates the division operation by iterative multiplications. The equations which govern Goldschmidt division are

$$F_i = 2 - D_{i-1} \tag{9.12}$$

$$D_i = F_i \times D_{i-1} \tag{9.13}$$

$$N_i = F_i \times N_{i-1} \tag{9.14}$$

The major advantage of Goldschmidt division is its low latency. In VLSI implementation where a designer wants to fasten the design performance then this divider can be used. The block diagram of this divider is shown in Fig. 9.5. It uses two multipliers per iteration. One of the disadvantages of this division is that pipeline implementation is costly. Thus serial implementation is beneficial to use. Another disadvantage of this divider is its accuracy. The need for the pre-scaling of input operands makes it not suitable for systems with high accuracy. More iteration means greater accuracy which again increases cost. Generally 3–5 iterations are sufficient to attain acceptable accuracy.

9.4.2 Newton–Raphson Division

Newton–Raphson's iterative algorithm is a very popular method to approximate a given function. It can be used to compute reciprocal of a given number. The Newton–Raphson's iterative equation is

$$X_{i+1} = X_i - \frac{f(X_i)}{f'(X_i)} \tag{9.15}$$

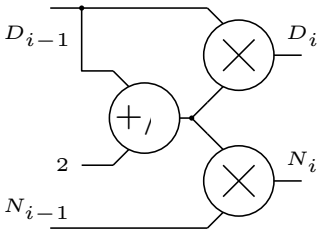


Fig. 9.5 Goldschmidt iterative division

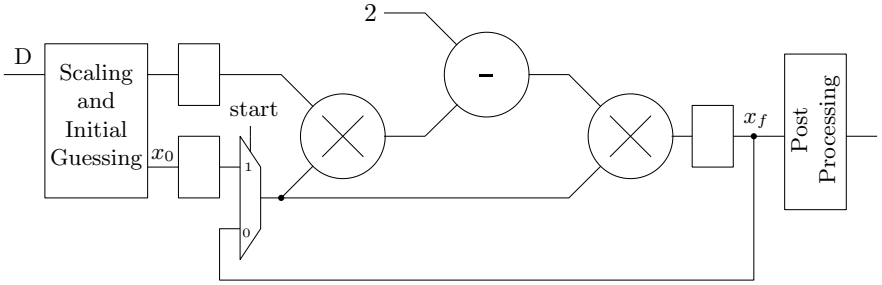


Fig. 9.6 Newton–Raphson iterative reciprocal computation

where $f^{'}(X_i)$ is the derivative of $f(X_i)$. The function which is used to compute the reciprocal of a number is $f(x) = (1/X) - D$, where D is the input operand. The Newton–Raphson iteration gives

$$X_{i+1} = X_i(2 - D.X_i) \tag{9.16}$$

After some finite iterations the above equation converges to the reciprocal of D. It is very obvious that initial value of X (X_0) must be chosen carefully to converge. The value of D is scaled to be in the range $0.5 \leq D \leq 1$ to choose the initial guess X_0 such that few number of iterations required for computation. In this case, D is shifted right or left to be in that range. In that interval of D, one must choose initial value X_0 as

$$X_0 = \frac{48}{17} - \frac{32}{17}D \tag{9.17}$$

The architecture for Newton–Raphson-based reciprocal computation is shown in Fig. 9.6. A serial architecture is given here as parallel architecture is costly and most system architectures use serial architecture. The pulsed control signal *start* starts the iteration and x_f is the final result. If D is not in the range, pre-processing is required and at the output a post-processing step is also required.

9.5 Computation of Modulus

Modulo operation is very important in many signal processing algorithms. Computation of modulus is mostly used in implementation of cryptographic algorithms. The modulus operation between two operands X and Y can be expressed as

$$X \, mod \, Y = X - \lfloor X/Y \rfloor Y \tag{9.18}$$

Here, $\lfloor X/Y \rfloor$ indicates that X is divided by Y and the result is floored by removing fractional part. $\lfloor X/Y \rfloor$ is multiplied by Y and the result is subtracted from X to compute modulus. An example of modulo operation is shown below:

$$17 mod\, 5 = 17 - \lfloor 17/5 \rfloor 5 = 17 - \lfloor 3.4 \rfloor 5 = 17 - 3 \times 5 = 2 \tag{9.19}$$

Computation of modulus is straightforward if Y is 2 or power of 2. This is achieved by simple wired shift method. But if the value of Y is not equal to power of 2 then computation of modulus becomes complex. This is because the modulus operation involves a division operation. This is why we are discussing modulus operation in this chapter.

Many algorithms are proposed in literature to compute modulus by avoiding division operation. One such algorithm is Barrett reduction algorithm [14] which computes the modulus according to Eq. (9.18). This algorithm states that if the value of Y is known then the reciprocal of Y can be pre-stored. Storing of $1/Y$ helps to avoid the division operation. The computation of modulus is done by the following equation:

$$X mod\, Y = X - \lfloor X \times (1/Y) \rfloor \times Y = X - D \times Y \tag{9.20}$$

Here, $D = \lfloor X \times (1/Y) \rfloor$. The above equation needs two multiplication operations and one subtraction operation. One multiplication is constant multiplication and another is of smaller size. The pre-computed value of $1/Y$ is accessed as a constant. The schematic for computation of $X mod\, Y$ is shown in Fig. 9.7. The value Y may not be fixed sometimes and in such case an LUT can be used which will store the pre-defined values of Y.

Another way of computing modulus is by number of addition and subtraction operations. This technique is based on the works reported in [20]. If X can be represented as summation of different terms then the modulus operation $X mod\, Y$ can be expressed as

$$\begin{aligned} X mod\, Y &= (2^{n-1} + ... + 2^2 + 2^1 + 2^0) mod\, Y \\ &= ((2^{n-1} mod\, Y) + ... + (2^2 mod\, Y) + (2^1 mod\, Y) + (2^0 mod\, Y)) mod\, Y \end{aligned} \tag{9.21}$$

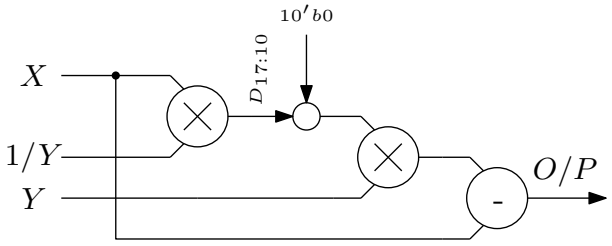


Fig. 9.7 A possible architecture to compute $X mod\, Y$

For example, $100 mod 7$ operation can be expressed as

$$100 mod 7 = ((64 mod 7) + (32 mod 7) + (4 mod 7)) mod 7 = (1 + 4 + 4) mod 7 \tag{9.22}$$

This technique is also based on prior knowledge of Y. In order to compute the modulus operation $100 mod 7$, modulus values for each term are stored in an LUT. The ith location of LUT stores $2^i mod Y$ for $i = 0, 1, 2..., (n-1)$. The modulus values for $n = 8$ and $Y = 7$ are shown below:

X	1	2	4	8	16	32	64	128
$X mod 7$	1	2	4	1	2	4	1	2

Architecture for the above technique can be easily designed by accessing the mod values from LUT and adding the mod values. Modulus of the result is computed by repetitive subtraction. The number of subtraction operations depends on size of both X and Y. For $n = 8$ and $Y = 7$ two subtraction steps are needed.

There are several modifications to this technique which are proposed. First modification is that, if a partition of X is less than Y then modulus need not to be computed. This reduces addition steps. For example, $4 mod 7$ need not be computed as 4 is already less than 7. Another modification can be applied by grouping partitions for which same modulus values are produced. For example, $4 mod 7 = 4$ and $32 mod 7 = 4$ are equivalent. Thus same mod values are stored for both the cases. An architecture is shown in Fig. 9.8 by combining these modifications.

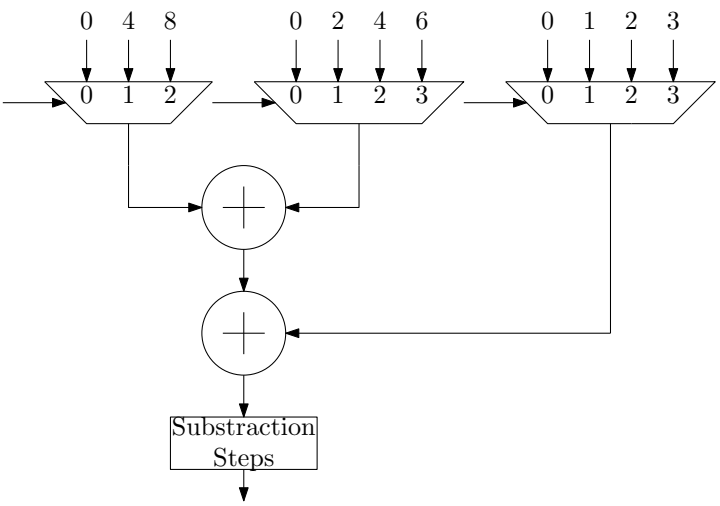


Fig. 9.8 Another possible architecture to compute $X mod Y$

9.6 Conclusion

In this chapter, various division algorithms are discussed and some architectures are also discussed. In digital systems, generally the array architectures for sequential division algorithms are preferred. Though the array structures have higher latency they are easier to implement. SRT algorithms are very critical to implement and also they sometimes do not support pipeline implementations. The iterative algorithms have less latency but their pipeline implementation is costly.

In this chapter, some architectures are explained to compute modulus operation without evaluating division operation as the hardware complexity of a divider is very high. The modulus architectures assume that the value of Y is known to the designers and modulus is evaluated with the help of LUTs. But if the value of Y is not known then these architectures will not be applicable. The readers are encouraged to design modulus architectures for variable Y.

Chapter 10
Square Root and its Reciprocal

10.1 Introduction

Generally the arithmetic operations like addition/subtraction, multiplication and division are used in implementation of signal processing algorithms. But square root and its reciprocal are other important arithmetic operations which are used in signal processing algorithms where Gram–Schmidt algorithm-based QR factorization or Cholesky factorization is used to solve linear equations.

The computational complexity involved in computation of square root and its reciprocal is similar to that of division operation. Hence, it is always better to avoid the computation of square root if possible. But not always these operations can be avoided. Thus we have also discussed some methods in this chapter to design hardware to compute square root and its reciprocal. Just like division, there are slow computing methods and also some algorithms for faster computation. A brief discussion is given in this chapter.

10.2 Slow Square Root Computation Methods

Computation of square root is similar to the division operation. The equation of a square root operation is

$$X = Q.Q + R \tag{10.1}$$

where X is the radicand, Q is the quotient and R is the remainder.

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_10

10.2.1 Restoring Algorithm

The restoring algorithm for square root is similar to the restoring division algorithm. Let X is the positive radicand and its square root is represented as $Q_n = 0.q_1q_2q_3...q_n$, when n is the total number of iterations. The bits of Q are generated in n steps, 1 bit per iteration. The Q_i is expressed as

$$Q_i = \sum_{k=0}^{i} q_k 2^{-k} \tag{10.2}$$

In restoring algorithm for square root the residual is updated as

$$r_i = 2r_{i-1} - (2Q_{i-1} + 2^{-i}) \tag{10.3}$$

Initially, r_0 is equal to N and the residual at second iteration is updated as

$$r_1 = 2r_0 - (0 + 2^{-1}) = 2X - 2^{-1} \tag{10.4}$$

The term $(2Q_{i-1} + 2^{-i})$ is equal to the term $(0.q_1q_2q_3...q_{i-1}01)$. If the intermediate remainder is positive then $q_i = 1$ and remainder is passed to the next step. Otherwise $q_i = 0$ and the remainder $r_i = 2r_{i-1}$. The restoring algorithm for square root operation can be considered as division operation with varying divisor. The restoring algorithm for square root computation is shown in Algorithm 10.1. An example of the square root computation of the radicand X = 0.1011 is shown below. The result is Q = 0.1101. The value of n is 4 here.

Algorithm 10.1 Restoring algorithm for square root computation

Input: Radicand X and word length n.
Output: Quotient Q and Remainder R.
1: **Initialization** $r_0 = X$.
2: **for** $i \leftarrow 1$ to n **do**
3: $r_i = 2 * r_{i-1} - (2Q_{i-1} + 2^{-i})$
4: **if** $r_i \geq 0$ **then**
5: $q_i = 1$
6: **else if** $r_i < 0$ **then**
7: $q_i = 0$
8: $r_i = 2r_{i-1}$
9: **end if**
10: **end for**

A simple architecture for the restoring square root algorithm is shown in Fig. 10.1 for 8-bit input operand. This is an array-type architecture similar to the architecture for restoring division. The architecture of the SB is also shown in Fig. 10.1. An SB consists of full subtractor and a MUX. The MUX is used for satisfying the restoring criteria.

$r_0 = X$	0 .1 0 1 1	
$2r_0$	0 1 .0 1 1 0	
Add $(-(0 + 2^{-1}))$	- 0 0 .1 0 0 0	
r_1	0 0 .1 1 1 0	set $q_1 = 1$
$2r_1$	0 1 .1 1 0 0	
Add $(-(2Q_1 + 2^{-2}))$	- 0 1 .0 1 0 0	
r_2	0 0 .1 0 0 0	set $q_2 = 1$
$2r_2$	0 1 .0 0 0 0	
Add $(-(Q_2 + 2^{-3}))$	- 0 1 .1 0 1 0	
r_3	1 1 .0 1 1 0	R is negative, set $q_3 = 0$
$r_3 = 2r_2$	0 1 .0 0 0 0	Restoring
Add $(-(Q_3 + 2^{-4}))$	- 0 1 .1 0 0 1	
r_4	0 0 .0 1 1 1	set $q_4 = 1$

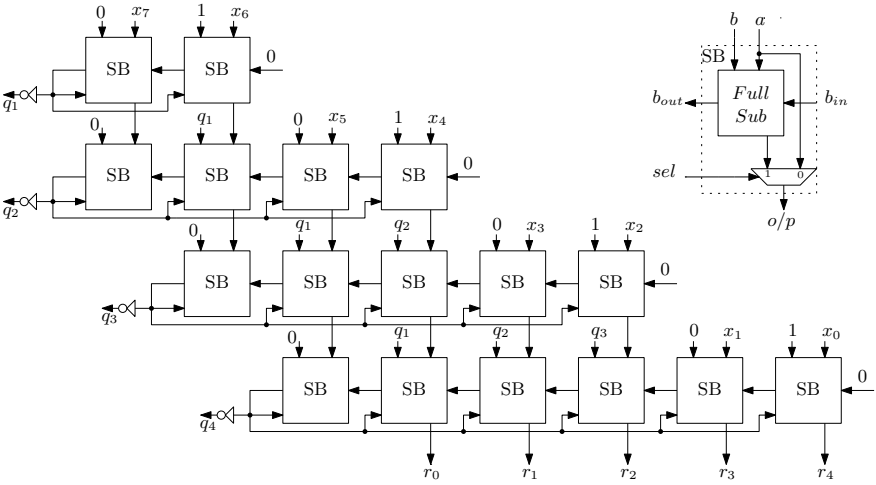


Fig. 10.1 Restoring square root architecture

10.2.2 Non-restoring Algorithm

The Non-restoring (NR) algorithm for square root operation is similar to the NR operation for division operation. It is similar to the restoring algorithm but it has no restoring step. In restoring step, the intermediate residue is restored. In non-restoring algorithm, the quotient values are updated as

$$q_i = \begin{cases} 1, & \text{if } 2r_{i-1} \geq 0 \\ \overline{1}, & \text{if } 2r_{i-1} < 0 \end{cases} \tag{10.5}$$

and the residual is updated as

$$r_i = 2r_{i-1} - q_i(2Q_{i-1} + q_i 2^{-i}) \tag{10.6}$$

Algorithm 10.2 Non-restoring division algorithm

Input: Radicand X and word length n.
Output: Quotient Q and Remainder R.
1: **Initialization** $r_0 = X$.
2: **for** $i \leftarrow 1$ to n **do**
3: **if** $2 * r_{i-1} \geq 0$ **then**
4: $q_i = 1$
5: $r_i = 2r_{i-1} - (2Q_i + 2^{-i})$
6: **else if** $2 * r_{i-1} < 0$ **then**
7: $q_i = -1$
8: $r_i = 2r_{i-1} + (2Q_i - 2^{-i})$
9: **end if**
10: **end for**

The NR algorithm for square root operation is shown in Algorithm 10.2. The initial residual r_0 is set to X. In non-restoring square root algorithm, quotient takes value from the digit set $\{-1, 1\}$. At the output, a conversion is needed to get the actual output in two's complement format. More about the conversion techniques are shown in Chap. 9. An example of non-restoring square root is shown below for X = 25/64 (0.011001).

$r_0 = X$	0 .0 1 1 0 0 1	
$2r_0$	0 .1 1 0 0 1 0	set $q_1 = 1$ & $Q_1 = 0.1$
Add $(-(0 + 2^{-1}))$	0 .1 0 0 0 0 0	
r_1	0 .0 1 0 0 1 0	
$2r_1$	0 0 .1 0 0 1 0 0	set $q_2 = 1$ & $Q_2 = 0.11$
Add $(-(2Q_1 + 2^{-2}))$	0 1 .0 1 0 0 0 0	
r_2	1 1 .0 1 0 1 0 0	
$2r_2$	1 0 .1 0 1 0 0 0	set $q_3 = \bar{1}$ & $Q_2 = 0.11\bar{1}$
Add $(+(2Q_2 - 2^{-3}))$	0 1 .1 0 $\bar{1}$ 0 0 0	
r_3	0 0 .0 0 0 0 0 0	

Thus the output is $0.q_1q_2q_3 = 0.11\bar{1}$. To get the actual output an on-the-fly conversion is needed as shown below:

- First mask bits for $\bar{1}$. That is Q1 = 0.110.
- Then assign Q2 as Q2 = 0.001.
- Subtract Q2 from Q1 to get actual result. That is Q = 0.101 (0.625)

NR algorithm for square root operation with bipolar Q has paved the way for many fast algorithms for square root operation. This may require an on-the-fly conversion but the high-speed SRT algorithms are based on the basic NR algorithm. The restoring and non-restoring algorithms may differ but in implementation of these two algorithms, both are very similar. A simpler architecture for NR algorithm for square root is shown in Fig. 10.2. Here, the SB is different than the SB block used in the restoring architecture.

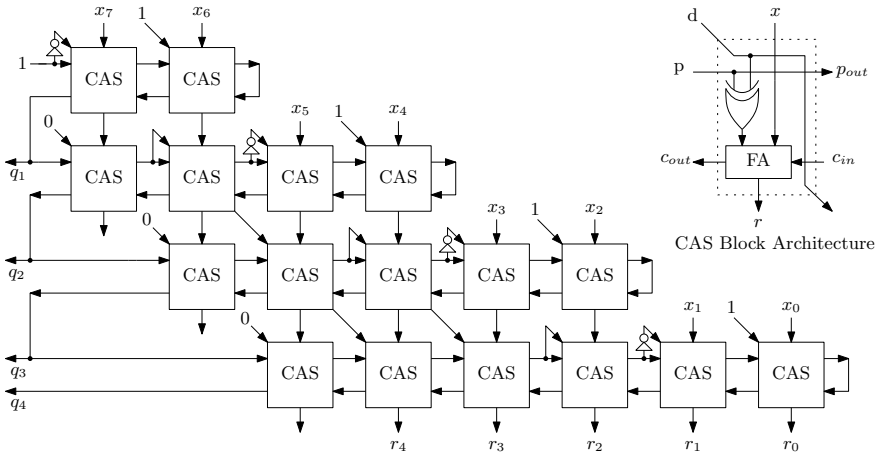


Fig. 10.2 Non-restoring square root architecture

10.3 Iterative Algorithms for Square Root and its Reciprocal

10.3.1 Goldschmidt Algorithm

The Goldschmidt algorithm [29] is one of the popular fast iterative methods. This algorithm can be used to compute division, square root or square root reciprocal. The basic version of the Goldschmidt algorithm to compute square root reciprocal of the radicand X is governed by the following equations. All these equations are sequentially executed in an iteration.

$$b_i = b_{i-1} \times Y_{i-1}^2 \tag{10.7}$$

$$Y_i = (3 - b_i)/2 \tag{10.8}$$

$$y_i = y_{i-1} \times Y_i \tag{10.9}$$

Initially $b_0 = X$, Y_0 = a rough estimate of $\frac{1}{\sqrt{X}}$ and $y_0 = Y_0$. The initial guess of Y_0 can be done by selecting a close value from a pre-defined LUT. The algorithm runs until b_i converges to 1 or for a fixed number of iterations. Finally square root reciprocal is computed as $y_n = \frac{1}{\sqrt{X}}$. Note that this algorithm can be used to compute both square root and its reciprocal. The square root function can be computed by multiplying the final value of y by X as $x_n = X \times y_n$. This algorithm involves three multiplications and one subtraction per iteration to compute y_n. Square root can be computed by another multiplication at the end.

Another version of Goldschmidt algorithm is shown below:

$$r_i = 1.5 - x_{i-1} \times h_{i-1} \tag{10.10}$$

$$x_i = x_{i-1} \times r_i \tag{10.11}$$

$$h_i = h_{i-1} \times r_i \tag{10.12}$$

Here initially y_0 equals to the closest approximation of the function $\frac{1}{\sqrt{X}}$, $x_0 = X \times y_0$ and $h_0 = 0.5y_0$. At the end of the iterations x_n converges to $\sqrt{X}$ and $2h_n$ converges to $\frac{1}{\sqrt{X}}$. These algorithms run until r_i reaches close to 0 or for fixed number of iterations. This improved version of Goldschmidt algorithm is faster than its previous version as at output stage no multiplier is needed to compute the square root function. But number of arithmetic operations per iteration remains the same.

10.3.2 Newton–Raphson Iteration

Newton–Raphson iterative algorithm is a very popular method to approximate a given function. It can be used to compute square root or square root reciprocal of a given number. The Newton–Raphson iterative equation is

$$x_{i+1} = x_i - \frac{f(x_i)}{f^{'}(x_i)} \tag{10.13}$$

where $f^{'}(x_i)$ is the derivative of $f(X_i)$. The function which is used to compute the square root reciprocal of a number is $f(x) = (1/x^2) - X$, where X is the input operand. The Newton–Raphson iteration gives

$$x_i = 0.5 \times x_{i-1}(3 - X \times x_{i-1}^2) \tag{10.14}$$

Here x_0 is taken as close approximation of $\frac{1}{\sqrt{X}}$. After some finite iterations, the above equation converges to the square root reciprocal of X. It is very obvious that the value of x_0 must be chosen carefully to converge.

The square root of X can be computed by multiplying the final output x_n by X or directly iterating for square root. The function $f(x) = (x^2) - X$ can be used in this case. Then the iterative equation will be

$$x_i = 0.5 \times (x_{i-1} + \frac{X}{x_{i-1}}) \tag{10.15}$$

Similar to the previous algorithms here also x_0 takes the closest approximation of $\sqrt{X}$. One of the disadvantage of direct computation of square root by Newton's theorem is that it involves a division operation which is much complex than a multiplication operation.

10.3.3 Halley's Method

Halley's method is actually the Householder's method of order two. The equation which governs this algorithm is

$$y_i = X \times x_{i-1}^2 \tag{10.16}$$

$$x_i = \frac{x_{i-1}}{8}(15 - y_i(10 - 3y_i)) \tag{10.17}$$

At the final step, x_n holds the value of square root reciprocal of x and y_n converges to 1. This method converges cubically but involves four multiplications per iteration.

10.3.4 Bakhshali Method

This method for finding an approximation to a square root or square root reciprocal was described in an ancient Indian mathematical manuscript. This algorithm is quadratically convergent. The equations are

$$a_i = \frac{(X - x_{i-1}^2)}{2x_{i-1}} \tag{10.18}$$

$$b_i = a_i + x_{i-1} \tag{10.19}$$

$$x_i = b_i - \frac{a_i^2}{2b_i} \tag{10.20}$$

The variable a_n approaches zero and x_n holds the value of square root.

10.3.5 Two Variable Iterative Method

This method is used to find square root of a number who's range is $1 < X < 3$. However the range of X can be increased. This method converges quadratically. The equations are

$$a_i = a_{i-1} - 0.5 \times c_{i-1}^2 \tag{10.21}$$

$$c_i = 0.5c_i^2(c_{i-1} - 3) \tag{10.22}$$

Initially $a_0 = X$ and $c_0 = X - 1$. Finally the variable a_n holds the final value of $\sqrt{X}$.

10.4 Fast SRT Algorithm for Square Root

The restoring and non-restoring algorithm for division and square root are very similar. Thus common hardware may be designed to perform both division and square root. In Chap. 9, we have discussed that how SRT algorithm can accelerate the computation of division. The same SRT algorithm can be applied to compute square root.

SRT algorithm says, if the radicand X satisfies the relation $1/4 \leq X < 1$, then the quotient Q can be restricted to $1/2 \leq Q < 1$ where q_1 is always 1. In this case the remainder r_{i-1} (for $i \geq 2$) satisfies the condition

$$-2(Q_{i-1} - 2^{-i}) \leq r_{i-1} \leq (Q_{i-1} + 2^{-i}) \tag{10.23}$$

Here, Q_{i-1} is partially calculated root at step $(i-1)$ and expressed as $Q_{i-1} = 0.q_1q_2..q_{i-1}$. The possible rule for selecting the quotient bits is

$$q_i = \begin{cases} 1, & \text{if } r_{i-1} \geq (Q_{i-1} + 2^{-i-1)}) \\ 0, & \text{if } -(Q_{i-1} - 2^{-i-1)}) \leq r_{i-1} \leq (Q_{i-1} + 2^{-i-1)}) \\ \bar{1}, & \text{if } r_{i-1} \leq -(Q_{i-1} - 2^{-i-1)}) \end{cases} \tag{10.24}$$

Since $(Q_{i-1} + 2^{-i-1)})$ and $(Q_{i-1} - 2^{-i-1)})$ are in the range $[1/2, 1]$, the above selection can be changed as

$$q_i = \begin{cases} 1, & \text{if } 1/2 \leq 2r_{i-1} \leq 2 \\ 0, & \text{if } -1/2 \leq 2r_{i-1} < 1/2 \\ \bar{1}, & \text{if } -2 \leq 2r_{i-1} \leq -1/2 \end{cases} \tag{10.25}$$

This selection rule is similar to the SRT selection rule for division process. An example of SRT square root is shown below for $X = 0.0111101_2 = 61/128$

The square root is $Q = 0.11\bar{1}1001 = 0.1011001 = 89/128$. The final remainder is $2^{-7}r_7 = -113/2^{14}$.

10.5 Taylor Series Expansion Method

10.5.1 Theory

Taylor series expansion method is very popular for computing reciprocal, square root, square root reciprocal and other elementary functions. This method is according to the proposed method in [25]. Here input data is restricted to a range as $1 \leq X < 2$. The

$r_0 = X$		0 .0 1 1 1 1 0 1	
$2r_0$		0 .1 1 1 1 0 1 0	set $q_1 = 1$ & $Q_1 = 0.1$
Add $(-(0 + 2^{-1}))$	-	0 .1 0 0 0 0 0 0	
r_1		0 .0 1 1 1 0 1 0	
$2r_1$		0 .1 1 1 0 1 0 0	set $q_2 = 1$ & $Q_2 = 0.11$
Add $(-(2Q_1 + 2^{-2}))$	-	0 1 .0 1 0 0 0 0 0	
r_2		1 .1 0 1 0 1 0 0	
$2r_2$		1 1 .0 1 0 1 0 0 0	set $q_3 = \bar{1}$ & $Q_3 = 0.11\bar{1}$
Add $(+(2Q_2 - 2^{-3}))$	+	0 1 .0 1 1 0 0 0 0	
r_3		0 0 .1 0 1 1 0 0 0	
$2r_3$		0 1 .0 1 1 0 0 0 0	set $q_4 = 1$ & $Q_4 = 0.11\bar{1}1$
Add $(-(2Q_3 + 2^{-4}))$	-	0 1 .0 1 0 1 0 0 0	
r_4		0 0 .0 0 0 1 0 0 0	
$2r_4$		0 0 .0 0 1 0 0 0 0	set $q_5 = 0$ & $Q_5 = 0.11\bar{1}10$
r_5		0 0 .0 0 1 0 0 0 0	
$2r_5$		0 0 .0 1 0 0 0 0 0	set $q_6 = 0$ & $Q_6 = 0.11\bar{1}100$
r_6		0 0 .0 1 0 0 0 0 0	
$2r_6$		0 0 .1 0 0 0 0 0 0	set $q_7 = 1$ & $Q_7 = 0.11\bar{1}1001$
Add $(-(2Q_6 + 2^{-7}))$	-	0 1 .0 1 1 0 0 0 1	
r_7		1 0 .0 0 0 1 1 1 1	

length of the input data X is m. The approximation of the functions is accomplished in three steps which are

1. Reduction : In this step, an n-bit number A deduced from the input operand such that $-2^{-k} \leq A < 2^k$. Here, $n = 4k$ and A is obtained as

$$A = XR - 1 \tag{10.26}$$

   Here, R is the reduction factor which is $(k + 1)$-bit approximation of reciprocal of X. The range of A is thus $1 - 2^{-k} \leq A < 1 + 2^k$. The value of R can be obtained from a $(k + 1) \times k$ look-up table. This look-up table is named as rb.
2. Evaluation : The evaluation of the function is carried out expanding the Taylor series and eliminating the terms that are less than 2^{4k}. According to the Taylor series, $B = f(A)$ is evaluated as

$$f(A) = c_0 + c_1 A + c_2 A^2 + c_3 A^3 + ... \tag{10.27}$$

   Here, the deduced number A can be written as

$$A = A_2 Z^2 + A_3 Z^3 + A_4 Z^4 + ... \tag{10.28}$$

   Here, $z = 2^{-k}$ and $|A_i| \leq 2^k - 1$. Then the following final equations are used to evaluate the functions:

$$\frac{1}{1 + A} \approx (1 - A) + A_2^2 z^4 + 2A_2 A_3 z^5 - A_2^3 z^6 \tag{10.29}$$

$$\sqrt{1+A} \approx 1 + \frac{A}{2} - \frac{1}{8}A_2^2 z^4 - \frac{1}{4}A_2 A_3 z^5 + \frac{1}{16}A_2^3 z^6 \tag{10.30}$$

$$\frac{1}{\sqrt{1+A}} \approx 1 - \frac{A}{2} + \frac{3}{8}A_2^2 z^4 + \frac{3}{4}A_2 A_3 z^5 - \frac{5}{16}A_2^3 z^6 \tag{10.31}$$

3. Post-processing : The post-processing is required as the number A is deduced by multiplying R. The final value of the function is obtained as

$$Y = M \times B \tag{10.32}$$

This multiplication factor varies for different functions as

(a) Reciprocal—$M = R$.
(b) Square Root—$M = \frac{1}{\sqrt{R}}$.
(c) Square Root Reciprocal—$M = \sqrt{R}$.

The multiplication factor is also obtained from a $n \times k$ look-up table. This look-up is named as ctb.

10.5.2 Implementation

The implementation is also divided into three parts which are reduction, evaluation and post-processing. The architecture in [25] is implemented with floating point format but can be implemented using the fixed point format also. Main advantage of using Taylor series expansion is to use smaller multipliers for computation of the functions.

The scheme for the reduction step is shown in Fig. 10.3. Here, two LUTs are used to store values of M and R. One small multiplier is used whose size is $n \times (k+1)$. The control signal sel selects suitable multiplication factor for reciprocal function. Output of the multiplier is of $3k$ bits.

The structure of the evaluation step and post-processing step is shown in Fig. 10.4. Here, one $k \times k$ multiplier is used to compute A_2^2, one $k \times k$ multiplier is used to compute $A_2 A_3$ and another $k \times k$ multiplier is used for computation of A_2^3. All the multiplication results are then added by the adder elements. In the post-processing step, M is multiplied by B to get the final result. Here only one big multiplier $(3k+1) \times (3k+2)$ is used. The Taylor series expansion method is very efficient in terms of resources when double precision is used. Some other functions can also be computed by these method which are reported in [25].

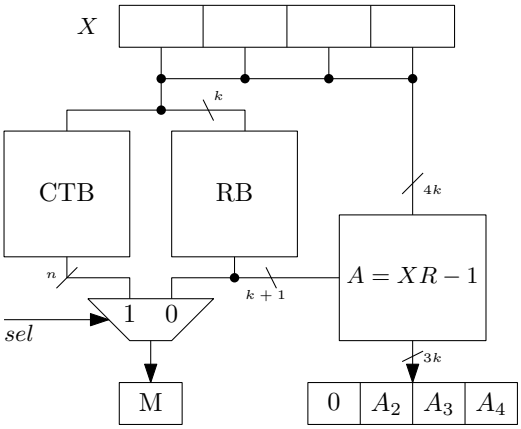


Fig. 10.3 Scheme for reduction step in Taylor series-based implementation of different functions

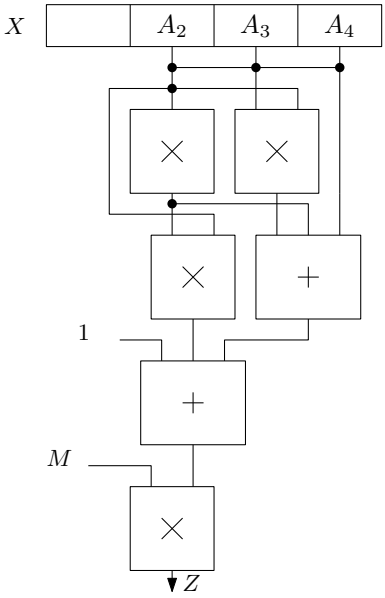


Fig. 10.4 Scheme for evaluation and post-processing step in Taylor series-based implementation of different functions

10.6 Function Evaluation by Bipartite Table Method

Various functions can be approximated using tables. Bipartite table method [63] is a very popular method to approximate different functions like reciprocal and square root reciprocal. The fractional part of the input operand is divided into three parts viz, x_0, x_1 and x_2 as shown in Fig. 10.5. If the input operand is of n-bits then x_0 is of n_0-bits, x_1 is of n_1-bits and x_2 is of n_2-bits where $n = n_0 + n_1 + n_2$.

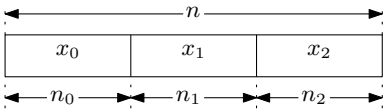


Fig. 10.5 Format of the input operand

This method is also based on the Taylor series expansion. In this method, a function $f(x)$ is approximated as

$$f(x) = f(x_0 + x_1 + x_2) \approx \alpha_0(x_0, x_1) + \alpha_1(x_0, x_2) \tag{10.33}$$

So, in this method there are two tables used. One table contains the co-efficients of the function $\alpha_0(x_0, x_1)$ and another table contains the co-efficients of the function $\alpha_1(x_0, x_2)$. The address line for the first table is (n_0, n_1) and the address line for the second table is (n_0, n_2).

If it is assumed that $0 \leq x < 1$ then following relations are true:

$$0 \leq x_0 \leq 1 - 2^{-n_0} \tag{10.34}$$

$$0 \leq x_1 \leq 2^{-n_0} - 2^{-n_0-n_1} \tag{10.35}$$

$$0 \leq x_1 \leq 2^{-n_0-n_1} - 2^{-n_0-n_1-n_3} \tag{10.36}$$

This method is based on the Taylor series expansion of the $f(x)$ around the point $x_0 + x_1 + \delta_2$ where

$$\delta_2 = 2^{-n_0-n_1} - 2^{-n_0-n_1-n_2-1} \tag{10.37}$$

Then the input function is approximated as

$$f(x) \approx f(x_0 + x_1 + \delta_2) + f^{'}(x_0 + x_1 + \delta_2)(x_2 - \delta_2) \tag{10.38}$$

where the first part is

$$\alpha_0(x_0, x_1) = f(x_0 + x_1 + \delta_2) \tag{10.39}$$

Since the second term cannot depend on x_1, x_1 is replaced with δ_1. The second part is

$$\alpha_1(x_0, x_2) = f^{'}(x_0 + \delta_1 + \delta_2)(x_2 - \delta_2) \tag{10.40}$$

The value of δ_1 is

$$\delta_1 = 2^{-n_0-1} - 2^{-n_0-n_1-1} \tag{10.41}$$

In evaluating the reciprocal or square root reciprocal, the input operand is limited as $1 \leq x < 2$. In case of reciprocal of the input operand x, the function is $f(x) = 1/x$. Then the approximation equation is

$$1/x = \frac{1}{(x_0 + x_1 + \delta_2)} - \frac{(x_2 - \delta_2)}{(x_0 + \delta_1 + \delta_2)^2} \tag{10.42}$$

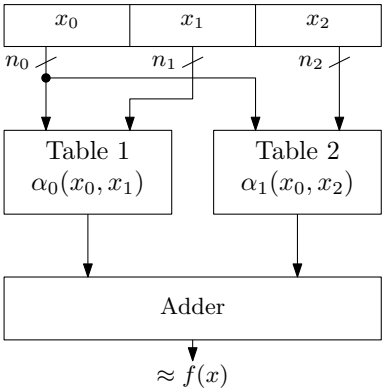


Fig. 10.6 Architecture for approximating any function using look-up tables

The simple architecture for table-based approximation of an unknown function is shown in Fig. 10.6. Here, this architecture is very simple and has very low latency. Except the two tables only hardware used here is the adder. Typical values for reciprocal computation for n-bit data width are $n_0 = 6$, $n_1 = 4$ and $n_2 = 5$ as $n_0 + n_1 + n_2 = n - 1$.

10.7 Conclusion

In this chapter, we have discussed various algorithms for computation of square root of a positive operand X. Square root operation is very similar to the division operation. Thus most of the algorithms for division are also applied to square root computation. Thus several hardware are reported which can compute both square root and division.

Initially sequential algorithms like restoring and non-restoring algorithms are discussed. The architectures of these algorithms are also presented in this chapter. The array architectures for square root are very popular as they are simple to use. This chapter also discusses the computation of reciprocal and square root reciprocal which are also very important arithmetic operations.

In digital systems, the opportunity must be utilized to replace division by reciprocal or square root by square root reciprocal to reduce hardware complexity. The iterative algorithms are very useful in approximating square root, square root reciprocal and reciprocal. Serial implementation of iterative algorithms is very efficient compared to other algorithms.

Chapter 11
CORDIC Algorithm

11.1 Introduction

CO-ordinate Rotation DIgital Computer (CORDIC) algorithm brings revolution in the field of computer arithmetic. Arithmetic, trigonometric, hyperbolic and many other functions can be computed by this algorithm. In order to optimize design performance, researchers are using CORDIC algorithm in many fields such as DSP, image processing, communication or in industrial sectors. In 1956, CORDIC algorithm was conceived by Jack E. Volder [74] thus sometimes it is called as Volder's algorithm. CORDIC algorithm paved the way for computing several functions by same hardware in an iterative fashion. Later several researchers polished and optimized the CORDIC algorithm for different applications.

Basic theory of CORDIC algorithm and its hardware implementation will be discussed in this chapter. This chapter also describes computation of some basic arithmetic operations using CORDIC and their architectures. Several advanced architectures are developed based on CORDIC algorithm over the last few years. These developments are also outlined here. Objective of this chapter is to familiarize the readers with theory of CORDIC and its basic applications.

11.2 Theoretical Background

This section discusses the basic theory behind the CORDIC algorithm. At the end of this discussion, the basic equations of CORDIC algorithm are formulated. Three points (x_0, y_0), (x_1, y_1) and (x_2, y_2) on a circular path in the x-y-co-ordinate system are shown in Fig. 11.1. Distance of all the points from the origin is same as the points are on the circular path. The distance is r here in this case. Here our objective is to rotate the point (x_0, y_0) towards the point (x_2, y_2) in the anticlockwise direction.

For the point (x_0, y_0) following equations can be written:

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_11

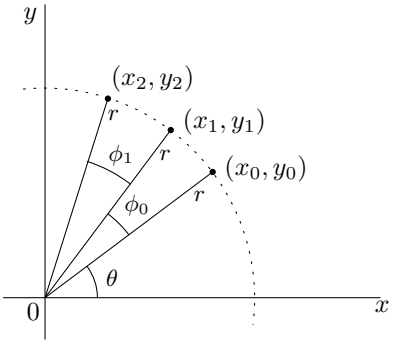


Fig. 11.1 Rotation on circular path

$$x_0 = rcos\theta \text{ and } y_0 = rsin\theta \tag{11.1}$$

and following equations can be written for the point (x_1, y_1):

$$x_1 = rcos(\theta + \phi_0) \text{ and } y_1 = rsin(\theta + \phi_0) \tag{11.2}$$

If the point (x_0, y_0) is to be rotated to the point (x_2, y_2) then total angle in anticlockwise direction is $(\phi_0 + \phi_1)$. This is achieved in two steps. First, the point (x_0, y_0) will be rotated by angle ϕ_0 in anticlockwise direction to reach the point (x_1, y_1). The equations for x_1 and y_1 are expressed in terms of points x_0 and y_0 as follows:

$$\begin{aligned} x_1 &= rcos(\theta + \phi_0) && (11.3)\\ &= rcos\theta cos\phi_0 - rsin\theta sin\phi_0 && (11.4)\\ &= x_0 cos\phi_0 - y_0 sin\phi_0 && (11.5)\\ &= cos\phi_0(x_0 - y_0 tan\phi_0) && (11.6) \end{aligned}$$

Similar equations can be written for y also

$$\begin{aligned} y_1 &= rsin(\theta + \phi_0) && (11.7)\\ &= rsin\theta cos\phi_0 + rcos\theta sin\phi_0 && (11.8)\\ &= y_0 cos\phi_0 + x_0 sin\phi_0 && (11.9)\\ &= cos\phi_0(y_0 + x_0 tan\phi_0) && (11.10) \end{aligned}$$

Initial angle to be rotated in anticlockwise direction was $(\phi_0 + \phi_1) = z_0$. The next angle to be rotated can be expressed as

$$z_1 = z_0 - \phi_0 = \phi_1 \tag{11.11}$$

In the second step, the point (x_1, y_1) will be rotated to the point (x_2, y_2). The equations for x_2 and y_2 in terms of x_1 and y_1 are expressed as follows:

$$x_2 = rcos(\theta + \phi_0 + \phi_1) \tag{11.12}$$
$$= rcos(\theta + \phi_0)cos\phi_1 - rsin(\theta + \phi_0)sin\phi_1 \tag{11.13}$$
$$= x_1 cos\phi_1 - y_1 sin\phi_1 \tag{11.14}$$
$$= cos\phi_1(x_1 - y_1 tan\phi_1) \tag{11.15}$$

Similar equations can be written for y_1 also

$$y_2 = rsin(\theta + \phi_0 + \phi_1) \tag{11.16}$$
$$= rsin(\theta + \phi_0)cos\phi_1 + rcos(\theta + \phi_0)sin\phi_1 \tag{11.17}$$
$$= y_1 cos\phi_1 + x_1 sin\phi_1 \tag{11.18}$$
$$= cos\phi_1(y_1 + x_1 tan\phi_1) \tag{11.19}$$

and the equation for the remaining angle is

$$z_2 = z_1 - \phi_1 = 0 \tag{11.20}$$

Now, the general expression of x_{i+1} and y_{i+1} is formulated after rotation by certain angle in anticlockwise direction. The general expression is

$$x_{i+1} = cos\phi_i(x_i - y_i tan\phi_i) \tag{11.21}$$
$$y_{i+1} = cos\phi_i(y_i + x_i tan\phi_i) \tag{11.22}$$
$$z_{i+1} = z_i - \phi_i \tag{11.23}$$

Here, i varies from 0 to n where n is required precision or the total number of iterations. The above expressions are written in terms of initial point x_0 and y_0 as

$$x_{i+1} = \prod_{j=0}^{n-1} cos\phi_j(.....) \tag{11.24}$$
$$y_{i+1} = \prod_{j=0}^{n-1} cos\phi_j(.....) \tag{11.25}$$

In every iteration i, a constant term is associated with the equation of x_{i+1} and y_{i+1}. The computation of x_{i+1} and y_{i+1} is done without the constant term for the shake of easy implementation. The actual results are obtained by dividing the final values (x_n and y_n) at the output stage by a constant term k_n. The expression of k_n is derived below.

The term $\prod_{j=0}^{n-1} cos\phi_j$ is a constant and it is not required to evaluate it at each iteration. The value of this constant can be evaluated as follows:

$$\prod_{j=0}^{n-1} cos\phi_j = \prod_{j=0}^{n-1} \frac{cos\phi_j}{\sqrt{cos^2\phi_j + sin^2\phi_j}} \tag{11.26}$$

$$= \frac{1}{\prod_{j=0}^{n-1} \sqrt{1 + tan^2\phi_j}} \tag{11.27}$$

$$= \frac{1}{\prod_{j=0}^{n-1} \sqrt{1 + 2^{-2j}}} = \frac{1}{k_n} \tag{11.28}$$

Here, $tan\phi_j = 2^{-j}$. Every iteration corresponds to an incremental rotation thus angle representation is very important in this case. Two's complement data representation technique is very common to represent angles. The most used angle rotation for n-bit resolution is

$$-\pi \quad \frac{\pi}{2^1} \quad \frac{\pi}{2^2} \quad \frac{\pi}{2^3} \quad \frac{\pi}{2^4} \quad \frac{\pi}{2^5} \cdots\cdots \frac{\pi}{2^{n-1}} \tag{11.29}$$

Any angle can be represented by this technique. Two MSB bits represent the location of the co-ordinate in any quadrant. For example, 45° angle can be represented in 16-bit data format as 16'b0010000000000000, which is in the first quadrant. In this format, the MSB bit indicates that the elementary angle is positive or negative.

The angles are represented in terms of power of 2. Equations (11.21)–(11.23) become simpler by putting $tan\phi_i = 2^{-i}$ and avoiding the constant term as

$$x_{i+1} = x_i - y_i 2^{-i} \tag{11.30}$$

$$y_{i+1} = y_i + x_i 2^{-i} \tag{11.31}$$

$$z_{i+1} = z_i - tan^{-1} 2^{-i} \tag{11.32}$$

The evaluation of the above equations is easy now as it involves division by power of 2 and addition or subtraction operations. Division by power of 2 is performed by wired shift method which needs no hardware. Only hardwares required to evaluate the above equations are one adder and two subtractors.

The above equations are established assuming that rotation is done only in the anti-clockwise direction. But in the actual scenario, the target angle rotation is achieved by successive incremental rotations until the difference between the target angle and achieved angle becomes zero. These rotations are called as micro-rotations and they can be anticlockwise or clockwise. These situations are described in Fig. 11.2.

The direction of the next micro-rotation is decided based on the sign of angle difference (z_i). A new parameter σ is introduced which is evaluated as

$$\sigma_i = \begin{cases} 1, & \text{if } z_i \geq 0 \\ -1, & \text{otherwise} \end{cases} \tag{11.33}$$

Another important point is that these micro-rotations do not follow the circular path because a constant term $cos\phi_i$ is associated with each micro-rotation. To make it

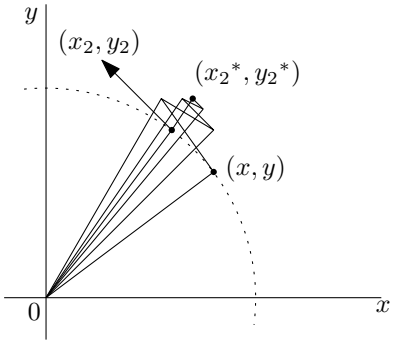


Fig. 11.2 CORDIC micro-rotations

simple, the constant term is not calculated in each iteration. Computation is carried out without the constant term and computed results are divided by the constant term after the final iteration. Thus the modified equations are

$$x_{i+1} = x_i - \sigma_i y_i 2^{-i} \tag{11.34}$$

$$y_{i+1} = y_i + \sigma_i x_i 2^{-i} \tag{11.35}$$

$$z_{i+1} = z_i - \sigma_i tan^{-1} 2^{-i} \tag{11.36}$$

The equations for rotating a point (x_0, y_0) by any angle on circular path are formulated at the end of the above discussion. This concludes that CORDIC can be used to rotate a point (x_0, y_0) in any direction.

The above discussion is restricted to rotation by maximum 90° in any direction. But if we want to rotate a point by an angle more than 90° then a different mechanism has to be adopted. Rotation by other angles can be realized in terms of rotation by 90° as per trigonometric rules. Table 11.1 explains this procedure. If any point lies in 2nd or 3rd quadrant then rotation is done by assuming that it lies in the 4th or 1st quadrant, respectively. A conditional sign change is required at the output to get the actual result. Figure 11.3 explains this situation. In order to rotate any point that lies in the 2nd quadrant, (initial angle θ lies between $\pi/2 \leq \alpha < \pi$) the point is assumed

Table 11.1 Quadrant transformation and sign change for CORDIC

Before transformation			After transformation		
$\theta_{15}\theta_{14}$	Range	Quadrant	$\theta_{15}\theta_{14}$	Quadrant	Sign change
00	$0 \leq \theta < \pi/2$	1st	00	1st	No
01	$\pi/2 \leq \theta < \pi$	2nd	11	4th	Yes
10	$-\pi \leq \theta < -\pi/2$	3rd	00	1st	Yes
11	$-\pi/2 \leq \theta < 0$	4th	11	4th	No

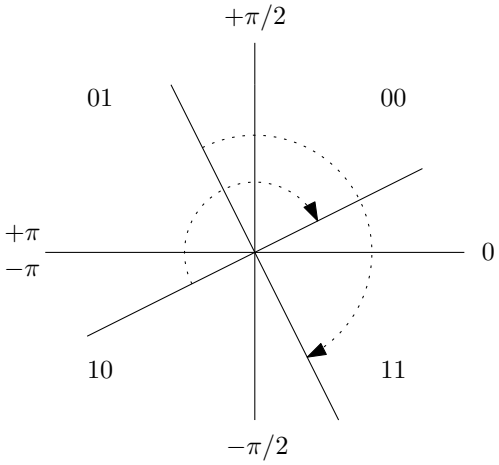


Fig. 11.3 Quadrant transformation for rotation on circular path [11]

to be in the 4th quadrant and rotation is done as if the point lies in 4th quadrant. Output of the final stage is just inverted to obtain actual result.

11.3 Vectoring Mode

In another situation, we have to find the angle where the co-ordinates are given. This situation is opposite to the above-mentioned rotation mode and this is called the vectoring mode. In this mode, one of the co-ordinates is nullified and the angle between them is found at the output. The general equations for this mode are shown below:

$$x_{i+1} = x_i - \sigma_i y_i 2^{-i} \tag{11.37}$$

$$y_{i+1} = y_i + \sigma_i x_i 2^{-i} \tag{11.38}$$

$$z_{i+1} = z_i - \sigma_i tan^{-1} 2^{-i} \tag{11.39}$$

Generally, the co-ordinate y is nullified and the final expressions are $x_n = k_n\sqrt{x^2 + y^2}$ and $z_n = tan^{-1}(y/x)$. So, at the output we get magnitude and phase. This mode is useful to compute the magnitude of a complex number or to find the phase angle between the two co-ordinates. The value of the constant term is same as mentioned above for the rotation mode. The equation of σ_i is different here and it is mentioned below:

$$\sigma_i = \begin{cases} 1, & \text{if } y_i \leq 0 \\ -1, & \text{otherwise} \end{cases} \tag{11.40}$$

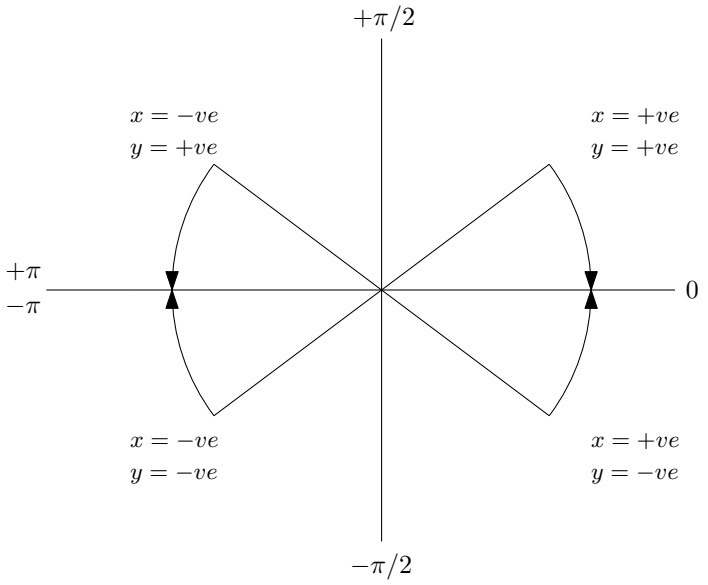


Fig. 11.4 Vectoring operation [11]

Table 11.2 Sign change for vectoring operation

$x_{15}y_{15}$	Rotation	Sign change (O/P)
00	Clockwise	No
01	Anticlockwise	No
10	Anticlockwise	Yes
11	Clockwise	Yes

The co-ordinates x and y can be positive or negative. The sign of the final output and the computed angle is to be modified accordingly at the output based on their sign. Figure 11.4 and Table 11.2 explain this situation.

11.3.1 Computation of Sine and Cosine

The general expression of the CORDIC algorithm is expressed as

$$x_{i+1} = x_i cos\phi_i - y_i sin\phi_i \tag{11.41}$$

$$y_{i+1} = y_i cos\phi_i + x_i sin\phi_i \tag{11.42}$$

$$z_{i+1} = z_i + \sigma_i tan^{-1}2^{-i} \tag{11.43}$$

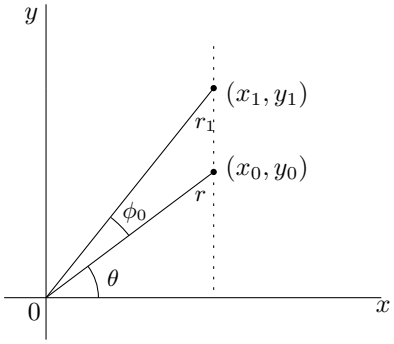


Fig. 11.5 Rotation on linear path

In rotation mode, if initially input y is set to zero and initial value of input x is set to $\frac{1}{k_n}$ then

$$x_1 = cos\phi_0 \tag{11.44}$$
$$y_1 = sin\phi_0 \tag{11.45}$$
$$z_1 = \phi + \sigma_i tan^{-1}2^0 \tag{11.46}$$

In the second iteration

$$x_2 = cos\phi_0 cos\phi_1 - sin\phi_0 sin\phi_1 = cos(\phi_0 + \phi_1) \tag{11.47}$$
$$y_2 = sin\phi_0 sin\phi_1 + cos\phi_0 sin\phi_1 = sin(\phi_0 + \phi_1) \tag{11.48}$$
$$z_2 = z_1 + \sigma_i tan^{-1}2^{-1} \tag{11.49}$$

This way final outputs of CORDIC (x_n and y_n) hold the value of $cos\phi$ and $sin\phi$, respectively. The initial angle is ϕ. The value of σ_i is evaluated as per value of z_i.

11.4 Linear Mode

The co-ordinates can also be rotated on linear path. Rotation on linear path is shown in Fig. 11.5. Final equations of the co-ordinates are obtained by the same way as it was previously done for circular path. The final equations for the linear mode are

$$x_{i+1} = x_{in} \tag{11.50}$$
$$y_{i+1} = y_i + \sigma_i x_i 2^{-i} \tag{11.51}$$
$$z_{i+1} = z_i - \sigma_i 2^{-i} \tag{11.52}$$

One of the benefits of rotating on linear path is that the scaling factor $k_n = 1$. Thus error due to scaling factor multiplication is eliminated in this case. Thus, linear

version of CORDIC is sometimes used to compensate the constant multiplication factor involved in circular operation. Linear version of CORDIC also has rotation and vectoring mode. Two important operations can be performed in this version which are multiplication and division.

11.4.1 Multiplication

Multiplication is performed in rotation mode and the value of σ_i is determined by the same equation as mentioned above for rotation mode. The CORDIC produces final result as

$$x_n = x_{in} \tag{11.53}$$

$$y_n = y_{in} + z_{in} * x_{in} \tag{11.54}$$

Initially y_0 is equal to zero, z_0 holds the value of multiplier (z_{in}), x_0 is set to x_{in} and y_n holds the final multiplication result. Multiplication operation by CORDIC has some limitations compared to common digital multipliers. The width of the multiplication result is bounded by the width of x, y and z. This is the disadvantage of this method. For 18-bit CORDIC, multiplication result will also be limited by 18 bit.

11.4.2 Division

Division is performed in vector mode and the value of σ_i is determined by the same equation as mentioned above for vectoring mode. The final expression for division using CORDIC is

$$z_n = z_0 + \frac{y_{in}}{x_{in}} \tag{11.55}$$

Initially, z_0 is set to zero and z_n holds the result of division operation. The major limitation of performing division operation by CORDIC is that the final result is bounded to be fit in the data width of input operands. But on the advantage side, compared to the other fast dividers CORDIC divider has low latency and consumes less hardware. Accuracy is a concern where high accuracy is needed.

11.5 Hyperbolic Mode

Rotation of co-ordinates can also be done along a hyperbolic path similar to circular and linear path. This is an extension of the CORDIC algorithm. This type of rotation mechanism enables computation of several other functions using CORDIC.

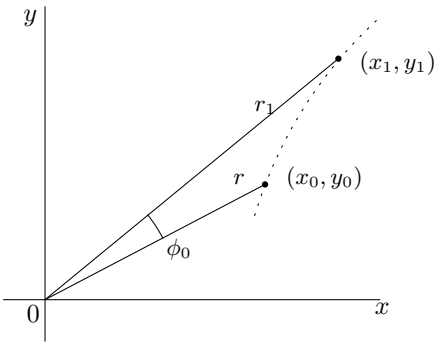


Fig. 11.6 Rotation on hyperbolic path

The equations can be derived the same way as they are derived in case of circular path. Figure 11.6 explains the rotation mechanism on hyperbolic path. The general expressions for CORDIC for micro-rotations along hyperbolic path are

$$x_{i+1} = x_i + \sigma_i y_i 2^{-i} \tag{11.56}$$

$$y_{i+1} = y_i + \sigma_i x_i 2^{-i} \tag{11.57}$$

$$z_{i+1} = z_i - \sigma_i tanh^{-1} 2^{-i} \tag{11.58}$$

The operation in hyperbolic version of CORDIC is slightly different from the other two versions. In hyperbolic version, computation starts from iteration 1 as the value $tanh^{-1}2^0 = \infty$. In hyperbolic version, convergence is an issue. Some iterations are repeated to make sure that output will converge. The repetition of the iterations is done by repeating iterations 4, 13, 40, ..., i, $3i + 1$, ... The scale factor in an iteration i is

$$k_h(i) = (1 - 2^{-2i})^{1/2} \tag{11.59}$$

11.5.1 Square Root Computation

To compute square root operation hyperbolic version of CORDIC is used. In hyperbolic CORDIC, the final equations of x and y in vectoring mode are

$$x_n = k_h \sqrt{(x^2 - y^2)} \tag{11.60}$$

In the above equation, if initial value of $x = a + 1/4$ and $y = a - 1/4$ then it converges to

$$x_n = k_h \sqrt{(4.a.1/4)} = k_h . \sqrt{a} \tag{11.61}$$

It is clear from the above discussion that the micro-rotations in CORDIC algorithm can be done along circular, linear and hyperbolic path. There exists two modes, rotation and vectoring, for each case. Thus generalized equations for radix-2 CORDIC algorithm can be written as

$$x_{i+1} = x_i - m\sigma_i y_i 2^{-i} \tag{11.62}$$

$$y_{i+1} = y_i + \sigma_i x_i 2^{-i} \tag{11.63}$$

$$z_{i+1} = z_i - \sigma_i e_i \tag{11.64}$$

Here, a new variable m is introduced which is 1 for circular, 0 for linear and -1 for hyperbolic. The value e_i is $tan^{-1}2^{-i}$ for circular, 2^{-1} for linear and $tanh^{-1}2^{-i}$ for hyperbolic. In addition to multiplication, division or square root, several other functions can be computed using CORDIC. Computation of all the different functions using CORDIC algorithm is summarized in Table 11.3.

11.6 CORDIC Algorithm Using Redundant Number System

Over the past few years CORDIC algorithm has been modified by several researchers to improve its performance. These modifications to the CORDIC are related to either efficient compensation technique or reducing the latency involved in computation. The above discussion on CORDIC was for non-redundant conventional radix-2 number system. Researchers also have used redundant number system to reduce the latency. CORDIC algorithm using redundant number system is discussed below.

11.6.1 Redundant Radix-2-Based CORDIC Algorithm

The redundant radix-2 CORDIC [52] uses redundant arithmetic to select the value of σ_i. The value of the σ_i is chosen from the set $\{-1, 0, 1\}$ instead of the set $\{-1, 1\}$. The value of $\sigma_i = 0$ indicates that no rotation is to be performed. The use of redundant arithmetic enables to reduce some of the iterations. But implementation of this kind of algorithm is not simple. The sign of the angle cannot be easily determined by looking the MSB bit. This is because in redundant arithmetic MSB can be one even for positive numbers. Thus value of σ_i is determined by evaluating few bits in the MSB side. This increases hardware complexity. Also, the computation of constant factor is data dependent as no rotation is performed if $\sigma_i = 0$. This makes computation of k at each iteration.

Table 11.3 Evaluation of different functions using CORDIC Algorithm

m	Mode	Initialization	Output
1	Rotation	$x_0 = x_{in}$	$x_n = k_m(x_{in}cos\theta - y_{in}sin\theta)$
		$y_0 = y_{in}$	$y_n = k_m(y_{in}cos\phi + x_{in}sin\phi)$
		$z_0 = \phi$	$z_n = 0$
		$x_0 = 1/k_n$	$x_n = cos\phi$
		$y_0 = 0$	$y_n = sin\phi$
		$z_0 = \phi$	$z_n = 0$
		$x_0 = 1$	$x_n = \sqrt{1 + a^2}$
		$y_0 = a$	$y_n = sin\phi$
		$z_0 = \pi/2$	$z_n = 0$
1	Vectoring	$x_0 = x_{in}$	$x_n = k_m(x_{in}sin(x_0)(x_{in}^2 + y_{in}^2)^{1/2})$
		$y_0 = y_{in}$	$y_n = 0$
		$z_0 = 0$	$z_n = tan^{-1}(y_{in}/x_{in})$
0	Rotation	$x_0 = x_{in}$	$x_n = x_{in}$
		$y_0 = y_{in}$	$y_n = y_{in} + x_{in}.z_{in}$
		$z_0 = z_{in}$	$z_n = 0$
0	Vectoring	$x_0 = x_{in}$	$x_n = x_{in}$
		$y_0 = y_{in}$	$y_n = 0$
		$z_0 = z_{in}$	$z_n = z_{in} + y_{in}/x_{in}$
–1	Rotation	$x_0 = x_{in}$	$x_n = k_m(x_{in}cosh\phi - y_{in}sinh\phi)$
		$y_0 = y_{in}$	$y_n = k_n(y_{in}cosh\phi + x_{in}sinh\phi)$
		$z_0 = \phi$	$z_n = 0$
		$x_0 = 1/k_n$	$x_n = cosh\phi$
		$y_0 = y_{in}, z_0 = \phi$	$y_n = sinh\phi, z_n = 0$
		$x_0 = a$	$x_n = ae^{\phi}\phi$
		$y_0 = a, z_0 = \phi$	$y_n = ae^{\phi}\phi, z_n = 0$
–1	Vectoring	$x_0 = x_{in}$	$x_n = k_n(x_{in}sin(x_0)(x_{in}^2 - y_{in}^2)^{1/2})$
		$y_0 = a, z_0 = \phi$	$z_n = \phi + tanh^{-1}(y_{in}/x_{in}), y_n = 0$
		$x_0 = a$	$x_n = \sqrt{a^2 - 1}$
		$y_0 = 0$	$y_n = 0, z_n = coth^{-1}a$
		$x_0 = a$	$x_n = 2\sqrt{a}$
		$y_0 = 0$	$y_n = 0, z_n = 0.5ln(a)$
		$x_0 = a + b$	$x_n = 2\sqrt{ab}$
		$y_0 = a - b$	$y_n = 0, z_n = 0.5ln(a/b)$

11.6.2 Redundant Radix-4-Based CORDIC Algorithm

The speed of the CORDIC algorithm can be improved by reducing the number of iterations. This can be achieved by using higher Radix. The CORDIC equations for Radix-4 are

$$x_{i+1} = x_i - \sigma_i 4^{-i} y_i \tag{11.65}$$

$$y_{i+1} = \sigma_i 4^{-i} x_i + y_i \tag{11.66}$$

$$w_{i+1} = w_i - tan^{-1}(\sigma_i 4^{-i}) \tag{11.67}$$

where $\sigma_i \in \{-2, -1, 0, 1, 2\}$. The final x- and y-co-ordinates are scaled by

$$k = \prod_{i \geq 0} k_i = \prod_{i \geq 0} (1 + \sigma_i^2 4^{-2i})^{1/2} \tag{11.68}$$

Here, the scale factor k depends on the values of σ_i, and hence has to be computed in every iteration. The range of k is (1, 2.52) for radix-4 CORDIC. In this CORDIC, the direction of rotation is computed based on the estimated value of w_i [6]. The w path involves the computation of estimated w_i and evaluation of selection function to determine σ_i resulting in increase of the iteration delay compared to that of radix-2. However, the number of iterations required for radix-2 CORDIC can be halved by employing the radix-4 CORDIC algorithm. It is sufficient to compute the constant for $n/4$ iterations.

11.7 Example of CORDIC Iteration

An example to understand the evaluation of the CORDIC micro-rotations is given below. Objective of this example is to compute the magnitude of a complex number whose real value is x and imaginary value is y. This function can be evaluated by rotating co-ordinates along the circular path in vectoring mode. Table 11.4 shows iteration-wise evaluation. Initially $x_0 = 5$, $y_0 = 3$, $z_0 = 0$ and thus $\sigma = -1$. The final output (x_n) is obtained by dividing 9.6022 by k_n which is equal to 1.64676. So, the magnitude value is 5.8310 and value of the angle between them is 30.9641.

11.8 Implementation of CORDIC Algorithms

CORDIC algorithm paved the way for new research ideas in the area of digital arithmetic. CORDIC is finding its application in every fields. CORDIC shows that same hardware can be used for computing many functions. Several researchers have reported different hardware implementations of CORDIC. But generally there are

Table 11.4 Iteration-wise CORDIC for computation of absolute of x and y

x_i	y_i	z_i	i	σ
8	–2	45	0	1
9	2	18.4349	1	–1
9.5	–0.25	32.4712	2	1
9.5313	0.9375	25.3462	3	–1
9.5898	0.3418	28.9225	4	–1
9.6005	0.0421	30.7124	5	–1
9.6012	–0.1079	31.6076	6	1
9.6020	–0.0329	31.1600	7	1
9.6022	0.0046	30.9362	8	–1
9.6022	–0.0141	31.0481	9	1
9.6022	–0.0048	30.9921	10	1
9.6022	0	30.9641	11	1

two kinds of hardware of CORDIC. First one is serial architecture and second one is parallel architecture.

11.8.1 Parallel Architecture

The parallel implementation of CORDIC [2] shown in Fig. 11.7 is designed with 16-bit fixed point data width. It supports both rotation and vector modes. Each stage corresponds to a micro-rotation and is having three add/sub units which performs addition or subtraction based on value of signal σ. $ph1$ block (Fig. 11.8) computes the value of σ. Add/sub blocks perform subtraction when σ is 1. Different angles for micro-rotation are fed to each stage. The rsh blocks are responsible for shifting data to the right side. These blocks perform wired shifting which is described in Chap. 3 in detail. For examples, $rsh1$ block shifts input data to the right side by 1 bit. Multiplication by a constant factor is achieved by a scale factor. The scale block divides output of the last stage by k_n by constant multiplication technique. This block is also illustrated in Chap. 3. An inversion stage is added at the last stage to invert the outputs of the scale block conditionally. $ph2$ block (Fig. 11.8) generates the control signal for the add/sub units at the inversion stage.

11.8.2 Serial Architecture

A CORDIC architecture can be bit-serial or word-serial. Both types of serial architectures are based on computing each step by the same hardware and thus cannot

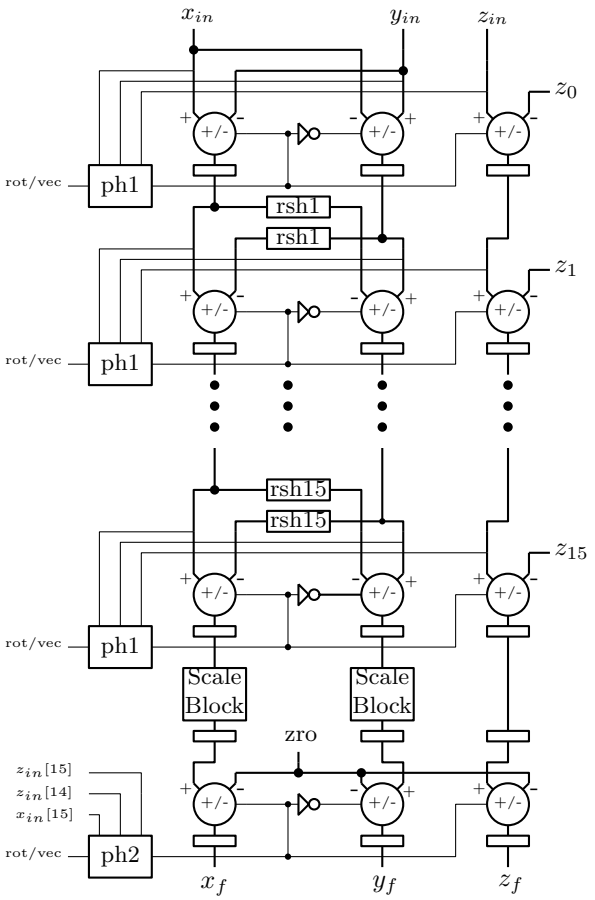
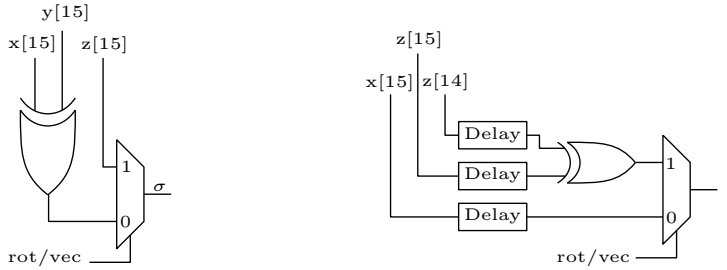


Fig. 11.7 Parallel implementation of CORDIC algorithm



(a) Schematic for the Ph1 block. (b) Schematic for the Ph1 block..

Fig. 11.8 Schematic for the Ph1 and Ph2 blocks

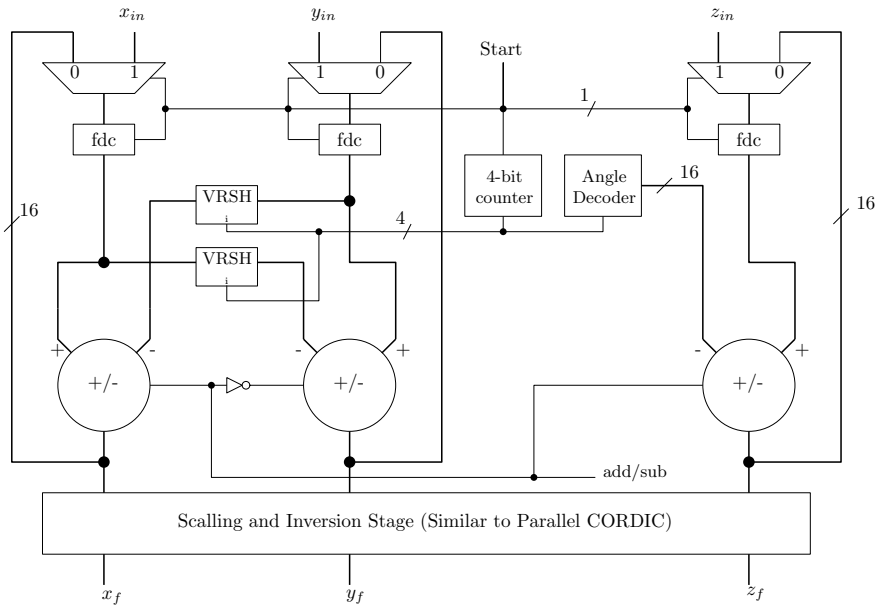


Fig. 11.9 Serial architecture of radix-2 non-redundant CORDIC

support pipelining. Bit-serial architectures [74] are very slow and thus not suitable for high-speed applications. Word-serial architectures [77] are also mainly used in low-frequency applications. But these architectures are hardware efficient than the parallel architecture.

The word-serial architecture of CORDIC is depicted in Fig. 11.9. The serial architecture is similar to any one stage of the parallel CORDIC architecture. Scaling and inversion stages are similar to those which were in the parallel CORDIC architecture. This architecture is of 16 bit. The different angles for micro-rotations are pre-stored in an LUT. Total number of 16 angles are stored in that LUT and a 4-bit counter is used to fetch those angles. The VRSH block stands for variable right shift and used to shift the input operands by variable number of bits. Details of this block can be found in Chap. 3. The fdc block is a controlled register block which stores data based on an enable signal. The control signal add/sub is similar to that of parallel CORDIC architecture. Data-width and precision decide number of iterations of the serial CORDIC block.

11.8.3 Improved CORDIC Architectures

Over the past few years many improvements are proposed in literature to improve the latency of the basic non-redundant radix-2 CORDIC architecture without increasing

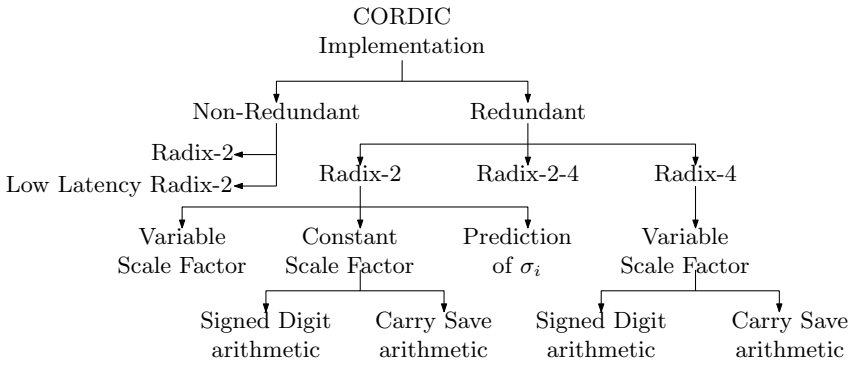


Fig. 11.10 Serial architecture of radix-2 non-redundant CORDIC

the hardware and without hampering the accuracy. CORDIC architectures can be classified as non-redundant architectures and redundant architectures. Redundant architectures also can be classified as redundant radix-2 architecture and redundant radix-4 architecture. The classification of the CORDIC architectures is shown in Fig. 11.10.

Most obvious modification to the non-redundant radix-2 CORDIC architecture is proposed in [3, 8] to reduce the latency. This modification is based on linear approximation of the rotation when the angle is too small. For example, if the angle θ is too small then $sin\theta \approx \theta$ and $cos\theta \approx 1$. Here, evaluation of the iterations up to $(n/2+1)$th iteration follows the basic CORDIC equation. But in the $(n/2+1)$th step the iterations are evaluated as

$$x_n = x_{(n/2+2)} = k_{(n/2+2)}(x_{(n/2+2)} - \phi_r y_{(n/2+2)}) \tag{11.69}$$

$$y_n = y_{n/2+2} = k_{(n/2+2)}(\phi_r x_{(n/2+2)} + y_{(n/2+2)}) \tag{11.70}$$

where $\phi_r = z_{(n/2+1)}$ and $k_{(n/2+2)}$ is the constant according to the $(n/2+1)$th iteration. Thus this implementation needs only $(n/2+2)$ iterations compared to n iterations of conventional CORDIC algorithm.

11.8.3.1 Constant Scale Factor Redundant CORDIC Using SD Arithmetic

In redundant CORDIC architectures, primary objective is to keep the multiplication factor constant so that additional hardware does not increase. There are some architectures proposed in literature which have constant multiplication factor and use SD representation. These CORDIC architectures are double rotation method [70], correcting rotation method [70], branching method [24] and double step branching method [55]. The implementation of redundant CORDIC with constant scale factor

using signed arithmetic results in an increase in the chip area and latency by at least 50% compared to redundant radix-2 CORDIC.

11.8.3.2 Constant Scale Factor Redundant CORDIC Using CS Arithmetic

Due to the disadvantages of SD arithmetic operation in improving the efficiency of the Constant Scale Factor Redundant CORDIC architectures, Carry Save arithmetic is used in some of the architectures proposed in literature. These architectures are Low Latency Redundant CORDIC [71] and Differential CORDIC [23] architectures.

11.8.3.3 Higher Radix Redundant CORDIC Architectures

Higher Radix CORDIC architectures reduce the number of iterations and thus improve latency and throughput. The generalized CORDIC algorithm for any Radix and its pipeline implementation using SD arithmetic for $m = 1$ in rotation mode are presented in [18]. A redundant radix 2-4 CORDIC architecture is presented in [5] which combines non-redundant radix2 with redundant radix-4 algorithm. A redundant radix-4 CORDIC algorithm is proposed using CS arithmetic in [7].

11.8.3.4 Direction Prediction-Based CORDIC Architectures

In the CORDIC theory, the angle ϕ can be represented in terms of set of elementary angles ϕ_i as

$$\phi = \sigma_0\phi_0 + \sigma_1\phi_1 + \sigma_2\phi_2 + ... + \sigma_{n-1}\phi_{n-1} \tag{11.71}$$

where $\phi_i = tan^{-1}(2^{-i})$ and $\sigma_i = \{-1, 1\}$ satisfying the convergence theory

$$\phi_i - \sum_{j=i+1}^{n-1} \phi_j < \phi_{n-1} \tag{11.72}$$

The value of σ_i is evaluated sequentially. The speed of the CORDIC architectures can be improved if the sequential computation of σ_i can be avoided. Thus some architectures are reported in literature based on predicting the value of σ_i in parallel. The architectures based on pre-computation of σ_i are Low Latency Radix-2 CORDIC [71], P-CORDIC [38], Hybrid CORDIC Algorithm [78], Flat CORDIC [28], Para-CORDIC [73] and Semi-Flat CORDIC [33].

11.9 Application

CORDIC algorithm has many applications in developing digital hardware for the algorithms. Few important applications are outlined below:

- CORDIC-based calculator is one example where CORDIC algorithm is used to calculate all the mathematical operations.
- In application-specific applications, CORDIC is used to approximate arithmetic functions. Division or finding reciprocal of a number is example of these functions.
- Rotation of co-ordinates can be performed using CORDIC. This property is used to develop hardware for FFT, DCT, Wavelet, Curve-let, etc.
- CORDIC can be used to solve linear equations. Performing QR decomposition using CORDIC algorithm is one example. This is based on finding magnitude of a complex number using CORDIC algorithm.
- Sinusoidal signals can be generated easily by CORDIC. This helps developing CORDIC-based Digital Controlled Oscillator (DCO).

11.10 Conclusion

In this chapter, the theoretical background of the CORDIC is discussed in detail. Further the non-redundant radix-2 CORDIC architectures in circular, linear and hyperbolic paths in $x - y$-co-ordinate are discussed. Evaluation of some of the major functions are discussed in detail. Architecture for word-serial and parallel CORDIC is explained in detail. In addition to the non-redundant radix-2 architectures based on two's complement arithmetic, many other CORDIC algorithms are discussed. The redundant radix-2 and radix-4 CORDIC algorithms aim to reduce the latency of the CORDIC by reducing the iterations. A brief overview on the different CORDIC architectures is also presented in this chapter.

Chapter 12
Floating Point Architectures

12.1 Introduction

In the previous chapters, we have discussed the fixed point architectures. The majority of FPGA-based architectures are fixed point based. In the fixed point architectures, the number of bits reserved for integer and fractional part is fixed. Thus there is a limitation in representing a wide range of numbers. This limitation results truncation error in the output. Accuracy can be increased by increasing the word length but this in turn increases hardware complexity. At some point further increase in word length cannot be tolerable.

A solution to this problem is to use floating point representation as discussed in Chap. 1. But the usage of floating point representation in real-time implementations is limited. The reason behind this fact is that implementation of floating point architectures is complex. Due to higher complexity, floating point architectures are not suitable for rapid prototyping. On the other hand, floating point architectures provide better accuracy than fixed point architectures.

The objective of this chapter is to discuss the basics of floating point representation and the basic architectures for floating point arithmetic operation. A floating point number can be represented in binary as shown in Fig. 12.1 So, a floating point number has three fields, viz., sign field (S), exponent field (E) and mantissa (M). The exponent field is added to a bias component to differentiate between negative and positive exponents. The decimal equivalent of this representation is

$$S.M.2^{E+bias} \tag{12.1}$$

In this chapter, first the standards in floating point representation are discussed and then different architectures for floating point arithmetic operation are presented.

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_12

Fig. 12.1 IEEE floating point data format

12.2 Floating Point Representation

There are two standard formats according to IEEE 784 which are single precision and double precision data format. IEEE single precision data format for floating point numbers is shown in Fig. 12.2. The double precision format accommodates even higher range of numbers. Double precision data format is shown in Fig. 12.3 In the floating point number representation, there are different representations for the same number and there is no fixed position for the decimal point. There may be loss of precision for a fixed number of digits in performing arithmetic operations. But for a fixed number of digits, the floating point representation covers a wider range of values compared to a fixed point representation. Data represented in this format are classified into five groups.

- Normalized numbers,
- Zeros,
- Subnormal (denormal) numbers,
- Infinity and not-a-number (NAN).

The exponent field is calculated by adding a bias component. The value of the bias is $(2^8 - 1)$ in single precision format. The valid range of exponents for single precision is from 1 to 254. The value of 0 and 255 are reserved for special numbers. This gives the range of data from 2^{127} to 2^{-126}. The interpretation of these numbers is shown in Table 12.1. The maximum number that can be represented in the single precision format is

$$0\_11111110\_111111..... = 1.99999988 \times 2^{127} \approx 3.4028 \times 10^{38}$$

Fig. 12.2 IEEE floating point single precision data format

Fig. 12.3 IEEE floating point double precision data format

Table 12.1 Data interpretation in floating point representation

Single precision		Data types
Exponent	Mantissa	
0	0	±0
0	Nonzero	± Subnormal number
1–254	Anything	± Normalized number
255	0	± ∞
255	Nonzero	NaN

and the minimum number that can be represented in this format is

$$0\_00000001\_000000..... = 1 \times 2^{-126} \approx 1.1754 \times 10^{-38}$$

Both the numbers can be negative or positive depending on the status of sign bit. The above range of data is shown for normalized data only. If they are subnormal then the minimum value will be

$$0\_00000000\_0000.....001 = 1 \times 2^{-126-23} = 1 \times 2^{-149} \approx 1.4012 \times 10^{-45}$$

Maximum subnormal number will be

$$0\_00000000\_1111.....111 = 0.99999988 \times 2^{-126} \approx 1.1754 \times 10^{-38}$$

Due to the presence of the subnormal numbers, there are 2^{23} numbers within the range $[0.0, 1.0 \times 2^{-126})$. The smallest difference between two normalized numbers is 2^{-149}. This is same as the difference between any two consecutive subnormal numbers. The largest difference between two consecutive normalized numbers is 2^{104}. This implies that the numbers in floating point representation are non-uniformly distributed. Also there are two zeros (±) in the IEEE representation. The similar type of analysis can be done for IEEE double precision format.

The condition overflow occurs when the true result of an arithmetic operation is too large that it cannot be represented with the given data format. On the other hand, underflow is a condition when the number is too small to be represented. The overflow condition in system design cannot be ignored, where the underflow condition can be checked by checking if the result is zero or not. The floating point architecture should be designed in such a way that overflow never occurs.

The data formats according to IEEE 754 standards are suitable for general purpose controllers. The single precision or double precision formats may lead to costlier designs. Thus designers should go for trade-offs between high hardware complexity and high accuracy. Thus any word length can be used for implementing a system depending on requirement of accuracy. In this chapter, 16-bit word length is con-

sidered for designing the architectures. In this 16-bit format, 4 bits are reserved for biased exponent and 11 bits are reserved for mantissa part.

12.3 Fixed Point to Floating Point Conversion

Generally the FPGA-based architecture or ASIC-based implementations are fixed point arithmetic based. But the majority of dedicated controllers are floating point based. It may require to interface a FPGA to a micro controller and then a fixed point to floating point conversion unit will be required. In order to discuss the floating point architectures, first the scheme for conversion from fixed point to floating point is discussed. The steps of converting a fixed point number to a floating point number are

1. Invert the number if the number is negative or if the MSB is logic 1.
2. Count the leading zeros present in the number.
3. Value of the mantissa is computed by left shifting the number according to the leading zeros present in the number.
4. Subtract the leading zero count from $(m - 1)$ where m represents the number of bits which are reserved for integer including the MSB bit.
5. Add the result to the bias value to get the exponent.
6. Sign of the fixed point number is the sign of the floating point number.

Example: Fixed Point to Floating Point Conversion

- The input fixed point number is $a = 001000\_0010000000$ whose decimal value is 8.125 for m equal to 6 bits.
- As this number is positive so no inversion is required
- There are two leading zero present in the number so it is left shifted by two bits. The result of shifting is 100000_1000000000.
- The value of the mantissa is $M = 00000\_100000$ by choosing 11 bits from MSB and excluding the MSB.
- The leading zero count is subtracted from $(m - 1)$ and the result is $(5 - 2) = 3$.
- The result is added to the bias to get the exponent $(E = (7 + 3) = 10)$.
- Thus the floating point number is 0_1010_00000100000.

An simple scheme of fixed point to floating conversion is shown in Fig. 12.4. Here two 4-bit adder/subtractor blocks are used to determine the exponent and one 16-bit adder/subtractor is used for inversion if necessary. The VLSH block stands for variable left shift according to the leading zero count. The leading zeros are counted by a block called leading zero counter. This block is discussed in the next section in detail. The operation of the architecture can be understood easily by following the steps mentioned above. In the mantissa computation path, 11 bits are taken as mantissa to form a 16-bit floating point output. This 16-bit representation cannot

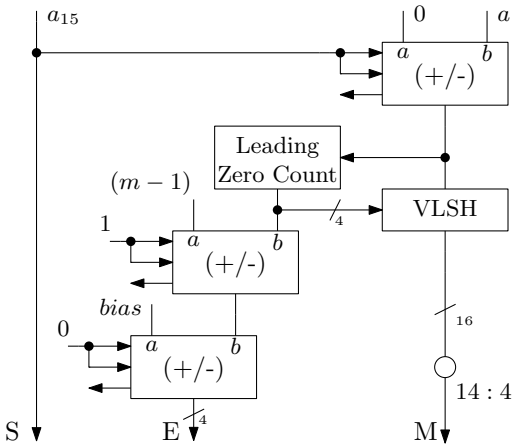


Fig. 12.4 A basic architecture for fixed point to floating point conversion

represent all the numbers which are represented in the 16-bit fixed point representation. For example, in the 16-bit format the numbers which are less than 2^{-4} cannot be represented for exponent value of 4. The circle at VLSH block output line stands for concatenation operation.

12.4 Leading Zero Counter

In a binary number, leading zeros are the zero digits in the most significant positions of data, up to the position in which the first one is present. Leading zero counter is a very important combinational circuit in designing the floating point architectures to do the normalization operation.

Here a design of a 16-bit leading zero counter is presented. A simple modular architecture is presented in [51]. In [51], authors designed higher order leading zero counter using 4-bit leading zero counters. In a binary number (x), the leading zero count (q) varies from 0 to 3. If all the bits are zero then it generates a signal (a) to indicate that the number is zero. The truth table for 4-bit leading zero counter (LZC-4) is shown in Table 12.2. Using the K-map optimization technique the Boolean expression for the outputs of LZC-4 can be obtained and these are

$$q_0 = \overline{x_3}.x_2 + \overline{x_3}.\overline{x_1} \tag{12.2}$$

$$q_1 = \overline{x_3 + x_2} \tag{12.3}$$

The architecture of the LZC-4 is shown in Fig. 12.5. It outputs the leading zero count $z = \{q_1, q_0\}$ and also outputs a signal a which indicates that all the bits are zero. Higher order leading zero counters can be designed using the basic LZC-4 blocks. The four bits are together called as nibble. A 16-bit binary number has four nibbles.

Table 12.2 Truth table for 4-bit leading zero counter

x_3	x_2	x_1	x_0	q_1	q_0
0	0	0	0	X	X
0	0	0	1	1	1
0	0	1	X	1	0
0	1	X	X	0	1
1	X	X	X	0	0

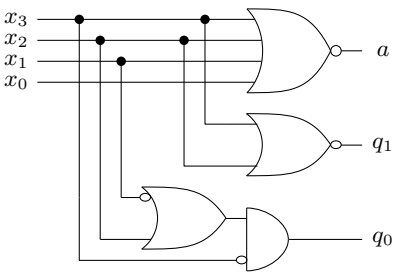


Fig. 12.5 A basic architecture for 4-bit leading zero counter

Table 12.3 Truth table for 16-bit leading zero counter

a_0	a_1	a_2	a_3	q_3	q_2	$\{q_1, q_0\}$
0	X	X	X	0	0	z_0
1	0	X	X	0	1	z_1
1	1	0	X	1	0	z_2
1	1	1	0	1	1	z_3

The outputs a_i (All zero) and z_i (zero count) are generated by the ith nibble from the MSB side. Lets consider to design an 8-bit leading zero counter. In this case, the count will vary from 0 to 7. If $a_0 = 0$, the zero count value depends on z_0 and if $a_0 = 1$ the overall zero count value is $\{a_0, z_1\}$. The truth table for 16-bit leading zero counter (LZC-16) is shown in Table 12.3.

The LZC-16 is designed using the basic LZC-4 block. The upper bits of the counter are evaluated by a block which takes the inputs a_i from the LZC-4 blocks. This block is called as leading zero encoder (LZE-4) which is shown in Fig. 12.6. The logical expressions for the outputs of this block are decided by Table 12.3. Using K-map the logical expressions are defined as

$$q_2 = a_0.(\overline{a_1} + a_2.\overline{a_3}) \tag{12.4}$$

$$q_3 = a_0.a_1.(\overline{a_2} + \overline{a_3}) \tag{12.5}$$

Finally the overall architecture of LZC-16 is shown in Fig. 12.7. There are four LZC-4 blocks used here and one LZE block is used. The output of the LZE-4 block

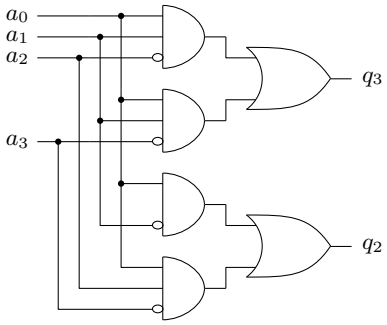


Fig. 12.6 A basic architecture for 4-bit leading zero encoder

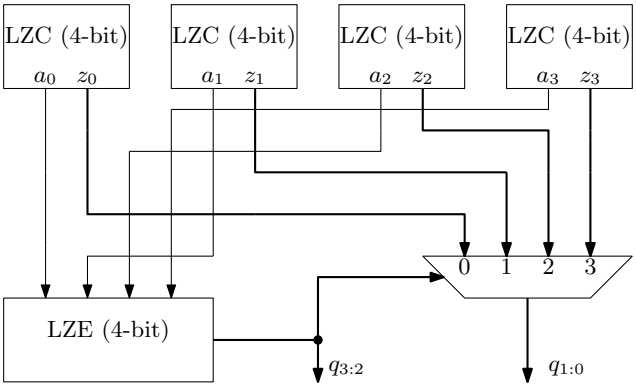


Fig. 12.7 A basic architecture for 16-bit leading zero counter

which are the upper bits of the counter selects the lower bits through a MUX. This leading zero counter block is an extra burden to the floating point architectures and it also increases the critical path. Some of the floating point architectures [69] adopted anticipation of leading zeros which anticipates leading zeros in parallel to the arithmetic operation on Mantissa.

12.5 Floating Point Addition

Compared to a fixed point adder, a floating point adder is more complex and hardware consuming. This is because exponent field is not present in case of fixed point arithmetic. A floating point addition of two numbers a and b can be expressed as

$$S_a.M_a.2^{E_a} + S_b.M_b.2^{E_b} = S.2^{E_b}(M_a + M_b^*) \tag{12.6}$$

Here, it is considered that $E_a > E_b$. In this case, M_b^* represents the right shifted version of M_b by $|E_a - E_b|$ bits. Similar operation can be carried out for $E_a < E_b$. Thus floating point addition is not as simple as fixed point addition. The major steps for a floating point addition are

1. Extract the sign of the result from the two sign bits.
2. Subtract the two exponents E_a and E_b. Find the absolute value of the exponent difference (E) and choose the exponent of the greater number.
3. Shift the mantissa of the lesser number by E bits considering the hidden bits.
4. Execute addition or subtraction operation between the shifted version of the mantissa and the mantissa of the other number. Consider the hidden bits also.
5. Normalization for addition: In case of addition, if carry is generated then the result is right shifted by 1-bit. This shift operation is reflected on exponent computation by an increment operation.
6. Normalization for subtraction: A normalization step is performed if there are leading zeros in case of subtraction operation. Depending on the leading zero count the obtained result is left shifted. Accordingly the exponent value is also decremented by the number of bits equal to the number of leading zeros.

Example: Floating Point Addition

- Representation: The input operands are represented as $a =$ 0_1001_00100000000 (4.5) and $b =$ 0_1000_11100000000 (3.75).
- Sign extraction: As both the numbers are positive then the sign of the output will be positive. Thus $S = 0$.
- Exponent subtraction: $E_a = 1001$ and $E_b = 1000$. Thus the result of the subtraction is $E = 0001$.
- Shifting of mantissa of lesser number: The mantissa $M_b =$ 1_11100000000 is shifted by 1 bit right and the result is $M_b =$ 0_11110000000.
- Result of the mantissa addition is 000010000000 and generates a carry. This means the result is greater than 1.
- The output of the adder is right shifted and the exponent value is incremented to get the correct results. The new mantissa value is now 00001000000 choosing the last 11 bits from the LSB and exponent is 1010.
- The final result is 0_1010_00001000000 which is equivalent to 8.25 in decimal.

Example: Floating Point Subtraction

- Representation: The input operands are represented as $a =$ 1_1010_00100000000 (-9) and $b =$ 0_1000_11111000000 (3.9375).
- Sign extraction: As sign of a is negative and a is greater than b thus $S = 1$.
- Exponent subtraction: $E_a = 1010$ and $E_b = 1000$. Thus result of the subtraction is $E = 0010$.

- Shifting of mantissa of lesser number: The mantissa $M_b = 1\_11111000000$ is shifted by 2-bit right and the result is $M_b = 0\_01111110000$.
- Result of the mantissa subtraction is 010100010000. This leading zero indicates that the result is lesser than 1.
- The output of the adder is left shifted by 1 bit as there is one leading zero and the exponent value is decremented by 1-bit to get the correct results. The new mantissa value is now 01000100000 choosing the last 11 bits from the LSB and the exponent is 1001.
- The final result is 1_1001_01000100000 which is equivalent to -5.0625 in decimal.

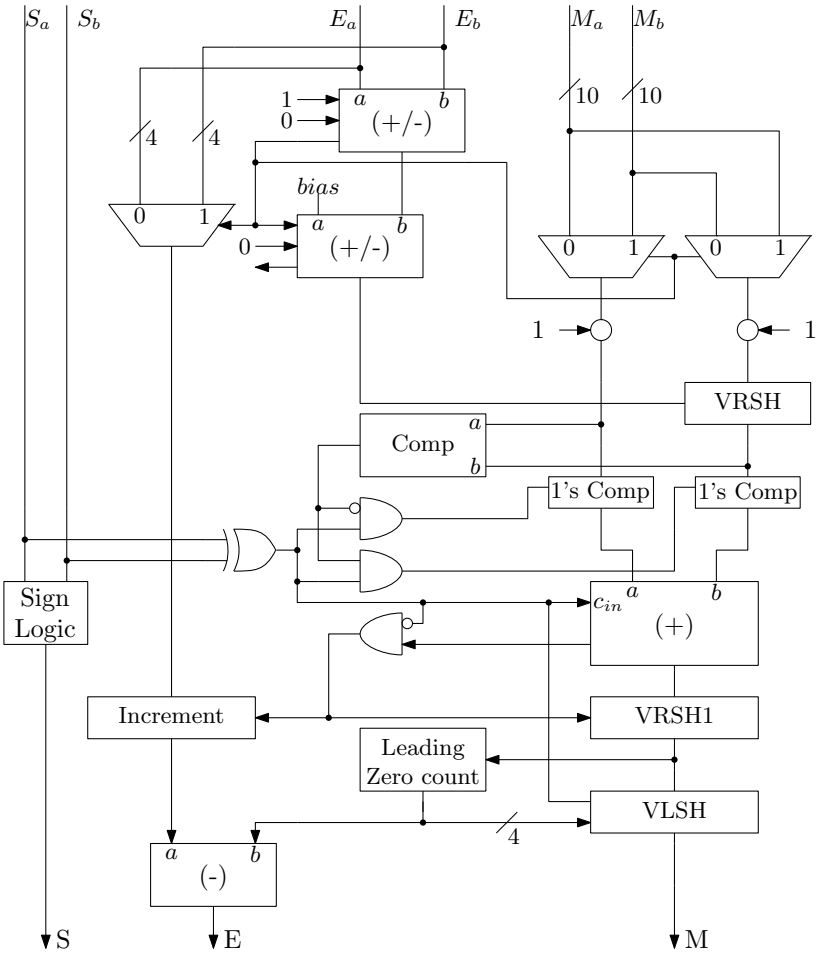


Fig. 12.8 A basic architecture for floating point addition

A simple architecture of a floating point adder is shown in Fig. 12.8. In this architecture, three 4-bit adders are used for computing the exponent, and a 12-bit adder is used for adding or subtracting the mantissa part. Two MUXes before the mantissa computation path selects the mantissa of the lower number for shifting. The shift operation is carried out by a VRSH block. This block shifts the mantissa according to the exponent difference. The addition or subtraction is done by Two's complement method. Thus a comparator is used to detect the smaller mantissa for inversion. The leading zero counter is for normalizing the result in case of subtraction operation when the mantissa part contains the leading zeros. This block has no meaning in case of addition operation. The VLSH block is a variable left shifter like VRSH block.

12.6 Floating Point Multiplication

Floating point multiplication is comparatively easy than the floating point addition algorithm but hardware consuming. Major hardware block is the multiplier which is same as fixed point multiplier. This multiplier is used to multiply the mantissas of the two numbers. A floating point multiplication between two numbers a and b can be expressed as

$$S_b.M_b.2^{E_b} \times S_a.M_a.2^{E_a} = (S_a \oplus S_b).M_b \times M_a.2^{(E_b+E_a)-bias} \tag{12.7}$$

Thus it can be said that in a floating point multiplication, mantissas are multiplied and exponents are added. The major steps for a floating point multiplication are

1. Extract the sign of the result from the two sign bits.
2. Add the two exponents (E). Subtract the bias component from the summation.
3. Multiply mantissa of b (M_b) by mantissa of a (M_a) considering the hidden bits.
4. If the MSB of the product is "1" then shift the result to the right by 1-bit.
5. Due to this, the exponent is to be incremented according to the one bit right shift.

Floating point multiplication can be more clearer with an example. Lets discuss a multiplication operation between two numbers $a = 6.5$ and $b = 3$. The result of the multiplication operation is 19.5.

Example: Floating Point Multiplication

- Representation: The input operands are represented as $a =$ 0_1001_10100000000 and $b =$ 0_1000_10000000000.
- Sign extraction: As both the numbers are positive then sign of the output will be positive. Thus $S = 0$.
- Exponent addition: $E_a = 1001$ and $E_b = 1000$. Thus result of the addition is $E = 10001$. The bias is subtracted from this value. Thus new value of E is 1010.

- Mantissa multiplication: Multiply the mantissas by any multiplicative algorithms used in the fixed point arithmetic. The width of the product is 24 bits but the final output is truncated to 11 bits. The 13 bits of the product starting from the MSB is 1001110000000.
- Generally the 11 bits from the LSB are the required result but here the MSB is "1" this indicates that the result is greater than "1". Thus this value is shifted right by 1-bit and the new result is 0100111000000. The final value of the mantissa (M) is 00111000000 excluding the hidden bit.
- This normalization step must reflect on exponent correction. The value of the exponent is corrected by an increment corresponding to a right shift. The new value of the exponent (E) is 1011.
- The final result is 0_1011_00111000000 which equivalent to 19.5 in decimal.

A simple architecture of a floating point multiplier is shown in Fig. 12.9. The addition of the exponents is done by a 5-bit adder as addition result can be greater than 15. The subtraction of the bias element can be done by another 5-bit adder. There is another 4-bit adder used in the design which is actually an incrementer. The major hardware block is the multiplier block. The multiplier used here is a 12-bit unsigned multiplier and that can be any multiplier circuit as discussed in Chap. 8. If MSB of the product is "1" then the output is normalized by right shifting. Here this right shift is simply achieved by using MUXes. In this case, as the hidden bit is also considered, the result will be always less than 4. Thus only the MSB is checked. Pipeline registers also must be inserted according to the pipelining stages of the multiplier.

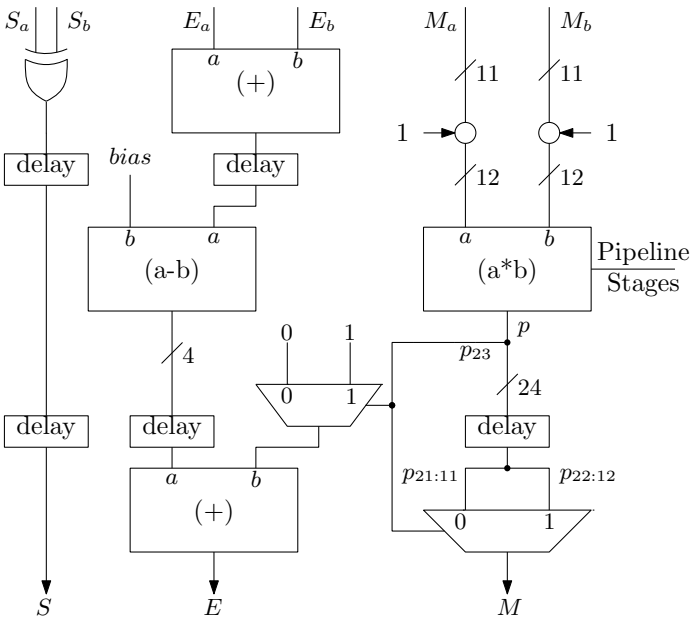


Fig. 12.9 A basic architecture for floating point multiplication

12.7 Floating Point Division

Performing division is a difficult task as we have seen in case of fixed point arithmetics also. Divider architectures are complex to implement. Floating point division is nothing but a fixed point division with some extra hardwares to take care of the exponents. This extra hardware makes the divider circuit more complex. A floating point division where a number a divides another number b can be expressed as

$$\frac{S_b.M_b.2^{E_b}}{S_a.M_a.2^{E_a}} = (S_a \oplus S_b).\frac{M_b}{M_a}.2^{bias+(E_b-E_a)} \tag{12.8}$$

Thus it can be said that in a floating point division, mantissas are divided and exponents are subtracted. The major steps for a floating point division are

1. Extract the sign of the result from the two sign bits.
2. Find the magnitude of the difference between two exponents (E). Add E to the bias if $E_b > E_a$ or subtract E from the bias if $E_b < E_a$.
3. Divide mantissa of b (M_b) by mantissa of a (M_a) considering the hidden bits.
4. If there is a leading zero then normalize the result by shifting it left.
5. Due to the normalization, the exponent is to be decremented according to the number of left shifts.

Floating point division can be more clearer with an example. Lets discuss a division operation between two numbers $b = 3$ and $a = -15.5$. The result of the division operation is -0.1935.

Example: Floating Point Division

- Representation: The input operands are represented as $a =$ 1_1010_11110000000 and $b =$ 0_1000_10000000000.
- Sign extraction: As one of the number is negative then sign of the output will be negative. Thus $S = 1$.
- Exponent subtraction: $E_a = 1010$ and $E_b = 1000$. Thus magnitude of their difference is $E = 0010$. As $E_b < E_a$ thus the resulted exponent is $0111 - 0010 = 0101$.
- Mantissa division: Divide the mantissas by any division algorithm used in the fixed point arithmetic. Considering the hidden bits, the division operation is restricted to 12 bits. The result of the division is 01100011000.
- There is a leading zero in the result thus a left shift can be applied to normalize the result. Thus the new result is 11000110000. The final value of the mantissa (M) is 1000110000 excluding the hidden bit.
- The action of normalization step must reflect on exponent correction. The value of the exponent is corrected by a decrement corresponding to a left shift. The new value of the exponent (E) is 0100.
- The final result is 1_0100_1000110000. The decimal value of this is -0.1933.

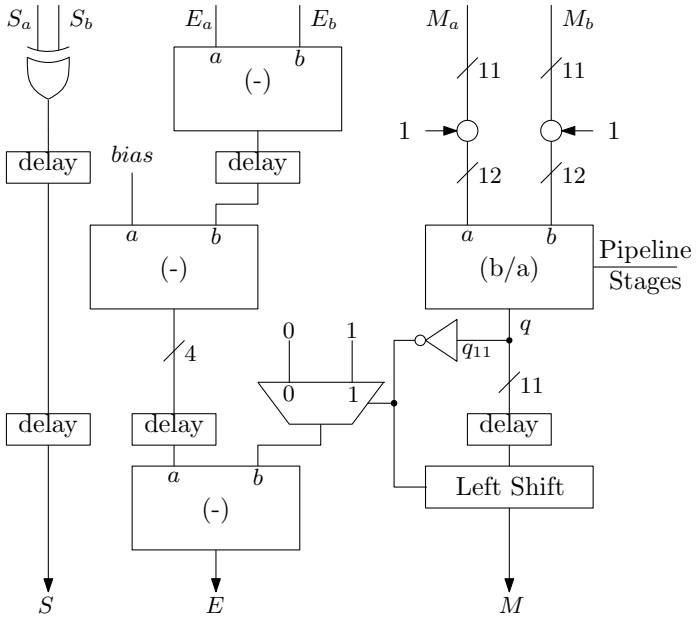


Fig. 12.10 A basic architecture for floating point division

A simple architecture of a floating point divider is shown in Fig. 12.10. There are three 4-bit subtractors used in the divider architecture, two for exponent subtraction and one for correction of exponents. The major hardware block is the divider block. The divider used here is a 12-bit unsigned divider and that can be any divider circuit as discussed in Chap. 9. If the result of the divider contains any leading zero then normalizing step is executed. But here in this case, as the hidden bit is also considered thus the result cannot go below 0.5. Thus there will be maximum of one leading zero present in the result. This is why only the MSB of the result (q) is considered and left shift block shifts only by one bit. Pipeline registers also must be inserted according to the pipelining stages of the divider.

12.8 Floating Point Comparison

Floating point comparison is similar to the comparison of two numbers in fixed point arithmetic. The same architecture can be used here also. Yet this architecture is discussed here to have a clear idea about floating point comparison. The steps involving the comparison of two floating point numbers a and b are

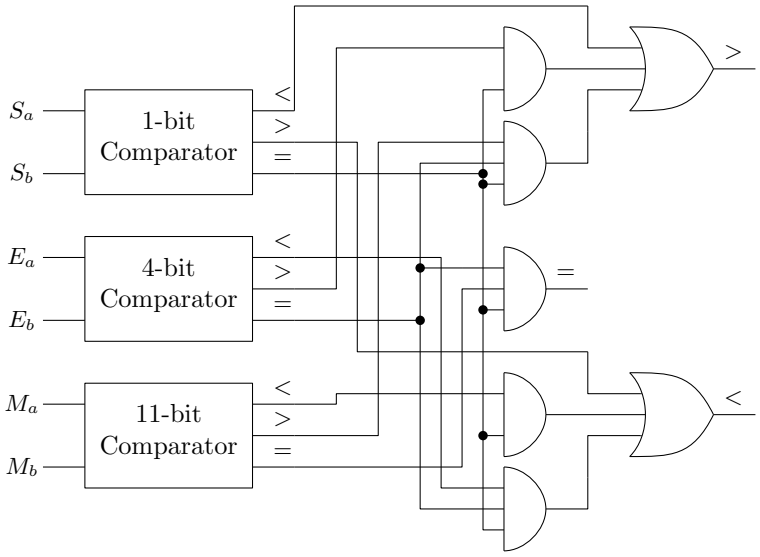


Fig. 12.11 A basic architecture for floating point comparison

1. Compare the sign bits. If both are of same sign then proceed to the next step.
2. Compare the exponents. If both exponents are also equal then go to the next step.
3. Compare the mantissas. Generate three outputs ($<$, $=$ and $>$).

Here we have shown a comparator for two 16-bit floating point numbers a and b. Two sign bits are compared using a simple 1-bit comparator. Here we will use the opposite of the convention used in case of 1 bit comparator. This is because, here if the sign bit is "1" then the number is negative and thus lesser. If sign of both the numbers are same then the exponents are compared. Here a 4-bit comparator is used to compare the exponents. If both the numbers have same sign and their exponents are also equal then at the last step mantissas are compared. The two mantissas are compared using an 11-bit comparator. The architecture of the comparator is shown in Fig. 12.11.

12.9 Floating Point Square Root

The architecture for square root computation can also be designed in the same way the other architectures are designed. Various fixed point square root architectures are discussed in Chap. 10. There are some modifications needed to convert a fixed point square root architecture to a floating point architecture. This modification is

mainly related to handling of the exponent field. Square root operation in floating point arithmetic can be expressed as

$$\sqrt{0.M_a.2^{E_a}} = 0.\sqrt{M_a}.2^{E_a/2} \tag{12.9}$$

Here the sign field is logic zero which means the square root block always expects positive numbers. The square root operation is carried out only on the mantissa part and this can be achieved by any square root algorithm mentioned in Chap. 10. The exponent part is divided by 2 which means right shifting by one bit. The steps for square root operation are

1. Subtract the bias component from the exponent and find the absolute difference.
2. Right shift the result by one bit then compute the final exponent.
3. Find the square root of the mantissa with considering the hidden bit.

Example: Floating Point Square Root

Floating point square root operation can be more clearer with an example. Lets discuss a square root operation for $a = 8$.

- The input data a is represented as 0_1010_00000000000 in 16-bit format.
- The absolute difference between the exponent and bias is 3 ($1010 - 0111 = 0011$). As $E > bias$, the number is greater than 1.
- This result is right shifted by 1-bit and this resulted an extra bit (rs) which is 1.
- Concatenate the hidden bit with mantissa. Thus the square root block gets 100000000000 as input.
- As the input data is greater than or equal to 1, the input is right shifted by 1 bit so that the fractional square root block can give accurate result.
- The output of the square root block is 101101010000. Thus the value of the mantissa part is $M = 01101010000$.
- The new exponent is $0111 + 0001 = 1000$.
- Thus the final result is 0_1000_01101010000.

A basic scheme of floating point square root computation is shown in Fig. 12.12. Here the architecture is self-explanatory. The VRSH1 block is control right shift block. The square root block is capable of computing square root of fractional numbers. In the exponent computation path there is a MUX placed which selects between bias and the exponent of the input number. This is due to the fact that in the square root process exponent of the fractional number should increase with respect to the original exponent.

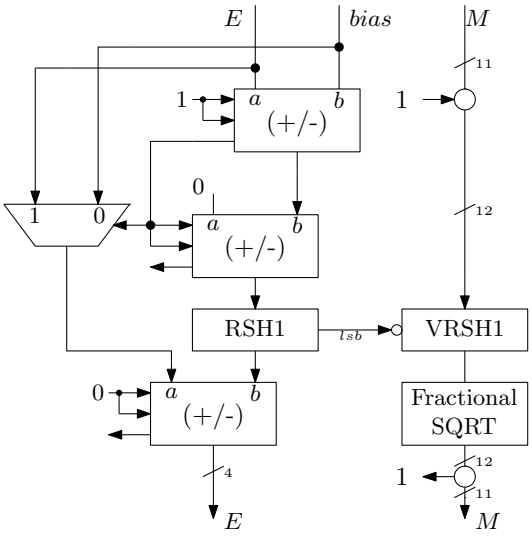


Fig. 12.12 A basic architecture for floating point square root computation

12.10 Floating Point to Fixed Point Conversion

The floating point to fixed point conversion is necessary to interface a floating point processor to a fixed point implementation. So that there maintains a smooth transition from one type of architecture to another. The steps involved in this conversion are

1. Concatenate the hidden bit and add leading zeros according to the fixed point length.
2. Find the absolute difference between the exponent of the floating point number and bias.
3. If $E > bias$ left shift the input number by their absolute difference. Otherwise if $E < bias$ right shift is executed.
4. Finally, invert the number if sign bit is 1.

Example: Floating Point Square Root

- Input data is represented in floating point as $a = 0\_1011\_01000000000$.
- Prepare the data as 0000_1_01000000000 for 16-bit fixed point representation with 6 integer bits.
- The difference between exponent and bias is $1011 - 0111 = 0100$ and exponent is greater than the bias.
- Left shift the number by 4-bit. Result is 1010000000000000.
- As the sign bit 0, no need of inversion. Discard the LSB and concatenate the sign bit at the MSB side. The final output is 0101000000000000.

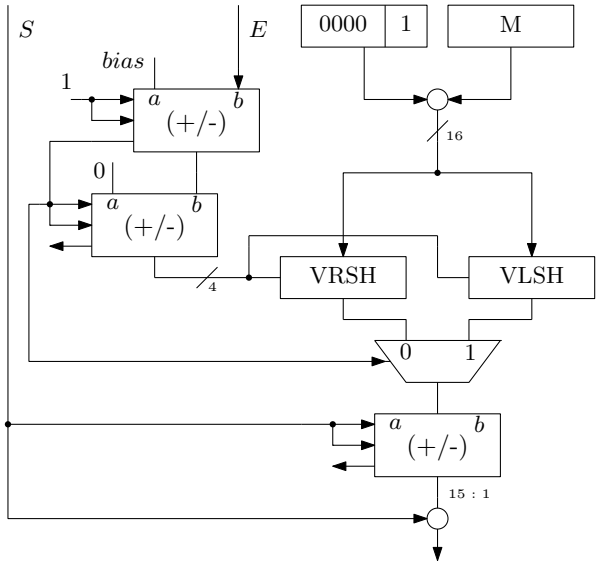


Fig. 12.13 Scheme for floating point to fixed point conversion

In this above example, the LSB is discarded to fit the result in 16-bit format. Thus there exists error in the conversion process. Only a certain range of floating point representation can be represented in fixed point for same word length. An architecture for floating point to fixed point conversion is shown in Fig. 12.13. Here two variable shifters are used, viz., VLSH and VRSH. VLSH does the right shifting like the VRSH block. Two 4-bit adder/subtractors and one 16-bit adder/subtractor are used. This architecture can be adopted for any word length.

12.11 Conclusion

In this chapter, various floating point architectures are discussed. Though mainly fixed point representation is used in this book, designers may choose floating point data format to achieve high accuracy. This chapter focuses on how floating point architectures can be designed for custom word length. This eliminates the use of single or double precision word lengths as per IEEE standard. Basic floating point architectures for addition, multiplication, division and square root are covered in this chapter. Readers are suggested to read more articles on efficient architectures.

Chapter 13
Timing Analysis

13.1 Introduction

The digital systems are designed using some software tools and then simulated by giving input test vectors. The designs may work at logic simulation level but may not work when these designs are implemented on hardware. This is because, practical aspects like routing delay, process variations are ignored at logic simulation level. Timing verification of digital systems is very important as without timing verification one cannot proceed towards fabrication. Timing verification increases the yield of good ICs.

There are two types of timing verification which can be carried out to verify a design. First one is Dynamic Timing Simulation (DTS) and the second one is Static Timing Analysis (STA). In the DTS, set of input test vectors which covers all the possible input combinations are given as input and output is verified. This process is slow as huge number of input vectors are to be checked. But this timing verification method can be very accurate. Some of the platforms of performing DTS are VCS, SPICE, ACE, etc.

STA is a process which performs the timing verification statically and independent of input test vectors applied at input pins. STA is a fast process as huge set of test vectors is not required for verification. STA gives better analysis checks when timing requirement is given. But STA can be less accurate when there are false paths or asynchronous designs. Thus timing exceptions are also needed to specify.

STA is the most accepted technique in the industry for timing verification. Tools to perform STA are Synopsys Primetime, Cadence Tempus, etc. STA is performed in three steps which are

1. Overall design is divided into set of timing paths
2. Delay of each path in each set is calculated
3. Path delays are checked to see if timing constraints have been met.

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_13

In this chapter, a basic theoretical background on how STA is performed is discussed. First the definitions related to the STA are discussed and then different criteria for timing checks are discussed.

13.2 Timing Definitions

A digital system can be combinational or sequential. Most of the complex designs are sequential and very few small designs are combinational. A sequential design can be synchronous or asynchronous. In a synchronous design, data propagates with respect to clock and in asynchronous designs signal transition can occur irrespective of clock. These definitions regarding the system design are important to understand the timing verification process. Different other timing related definitions are discussed below.

13.2.1 Slew of Waveform

The slew rate of a signal is defined as the rate of change of that signal. This is also can be defined as the time taken by the signal to switch from one specified level to another level. Larger transition time means slower slew rate. The slew can be of two types, viz., fall slew and rise slew. If a signal falls from 70 % to 30 % of its maximum value, then it is called fall slew. Similarly, the rise slew is defined. The slew of a waveform is explained in Fig. 13.1.

13.2.2 Clock Jitter

The clock signal is generated by real physical devices and thus there is no ideal clock signal. Every real clock has some finite amount of jitter. The jitter is defined as a window within which the clock edge can occur. This is responsible for amount of cycle-to-cycle variation in a clock period. This uncertainty in clock signal is shown

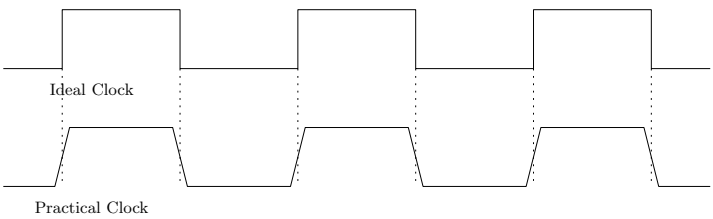


Fig. 13.1 Slew of the clock signal

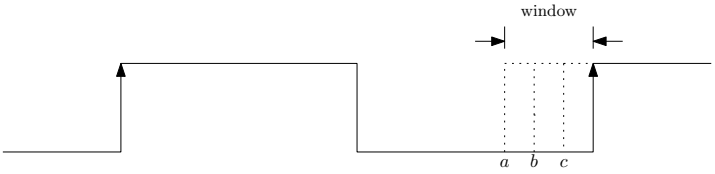


Fig. 13.2 Clock jitter window

in Fig. 13.2. Here, the second rising edge can be started from point a, b or c due to any disturbances in the clock source. The sink flip-flops receive a faster clock due to this jitter. The window during which the clock edge can occur is called jitter window.

13.2.3 Clock Latency

Clock latency is defined as the time taken by the clock signal to reach the sink flip-flops from the clock source like Phase Locked Loops (PLLs). The clock signal at the source is defined as Master Clock (clk_M) and the clock signal at the sink flip-flops is defined as Sink Clock (clk_S). Sometimes the Master clock signal is passed through a clock divider circuit to produce generated clock (clk_G). The concept of clock latency is explained in Fig. 13.3. Clock latency can be divided into two parts, viz., Clock Source Latency and Clock Network Latency.

Clock Source Latency

Source latency is defined by the delay between the clk_M and the clk_G. In situations where the requirement of generated clock is not required, clock latency is defined as the delay between the clk_M and the clock signal at the definition point.

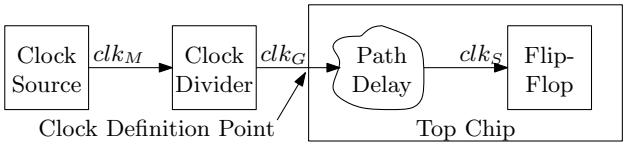


Fig. 13.3 Definition of clock latency

Clock Network Latency

The Network latency is defined as the delay between the clk_G and the clk_S. In situations where clk_G is absent, the Network delay is defined as the delay between the clock signal defined at clock definition point and the clk_S. This is an internal delay due to the clock tree from the clock definition point to all the clock sinks. Sometimes another term insertion delay is defined to indicate the delay of clock signal from external source to the sink flip-flop.

13.2.4 Launching and Capturing Flip-Flop

In a digital system, data propagates from one flip-flop to another. In a synchronous system, this propagation is achieved through edge triggering. One flip-flop launches a data and another flip-flop captures the data. This concept is shown in Fig. 13.4. Here same clock is connected to both FF1 and FF2. At the first rising edge, FF1 is launching the data. That is, the D value causes a change in Q of FF1. So, FF1 is called as launching flip-flop for this timing path and the first rising edge is launching edge. This signal will be captured at FF2 after one cycle. So FF2 becomes the capturing flip-flop and 2nd rising edge becomes capturing edge.

13.2.5 Clock Skew

The latency of the sink clock (clk_S) at the two different flip-flips can be different. This latency difference is called as the clock skew which is due to the delay involved in the respective clock tree path. The clock skew is explained in Fig. 13.5. Here, the clock signal from the clock definition point goes through the three sink flip-flops. The difference between the latency of any two clocks among clk_a, clk_b or clk_c is clock skew. In a circuit, the maximum possible clock skew is considered to meet the timing.

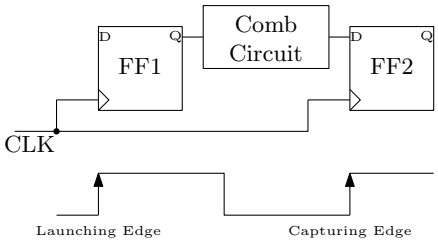


Fig. 13.4 The concept of launching edge and capturing edge for a flip-flop

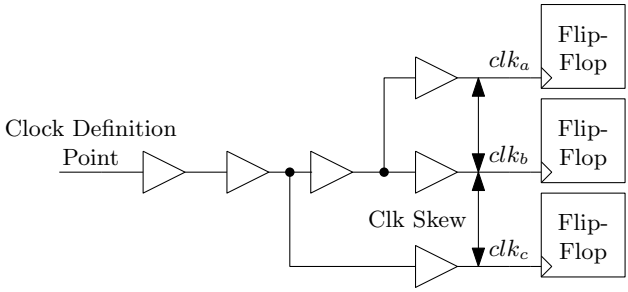


Fig. 13.5 Definition of clock skew

Here, the definition of ideal clock tree is important. A clock tree is ideal when it can drive infinite drives with zero delay. This means that the clock skew is zero for ideal clock tree. If capture clock comes late than launch clock then it is called positive skew. If capture clock comes early than launch clock it is called negative skew. Clock skew can be defined in other words as the difference in arrival of clock at two consecutive pins of a sequential element. Global skew is defined as the difference between max insertion delay and the min insertion delay of any flops

13.2.6 Clock Uncertainty

The uncertainty in the clock signal can be account from several factors. But mainly in timing verification clock jitter and clock skew are considered as clock uncertainty. Additional margins are also considered to meet timing. The clock uncertainty can be specified separately for setup and hold check ups.

13.2.7 Clock-to-Q Delay

The clock-to-Q delay time (t_{c2q}) is the time measured between the clock edge and flip-flop output (Q) for an edge triggering flip-flop. This timing parameter of a flip-flop can be minimum and maximum. The minimum amount of time after which the Q signal might be unstable or starts changing is called minimum clock-to-Q delay time (t_{cc2q}). This minimum clock-to-Q delay time is also called as contamination delay for clock-to-Q delay. On the other hand, the maximum amount of time after which the Q signal is guaranteed to stable or stops changing is called maximum clock-to-Q delay time (t_{pc2q}). This minimum clock-to-Q delay time is also called as propagation delay for clock-to-Q delay. The concept of clock-to-Q delay is shown in Fig. 13.6.

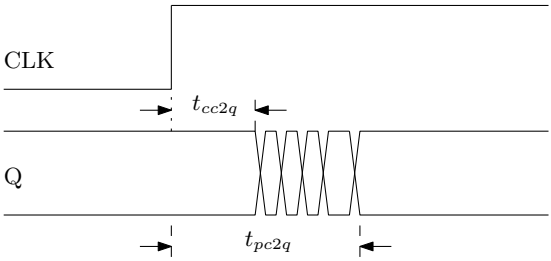


Fig. 13.6 Clock-to-Q delay time concept

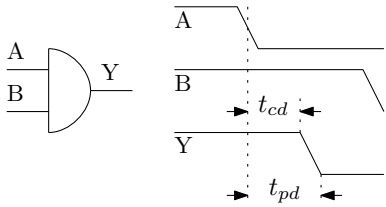


Fig. 13.7 Logic gate output delay time concept

13.2.8 Combinational Logic Timing

Similar to the output timing characteristic of the flip-flops, timing of the combinational logic gates also can be defined. Minimum mount of time from when an input changes until the output starts to change is called contamination delay (t_{cd}) of a logic gate. It is also called as minimum delay of a logic gate. Similarly, maximum amount of time from when an input changes until the output is guaranteed to reach its final value (i.e. stop changing) is called propagation delay of a logic gate. It is also simply called as maximum delay of a logic gate. The concept of these two delays is shown in Fig. 13.7.

13.2.9 Min and Max Timing Paths

The path delay is defined as the delay that is required for a logic to propagate through a logic path. This path delay accounts for both delay through logic cells and delay due to nets. There may be multiple paths from the source to the destination. The delay associated with each path is different. The max path has the largest delay and the min path has smallest delay. These two paths are very important in timing analysis. A path is called as critical if the path violates any timing constraints. A critical path can be a max path or a min path.

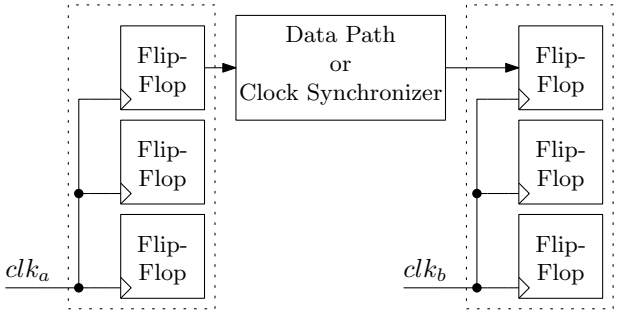


Fig. 13.8 Two clock domains and their connectivity

13.2.10 Clock Domains

A clock signal can be connected to multiple sink flip-flops. The set of flip-flops which is connected to a clock is called domain of that clock. In a complex design there may be many clock signals used. For example, clk_a can be connected to 100 flip-flops and clk_b can be connected to 50 flip-flops. The concept of clock domain is shown in Fig. 13.8. Here, in this design there are two clock domains and each clock domain there are three flip-flops. Two clock domains can be connected in two ways. Firstly, a data path can start in domain of clk_a and end up in domain of clk_b. Secondly, there may be a clock synchronizer placed between the two clock domains.

13.2.11 Setup Time

Setup time (T_{su}) is the minimum amount of time during which the data signal should be held steady before the clock event so that the data can be reliably sampled by the clock. This applies to synchronous circuits such as the flip-flop. This implies the amount of time during which the synchronous input (D) must be stable before the active edge of the Clock. The time within which the input data is available and stable before the clock pulse is applied is called Setup time. The concept of setup time is shown in Fig. 13.9.

13.2.12 Hold Time

Hold time (T_h) is the minimum amount of time during which the data signal should be held steady after the clock event occurs so that the data can be reliably sampled. This applies to synchronous circuits such as the flip-flop. This implies the time during which input (D) of the synchronous flip-flop must be stable after the active edge of

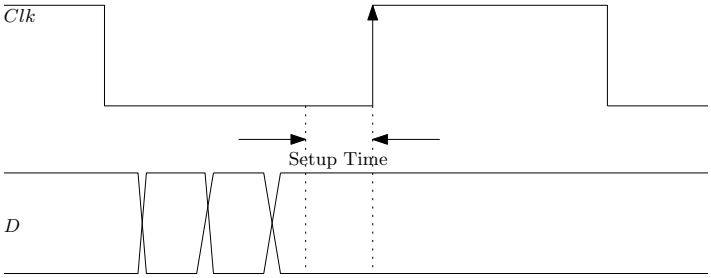


Fig. 13.9 The concept of setup time for a flip-flop

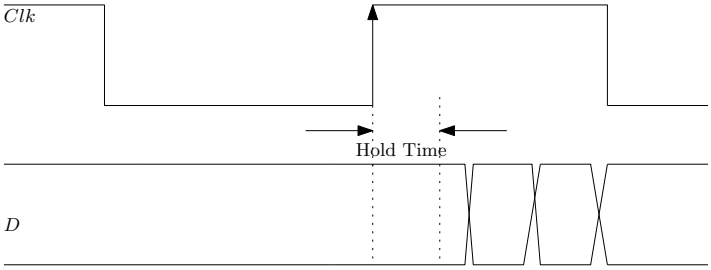


Fig. 13.10 The concept of Hold time for a flip-flop

the clock. The time after clock pulse within which the data input is held stable is called hold time. The concept of hold time is shown in Fig. 13.10.

13.2.13 Slack

Slack is the difference between the desired arrival times and the actual arrival time for a signal. Slack determines if a design is working at a desired frequency or not. Positive Slack indicates that the design is meeting the timing and still it can be improved. Zero slack means that the design is critically working at the desired frequency. Negative slack means, design has not achieved the specified timings at the specified frequency. Slack has to be positive always and negative slack indicates a violation in timing. Setup and Hold slack are calculated as

$$\text{SetupSlack} = \text{Requiredtime} - \text{Arrivaltime} \tag{13.1}$$

$$\text{Holdslack} = \text{Arrivaltime} - \text{Requiredtime} \tag{13.2}$$

13.2.14 Required Time and Arrival Time

Required time is the time within which data is required to arrive at internal node of flip-flop. Required time is constrained by the designers. The time in which data arrives at the internal node is the arrival time. It incorporates all the net and logic delays in between the reference input point and the destination node.

13.2.15 Timing Paths

The different kinds of paths when checking the timing of a design are as follows.

1. Flip-flop to flip-flop timing path.
2. External input device to on-chip flip-flop timing path.
3. On chip flip-flop to external output devices.
4. External input device to external output device through on-chip combinational block.

13.3 Timing Checks

There may be two types of paths which can cause timing violation, one is max path and the second one is min path. Two types of timing checks are needed to be performed one is hold check for min path and setup check for max path. These two checks are explained below.

13.3.1 Setup Timing Check

In a setup timing check, the timing relationship between the clock and the data pin of a flip-flop is checked to observe that the setup timing requirement is met or not. The setup check ensures that the data is stable for a certain amount of time, which is the setup time of the flip-flop, before the active edge. This is done to ensure that the capture flip-flop correctly captures the data. Figure 13.11 illustrates the setup condition. The condition for setup check can be written as

$$T_{launch} + T_{pc2q} + T_{pd} < T_{capture} + T_{cycle} - T_{su} \tag{13.3}$$

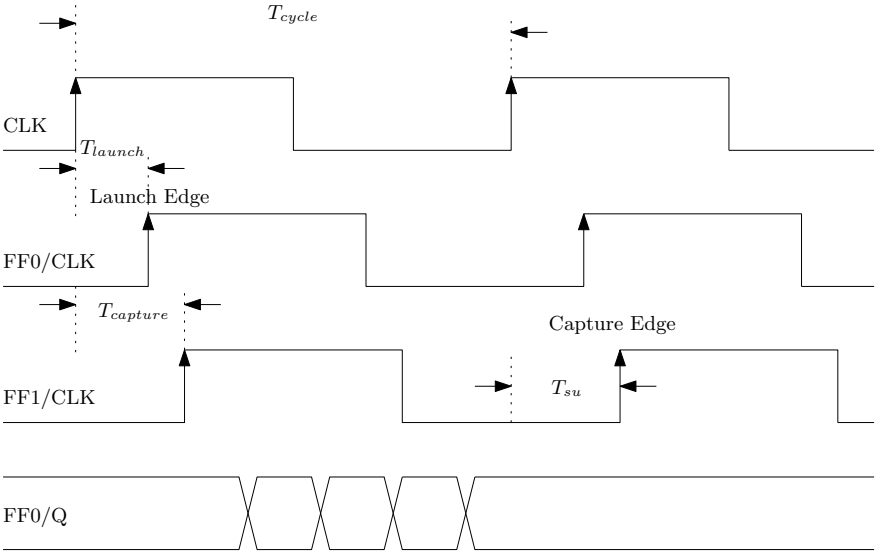


Fig. 13.11 The concept of setup timing check for a sequential circuit

13.3.2 Hold Timing Check

A hold timing check is performed to ensure that the hold specification of flip-flop is met. According to the hold specification of a flip-flop, the data should be stable for a specified amount of time after the active edge of the clock. The Fig. 13.12 shows the concept of hold timing check.

$$T_{launch} + T_{cc2q} + T_{cd} > T_{capture} + T_h \tag{13.4}$$

13.4 Timing Checks for Different Timing Paths

The different types of paths are already discussed above. Now, setup and hold check-ups are to be performed for each type of path. In this section, the timing checks are performed for different types of paths.

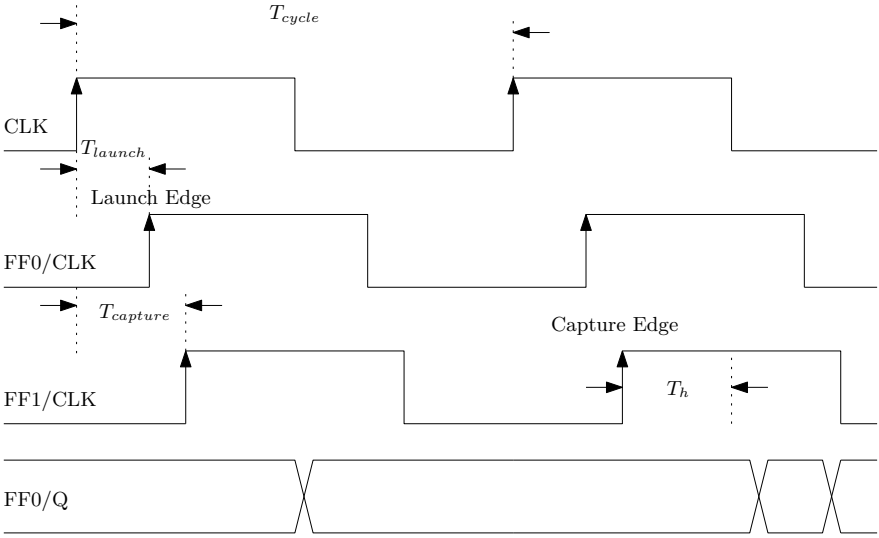


Fig. 13.12 The concept of Hold timing check for a sequential circuit

13.4.1 Setup Check for Flip-Flop to Flip-Flop Timing Path

In a flip-flop to flip-flop timing path, both the flip-flops can be connected to same or different clock signal. Figure 13.6 demonstrates this timing path. Here, the data is launched by the launch flip-flop and reaches another flip-flop through a combinational circuit.

13.4.1.1 Computation of Setup Slack

In this case the required time and the arrival time are defined as

$$\text{Required Time} = T_{clock} + T_{capture} - T_{su} \tag{13.5}$$

$$\text{Arrival Time} = T_{launch} + T_{pc2q} + T_{logic} + T_{net} \tag{13.6}$$

Lets consider an example to understand the setup slack calculation for flip-flop timing path.

Example 13.1 The launching active clock edge occurred at 4.30 ns with $T_{launch} = 0.20$ ns. The clock signal has the period of 10 ns and $T_{capture} = 0.50$ ns. The total data path delay is 6.50 ns and the setup constant can be considered as $T_{su} = 0.46$ ns.

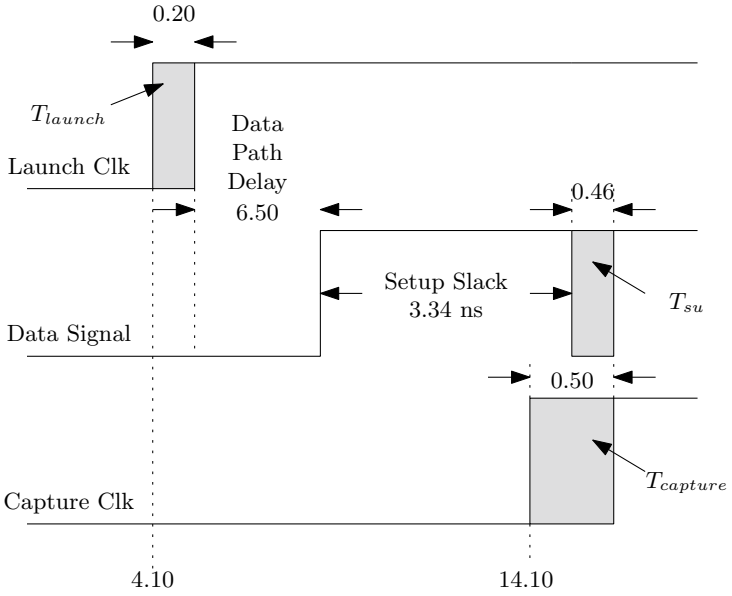


Fig. 13.13 The setup slack computation for flip-flop to flip-flop timing path

Solution: If the clock was ideal, the launching edge would have started at 4.10 ns. The setup slack is computed as

$$\text{Setup Slack} = (10 + 0.50 - 0.46) - (0.20 + 6.50) = 10.04 - 6.70 = 3.34 \tag{13.7}$$

The setup slack is positive, it means the design met the timing requirement and the design can tolerate combinational delay up to 3.34 ns. This example is explained graphically in Fig. 13.13.

13.4.1.2 Computation of Hold Slack

Similarly the hold slack can be computed for this timing path. The required time for this is

$$\text{Required Time} = T_{capture} + T_h \tag{13.8}$$

The arrival time is calculated as

$$\text{Arrival Time} = T_{launch} + T_{cc2q} + T_{logic} + T_{net} \tag{13.9}$$

For the above example, the hold slack can be computed as for hold timing constant of 0.4 ns as

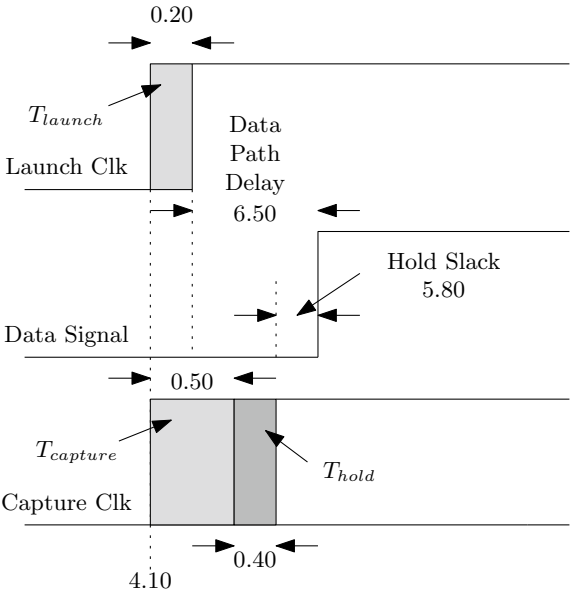


Fig. 13.14 The hold slack computation for flip-flop to flip-flop timing path

$$\begin{aligned}\text{Hold Slack} &= \text{Arrival time} - \text{Required Time} \\ &= (6.5 + 0.20) - (0.50 + 0.40) = 5.8 \end{aligned} \tag{13.10}$$

This situation is explained in Fig. 13.14 for computation of hold slack.

13.4.2 Setup and Hold Check for Input to Flip-Flop Timing Path

In this timing path, input signal from an off-chip external device is fed to an on-chip sequential block or flip-flop. This path is shown in Fig. 13.15. Here, a combinational path is considered before the sequential block which is connected to the master clock (clk_m). The external input device can also be sequential or combinational. The timing analysis is carried out by considering that an external sequential device is used to feed the input signal. In this case, the external sequential device is connected to a virtual clock (clk_v). Using this virtual clock the timing analysis is carried out. The total delay in this path is

$$\begin{aligned}\text{Total Path Delay} = {} & \text{off-chip delay or external I/P delay} + \\ & \text{the combinational path delay before the first on-chip flip-flop.}\end{aligned} \tag{13.11}$$

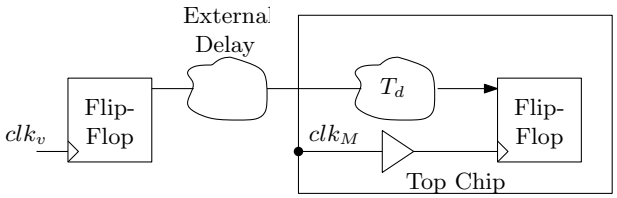


Fig. 13.15 The hold slack computation for input to flip-flop timing path

13.4.3 Setup Check for Flip-Flop to Output Timing Path

The timing analysis for this path is carried out in the same way as done in case of flip-flop timing path. Here also the off-chip output device can be connected to virtual clock or actual clock. The output data is taken from a flip-flop which is called the launching flip-flop. The external flip-flop can be considered as capturing flip-flop. In between these two on-chip and off-chip flip-flops the data path delay is due to the delay of combination block and the external o/p delay. This timing path is shown in Fig. 13.16. This path delay is

$$\text{Total Path Delay} = \text{On chip combinational path delay after the last on-chip flip-flop} + \text{External O/P delay} \quad (13.12)$$

13.4.4 Setup Check for Input to Output Timing Path

This data path can start from an external input device and can end up in external output device after propagating through some on-chip combinational path. This path is possible when designer wants to verify the complete combinational design. In such cases it is assumed that both sequential blocks for input and output are connected to

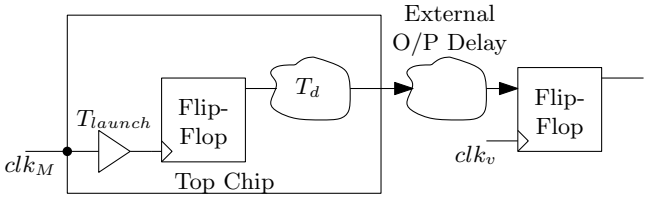


Fig. 13.16 The concept of flip-flop to output device timing path

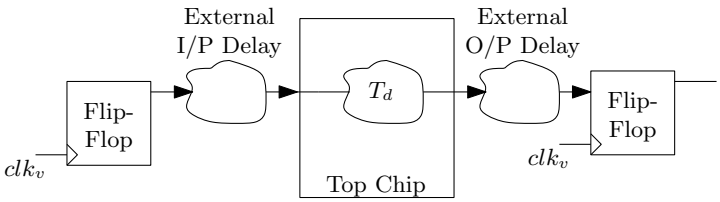


Fig. 13.17 The concept of timing path between input device to output device

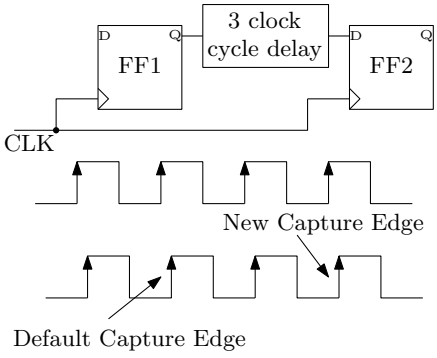
same virtual clock. This timing data path is shown in Fig. 13.17. The total data path delay in such cases is

$$\text{Total Path Delay} = \text{External I/P delay} + \text{On chip combinational path delay} + \text{Output external delay} \tag{13.13}$$

13.4.5 Multicycle Paths

In implementations of complex digital systems, some special cases may arise where the combinational path between two flip-flops has delay of the order of multiple clock cycles. In these special cases, these special paths are called as multicycle paths. In other cases, data is captured in every clock cycles but for multicycle paths data are captured after specified number of clock cycles. This situation is explained with the help Fig. 13.18. Here, the combinational path has delay of 3 clock cycles. The capture edge for timing analysis is now changed.

Fig. 13.18 The concept of multicycle path



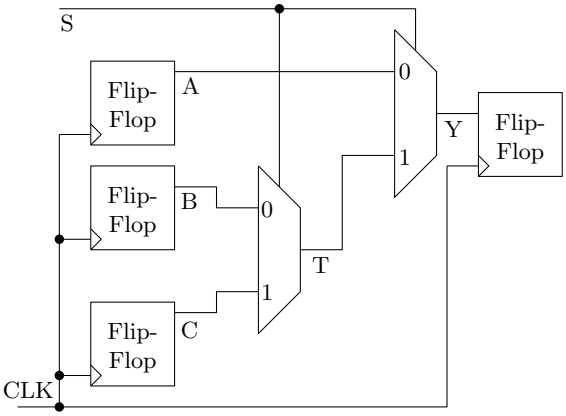


Fig. 13.19 The concept false path

13.4.6 False Paths

Some paths in a design are not real and termed as false paths. These paths should be excluded from the timing analysis otherwise the timing verification process will take unnecessarily extra time and storage. The false paths exist physically in a design but they are not logical or functional. Examples of such false paths are, path from one clock domain to another clock domain, from a clock pin to input of a flip-flop, paths which are never selected, paths which are not sensitized under any input conditions.

Figure 13.19 shows an example of a false timing path. Here, two MUXes are connected to same control line. Any of the three inputs A, B and C are connected to the output line Y based on the status of signal S. There is no chance that the path B-T-Y is selected by the control input S. This path is a false path and must be excluded from the timing verification process.

13.4.7 Half Cycle Paths

A design can use both negative edge triggered flip-flops and positive edge triggered flip-flops. In such designs, half cycle paths can exist. The half cycle path exists between the positive edge triggered flip-flop to negative edge triggered flip-flop or vice-versa. The concept of half cycle is shown in Fig. 13.20. The data has to propagate within the half cycle period in such cases.

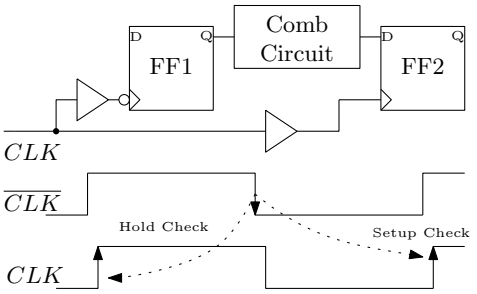


Fig. 13.20 The concept of half cycle path

13.5 Asynchronous Checks

Asynchronous circuits are generally not suitable for complex designs as performing STA for asynchronous circuits is difficult. Flip-flops can have asynchronous set or reset inputs. A path from any type of pin to these set or reset inputs is a asynchronous path. The functionality of set/reset pin is independent from the clock edge. These pins are level triggered and can function at any point of time. An example of this type of path is shown in Fig. 13.21. Though these paths are asynchronous, timing checks are needed to be performed so that the active edges of the flip-flops become unaffected. These checks are explained below.

13.5.1 Recovery Timing Check

The recovery time is a minimum amount of time which must be spent between the time instant when the asynchronous control signal becomes inactive and the next active clock edge. A recovery timing check ensures that recovery timing is maintained. This check ensures that after the asynchronous signal becomes inactive, there is adequate time to recover so that the next active clock edge can be effective. The concept of recovery timing is shown in Fig. 13.22. Here, the asynchronous control signal is rst which is become inactive before the recovery time. This timing check is similar to max path timing check in case of synchronous designs.

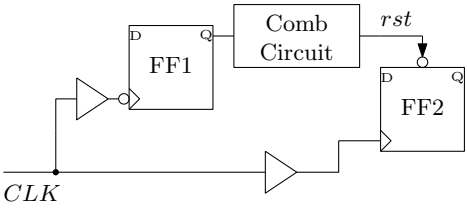


Fig. 13.21 The concept of asynchronous path

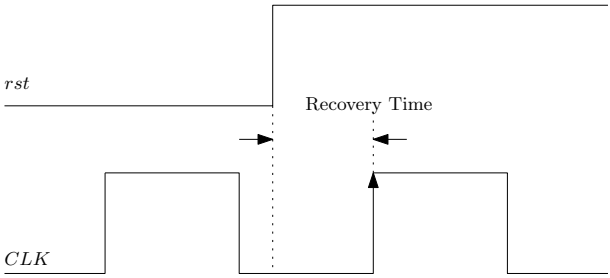


Fig. 13.22 The concept recovery timing check

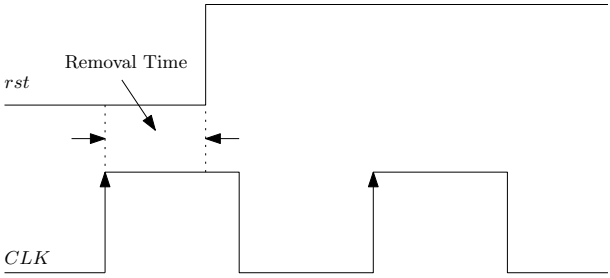


Fig. 13.23 The concept of removal timing check

13.5.2 Removal Timing Check

The removal timing check is performed to ensure that there is adequate time spent between the release of the asynchronous control signal and the active clock edge. This time gap is called as removal time. The removal time ensures that active edge has no effect as the asynchronous control signal is still active. In other words, the asynchronous control signal is released (becomes inactive) well after the active clock edge so that the clock edge can have no effect. This is illustrated in Fig. 13.23 where the asynchronous control signal is rst which is an active low signal. Like hold path check in case of synchronous circuits, removal check is min path check.

13.6 Maximum Frequency Computation

Data in synchronous designs are propagated with respect to clock edge. The sequential devices can be either positive edge triggered or negative edge triggered. Sometimes, both the edges are used for faster response. Maximum frequency of clock signal or minimum period of time is an important parameter to estimate a design's performance. Estimating this parameter one can say that the design is how much fast.

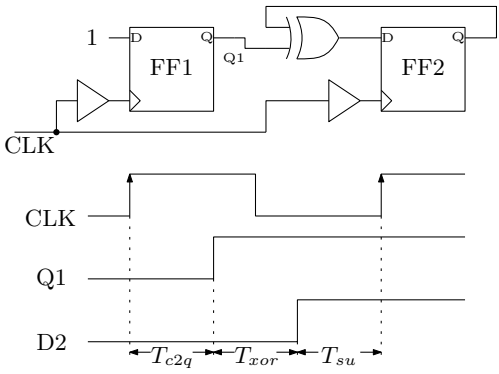


Fig. 13.24 Example for computation of maximum frequency

Thus this parameter must be estimated and estimation of minimum clock period can be done using the following relation.

$$T_{clock} = T_{pc2q} + T_{logic} + T_{path} + T_{su} + T_{skew} \tag{13.14}$$

Here, the skew can be positive or negative. Maximum frequency computation for an example circuit is shown in Fig. 13.24. Here, an XOR gate is used as a combinational circuit. The input $D0$ is connected to Vdd. Input $D0$ is reflected in $Q0$ after delay of T_{pc2q}. Then after the delay of XOR gate (T_{xor}) the $D1$ is evaluated. There are two paths, one from $Q1$ to $Q2$ and another path is from $Q2$ to $Q2$. The maximum frequency for the first path is

$$T_{clock} = T_{pc2q-FF1} + T_{xor} + T_{su} \tag{13.15}$$

The maximum frequency computation as per the second path will be

$$T_{clock} = T_{pc2q-FF2} + T_{xor} + T_{su} \tag{13.16}$$

13.7 Maximum Allowable Skew

The clock skew affects the maximum frequency computation. There should be a maximum allowable clock skew which can be tolerated in a design. Estimation of this parameter can tell the designers that how the clock trees can be designed or how the buffers can be placed. Two cases can arise in estimating maximum allowable skew. The first case is for positive skew where $T_{capture}$ is greater than the T_{launch}. This means that clock at the capture flip-flop has greater delay than the clock signal at launch flip-flop. Thus situation is depicted in Fig. 13.25. Another case may arise for negative skew where the clock at launch flip-flop has more delay than the clock

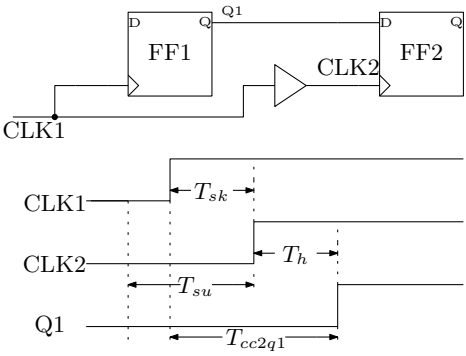


Fig. 13.25 Maximum allowable skew computation for positive skew

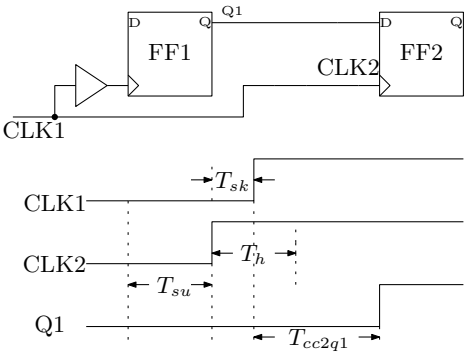


Fig. 13.26 Maximum allowable skew computation for negative skew

at capture flip-flop. This situation is shown in Fig. 13.26. The maximum allowable skew can be estimated using the following equation

$$T_{cc2q} > T_{sk} + T_h \tag{13.17}$$

From this the derivation for T_{sk} is

$$T_{sk} < T_{cc2q} - T_h \tag{13.18}$$

Example 13.2 Analyse the circuit shown in Fig. 13.27 for setup and hold check. Find for any setup and hold violation. If there is any violation then suggest a method to fix the timing violation. Consider the following parameters about the circuit.

1. $t_{cc2q} = 30ps$
2. $t_{pc2q} = 50ps$
3. $t_{su} = 60ps$
4. $t_h = 70ps$
5. $t_{cd-xor} = 35ps$
6. $t_{pd-xor} = 25ps$.

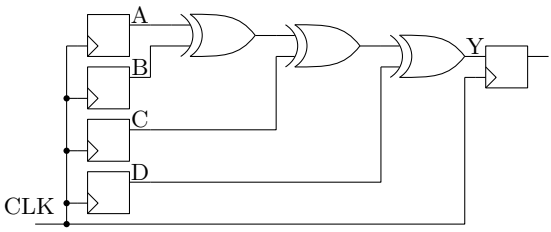


Fig. 13.27 A example circuit for timing analysis

Solution: In order to analyse the above circuit, the first job is to compute the minimum and maximum combinational delay of the circuit. The minimum delay of the circuit is $t_{cd} = t_{cd-xor} = 35ps$ corresponding to minimum path (D-Y) and the maximum combinational delay is $t_{pd} = 3 \times t_{pd-xor} = 75ps$ corresponding to maximum path (A-Y). After computing these two parameters setup and hold checks can be performed.

Lets find out the maximum frequency in order to check for setup. The maximum frequency for this circuit will be

$$T_{clock} = t_{pcq} + t_{pd} + t_{su} = (50 + 105 + 60)ps \tag{13.19}$$

Thus the circuit can operate at maximum frequency of $f = 1/T_{clock} = 4.65GHz$.

The hold check up is performed by satisfying the following hold criteria

$$t_{ccq} + t_{cd} > t_h \tag{13.20}$$

According to the data available following equation can be written

$$(30ps + 25ps) < 70ps \tag{13.21}$$

Thus the circuit fails to meet the holding timing constraint.

The circuit can be improved to meet the holding timing constraint. In order to meet the hold criteria, the contamination delay of the circuit must be increased. This can be done by placing buffers before the XOR gates in the shortest paths. Insertion of buffers is shown in Fig. 13.28. Here, assume the buffers has same amount of contamination delay and then the overall contamination delay is $t_{cd} = 2 \times 25ps = 50ps$. Then the above hold equation can be written as

$$(30ps + 50ps) > 70ps \tag{13.22}$$

Hence, the optimized circuit is now meets the hold criteria by adding buffers.

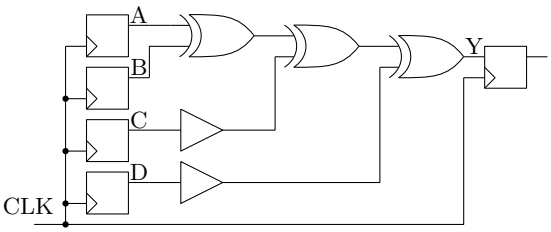


Fig. 13.28 Optimized circuit for Fig. 13.27 for meeting hold criteria

13.8 Frequently Asked Questions

Q1. **What are setup and hold checks for clock gating and why are they needed?**
A1. The clock gating circuits should be carefully designed such that the shape of the clock is not harmed and no glitch is produced. A scheme of clock gating is shown in Fig. 13.29. In order to avoid glitches, the enable signal has to be asserted in advance of the clock rising edge. This way the rising edge of the clock is protected. This is called as clock gating setup or clock gating default max check. Similarly, the disabling of the enable signal must be done in such a way that the falling edge is unharmed. This is called as clock gating hold or clock gating default min check.

Q2. **Why hold time is not included in the calculation for the maximum frequency?**
A2. Setup check fails when the clock period is less than the maximum timing path. Thus setup failure is frequency depended. But hold check is frequency independent and thus not included in maximum frequency calculations.

Q3. **One chip which came back after being manufactured fails setup test and another one fails a hold test. Which one may still be used how and why?**
A3. Setup failure is frequency dependent. If certain path fails setup requirement, you can reduce frequency and eventually setup will pass. This is because when you reduce frequency you provide more time for the flop/latch input data to meet setup. Hence we call setup failure a frequency dependent failure. While hold failure is not frequency dependent. Hold failure is functional failure.
Q4. **Perform the hold analysis for the example circuit shown in Fig.** 13.30?

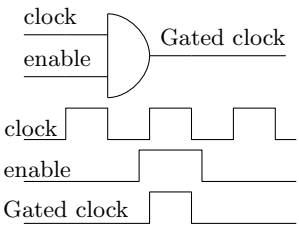


Fig. 13.29 An simple scheme of clock gating

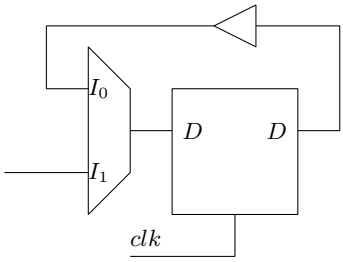


Fig. 13.30 Example circuit configuration for timing check

A4. In case of the example circuit shown in Fig. 13.30 tool can report hold violation or not. Hold violation normally arises when the generated and the sampling clock are different. But in this circuit both the clocks are same. This timing path will not have a real hold violation as same clock edge is referred here. This path would never have a real hold violation as we are referring to the same CLK edge. We will never have a hold violation as long as combined delay for t_{c2q}, t_{buf} and t_{mux} is more than the intrinsic hold requirement of the flip-flop.

Q5. **What is WNS and TNS in case of timing analysis?**
A5. Worst Negative Slack (WNS) is a parameter which is checked to see if the design meets timing or not. A positive value of WNS indicates that the design meets timing. On the other hand Total Negative Slack (TNS) is summation of all the negative slacks. If it is zero then the design meets timing. It is better to target TNS instead of WNS as TNS reduces all slack of all the paths.

Q6. **Give a example of a simple timing constraint file (SDC file)?**
A6. A simple example of timing constraint file which is generally uses the extension of .sdc as shown in Fig. 13.31 for a simple counter.

```
# Specifying process parameters
set_units -capacitance 1000.0fF
set_units -time 1000.0ps
create_clock -name "CLKIN" -add -period 10.0 -waveform {0.0 5.0} [get_ports clk]
set_clock_transition -min 0.2 [get_clocks CLKIN]
set_clock_transition -max 0.25 [get_clocks CLKIN]
set_clock_gating_check -setup 0.0
set_clock_uncertainty -setup 0.2 [get_clocks CLKIN]
set_clock_uncertainty -hold 0.2 [get_clocks CLKIN]
# Specifying external delay for inputs
set_input_delay -clock [get_clocks CLKIN] -add_delay 0.3 [get_ports reset]
set_input_delay -clock [get_clocks CLKIN] -add_delay 0.3 [get_ports en]
# Specifying external delay for outputs
set_output_delay -clock [get_clocks CLKIN] -add_delay 0.3 [get_ports out[0]]
set_output_delay -clock [get_clocks CLKIN] -add_delay 0.3 [get_ports out[1]]
set_output_delay -clock [get_clocks CLKIN] -add_delay 0.3 [get_ports out[2]]
set_output_delay -clock [get_clocks CLKIN] -add_delay 0.3 [get_ports out[3]]
```

Fig. 13.31 Timing constraint file for a simple up counter

13.9 Conclusion

In this chapter, basic of timing verification of digital system designs is discussed. STA is a popular technique for timing verification. First different terms related to STA are explained and then criteria for setup and hold check-ups are discussed. Timing verification is very important to guaranty a successful hardware implementation. There are many topics on STA which cannot be covered in this chapter but readers are advised to cover those topics also to have a very clear view on STA.

Chapter 14
Digital System Implementation

14.1 Introduction

In the previous chapters, we have discussed the theoretical details of different digital architectures. In this chapter, details of implementation platforms will be discussed. The algorithms in the field of signal processing, image processing or in the field of communication must be implemented on some hardware platforms to achieve faster execution speed. These implementations must be optimized in terms of power and area to save battery life and cost, respectively.

Initially the algorithms are preferred to be implemented on Central Processing Units (CPU) (e.g. Intel Processors). The use of multi-core processors improves the execution time. Researchers propose the use of Graphics Processing Units (GPUs) (e.g. NVIDIA GPUs) to further increase the execution speed. It is reported that performance of GPU-based implementations has better speed compared to CPU-based implementations. It is further reported in literature that the multi-core controller units can be used as accelerators for GPU-based implementations. Details of these implementation platforms can be found in [16, 39, 40].

The above-mentioned platforms are still not suitable for many real-time applications because of high demand of processing speed. Further speed can be achieved by using greater parallelism and this can be achieved using dedicated Integrated circuits (IC). The ICs can be broadly classified as

1. Custom ICs
2. Semi-Customs ICs
3. Standard Cell-Based ICs
4. Gate array-based ICs.

In this chapter, we will discuss a brief theoretical background on FPGA implementation and ASIC implementation using standard cells.

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_14

14.2 FPGA Implementation

As the name suggests, FPGA contains arrays of reconfigurable logic blocks arranged as a matrix. The reconfigurable logic blocks are programmed in such a way that a logic expression is realized. Any logic functions can be realized using FPGA which can be executed in parallel. FPGAs do not have any operating system like dedicated processors. In dedicated processors, the organization or the connectivity of the resources is fixed. But in FPGA, an user connects the wires with some programming technology to implement a logic function. User can reprogram or modify the connectivity according to the logic to be implemented. Based on the techniques of programming, FPGA devices can be classified as

1. Anti-fuse Based: Initially all the contacts are open and selected locations are conduct when programmed. This One time Programmable (OTP) technique is non-volatile and does not use a standard CMOS process.
2. SRAM Based: In Static RAM (SRAM) based FPGAs, static memory cells act as the basic cells. Presently most FPGA makers use SRAM-based technique. Here SRAM cells serve the following purposes
   (a) Programming of the Configurable Logic Blocks (CLBs) to implement a logic function.
   (b) Programming the connectivity or routing to connect the CLBs according to a logic function.
3. E2ROM or Flash based: Flash-based programming is non-volatile and area efficient than SRAM-based programming. But flash-based FPGA devices cannot be configured infinite times and also flash-based devices use non-standard CMOS process.

14.2.1 Internal Structure of FPGA

A simplified structure of an FPGA device is shown in Fig. 14.1. An FPGA consists of three major components which are.

1. Configurable Logic Blocks (CLB), which implement logic functions.
2. I/O blocks (IOB), which are used to make off-chip connections.
3. Programmable Routing (interconnects), which connects I/O blocks and CLBs.

Configurable Logic Block

CLBs inside the FPGA device are used to realize any logic expression. A CLB can be based on basic logic gates, MUX based or LUT based depending upon the technology. A term granularity is defined to indicate size of a CLB (basic unit) and depending on this, reconfigurable devices can be of following types.

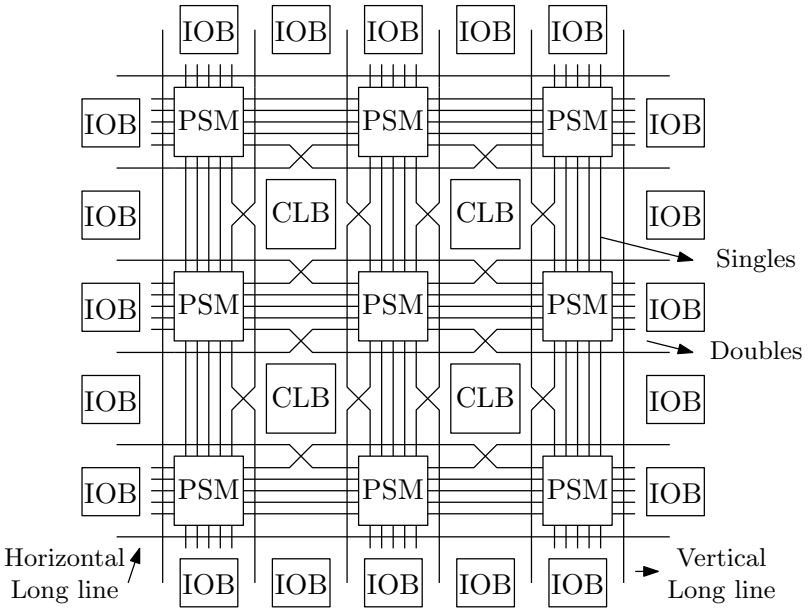


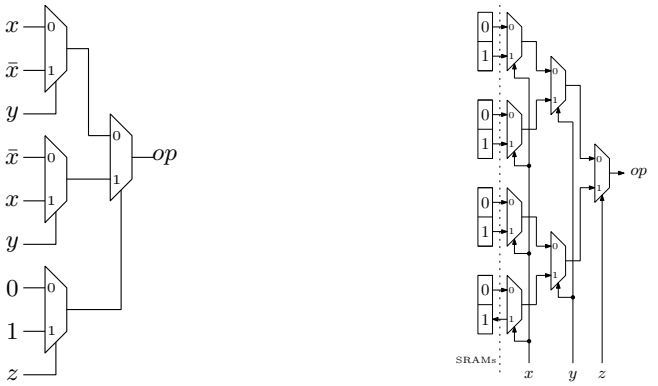
Fig. 14.1 Simplified architecture of an FPGA device

1. Fine Grained—Universal gates like NAND or AND-OR-NOT are basic blocks.
2. Middle Grained—CLBs are either based on MUX or ROM/RAM.
3. Coarse Grained—A processor (like FFT) work as a basic unit.

Trade offs: Fine grain FPGAs are having more interconnection overhead whereas coarse grain FPGA devices are application specific. Two types of CLBs are shown in Fig. 14.2 which implements the function

$$op = x \oplus y \oplus z \tag{14.1}$$

The major element of a CLB is a Look-up Table (LUT) which implements a logic function using its truth table. Memory elements also can be implemented using LUTs. Fast arithmetic operations can be implemented using inbuilt carry and control logic. CLB also contains flip-flops which are required to implement sequential logic blocks. Each CLB of Spartan 3E (xc3s500e) FPGA has four slices, as shown in Fig. 14.3. These slices in Spartan 3E FPGA are grouped in two pairs which are SLICEM and SLICEL. SLICEM which is also the left side pair supports both logic and memory functions (RAM16 and SRL16). The right side pair (SLICEL) supports only logic functions. Slice of the Spartan 3E FPGA device has two 4-input LUT function generators (G and F) and two programmable storage elements with other control circuitry. Both SLICEM and SLICEL have the following common functionalities:



(a) MUX based CLB construction.

(b) LUT based CLB construction.

Fig. 14.2 Different ways of constructing CLBs

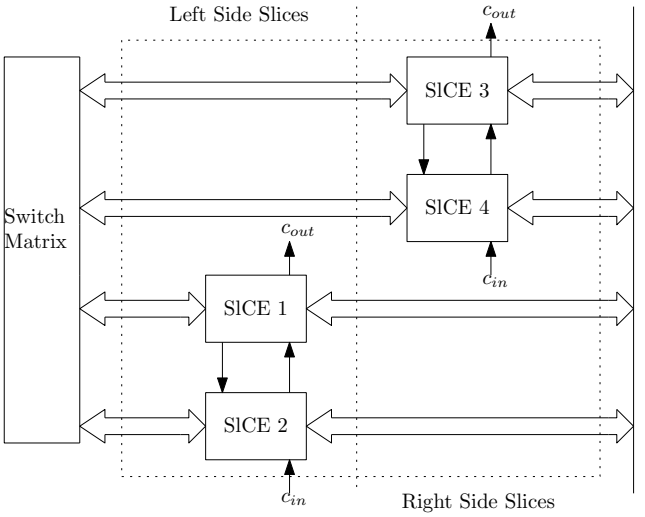


Fig. 14.3 Spartan 3E CLB architecture

1. Two 4-input LUT function generators (F and G)
2. Carry and arithmetic logic
3. Two storage elements
4. Two wide-function multiplexers.

The SLICEM supports two additional functions:

1. Two 16-bit shift registers (SRL16).
2. Two 16×1 distributed RAM blocks (RAM16).

Realization of a half-adder with inputs a and b is shown in Fig. 14.4 for slice of Spartan 3E FPGA. The summation output (sum) of the half-adder is computed by G-LUT and carry out (co) output is computed by F-LUT. The dotted lines are connected to realize the half-adder. A 'Logic Cell' is defined as combination of a storage element and a LUT.

Input/Output Block

Input/Output Block (IOB) connects a package pin to the internal logic of the FPGA. IOBs are programmable and can be unidirectional or bi-directional with various interacting techniques. A block diagram of Spartan 3E IOB is shown in Fig. 14.5. IOB has three main signal paths which are output path, input path and 3-state path.

1. Data from the outside world (IO pad) enters the FPGA through the input path. A programmable delay block placed to delay the input signal as per the requirement. There are two registered inputs which are $IQ1$ and IQ. Both the flip-flops have separate clock signals. These flip-flops can be used for logic also through pin $TDDRIN$.
2. Through output path, data from the FPGA device exits the FPGA IC and terminate on an external pad. Similar to the input path, registers are also can be placed before exiting the FPGA IC. Output signals can be passed through DDR MUXes or buffered.
3. This path is for controlling output path and the high impedance state of the output path is determined by this path. This path is also can be registered or non-registered just like the output path.

Programmable Interconnect

Apart from the CLB and IOB, programmable Interconnect is the another major block in an FPGA chip. Programmable interconnect connects all the CLBs and IOBs inside the chip. These connections are programmable through any of the techniques mentioned above but generally the SRAM-based programming is mainly used. Programmable interconnect has mainly two sections which are Programmable Switching Matrix (PSM) and Interconnect Lines. These are discussed below

1. Programmable Switch Matrix (PSM): The major element of the programmable interconnect in the FPGA is the PSM. All the CLBs, IOBs or DCMs are connected through the PSM. PSM directs a connection from one direction to another. The basic scheme of the PSM is shown in Fig. 14.6. Here input/output signals of the

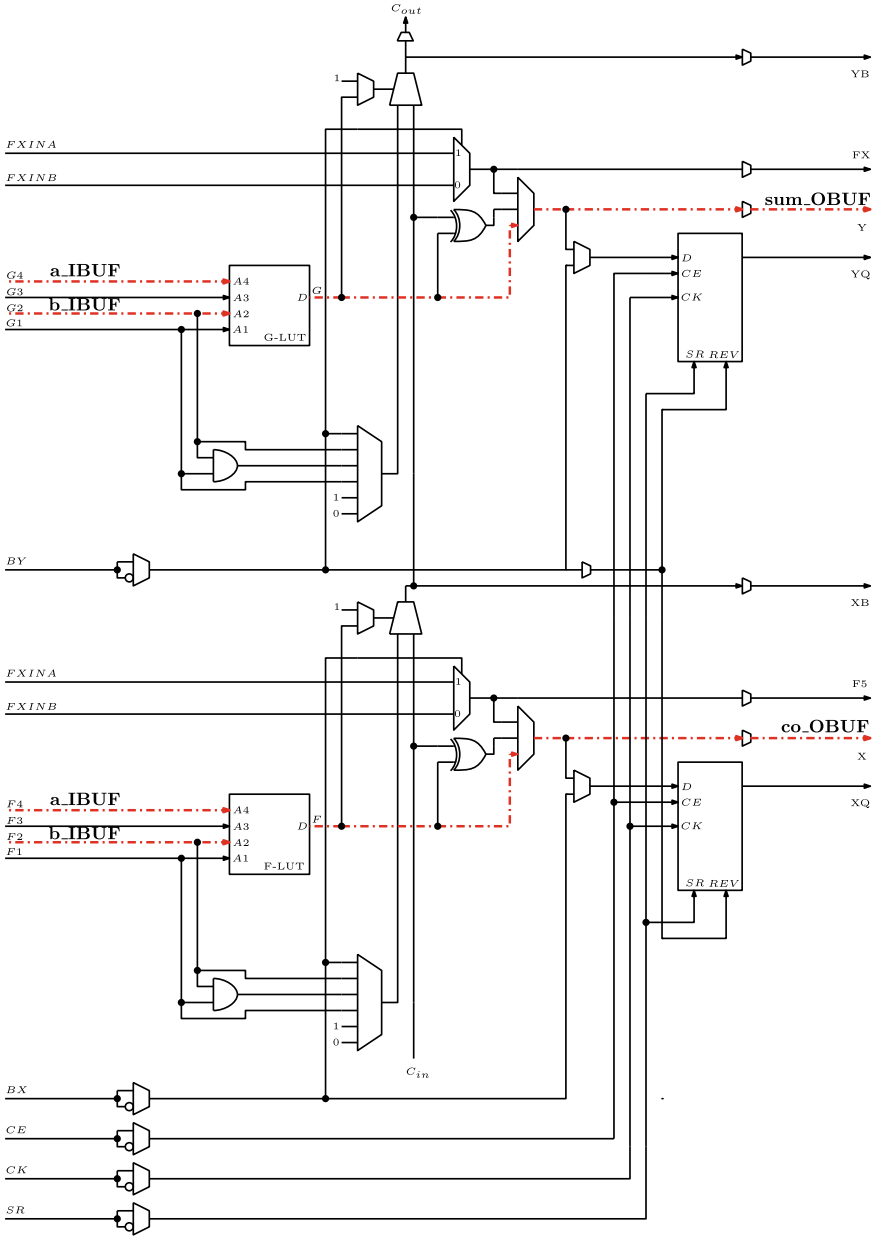


Fig. 14.4 Architecture of slice of Spartan 3E CLB

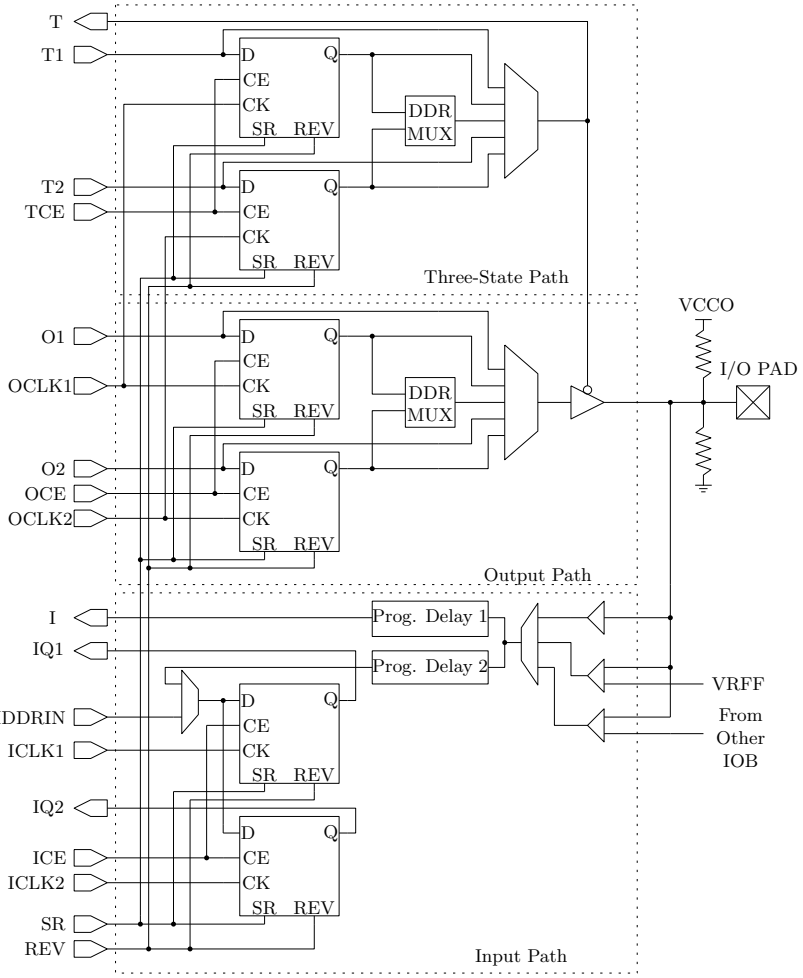


Fig. 14.5 Spartan 3E IOB architecture

CLB can be connected through any of the PSM. PSM is programmed through various pass transistors as shown in Fig. 14.6. PSM along with other different interconnect lines form the programmable interconnect.

2. Interconnect Lines: The interconnect lines which make the connectivity in a FPGA are Long lines, Hex lines, Double lines and Direct connections. Programmable interconnect consists of many sets of long lines which spans the entire die. These lines can be horizontal or vertical. These lines have low capacitance and thus suitable to carry high frequency signals with minimum loading effect. Global clock connections can be done through these lines to have low skew. In case of Hex lines signals can be started from one end only. Double line covers the

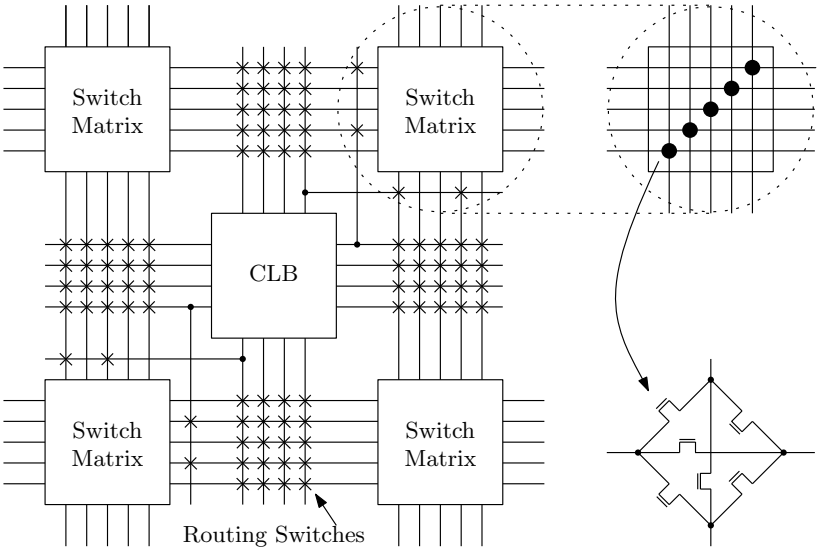


Fig. 14.6 A possible connection for programmable switch matrix

majority of connections and has more flexibility than Long and Hex lines. Direct connections are for local connections among CLB, PSM and IOB.

14.2.2 FPGA Implementation Using XILINX EDA Tool

In this section, FPGA implementation of digital systems using XILINX EDA tool is discussed. A simplified version of FPGA-based design flow according to XILINX EDA tool is given in Fig. 14.7.

Design Entry

There are different techniques for design entry.

1. Schematic based.
2. Hardware Description Language (HDL).
3. Combination of both.

Designer can choose any of the methods for design entry. Initially, all the EDA tools were offering schematic-based design entry. But nowadays as digital circuitries are getting more complex, language-based design entry is the only option available to the users. But schematic-based entry has certain advantages like it gives designer

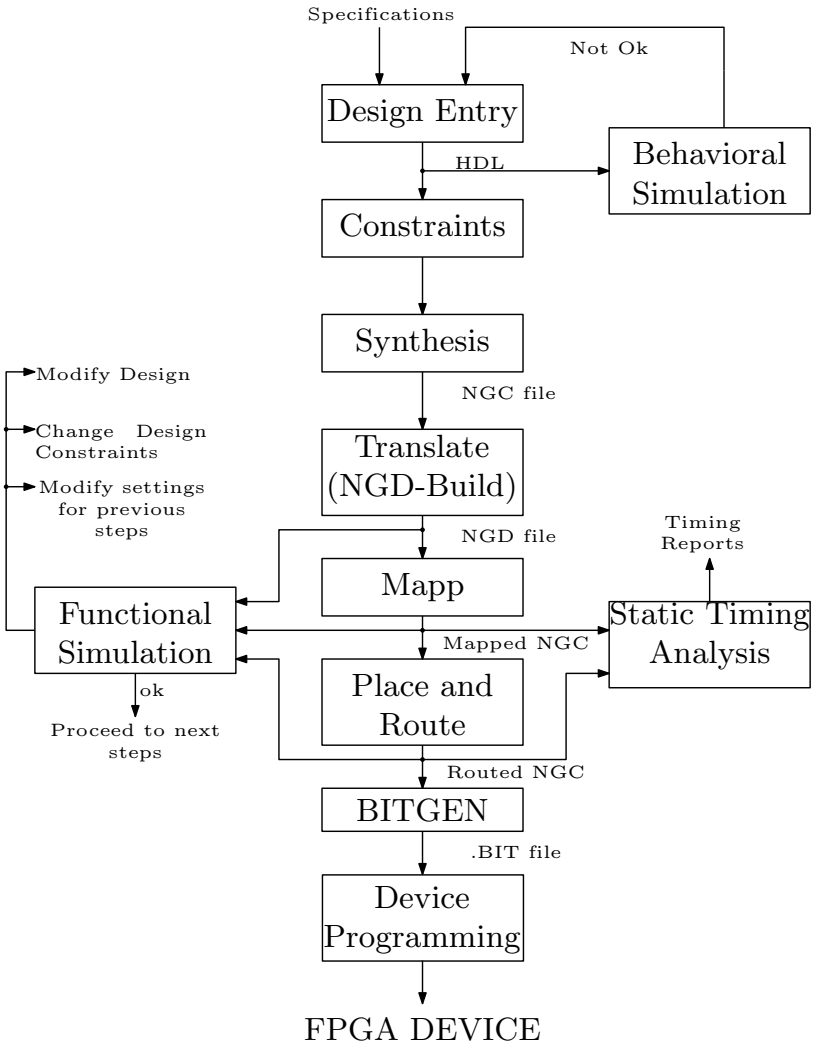


Fig. 14.7 FPGA implementation flow chart

more idea about hardware. But for complex designs language-based designs are better suitable. There are three popular HDL languages which are VHDL, Verilog, and System Verilog. HDL-based entry is faster but lags in performance and density. Some EDA tool offers state machine-based design entry where a system is broken into series of states. But this method is very rare. In this book, we have followed design entry using Verilog or System Verilog.

Synthesis

Synthesis is a process which receives the HDL code and translates into a netlist file where the design is realized with actual hardware elements like gates, flip-flops, etc. Sometimes a complex design might have more than one sub designs. For example, in order to implement a processor we need CPU as one design element and RAM as another design element. Netlist for each design element is generated in the synthesis process. In the synthesis step, syntax and hierarchy of the design are checked. Logical optimization also can be performed in this step. The resulting netlist(s) is saved to a Native Generic Circuit (NGC) file (for Xilinx Synthesis Technology (XST)). The synthesis step gives an estimate of the hardware utilization in terms of LUT and registers consumed. Implementation step gives more accurate hardware utilization report.

Implementation

This process consists of a sequence of three steps

1. Translate process—In the translate process, all the input netlists and constraints files are combined to a logic design file. A Native Generic Database (NGD) file saves this information. This is done using NGD Build program. There are mainly two kinds of constraints which are hardware and timing constraints. In hardware constraints, ports in the design are connected to the physical components like pins, switches, buttons, etc. All the timing requirements for the design are written as the timing constraints. These constraints are written in file formats like User Constraints File (UCF) or Synopsys Design Constraint (SDC) file.
2. Map process—In the Map process, the design that contains the logical elements is subdivided into subblocks such that they can be fitted into the FPGA logic blocks. Map process tries to fit the logic defined by the NGD file into targeted FPGA elements (CLB, IOB) and also generates a Native Circuit Description (NCD) file. NCD file physically represents that the design is mapped to the components of FPGA. All the constraint related information is saved in another file called Physical Constraints File (PCF).
3. Place and Route (PAR)—It is a combination of two processes, place and route. The first process tries to place the subblocks generated from the Map step into the logic blocks according to the constraints defined in the constraint file. The route process tries to build connection between the subblocks or between physical buttons to input/output ports. PAR process tries to place and route the design in such a way that design meets timing requirements mentioned in the constraint file. For example, if a sub block is placed in a logic block which is very near to I/O pin, then it may save the time but it may affect some other constraint. So, a trade off between all the constraints is taken into account by the PAR process. The PAR tool takes the mapped NCD file as input and produces a completely routed NCD file as output. The routing information is saved in output NCD file.

Hardware utilization summery is generated after the synthesis process and a more detailed summery is generated after Map process. Several optimization algorithms are run during the Map process which can trim or remove irrelevant, duplicate and unused logic elements. EDA tools compute maximum frequency at which the design can be operated. Maximum frequency post-synthesis process and post-Map process can be the same or different depending on the complexity of the design.

Device Programming

Once the implementation process is done, the next step is to load the design to an FPGA target. The design after the Map process is converted to a format that is acceptable to FPGA. A BITGEN program takes the routed NCD file and converts it to a bit stream file (.bit file). This file is then loaded to the FPGA device using programming cables.

14.2.3 Design Verification

Verification can be done at different stages of the implementation process.

1. Behavioural Simulation (RTL Simulation): In this step, behaviour of the design is tested. The design that is needed to be tested is selected as top module. This top module is also called as unit under test (UUT). Some input test vectors are given to the UUT and output is observed in a test bench simulation window. If outputs are satisfactory then we can say that the design is correct. But this step does not reveal anything about timing constraints and thus it does not guarantee that the design will work when implemented on FPGA.
2. Functional simulation: Functional simulation gives information about the logic operation of the design. Designer can verify the functionality of the design using this process after synthesis and implementation step. Functionality of design must be checked as both synthesis and implementation step run some optimization algorithm to optimize the design. In this optimization process, functionality of the design may be changed. If functionality is not as expected, then designer has to make changes in the code and again follow the implementation steps.
3. Timing simulation: A design may not give correct results even though behavioural and functional simulation are giving satisfactory results. This is because timing constraints are not checked in the previous simulations. Timing simulation can be done after synthesis and implementation step. In this simulation, all the timing constraints are analysed. This generates reports if there is any timing violation. This timing analysis is sometimes called as static timing analysis (STA). STA helps designer to achieve higher maximum frequency and also in troubleshooting.

14.2.4 FPGA Editor

FPGA Editor is a graphical tool for displaying and configuring FPGAs. The FPGA Editor reads from and writes to NCD files, macro files (NMC) and PCF files. Following functions can be performed using FPGA Editor.

1. Place and route critical sub-modules before running the automatic place and route tools.
2. Finish PAR process if the routing program does not completely route your design.
3. Add probes to the design to examine the signal states of the targeted device. Designers use probes to route the value of internal nets to an IOB to analyse the design during the debugging process.
4. Run the BitGen program and download the generated bit stream file to the targeted device.
5. View and change the nets that are connected to the capture units of an Integrated Logic Analyzer (ILA) core in your design.
6. Create an entire design manually (advanced users).

Figure 14.8 represents a fully routed implementation of a simple half-Adder. The bigger rectangles are the PSMs. Total four number of IOBs are used where IOB-I stands for input and IOB-O are for output. Out of four SLICES of a CLB, only one SLICE is occupied.

14.3 ASIC Implementation

In the earlier section, FPGA implementation steps are discussed. FPGA based hardware platform is very useful for rapid prototyping of digital circuits. But if FPGA is used for application specific implementation, then FPGA has the following limitations

1. The area is fixed for simple to complex circuits.
2. Custom input/output ports cannot be available.
3. Limited number of input/output ports.
4. Slow speed if same technology used for FPGA and custom IC design.
5. Problem in interfacing with mixed-style designs.

Thus ASIC implementation of digital systems is important for application specific designs. Standard cell-based design methodology is suitable for complex digital circuits. In FPGA implementation using EDA tools like Vivado, tool takes care of every design issues. But in case of ASIC implementation, designer has to address many design related issues even though there are tools for automatic implementation. The tools used for ASIC level design are Cadence, Synopsys, Mentor-graphics, etc. In this work, we will discuss the Cadence tool-based ASIC implementation. The standard cell-based ASIC implementation can be divided in two following major processes

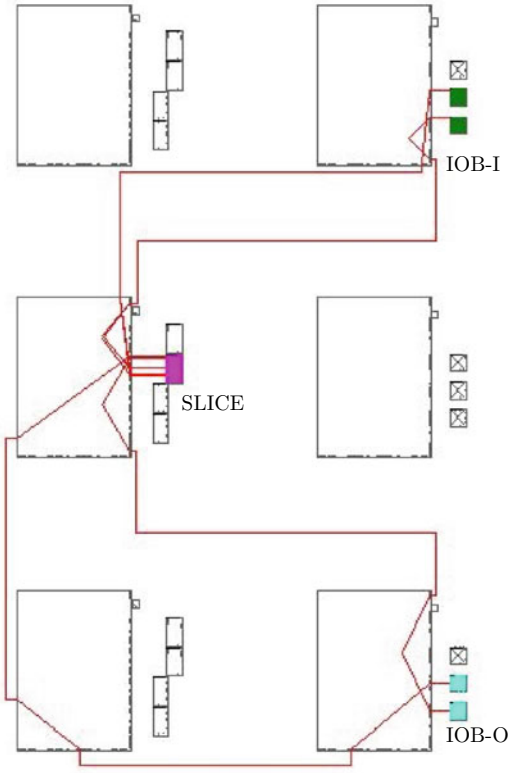


Fig. 14.8 Placed and routed design of half-adder observed using FPGA editor

1. Simulation and Synthesis
2. Placement and Routing.

14.3.1 Simulation and Synthesis

Simulation

Verilog files are simulated by the same way as they were simulated in case FPGA implementation but in this case standard cell libraries are invoked. The simulation can be failed if there are syntax errors. Linting is a process of checking syntax errors in case of ASIC design. Another important operation to perform is code coverage which tells that how well the design is exercised or covered by the test bench.

Synthesis

Synthesis step is performed only when the behavioural simulation of the design shows correct result. This process is almost same as that in case of FPGA implementation flow and Fig. 14.9 shows the synthesis process. Two types of files are required in synthesis process to proceed. First type of files are library files and second file contains the timing constraints. Library files carry the specifications about the standard cells. In an SDC file, timing constraints are written which is mentioned in the previous chapter. Design for testability (DFT) analysis can be performed by inserting scan cells to the design in this step. Figure 14.10 shows a simple example of testing a sequential design. In this example, the path from input to any flip-flop can be seen at output when the *scan_en* signal is high. This technique is popularly known as MUX based scanning. Here, three extra inputs and outputs along with four multiplexers are included. This extra logic overhead must be considered if a design is to make

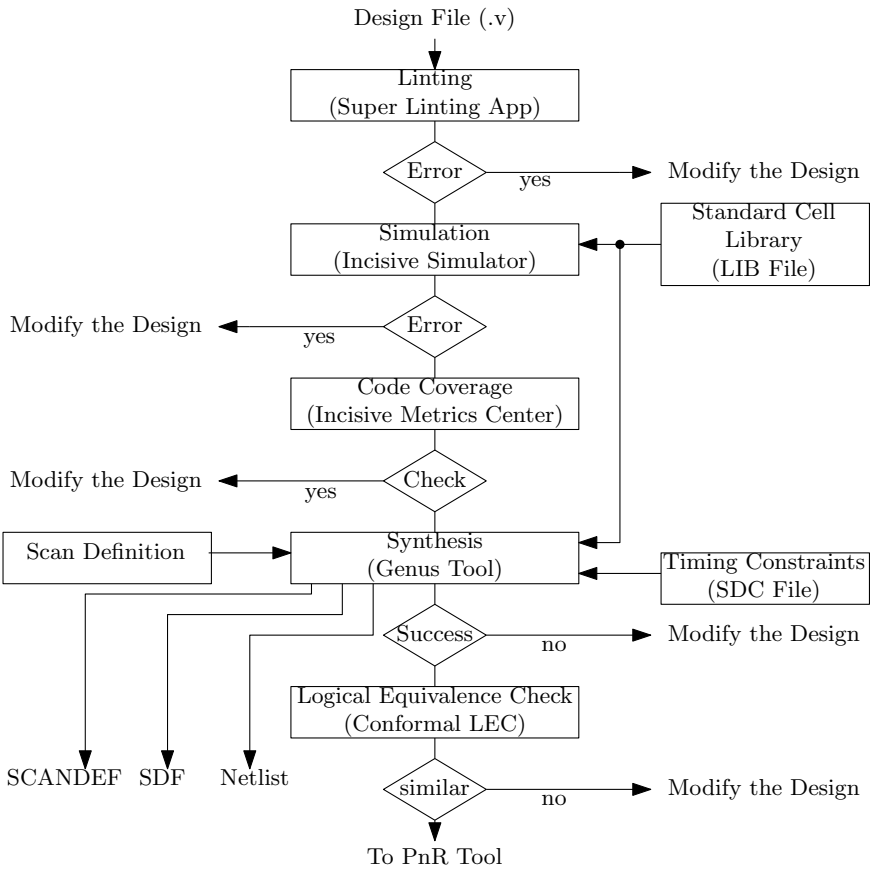


Fig. 14.9 Simulation to synthesis flow using Cadence EDA tool

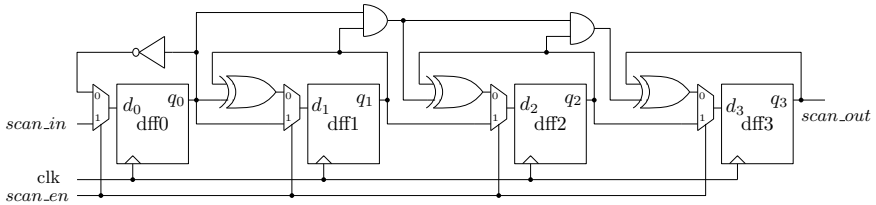


Fig. 14.10 An example of MUX-based scanning for a simple up counter

testable. Automatic Test Pattern Generator (ATPG) can also be used to verify the full functionality during synthesis process. This ATPG testing is performed by Cadence Modus Test tool. The synthesis process generates the following three kinds of files

- Design Netlist—Netlist file gets the same extension that the original design file has. Netlist file implicates how the design is mapped using the actual standard cells.
- SDF files—Timing related data of the original design are stored in a Standard Delay Format (SDF) file.
- SCANDEF Files—The scandef file keeps the information about scan cells and the ordering of the scan chains. This file is used in the PnR stage.

Design performance parameters like logic count, area, and power are estimated at synthesis process and separate reports can be generated. Pre-placement timing analysis is performed during the synthesis step based on the timing constraints. Design should meet timing requirements and the tool may optimize the design for meeting timing checks. Another step called Logic Equivalence Check (LEC) is performed after synthesis. This step checks the similarity between the original design and the netlist file that is generated after synthesis process.

14.3.2 Placement and Routing

Placement and routing of the design can be done only after the synthesis process is successfully executed. Placement and routing using Cadence Innovus tool is discussed in this section. Figure 14.11 shows the overall flow for placement and routing. Each step is explained one by one below.

Design Initialization

The first step of the design initialization is to import the design files. The Netlist file that is generated after the synthesis process is the main design file to the PNR tool. Only the post-synthesis netlist file is not enough for the PNR tool. There are some other files which should be imported along with the Netlist file. These files are

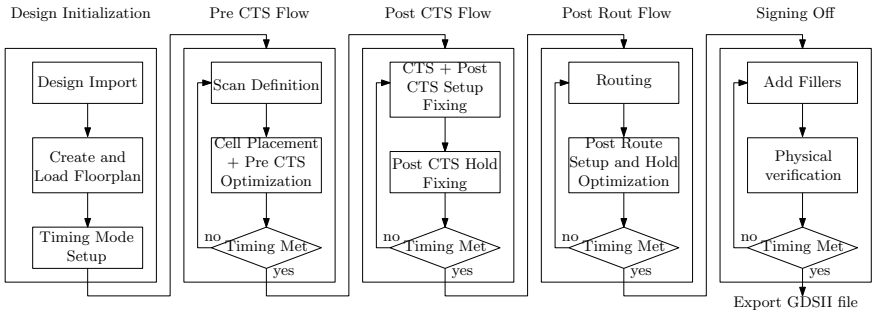


Fig. 14.11 Flow chart for placement and routing (PnR) for ASIC implementation

1. LEF Files: Library Exchange Format (LEF) contains the physical layout information of an IC in ASCII format. Abstract information about the standard cells or I/O pads and design rules for layout are stored in LEF files. Three kinds of LEF files are mentioned below

   (a) Technology LEF File
   (b) Standard Cell LEF file
   (c) I/O LEF file

2. I/O Assignment File: I/O assignment file is used for custom assignment of I/O pads and corner cells (for example counter.io).
3. Multi Mode Multi Corner (MMMC) View Definition File: The MMMC view definition file (for example counter.view) holds pointers to two type of files which are

   (a) Liberty Timing Models LIB Files: The information of a cell, e.g. name of the pins, name of the cell, loading information of the pins, and cell area are stored in the liberty files. These files also carry the necessary conditions for meeting timing check-ups such as maximum transition time of a net. These libraries also contain different wire load models and operating conditions along with Design Rule Constraints (DRC).
   (b) Synopsys Design Constraints (SDC): Different timing constraints are written in the SDC file to perform the STA.

   Three types of LIB files can be available which are fast-fast, slow-slow and typical for both IO and standard cells. Considering these libraries along with SDC files and PVT variations two different timing libraries can be set which are MIN and MAX. Considering different RC values two different $RC\_CORNERS$ are set which are RC_WRORST and RC_BEST. Two types of delay models MIN_TIMING and MAX_TIMING are created by considering the above mentioned $RC\_CORNERS$ and timing libraries. This way two different analysis views are formed which are setup and hold.

4. Power Management Files: In low power designs, we need another file to achieve low power which is Common Power Format (CPF). Different power saving techniques and power constraints are defined in this file. Multiple power domains according to multiple supply voltages are also mentioned in this file.

Create and Load Floor Plan: If the design files are loaded then the area can be automatically calculated or can be customized. Logic area is automatically calculated based on core utilization factor. This factor is defined as the ratio of the area of the design (area of standard cells + area of macro cells) to the core area. In custom mode, designers have to specify the dimension for floor plan.

Figure 14.12 shows a simplified floor plan where it is divided into two areas which are the core area and the I/O ring around the core area. Rings are required to carry supply (Vdd) and ground (Gnd) signals around the core area. The stripes take connection of Vdd and Gnd inside the core area. Each cell in the core area gets Vdd and Gnd connection via the power rails. I/O ring is also shown in Fig. 14.12. The blocks at the four corners are termed as corner cells. Different orientations of the corner cells are available in the I/O library. I/O pads are placed for metal tape-outs and I/O filler cells are placed between the I/O pads.

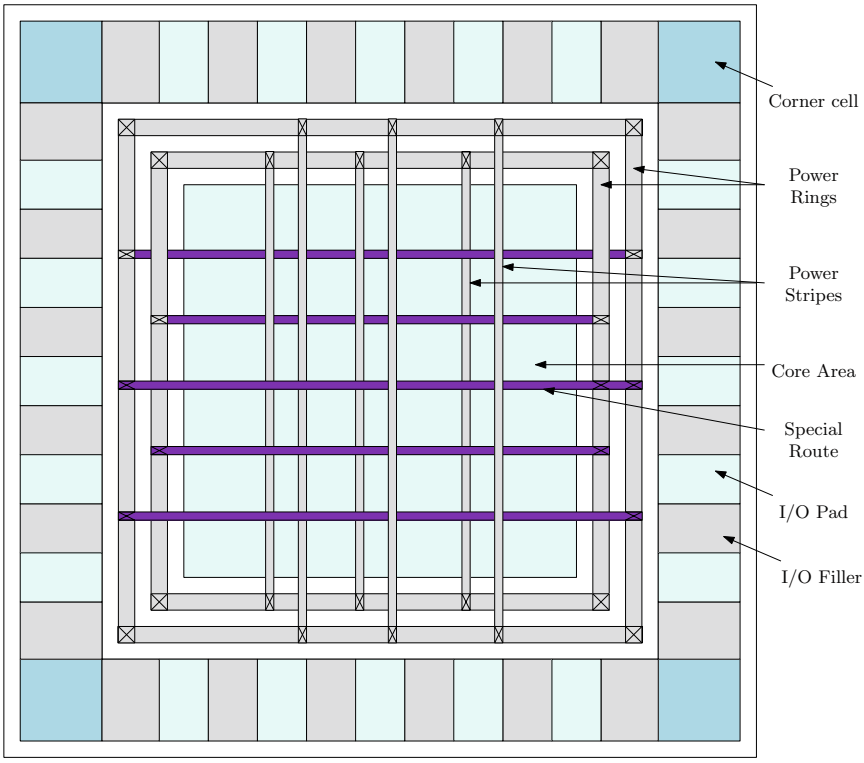


Fig. 14.12 A simplified floor plan for a standard cell-based digital IC

Pre-CTS Flow

In this flow, scan path functionality of the design is checked first. Scan cells are generally defined in the timing libraries but they can be defined during this flow also. In the synthesis stage, scan cells and the scan chains are inserted in the netlist. Designer can change the scan chain order in this flow. This means designer can change the order of how the flip-flops are connected along the scan chain for a single scan group or for all the scan groups. This process changes the connection constraints regarding the scan cells not the placement constraints. The reordering of scan chains can be performed in the following two ways

- Native Scan Reordering Approach—If scandef file is not available then this method is applicable. In this approach, the designers have to specify information regarding the scan cells, scan paths, and scan chains.
- Scandef Method—This approach takes the help of scandef file to reorder the scan chains. Cadence prefers these methods as it is more easier to execute.

The design can be placed once the scan cells are configured. The tool offers various optimization steps for the design. Placement of the design can be done with or without optimization. Once the optimization step is completed, the design is checked for timing. Timing checks are performed to see whether the design meets the setup and hold criteria. Design should satisfy the timing checks otherwise modification has to be performed in the Pre-CTS flow or designer can change the design. If design meets the timing then Post-CTS flow can be started.

Post-CTS Flow

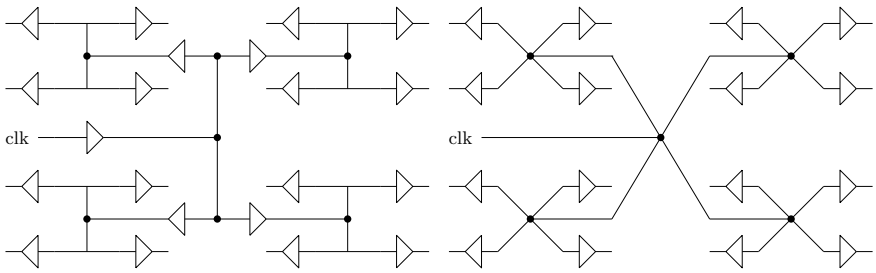
Clock Tree Synthesis (CTS) is an important step that must be successfully executed. CTS is a process of building physical structure between all the sink flip-flops and the clock source. CTS is always performed after placing and fixing the location of the standard cells. CTS is performed to ensure that the clock signal is distributed evenly to all the sequential elements.

The goal of the CTS is to meet the followings

1. Design Rule Checks (DRC)—DRC are satisfied in such way to optimize transition delay, fan out and load capacitance.
2. Targets of Clock—The objective of clock tree distribution is to optimize clock skew and clock latency.

CTS ensures the following parameters

- Minimum clock skew.
- No pulse width and duty cycle violation.
- Minimum clock latency.
- Minimum clock tree power by minimizing the transition and latency.
- Minimum signal cross talk.

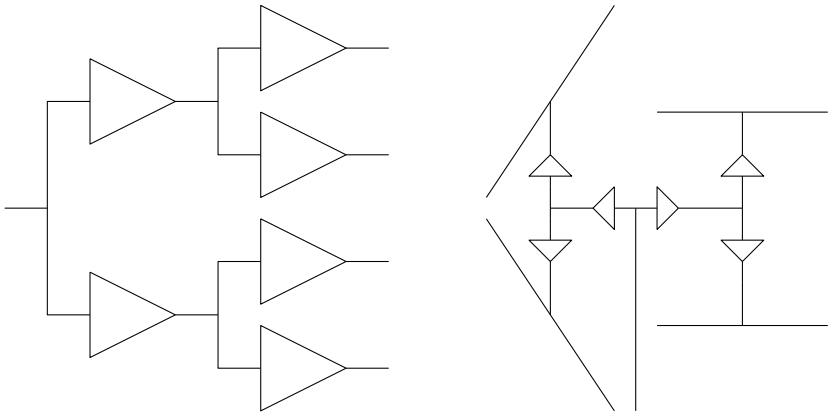


(a) H-type Clock Tree. (b) X-type Clock Tree

Fig. 14.13 Regular tree type clock distribution schemes

CTS chooses a suitable clock tree configuration out of many configurations to achieve these targets. These configurations are

1. Tree Structures: Tree structures are buffered as buffers restore the signal for better slew rates. Total insertion delay and total switching capacitance are less in case of tree structures. Following types of tree structures are there
   (a) H-Tree: H-Tree (Fig. 14.13a) provides equal timing path from source clock to destination flip-flop. Very low skew is guaranteed in case of H-tree but has very low floor plan flexibility.
   (b) X-Tree: X-Trees (Fig. 14.13b) are similar to the H-trees but not rectilinear.
   (c) Method of Mean and Median: This is an efficient algorithm to achieve H-tree by proper partitioning the module.
   (d) Geometric Matching Algorithms: Flip-flips are not symmetrically located and thus regular H-tree structure is difficult to achieve. Hence this algorithm gives an efficient way to obtain tapered tree (Fig. 14.14b).
   (e) Pi Configuration: In Pi configuration (Fig. 14.14a), total number of buffers inserted in a level is multiple times the buffers at previous level. This is a balanced configuration.
2. Mesh type: Initially, along the whole module a grid is formed and wire is laid along the grid lines. Figure 14.15 shows this type of clock distribution. Here, clock skew is determined by grid density and everywhere clock signals are available. Mesh type clock tree produces low value of clock skew and is tolerant to PVT variations. This clock distribution needs huge amount of wiring and power but they have good floor plan flexibility.
3. Spine type: Spine type clock distribution (Fig. 14.16) is a hybrid type of clock distribution which combines mesh type with tree structures. This type of tree structures have medium floor plan flexibility and medium capacitance but have higher skew.



(a) π-type clock tree. (b) Tapered H-type clock tree.

Fig. 14.14 Other type of tree type clock distribution techniques

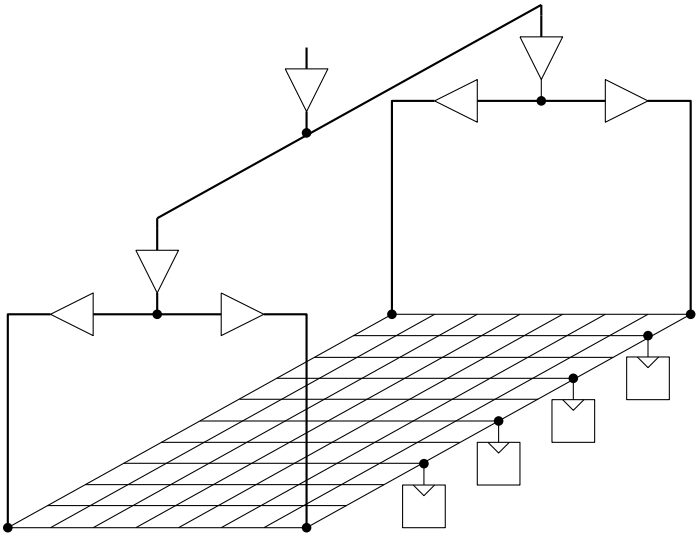


Fig. 14.15 An example of mesh type clock distribution

The clock tree optimization techniques are

1. Buffer and Gate Relocation—Timing performance can be improved by relocating buffers and gates. The arrangement shown in Fig. 14.17b is the modified version of the arrangement shown in Fig. 14.17 an after relocation. No new clock tree hierarchy is introduced in this operation.

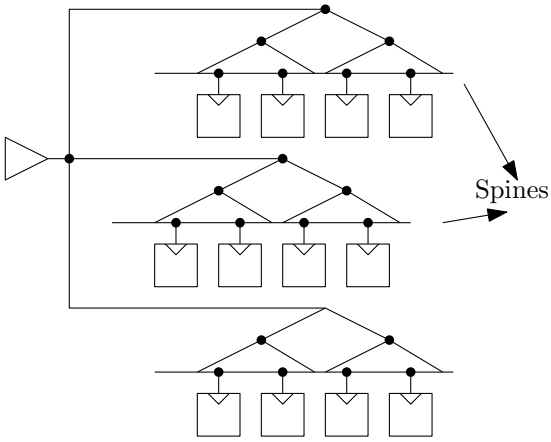


Fig. 14.16 An example of spine type clock distribution

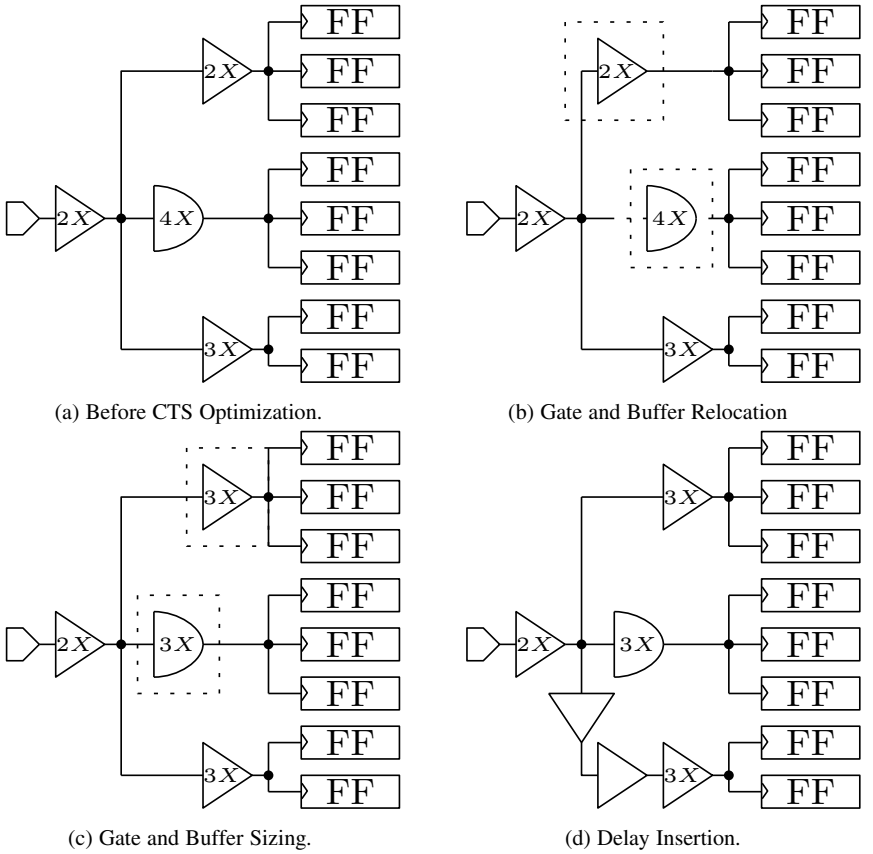
2. Buffer and gate sizing—Size of the buffers and the gates can be increased or decreased to improve skew and insertion delay as shown in Fig. 14.17c. Clock tree hierarchy is unchanged here.
3. Delay Insertion—Delay insertion for shorter paths is also a solution as shown in Fig. 14.17d. Here clock hierarchy is changed as new buffers are inserted.
4. Level Adjustment—In level adjustment (Fig. 14.18), level of the clock pins can be adjusted to upper or lower level. No new clock tree hierarchy is introduced.
5. Reconfiguration—Sequential elements can be clustered and then buffers can be placed as shown in Fig. 14.19.
6. Dummy load insertion—Insertion of dummy loads helps in load balancing. Dummy loads can be added to fine tune the clock skew by increasing the shortest path delay.

Timing checks are again performed after execution of the CTS. The design meets the setup timing criteria based on the delay of the critical path. CTS step takes care that the critical path does not get much delay. Host criteria depend on the minimum path of the design. CTS tries to fix the hold problem by inserting buffers in the minimum paths. Fixing the hold slack can violate the setup condition and vice-versa. If all the timing checks are met, then routing can be started or the previous steps can be executed again and again.

Post-Route Flow

Physical connections are made between the cells in the routing flow based on the logical connectivity. The objective of the routing is that all the paths should meet the timing constraints. Three types of routing processes are there which are

1. Power Routing—In the floor planning stage, routing of power nets is partially done but connection of Vdd and Gnd pins of a cell is done in the routing flow.



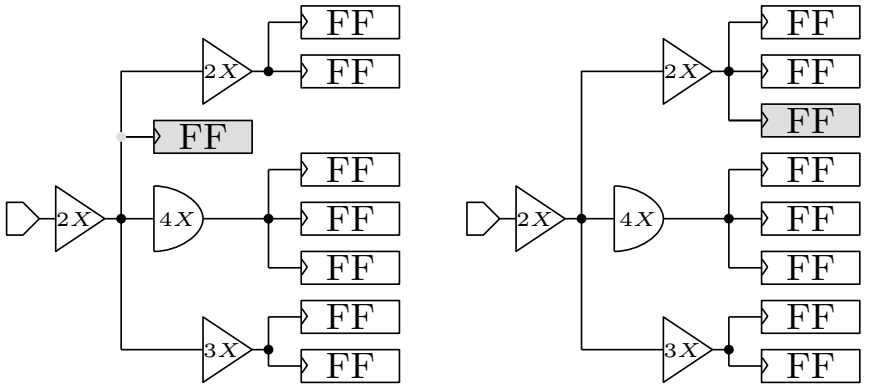
(a) Before CTS Optimization.

(b) Gate and Buffer Relocation

(c) Gate and Buffer Sizing.

(d) Delay Insertion.

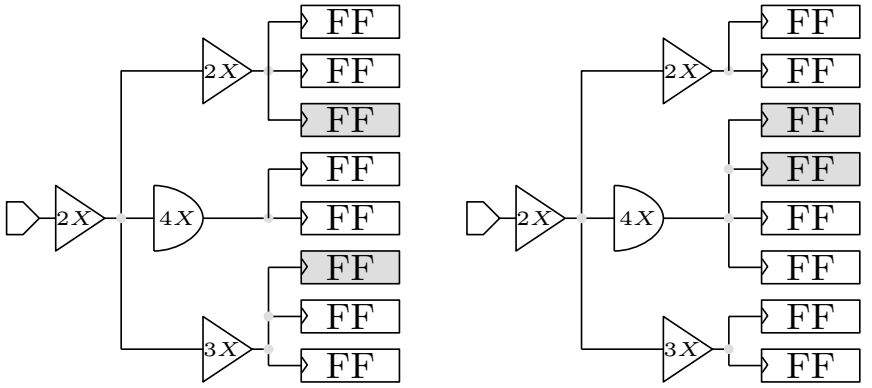
Fig. 14.17 Clock tree before CTS and after CTS optimization



(a) Before CTS Optimization.

(b) Level Adjustment.

Fig. 14.18 Clock tree before CTS and after level adjustment



(a) Before CTS Optimization. (b) Reconfiguration of Clock tree.

Fig. 14.19 Clock tree before CTS and after reconfiguration

2. Clock Routing—During the CTS flow, clock tree is formed and reorganized. Leaf nodes in tree are connected to clock signal in the routing stage.
3. Signal Routing—All the signal nets are routed in routing stage and critical nets can be routed in a special way if mentioned as exception.

The overall routing job is executed in the following three steps

1. Global Routing—Overall design is partitioned in small routing regions like tiles/rectangles in this step. Region-to-region paths are decided in a way to optimize the wire lengths and timing. Planning of the routing is actually done in this stage and no actual routing is done.
2. Track Assignment—In this step, metal layers replace the routing tracks assigned by the global stage in the previous stage. Tracks are assigned in vertical and horizontal directions. Re-routing is performed if overlapping has occurred.
3. Detailed Routing—Some paths still can violate the setup and hold criteria even if the metal layers are laid. In this stage, the critical paths are identified and fixed. This is an iterative process and iterations will continue until all the constraints are met.

Signing Off Flow

Post-route timing checks are performed after routing step. The timing constraints can be met or not met by a small margin. The signing off stage can be executed after the route step and this is the last stage of ASIC implementation. Filler cells are added in the core area and in the I/O ring and various sizes of filler cells are available in the library. Following checks are performed in this stage

1. Logical Checks—Two types of logical checks are performed and they are
   (a) Logic Equivalence Check (LEC)—LEC is again performed to check the logical similarity between the Netlist generated after post-route optimization and the actual Verilog file.
   (b) Timing Checks—Timing checks are again performed to check if the design violates any timing constraints (setup, hold or transition).
2. Physical Checks—Mandatory physical checks are
   (a) Layout Versus Schematic (LVS) Check—Actual devices are identified by checking shape and connectivity of a layout.
   (b) Design Rule Checks (DRC)—Layout should be generated as per the specified layout related rules otherwise DRC violation will occur.
   (c) Electrical Rule Checks (ERC)—ERC is performed to verify connections of all the power nets.
   (d) Antenna Check—The antenna effect is caused when a particular net gains higher potential than the operating voltage of the chip. This happens due to charge accumulation on an isolated node. There are techniques available to remove antenna errors.
3. Power Checks—The power checks are
   (a) Dynamic IR—IR drop is the difference of potential in the metal layers which constitutes the power grid.
   (b) Electromigration (ER)—The EDA tools take care of the ER checks and ensure it never occurs.

The design is ready for fabrication once all the physical checks are successfully verified. The layout of the IC can now be generated in the GDSII format. This format is sent to the foundry for fabrication.

14.4 Frequently Asked Questions

Q1. **What are the checks performed by a linting tool?**
A1. In general, a linting tool checks for following

1. Undriven and Unconnected ports
2. Port/net size mismatches
3. Unsynthesizable constructs
4. Case statement style issues
5. Unintentional latches
6. Incorrect usage of blocking and non-blocking assignments
7. Unused declarations
8. Driven and undriven signals
9. Race conditions

10. Incomplete assignments in subroutines
11. Set and reset conflicts
12. Out-of-range indexing.

Q2. **Explain the common Logic optimization errors and warnings that arise during the implementation of a Verilog design?**
A2. In this section, some of the common logical errors and warnings which arises when a digital system is checked using a linting software or while synthesis process is discussed. The errors must be addressed in order to proceed to the next step but one can proceed to the next state without addressing the warnings. But these warnings if not addressed can lead to significant logic trimming and unwanted logic optimization. In the end the implementation will fail. These errors and warnings are

1. **Error: Signal** $t1$ **in unit siso is connected to following multiple drivers**: The net $t1$ is driven by multiple inputs in the logic block *siso*. This error occurs as the net $t1$ is output of two logic blocks.
2. **Net** $< t2 >$ **does not have a driver.**: This type of warning is generated when the net $t2$ is defined but not assigned in the code. These type of warnings must be taken care of as the tool can trim or discard the logic block which has $t2$ as input.
3. **FF/Latch** $< m2/q >$ **has a constant value of 0 in block** $< siso >$. **This FF/Latch will be trimmed during the optimization process.**: This type of warning is very critical. If a logic block faces constant input either logic 1 or 0 is optimized automatically by the tool. This is because the block can be simply replaced by either a connection to Vdd or Gnd. The implementation can give incorrect results if this type of optimization occurs.
4. **Line 26: Size mismatch in connection of port** $< q >$. **Formal port size is 1-bit while actual signal size is 4-bit.**: This warning arises due to the mismatch of the size of the wires. Size of all the wires must be defined correctly otherwise the tool will generate warnings.
5. **Assignment to** q **ignored, since the identifier is never used**: The tool will never assign to a wire or net which is not used in the design.
6. **All outputs of instance** $< m1 >$ **of block** $< DFF >$ **are unconnected in block** $< siso >$. **Underlying logic will be removed.**: If the output of a logic block is not used in a design, then the underlying logic responsible for assigning that output net are removed.
7. $< m3 >$ **is already declared.**: This warning arises when same name for module instantiation is used.
8. **Mix of blocking and non-blocking assignments to variable** $< out >$ **is not a recommended coding practice.**: This warning is self-explaining and directs user not to mix up blocking (=) and non-blocking (<=) assignment operators.
9. **Procedural assignment to a non-register out is not permitted, left-hand side should be reg/integer/time/genvar**: This warning arises as the reg and $wire$ operators are mixed up. In Verilog language practice, these two operators should be correctly used.

Trouble shooting of complex digital system is very critical. All the warnings and errors must be addressed so that the implementation will give correct results. The

overall design should be broken into several smaller sub-modules so that block wise trouble shooting can be done.

Q3. **What is Timing-driven placement?**
A3. The objective of all the placement algorithms is to reduce the delay associated with a net by reducing its length. Timing-driven placement (TDP) algorithms focus on timing the critical path. Generally one cell can be connected to multiple cells. If the critical path from one cell to another cell is targeted, it may be possible that any other path originated from that cell can become critical. Thus TDP should be carefully applied.

Q4. **What is SI driven routing?**
A4. Signal Integrity (SI) driven routing algorithms aim to reduce the interference between two wires and thus try to reduce the noise. The timing driven routing algorithms on the other hand are based on reducing the critical path.

Q5. **How power measured in FPGA implementations?**
A5. In order to measure power, a design must be completely routed without any warnings and the design meets all the timing constraints. In case of XILINX ISE 14.7 tool, power measured using XPower analyzer tool. The files required for power measurement are Design file (.ncd), settings file (XML format), constraints file (.ucf) and simulation activity file (Value Change Dump (VCD) or Switching Activity Interchange format (SAIF)). Simulation activity file can be created by inserting the following section in the test bench file.

```
initial begin
$dumpfile ("test_design.vcd"); // Change filename as
appropriate.
$dumpvars(1, test_tb.uut); //Test Bench name...
end
```

Q6. **Mention the major blocks that an FPGA IC can contain?**
A6. The advanced FPGA devices contain many advanced blocks along with CLB, IOB and Programmable Interconnect and these are

1. Static RAM (SRAM) blocks for BRAMS.
2. DSP Blocks
3. PLL for clock generation.
4. Clock Managers
5. General Purpose Input Output (GPIO) banks
6. High-speed serializer/deserializer (SERDES) blocks
7. Hard Processor Cores
8. Peripherals (UART, CAN, I2C, SPI, USB, etc.).

Q7. **Mention the XILINX architectural evolution from FPGA to SoCs?**
A7. Over the past few years XILINX has offered several advanced FPGA devices. Its journey from a simple FPGA device to multi-core SoCs is

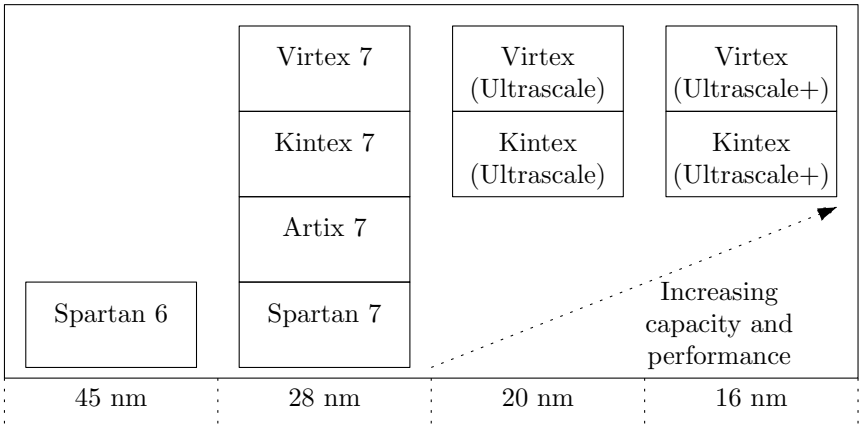


Fig. 14.20 Different FPGA offering by XILINX

1. FPGA—Only programmable fabric
2. System on Chip (SoC)—FPGA programmable fabric with a single hard core processor (e.g. ZYNQ SoC).
3. MPSoCs—FPGA programmable fabric with a multiple hard core processors.
4. RFSoCs—MPSoCs with RF capability
5. ACAPs—Adaptive Compute Acceleration Platforms.

Q8. **Mention different FPGA Vendors who offer high performance FPGA devices?**
A8. Different Vendors for FPGA devices are

1. XILINX FPGAs
2. Altera FPGAs (acquired by Intel)
3. Atmel (acquired by Microchip Technology)
4. Efinix, Lattice Semiconductor
5. Microsemi (acquired by Microchip Technology).

Q9. **Mention different FPGA offerings by XILINX?**
A9. Figure 14.20 shows the various FPGA offerings by XILINX and it also shows how performance is improved over the years in terms of speed and area.

14.5 Conclusion

In this chapter, we have discussed implementation of digital systems on FPGA platform and for ASIC. A brief theory about the internal structure of the FPGA, programming of the FPGA and FPGA implementation flow. The SPARTAN 3E FPGA

device is considered here for demonstrating basic operation of the FPGA device. The implementation flow is explained with the help of XILINX tool.

In this chapter, we have also discussed the ASIC implementation steps. The ASIC implementation steps are explained with the help of CADENCE EDA tool for standard cell-based system design. There are two major steps in ASIC implementation, viz., Synthesis and Placement and Routing process. The placement and routing flow is explained here step by step using the INNOVUS tool. At the end, some errors and warnings are explained for hassle free trouble shooting.

Chapter 15
Low-Power Digital System Design

15.1 Introduction

Design for low-power consumption is as important as design for less area and high speed. Low-power digital system design is not new to researchers as there are many research works reported in literature on techniques to achieve low-power consumption. A separate chapter is included here to familiarize the readers with the basic concepts of the power consumption reduction techniques.

In an integrated IC, there may be two types of power consumption, viz., peak power consumption and time average power consumption. Peak power consumption above a certain limit can instantly cause harm to an IC. The time average power consumption is very critical as it is directly proportional to the size of an IC and it also decides the life of the power source.

The power consumption of a digital system can be reduced at various levels. Hierarchy of these levels is given in Fig. 15.1. There are opportunities at each level to reduce power consumption. In the present scenario of the VLSI technology, there are more scopes of power reduction in the higher levels than in the lower levels. Initially, different sources of power in a digital circuit will be discussed and then various techniques for power reduction are briefly discussed.

15.2 Different Types of Power Consumption

In the section, different sources of power consumption will be discussed. The overall average power consumption has four components

$$P_{avg} = P_{switching} + P_{short-circuit} + P_{leakage} + P_{static} \tag{15.1}$$

A brief discussion on all the components of power consumption is given below.

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_15

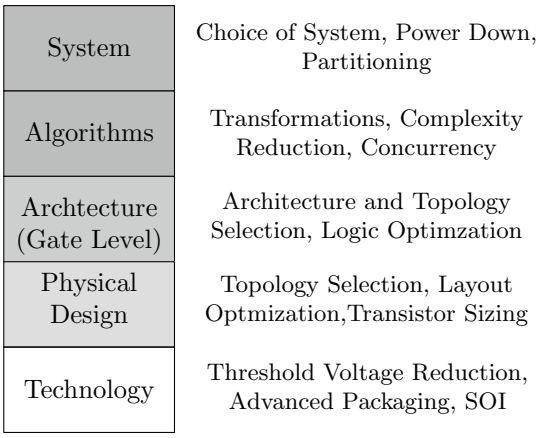


Fig. 15.1 Power consumption reduction techniques at various design abstraction levels

15.2.1 Switching Power

The switching power in a circuit is due to the switching activities at different nodes of an IC. The term switching activity means transition of signal value from 0 to 1 or 1 to 0. During these switching activities, circuit draws energy from the power supply to charge output load capacitance. The expression for the switching power can be expressed as

$$P_{switching} = \alpha.C_L.V.V_{dd}.f_{clk} \tag{15.2}$$

Here, f_{clk} is the clock frequency at which the sequential circuits operate, V_{dd} is the power supply voltage and V is the voltage swing. In most cases, V is equal to V_{dd}. C_L is the load capacitance and this capacitance can be expressed as

$$C_L = C_{gate} + C_{dif} + C_{int} \tag{15.3}$$

where C_{gate} is the gate capacitance, C_{dif} is the diffusion capacitance and C_{int} is the interconnect capacitance. α is the node switching activity factor and indicates the average number of times a node makes switching per sample clock.

The switching activity in a circuit can have two components which are

1. Static Component: This component doesn't take timing behaviour into account and only depends on logic block structure and input signal behaviour. This switching activity on a node of a logic block depends on the type of logic function, type of logic style, signal statistics and inter-signal correlations. The probable switching activity can be estimated at the output of logic gate as

$$p_{0\to1} = p_0.p_1 = p_0.(1 - p_0) \tag{15.4}$$

Table 15.1 Probability of output transition for various logic gates

Logic Gates	$p_{0\to1}$
AND	$(1-p_a p_b)p_a p_b$
OR	$(1-p_a)(1-p_b)(1-(1-p_a)(1-p_b))$
XOR	$(1-(p_a+p_b-2p_a p_b))(p_a+p_b-2p_a p_b)$

Here, $p_{0\to1}$ indicates the probability of transition from 0 to 1, p_0 is the probability that the output is at logic 0 state and p_1 indicates the probability that the output is at logic 1 state. The transition probability for N input gate can be expressed as

$$p_{0\to1} = \frac{N_0}{2^N}.\frac{N_1}{2^N} = \frac{N_0.(2^N - N_0)}{2^{2N}} \tag{15.5}$$

Here, N_0 is the number of zero entries in the truth table, N_1 is the number of ones in the truth table and 2^N represented the number of possible sets for N inputs. Here, inputs are considered as independent and equi-probable. The value of $p_{0\to1}$ for two-input NOR gate is 3/16 and that of a two-input XOR gate is 1/4.
In the above discussion, the inputs are considered as equi-probable means in a two-input gate both the inputs a and b have 50% probability of getting logic 1. But in the actual case, input can be logic 1 at any probabilities. In the case of a two-input NOR gate, the probability that the output node will be at logic 1 is

$$p_1 = (1-p_a)(1-p_b) \tag{15.6}$$

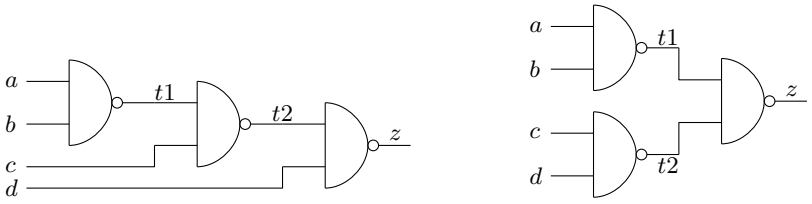
Here, p_a is the probability of the input a and p_b is the probability of the input b. The $p_{0\to1}$ can be estimated as

$$p_{0\to1} = (1-(1-p_a)(1-p_b))(1-p_a)(1-p_b) \tag{15.7}$$

The expression of $p_{0\to1}$ for other different types of logic gates are shown in Table 15.1.
Till now, input signals are considered as independent. But in complex digital systems signals are interrelated. Let us consider in a two-input gate input b depends on input a. In such cases, concept of conditional probability must be taken into account. Thus, the $p_{0\to1}$ for the output signal is expressed as

$$p_{0\to1} = p(a=1|b=1).p(b=1) \tag{15.8}$$

A logic expression can be implemented using either static CMOS configuration or dynamic logic configuration. The logic style has a great effect in determining switching effect. For example, the value $p_{0\to1}$ in the dynamic logic style is 3/4 compared to that using the static CMOS configuration.

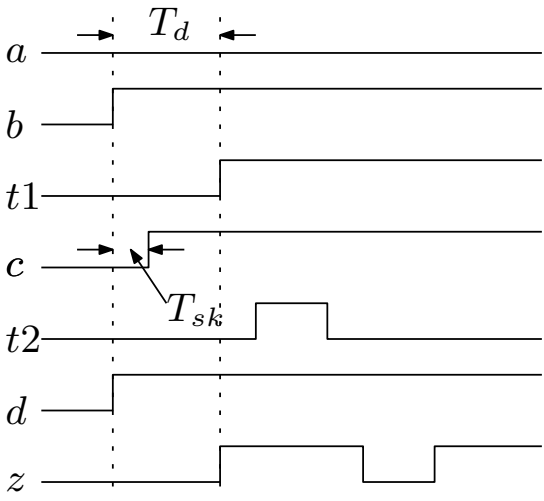


(a) Chain Structure. (b) Balanced Structure.

Fig. 15.2 Two topology to compute z

2. Dynamic Transitions: A node in circuit topology can have multiple transitions before it settles to a stable state. These extra transitions are due to circuit topology, logic depth, signal skew and signal patterns. These extra transitions are sometimes called glitches. The example of glitch generation can be explained by chain typology shown in Fig. 15.2 to compute $z = \overline{a.b.c.d}$.
The chain topology can generate glitches in the circuit if signal skew is present. The generation of glitch is shown in Fig. 15.3 where signal c has skew. The stable value of z is logic 1 but before reaching the stable state one extra transition is observed. This generation of glitch can be avoided by using balanced tree topology which is shown in Fig. 15.2b.

Fig. 15.3 Generation of glitch in the chain structures



15.2.2 Short Circuit Power

The short circuit power consumption is due to the creation of short circuit path in the static CMOS-based digital circuits. This component of power consumption can be expresses as

$$P_{short} = V_{dd}.I_{sc} \tag{15.9}$$

Here, I_{sc} is the short circuit current in the circuit.

15.2.3 Leakage Power

The leakage power consumption is due to the existence of leakage current ($I_{leakage}$) in the circuit. The leakage current can be due to reverse biasing of the P–N junctions or from the sub-threshold effects. This component of power consumption can be expressed as

$$P_{leakage} = V_{dd}.I_{leakage} \tag{15.10}$$

This component mainly depends on the technology parameters.

15.2.4 Static Power

The fourth component of power is static power consumption and it is expressed as

$$P_{static} = V_{dd}.I_{static} \tag{15.11}$$

Here, I_{static} is the static current that flows continuously in circuit between the power rails. Ideally, the static current should be zero in CMOS circuits. However, there is a small amount of static power consumption due to the reverse-bias leakage between diffused regions and the substrate.

In the above section, all the components of power consumption are discussed. Switching power consumption is the major issue in digital circuits. According to Eq. (15.1), the switching power can be reduced by operating the circuits at lower speed or by reducing the operating voltage or by reducing the switching activity. Switching activity can be reduced either by algorithmic modifications or by architectural modifications. In further sections of this chapter, we will discuss these techniques to reduce the switching power consumption by all means. But only the gate level or architecture level techniques will be discussed in this chapter.

15.3 Architecture-Driven Voltage Scaling

Selection of suitable architectures can allow designers to reduce the voltage. The reduction in voltage means the circuit will operate at a slower speed. Thus the chosen architecture should allow the designers to reduce voltage at affordable throughput. Three types of architectures are used to implement a system which are Serial Architecture, Parallel Architecture and Pipeline Architecture. All these architectures are investigated below in terms of power consumption.

15.3.1 Serial Architecture

A simple serial architecture to compute mean of eight 18-bit data samples is shown in Fig. 15.4. Here the execution unit is a simple 18-bit adder (RSH3 is a wired shift block). Register is placed before and after the combinational path. The desired throughput is one mean sample per unit clock cycle with period $8T_{sr}$. Thus the registers should be connected to a high speed clock with period T_{sr}. In eight clock cycles, first mean is computed. The overall capacitance switched in this circuit per cycle is

$$C_{sr} = C_{regin} + C_{comb} + C_{regout} \tag{15.12}$$

Here, C_{regin} is the capacitance involved in the input register, C_{regout} is the capacitance of the output register and the C_{comb} is capacitance of the combinational path. The serial architectures are hardware efficient but may not be suitable for lower power consumption as a high frequency clock is needed to be applied for desired throughput.

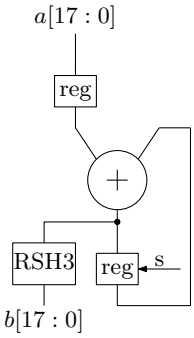


Fig. 15.4 Serial architecture for mean computation

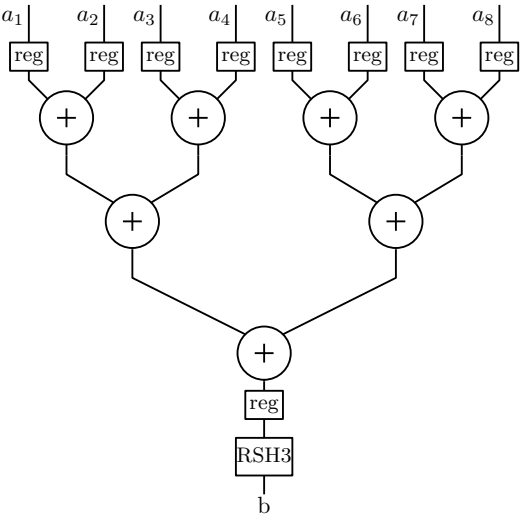


Fig. 15.5 Parallel architecture for mean computation

15.3.2 Parallel Architecture

In a parallel architecture, several execution units are used to compute a function. Here, a parallel architecture is shown in Fig. 15.5 for the same example of mean computation. In this parallel architecture, seven execution units or adders are used. The advantage of the parallel architecture is that the adders can add the samples by accessing the samples in parallel.

The clock period need not to be at higher rate as in the case of serial architecture. Thus, the clock period is $T_{par} = N.T_{sr}$ and one mean sample is computed in each clock cycle. The reduction of clock frequency allows designers to lower the operating voltage as long as the critical path delay is less than the clock period. If the relation $T_{par} = N \times$ critical path delay is true, then the voltage can be approximately reduced as $V_{par} = V_{sr}/N$, where V_{sr} is the operating voltage of the serial architecture.

The total power in parallel mean architecture can be expressed as

$$P_{par} = C_{par}.V_{par}.f_{par}^2 = (8.C_{regin} + 7.C_{comb} + C_{regout}).V_{par}.f_{par}^2 \qquad (15.13)$$

In the parallel architecture, overhead capacitance must also be included in computation of C_{par}. This overhead capacitance is due to routing paths, input/output MUXes or capacitance due to control circuits. Thus, the parallel architectures can be power efficient as long as the overall capacitance is not dominating the power equation.

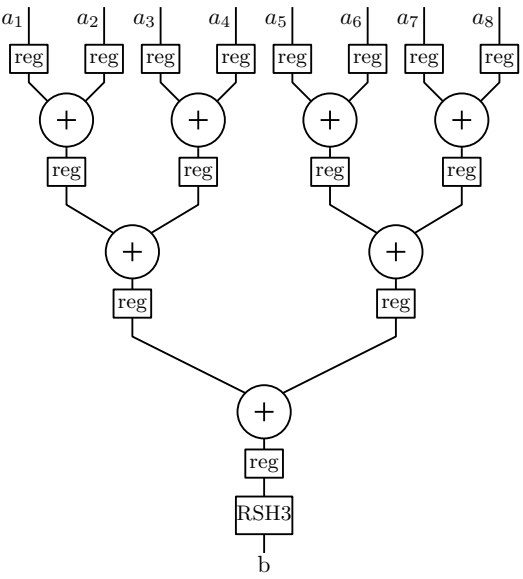


Fig. 15.6 Pipeline architecture for mean computation

15.3.3 Pipeline Architecture

The major drawback of the parallel architectures is its high critical path delay. This delay is reduced by inserting pipeline registers in the parallel architectures and the resulting architectures are called pipeline architectures. The pipeline architecture for mean computation is shown in Fig. 15.6. Here, the clock period is same as $T_{pipe} = T_{par}$ for fixed throughput.

In a pipeline architecture, the critical path is reduced by a factor of p for p pipeline stages. Thus, the circuit can operate at slower speed and the voltage (V_{pipe}) can be reduced by a factor of p. The power for the pipeline mean architecture can be expressed as

$$P_{pipe} = C_{pipe}.V_{pipe}.f_{pipe} = (8.C_{regin} + 6.C_{regpipe} + C_{regout}).V_{pipe}^2.f_{pipe} \quad (15.14)$$

15.4 Algorithmic Optimization

Choice of efficient algorithm is very important decision in designing a low-power digital system. An efficient algorithm can reduce maximum power by the reduction in complexity, algorithmic transformations, by improving concurrency or by giving flexibility to choose power-efficient data representation.

15.4.1 Minimizing the Hardware Complexity

Solving a linear equation is very important in signal processing applications like compressed sensing-based solutions. A simple linear equation is $y = \phi x$ where $y \in \mathscr{R}^{n\times 1}$ is the output vector, $\phi \in \mathscr{R}^{n\times n}$ is the system matrix and $x \in \mathscr{R}^{n\times 1}$ is the input. This equation can be solved in many ways. The solution of this equation is $x = \phi^{-1}y$. Most common ways to solve this equation are by Gaussian Elimination (GE) technique [61], Modified Cholesky Factorization (MCF) [56] and QR Decomposition (QRD) [62].

GE is the most common technique to solve a linear equation. In this method, a new matrix is formed as $\begin{bmatrix}\phi & y\end{bmatrix}$. Then this matrix is converted into an upper triangular matrix as $\begin{bmatrix}U & Y\end{bmatrix}$. Here, U is an upper triangular matrix. The solution is computed as $\hat{x} = U^{-1}y$ where $\hat{x}$ is the estimation of x.

MCF is a modified version of Cholesky factorization. MCF does not need square root operation as compared to Cholesky factorization. In this method, ϕ is factorized as $\phi = LDL^T$ where L is the lower triangular matrix with unit diagonal and D is a diagonal matrix. The matrix inverse is computed as $\phi^{-1} = (L^{-1})^T D^{-1} L^{-1}$. Then ϕ^{-1} is multiplied with y to compute $\hat{x}$.

QRD is another technique to solve this equation. The ϕ matrix can be written as $\phi = QR$ where R is an upper triangular matrix and Q is an orthonormal matrix. The matrix inverse is computed as $\phi^{-1} = R^{-1}Q$. This is then multiplied with y to get the final output. QR decomposition can be performed by three different techniques which are Gram–Schimdt algorithm, Givens Rotation algorithm and Householder algorithm. Here, a modified Gram–Schimdt algorithm is used.

A comparison of computational complexities for the above-mentioned solution techniques is shown in Table 15.2. Computational complexity is a measure of how many arithmetic operations are needed for evaluation of the function. The computational complexity is very important to measure the performance of an algorithm. Higher complexity can result in higher area or low speed. Low computational complexity is desired to achieve high SNR. Algorithms with low computational complexity enable power reduction either by reducing area or by reducing word length.

GE has less multiplication complexity than the other methods but it has high division complexity. Division operation is most critical to perform as discussed in Chap. 9. Thus, GE is computationally inexpensive but parallelism is difficult to

Table 15.2 Comparison of computational complexity of different solution techniques for linear equation

Operations	GE	MCF	QRD
Multiplication	$n^2(n-1)/2$	$(10n^3 + 6n^2 - 10n)/6$	$(5n^3 + 3n^2 + n)/3$
Addition/subtraction	$n^2(n+1)/2$	$(n^3 + 3n^2 - 4n)/6$	$(10n^3 - 5n^2 - 7n)/6$
Division	$n(n+1)/2$	n	$n(n-1)/2$
Reciprocal	0	0	n

adopt. MCF has less computational complexity than QRD and is also suitable for systolic-type architectures.

15.4.2 Selection of Data Representation Techniques

In communication between two chips or between a master and a slave in an embedded system, transition activity on the bits can be reduced by choice of data representation. The concept is to increase the correlation between two samples. Various techniques to achieve this are discussed below.

One-Hot Coding

One-hot coding technique is very powerful in reducing switching activity. This can be used in digital systems to reduce dynamic power. One-hot coding technique, a redundant coding scheme, is sometimes used in finite state machines to define states. In the inter-chip communication, this coding scheme can be used. The one-hot coding scheme is a one-to-one mapping between the n-bit data words which are to be sent and the m-bit data words which are transmitted. Here, m wires are connected between the transmitter chip and the receiver chip. The parameter m and n are related as $m = 2^n$. The one-hot coding scheme is shown in Table 15.3.

Here, '1' is placed in the ith place where $0 \leq i \leq (2^n - 1)$ and '0' is placed in the $(m - 1)$ places. In one-hot coding scheme, one $1 \rightarrow 0$ and one $0 \rightarrow 1$ transition occur between two data words. Thus, this scheme reduces a huge amount of switching activity. But this technique uses 2^n number of wires which increases the wiring overhead. One-hot coding sometimes becomes impractical for the exponential increases in the wiring overhead. For example, 75% switching activity is reduced for $n = 8$ but $m = 256$ number of wires are required.

Gray Coding

Gray coding is a powerful coding strategy which reduces the switching activity [68]. In Gray coding, only 1-bit transition is occurred between two words. Table

Table 15.3 One-hot coding scheme

2-bit data words	4-bit coded words
00	0001
01	0010
10	0100
11	1000

3.7 in Chap. 3 shows the binary representation and Gray code representation for decimal numbers 0–15. Gray codes are often used for assigning states in designing FSM-based systems. Gray code also can be used for fetching address and data from memory elements.

Bus Inversion Coding

Bus Inversion Coding (BIC) [66] is another coding technique to reduce dynamic power. In BIC n-bit data word which is to be sent is encoded in m-bit (where $m = n + l$) data which is transmitted. Here, l denotes the extra bits which are sent. BIC is based on sending either x or $\bar{x}$ depending upon which will results in less switching activity.

Now receiver has to be informed that which data is transmitted and thus an extra polarity bit (p) is also sent. Thus, the total number of bits will be $m = n + 1$. The transition on the polarity bit is also taken into consideration while sending x or $\bar{x}$. For a 32-bit bus, 33-bits are transmitted for one polarity bit. It is also possible to divide the 32-bit bus into smaller groups of width n and add polarity bits to each group.

BIC works better when the value of n is small but smaller value of n increases the value of extra polarity bits (l). Thus, smaller value of n increases wiring overhead. For example, switching activity is reduced by 11.3% if 32-bit data is sent with a single polarity bit but switching activity is reduced by 18.3% if 32-bit data is sent by dividing it into four groups with $l = 4$.

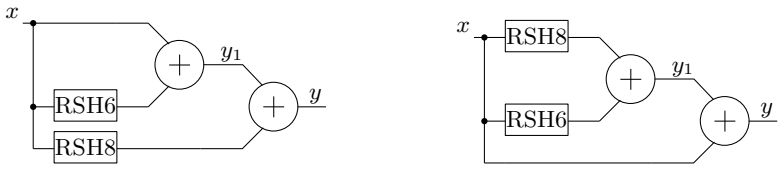
15.5 Architectural Optimization

15.5.1 Choice of Data Representation Techniques

Choice of data representation can also reduce switching activity. Two most used data representation techniques are Two's complement data representation and signed magnitude data representation. Two's complement data representation is most preferred in implementation of signal processing algorithms as addition and subtraction operations can be easily performed.

Two's complement data representation can produce significant switching activity due to its technique of sign extension. The transition on the MSB bits occurs when data transitions from positive to negative. In transition from –1 to 0, all the bits switched from 1 to 0. Situation becomes worse when the signal frequently switches around 0 and entire bandwidth is not used.

Signed magnitude representation is another technique to represent data. Here, only one bit (mostly MSB) is used to represent the polarity of data. Now if the data change from –1 to 0, then only one bit will change. Even if the entire bandwidth is



(a) Basic implementation. (b) After ordering operations.

Fig. 15.7 Example of operation re-ordering for low transition activity

not used, transition due to sign change occurs at one bit. Thus, signed magnitude representation is a good alternative to reduce power.

15.5.2 Ordering of Input Signals

The switching activity can be reduced by optimizing the order of operations in a design. This can be illustrated by taking the example of constant multiplication. Details of constant multiplication is shown in Chap. 3. The output (y) evaluated as $y = x(2^0 + 2^{-6} + 2^{-8})$. The basic implementation of this multiplication is shown in Fig. 15.7a. Here, RSH blocks are wired shift blocks as discussed in Chap. 3. The input signal has large variance which covers the entire bit-width. But the shifted signals are scaled and thus have lower dynamic range. Thus, the signals y_1 and y have very similar transition activity due to their dependence on x. An alternative structure is shown in Fig. 15.7b. In this structure, the signal $y1$ will have lower signal transition rate than the signal y. This is because $y1$ is now function of scaled signals not of x.

15.5.3 Reducing Glitch Activity

A node in a design can have multiple transitions in a single clock cycle before settling to the correct logic level. These extra transitions are called glitches. These glitches increase the power consumption. To minimize these glitches, it is important to balance all signal paths and reduce the logic depth. An example of glitch reduction by balanced structure is shown in Fig. 15.2b.

Table 15.4 Average number of gate transitions per addition [19]

Adder type	16-bit	32-bit	64-bit
Carry look ahead	90	182	366
Carry look ahead	100	202	405
Carry skip	108	220	437
Carry select	161	344	711
Conditional sum	218	543	1323

15.5.4 Choice of Topology

Choice of topology of the basic arithmetic blocks like adder, multiplier, divider or square root is very important to design a low-power digital system. The topologies can be selected based on trade-off among the design metrics like area, speed and power consumption. Table 15.4 shows a comparison of different adder topologies. Though the ripple carry adder has the low average gate transitions per addition, the look-ahead adder performs better among all these topologies in terms of speed and power consumption.

15.5.5 Logic Level Power Down

Power consumption can be reduced by deactivating or disabling execution units which are not in operation at a certain period of time. This shutting down of the execution units can save lots of power. This method includes control enabling memory elements or disabling the clock for an execution unit. If a memory element is not participating in an executing step then it can be disabled and can be again enabled when needed. Clock can be disabled by using clock gating techniques which are discussed in the FAQ section.

Consider an example of a 4-bit comparator architecture shown in Chap. 3. An alternative architecture is shown in Fig. 15.8. The comparator structure is divided into two parts, one for lower 2-bits and one for upper 2-bits. If the upper 2-bits are not equal, then the comparator for lower 2-bits can be deactivated. Thus, transition activity for comparator 2 is reduced. This is a simple way to show power-down mechanism which also serves the purpose of pipeline register insertion.

15.5.6 Synchronous Versus Asynchronous

In synchronous circuits, every operation is performed with respect to clock. Switching power depends on switching activity within a clock cycle. The power can be reduced

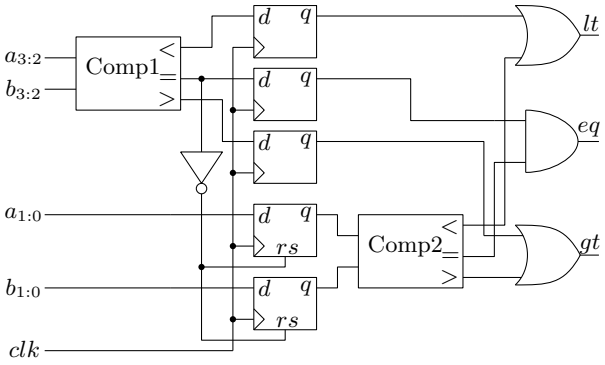


Fig. 15.8 A scheme of power down for 4-bit comparator

by deactivating the execution units which are not operating. This deactivation requires implementation of extra logic. On the other hand, asynchronous circuits do not follow the clock. Deactivation of execution units can be done easily. But, asynchronous designs on the other hand is not suitable for very complex designs due to timing synchronization problems.

15.5.7 Loop Unrolling

Efficient implementation of the digital systems which have feedback path are very challenging. This is because pipelining and parallelism can not be applied if there is a feedback path in a system. An example of such system is IIR filters. There are techniques by which pipelining and parallelism can be applied to these IIR filters (Chap. 16). But loop unrolling is an efficient technique to introduce pipelining as well as to reduce power consumption of these recursive systems.

Let's consider the case of a simple first-order IIR filter. The timing difference equation of this IIR filter is shown below:

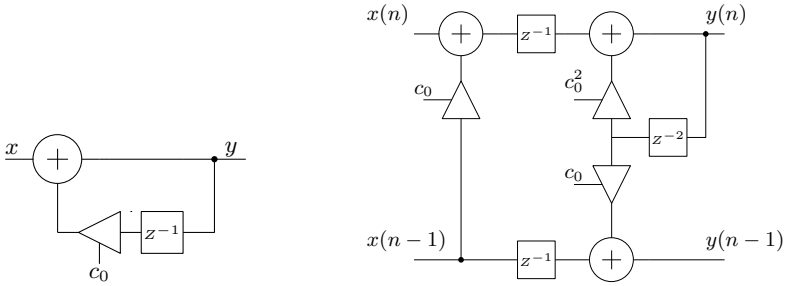
$$y(n) = x(n) + c_0.y(n-1) \tag{15.15}$$

The architecture according to equation 15.15 is shown in Fig. 15.9a. Similarly, the following timing difference equation can also be written by replacing n by $(n-1)$:

$$y(n-1) = x(n-1) + c_0.y(n-2) \tag{15.16}$$

This equation can be substituted in equation 15.15 and the resulting timing difference is written as

$$y(n) = x(n) + c_0 x(n-1) + c_0^2.y(n-2) \tag{15.17}$$



(a) Direct implementation. (b) Loop unrolled structure.

Fig. 15.9 Example of loop unrolling for first-order IIR filter

The basic concept of loop unrolling is to compute multiple samples per clock against multiple inputs and to allow application of slower clock for fixed throughput. Here, two output samples are computed concurrently. Thus, loop unrolling reduces power consumption. The IIR filter structure after loop unrolling is shown in Fig. 15.9b.

Loop unrolling reduces the power consumption by allowing slower clock. But this technique increases hardware resources as three multipliers are used in place of one multiplier. The use of extra resources increases the overall capacitance. Thus, a trade-off should be maintained while applying the loop unrolling.

15.5.8 Operation Reduction

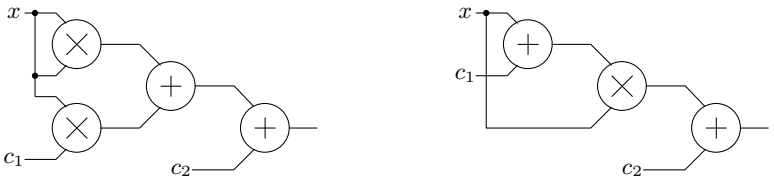
Operation reduction is an obvious way to reduce switching capacitance or to reduce power consumption. If any algorithm is to be implemented, then the designer must investigate to reduce the number of operations. A simple example of operation reduction can be shown by implementing the following polynomial:

$$f(x) = x^2 + c_1.x + c_2 \tag{15.18}$$

Direct implementation of this polynomial is shown in Fig. 15.10a. This implementation uses two multipliers and two adders. This polynomial can also be written as

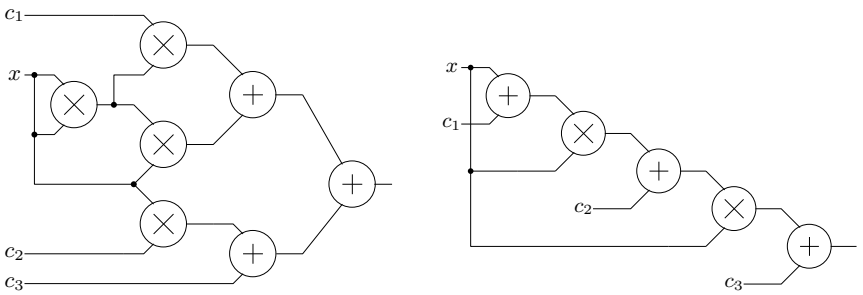
$$f(x) = x(x + c_1) + c_2 \tag{15.19}$$

Implementation of this form of polynomial is shown in Fig. 15.10b. Here, one multiplication operation is reduced. This reduction automatically reduces the total switching events. There are many examples of operation reduction in implementations. For example, if the integer part in a division operation is limited to fewer bits then some stages in a non-restoring divider can be removed. In constant multiplication process, use of dedicated constant multipliers instead of complete multipliers is another



(a) Direct implementation. (b) After operation reduction.

Fig. 15.10 Example of operation reduction for a second-order polynomial



(a) Direct implementation. (b) After operation reduction.

Fig. 15.11 Example of operation reduction for a third-order polynomial

example of operation reduction. Dedicated reciprocal or square root reciprocal units based on iterative algorithms like Newton–Raphson or Gold-Smith reduce lots of operation.

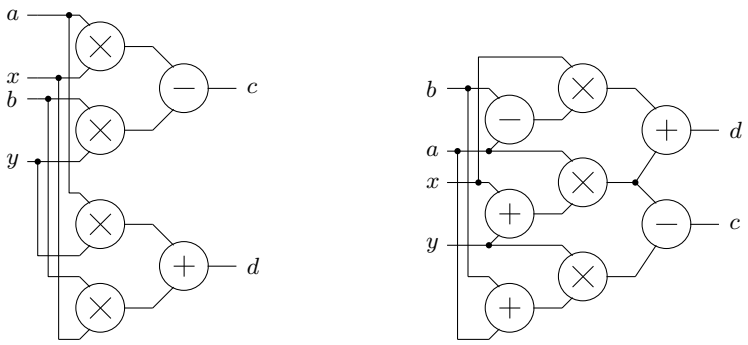
Sometimes, operation reduction method may increase critical path. This can be understood by implementing the polynomial

$$f(x) = x^3 + c_1.x^2 + c_2.x + c_3 \tag{15.20}$$

The straightforward implementation of this equation is shown in Fig. 15.11a. It has two multipliers and two adders on its critical path. Reduced equation is

$$f(x) = x.(x^2 + c_1.x^2 + c_2) + c_3 \tag{15.21}$$

This reduction of operations is shown in Fig. 15.11b. The critical path has an extra multiplier. This will affect the execution speed of the design. Thus, designers have to carefully apply the operation reduction methodology.



(a) Direct implementation. (b) After operation substitution.

Fig. 15.12 Example of operation substitution for a complex multiplier

15.5.9 Substitution of Operation

Operation substitution is another method to reduce the power consumption in a design. This technique replaces the logic blocks which consume high energy with the blocks which consume lower energy. This statement is explained with the help multiplication of two complex numbers $(a + ib)$ and $(x + iy)$. The direct implementation of this multiplication is shown in Fig. 15.12a which is according to the following equation:

$$(a + ib) \times (x + iy) = c + id = (ax - by) + i(bx + ay) \tag{15.22}$$

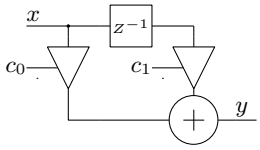
The above equation can be rewritten as

$$c + id = \{a(x + y) - (b + a)y\} + i\{(b - a)x + (x + y)a\} \tag{15.23}$$

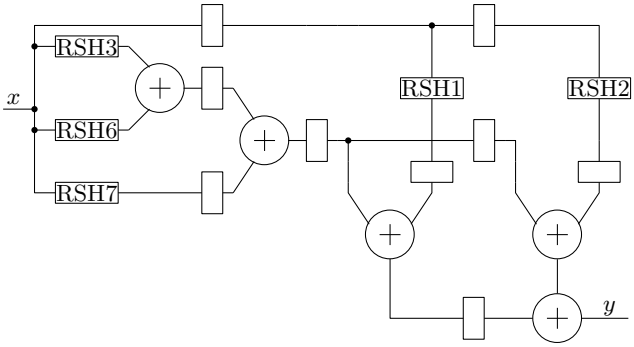
In this equation, complex multiplier is implemented using substituting multiplication by addition and subtraction operation. This implementation is shown in Fig. 15.12b.

The implementation in Fig. 15.12b has less overall capacitance than the earlier implementation. But the second implementation has a longer critical path. This means the second path will achieve less maximum frequency. Thus, care should be taken in case of operation substitution.

In the above example, multiplier is substituted by adders to reduce overall capacitance. The constant multiplication operation can also be replaced with shift and add operations. The theory of constant multiplication is already discussed in Chap. 3. Now, if a signal is multiplied by two or more constants then two power reduction techniques can be applied. The first one is operation substitution and the second one is resource reduction. This can be explained with the help of design of a simple FIR filter whose timing difference equation is



(a) Direct implementation.



(b) Implementation after operation substitution.

Fig. 15.13 Example of operation substitution for a simple FIR filter

$$y(n) = c_0 x(n) + c_1 x(n-1) \tag{15.24}$$

Direct implementation of this equation is shown in Fig. 15.13a. Let the word length is 16-bits and 8-bits are reserved for fractional part. The values of the constants are as $c_0 = 00000000\_10100110$ and $c_1 = 00000000\_01100110$. Here, both the constants have ones at positions 3, 6 and 7. Thus, resource sharing can be applied. The modified structure is shown in Fig. 15.13b.

15.5.10 Re-timing

Energy consumption depends on the number of resources used per clock cycle. If resources increase then the energy consumption will also increase. The usage of resources per clock cycle can be reduced with the help of re-timing of the design. Re-timing is nothing but placing the registers at suitable places without affecting the circuit functionality.

The concept of re-timing can be explained with the help of a simple second-order IIR filter. A Biquad structure of IIR filter in the Direct Form II is shown in Fig. 15.14.

Two designs are obtained by introducing the re-timing and these circuits are shown in Fig. 15.14. Both the structures improve the frequency performance of the IIR filter. The structure shown in Fig. 15.15a has five multipliers and two adders in the critical path. The second structure shown in Fig. 15.15b has three multipliers and two adders

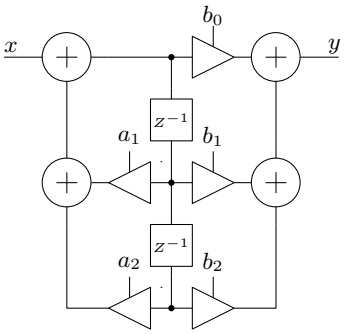
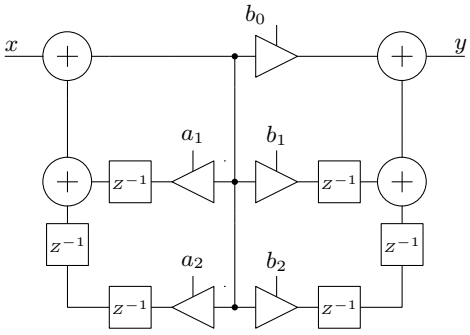
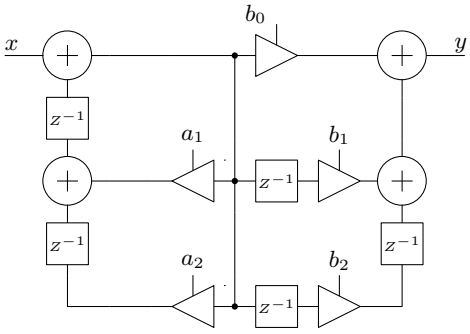


Fig. 15.14 Simple biquad structure in the direct form



(a) First structure after Re-timing.



(b) Second structure after Re-timing.

Fig. 15.15 Example of different structures of IIR Biquad after Re-timing

in the critical path. Thus, there are a lesser number of resources switched in the second structure and thus less power is consumed in the second structure.

15.5.11 Wordlength Reduction

In an implementation, effect of word length reduction on power consumption is significant. If the word length is reduced, then hardware resources are reduced and so the overall capacitance. Thus, reduction in word length reduces the power. But how far the word length can be reduced? Word length reduction directly affects the accuracy of an implementation. Thus, a trade-off should be maintained so that reduction in word length does not hamper accuracy significantly.

The choice of algorithms or architectures can afford a significant reduction in word length. The algorithms which have low computational complexity show better accuracy with lesser word length. For example, modified Cholesky decomposition results better SNR than QR decomposition in computation of pseudoinverse matrix. Some architectures have better immunity towards the word length change. For example, parallel and cascade structures of FIR filters are capable of showing better results at lower word length.

15.5.12 Resource Sharing

Resource sharing is a very promising technique to reduce power consumption in complex digital systems. In implementations of any complex algorithm, there is always scope of sharing hardware resources. Resource sharing is important as the area constraint permits parallelism up to a certain limit. In some cases where the steps of an algorithm must be executed sequentially, resource sharing technique must be adopted to reduce the overall capacitance.

The concept of resource sharing can be understood using the example of implementation of vector arithmetic operations. Signal processing algorithms involve many vector arithmetic operations such as scalar–vector multiplication and vector–vector multiplication. An algorithm may involve sequential execution of both these two types of operations. In such cases, execution unit can be designed such a way that it can perform both the operations. Separate execution units may not be feasible because of area constraint. The common execution unit to perform both the operations is shown in Fig. 15.16 for vector length 4.

In the architecture shown in Fig. 15.16, there is a control signal s which selects between two operations. The inner product operation or vector–vector multiplication is performed when $s = 0$. The scalar–vector multiplication and accumulation are performed when $s = 1$. Here, one extra adder and four MUXes are used but four sets of multipliers, adders and registers are saved. Thus, overall capacitance can be reduced using resource sharing. The switching activity will increase due to control circuitry.

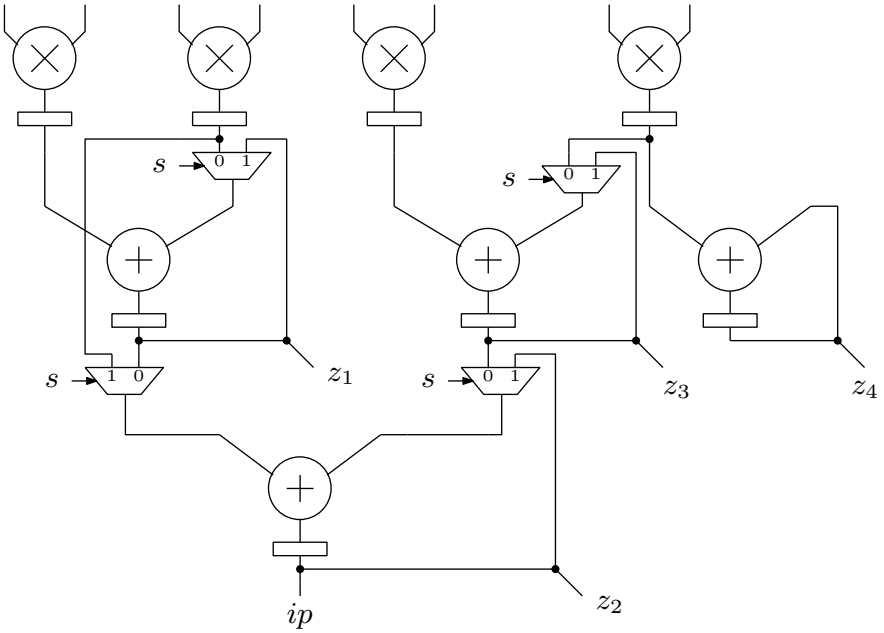


Fig. 15.16 A shared architecture to perform both scalar–vector multiplication and vector–vector multiplication

15.6 Frequently Asked Questions

Q1. **What is clock gating and its advantage in reducing power of a circuit?**
A1. Clock gating is a very power full technique in sequential circuits to reduce dynamic power. Switching activity of the clock signal accounts for major power consumption in a complex digital system. But there are scopes to reduce power due to clock distribution in such circuits. Clock gating is a technique of gating the clock or disabling the clock signal for the modules which are idle or not functioning for a certain period of time. Thus, majority of dynamic power saved by clock gating.

Q2. **Mention different clock gating techniques?**
A2. Different clock gating techniques are

1. Gate-based clock gating
2. Latch-based clock gating
3. Flip-flop-based clock gating
4. Synthesis-based clock gating
5. Data-driven based clock gating
6. Auto-gated clock gating [67]
7. Look-ahead-based clock gating [80].

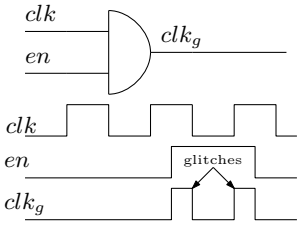


Fig. 15.17 AND gate-based clock gating

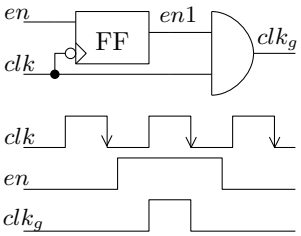


Fig. 15.18 Glitch-free gate-based clock gating

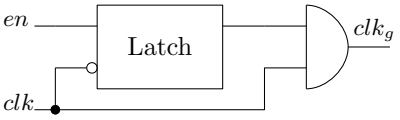


Fig. 15.19 Latch-based clock gating scheme

Q3. **Explain Gate-based clock gating technique and its problems?**
A3. The gate-based clock gating technique is shown in Fig. 15.17. Gate-based clock gating can be achieved by either an AND gate or an OR gate. The problem with gate-based clock gating is that glitches can be generated if the enable signal is not properly aligned.

Q4. **How the glitch generation problem with gate-based clock gating scheme can be avoided?**
A4. The glitch generation in the gate-based clock gating scheme can be avoided by generating the enable (en) signal using a negative edge triggered flip-flop as shown in Fig. 15.18.

Q5. **Explain the Latch-based clock gating scheme.**
A5. The latch-based clock gating scheme is shown in Fig. 15.19. Here, there is no need of generating the enable signal through a negative edge-triggered flip-flop to avoid glitch generation. Similarly, flip-flop-based clock gating technique can be designed by replacing the latch with a flip-flop.

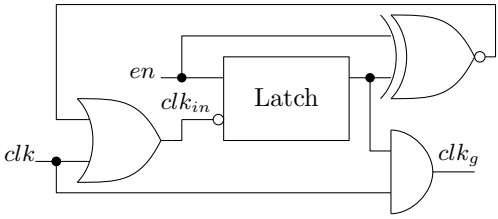


Fig. 15.20 Clock-gated clock gating cell

Q6. **Clock gating schemes like latch-based lock gating schemes reduce dynamic power but the clock gating circuit itself consumes power. Give an example of one such scheme which reduces the power of clock gating circuit?**
A6. One such scheme is shown in Fig. 15.20. When the *en* signal is low then the output of the latch is also low. This makes output of the EX-NOR gate high. The output of the EX-NOR gate disables the clock to the latch of the clock gating circuit. This circuit further reduces the dynamic power but glitch can be generated at the output of the OR gate. These glitches do not affect the clock-gated output but can cause silicon failure.

15.7 Conclusion

In this chapter, various techniques to reduce dynamic or switching power consumption in a digital system are discussed. Switching power can be reduced by reducing the operating voltage or by reducing operating frequency or by reducing switching activity. All the techniques discussed in this chapter are either gate level or algorithmic level.

Algorithmic-level power reduction is most important in implementing an algorithm-based system. In the algorithmic level, designers must try to reduce the computational complexity and to increase the concurrency. Reduction in computational complexity will reduce overall load capacitance and gives designers a choice to use less number of bits. More concurrency in an algorithm means the architecture can adopt parallel architectures.

The second most important step is to reduce power at architecture level. Here, gate-level reduction techniques are discussed. Parallelism and insertion of pipeline registers are very important steps to reduce power. Operation reduction, operation substitution, choice of data representation, re-timing and unrolling are some of the techniques to reduce power. The techniques discussed above must be applied by considering the other design metrics which are speed and area. Thus, a trade-off should be maintained.

Chapter 16
Digital System Design Examples

In this chapter, we will discuss FPGA implementation of some of the digital systems to give an idea to the readers how complex signal, video or image processing algorithms can be implemented. These architectures are implemented using Verilog HDL and mainly structural programming procedure is followed. All the systems are implemented using XILINX 14.7 EDA tool.

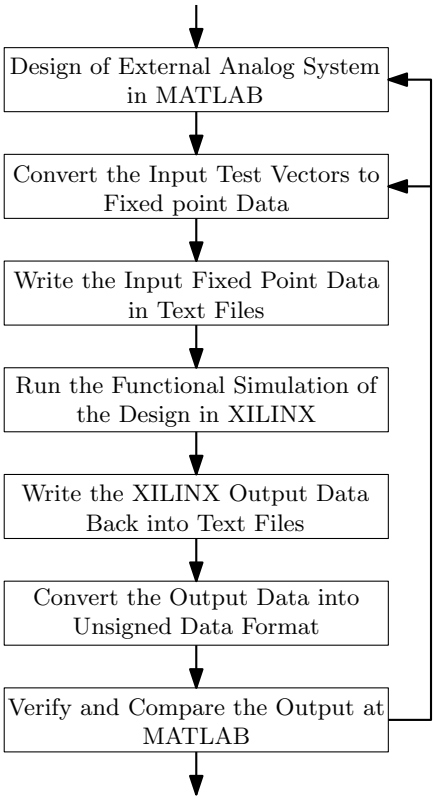
Designs must be functionally verified before loading the bit-stream to the FPGA. Smaller designs can be verified easily by giving inputs directly through the FPGA inputs. The verification of complex designs is critical. The digital system may take input data from an analog system. In such cases, verification of the complete system needs an interface of XILINX and MATLAB environment which models the analog system. This verification environment is shown in Fig. 16.1.

The MATLAB environment simulates the analog design in integer format. But the XILINX accepts the binary data either in fixed or floating format. Thus, the data which are to be imported from MATLAB to XILINX should face a data conversion step. This data conversion step takes care of the data width variation. The result of the FPGA implementation also can be verified in the MATLAB by taking it back to MATLAB. The steps which are followed to verify a design are shown in Fig. 16.2. All the architectures are designed with fixed point data format and two's complement data representation is used. Simple Verilog instructions are used to implement the arithmetic blocks instead of special fast hardware to simply the implementation. Performance of all the designs is estimated in terms of timing complexity, hardware complexity and power consumption.

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_16

Fig. 16.1 Digital system verification environment

Fig. 16.2 Design verification flow chart



16.1 FPGA Implementation FIR Filters

In signal processing applications, mainly two types of digital filters are used. The first type is Finite Impulse Response (FIR) filters and Infinite Impulse Response (IIR) filters are the other types of digital filters. If impulse response of any filter is of finite duration, then the filter is of FIR type. So, in the case of FIR filters, impulse response settles to zero in finite duration. There is no feedback in FIR filters unlike IIR filters and thus FIR filters are always stable. FIR filters can produce linear phase and this property is used in many applications. Compared to IIR filters, implementation of FIR filters is straightforward.

In many applications, FPGA is nowadays preferred as hardware platform and FIR filters are very often required to implement on FPGA. Many literatures reported different kinds of FPGA implementations for FIR filters over the past few decades. FIR implementations are so popular and frequently used that FPGA manufacturers are providing inbuilt advanced features for rapid prototyping of FIR filters. Advanced digital signal processing (DSP) blocks that are capable of doing several functions are used as basic building blocks for FIR filters. DSP blocks made implementation of FIR filters an easy job.

Different aspects of FPGA implementation of FIR filters are discussed in this work. Different structures for implementation of a simple low-pass FIR filter are discussed here. The usage of advanced DSP blocks to implement the FIR filters is also shown here. Lastly, design performances of all kinds of architectures are compared in terms of parameters like resource utilization, latency and maximum frequency.

16.1.1 FIR Low-Pass Filter

Design of a low-pass filter (LPF) is taken as an example to describe the implementation of FIR filter. The frequency response of an ideal LPF is

$$H(e^{jw}) = \begin{cases} 1 & \text{for } -\frac{\pi}{2} \leq w \leq \frac{\pi}{2} \\ 0 & \text{for } -\frac{\pi}{2} \leq |w| \leq \pi \end{cases} \tag{16.1}$$

Here, $w = \frac{\pi}{2}$ denotes the cutoff frequency in radian and w is the normalized frequency. Many techniques can be used to realize this low-pass FIR filter as illustrated in [10]. Here, Hamming window-based design is followed.

The transfer function $H(z) = y(z)/x(z)$ of a N tap FIR filter can be written as

$$H(z) = \sum_{n=0}^{N-1} c_n z^{-n} \tag{16.2}$$

Here, z^{-n} denotes the delay by n samples and c_n is the co-efficients of the transfer function. Here, n can take both even or odd values. In time domain, the low-pass FIR filter for $N = 13$ can be expressed as

$$y = x.(c_0 + c_1.z^{-1} + c_2.z^{-2} + c_3.z^{-3} + c_4.z^{-4} + ... + c_{12}.z^{-12}) \tag{16.3}$$

Here, N co-efficients and $(N - 1)$ delay elements are present. Figure 16.3 shows the frequency plot of the low-pass FIR filter.

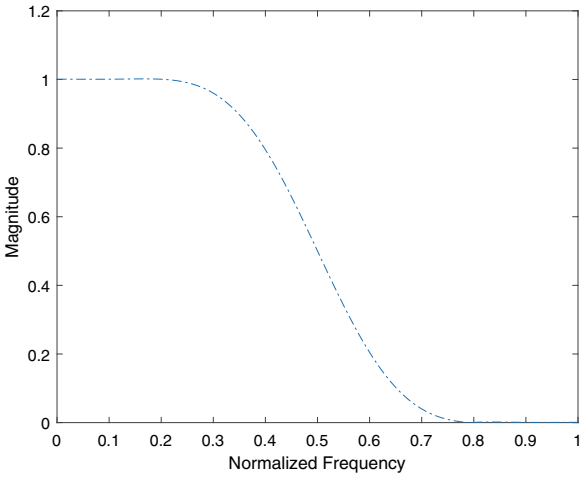


Fig. 16.3 Frequency spectrum of the low-pass filter

16.1.2 Advanced DSP Blocks

Advanced FPGAs are providing high-speed arithmetic blocks to perform different operations using hardware sharing technique. These blocks are known as DSP blocks and different FPGAs are different architectures for these DSP blocks. DSP48 [83] blocks inside the ARTIX7 FPGA device are example these kind of blocks. Operations like addition, subtraction, multiply, multiply-accumulate or shifting, etc., can be done by a same DSP block based on a multi-bit select signal. Figure 16.4 shows a simple block diagram of a DSP block. The DSP block shown in the figure is capable of performing specific operations which are required for FIR filters. A DSP block includes a pre-adder and a multiplier and an ALU. Various functions can be realized using ALU but it performs only subtraction and addition operations.

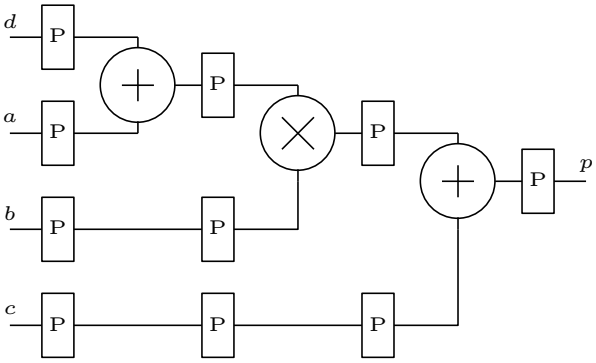


Fig. 16.4 A simplified architecture of a DSP block

DSP block has a *sel* control signal based on which DSP block performs various functions. The functions, $p = c \pm a \times b$ and $p = c \pm (a + d) \times b$ can be evaluated by the DSP block shown in the figure. Pipeline registers can be inserted or removed and the number of pipeline registers is programmable. These DSP blocks are faster compared to LUT-based counterparts as they are inbuilt to the FPGA devices.

16.1.3 Different Filter Structures

Different structures [10] to implement FIR filters are reported in literature and these are

1. Direct Form Structures
2. Linear Phase Structures
3. Polyphase Structures
4. Cascaded Structures
5. Lattice Structures

Here, an LPF is implemented using Direct Form, Linear Phase Form, Polyphase Form and Cascaded Form. In speech processing, lattice structures are extensively used but they are costly. Thus, it is avoided here but other structures are discussed in detail along with their FPGA implementation.

Direct Form Structures

In direct form structures, co-efficients of the transfer function are same as the multiplier co-efficients. In an N-tap FIR filter, N multipliers and $(N - 1)$ two-input adders are required. The direct form structure for the LPF is shown in Fig. 16.5 and it is known as the Direct Form 2 structure. Here, pipeline registers and discrete delay elements are shown separately. Pipeline registers are shown by 'P' . Here, the maximum operating frequency is limited by the delay in the multiplier and adder path. Combination of an adder and a multiplier can be realized using high-speed DSP blocks as shown in Fig. 16.5 by the dotted rectangular box.

The Direct Form 2 structure (Fig. 16.5) has a latency of 12 clock cycles. The same filter structure can be realized in another way as shown in Fig. 16.6. This is called

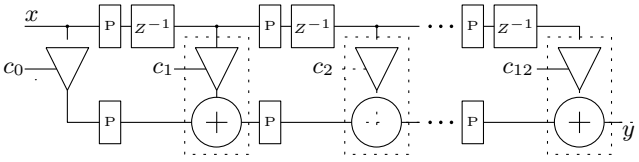


Fig. 16.5 Direct form architecture of the FIR low-pass filter

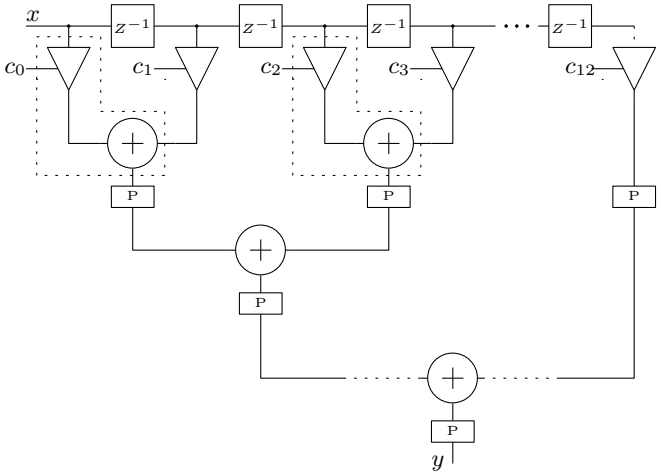


Fig. 16.6 Another implementation of the direct form architecture of the FIR low-pass filter

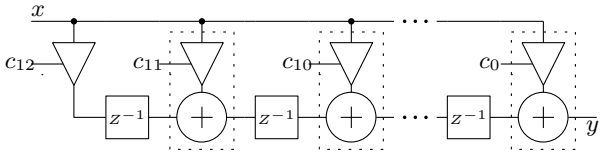


Fig. 16.7 Transpose of the direct form architecture of the FIR low-pass filter

Direct Form 1 structure and this structure has less latency. Output of the multipliers are fed to an adder tree and the adder tree has less latency. Same number of adders and multipliers are used in this architecture but has a latency of only three clock cycles. The DSP blocks (Fig. 16.4) also can be used here.

In Fig. 16.6, FIR filter is directly implemented following its equation but the architecture shown in Fig. 16.5 is a systolic array type architecture. An inverted architecture of the same FIR filter is possible and it is shown in Fig. 16.7. This inverted structure is known as Direct Form 3 and it has less latency compared to other two structures mentioned above. The major advantage of this inverted structure is that pipeline registers are not required.

Linear Phase Structures

FIR filters can be more efficiently implemented if the transfer function is symmetric or anti-symmetric. The symmetry or anti-symmetry property can be written as

$$c_n = \pm c_{(N-1-n)} \tag{16.4}$$

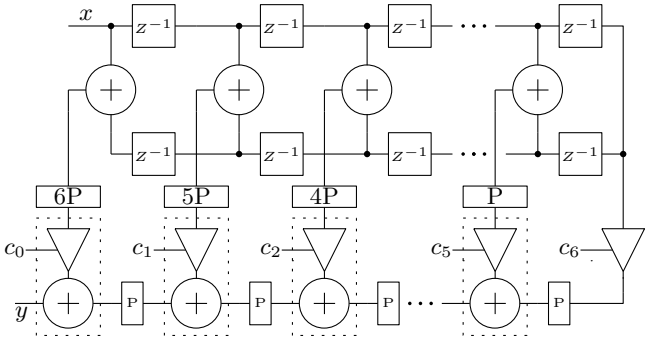


Fig. 16.8 Linear phase architecture of the FIR low-pass filter

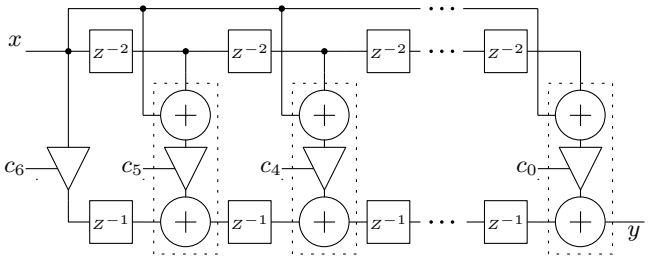


Fig. 16.9 Transposed linear phase architecture of the FIR low-pass filter

The above property can be used to reduce the number of multipliers by half of that in the direct form implementations. According to the above property, the transfer function can be written as

$$H(z) = \begin{cases} \sum_{n=0}^{\frac{N-2}{2}} [z^{-n} + z^{-(N-1-n)}], & \text{When N is even.} \\ c_{(\frac{N-1}{2})} z^{-(N-1)/2} + \sum_{n=0}^{\frac{N-3}{2}} [z^{-n} + z^{-(N-1-n)}], & \text{When N is odd.} \end{cases} \tag{16.5}$$

The LPF implemented using the linear phase structure is shown in Fig. 16.8 and this is called Linear Phase 1 structure. The critical path of a multiplier and an adder is retained here and thus more pipeline registers are required to achieve high frequency.

Linear Phase 2 structure (Fig. 16.9) is developed by modifying the architecture shown in Fig. 16.8. Linear Phase 2 structure can be called transposed architecture of the previous linear phase architecture. Hence, less pipeline registers are required and the critical path is adder–multiplier–adder. Lesser maximum frequency is achieved for this structure compared to other linear structures. DSP blocks can be used to realize this structure to improve its timing performance. This new version of linear phase filter using DSP blocks is called Linear Phase 3 structure.

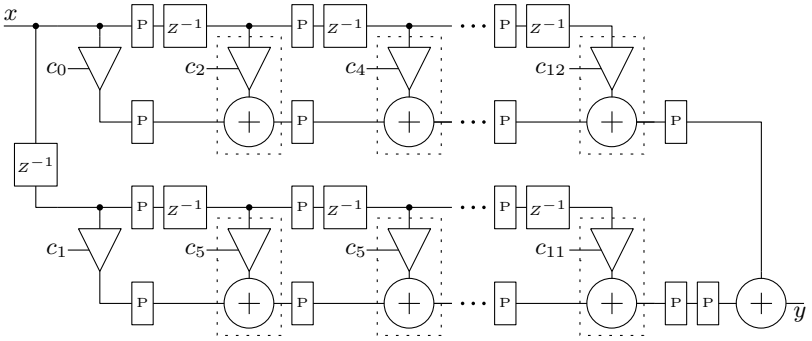


Fig. 16.10 Polyphase structure of the FIR low-pass filter

Polyphase Structures

The transfer function of an FIR filter can be written as the summation of two terms where the first term contains all the odd indexed co-efficients and the second term contains all the even indexed co-efficients. For example, the transfer function of the LPF can be expressed as

$$H(z) = (c_0 + c_2 z^{-2} + c_4 z^{-4} + ... + c_{12} z^{-12}) + (c_1 z^{-1} + c_3 z^{-3} + ... + c_{11} z^{-11}) \tag{16.6}$$

This equation can be re-written as

$$\begin{aligned} H(z) &= (c_0 + c_2 z^{-2} + c_4 z^{-4} + ... + c_{12} z^{-12}) + z^{-1}(c_1 + c_3 z^{-2} + ... + c_{11} z^{-10}) \\ &= P_0(z^2) + z^{-1} P_1(z^2) \end{aligned} \tag{16.7}$$

In the above equation, there are two branches but, in general, there can be more number of branches. Each branch can be realized using direct form. This type of filter structure is popularly known as polyphase structure and is shown in Fig. 16.10 (Polyphase 1). In multi-rate signal processing, these polyphase filters have many uses and also they have less latency than direct form realizations. Another type of polyphase structure (Polyphase 2) is shown in Fig. 16.11 which has shared delay elements.

Cascade Structures

Higher order FIR filters are realized efficiently using the cascaded form. In this form, lower order FIR sections are cascaded to realize a higher order transfer function. Each lower order FIR section may be realized using either a first-order or second-order transfer function. An FIR transfer function of higher order can be written as

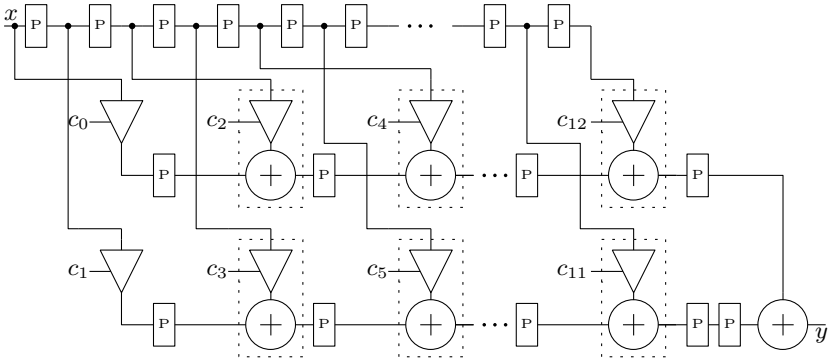


Fig. 16.11 Polyphase structure of the FIR low-pass filter by sharing delay elements

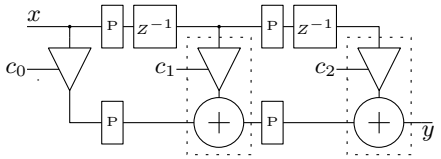


Fig. 16.12 Second order stage for a cascade form of FIR low-pass filter

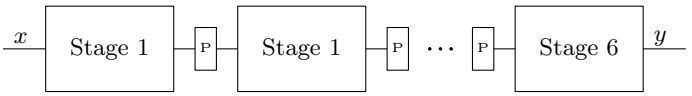


Fig. 16.13 Cascade form of FIR low-pass filter

$$H(z) = \begin{cases} (c_{10} + c_{11}z^{-1}) \prod_{n=2}^{\frac{N}{2}} (c_{n0} + c_{n1}z^{-1} + c_{n2}z^{-2}), & \text{When N is even.} \\ \prod_{n=1}^{\frac{N-1}{2}} (c_{n0} + c_{n1}z^{-1} + c_{n2}z^{-2}), & \text{When N is odd.} \end{cases} \tag{16.8}$$

Here, the higher order transfer function is factorized into second-order transfer functions. Hardware realization of the second-order transfer function is shown in Fig. 16.12. Overall, cascaded realization of the LPF is shown in Fig. 16.13. Here, total $(N + 5)$ multipliers and $(N - 1)$ adders are used. In the cascade implementation, the co-efficients are modified unlike in other implementations. Cascade structures are less sensitive towards the word length.

16.1.4 Performance Estimation

Implementation Issues

In hardware implementation of FIR filters, three major blocks are used, which are adders, multipliers and delay elements. Multiplier is the major block here which is used to do multiplication operation using known constants. Constant multipliers can replace complete multipliers in this application. Constant multipliers use add and shift method to do multiplication by a known constant. This customization of multipliers may be useful in ASIC implementation but in FPGA implementation, DSP blocks optimize themselves according to the operands.

Both serial or parallel architecture for FIR implementation is possible. All the possible parallel architectures are mentioned in this work. But serial implementations may be useful in some specific applications. Maximum achievable frequency of an architecture is an important parameter to measure performance. Pipeline registers may be inserted to achieve high frequency for high-speed applications, but if speed of operation is low, then these registers must be removed to save hardware.

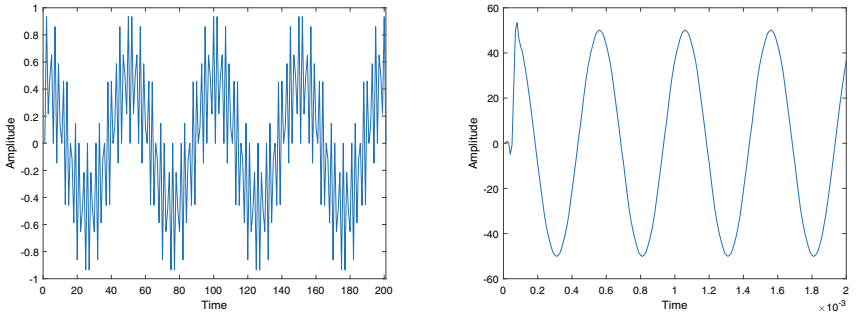
Performance of FIR filter also degraded due to quantization error which occurs because of insufficient data format, repetitive multiplication and scaling. Both fixed point data format and floating point data format can be used. Floating point format achieves high accuracy compared to fixed point format for same data width, but the design of floating point digital blocks is complex and expensive. Increasing the data width will increase accuracy but will also increase hardware. Thus, a trade-off is to be maintained between hardware and acceptable accuracy.

Design Performance

A 13-tap FIR LPF is implemented on NEXYS DDR2 artix7 FPGA development board (xc7a100t-3csg324) in this work. The LPF is verified by taking a multi-tone signal as input. This multi-tone signal is created by mixing two sinusoidal signals of frequencies 22 and 20 KHz. The sampling frequency is set at 100 KHz and thus the cut-off frequency of the LPF is 25 KHz. A tone of 2 KHz is the output of the LPF which is shown in Fig. 16.14b. The LPF output obtained from MATLAB and the FPGA implementation output is compared in Fig. 16.14. A 20-bit fixed point data width is used where 12-bits are reserved for the fractional part. Here, the metric Root Mean Squared Error (RMSE) measures the design performance. RMSE is computed as

$$RMSE = \frac{\left\|(\hat{y} - y)\right\|_2}{\|y\|_2} \tag{16.9}$$

Here, $\hat{y}$ is FPGA output and y is MATLAB-based filtered output. An RMSE of 0.0006728 is achieved using a data width of 20-bits.



(a) Input signal to the FIR filter

(b) Output of the low pass filter

Fig. 16.14 Input and output signal of the filter

Table 16.1 Performance comparison of different structures

Structures	CLK_{min} (ns)	Slice reg	Slice LUT	DSP48	Occupied slices	Power (W)
Direct form 1	5.5	432	327	11	155	0.048
Direct form 2	5.7	631	337	11	283	0.048
Direct form 3	5.5	259	273	11	83	0.044
Linear phase 1	5.5	533	347	6	125	0.044
Linear phase 2	7.5	390	302	6	159	0.048
Linear phase 3	6	352	168	7	70	0.064
Cascaded	5.5	290	279	16	105	0.036
Polyphase 1	6	542	309	11	261	0.051
Polyphase 2	5.5	522	334	11	212	0.053

Comparison of the Different Structures

A comparison of different FIR filter implementations is shown in Table 16.1. The parameters which are used to measure design performances are hardware utilization (consumption of slice registers, LUTs, DSP blocks and occupied slices), maximum achievable frequency and dynamic power. Dynamic power is computed at maximum frequency. Transposed direct form architecture is better than the other direct form structures as it does not need pipeline registers. If the co-efficients are symmetrical then linear phase structures can be used. Linear phase 2 architecture achieves less maximum frequency as its critical path is long. Linear phase 3 architecture imple-

ments same linear phase 2 structure but with DSP blocks. Linear phase 3 structure achieves higher maximum frequency, consumes less resources but consumes higher power. This is an advantage of using the inbuilt DSP blocks. Cascaded and polyphase structures are not frequently used but they have other advantages which is not in the scope of discussion.

16.1.5 Conclusion

FPGA implementation of different FIR filter structures and a comparison of the performances is presented here. Here, an LPF is designed using different structures to demonstrate the difference in implementation. FIR transfer functions are directly implemented by direct form structures and these are systolic architectures. Transposed structures don't need the pipeline registers and thus they are popular. Linear phase structures are only used when the transfer function is symmetric or anti-symmetric. Cascaded structures are useful to ignore the effect of word length but need more hardware. The DSP blocks can efficiently increase the performance of these filter structures.

16.1.6 Top Module for FIR Filter in Transposed Direct Form

```
module LPF_FIR_rv(start,clk,reset,t25);
input start,clk,reset;
output [19:0] t25;
wire [19:0] t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,t13,t14,
t15,t16,t17,t18,t19,t20,t21,t22,t23,t24,t25,t26;
parameter c1 = 'd2, c2 = 'd3693, c3 ='d1048561,
c4 = 'd1025100, c5= 'd78, c6='d122466, c7 = 'd204799;
parameter c13 = 'd2, c12 = 'd3693, c11 ='d1048561,
c10 = 'd1025100, c9= 'd78, c8='d122466;
wire [7:0] adb;
wire [19:0] x_o,x,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,
x13,x14,x15,x16,x17,x18,x19,x20,x21,x22,x23,x24,x25;
/////Reading the input data from a ROM.........
ym1 mem(clk,1'b1,adb,x_o);
count8 cnt(adb,8'b00000000,tc,enb,clk,reset,tc,'d202);
pg pg1(start,tc,enb,clk,reset);
reg18 rgw(x,clk,reset,x_o);
//////////////////////////
mult mu(x, c13, t1);
reg18 rg1(t2,clk,reset,t1);

MAC m1(x,c12,t2,clk,1'b0,reset,t3);
reg18 rg2(t4,clk,reset,t3);
```

```
MAC m2(x,c11,t4,clk,1'b0,reset,t5);
reg18 rg3(t6,clk,reset,t5);

MAC m3(x,c10,t6,clk,1'b0,reset,t7);
reg18 rg4(t8,clk,reset,t7);

MAC m4(x,c9,t8,clk,1'b0,reset,t9);
reg18 rg5(t10,clk,reset,t9);

MAC m5(x,c8,t10,clk,1'b0,reset,t11);
reg18 rg6(t12,clk,reset,t11);

MAC m6(x,c7,t12,clk,1'b0,reset,t13);
reg18 rg7(t14,clk,reset,t13);

MAC m7(x,c6,t14,clk,1'b0,reset,t15);
reg18 rg8(t16,clk,reset,t15);

MAC m8(x,c5,t16,clk,1'b0,reset,t17);
reg18 rg9(t18,clk,reset,t17);

MAC m9(x,c4,t18,clk,1'b0,reset,t19);
reg18 rg10(t20,clk,reset,t19);

MAC m10(x,c3,t20,clk,1'b0,reset,t21);
reg18 rg11(t22,clk,reset,t21);

MAC m11(x,c2,t22,clk,1'b0,reset,t23);
reg18 rg12(t24,clk,reset,t23);

MAC m12(x,c1,t24,clk,1'b0,reset,t25);
////Writing the output in a text file for Verification..
mem_write mn(t25,enb,clk);
Delay16 dly(clk,enb,en);
endmodule
```

16.2 FPGA Implementation of IIR Filters

In digital domain, majority of filters are of FIR type due to ease of implementation. IIR filters can achieve same performance with less co-efficients and delay elements compared to FIR filters. IIR filters are also suitable for achieving narrow bandwidth. FIR filters can easily work at high frequencies with the help of the DSP blocks [83] but a higher frequency use of IIR filters is limited. IIR filters are generally preferred at low frequencies as IIR filters are recursive. In recursive filters, output is reused at the input end and has a long critical path. Thus, inserting pipeline registers in IIR filters is a challenging task.

In [60], FPGA implementation of the FIR filters is discussed. In this section, different parallel FPGA implementation of the IIR filters is presented. The pipeline implementation of IIR filters is also investigated here. A simple LPF is taken as an

example and this filter is implemented with every type of IIR structure. Performance comparison of these structures is also carried out here.

16.2.1 IIR Low-Pass Filter

Design of a LPF is taken as an example to illustrate the implementation of IIR filters. The frequency response of the LPF is specified as

$$\alpha_p = 0.01\,dB \text{ for } 0 \leq w \leq \frac{\pi}{3} \tag{16.10}$$

$$\alpha_s = 60\,dB \text{ for } \frac{\pi}{3} \leq w \leq \frac{\pi}{2} \tag{16.11}$$

Here, w is the normalized frequency. The parameter α_p and the α_s denotes attenuation in the pass band and stop band, respectively. This low-pass IIR filter can be realized using many techniques [10]. Elliptical-based design is followed here.

The transfer function $H(z) = y(z)/x(z)$ of an IIR filter can be written as

$$H(z) = \frac{\sum_{n=0}^{M} b_n z^{-n}}{1 - \sum_{n=1}^{N} a_n z^{-n}} \tag{16.12}$$

Here, b_n denotes the co-efficients of the all-zero transfer function and a_n denotes the co-efficients of the all-pole transfer function. Here, $N = M = 6$ for the given LPF. The frequency plot of the low-pass IIR filter is shown in Fig. 16.15.

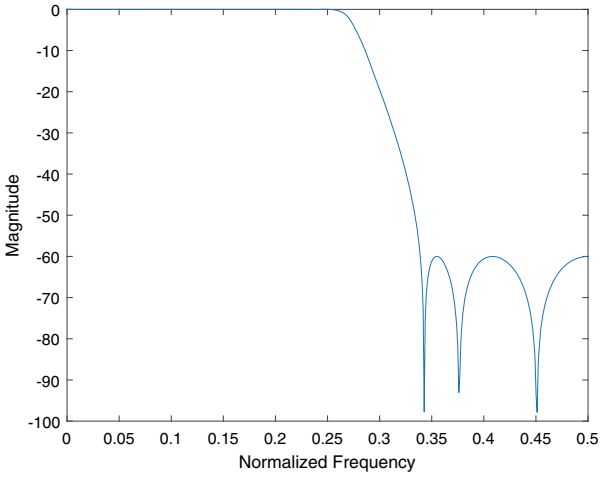


Fig. 16.15 Frequency spectrum of the low-pass filter

16.2.2 Different IIR Filter Structures

In literature, many structures [10, 27] to implement the IIR filters are reported and these are

1. Direct Form I Structures
2. Direct Form II Structures
3. Cascaded Structures
4. Parallel Structures
5. Lattice Structures

In this work, an LPF is implemented using Direct Form I, Direct Form II, Cascaded Form and Parallel Form. Lattice structures are not discussed here as they are extensively used in speech processing and are costly. Other structures are discussed in detail with their FPGA implementation.

Direct Form I Structures

The time difference equation of the IIR filter can be expressed as

$$y(n) = \sum_{k=1}^{N} a_n y(n-k) + \sum_{n=0}^{M} b_n x(n-k) \tag{16.13}$$

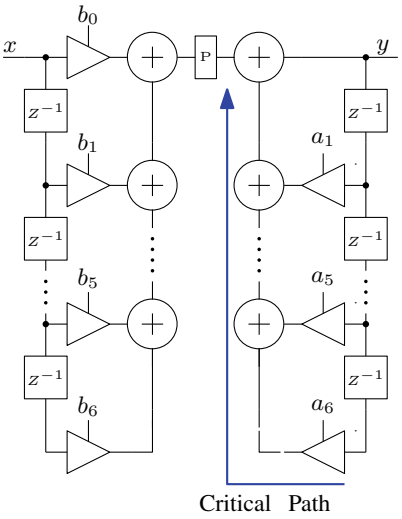
The Direct Form I structures directly implement the IIR filters. Figure 16.16a shows the basic Direct Form I structure. This structure has a long critical path in the recursive section (all-pole transfer function section). Hence, this structure is not suitable for implementation in many applications. Pipeline registers directly cannot be inserted like in the FIR filters. Incorrect insertion of pipeline registers may lead to the realization of different transfer functions. Figure 16.16b shows an alternate direct form structure which is known as transposed Direct Form I structure. Critical path is reduced in this structure.

Direct Form II Structures

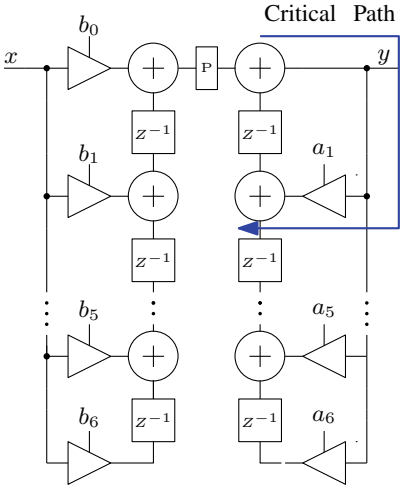
The Direct Form II structures implement the difference function based on the fact that the filter transfer function can be represented as

$$H(z) = \frac{y(z)}{w(z)} \cdot \frac{w(z)}{x(z)} \tag{16.14}$$

Figure 16.17a shows the Direct Form II realization of the IIR LPF. Compared to the Direct Form I structure, this structure has same critical path but has less number of delay elements. Transposed version of this structure is shown in Fig. 16.17b and

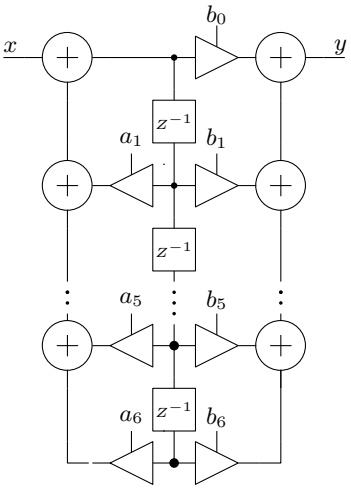


(a) Direct form I structure of IIR filters.

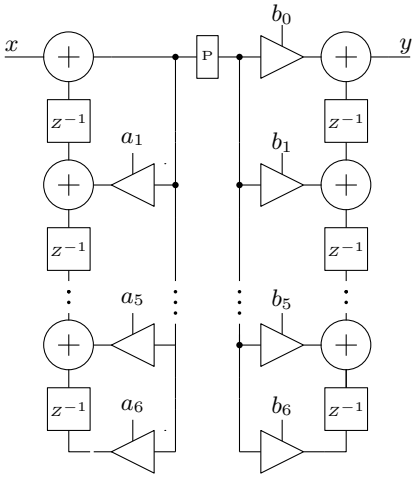


(b) Transposed structure of the direct form I.

Fig. 16.16 Direct form I structures for IIR filters



(a) Direct form II structure of IIR filters.



(b) Transposed structure of the direct form II.

Fig. 16.17 Direct form II structures for IIR filters

it has less critical path. Both Direct Forms I and II have similar kind of transposed structures. But Direct Form II structures are not suitable in many applications. In these kinds of structures, a high-gain all-pole network is followed by a all-zero network. This may increase the data width of the multipliers and adders.

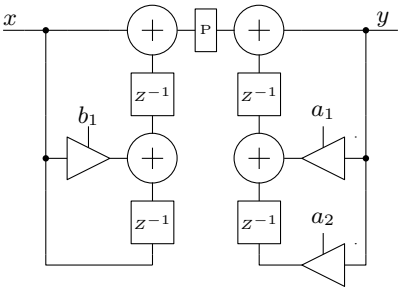


Fig. 16.18 Architecture for the Biquad block for cascaded IIR filter

Cascaded Form

IIR filters also can be implemented using cascaded form. In the cascaded form, IIR filter transfer function is broken into smaller transfer functions (IIR Type) and then these smaller transfer functions are cascaded. An IIR filter transfer function can be written as

$$H(z) = H_1(z).H_2(z)...H_k(z) \tag{16.15}$$

Here, $H(z)$ is written in terms of k number of smaller transfer functions. These smaller transfer functions can be of either first order or second order. The second-order IIR section is known popularly as the Biquad structure. The Biquad transfer function is

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}} \tag{16.16}$$

The Biquad structure for the above-mentioned LPF is shown in Fig. 16.18. These Biquad structures can be implemented using any of the direct form structures. The cascaded structure of the IIR LPF is shown in Fig. 16.19.

Parallel Forms

Similar to the cascaded structures, parallel filter structures also provide immunity towards the quantization effect due to the co-efficients. But parallel forms are more popular as they provide parallelism in the design also. Transfer function for IIR filter can be written as summation of first-order or second-order sections in this

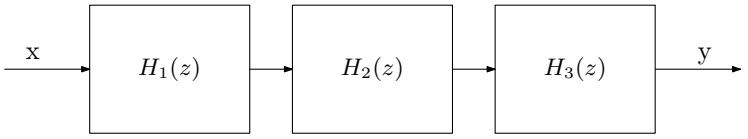


Fig. 16.19 Cascaded structure of the IIR LPF

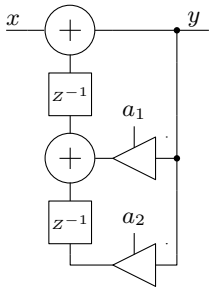


Fig. 16.20 Architecture for the Biquad block for Cascaded IIR filter

form. Partial fraction technique is used to break a transfer function as summation of smaller transfer functions. Biquad structures are generally preferred as individual sections. The IIR LPF transfer function considered here can be written as

$$H(z) = FIR + \sum_{i=1}^{L} \frac{d_{i1} + d_{i2}z^{-1}}{1 - a_{i1}z^{-1} - a_{i2}z^{-2}} \tag{16.17}$$

Here, FIR is a simple constant and the number of parallel sections is denoted by L. The above equation is known as non-delayed architecture [12]. If the order of numerator is greater than the order of denominator, then delayed input signal is some times provided to the all-pole section. This form is called delayed parallel structure [13]. In this case, FIR part can be an FIR-like transfer function. The second-order section for the parallel structure of the LPF considered here is shown in Fig. 16.20. The non-delayed and delayed parallel structures are shown in Figs. 16.21 and 16.22, respectively.

16.2.3 Pipeline Implementation of IIR Filters

Previously, different structures of IIR filters are discussed. The direct forms directly implement the IIR difference equation. In high-frequency applications, the direct form filter structures are not suitable as hey are having long critical path. The transposed architectures are suitable for comparatively higher frequencies but still they are having higher critical path. Direct insertion of pipeline registers is not possible as it will implement another transfer function. Thus, special algorithms are reported in literature to implement pipeline IIR filters.

First, we will try to achieve pipelining in a simple first-order filter. The transfer function of the first-order filter is

$$H(z) = \frac{1}{1 - az^{-1}} \tag{16.18}$$

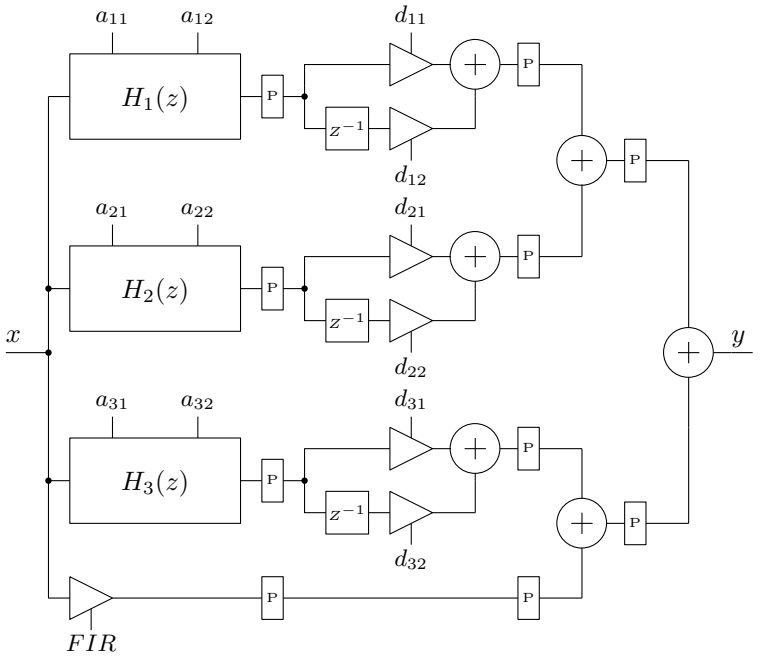


Fig. 16.21 Non-delayed parallel structure of IIR filter

This filter can be implemented either using direct form or using transposed form. Figure 16.23 shows these structures. In both cases, the maximum frequency is limited by the sum of delay provided by an adder and a multiplier. Insertion of pipeline registers can not reduce this delay.

The look-ahead pipelining techniques [54] are very helpful in inserting pipeline registers in IIR filters. The original transfer function has a single pole at $z = a$. P-stage pipeline implementation can be derived by adding $(P - 1)$ poles and zeros at identical locations. Transfer function of the first-order filter is now modified for pipeline implementation and it is

$$H(z) = \frac{\prod_{i=0}^{log_2^{P-1}} (1 + a^{2^i} z^{-2^i})}{1 - a^P z^{-P}} \tag{16.19}$$

The transfer function for the three-stage pipeline implementation of the first-order filter is shown below

$$H(z) = \frac{1 + az^{-1} + a^2 z^{-2}}{1 - a^3 z^{-3}} \tag{16.20}$$

Figure 16.24 shows the pipeline structure of the first-order IIR filter. Critical path is obviously reduced here by at least three times. This circuit is now suitable for

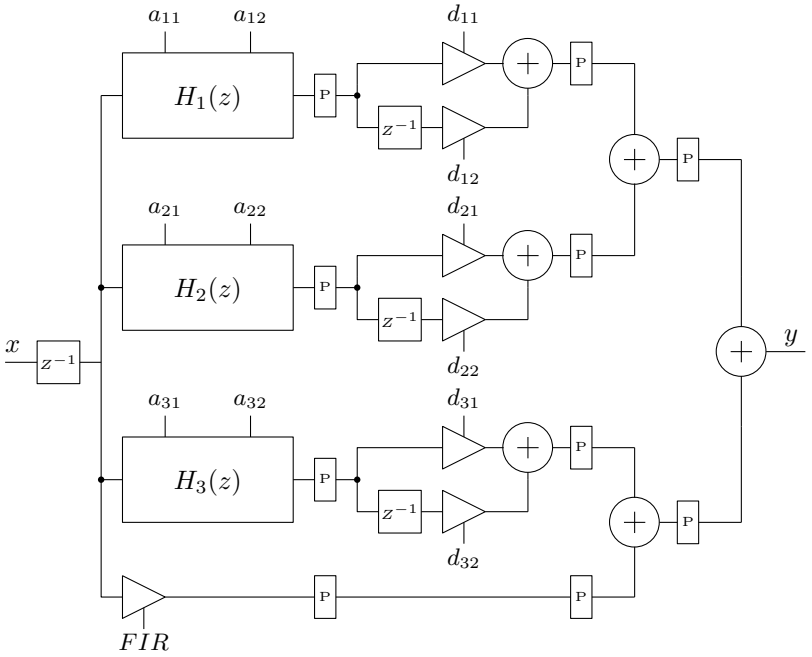
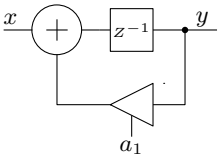
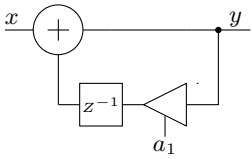


Fig. 16.22 Delayed parallel structure of IIR filter



(a) Structure of a typical first order IIR filter.



(b) Transposed structure of first order IIR filter.

Fig. 16.23 First-order IIR filter structures

high-frequency applications but the hardware complexity of the implementation is increased.

Two look-ahead techniques are popular to implement pipeline IIR filters and these are

1. Clustered Look-ahead Technique
2. Scattered Look-ahead Technique

In both these techniques, pipeline implementation of the first-order IIR filter is same. *P* stage pipeline implementation of the higher order filters using the clustered look-ahead technique is obtained by multiplying the numerator and denominator by

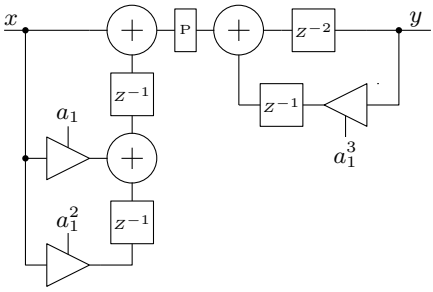


Fig. 16.24 Pipelined version of the first-order filter

$$\sum_{i=0}^{P-1} r_i z^{-i} \tag{16.21}$$

where r_i is evaluated as follows:

$$r_i = 0, for\ i = 0, 1, 2, ..., (N-1) \tag{16.22}$$

$$r_0 = 0 \tag{16.23}$$

$$r_i = \sum_{k=1}^{N} a_k r_{i-k},\ i > 0 \tag{16.24}$$

$(P-1)$ additional cancelling poles and zeros are inserted here. This method is having stability issues. This method can produce an unstable transfer function even if the original transfer function was stable. A higher number of pipeline stages are preferred for making stable implementation.

Scattered look-ahead technique is another technique to obtain pipeline implementation. Let us consider the denominator of a transfer function is represented as

$$D(z) = \prod_{i=1}^{P} (1 - p_i z^{-1}) \tag{16.25}$$

Then P-stage pipeline implementation in this technique is obtained by the following equation:

$$H(z) = \frac{N(z)}{D(z)} = \frac{N(z) \prod_{i=1}^{N} \prod_{k=1}^{P-1} (1 - p_i e^{j2\pi k/P} z^{-1})}{\prod_{i=1}^{N} \prod_{k=0}^{P-1} (1 - p_i e^{j2\pi k/P} z^{-1})} \tag{16.26}$$

In comparison to the clustered look-ahead technique, this technique generates a stable transfer function with less number of pipeline stages. Both the techniques implement pipeline IIR filters with increased hardware resources. This technique is more suitable to the second-order transfer functions instead of directly applying

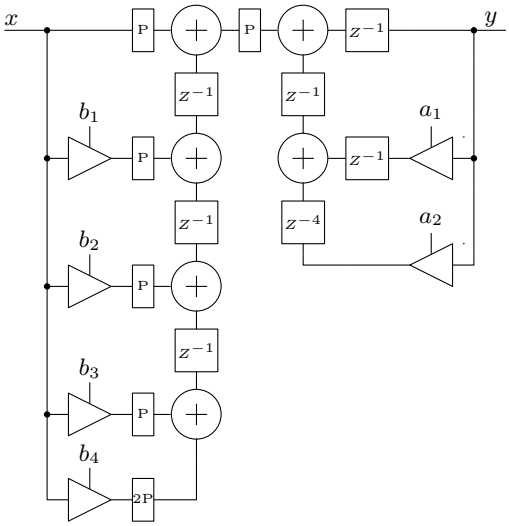


Fig. 16.25 Pipelined version of the Biquad for parallel IIR filter

to the main transfer function. The second-order transfer function $H(z) = 1/(1 - a_1 z^{-1} - a_2 z^{-2})$ is transformed into the following transfer function:

$$H(z) = \frac{1 + a_1 z^{-1} + (a_1^2 + a_2) z^{-2} - a_1 a_2 z^{-3} + a_2^2 z^{-4}}{1 - (a_1^3 + a_1 a_2) z^{-3} - a_2^3 z^{-6}} \tag{16.27}$$

Pipeline implementation of the IIR LPF is presented here. The parallel IIR LPF is chosen here for pipeline implementation. The scattered look-ahead technique is applied to the second-order Biquads. Figure 16.25 shows the modified Biquad structure and the resulting pipeline parallel IIR LPF is shown in Fig. 16.26. Co-efficients of the pipelined version of the IIR filter are star-marked.

16.2.4 Performance Estimation

Implementation Issues

IIR filter architectures can be serial or parallel like FIR filters. All the possible parallel architectures are mentioned here. Serial architectures also may be useful in some design applications. IIR filters are some advantages over FIR filters which are already discussed. But IIR filters have one dis-advantage also. It is not easy to insert pipeline registers in IIR filters. This is why techniques like look-ahead must be used. But these techniques increase hardware resources and also inexact pole-zero cancellation may occur due to finite word length. Another method to use IIR filters at high frequency is that design the IIR filters in a way such that pipline registers can be

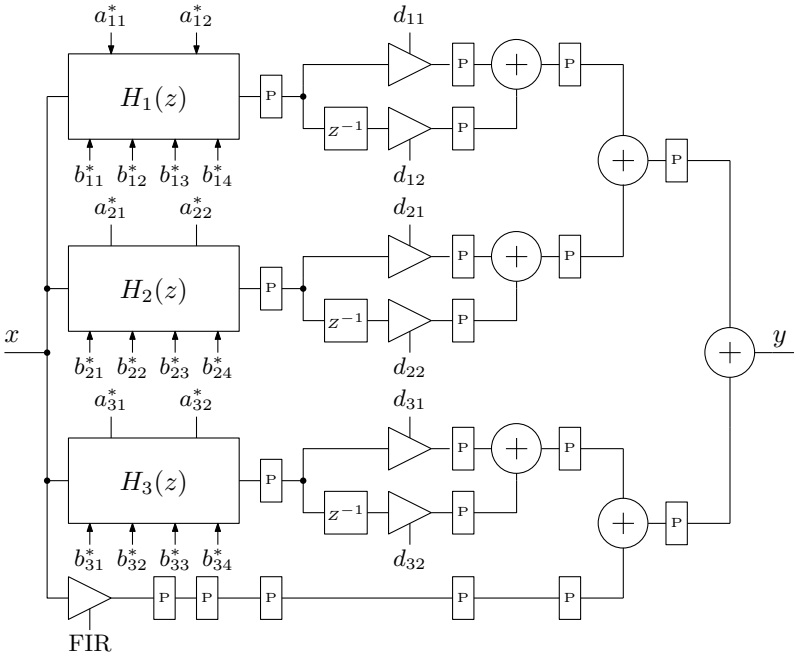


Fig. 16.26 Pipelined version of the parallel IIR low-pass filter

inserted. These methods are called constrained filter design techniques. Hardware-efficient pipelinable transfer functions can be resulted from these techniques.

Design Performance

Low-pass IIR filter of order 6 is implemented on NEXYS DDR2 artix7 FPGA device (xc7a100t-3csg324). This LPF is verified by taking two sinusoidal signals of frequencies 22 and 20 KHz. These two signals are multiplied and output of the multiplier is given to the LPF. The sampling frequency is taken as 100 KHz and thus the LPF filters out the signals whose frequency is greater than 25 KHz. Figure 16.27 shows output of the LPF which is a tone of 2 KHz. The original signal (MATLAB output) and the FPGA-based filtered output (delayed version) is compared in Fig. 16.27. An 18-bit fixed point data width is used in this design where 10-bits are used for fractional part. An RMSE of 0.0059 is achieved using a word length of 18-bits for parallel IIR implementation. An RMSE of 0.00584 is achieved when it is implemented for higher-frequency applications.

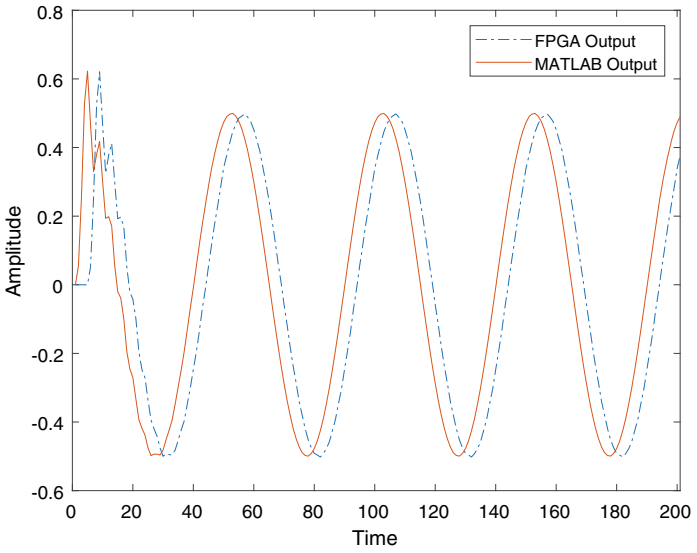


Fig. 16.27 Output of the low-pass IIR filter for same input as applied to FIR filter in the previous section

Comparison of Different IIR Filter Structures

Table 16.2 shows the comparison of the performances of FPGA implementation of different IIR LPF structures. Transposed direct form structures are having almost same performance. The cascaded structures may be preferred over direct forms because of having less quantization noise. Cascaded structures have higher latency and consume slightly higher resources. The parallel structures are more popular and also they have similar quantization noise immunity. Parallel structures are having parallel branches and thus latency is reduced but consumes more resources. Pipelining in IIR filters is very necessary as they have higher critical path. This way higher maximum frequency can be achieved at the cost of extra hardware.

16.2.5 Conclusion

FPGA implementation of the different varities of IIR filter structures and their performance comparison is presented here. An LPF is designed using different IIR structures to demonstrate the difference in their implementation. Direct form structures, systolic in nature, directly implement the IIR transfer functions. Transposed structures have shorter critical path. Cascaded and parallel structures have some immunity towards quantization noise compared to the direct forms. All IIR filters are not suitable for high-frequency applications and thus pipelining is important. Look-ahead

Table 16.2 Performance comparison of different structures

Structures	CLK_{min} (ns)	Slice Reg	Slice LUT	DSP48	OS*	Power
Transposed Direct Form I	6.9	222	228	10	71	0.029 W
Transposed Direct Form II	6.9	240	231	10	85	0.027 W
Cascaded form	6.9	226	228	13	67	0.030 W
Parallel form	6.9	226	228	13	67	0.0584 W
Pipelined parallel form	4	591	527	22	206	0.067 W

*: OS—Occupied slices

techniques to insert pipeline registers in IIR filters are discussed. Scattered look-ahead technique is adopted to implement parallel IIR filter and high frequency is achieved. Design performance is measured by measuring RMSE.

16.3 FPGA Implementation of K-Means Algorithm

In machine learning or data science, clustering is a very important technique to analyse huge data sets by segregating them according to their features. A bigger sub-set is divided into several sub-sets where each sub-set is having similar data samples. Knowledge of homogeneous groups can lead to apply many other optimization techniques on the data set. Many clustering algorithms exist in literature but K-means [44] algorithm is discussed in this section.

K-means algorithm is popular because of its simplicity and also this algorithm can be easily applied to the unsupervised data set. This algorithm tries to partition a data set into K groups in an iterative fashion. It tries to group the similar data samples in a cluster in every iteration. There are some techniques based on which similarities between two data samples or two vectors are computed. K-means algorithm produces K homogeneous groups where each group is having similar data samples. Groups are non-overlapping means no data sample can belong to two or more groups.

If any application, real-time clustering is required then it is important to implement the K-means algorithm on some hardware platform like FPGA. In this section, an innovative architecture for the K-means algorithm is proposed and implemented on FPGA. This is described below in detail.

Algorithm 16. 1 K-means algorithm

Input: Data Matrix $A \in \mathscr{R}^{M \times N}$, number of clusters ($K$) and number of iterations (I).
Output: K clusters with its centroids ($cntd$).
1: **Initialization** Randomly selects K number of elements from the matrix A. Initially set them as $cntd^k = A_k$ for $k = 1$ to K. Initially, $cltr^k$ is a null vector for $k = 1$ to K.
2: **for** $i \leftarrow 1$ to I **do**
3: **for** $j \leftarrow 1$ to N **do**
4: **for** $k \leftarrow 1$ to K **do**
5: $\lambda = \text{argmin} \, |A_j - cntd^k|_2^2$
6: **end for**
7: $cltr^{\lambda} = [cltr^{\lambda} A_j]$
8: **end for**
9: $d \in \mathscr{R}^{M \times 1} = 0$
10: **for** $k \leftarrow 1$ to K **do**
11: **for** $j \leftarrow 1$ to $size(cltr^k)$ **do**
12: $d = d + cltr_j^k$
13: **end for**
14: $cntd^k = d/size(cltr^k)$
15: **end for**
16: **end for**

16.3.1 K-Means Algorithm

Algorithm 5 summarizes the K-means algorithm. The data set $A \in \mathscr{R}^{M \times N}$ is taken as input set to the algorithm. In this algorithm, K represents the number of clusters and I denotes the total number of iterations. This algorithm segregates the data samples in K clusters in I iterations. Initially, the centroids are set as $cntd^k = A_k$ for $k = 1$ to K. Here, A_j denotes the jth column of A and $cntd^k$ is the kth centroid.

Similarities between the centroids and the columns of A are computed in the next stage. Techniques such as correlation, Euclidean distance and Manhattan distance can adopted to find similarity between two vectors. Here, euclidean distance is chosen. The Euclidean distance between two vectors of length M is computed as

$$d(x, y) = \sqrt{(x_1 - x_2)^2 + (x_1 - x_2)^2 + ... + (x_M - x_M)^2} \tag{16.28}$$

The arithmetic operations required to evaluate $d(x, y)$ are subtraction, squaring, addition and square root. Computational complexity of the square root operation is high compared to other operations. Here, partial Euclidean distance is computed by avoiding the square root operation. This partial Euclidean distance is evaluated in step 5 of Algorithm 5. The parameter λ represents the index of the column that gives minimum Euclidean distance.

The cluster formation is done in step 7 once the similar columns are identified. The parameter λ can take any value from 1 to K. Steps 9–15 of Algorithm 5 are for the averaging step. The elements of each cluster are accumulated and the accumulation result is divided by the size of that cluster. New values of the centroids are computed

after this step. The above-mentioned steps are repeated for maximum I number of iterations.

16.3.2 Example of K-Means Algorithm

An example of K-means algorithm is shown in this section. In this example, K-means algorithm is applied to group 8 data elements into three clusters, viz, Cluster 1, Cluster 2 and Cluster 3. Initially, data elements are $A1 = (2, 10)$, $A2 = (2, 5)$, $A3 = (8, 4)$, $A4 = (5, 8)$, $A5 = (7, 5)$, $A6 = (6, 4)$, $A7 = (1, 2)$ and $A8 = (4, 9)$.

Iteration 1

1. Choose initial centroids (seeds) of three clusters as A1 (Cluster 1), A4 (Cluster 2) and A7 (Cluster 3). Seed1 = A1 = (2,10), Seed2 = A4 = (5,8), Seed3 = A7 = (1,2) ad A8 = (4,9).
2. Calculate Euclidean distance between each data point with respect to the three seeds according to the following formula:

$$d(a, b) = (x_b - x_a)^2 + (y_b - y_a)^2$$

   (a) w.r.t A1: d(A1,seed1) = 0, d(A1,seed2) = 13, d(A1,seed3) = 65, A1 → Cluster 1.
   (b) w.r.t A2: d(A2,seed1) = 25, d(A2,seed2) = 18, d(A2,seed3) = 10, A2 → Cluster 3.
   (c) w.r.t A3: d(A3,seed1) = 72, d(A3,seed2) = 25, d(A3,seed3) = 53, A3 → Cluster 2.
   (d) w.r.t A4: d(A4,seed1) = 13, d(A4,seed2) = 0, d(A4,seed3) = 52, A4 → Cluster 2.
   (e) w.r.t A5: d(A5,seed1) = 50, d(A5,seed2) = 13, d(A5,seed3) = 45, A5 → Cluster 2.
   (f) w.r.t A6: d(A6,seed1) = 52, d(A6,seed2) = 17, d(A6,seed3) = 29, A6 → Cluster 2.
   (g) w.r.t A7: d(A7,seed1) = 65, d(A7,seed2) = 52, d(A7,seed3) = 0, A7 → Cluster 3.
   (h) w.r.t A8: d(A8,seed1) = 5, d(A8,seed2) = 2, d(A8,seed3) = 58, A8 → Cluster 2.

   At the end of this step, we have Cluster 1: (A1), Cluster 2: (A3,A4,A5,A6,A8), Cluster 3: (A2,A7)
3. Find the centroid (seed) of the newly formed clusters by averaging. The new centroids are C1: (2,10), C2: ((8+7+5+6+4)/5,(4+8+5+4+9)/5) = (6,6), C3: ((2+1)/2,(5+2)/2) = (1.5,3.5).

Iteration 2

At the end of iteration 2, we get Cluster 1: (A1,A8), Cluster 2: (A3,A4,A5,A6) and Cluster 3: (A2,A7) with centroids C1: (3,9.5), C2: (6.5,5.25) and C3: (1.5,3.5).

Iteration 3

At the end of the iteration 2, we get Cluster 1: (A1,A4,A8), Cluster 2: (A3,A5,A6) and Cluster 3: (A2,A7) with centers C1: (3.66,9), C2: (7,4.33) and C3: (1.5,3.5).

16.3.3 Proposed Architecture

Figure 16.28 shows the proposed data path architecture of the clustering algorithm. Parameters chosen for the implementation are $M = 2$, $N = 8$ and $K = 3$. This is a prototype implementation but can be adopted for higher data samples. This algorithm has three main steps, viz., Euclidean distance computing, sorting and average computation. These steps are executed sequentially. This architecture is hardware efficient and has moderate timing complexity. All the major blocks are explained below.

Data Acquisition and Initialization

The input data matrix (A) can be acquired in real-time or can be pre-stored. Here signal acquisition is ignored for simplification. Matrix A is stored in *A_mem* which is a bank of m memory elements. Each element is realized using dual port rams. Port B is used for reading and port A is used for writing. In K-means algorithm, K elements are randomly chosen. Thus, K data samples from the *A_mem* are read and written to

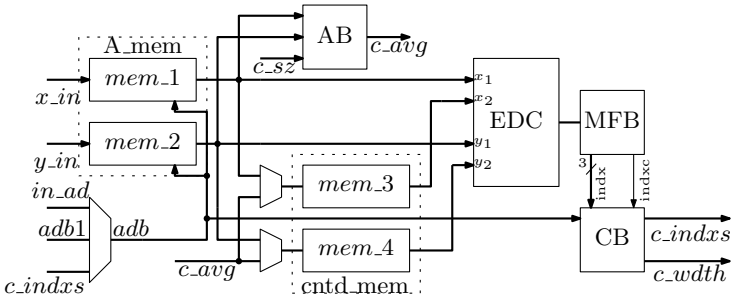


Fig. 16.28 Proposed architecture of the K-means algorithm

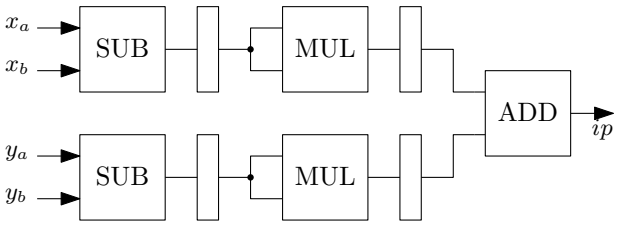


Fig. 16.29 Proposed architecture of EDC

C_mem. A ROM can be used to provide these initial indices or they can be generated randomly. This proposed scheme is shown in Fig. 16.28.

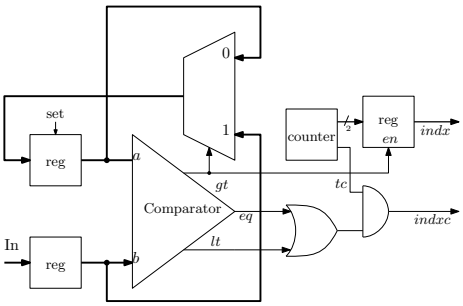
Euclidean Distance Calculator

Partial Euclidean distance between the centroids and the data samples is computed by the Euclidean Distance Calculator (EDC) block. Figure 16.29 shows the proposed architecture of the EDC block. Subtractors are placed at the first stage, and in the next stage, squaring operation is performed. Full-length multiplier is not required to compute square operation. An adder tree is placed at the last stage to add all the square values.

Minimum Finder Block

The output of the EDC block is fed to a Minimum Finder Block (MFB) which sorts a serial stream of data and finds minimum of it. Figure 16.30 shows the architecture of the MFB block. Initially, input *A* of the comparator is set and this input is selected by signal *gt*. MFB gives the index (*indx*) of the vector for which *ip* is minimum. Another output of this block is *indxc*.

Fig. 16.30 Proposed architecture of MFB



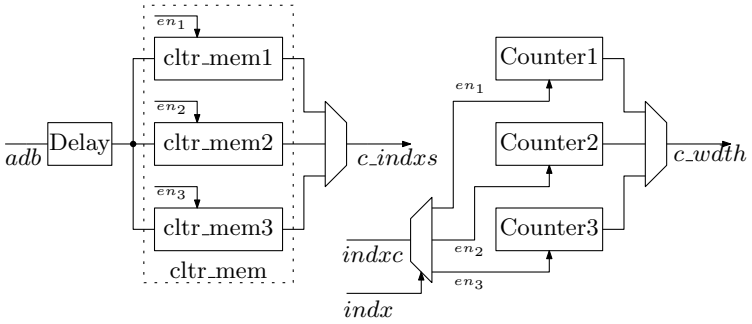


Fig. 16.31 Proposed architecture of CB

Cluster Block

Cluster Block (CB) counts the number of data samples in a cluster. Its structure is shown in Fig. 16.31. It receives *indxc* and *indx* outputs from MFB. CB generates K control signals en_1, en_2, en_3. These control signals are used to increment K number of counters. Counters hold the size of their respective clusters after the sorting operation. Simultaneously, these control signals are used to write the value of addresses (*adb*) in respective memory elements for which they are generated. Here, K memory elements are used to store the indices that belong to each cluster.

Average Block

Averaging is the last step of the K-means architecture. In the averaging step, average of the elements selected in each cluster is computed. An Average Block (AB) is proposed here and this block is shown in Fig. 16.32. Serial approach is adopted here to compute average as division operation involves very high hardware complexity. Averaging operation is done for elements in the X-co-ordinates first and then for elements in the Y-co-ordinates. A single divider is used here and it is non-restoring

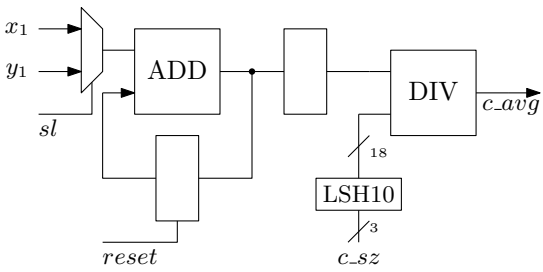


Fig. 16.32 Proposed architecture of AB

Table 16.3 Design performance

Design metrics	Values
Device	(xc7a100t)
Parameter (M, N, K, I)	2, 8, 3, 3
Slice registers	557
Slice LUTs	527
Occupied slices	268
DSP blocks	2
Maximum frequency	250
Dynamic power (μW)	57

algorithm based radix-2 divider [59]. The LSH10 block stands for 10-bit wired left shift.

16.3.4 Design Performance

K-means algorithm is implemented on NEXYS DDR2 artix7 FPGA device (xc7a100t-3csg324) here for parameters $M = 2$, $N = 8$ and $K = 3$. The proposed architecture uses 18-bit fixed point data width where 10-bits is used for the fractional part. Table 16.3 shows the design performance of the proposed architecture.

Hardware Complexity

Consumption of memory elements and resources for the proposed prototype architecture is shown in Table 16.4 and in Table 16.5, respectively. Here, k and n denote the number of bits to represent K and N, respectively. This proposed architecture is hardware-efficient as a single divider block and a single accumulation unit is used.

Table 16.4 Estimation of memory elements

Memory elements	Word per cycle		Size
	Write	Read	
A_mem	M	M	$M \times N \times 18$
cntd_mem	M	M	$M \times K \times k$
cltr_mem	K	K	$K \times N \times k$
indx_mem	1	1	$K \times n$

Table 16.5 Estimation of hardware complexity

Blocks	Multiplier	Comparator	*Add_sub*	Divider
EDC	K	0	$2M-1$	0
MFB	0	1	1	0
AB	0	0	1	1

Timing Complexity

Timing complexity of the proposed design is analysed to form an expression for the overall execution time. Initial centroids are loaded to *cntd_mem* from *A_mem* in K clock cycles. Euclidean distances are computed in the next step. Euclidean distance computation stage takes $(K+1) \times N + l_{ip}$ number of clock cycles, where $l_{ip} = 3$ is the latency of the EDC block. Even though sorting operation is executed in parallel to the computation of Euclidean distance but two clock cycles are wasted to start the next step. Averaging operation is the next step and its timing complexity is $2K + N + l_{dv}$ clock cycles. Here $l_{dv} = 30$ is the latency of the divider block. Timing complexity for this prototype design is $(3K + N(2+K) + l_{ip} + l_{dv} + 2)I$, where I denotes the total number of iterations. If $I = 3$ number of iterations is considered, then 246 clock cycles are required and, for a frequency of 250 MHz, total execution time is 984 μs.

16.3.5 Conclusion

Novel architecture is proposed here to implement the K-means algorithm on FPGA device. Prototype design is targeted to the NEXYS4 DDR2 FPGA device. This design is scalable and any size of matrix A can be clustered. This design is hardware efficient with moderate execution time. A single divider is used here to reduce hardware resources. An analysis of timing and hardware complexity is also presented

16.4 Matrix Multiplication

An algorithm in any field of application can involve matrix multiplication operation for evaluation of the final output. The number of matrix multiplication operations present in an algorithm decides computation complexity of that algorithm. High computation complexity means greater hardware complexity. Thus, efficient implementation of matrix multiplication operation is important. Matrix multiplication operation is divided into three categories in this section and these are

1. Matrix Multiplication by Scalar–Vector Multiplication
2. Matrix Multiplication by Vector–Vector Multiplication
3. Systolic Array for Matrix Multiplication

The pros and cons of these techniques are discussed below.

16.4.1 Matrix Multiplication by Scalar–Vector Multiplication

Almost all signal or image processing algorithms involve Scalar–Vector multiplication which is a part of matrix multiplication. Hardware implementation of this operation is discussed here with an example. A 6×3 Matrix (A) is multiplied by a 3×1 Matrix (B) and the result is a 6×1 column vector (C). This matrix–vector multiplication operation is shown in Fig. 16.33. This multiplication operation can be done either by vector–vector multiplication or by scalar–vector multiplication. In the latter method, one column of matrix A and one element of column vector B are fed to the computing processor. Multiplication result of this step is accumulated with the multiplication of the second column of A and the second element of B. Thus, this matrix–vector multiplication objective is achieved through scalar–vector multiplication and accumulation operation. The multiplication between A and B can be expressed as

$$A.B = A_1.B_{1,1} + A_2.B_{2,1} + A_3.B_{3,1} \tag{16.29}$$

The computing unit is designed with the help of a basic Multiply and ACumulate (MAC) unit. MAC unit multiplies two elements and accumulates. MAC unit schematic is shown in Figs. 16.34 and 16.35 shows the overall computing unit. Six MAC blocks are used in this computing unit. Each MAC block has latency of two clock cycles and vector C is computed after four clock cycles. The role of the reset (*rst*) input signal is very important here. The register after the adder in the MAC block is needed to be cleared by this *rst* signal before multiplication and before starting another multiplication operation. The timing diagram for multiplication and accumulation operation by a MAC block is shown below in Fig. 16.36. Here, P is the output of the MAC, $P1$ is the first multiplication output, $P2$ is the first accumulation output and $P3$ is the final output. The overall computing unit consumes six multipliers and six adders.

Fig. 16.33 Multiplication of a matrix and a vector

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \\ A_{41} & A_{42} & A_{43} \\ A_{51} & A_{52} & A_{53} \\ A_{61} & A_{62} & A_{63} \end{bmatrix} \times \begin{bmatrix} B_{11} \\ B_{21} \\ B_{31} \end{bmatrix} = \begin{bmatrix} C_{11} \\ C_{21} \\ C_{31} \\ C_{41} \\ C_{51} \\ C_{61} \end{bmatrix}$$

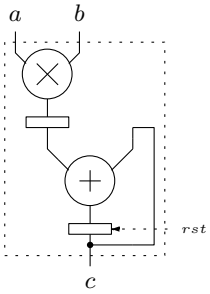


Fig. 16.34 MAC schematic for scalar–vector multiplication

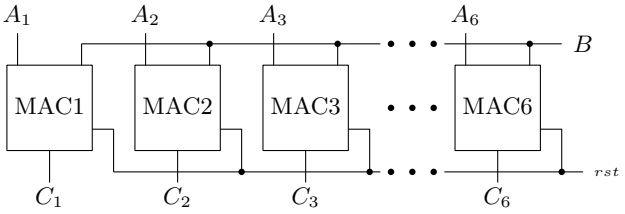


Fig. 16.35 Scalar–vector multiplier

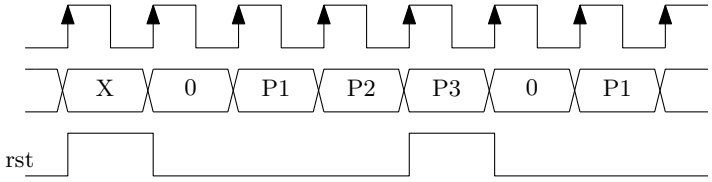


Fig. 16.36 MAC block output versus reset input status

16.4.2 Matrix Multiplication by Vector–Vector Multiplication

Vector–vector multiplication operation is another way to achieve matrix multiplication. Algorithms can also involve separate vector–vector multiplication operation. A vector–vector multiplication is sometimes called inner product.

In this section, matrix A (6×6) is multiplied with matrix B (6×6) using vector–vector multiplication technique. The multiplication result is matrix C (6×6). The schematic of the vector–vector multiplier is depicted in Fig. 16.37. Matrix A is stored in one memory bank and B is also stored in another memory bank. These memory blocks can be configured either using ROM or RAM. The memory banks have six outputs and each output is a word. In this example, A^i denotes ith row of A and B_i denotes the ith column of matrix B.

Multipliers belong to the first stage of the Vector–vector multiplier and the second stage is an adder tree. If there are n elements in a vector, then n multipliers should

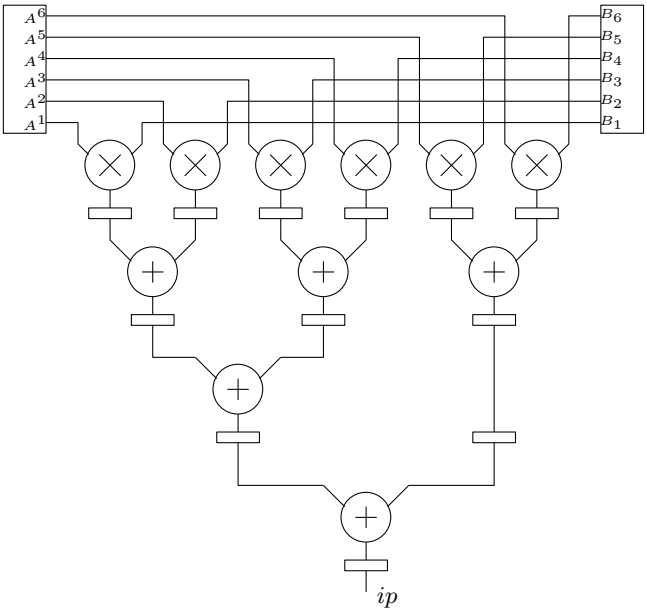


Fig. 16.37 Vector–vector multiplier

be used and $(n-1)$ adders will be required in the adder tree. The pipeline registers are placed between a multiplier and an adder or inserted between two adders.

Timing Complexity

Timing complexity of this operation in multiplying two 6×6 matrices is expressed as

$$T = T_{lat} + n^2 \tag{16.30}$$

Here, T_{lat} is the latency of the vector–vector multiplier and the second term is for multiplying two $(n \times n)$ matrices. Thus, total time taken to multiply two 6×6 matrices is $(4 + 36 = 40)$ clock cycles where $T_{lat} = 4$. After the latency period, one inner product is produced per cycle. These inner products can be written to any other memory blocks for further use.

16.4.3 Systolic Array for Matrix Multiplication

Another technique to achieve matrix multiplication operation is discussed in this section. This technique is called systolic array multiplication. An architecture is

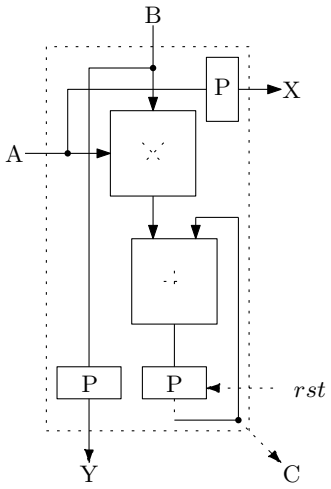


Fig. 16.38 MAC architecture

called systolic if it is homogeneous so that data processing units can be tightly coupled. In this type of architecture, each data processing units compute partial result independently. Systolic architectures are tightly coupled so that timing requirements can be easily met and thus they are preferred more.

Here, matrix A (6×6) and matrix B (6×6) are multiplied together and the result is matrix C (6×6). The A and B are stored in memory banks *A_mem* and *B_mem*, respectively. These memory banks have six memory elements and each can be configured as ROM or RAM. The columns of B and rows of A can be accessed serially. Here, A^i denotes ith row of A and B_i denotes the ith column of B.

Systolic matrix multiplier for 3×3 matrices is discussed first and this architecture is used to develop systolic architecture for 6×6 matrices. In this section also Multiply–Acumulate unit (MAC) is used as a basic building block but this MAC has a different structure compared to the previous MAC structure. Figure 16.38 shows the structure of the MAC and it implements the function $P = A \times B + P$. It has a reset input for starting a new accumulation. The timing diagram of the MAC block is very similar to the timing diagram shown in Fig. 16.36.

The systolic architecture to multiply 3×3 matrices is shown in Fig. 16.39 and this architecture uses six MAC blocks. Rows of matrix A from one side and columns of B from another side are fed to the structure. The second row of A and the second column of B are delayed by one clock cycle. Similarly, the third row of A and the third column of B are delayed by two clock cycles. This is because the objective is to compute one element of matrix C per cycle.

The timing diagram for the above-mentioned architecture is shown in Fig. 16.40. The systolic architecture for 3×3 matrices can be used to multiply two 6×6 matrices. Figure 16.41 shows the systolic matrix multiplier for 6×6 matrices. The columns of matrix B and rows of matrix A are fed to the systolic array through multiplexers. Multiplication of two 6×6 matrices is achieved in four phases. In the first

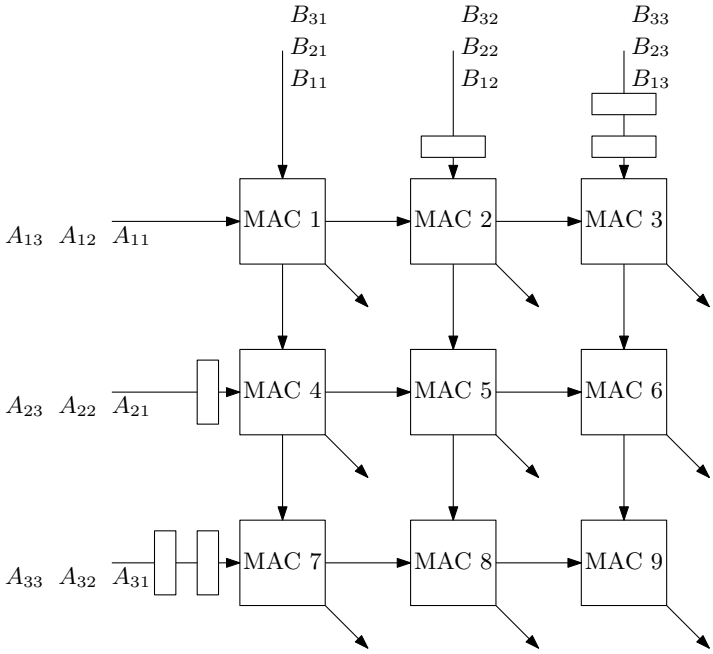


Fig. 16.39 Systolic matrix multiplier for 3 × 3 matrix

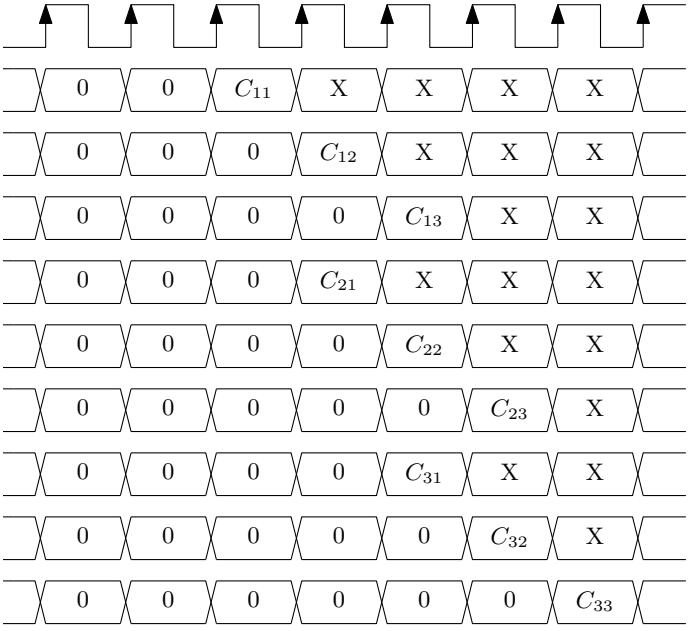


Fig. 16.40 Timing diagram for systolic multiplication of two 3 × 3 matrices

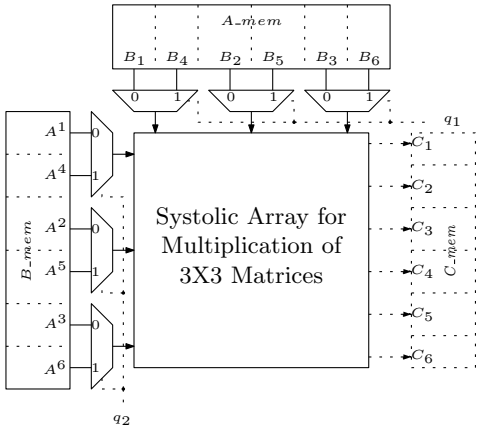


Fig. 16.41 Systolic matrix multiplier for 6 × 6 matrices

phase, the first three rows of A are multiplied by the first three columns of B. In the second phase, the first three rows of A and the last three columns of matrix B are multiplied. Other two phases follow similar procedure. The phase-wise evaluation of C is shown in Fig. 16.42.

Timing Complexity

The first part of timing complexity is (4 × 6 = 24 clock cycles) as multiplication is achieved in four clock cycles. Reset signal is asserted three times in between the phases and this is why three clock cycles are elapsed for clearing the register in the MAC. Extra four clock cycles are required as the architecture is systolic. The total number of clock cycles are

$$T_{lat} = (4 \times 6) + 3 + 4 = 31 \text{ clock cycles} \tag{16.31}$$

This systolic structure has better timing complexity than the vector–vector multiplication architecture. But every architecture has advantages or disadvantages depending

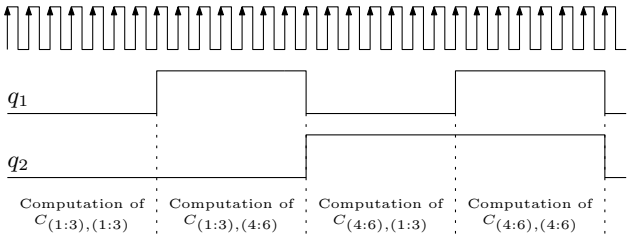


Fig. 16.42 Timing diagram for the multiplication of two 6 × 6 matrices

upon the application. If hardware resources are limited then systolic architecture may not be suitable as compared to other methods. Designers need to choose architecture based on their application.

16.5 Sorting Architectures

16.5.1 Parallel Sorting Architecture 1

This is a basic parallel sorting architecture and it is discussed here for $n = 8$. Sorting the adjacent elements is the technique behind this architecture which is shown in Fig. 16.43. It consumes 25 BN blocks and it sorts eight data elements in the descending order. A Basic Network (BN) block is designed to sort two data elements in descending order. A BN block has a comparator and two MUXes. It outputs MAX and MIN out of two data elements. Figure 16.44 shows the architecture of the BN block.

16.5.2 Parallel Sorting Architecture 2

Marge parallel sort algorithm is based on dividing the array into sub-arrays and then merging them one by one. Based on this technique, a parallel sorting structure

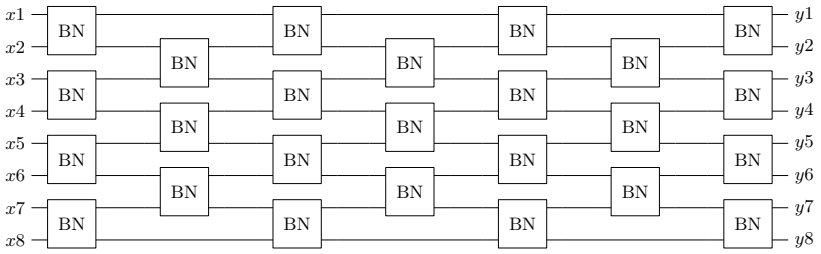


Fig. 16.43 An alternate architecture for sorting 8 data elements

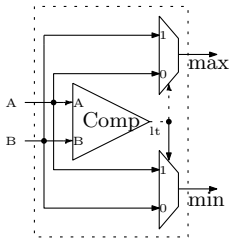


Fig. 16.44 Block diagram of the basic network block

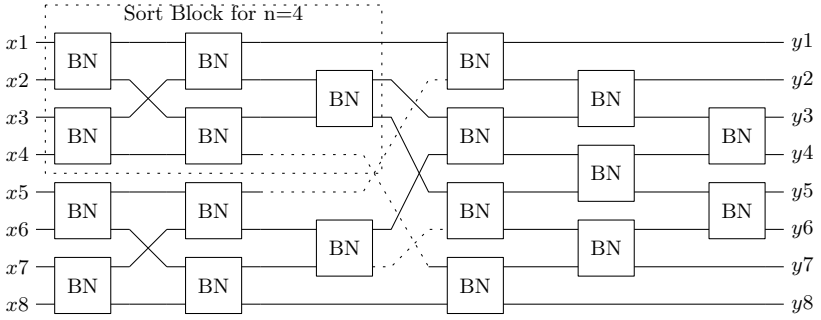


Fig. 16.45 Architecture of the parallel sort architecture based on marge sort algorithm

is discussed here for $n = 8$. This structure is hardware-efficient than the previous structure. Array of four elements is sorted first and then the sorted sub-arrays are sorted. Figure 16.45 shows this parallel sort architecture and it consumes total 19 basic node blocks compared to the previous architecture. The sort block for $n = 4$ is shown by a dotted box.

16.5.3 Serial Sorting Architecture

The parallel sort architecture 1 is regular and can be divided into stages with each stage having same hardware. This structure consumes 23 BN blocks and sorts 8 data elements in just single iteration. If it is required to sort eight data elements in some iterations, then a serial sort architecture [48] can be derived from this structure. A basic SB is designed and this SB is reused in every iteration to get a serial architecture. Figure 16.46 shows the architecture of this SB.

This SB consumes seven BN blocks and BN blocks are described previously. The serial sort architecture is depicted in Fig. 16.47. In this serial architecture, inputs are fed to the SB through eight MUXes. A start signal initially selects the inputs and fed

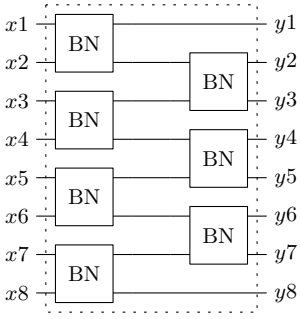


Fig. 16.46 Structure of the sub-block (SB)

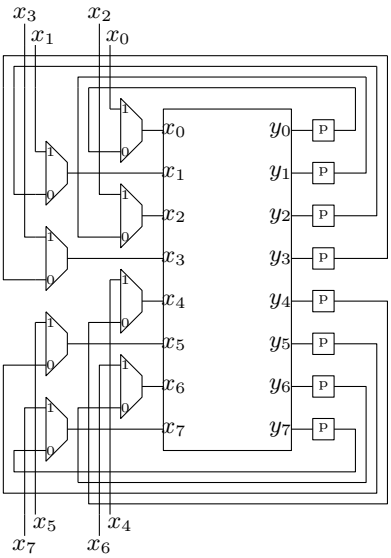


Fig. 16.47 An architecture for sorting 8 data elements serially

them to the SB in the first iteration. In the next iterations, output signals of the SB are fed back to the input of SB. The eight data elements are sorted after some iterations. Four iterations are sufficient to run to sort eight data elements. This serial structure takes more time to sort but uses less resources. Thus, can be used in architectures where less resources must be consumed.

16.5.4 Sorting Processor Design

In all the previous architectures, data samples are accessed parallely. But in some cases, we have to access the data samples serially from a memory element as parallel data access is very costly. Serial to parallel data conversion is possible but again this process also consumes extra hardware. Parallel sorting architectures are costly as they consume more number of comparators. This is why alternate sorting architectures should be adopted.

A serial sorting architecture is discussed here which is based on the insertion sort technique. This architecture works on a serial stream of M data words. Values from the unsorted part are inserted at the correct position of the sorted part in the insertion sort technique. Two data elements are sorted in a similar way as it was done previously for the parallel structures. But here architecture of the BN block is slightly different and architecture of this BN block is shown in Fig. 16.48. Maximum of two data elements is stored in a register and again fed back to the comparator. One *rst* signal is there for resetting the registers as per requirement.

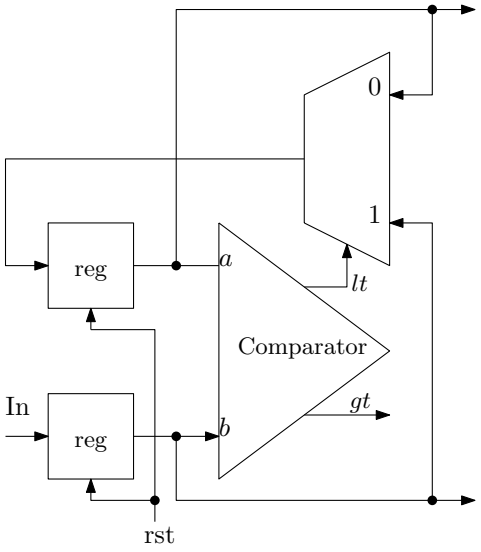


Fig. 16.48 Architecture of the BN block for the sorting processor

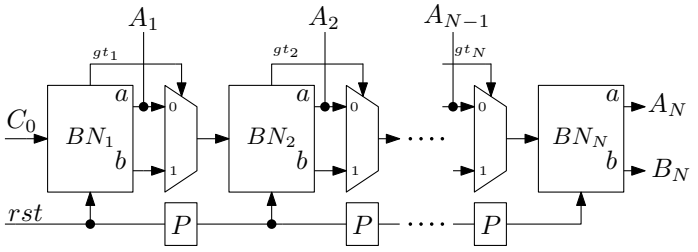


Fig. 16.49 Architecture of the sorting network-N using BN blocks

Figure 16.49 shows the insertion sorting algorithm-based sorting network to sort M data elements. N number of BN blocks are used in this network. Serial data is applied to the C_0 pin. The BN_1 block sorts two data and A_1 is maximum of them. The minimum value from the BN_1 block is passed to the next block through a multiplexer. If a data greater than the present value of A_1 reaches at input of BN_1, then this value will be assigned to A_1 and the previous value of A_1 is passed to the next BN block. Other BN blocks follow the same operation. After the M number of clock cycles, A_1 holds the maximum of M data elements. Similarly, values of A_2, A_3, ... , A_N are evaluated. The sorting network-N block takes $(M + N)$ clock cycles to sort N data elements.

The next job is to sort the $(M - N)$ elements. Figure 16.50 shows the sorting processor architecture which is built using the sorting network-N block. An output memory can also be used to store the sorted elements through a multiplexer unit.

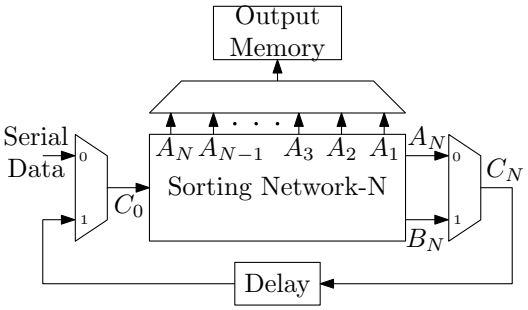


Fig. 16.50 Data path of the insertion sort algorithm-based sorting processor

Unsorted elements are again fed back to the sorting network-N through the C_N net. The unsorted elements are now sorted serially using the same block. A_1 holds the maximum value of the unsorted part after N clock cycles. This way the whole array of M data words is sorted.

An unsorted array is input to this sorting processor and this processor sort N elements in one iteration. The sort processor mainly consists of N BN blocks apart from the multiplexers and registers. If the value of N is increased, then the number of BN blocks will also increase. The timing complexity of this sort processor can be estimated in terms of M and N. The parameter M is preferably to be expressed as $M = k \times N$. Then timing complexity is $T_{sort} = M + (M - N) + (M - 2N) + ... + (M - (k - 1)N) + N$. If $N = 8$, then total 88 number of clock cycles are required to sort 32 data elements. This is obviously higher than the processing time of the parallel sort structures but this sort processor is hardware efficient.

16.6 Median Filter for Image De-noising

Implementation of a simple Median filter to remove noises from an image is discussed in this section. Various types of noises [57] like salt and pepper noise, Gaussian noise, periodic noise etc. can be present in an image. Salt and pepper noise is very common to any image. This noise can be easily characterized and removed in spatial domain using Median filter.

16.6.1 Median Filter

Many spatial filters are proposed in literature to remove salt and pepper noise. Median filter is a very common and popular for image denoising. Many variations of Median filter are also reported in literature but here the basic version is discussed. In median filtering, the pixel on which the window is operated is replaced by the Median value of all the pixels inside that window. A simple noisy greyscale image (Fig. 16.51) is

(a) An Grey scale image distorted with Salt and pepper noise.

(b) The filtered image output using Spatial Median Filter.

Fig. 16.51 An image which has salt and pepper noise is filtered out by spatial median filter

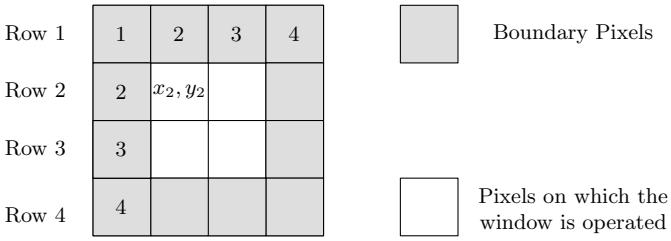


Fig. 16.52 Window operation for 4 × 4 image

taken as an example for implementation of median filter. In greyscale image, pixels values are ranging from 0 to 255. In an image affected by salt and pepper noise, noisy pixel value can be either very close to 255 or as small as 0. The filtering of an image depends on the sliding window operation. A $W \times W$ window can be of size 3×3, 5×5 or 7×7 for an $n \times n$ image and in a window their are W^2 pixels. Figure 16.52 shows sliding window operation for a simple 4×4 image. Here, there are four rows of pixels. In the first step, Row 1, Row 2 and Row 3. (x_2, y_2) is the centre pixel on which the 3×3 window is operated. Then the window slides to the right side and (x_3, y_3) becomes the centre pixel. Once the window slides reache the extreme right, another row takes part in the denoising operation. For example, here in the second step, Row 2–Row 4 are operated. Denoising operation can not be done on the boundary pixels as window operation can not be operated on them.

16.6.2 FPGA Implementation of Median Filter

A simple hardware implementation of the median filter is depicted in Fig. 16.53. In implementation of this median filter, there are three major steps which are sliding window operation, filtering operation and filtered image restoration. Image can be acquired and stored in a dual-port Input RAM of size $N \times N \times 8$. An up counter is used to write the pixels in the RAM block. Image pixels can be simultaneously read from the Input RAM block and fed to another block which performs window operation. This block feeds all the pixels of a current window to the median computation block. In the restoration phase, the median value replaces the pixel on which the window operated.

Here, the 3×3 window is chosen for image denoising. Figure 16.54 shows a simple scheme for sliding window operation [21]. Two line buffers are used in this scheme and size of each buffer is $N \times 8$. Initially, Row 1 is written to the Line buffer 1 through the DeMUX when *phI* signal is high. Line buffer 2 gets Row 2 when the *phII* signal is high. Phase signals *phI* and *phII* are opposite to each other and non-overlapping. Now, Line buffer 1 has Row 1 and Line buffer 2 has Row 2. Reading Row 3 from the input RAM and from both the buffers is done simultaneously. During this time, Row 3 is also written to Line buffer 1. Three clock cycles are required to form the 3×3 window using the nine registers.

Various architectures are also reported in literature to find median efficiently. Figure 16.55 shows the median computation block [31] which is used here. This

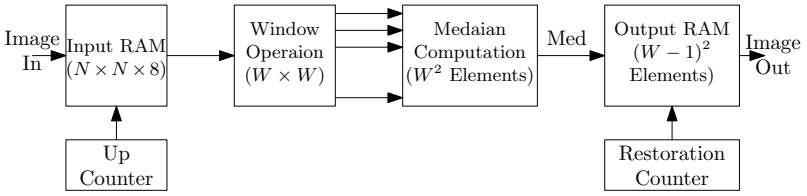


Fig. 16.53 A possible architecture of the spatial median filter

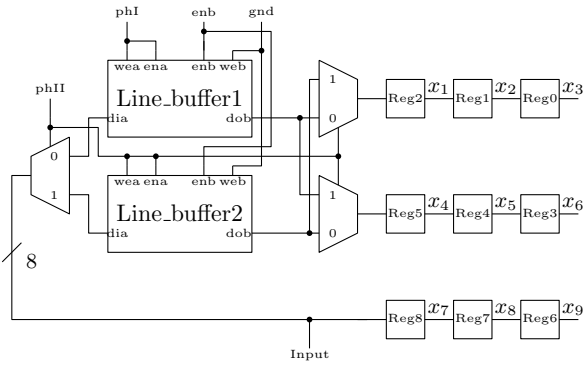


Fig. 16.54 A scheme for 3×3 window operation

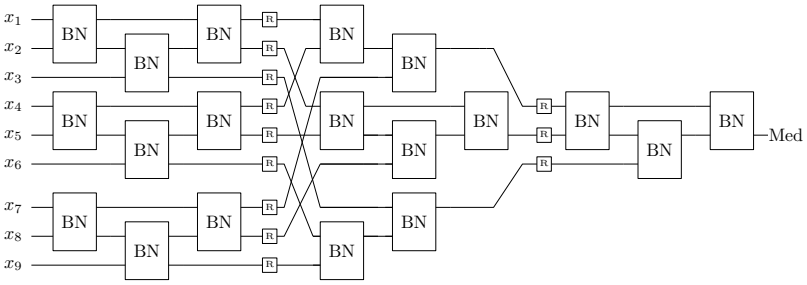


Fig. 16.55 An architecture for computing median out of 8 data elements

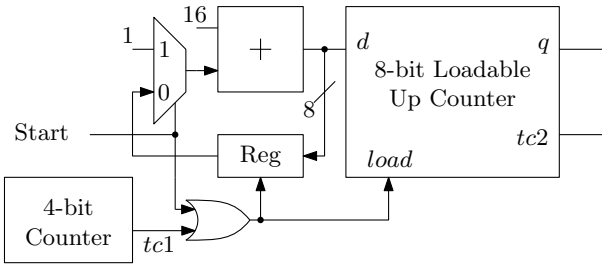


Fig. 16.56 An architecture for the restoration counter

parallel structure has total 19 BN blocks. The architecture of the BN block is already shown in the previous Sect. 16.5. Timing performance of this block is improved by placing pipeline registers. The number of BN blocks will increase if the value of W is increased.

In the image restoration step, noisy pixels are replaced with Median values. This restoration operation can be done separately or on the input image. But for simplification, a separate output RAM is used here. Now, the output RAM should initially contain the boundary pixels to retain the boundary pixels. A restoration counter is used to write the median values in the output RAM at their exact location. For a 16×16 image, the restoration counter will count as

$$17\ 18\ 19....30\ 33\ 34\ 35....46\ 49.... \tag{16.32}$$

Figure 16.56 shows a possible scheme for the restoration counter. Two loadable counters are used, one is of 8-bits and another is of 4-bits. 4-bit counter is used to generate the starting addresses ($d = 17, 33, 49, ...$) for the 8-bit counter. *start* signal initially loads the 8-bit counter with $d = 17$ and then both the counters start counting. The 4-bit counter counts up to 13 and generates terminal count signal ($tc1$) which again loads the 8-bit counter with the next load value. Both the counters have a common enable input.

16.7 FPGA Implementation of 8-Point FFT

Fast Fourier Transform (FFT) is very common in many signal processing applications to analyse signals in frequency domain. Eight-point FFT, based on Decimation In Time (DIT) algorithm, is implemented in this section. This implementation is targeted to Spartan 3E FPGA target and its design performance is obtained. This design is a basic prototype and thus may not be directly usable to a user but it will help to understand the implementation of FFT.

Figure 16.57 shows the signal flow diagram (SFD) for 8-point DIT-based FFT. Here, input samples are in reverse order and output samples are in correct order. Twiddle factors are also mentioned in that SFD. FPGA implementation of FFT algorithm is directly described here, and for the detailed theory of FFT algorithm, students are advised to follow the books on signal processing techniques.

Figure 16.58 shows the FFT processor. It has three inputs which are *clk*, *reset* and *start*. As soon as a pulse is given to the *start* input, the transform operation will be started. This processor has two output vectors which are *Real* and *Imag* representing real and imaginary values respectively. A 16-bit fixed point data representation is used in this design where 8-bits are reserved for the fractional part. Two's complement data representation is used in this case to represent both signed and unsigned data.

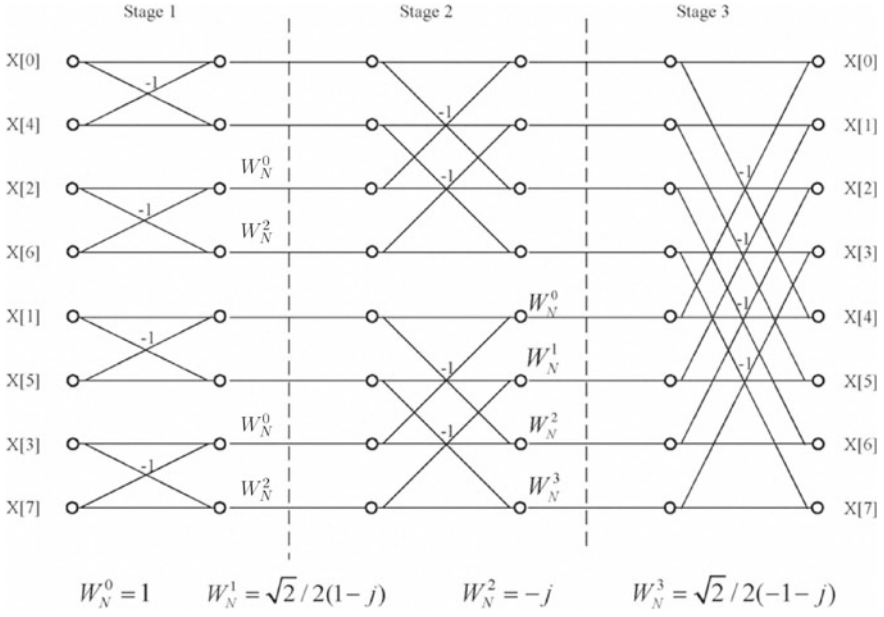


Fig. 16.57 Signal flow diagram for 8-point DIT based FFT

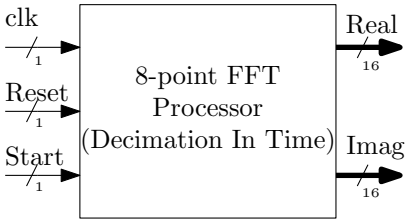


Fig. 16.58 8-point FFT processor

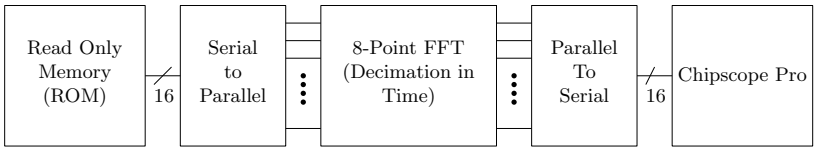


Fig. 16.59 8-point FFT processor

16.7.1 Data Path for 8-Point FFT Processor

The overall datapath of the FFT processor is shown in Fig. 16.59. Input data samples are stored in a ROM to verify the design. In a single clock cycle, one data sample is read from the ROM. Serial to Parallel (S2P) block converts the serial data stream from the ROM into a parallel data stream. FFT processor takes the data samples from the S2P block. Parallel to Serial (P2S) block converts the serial data stream from the FFT block to a serial data stream. Two P2S blocks are there, one for real and one for imaginary data. The output of the P2S blocks is sent to Chipscope Pro or directly to a personal computer for analysis.

FFT block in the data path is the major block which implements the SFD of the FFT algorithm. The FFT block is designed using the structural style by designing the sub-blocks first and then integrating them. FFT block architecture is shown in Fig. 16.60. Architecture is moderately optimized for performance enhancement. Figure 16.61 shows the smaller sub-blocks. In the SFD shown in Fig. 16.57, a basic butterfly block performs two basic operations, which are addition and subtraction. Thus, a basic Butter-Fly Block 1 (BF1) is designed which performs both addition and subtraction operations. The Butter-Fly block 2 (BF2) block is actually an optimized version of two BF1 blocks.

Multiplication of two complex numbers is expressed as

$$(a + jb) \times (c + jd) = ac + jad + jbc - bd = (ac - bd) + j(ad + bc) \quad (16.33)$$

As $(a + jb)$ is variable and $(c + jd)$ is constant, the complex multiplication operation is reduced to the following:

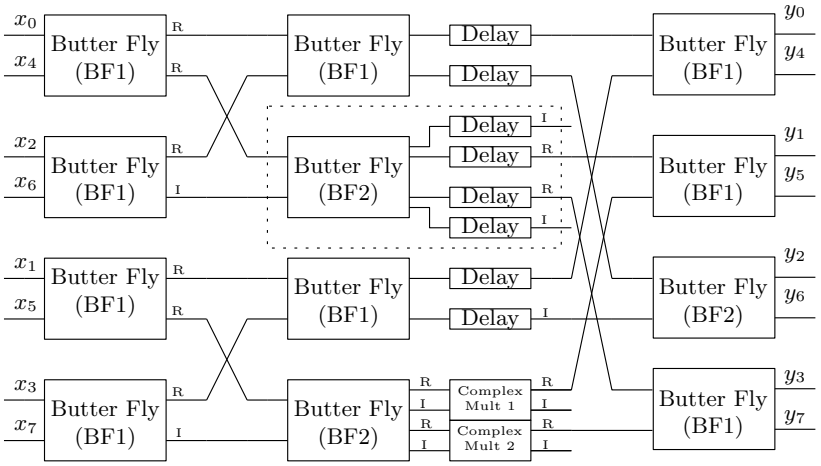
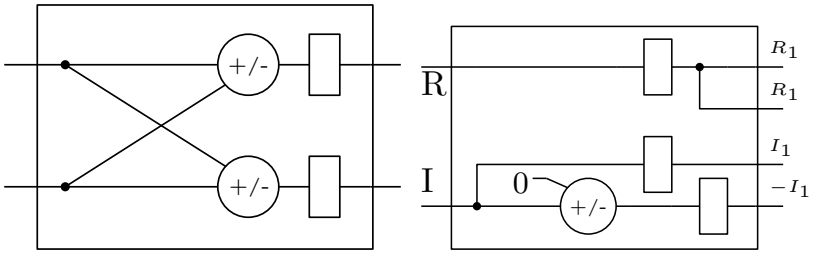


Fig. 16.60 8-point FFT processor



(a) Structure of BF1. (b) Structure of BF2.

Fig. 16.61 Structures of the sub-blocks of FFT processor

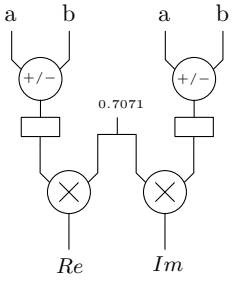


Fig. 16.62 Architecture of the complex multiplier

$$(a+jb)(0.7071-j0.7071) = 0.7071(a+b) - j0.7071(a-b) \quad (16.34)$$

$$(a+jb)(-0.7071-j0.7071) = -0.7071(a-b) - j0.7071(a+b) \quad (16.35)$$

Figure 16.62 shows the complex multiplier block. Multiplication by 0.7071 is performed by a constant multiplier. This reduces hardware consumption.

16.7.2 Control Path for 8-Point FFT Processor

Design of a signal controller block is very important in order to guarantee proper operation of the data path. Figure 16.63 shows the control path for the FFT processor mentioned above. Phase Generation (PG) and loadable counter are the two major blocks used in control path. Details of the PG block can be found in the Chap. 4. The phase signal $en1$ is generated by the *start* pulse. The $en1$ signal activates the ROM and also activates the S2P block for serial to parallel conversion. The counter 1 is used to provide addresses to the ROM and it generates $tc1$ signal after eight clock pulses. This terminal count signal ($tc1$) triggers another PG block. The latency of the FFT block is tracked by another set of PG and counter block. After the latency period of the FFT block, the second terminal count signal ($tc2$) is generated which again triggers the third set of PG-Counter blocks. The $tc2$ signal is used to load the parallel outputs to load to the P2S blocks and $en3$ signal is used for serial output from the P2S blocks. The first output of FFT block and $tc2$ signal is synced.

The delay blocks are added for synchronization. In the delay blocks, a number of registers are connected serially. Further optimization can be done in the FFT block and an example is shown by a dotted box in Fig. 16.60. Here, two delay blocks can be placed before the BF2 block instead of four delay blocks placed after BF2 block. This way searching the opportunity for optimization is necessary. The FFT processor is implemented on SPARTAN 3E starter kit and its performance is shown in Table 16.6.

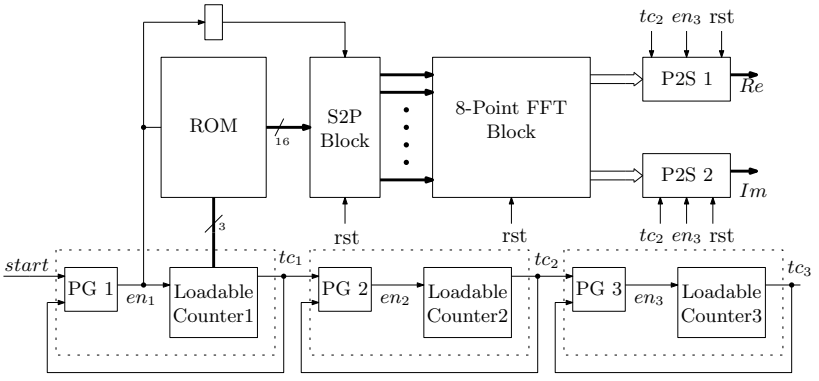


Fig. 16.63 Control path for the FFT processor

Table 16.6 Performance of FFT processor

Parameters	FFT implementation
FPGA device	xc3s500e-4fg320
Slice register count	1030 (11%)
4-input LUTs count	831 (%)
Occupied slices count	777 (16 %)
Dynamic power (mW)	229.51
Maximum frequency (MHz)	329

16.8 Interfacing ADC Chips with FPGA Using SPI Protocol

Real-world signals are digitized by ADC ICs and digital ICs like FPGAs are used to process the digitized signals. Thus, it is important to learn how to interface ADC ICs with the FPGA. In doing that, a separate HDL code is to be written according to the specifications related to a particular ADC IC. A tutorial on interfacing an ADC IC with FPGA is given in this section. This tutorial can be used to interface with other ADC ICs with minor modifications.

The selected ADC IC (MCP3008) [49] is a product of Microchip Company. This IC is having a 10-bit ADC and 8 input channels. The sampling speed is 200 ksps at VDD of 5.5 V. Minimum clock frequency is $18 \times f_{sample}$ where f_{sample} is the sampling frequency of the ADC. More details about this ADC IC can be found in the specification document of the IC.

The pin diagram of the ADC IC is shown in Fig. 16.64. This IC is having a SAR logic-based ADC. Figure 16.65 shows the basic functional diagram. Any input channel can be selected using the control inputs. The formula for interpreting the digital value from a 10-bit ADC chip is

$$\text{Digital Code} = \frac{1024 \times V_{IN}}{V_{REF}} \tag{16.36}$$

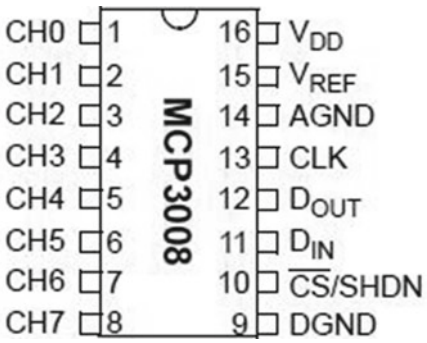


Fig. 16.64 The pin diagram of the ADC IC

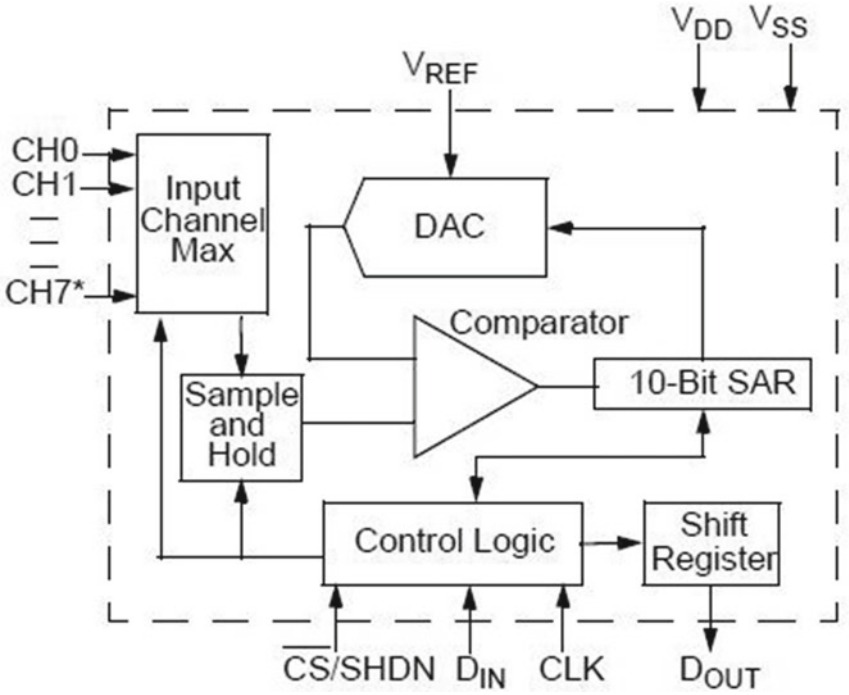


Fig. 16.65 The functional diagram of the ADC inside the IC

Any IC uses either SPI or I2C protocol for easy interfacing with digital controllers or FPGAs. This ADC IC uses SPI protocol and the shift register produces output data serially through the pin D_{OUT}. The frequency of the CLK signal is 18 times higher than the sampling frequency. The active low signal $\overline{CS}/SHDN$ controls the SAR logic of the ADC. The control word is provided through the D_{IN} serially. The timing diagram is shown in Fig. 16.66.

A Verilog code is written to interface the SPI protocol-based ADC chip and the SPI protocol is modelled using FSM based design style. Separate states for every status on the D_{IN} pin are defined. The start bit is kept high and it indicates the start of control signal. The next bit defines how the analog signals are acquired. It is high for single-ended application and it is low for differential application. The next three bits are used for selecting input channels. A valid word is formed after receiving the output data from the ADC IC through D_{OUT} pin. The SPI-based controlling of the ADC IC is shown in Fig. 16.67.

NEXYS 4 DDR board is chosen here to demonstrate that the ADC IC is interfaced with FPGA. The ADC IC is mounted on a PCB along with other ICs. A DC voltage source is used to feed analog input to the ADC IC. The voltage source is varied from 0 to 3.3 volts that is maximum voltage of an FPGA. The experimental setup is shown in Fig. 16.68 and all the 10 LEDs are blinking when voltage is 3.3.

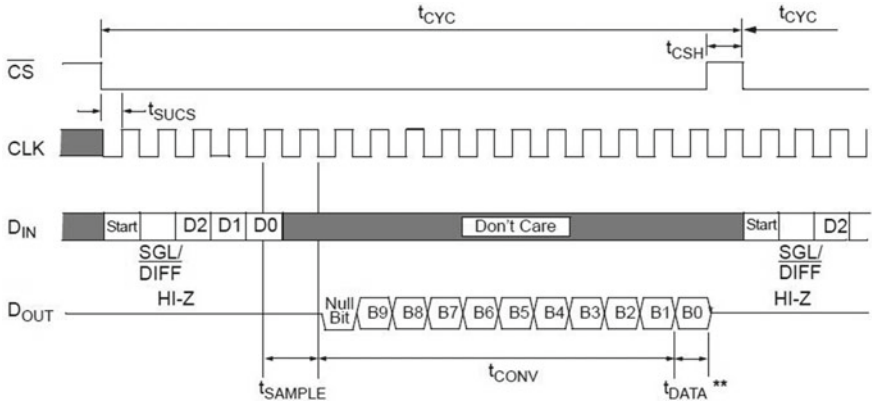


Fig. 16.66 Implication chart after second pass

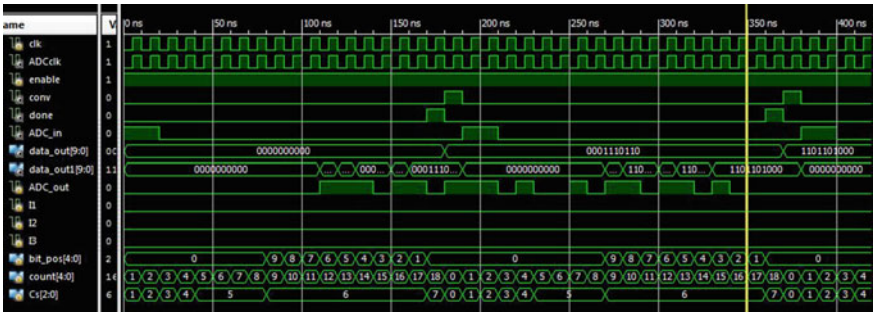


Fig. 16.67 XILINX simulation for SPI interfacing protocol

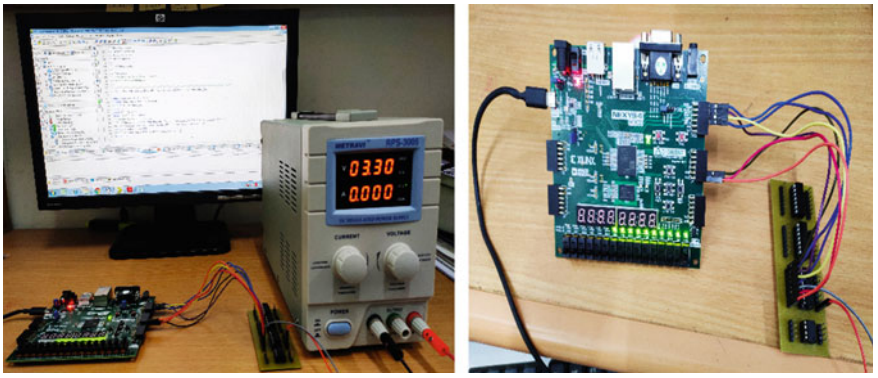


Fig. 16.68 Experimental set-up for verification of interfacing ADC with the FPGA board

```
module ADC_top(clk,enable,I1,I2,I3,ADC_in,ADC_out,
ADCclk,conv,done,data_out);
input clk,enable,I1,I2,I3,ADC_out;
output conv,done,ADC_in,ADCclk;
output [9:0] data_out;
wire tc;
wire [31:0] out;
wire [9:0] data_out1;
///clock division.........
cnt1 counter(out,32'b0,tc,enable,clk,tc,
32'b11111111111111111111111111111111);
ADC m1(ADCclk, enable, I1,I2,I3,conv, done,
ADC_in, ADC_out, data_out1);
assign ADCclk = out[6];
fdc10 m2(data_out1,ADCclk,done,1'b0,data_out);
endmodule

module cnt1(out,data,load,en,clk,tc,lmt);
output [31:0] out;
output reg tc;
input [31:0] data;
input load, en, clk;
reg [31:0] out;
parameter reset=0;
input [31:0]lmt;
initial begin out=32'b0;
tc=0; end
always @(negedge clk)
if (reset) begin
  out <= 32'b0 ;
end else if (load) begin
  out <= data;
end else if (en)
  out <= out + 32'b00000000000000000000000000000001;
else out <= out;
always @(posedge clk)
if (out ==lmt)
tc<=1;
else tc<=0;
endmodule

module ADC(clk, enable, I1,I2,I3,conv, done,
ADC_in, ADC_out, data_out);
    // input and outputs
    input   I1,I2,I3,clk, enable, ADC_out;
    output reg ADC_in;
    output reg conv,done;
    output reg [9:0] data_out;
    // internal variables
    reg [2:0]   Cs = 0;
    reg [4:0]   count = 0;
    reg [4:0]   bit_pos = 0;
    always @(negedge clk) begin// when enable is on
```

```
if (enable == 1) begin
case (Cs)
0:  begin
// initial
conv     <= 0;
done     <= 0;
ADC_in = 1;
Cs <= Cs + 3'b001;
count <= count + 5'b00001; //1
data_out <= 0;
end
1:  begin
// initial
conv     <= 0;
done     <= 0;
ADC_in = 1;
Cs <= Cs + 3'b001;
count <= count + 5'b00001; //2
end
2:  begin
// initial
conv     <= 0;
done     <= 0;
ADC_in = I1;
Cs <= Cs + 3'b001;
count <= count + 5'b00001; //3
end
3:  begin
// initial
conv     <= 0;
done     <= 0;
ADC_in = I2;
Cs <= Cs + 3'b001;
count <= count + 5'b00001; //4
end
4:  begin
// initial
conv     <= 0;
done     <= 0;
ADC_in = I3;
Cs <= Cs + 3'b001;
count <= count + 5'b00001; //5
end

5:  begin

if (count < 8) begin
conv     <= 0;
done     <= 0;
ADC_in = 0;
Cs <= 5;
count <= count + 5'b00001; end  //6
else begin
```

```
conv     <= 0;
done     <= 0;
ADC_in = 0;
Cs <= 6;
bit_pos = 5'b10001 - count;
data_out[bit_pos] <= ADC_out;
count <= count + 5'b00001;
end
end
6: begin
if (count < 17) begin
conv     <= 0;
done     <= 0;
ADC_in = 0;
Cs <= 6;
bit_pos = 5'b10001 - count;
data_out[bit_pos] <= ADC_out;
count <= count + 5'b00001;end
else begin
conv     <= 0;
done     <= 1;
ADC_in = 0;
Cs <= Cs + 3'b001;
bit_pos = 5'b10001 - count;
data_out[bit_pos] <= ADC_out;
count <= count + 5'b00001;
end
end
7: begin
if (count == 18) begin
conv     <= 1;
done     <= 0;
ADC_in = 0;
Cs <= 0;
conv     <= 1;
count <= 0;
end
end
default: begin
Cs   <= 0;
conv  <= 0;
count    <= 0;
done     <= 0;
end
endcase
end else begin
// reset
conv      <= 0;
done         <= 0;
Cs       <= 0;
count     <= 0;
data_out     <= 0;
end
```

```
    end
endmodule

module fdc10(a,clk,en,reset,y);
     input [9:0] a;
     input clk,en,reset;
     output [9:0] y;
fdce1 d1(y[0],clk,en,reset,a[0]);
fdce1 d2(y[1],clk,en,reset,a[1]);
fdce1 d3(y[2],clk,en,reset,a[2]);
fdce1 d4(y[3],clk,en,reset,a[3]);
fdce1 d5(y[4],clk,en,reset,a[4]);
fdce1 d6(y[5],clk,en,reset,a[5]);
fdce1 d7(y[6],clk,en,reset,a[6]);
fdce1 d8(y[7],clk,en,reset,a[7]);
fdce1 d9(y[8],clk,en,reset,a[8]);
fdce1 d10(y[9],clk,en,reset,a[9]);
endmodule

module fdce1(q,clk,ce,reset,d);
    input d,clk,ce,reset;
    output reg q;
initial begin q=0; end
always @ (negedge (clk)) begin
if (reset)
  q <= 1'b0;
 else if (ce)
  q <= d;
 else
 q<= q ;
end
endmodule

# clock
# Choose IOSTANDARD = LVCMOS33;
# Choose SLEW = SLOW and DRIVE = 16;
NET "clk" LOC = E3 | IOSTANDARD = LVCMOS33 ;
NET "clk" PERIOD = 10.0ns HIGH 50%;
# Selection of ADC Channels
NET "I1"    LOC = R15   ;
NET "I2"    LOC = M13   ;
NET "I3"    LOC = L16   ;
# other SPI interface control
NET "conv"  LOC = A14   ;
NET "ADC_in"    LOC = A16   ;
NET "ADC_out"   LOC = B17   ;
NET "ADCclk"    LOC = A18   ;
# other control signals
NET "enable"    LOC = J15   ;
NET "done"  LOC = V11   ;
# Output data to the LEDs
NET "data_out[0]"   LOC = H17   ;
NET "data_out[1]"   LOC = K15   ;
```

```
NET "data_out[2]"   LOC = J13   ;
NET "data_out[3]"   LOC = N14   ;
NET "data_out[4]"   LOC = R18   ;
NET "data_out[5]"   LOC = V17   ;
NET "data_out[6]"   LOC = U17   ;
NET "data_out[7]"   LOC = U16   ;
NET "data_out[8]"   LOC = V16   ;
NET "data_out[9]"   LOC = T15   ;
```

16.9 Interfacing DAC Chips with FPGA Using SPI Protocol

Like ADC chips, it is also important to know how to interface Digital to Analog Converter (DAC) ICs with the FPGA Board. It is also required to convert the digital outputs from the FPGA to analog domain. For example in an FPGA-based control system, FPGA-based PID controller controls the control valves to regulate process parameters. In this section, we will talk about how to interface a DAC IC using Verilog HDL.

The MCP4922 [50] is a dual 12-bit buffered voltage output DAC IC. It operates from a single 2.7–5.5 V supply with SPI compatible serial peripheral interface. The IC supports SPI Interface with 20 MHz Clock. The pin diagram of the DAC IC is shown in Fig. 16.69. The IC has two output pins (V_{OUTA} and V_{OUTB}) for two DAC channels A and B. Thus, has two reference points (V_{REFA} and V_{REFB}). The $\overline{LDAC}$ pins enables the synchronous update of DAC outputs. Both channels or any of the channels can be shut down by applying active low signal on $\overline{SHDN}$ pin. The major controlling pins are $\overline{CS}$, *SCK* and *SDI*. The active low signal $\overline{CS}$ enables the DAC IC, *SCK* pin provides the sampling clock and *SDI* pin provides the serial input signal from the FPGA. The functional diagram is shown in Fig. 16.70.

The equation which controls the digital to analog conversion is

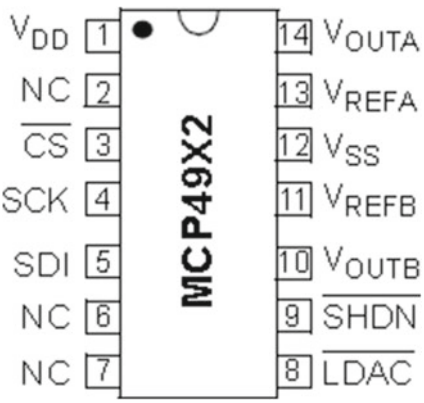


Fig. 16.69 The pin diagram of the DAC IC chip

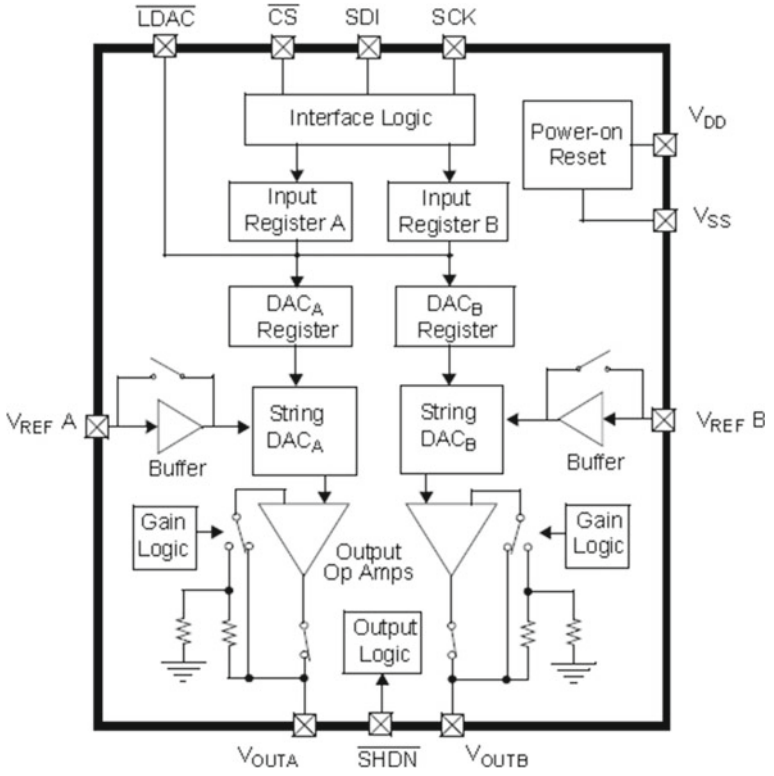


Fig. 16.70 The functional diagram of the DAC chip

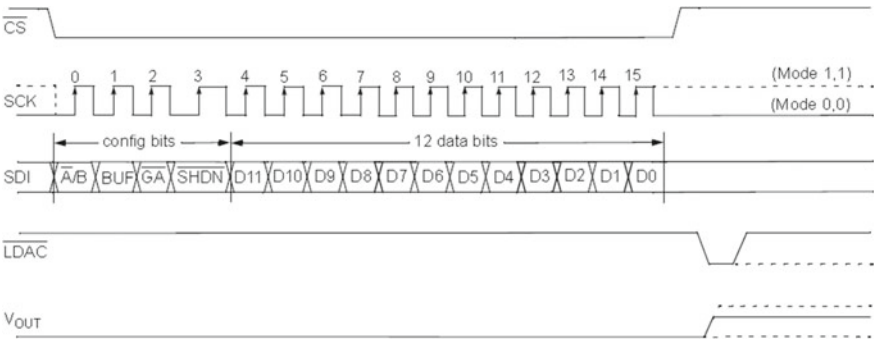


Fig. 16.71 Timing diagram for the DAC chip

$$V_{OUT} = \frac{V_{REF} \times D_n}{2^n} \times G \tag{16.37}$$

Here, V_{OUT} is the output voltage, V_{REF} is the external voltage reference, D_n is the n-bit digital word and G is the gain. The timing diagram for the DAC IC is shown in Fig. 16.71. The serial control word is divided into two sections, viz., 12-bits for digital word and 4-bits reserved for configuration bits. The 15th bit is to select the DAC and if it is 1, then channel B is selected. If the 14th bit is high, then the output is buffered. The 13th bit is gain selection bit and it sets to 0 to have unit gain. If the 12th bit is 1, then only the IC is in active mode. The interfacing is done by writing a Verilog code. The Verilog code is shown below.

```
module DAC(
    clk, enable, done, data,
    SPI_MOSI, DAC_CS, SPI_SCK, DAC_CLR, SPI_MISO);
    // input and outputs
    input   clk, enable, SPI_MISO;
    output  done;// goes high for one clock cycle
    input   [11:0]  data;// desired DAC value
    output  SPI_MOSI, DAC_CS, SPI_SCK, DAC_CLR;
    wire    clk,SPI_MISO;
    reg done;
    wire    [11:0]  data;
    reg     SPI_MOSI, DAC_CS, SPI_SCK, DAC_CLR;
    // internal variables
    reg [2:0]   Cs = 0;
    reg [15:0]  send;
    reg [4:0]   bit_pos = 16;
    always @(posedge clk) begin
        if (enable == 1) begin
        case (Cs)
        0:  begin
        // initial
        DAC_CS  <= 1;
        SPI_MOSI <= 0;
        SPI_SCK <= 0;
        DAC_CLR <= 1;
        done        <= 0;
        Cs <= Cs + 3'b001;
        end
        1: begin
        // set data to be sent
        send <= { 4'b0011,data};
        // set for next
        bit_pos <= 16;
        Cs <= Cs + 3'b001;
        end
        2: begin
        // start sending
        DAC_CS  <= 0;
        // lower clock
        SPI_SCK <= 0;
```

```
        // set data pin
        SPI_MOSI <= send[bit_pos-1];
        bit_pos <= bit_pos - 5'b00001;
        Cs <= Cs + 3'b001;
        end
        3: begin
        // rise spi clock
        if (bit_pos > 0) begin
        SPI_SCK <= 1;
        Cs <= 2;
        end else begin
        SPI_SCK <= 1;
        Cs <= Cs + 3'b001;
        end
        end
        4: begin
        SPI_SCK <= 0;
        Cs <= Cs + 3'b001;
        end
        5: begin
        DAC_CS <= 1;
        Cs <= Cs + 3'b001;
        end
        6: begin
        done <= 1;  // send done signal
        Cs <= Cs + 3'b001;
        end
        7: begin
        done <= 0;  // go back to loop
        Cs <= 1;
        end
        default: begin
        DAC_CS  <= 1;
        SPI_MOSI <= 0;
        SPI_SCK <= 0;
        DAC_CLR <= 1;
        end
        endcase
        end else begin
            // reset
            DAC_CS  <= 1;
            SPI_MOSI <= 0;
            SPI_SCK <= 0;
            DAC_CLR <= 1;
            done        <= 0;
            Cs      <= 0;
            bit_pos     <= 16;
        end
    end
endmodule
```

16.10 Interfacing External Devices with FPGA Using UART

In the previous section, we have seen that how any IC can be interfaced with FPGA device through SPI protocol. Universal Asynchronous Receiver/Transmitter (UART) is another means of establishing an interface between FPGA and IC or between FPGA and a microcontroller. The UART interfacing protocol doesn't need clock information. Many ICs follow this protocol for interfacing. Interfacing between an FPGA and a microcontroller (MCU) is shown below in Fig. 16.72.

UART is a serial communication where bit-by-bit transmission takes place. Here, an MCU doesn't transmit clock signal but has an internal clock to define the time duration of the bits. MCU sends the bits in the form of a control word or packet. The user understands the control word or packet and forms words from it.

The UART frame is shown in Fig. 16.73 for 8-bits. The overall length of the frame is 10-bits. First, there is a start bit which is a overhead bit as it doesn't carry any information. It indicates the transition from the idle state to active state. The data line is active high in idle state and thus the start bit is active low. Then the 8-bits are sent serially from LSB bit. The last bit in the frame is stop bit which is also a overhead bit. The stop bit is active high and indicates that data is sent and can go to the idle state.

The rate at which the bits are transmitted is called Baud rate and it is measured in terms of bits per second. If the Baud rate is 9600, then it can be said that a bit is transmitted in $1/9600 = 104.2\,\mu$s. Setting of proper Baud rate is very important in order to establish a successful UART interface

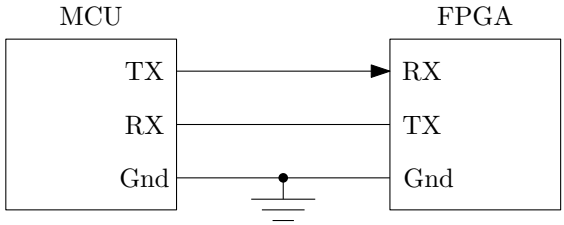


Fig. 16.72 Interfacing an FPGA with MCU

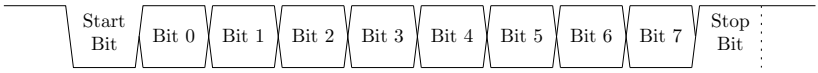


Fig. 16.73 UART frame for 8-bits

```
module UART_top(
    input clk,
    input RxD,
    output TxD,
     output [7:0]GPin,
    output [7:0]GPout
);

reg [7:0] GPout;// general purpose outputs
reg [7:0] GPin;  // general purpose inputs
wire RxD_data_ready;
wire Tx_Done,Tx_Active;
wire [7:0] RxD_data;
wire [7:0]GPout1;
uart_rx RX(clk, RxD, RxD_data_ready, RxD_data );
always @(posedge clk)
if(RxD_data_ready)
GPin <= RxD_data;
else
GPin<=GPin;

uart_tx TX(clk, RxD_data_ready, GPout,
        Tx_Active, TxD, Tx_Done);
always @(negedge clk)
if(Tx_Done)
GPout<=GPin;
else
GPout<=GPin;
endmodule

`timescale 1ns / 1ps
// Transmitter transmits 8 bits of serial data,
// one start bit, one stop bit and no parity bit.
// When transmit is complete o_Tx_done will be high
// Set Parameter CLKS_PER_BIT as follows:
// CLKS_PER_BIT=(Frequency of i_Clock)/(Frequency of UART)
// Example: 10 MHz Clock, 115200 baud UART
// (10000000)/(115200) = 87
module uart_tx
  #(parameter CLKS_PER_BIT=10416)
  (
   input        i_Clock,
   input        i_Tx_DV,
   input [7:0] i_Tx_Byte,
   output       o_Tx_Active,
   output reg   o_Tx_Serial,
   output       o_Tx_Done
   );

  parameter s_IDLE          = 3'b000;
  parameter s_TX_START_BIT  = 3'b001;
  parameter s_TX_DATA_BITS  = 3'b010;
  parameter s_TX_STOP_BIT   = 3'b011;
  parameter s_CLEANUP       = 3'b100;
```

```
  reg [2:0]     r_SM_Main      = 0;
  reg [13:0]     r_Clock_Count = 0;
  reg [2:0]     r_Bit_Index    = 0;
  reg [7:0]     r_Tx_Data      = 0;
  reg           r_Tx_Done      = 0;
  reg           r_Tx_Active    = 0;

  always @(posedge i_Clock)
    begin

case (r_SM_Main)
  s_IDLE :    ////State 0
    begin
  o_Tx_Serial    <= 1'b1; // Drive Line High for Idle
   r_Tx_Done      <= 1'b0;
   r_Clock_Count <= 0;
   r_Bit_Index    <= 0;

 if (i_Tx_DV == 1'b1)
 begin
   r_Tx_Active <= 1'b1;
//r_Tx_Data    <= i_Tx_Byte;
   r_SM_Main    <= s_TX_START_BIT;
end
else
   r_SM_Main <= s_IDLE;
end // case: s_IDLE

// Send out Start Bit. Start bit = 0
s_TX_START_BIT :     ////State 1
begin
    o_Tx_Serial <= 1'b0;

// Wait CLKS_PER_BIT-1 clock cycles for start bit to finish
if (r_Clock_Count < CLKS_PER_BIT-1)
   begin
if (r_Clock_Count == CLKS_PER_BIT/2)
    r_Tx_Data    <= i_Tx_Byte;
    r_Clock_Count <= r_Clock_Count + 1;
   r_SM_Main      <= s_TX_START_BIT;
end
else
begin
    r_Clock_Count <= 0;
    r_SM_Main      <= s_TX_DATA_BITS;
 end
end // case: s_TX_START_BIT

// Wait CLKS_PER_BIT-1 clock cycles for data bits to finish
s_TX_DATA_BITS :    ////State 2
begin
   o_Tx_Serial <= r_Tx_Data[r_Bit_Index];
```

```
if (r_Clock_Count < CLKS_PER_BIT-1)
begin
    r_Clock_Count <= r_Clock_Count + 1;
   r_SM_Main     <= s_TX_DATA_BITS;
end
else
begin
   r_Clock_Count <= 0;

// Check if we have sent out all bits
if (r_Bit_Index < 7)
begin
   r_Bit_Index <= r_Bit_Index + 1;
   r_SM_Main   <= s_TX_DATA_BITS;
end
else
begin
   r_Bit_Index <= 0;
   r_SM_Main   <= s_TX_STOP_BIT;
end
end
end // case: s_TX_DATA_BITS

// Send out Stop bit.  Stop bit = 1
    s_TX_STOP_BIT : ////State 3
begin
    o_Tx_Serial <= 1'b1;

// Wait CLKS_PER_BIT-1 clock cycles for Stop bit to finish
if (r_Clock_Count < CLKS_PER_BIT-1)
begin
    r_Clock_Count <= r_Clock_Count + 1;
   r_SM_Main     <= s_TX_STOP_BIT;
end
else
begin
   r_Tx_Done     <= 1'b1;
   r_Clock_Count <= 0;
   r_SM_Main     <= s_CLEANUP;
   r_Tx_Active   <= 1'b0;
end
end // case: s_Tx_STOP_BIT

// Stay here 1 clock
    s_CLEANUP :   ////State 4
begin
   r_Tx_Done <= 1'b1;
   r_SM_Main <= s_IDLE;
end

default :
   r_SM_Main <= s_IDLE;
```

```
endcase
end

assign o_Tx_Active = r_Tx_Active;
assign o_Tx_Done   = r_Tx_Done;
endmodule

`timescale 1ns / 1ps
module uart_rx
  #(parameter CLKS_PER_BIT=10416)
  (
   input        i_Clock,
   input        i_Rx_Serial,
   output       o_Rx_DV,
   output [7:0] o_Rx_Byte
   );

  parameter s_IDLE         = 3'b000;
  parameter s_RX_START_BIT = 3'b001;
  parameter s_RX_DATA_BITS = 3'b010;
  parameter s_RX_STOP_BIT  = 3'b011;
  parameter s_CLEANUP      = 3'b100;

  reg            r_Rx_Data_R = 1'b1;
  reg            r_Rx_Data   = 1'b1;

  reg [13:0]      r_Clock_Count = 0;
  reg [2:0]      r_Bit_Index   = 0; //8 bits total
  reg [7:0]      r_Rx_Byte     = 0;
  reg            r_Rx_DV       = 0;
  reg [2:0]      r_SM_Main     = 0;

  // Purpose: Double-register the incoming data.
  // This allows it to be used in the UART RX Clock Domain.
  // (It removes problems caused by metastability)
  always @(posedge i_Clock)
    begin
      r_Rx_Data_R <= i_Rx_Serial;
      r_Rx_Data   <= r_Rx_Data_R;
    end

  // Purpose: Control RX state machine
  always @(posedge i_Clock)
    begin

   case (r_SM_Main)
   s_IDLE :
   begin
   r_Rx_DV       <= 1'b0;
   r_Clock_Count <= 0;
   r_Bit_Index   <= 0;

   if (r_Rx_Data == 1'b0)// Start bit detected
    r_SM_Main <= s_RX_START_BIT;
```

```
    else
   r_SM_Main <= s_IDLE;
   end

// Check middle of start bit to make sure it's still low
s_RX_START_BIT :
          begin
if (r_Clock_Count>=(CLKS_PER_BIT-1)/2)
 begin
if (r_Rx_Data == 1'b0)
 begin
if(r_Clock_Count==CLKS_PER_BIT-2)
begin
r_SM_Main      <= s_RX_DATA_BITS;
r_Clock_Count <= 0;
end
else
r_Clock_Count <= r_Clock_Count + 1;
end
else
r_SM_Main <= s_IDLE;
end
else
begin
r_Clock_Count <= r_Clock_Count + 1;
r_SM_Main      <= s_RX_START_BIT;
end
end // case: s_RX_START_BIT


// Wait CLKS_PER_BIT-1 clock cycles to sample serial data
 s_RX_DATA_BITS :
begin
if (r_Clock_Count < CLKS_PER_BIT-1)
begin
r_Clock_Count <= r_Clock_Count + 1;
r_SM_Main <= s_RX_DATA_BITS;
end
else
begin
r_Clock_Count <= 0;
r_Rx_Byte[r_Bit_Index] <= r_Rx_Data;

// Check if we have received all bits
if (r_Bit_Index <7)
begin
r_Bit_Index <= r_Bit_Index + 1;
r_SM_Main   <= s_RX_DATA_BITS;
end
else
begin
r_Bit_Index <= 0;
r_SM_Main   <= s_RX_STOP_BIT;
```

```
end
end
end // case: s_RX_DATA_BITS

// Receive Stop bit.  Stop bit = 1
        s_RX_STOP_BIT :
          begin
// Wait CLKS_PER_BIT-1 clock cycles for Stop bit to finish
 if (r_Clock_Count < CLKS_PER_BIT-1)
 begin
r_Clock_Count <= r_Clock_Count + 1;
r_SM_Main     <= s_RX_STOP_BIT;
 end
else
 begin
 r_Rx_DV       <= 1'b1;
r_Clock_Count <= 0;
 r_SM_Main     <= s_CLEANUP;
 end
 end // case: s_RX_STOP_BIT

 // Stay here 1 clock
s_CLEANUP :
   begin
    r_SM_Main <= s_IDLE;
      r_Rx_DV   <= 1'b0;
      end

 default :
 r_SM_Main <= s_IDLE;

 endcase
 end

  assign o_Rx_DV   = r_Rx_DV;
  assign o_Rx_Byte = r_Rx_Byte;
endmodule // uart_rx
```

16.11 Conclusion

In this chapter, we have discussed various examples of digital system designs. The architectures are explained in detail to give readers a clear view that how to start designing their own digital system. The systems are chosen in such a way to cover all the important areas. These architectures may not be directly needed in a project but the design concepts may be very useful.

First, we have discussed the design of digital filters. In designing a filter, the first thing we have to know that which type of filter we are going to use. It may be an FIR or an IIR filter. In our earlier discussion, it is clear that FIR filters are

the general choice for high-frequency applications. IIR filters can achieve better response with less filter order than the FIR filters. But IIR filters are recursive and thus pipelining is difficult. Pipeline registers can be inserted in IIR filters using look-ahead techniques but becomes costly. Secondly, we have to be careful about the filter topology. Transpose structures are always better than the direct forms. Topology must be chosen in such a way to reduce power consumption and also to increase SNR performance.

We have discussed the implementation of a machine learning algorithm which is K-means algorithm. This will give the readers an idea that how to implement an algorithm on FPGA and analyze its performance. FPGA implementation performance is measured in terms of hardware complexity (Resource utilization), timing complexity (Developing the timing expressions) and dynamic power consumption.

In implementing signal processing algorithms, matrix multiplication is very important. Thus, we have also discussed some methods of matrix multiplication. Three methods are discussed here which are scalar–vector multiplication, vector–vector multiplication and systolic array multiplication. Each technique has its own advantage and must be used based on application needs.

Some of the sorting architectures are also discussed in this chapter. Designers may use parallel or serial architecture based on their requirement. Along with the sorting architectures, efficient computation of median of nine elements is also shown here. This median computation block is very useful in designing median filter to remove noises from an image. Here, FPGA implementation of a spatial median filter is also given.

At last, we have discussed how to interface ADC and DAC chips to an FPGA device kit. Interfacing of ADC or DAC chips is very important for real-time signal acquisition or demonstration of FPGA performance, respectively. This interfacing is achieved through an FSM design style written using Verilog HDL. These two interfacing examples will be very useful in interfacing any SPI protocol-based ICs to an FPGA kit. In addition to SPI-based interfacing, interfacing using the UART protocol is also explained here.

Chapter 17
Basics of System Verilog

17.1 Introduction

In the Chap. 3, we have discussed the fundamentals of Verilog HDL which is the most used language for description of hardware. In the case of FPGA implementations Verilog was enough as timing verification is mostly handled by the EDA tools. But for ASIC designs designers were using other tools to perform STA and other verification tasks. The usage of different tools was becoming a headache for the engineers.

Gradually, a new language is developed, called system Verilog, which is becoming very popular nowadays for both RTL design and verification. The feature set of system Verilog can be divided into two distinct roles:

- System Verilog for RTL design is an extension of Verilog-2005. All features of that language are available in system Verilog. Therefore, Verilog is a subset of system Verilog.
- System Verilog for verification uses extensive object-oriented programming techniques and is more closely related to Java than Verilog. These constructs are generally not synthesizable.

It is not possible to discuss all the concepts of system Verilog in a single chapter. In this chapter, fundamentals of system Verilog are discussed. The concepts which are mostly require for RTL design are discussed here.

17.2 Language Elements

17.2.1 Logic Literal Values

Unlike Verilog, system Verilog supports automatic expansion of the bits. Following example shows how unsized single bit value can be used for expansion

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_17

1. ‘0 : fill all bits with 0.
2. ‘1 : fill all bits with 1.
3. ‘x : fill all bits with x.
4. ‘z : fill all bits with z.

In the following example the net x can be filled with all zeros.

```
wire [7:0] x;
assign x = ‘0;
```

17.2.2 Basic Data Types

All the data types of Verilog are supported in system Verilog. System Verilog additionally includes some extra data types to give flexibility to the designers. All the data types of Verilog and system Verilog are shown below.

- 2-state type.
  - **longint** : 64-bit signed integer.
  - **shortint** : 32-bit signed integer.
  - **int** : 16-bit signed integer.
  - **bit**: 1-bit.
  - **byte**: 8-bit signed integer.
- 4-state type.
  - **logic** : 1-bit.
  - **reg** : 1-bit //Already defined in Verilog
  - **integer** : 32-bit signed integer //Already defined in Verilog
  - **time** : 64-bit unsigned integer //Already defined in Verilog
- Other types
  - **real** : 64-bit double precision floating point. It is already defined in Verilog.
  - **realtime** : This is of type real and is used to store time. It is also defined in Verilog.
  - **shortreal** : 32-bit single precision floating point.
  - **void** : This type indicates no storage. It is used in functions to return nothing.
  - **chandle** : This type is used to store pointers.

The difference between 2-state and 4-state variables is that a 2-state variable can take either 0 or 1 but a 4-state variable can take any values from the set $\{0, 1, x, z\}$. Type **logic** and type **reg** are equivalent and thus can be used interchangeably.

In the system Verilog kind (net or variable) and type of an object are separately mentioned. Some examples are shown below.

```
wire logic ctrl,add_sub;//1-bit logic type net.
var logic [7:0] bus; //8-bit 4-state variable
```

If type is not mentioned, then it is assumed that it is of variable type. Also, if kind of an object is not mentioned, then it is assumed that it is of logic type.

```
wire ctrl,add_sub;//by default of type logic.
logic [7:0] bus; //by default of kind variable.
```

17.2.3 User Defined Data-Types

System Verilog offers user-defined data types for comfortably handling the objects. Not that no new type of data type can be used. User can choose any data type which are already defined. An example is shown below

```
typedef int my_int;
```

A new data type *my_int* is defined which is equivalent to data type **int**. Thus, the followings are equivalent

```
int x;
my_int x;
```

User-defined data types are useful in creating parametrized modules where data type is not explicitly defined in the sub-modules.

17.2.4 Enumeration Data Type

In **enumeration** data type, an object can get any value from a set of constant values. Some examples are shown below:

```
enum {s0,s1,s2,s3} fsm_states;
enum {add, sub} ctrl;
```

In the above example, variable *fsm_states* can take any values from the four values and the variable *ctrl* can take value of either *add* or *sub*. Each literal in an **enumeration** data type is of **int** data type and has a specific value associated with it. For example, *add* has a value of 0 and *sub* has a value of 1. The integer values are assigned from left to right in increasing order. For example, *s*0 gets a value of 0 and *s*3 gets a value of 3. No literal can have same value in an enumeration data type. Specific values also can be specified by overriding the original values. For example,

```
enum {s0=0,s1=1,s2=2,s3=4} fsm_states;
enum {add=2, sub} ctrl;
```

It is not necessary to give values for all the literals. In the above example, literal *sub* automatically gets a value of 3.

In the above examples, type of the objects is not defined. Type of an enumeration object can be defined using **typedef**. For example,

```
enum logic [1:0] {s0,s1,s2,s3} fsm_states;
```

Here, all the literals are of type **logic**. Default type of the literals is **int**.

```
typedef enum
{s0=0,s1=1,s2=2,s3=4} t_func;
var t_func alu_func;
```

Here, our main object is alu_func which is of type t_func (user-defined data type). Again t_func is of **enumeration** data type and thus alu_func is of **enumeration** data type and of kind variable.

System Verilog offers some shorthand notations to specify enumeration data-type literals. The following expression:

```
typedef enum
{s[3],fn[7:4]} t_func;
var t_func alu_func;
```

can be interpreted as

```
typedef enum
{s3,s2,s0,fn7,fn6,fn5,fn4} t_func;
var t_func alu_func;
```

17.2.5 Arrays

System Verilog adds more features to handle the arrays. In an array, data can be stored and an array can be one-dimensional or two-dimensional. There are two types of arrays, packed and unpacked. unpacked array means only locations are mentioned but, at each location, only single-bit data can be stored. Packed array means that the width of the data is more than a single bit. Below some examples are shown

```
reg [3:0] count ///packed array
reg x [3:0]  ///one dimensional unpacked array
reg [3:0] mem [7:0]  ///8 locations and each having 4-bit data
reg [3:0] rom [7:0][3:0]  ///Total 32 locations and
//each having 4 bit data.
reg [3:0][7:0] y [7:0]  ///Total 8 locations and
//each having 32 bit data.

//Data width is partitioned into two sections.
typedef bit x [7:0];
x [3:0] y; /// not allowed.
//Can not mix-up packed and unpacked array.

byte [31:0] prg; //not allowed
int [7:0] z; //not allowed.

//How to access a data from a packed array...
data [3][3:2];//First is the unpacked index and
//second is the part select from the packed range.
```

```
//the range can be implicitly specified.

reg [3:0] dpram [4][256]; // is equivalent to
reg [3:0] dpram [3:0][255:0]

//but data width cannot be implicitly specified for packed arrays
    .
reg [3] q;  // is not equal to
reg [3:0] q;

///Consider an array having both two-dimensional
//packed and unpacked range.
reg [3:0][7:0] xy [31:0][255:0];
///[7:0] dimension varies more rapidly than the [3:0] dimension
//and [255:0] varies more rapidly than the [31:0] dimension.

//Thus in
xy [3][250][2][6];
//Here, 3 is from the [31:0] range, 250 is from the [255:0],
//2 is from the [3:0] range and 6 is from the [7:0] range.
```

The following operations can be performed on arrays:

- Read and write an array $array1 = array2$;
- Read and write slice an array $array1[i : j] = array2[m : n]$;
- Read and write a variable slice an array $array1[x+ : i] = array2[y+ : i]$;
- Read and write element of an array $array1[i] = t; t = array1[j]$;
- Use equality operator on an array or on a slice $array1 == array2$; $array1[i : j] = array2[m : n]$;

Array Literals

Array literals are values of an array. Such values are specified in the following manner.

```
logic [3:0] count = '{1,0,1,0};
bit mdat [1:0][2:0] = '{'{0,0,1},'{1,0,1}};
int hit_tbl [0:1][1:5] = '{2{'{5,4,5,6,3}}};
int corr [0:7] = '{0:4,2:5, default:2};
///0th element gets 4, 3rd element gets 5
//and rest of the elements gets 2 value.
```

17.2.6 Dynamic Arrays

Dynamic array is an array whose size is specified only at the runtime. Size of the dynamic arrays can be varied. For example,

```
integer mem [];
int array [];
```

The storage for dynamic array is not exist until it is created at runtime. The function **new** can be used to allocate storage.

```
mem = new[50]; //allocates 50 elements to the array.
```

The elements of the dynamic array are by default initialized to the default value of the type of array. Another array also can be used to initialize the dynamic array.

```
int init_array[19:0];
...
mem = new[50](init_array);//Initialize first 20
//values using the array init_array.
```

A dynamic array can be of any size and can have any number of elements. It also can be multidimensional. Size of a dynamic array can be known by method **size** in the following way:

```
j = mem.size;//it returns 50
```

The elements of the dynamic array can also be deleted in the following way by method **delete**:

```
mem.delete;//mem has 0 element after execution of this statement.
```

17.2.7 Associative Array

Associative arrays are like a look-up table. The index type is used as look-up key. The array index can be of any arbitrary type. Here is a variable declared as an associative array.

```
typedef enum (MON, TUE, WED, THU, FRI) weekdays;
bit [3:0] lookup [weekdays];
```

Here, an associative array of 4-bit is created and index *weekdays* is enumeration type. So, the element from the look-up array is accessed based on the index. Storage for associative arrays is created as when required.

17.2.8 Queues

Queue is a list of variable sizes where ordered elements of same type can be stored. An example of a queue is

```
byte q_a [$];
```

Queue *q_a* stores elements of byte data type. A queue can represent first in first out (FIFO), last in last out (LILO) or last in first out (LIFO). More examples of queue are

int q_data [$]; //queue of integer
bit [3:0] q_b [$:31]; //31 is the max size of the queue.

A queue is a one-dimensional array that can grow or shrink dynamically. Thus, queue can be used with indexing, slicing, concatenation and equality operators. Following are the operations on queue:

```
int q_c[$] = '{2,3,4};
string code[$] = {"AM01","Am02"};
q_send[0] = 9;
q_send = {q_send,2};
q_rec = q_send;
q_rec = {};
temp = q_send[0] + q_send[$];
q_rec = {q_rec[0:pos-1],m,q_rec[pos:$]};
```

The followings are the methods for queues:

- queue.size : returns the number of elements.
- queue.insert(i,e) : inserts element e at position i.
- queue.delete(i) : deletes element at ith position.
- e= queue.pop_front() : gets element in front and removes it from queue.
- e = queue.pop_back : gets element from back and removes it from queue.
- queue.push_front : pushes element at the front of the queue.
- queue.push_back : pushes element at the end of the queue.

17.2.9 Events

A variable can be declared to be of **event** type. A process can wait for

```
event clock_evnt;
event a1,a2;
always @(clock_evnt)
@(clock_evnt);//wait for an event.
-> clock_evnt;//create an event on clock_evnt.
```

17.2.10 String Methods

String data type store string values. Variables of type string are dynamic in their length.

```
string my_name;
string his_name = "ramesh";
string empty_string = "";//This is an empty string.
```

A single character of a string is of byte type and indexes are from 0 to $(N-1)$. So $his\_name[0]$ is r and $his_name[3]$ is e. Characters from a string can be accessed just like an array.

```
byte tb = his_name[0];
```

To convert integer and bit-vectors to a string type, use a type cast.

```
string str = string'(10'b11_0011_0001);
//Automatically padded with zero to make multiple of 8.
string str_int = string'(52);
```

The followings are the string operations:

- str_int == str_bit : Compare equality.
- str_int != str_bit : Compare inequality.
- str_int > str_bit : Compare strings.
- {str_int,str_bit} : Concatenation of strings.
- {num_rep{str_int}} : Replication (*num_rep* can be a variable).
- {{3{str_int}},str_bit}} : Concatenation and replication.
- str_int[i] : Index, returns a byte.

Events can be assigned to other events. Like $a1 = a2$, any event that occurs on $a1$ will appear on $a2$. An event also can have null values, so that no event can occur. Such as

17.3 Composite Data Types

17.3.1 Structures

Collection of elements of different data types can be written in a single data type called structures. Structures help organizing the data used in similar purpose. An example of a structure is

```
struct {
int x;
bit sign;
logic [3:0] op_code;
} my_structure;
```

Here, *my_structure* is of structure data type and it has three elements of three data types. Elements that comprise the structure are called members. Members of structure are accessed by the following way:

```
y = structure_name.member_name
```

Members can be accessed individually and can be assigned.

```
y = my_structure.x
my_structure.sign = 1;
```

This way members of structure can either be used in procedural or continuous assignment. User-defined type definition also can be used with structure type. This is shown in the following example:

```
typedef struct {
logic [3:0] s;
logic [7:0] a;
logic [7:0] b = 8'h0d;//initialization is also possible.
} struct_logic;
struct_logic int_logic;
```

In the above example, *int_logic* is of type structure. Structures also can be nested like in the following example:

```
typedef struct {
int x;
int y;
} struct1;

typedef struct {
struct1 a;
struct1 b;
} struct2;
```

In the second structure, members are of another structure which is *struct*1. Members of nested structure can be accessed as

```
struct2.a.x ///Accessing member of struct1
struct2.b ///Accessing another structure which is b.
```

Usage of structure data type is very similar to that of array data types. Assignment, initialization and replication are possible for members of a structure.

A structure is packed or unpacked just like an array data type. If nothing is mentioned then a structure data type is unpacked. The structures which were discussed above are unpacked type. In this case, all the members are accessed individually and no order has to be maintained. But in the case of packed type, bits of the members are stored in specific order. An example of packed data type is shown below.

```
struct packed{
bit sign;
bit [3:0] exponent;
bit [10:0] mantissa;
} float_hp;
```

The above-packed structure is equivalent to just an unpacked array of the following type:

```
reg [15:0] float_hp_array;
```

Individual bits or part of the vector can be accessed as

```
float_hp [3:0]; //This is the exponent field.
```

The packed structures provide an easy way to divide the vector into different parts. Then part selection becomes easier. Any operation that can be applied to the unpacked array, can also be applied to an unpacked structured data type.

17.3.2 Unions

Unions in system Verilog is another composite data type which is similar to structure data type. Only difference is that in union data data type storage is allocated for only one member and this storage is shared by other members. Storage is allocated for the member which requires the highest storage. An example of union data type is shown below:

```
typedef union{
int x;
logic [7:0] y;
} my_union;
my_union z;
```

In the above example, z is a union data type and it has two members. One member is of integer data type and another is of logic data type. The storage requirement for integer is greater than the member of logic type. Thus, 32-bits are allocated for x and this storage is shared by other members.

The operations, assignments and usage of union and structure data type are same. Like structure data type, union data type can also be packed type. An example of packed data type is shown below

```
typedef union packed{
logic [3:0] x_reg;
logic [7:0] y_reg;
} my_packed_union;
```

In a packed union, the size of the members must be same and packed union does not support real, short real, unpacked arrays, unpacked unions or unpacked structures. Packed unions are like vectors. In the above example, the storage is shared by x_reg and y_reg. First, 4-bits are for x_reg, and last, 8-bits from the MSB side are for y_reg.

Apart from packed and unpacked unions, there is another type of union which is tagged union. A tagged union is the type of union data type where every member is tagged or every member has an extra field associated with them to differentiate. An example is

```
typedef union tagged{
int int_val;
logic [7:0] logic_val;
} my_tagged_union;
my_tagged_union t_union1;
```

Here, each member is associated with a tag. Like, an additional bit is used in storage to differentiate the members. It is due to track which member is modified. Say, bit 0 is for int_val and bit 1 is for $logic\_val$. A member in the tagged union is assigned as

```
t_union1 = tagged int_val 35;
```

Here, 35 is assigned to member int_val of t_union1 variable which is of tagged union data type.

17.3.3 Classes

System Verilog introduces class, similar to structure, which not only includes different types of members but also can include functions or tasks which can operate on the members residing inside a system Verilog class. Members of a class called class properties and functions or tasks are called class methods. An example of a system Verilog class is shown below

class my_class;
bit [2:0] data;
bit write;
bit [3:0] address;
function new (bit [2:0] data = 3'h3, bit [3:0] address=4'h2",
this.data,this.write,this.address);
this.data = data;
this.write = 0;
this.address = address;
endfunction
function display
$**display** ("data = 0x%0h, write = 0x%0b,address = 0x%0h",
this.data, **this**.write, **this**.address);
endfunction
endclass

In the above definition of the class, there are three class properties and two class methods. A class is used for grouping variables and their functionality in a single scope. Here, two new keywords **this** and **new** are used. The keyword **this** is used to refer the current class. Normally used within a class to call its own objects. The keyword **new** after a class method is used to initialize a function or a task. In order to use a class, first the class is instantiated by another object and then the class properties are accessed just like structure members are used.

```
module class_tb;
my_class t_class1, t_class2; //instantiation of the class

initial begin
t_class1 = new(3'h4, 4'h7); //data = 3'h4, address = 4'h2.
t_class1.display;

t_class2 = new(); //The function will initial values
//data = 3'h3 and address = 4'h2.
t_class2.display;
end
endmodule
```

If the functionality of the above class is to be increased without changing it then this can be done using the **extend** keyword. This keyword will extend the features of the previous class. Now previous class *my_class* is base class and functions or objects

are added to the new class using the super keyword. An example is shown below

```
class my_class1 extends my_class;
bit read;
bit enable;
function new();
super.new;
this.enable = 1;
this.read = 0;
endfunction
function display
super.display();
$display("enable = 0x%0b, read = 0x%0b",this.enable, this.read);
endfunction
endclass
```

17.4 Expressions

In system Verilog, many new expressions and mechanisms are added. The newly added features in system Verilog are highlighted here.

17.4.1 Parameters and Constants

A parameter can be specified by the symbol $. This represents unspecified or unbounded constants.

```
parameter N = $; //parameter declaration..
module add #(parameter N = 16)(..); //parameter definition.
  ...
endmodule

add (.N($)).... parameter value assignment
```

In system Verilog, type of a port can also be passed as constants. Such a parameter is called a *typeparameter* as opposed to *valueparameter*.

```
 module modem #(parameter type ptype = logic) (....
ptype modreg;
endmodule

modem #(.ptype(bit),....//usage of type parameter.
```

In system Verilog, the word parameter is optional. One could simply write as

```
module alu #(N = 16, type logic,int i) (....
endmodule
```

System Verilog has the capability to specify constants using the *const* keyword. Const constants are like local parameters. Local parameters get their value at the elaboration time, whereas const constants get their value after elaboration is complete. Some examples are

```
const int data = 9;
const k= 7.4; ///wrong...type must be specified.
const logic path = top.sb.fa; //can take hierarchical path also.
```

17.4.2 Variables

Variables are declared with their type. Some declarations of the variables are shown here.

```
int count;
bit done;
enum int (S0,S1,S2,S3) state;
logic hsize;
```

Some time variables are declared with **var** keyword.

```
var reg x;
var byte y;
var z; //type is not specified. Default type is logic.
var int i = 5; //can take initial values.
```

Variables are used in continuous assignments or procedural assignments. One important point about variables is that only one input can drive a variable. If multi-driver is there, then a net type should be used.

A variable can be declared as static or automatic by the keywords **static** and **automatic**.

```
automatic logic [3:0] lmt;
static int i;
```

By default a variable is of type static. Automatic variables can only be used within the block in which it is defined. But static variables are global and can also be used outside the block.

a variable can be initialized in two ways. The first method is inline initialization and the second method is with the initial statement. An example is

```
integer num = 6; //inline initialization
....
integer num;
initial num = 6;
```

In Verilog HDL, both types of initialization methods are same and it creates a simulation event. But in system Verilog, they are not same. In the latter case, the value is 6 is assigned to variable *num* after the simulation starts at time 0 and thus it causes a simulation event and *num* gets value at time 0. But in the former case, simulation

starts with variable *num* having a value of 6 and thus it does not cause any simulation event.

17.4.3 Operators

Operators of Verilog HDL are same in the system Verilog. The only difference is that they are differently written. In system Verilog, an assignment operator is written as

```
a *= 2; ///System Verilog usage..
a = a * 2 /// Verilog HDL equivalent.
```

This way other assignment operators can be used in the system Verilog. Blocking assignments can be written as

```
b = (a += 1);
```

In the system Verilog following is

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

is equivalent to

```
a = b;
if (a =b)...
```

The increment or decrement operators (bump operators) are just like they are in language C. A variable can be incremented or decremented in the following way:

```
j = j++;//Post-increment operator...assign then increment
// is equivalent to
j = j+1;
///// Similarly, other bump operators are
k = ++k;//Pre-increment operator...increment then assign
j = j--;//Post-decrement operator.
k = --k;//Pre-decrement operator.
```

System Verilog adds two additional comparison operators. These are

```
==?  ///Wildcard equality operator
!=?  ///Wildcard inequality operator
```

These two operators are used to check equality and inequality even if the -hand side expression has x or z values. If the right-hand expression has x and z values, then they are treated as don't care and don't care bits are masked for comparison. But x and z values in the left-hand side expression are not treated as don't cares. Now, if the left-hand side expression has x values then the comparison result is x. For example,

```
a = 4'b1010, b = 4'bx01z;  //then
a ==? b ;  // is 1
a !=? b ;  // is 0
  .....
a = 4'b1x10, b = 4'bx01z;  //then
a ==? b ;  // is x
a !=? b ;  // is x
```

These operators also can compare the expression of different sizes. For these case, zeros are appended to the expression which has less number of bits.

System Verilog introduces two additional logical operators and these are

```
-> //logical implication operator
<-> //logical equivalence operator
```

Examples of these operators are

```
a -> b; // is equal to (!a || b)
a <-> b; // is equal to (a -> b) && (b -> a)
```

17.4.4 Set Membership Operator

System Verilog offers a set membership operator which is defined by the keyword **inside**. This keyword is used to check that if the value of an expression present in a set of values. For example,

```
state inside {0,1,2,3}
```

If the value of state variable matches any value from the set, then the output is 1. This operator can be easily used in a loop as a condition. More examples of *inside* operator are

```
add inside {4'b0x1z}//compares with 4'b0010, 4'b0011, 4'b0110, 4'
    b0111
// x and z values are treated as don't cares.
adt inside {x,y,z}
//Set values are variables
```

17.4.5 Static Cast Operator

In, a design different data types may be defined and we may require to change the type of a variable or net. This type change is done by casting operator (′). Examples of type casting are

```
int'(2.0) //real to integer
i = int'(3.16*0.5)
shortint'(8'hFA)
```

It is possible to change the user defined types also using this casting operator. Similar to type casting, size casting is also possible in the following manner

```
a = 10'(2+5) // stored in 10-bits
32'(15) //integer is getting 32 bits.
```

Similar to type and size casting, sign of an operand also can be changed. This is done in the following way

```
signed'(4+9)
```

The result will be signed in this case. Type casting can also be performed between composite data types. This type of type casting is achieved through bit streaming. Type casting between composite data types can be possible only between two bit-stream types of same size. In this conversion, first the right hand side object is converted into a vector via bit-streams. This stream of bits are assigned to the destination from left to right. An example is shown below

```
typedef struct{
 bit [3:0] x;
 byte y [3:0];
} t_str1; //size 36

typedef struct{
 byte a;
 bit b [27:0];
} t_str2; //size 36

t_str1 t_dr1;
t_str2 t_dr2;

typedef logic [35:0] t_array;
t_array t_arr;

t_dr2 = t_str2'(t_dr1);
t_dr1 = t_str1'(t_dr2);
t_arr = t_array'(t_dr1);
```

17.4.6 Dynamic Casting

Static casting does not report an error if the casting is not right. For example, if a long real data is converted into an integer, then it is possible that it may not be accommodated. Dynamic casting reports an error if the casting is illegal and the object on which the casting is operated is left unchanged. Dynamic casting is like a function and can be used inside any function or task. An example of dynamic casting is

$**cast**(dest_var,source_expression) /// format of dynamic expression..
$ **cast**(x,y*9);
report = $ **cast**(bn, tm*3.9); //report = 1 only conversion is valid.

17.4.7 Type Operator

The **type** operator gives type of an variable or net or expression. This command can be used whenever type is required to know. This operator can be used in the following way:

```
var type(rec) cst;
x = type(y)'(z);
```

In the first example, type of variable *cst* is defined as same type of variable *rec*. In the second example of typecasting, z is converted to x and type of x is same as that is of y.

17.4.8 Concatenation of String Data Type

In the system Verilog, concatenation of string type data type is possible. For example,

```
string test = "geh";
string alph = {"abc","def",test};
```

The output of the following is "abc def geh". Replication of the strings is also possible. Like

```
string str_a ;
str_a = {3{"string"}}; //Output is "string string string"
```

17.4.9 Streaming Operators

The streaming operators are

- >>: This operator causes the data to be streamed out from left to right.
- <<: This operator causes the data to be streamed out from right to left.

These operators, when applied on the right-hand side, perform packing of a bit stream into sequence of bits in a user-specified order. This operator can be applied to either structures, unpacked arrays or class objects of any length. The stream of bits then can be allocated to a destination. If these operators are applied on the left-hand side, then they unpack a bit stream into one or more variables. The format of this operator is

$$stream\_operator\ slice\_size\ list\_of\_expr\_concat$$

The *slice_size* specifies how the bit-stream is broken into slices where each slice has a specific number of bits. If not specified, then it is assumed that *sluce_size* is 1. Examples of streaming operators are

```
byte x = 8'b01100101;
 bit a,b,c,d;
 bit [3:0] e;
y = {<< {x}}; // Bit reverse..y = 1_0_1_0_0_1_1_0;..
//from right to left.
y = {>> 2{x}}; // Same..y = 01_10_01_01;..
///from left to right.
{>> {a,b,c,d}} = z; // /if z = "1011_1110"..
///then a = q, b = 0, c=1,d=1 and e = 1110.
```

17.5 Behavioural Modelling

17.5.1 Procedural Constructs

System Verilog has three additional always statements which are

- **always_comb** statement.
- **always_latch** statement.
- **always_ff** statement.

always_comb Statement

In the Verilog HDL, the statement **always @*** is used to realize combinational circuits. In system Verilog, this statement is modified and a new procedural construct **always_comb** is used to realize only combinational circuits. The major characteristics of this statement are

- Automatically detects the sensitivity list of a design.
- **always_comb** automatically executes once at time zero, whereas **always @*** waits until a change occurs on a signal in the inferred sensitivity list.
- Variables on the left-hand side of assignments within an **always_comb** procedure, including variables from the contents of a called function, cannot be written to by any other processes, whereas **always @*** permits multiple processes to write to the same variable.
- Statements in an **always_comb** cannot include those that block, have blocking timing or event controls, or fork-join statements.
- It generates warning if combinational circuits can not be generated.
- Inputs of functions, called in procedural statements, are considered part of the sensitivity list.

Consider the simple system Verilog code of a half adder. Here, the **always_comb** procedural statement automatically considers inputs a and b in the sensitivity list. But it does not include the variable which is declared locally in the statement in the sensitivity list. Here, sensitivity list is ‘a or b’.

```
module FA(input a,b, output reg s,c);
always_comb
begin
  s <= b ^ a;
  c <= a & b;
end
endmodule
```

Consider another simple realization of 3:1 multiplexer using case statement. Here, the default case is not mentioned and thus there is a possibility that a latch will be inferred. Thus, it will not be a pure combinational circuit and **always_comb** will generate a warning.

```
module MUX(input a,b,c, input [1:0] s, output reg y);
always_comb
case(s) //: ADDER
  2'b00 : y = a;
  2'b01 : y = b;
  2'b10 : y = c;
  //default : y = 0;
endcase
endmodule
```

In the following system Verilog code, a function is defined and called. The sensitivity list for the **always_comb** statement is 'a or b or cin'

```
module Top(input a,b,cin, output reg s);
function logic my_func(input logic cin);
    my_func = a | b | cin;
endfunction
always_comb
s = my_func(cin);
endmodule
```

always_latch Statement

Just like **always_comb** targets to generate combinational circuits, **always_latch** statement targets to generate latch-like circuits. The characteristics of these two statements are same and the only difference is that **always_latch** statements generate warning if latch-like logic is not realized. A simple system Verilog code is shown below:

```
module FA_latch(input a,b,en, output reg s,c);
always_latch
begin : ADDER
  if (en) begin
    s    <= b ^ a;
    c <= a & b;
  end
end
endmodule
```

Here, the sensitivity list for the above design is automatically inferred and it is 'a or b or en'.

always_ff Statement

Like other two procedural statements, **always_ff** is used to realize sequential logic circuits which use flip-flops. The timing control for the flip-flop circuits must be edge-triggered, like positive or negative edge of clock or reset signals. A simple example of system Verilog code using this statement is given below:

```
module DFF_sv(input d,clk,rst, output reg q);
always_ @(posedge clk or posedge rst)
if (rst)
q <= 0;
else
q <= d;
endmodule
```

The above code models a simple D flip-flop using system Verilog.

17.5.2 Loop Statements

System Verilog improves the existing loop statements and also introduces some new loop statements for easy programming. These are

- For loop
- Do-while loop
- If-else loop
- Foreach loop.

For-Loop Statement

In system Verilog, variables can be declared locally in the **for-loop**. These variables are called automatic variables. In addition, system Verilog allows multiple initial assignments and multiple-step assignments.

```
module for_loop ();
initial begin
    for (int i = 0,j = 0 ; i < 4; i ++, j ++) begin
      #1 $display ("Current value of i = %g, j = %g", i, j);
    end
  #1 $finish;
end
endmodule
```

Here, i and j are automatic variables which are locally declared. These locally declared variables can neither be referred to in hierarchy nor dumped in VCD files.

While and Do-While-Loop Statement

System Verilog additionally introduces **while** and **do-while** loops. The **while** loop works exactly like it works in C language. Statement executes when certain condition satisfies. An example is shown below

```
module while_tb;
initial begin
int cnt = 0;
while (cnt<5) begin
$display("cnt = %0d",cnt);
cnt ++;
end
end
endmodule
```

The **do-while** loop first executes the procedural statements and then evaluates the condition. An example of this type of loop is shown below.

```
module do-while_tb;
initial begin
int cnt = 0;
do begin
$display(" cnt = %0d",cnt);
cnt ++;
end while (cnt<5);
end endmodule
```

Foreach-Loop Statement

This is a newly introduced loop statement in system Verilog. This loop can be used to iterate over the elements of a single or multidimensional array without knowing the size of the array. The argument must be an array with a list of loop variables. These loop variables are local or automatic by default.

```
byte word [0:3] [0:15];
.....
foreach(word [i,j])
word[i][j] = i * j;
```

Here, i and j are local variables. The iteration takes place from left to right. For each value of i, loop iterates over each value of j. This loop can also be possible for three variables as shown below. Here, innermost loop is k and outermost loop is i.

```
logic matrix [1:4][2:5][4:0];
...
foreach(word [i,j,k])
```

Jump Statements

Like in C language, system Verilog introduces some jump statements to jump out of a loop when required. This facilitates the programmer to model a hardware block easily. These jump statements are

- Break Statement.
- Continue Statement.
- Return Statement.

The **break** statement causes the loop to terminate and exit. An example is shown below. As soon as the count for i reaches 5, the for loop stops executing. The statement which immediately follows the loop executes after the execution of break statement.

```
module break_loop(input a[7:0], output reg b [7:0]);
always @*
begin
for (int i = 0; i <10; i++) begin
b[i] = a[i + 1];
if (i == 5)
break; end
end
endmodule
```

The **continue** statement is used to execute the loop if some condition is satisfied. This statement is less often used. An example is shown below. Here, assignment of array a to another array b is executed only if i is equal to 0 or 2.

```
module continue_loop(input a[3:0], output reg b [3:0]);
always @*
begin
for (int i = 0; i <10; i++) begin
if (i == 0 || i==2)
b[i] = a[i + 1];
continue; end
end
endmodule
```

The **return** statement is used in a task or function to exit from a task or a function, respectively. In the following example, **return** statement returns $(a + b)$ or $(a - b)$ depending upon whether a is lesser or greater than b, respectively.

```
module return_loop(input [3:0] a,b, output reg [3:0] s);
function [3:0] my_func(input [3:0] a,b);
    if(a<b)
    return(a+b);
    else
    return(a–b);
  endfunction
always_comb
s = my_func(a,b);
endmodule
```

Block and Statement Labels

Labelling in Verilog was done by adding a label after the keyword **begin**. In system Verilog, a label can be added after the keyword **end** also but both the label after **begin** and **end** should be same. System Verilog also allows statement levels in order to give coders good readability and for the ease of troubleshooting. An example is shown below:

```
begin : ADDER //Label for block
  summation: s <= b ^ a;//Statement Label
  carry : c <= a & b;//Statement Label
end : ADDER  //Closing Label for block
```

17.5.3 Case Statement

The **case** statement in system Verilog is improved and three keywords unique, **unique0** and **priority** are added so that violations can be checked and reported.

Unique and Unique0 Case Statement

The **unique case** statement specifies that the case expression evaluates to only one of the case values. Thus, the case values can be written in any order and selection can be done in parallel. In **unique case**, statement case values must be non-overlapping. This statement will report violation if no case value is matched or more than one case values are same.

```
module MUX( input a,b,c, input [1:0] s, output reg y);
always_comb
unique case(s) //: ADDER
  2'b00 : y = a;
  2'b01 : y = b;
endcase
endmodule
```

In the case of **unique0 case** statement, no violation is reported if case expression does not match any case values.

Priority Case Statement

Priority case statement specifies that case expressions should match at least one case item. It does not report violation if more than one match is found. If more than one matching case items are found, then the first matching branch is evaluated. In the following example, if a is equal to 0 or 1, then statement 'Priority Casez 0 or 1' is displayed.

```
always @ (*)
begin
priority casez(a)
3'b00?: $display("Priority Casez 0 or 1");
3'b0??: $display("Priority Casez 2 or 3");
endcase
end
```

Case Inside Statement

The **case inside** statement (case statement with **inside** operator) can be used to check whether the case expression is a member of the case items or not. A simple example is shown below where case expression *s* checked whether any match occurs or not inside the array of case items [3:0]. Along with **inside** operator, **unique**, **unique0** or **priority** word can be added.

```
module MUX(input a,b,c, input [1:0] s,output reg y);
always_comb
case(s) inside//: ADDER
  [0:3] : y = a;
endcase
endmodule
```

17.5.4 If Statement

The keywords **unique**, **unique0** or **priority** can be added to if statement also.

Unique and Unique0 if Statement

Similar to the **unique case** statement, in the case of a series of **if–else–if** statements order of the conditions are not important. The conditions must be unique or mutually exclusive and at least one condition must be satisfied. Thus, **unique if** statement gives warning if more than one conditions are same. An example is shown below. **Unique0** does not report warning if more than one conditions are same. The last **else** statement is required to avoid warning.

```
always @ (*)
begin
unique if ((a==0) || (a==1)) begin
$display("Unique if : 0 or 1");
end else if (a == 2) begin
$display("Unique if : 2");
end else if (a == 4) begin
$display("Unique if : 4");
end end
```

Priority if Statement

In the case of **priority if** statement, order of the conditions is important. All the conditions are evaluated as they are specified. This is like the normal **if** statement only difference is that **priority if** statement generates warning if no conditions are satisfied and hence the last **else** statement is required. In the following code, if $a = 5$ then no conditions are satisfied and a warning will be generated as the last **else** statement is not present.

```
always @ (*)
begin
priority if ((a==0) || (a==1)) begin
$display("Priority if : 0 or 1");
end else if (a == 2) begin
$display("Priority if : 2");
end else if (a == 4) begin
$display("Priority if : 4");
end end
```

17.5.5 Final Statement

The **final** block is like an **initial** block, defining a procedural block of statements, except that it occurs at the end of simulation time and executes without delays. A

final block is typically used to display statistical information about the simulation. The only statements allowed inside a final block are those permitted inside a function declaration. This guarantees that they execute within a single simulation cycle. Unlike an **initial** block, the final block does not execute as a separate process instead it executes in zero time, the same as a function call.

```
module final_block ();
initial begin
for (int i = 0 ; i < 10; i ++) begin
$display ("@%g Continue with next iteration", $time);
end
#1 $finish;
end
final begin
$display ("Final block called at time %g", $time);
end
```

17.5.6 Disable Statement

System Verilog has **break** and **continue** statement to break out of or continue the execution of loops. The **disable** statement is also allowed to disable a named block, which does not contain the **disable** statement. If the block is currently executing, this causes control to jump to the statement immediately after the block. If the block is a loop body, it acts like a **continue**. If the block is not currently executing, the disable has no effect.

```
module disable_block ();
initial begin
  fork : FORK
    for (int i = 0 ; i < 9; i ++) begin
      #1 $display ("First  -> Current value of i = %g", i);
    end
    for (int j = 0 ; j < 10; j ++) begin : FOR_LOOP
      if (j == 4) begin
        $display ("Disable FORK");
        disable FORK;
      end
      #1 $display ("Second -> Current value of j = %g", j);
    end
  join
  #10 $finish;
end
endmodule
```

17.5.7 Event Control

If Conditional Event

System Verilog introduces **iff** qualifier (if and only if) @ event control. The event expression triggers only if the condition is true. In the following example, the always block does not get triggered when there is a positive edge of clock signal and 'rst == 1'. This is shown in the below example.

```
module DFF_sv(input d,clk,rst, output reg q);
always_ @(posedge clk iff rst == 0 or posedge rst)
if (rst)
q <= 0;
else
q <= d;
endmodule
```

Edge Events

In addition to **posedge** event and **negedge** event, system Verilog introduces **edge** event. An **edge** event occurs whenever a **posedge** or **negedge** event occurs and it can be used to form event expressions just like **posedge** events and **negedge** events.

```
module DFF_sv(input d,clk,rst, output reg q);
always_ @(edge clk)
if (rst)
q <= 0;
else
q <= d;
endmodule
```

17.5.8 Continuous Assignment

System Verilog allows the continuous assignment of any variables of any data type. While nets can be driven by multiple continuous statements, variables can only be driven by one continuous assignment or one procedural block.

```
var int address;
assign address = q << 2
```

17.5.9 Parallel Blocks

The parallel processing in behavioural style in system Verilog is modified and two types of **fork-join** statements are introduced. These are **fork-join_any** and **fork-join_none** statements. These two have some differences from the original fork-join statements.

The **fork-join_any** statement blocks until any of the processes spawned by the fork completes. The parallel block completes if any of the processes completes. Here is an example of **fork-join_any** statement. At 5ns, all the parallel processes p1, p2 and p3 are spawned in parallel. The process p2 is the process which completes after 7 ns. Thus, the **fork-join_any** block exits after 12 ns. The display statement will display 12 ns.

```
module forK_join_any_tb();
initial begin
#5ns;
fork begin :p1
#10ns;
end
begin :p2
#7ns;
end
begin :p3
#15ns;
end
join_any

$display("It is now time %t", $time);
end
endmodule
```

In the **fork-join_none** statement, the parent process does not get suspended and it continues executing. In the following example, all the processes are spawned at after 5 ns. Thus, the first display statement displays 5 ns. Then all the parallel processes start executing. Process p1 ends after 15 ns, p2 ends after 12 ns and p3 ends after 20 ns but the parent process is executing. The second display statement executes after 55 ns. If normal **fork-join** statement was used, then the first display statement would be executed at 20 ns and the second display statement would be executed at 70 ns time.

```
module forK_join_none_tb();
initial begin

#5ns;
fork begin :p1
#10ns;
end
begin :p2
#7ns;
end
begin :p3
```

```
#10ns;
end
join_any

$display("It is now time %t", $time);
#5ns;
$display("End of time %t", $time);
end
endmodule
```

17.5.10 Process Control

System Verilog also offers constructs to either disable a process or wait for processes to complete. Example of these constructs are wait-fork and **disable-fork** statement. The **wait-fork** statement ensures that all the child process of a parent process are completed their execution. An example is shown below. If **wait-fork** statement was not used, then the display statement would be executed at 12 ns as p2 process takes only 7 ns. But the wait-fork statement causes to wait for all the processes to complete and thus the display statement executes at 15 ns.

```
module wait_fork_tb();
initial begin
#5ns;

fork
begin :p1
#10ns;
end
begin :p2
#7ns;
end
begin :p3
#10ns;
end
join_any
wait fork;

$display("End of time %t", $time);
end
endmodule
```

The **disable-fork** statement disables all the child processes of the parent process. In the following example, all the processes are spawned at 5 ns. The **disable-fork** statement executes after 8 ns. Thus, this statement disables the processes p1 and p3. Only the display command under process 2 will be executed.

```
module disable_fork_tb();
initial begin
```

```
#5ns;
fork

begin :p1
#10ns; $display("End of p1 %t", $time);
end
begin :p2
#7ns; $display("End of p2 %t", $time);
end
begin :p3
#10ns; $display("End of p3 %t", $time);
end

join_none
#8ns;
disable fork;
end
endmodule
```

17.6 Structural Modelling

17.6.1 Module Prototype

In system Verilog, a module can be declared without specifying its body using a special keyword **extern**. In an **extern module** declaration, all the ports or parameters are specified. This makes easy compilation and handling of the system Verilog files. An example is shown below.

```
////FA file...
extern module Adder_2bit #(parameter N=2)
(input [N-1:0] a,b, output [N-1:0] s);

module Adder_2bit(.*);
assign s = a+b;
endmodule:Adder_2bit
```

A module prototype is declared using **extern** keyword. Now, a module prototype can be declared either inside another module or outside all the modules in a single system Verilog file. If declared outside, then the module prototype can be instantiated in other modules. In system Verilog, module name can also be mentioned after **endmodule**.

In system Verilog, a module can be declared inside another module and this is called nesting of module declaration. If a module is declared inside a module, then this module is only visible to the module in which it is declared. An example is shown below. Here, the *Adder_2bit* module is declared inside the *Add_3op_2bit* module and thus *Adder_2bit* module is only useful by *Add_3op_2bit* module.

```
module Add_3op_2bit(input [1:0] a,b,c,output [1:0] s);
wire [1:0] s1;
```

```
Adder_2bit #(2) m1(a,b,s1);
Adder_2bit #(2) m2(c,s1,s);
extern module Adder_2bit #(parameter N=2)
(input [N-1:0] a,b, output [N-1:0] s);
module Adder_2bit(.*);
assign s = a+b;
endmodule: Adder_2bit
endmodule
```

In designing a complex design with many sub-modules, it is best practice to use separate system Verilog files. In handling more files, system Verilog offers a command **'inlcude** written in a file to include the modules declared in another file. An example is shown below. Here, module *Adder_2bit* is defined in the FA.sv file which is shown earlier.

```
///Adder Fil.....
module Add_3op_2bit(input [1:0] a,b,c,output [1:0] s);
wire [1:0] s1;
'include "FA.sv"
Adder_2bit #(2) m1(a,b,s1);
Adder_2bit #(2) m2(c,s1,s);
endmodule
```

If no port type or direction is specified for the first port, then all the ports with their type and direction should be specified in the body of the module not in the port list. If the type or kind (variable or net) of the first port is specified but direction is not specified, then it is assumed that it is an **inout** port by default. In the port list, if direction of a port is not specified, then by default the direction of the previous port is considered. If direction of the first port specified but kind or type not defined, then by default it is assumed to be a wire of logic type. System Verilog allows a port to be variable instead of a net. A variable can have only a driver but a net can have multiple driver. For bidirectional ports, net type ports can be used. In Verilog, we have seen that a module can be instantiated either by maintaining the position of the ports or by mentioning the port name. System Verilog offers some features to instantiate a module with many ports with little effort. An example of module instantiation is shown below where a D flip-flop module is instantiated in a parallel in parallel out register.

```
module DFF(input clk,rst,d, output reg q);
always_ @(posedge clk) begin
if (rst)
q <= 0;
else
q <= d; end
endmodule
module pipo_2bit(input clk,rst,input [1:0]d, output reg [1:0]q);
DFF f1(.clk,.rst,.d(d[0]), .q(q[0]));//As same clk and rst pin is
//connected to each flip-flops
DFF f2(.*,.d(d[1]), .q(q[1]));//.* is used to reduce writing
//effort to declare same ports.
endmodule
```

In this instantiation, *clk* and *rst* signals are connected to each flip-flop and thus writing *.clk* and *.rst* is enough. Sometimes, similar ports of the sub-module are used. In this case, to reduce to need of writing all the ports .∗ operator is used. This is called an implicit port connection. If the port names vary, then explicit port connection is required. Like, d pin of DFF module is connected to d[1] pin of *pipo_2bit* module.

In system Verilog, arrays either packed or unpacked can be passed through ports. The port also must be an array and values are passed just like an assignment. System Verilog also allows any type of port including real values, structures or unions in the port list.

System Verilog adds a fourth port connection named **ref** other than **input**, **output** and **inout**. A **ref** port passes hierarchical reference to a signal instead of passing its value. By using **ref** port, a port can be shared by both sub-module or top module. This means that if the **ref** port is a variable then it can be accessed by multiple modules. An example is shown below where an array *val*1 is accessed by both sub-module and top-module. One point is to be noted here that a module which uses the **ref** port can not be synthesized. This helps only in the verification of complex designs.

```
module submod(input [1:0] s,ref val [3:0]);
always @*
case(s)
2'b00 : val[0] = 1'b1;
2'b01 : val[1] = 1'b1;
default : val[0] = 1'b0;
endcase
endmodule

module topmod(input [3:0] s1, output reg val1 [7:0]);
submod m1(.s(s1[1:0]), .val(val1[3:0]));
always @*
case(s1)
4'b0101 : val1[5] = 1'b1;
4'b0110 : val1[6] = 1'b1;
default : val1[0] = 1'b0;
endcase
endmodule
```

In system Verilog, type or kind of an input or output port can also be parameterized. In this way, type of a port in sub-modules is not required to be fixed. An example is shown below.

```
extern module portType #(parameter VAR_TYPE = shortint)
        (input VAR_TYPE a,b);

interface ram_if(
input clka,clkb,ena,wea,enb,web,
input [2:0] ada,adb);
endinterface

module dpram(ram_if ifa, input [7:0] ina,inb, output reg [7:0]
    outa,outb);
reg [7:0] mem [0:7];
```

```
always@(posedge ifa.clka)
if(ifa.ena)begin
if(ifa.wea)
mem[ifa.ada]=ina;
else
outa = mem[ifa.ada];
end
else
outa = outa;
always@(posedge ifa.clkb)
if(ifa.enb)begin
if(ifa.web)
mem[ifa.adb]=inb;
else
outb = mem[ifa.adb];
end
else
outb = outb;
endmodule

module memmory_bank(ram_if ifa, input [7:0] a1,a2,a3,a4,
output reg [7:0] b1,b2,b3,b4);
dpram m1(ifa,a1,a2,b1,b2);
dpram m2(ifa,a3,a4,b3,b4);
endmodule


.....
ram_if ifaa(clka,clkb,ena,wea,enb,web,ada,adb);
memmory_bank uut(ifaa,a1,a2,a3,a4,b1,b2,b3,b4);
.....
```

17.7 Summary

In this chapter, some fundamental principles of system Verilog are discussed. The objective was not to discuss the whole course of system Verilog here as it is beyond the scope of this chapter. Major features of system Verilog for describing hardware using system Verilog are focussed here. Verification of a design using system Verilog is another topic which is not covered here. In contrast to Verilog HDL, system Verilog offers many attractive features which are very useful to system designers. In conclusion, it can be said that it is not mandatory to learn system Verilog for RTL design as system Verilog only helps designer for quick prototype. But it is mandatory to learn system Verilog if verification is required. Nowadays, it is industrial practice whether it is RTL design, verification or software development for intellectual properties system Verilog is the solution.

Chapter 18
Advanced FPGA Implementation Techniques

18.1 Introduction

In the previous chapters, many basic and advanced concepts of digital system design are discussed. All these systems can be implemented on either FPGA or ASIC platform but this book mainly discusses FPGA implementation of digital systems. In order to implement the digital systems on FPGA, concepts of Verilog and system Verilog are described in this book. General procedure of implementing a system on FPGA is to model the digital system using HDL and then implement (synthesize, translate, place and route) the system on FPGA via EDA tools. The implementation steps are discussed in detail in Chap. 14.

Over the past few years, many new techniques are adopted for FPGA implementation and some advanced features are also included in modern FPGAs. This enhancement in the features was necessary so that FPGAs can be easily used in the embedded systems. The objective of this chapter is to discuss these advanced techniques. This chapter focuses on how an FPGA can be easily adopted as an embedded processor.

18.2 System-On-Chip Implementation

Processors have high-level management functionality in terms of managing memory devices, input–output interfaces or high-speed transceivers. Also, processors have high system clock frequency to quickly execute functions having high timing complexity. On the other hand, FPGAs are having high data processing capability and ability to reconfigure the architecture if necessary. Thus, embedded systems also include FPGA as a part of the system. The whole embedded system using FPGA and CPU is shown below in Fig. 18.1. In the above-embedded system, FPGA and CPU are integrated as separate parts. This results in greater area on the printed circuit board (PCB) and results in greater power consumption. Thus, researchers came up with the

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7_18

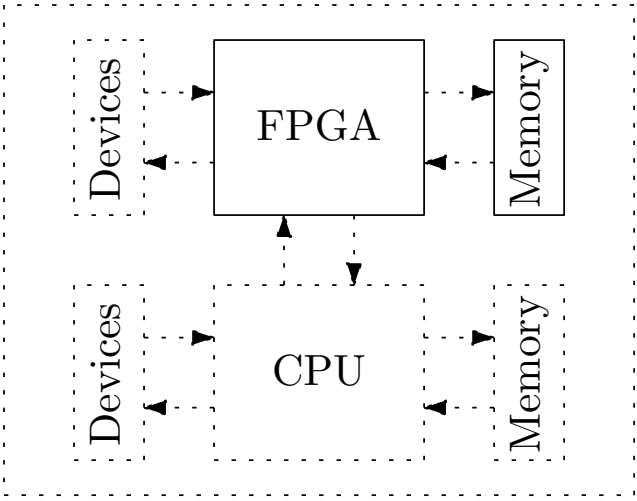


Fig. 18.1 Earlier use of FPGA as a processing system in an embedded system

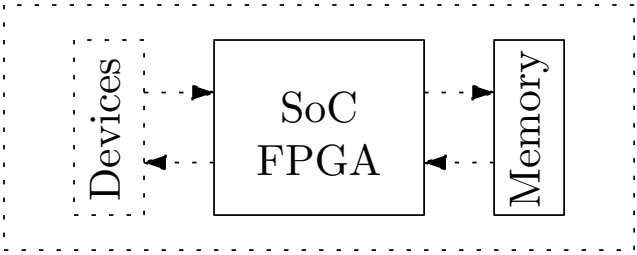


Fig. 18.2 Presently how FPGA can be used as embedded system processor

idea that both processor and FPGA can be integrated into a single IC. Both power and area are reduced by integrating FPGA and CPU on a single IC. This is called system-on-chip (SoC) as the whole embedded system is now fabricated on a single IC. FPGA can now be mainly responsible for fast data processing and parallelism of the architectures. The processor will handle the interfacing devices and take care of the serial functions which take huge time. This SoC architecture scheme is shown in Fig. 18.2. SoCs can be of great use in the case of real-time operations in comparison to the ASIC-based embedded systems. In SoCs, logic implemented on FPGA is equivalent to the logic implemented on the ASIC part. FPGA-based SoC system has an extra advantage over the ASIC-based SoCs and that is reconfigurability. Both types of embedded systems are shown in Fig. 18.3. At present, all the FPGA vendors are making their own SoC FPGAs. A simple comparison of three basic SoC FPGAs from three key FPGA vendors is shown in Table 18.1.

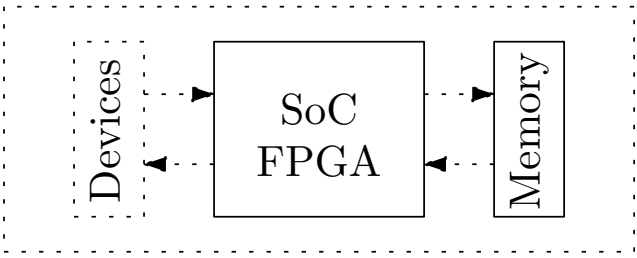


Fig. 18.3 ASIC-based embedded processor vs FPGA-based embedded processor

Table 18.1 Comparison of basic SoC FPGAs from three key manufacturers

Parameters	Intel SoC	Xilinx Zynq 7000 EPP	Microsemi SmartFusion2
Processor	ARM Cortex-A9	ARM Cortex-A9 ARM	Cortex-M3
Processor class	Application processor	Application processor	Microcontroller
Single or dual core	Single or dual	Dual	Single
Processor max. frequency	1.05 GHz	1.0 GHz	166 MHz
FPGA fabric	Cyclone V, Arria V	Artix-7, Kintex-7	Fusion2

SoC FPGAs have the following advantages over the ASIC-based embedded system:

- No expensive non-recurring engineering (NRE) charges or minimum purchase requirements, for a single, SoC FPGA or millions of devices, cost-effectively.
- Faster time to market. Devices are available off-the-shelf.
- The SoC FPGA can be reprogrammed at any time, even after shipping thus have lower risk.
- Adaptable to changing markets requirements and standards, supporting for in-field updates and upgrades.
- No additional licensing or royalty payments are required for the embedded processor, high-speed transceivers or other advanced system technology.

18.2.1 Implementations Using SoC FPGAs

In this section, implementation of a digital system using SoC FPGAs is discussed. Though many SoC FPGAs are available in the market from different manufacturers, Xilinx XC7Z010 SoC FPGA is adopted in this chapter to illustrate the procedure of implementation. A SoC FPGA has two parts, processing system (PS) and programmable logic (PL). PS part corresponds to logic implementation on the dedicated

processor (XILINX 7000 processing system) and PL part corresponds to the FPGA part. An user can place part of the complex design in PS part and rest part on PL part according to the requirement.

Implementation of a simple NAND gate on the SoC FPGA is described here. The NAND gate is opposite of the AND gate. Thus, AND logic is implemented on the PL part and the NOT logic part is implemented on the PS part. A simple Verilog code for AND gate is shown below.

```
module nand1(input a, b, output s);
assign s = a&b;
endmodule
```

Here, AND gate has two inputs a and b. The signal s is the output of the AND gate. This output will go to the NOT gate which will be implemented on the PS part.

XILINX provides intellectual property (IP) blocks for invoking the ZYNQ 7000 processing systems for free. Also, the interfacing systems for the PS part are also available for free in the XILINX EDA tool. In the block design mode of the XILINX Vivado tool, ZYNQ 7000 processing system and other IPs for interfacing are invoked. The HDL is also converted to a block and invoked in the same block design. A snapshot of the block design window is shown in Fig. 18.4. In the block design, shown in figure, RTL block corresponds to the HDL code for AND gate. The output of the AND gate is connected to $ps7\_0\_axi\_periph$ block through the axi_gpio_0 block. ZYNQ processing system gets the output of the AND gate through the $ps7\_0\_axi\_periph$ block. Now, output of the AND gate is inverted in the PS part to get the final output of NAND gate. Output of the PS part again reaches the axi_gpio_1 block through the same $ps7\_0\_axi\_periph$ block. Finally, output of the NAND gate s is connected to the external world through the axi_gpio_1 block. So, in this implementation, two general-purpose input–output (GPIO) interface blocks

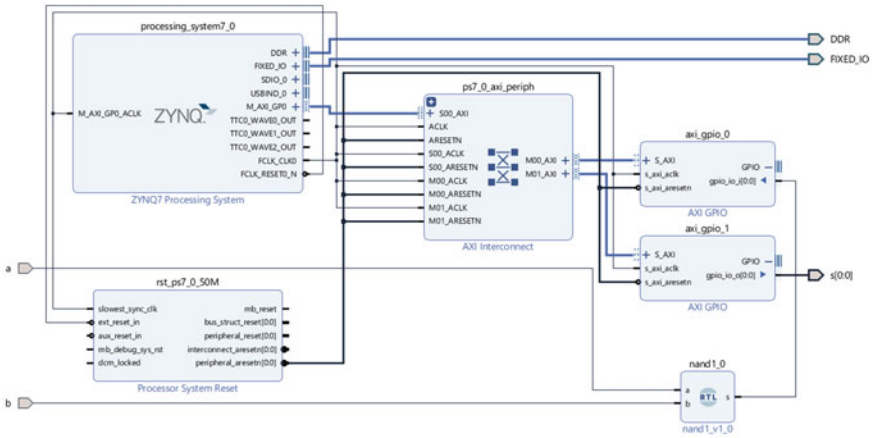


Fig. 18.4 Block design for implementing a NAND gate on XILINX ZYNQ SoC

are used and these blocks are controlled by $ps7\_0\_axi\_periph$ peripheral control block. Another block, $rst\_ps7\_0\_50M$, is used which is for controlling the reset signal.

Once this block design is completed, the block design must be converted to an HDL equivalent file called HDL wrapper file. The HDL wrapper file is then simulated and functionally verified. Once the verification is completed, the hardware constraint file is written and implementation process is started. The output of the implementation process is a .bit file. This bit file is not enough to give the output as the output of the AND gate is connected to the PS system. Thus, the next step is to realize the NOT gate in the PS system.

The PS part of the SoC system can be programmed by SDK tool provided by the XILINX Vivado. The PS part can be programmed by either C or C++ language. A small code is needed to be written in C language to realize the NOT gate. This program is shown below.

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpio.h"
#include "xparameters.h"
int main()
{
init_platform();

XGpio input, output;
int a;
int y;
XGpio_Initialize(&input, XPAR_AXI_GPIO_0_DEVICE_ID);
XGpio_Initialize(&output, XPAR_AXI_GPIO_1_DEVICE_ID);
XGpio_SetDataDirection(&input,1,1);
XGpio_SetDataDirection(&output,1,0);
print("We are up");
while(1)
  {
   a = XGpio_DiscreteRead(&input,1);
   if (a==1)
   {
   y = 0;
   }
   else
   {
   y = 1;
   }
   XGpio_DiscreteWrite(&output,1,y);
   }
   cleanup_platform();
   return 0;
}
```

18.2.2 AXI Protocol

The Arm advanced microcontroller bus architecture (AMBA) is an open-source on-chip interconnect specification for the connection and management of functional blocks in SoC designs. Advanced extensible interface (AXI) protocol is one of the variations of AMBA-type interface management system. Most of the SoC makers are nowadays using either AXI3 or AXI4 protocols to manage the interfaces which come as packed IPs. Thus, basic knowledge of AXI protocol is now necessary.

In AXI3 and AXI4, there are only two AXI interface types, master and slave. These interface types are symmetrical. All AXI connections are between master interfaces and slave interfaces. AXI interconnect interfaces contain the same signals, which makes integration of different IPs relatively simple. The diagram shown in Fig. 18.5 shows AXI connections join master and slave interfaces. The direct connection gives maximum bandwidth between the master and slave components with no extra logic and with AXI, there is only a single protocol to validate. In this figure, two masters and slaves are connected. It is also possible to connect multiple master and slave modules to a single AXI bus.

18.2.2.1 AXI Channels

The AXI specification describes a point-to-point protocol between two interfaces: a master and a slave. AXI protocol uses five main channels to establish communication between a master and a slave. These channels are used for either write or read operation as shown in Fig. 18.6.

The following channels are used for write operation.

- The master sends an address on the write address (AW) channel and data is transferred on the write data (W) channel to the slave.
- The slave writes the received data to the address sent by master. Once the slave has completed the write operation, slave responds with a message to the master on the write response (B) channel after it completes the write operation.

Read operations use the following channels:

- The master sends the address on the read address (AR) channel.
- The slave sends the data on the read data (R) channel from the requested address to the master. The slave can also send an error message through this channel if read operation is not valid or address is not correct.

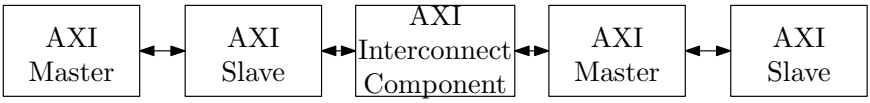


Fig. 18.5 Connection of master and slave interfaces using AXI interface protocol

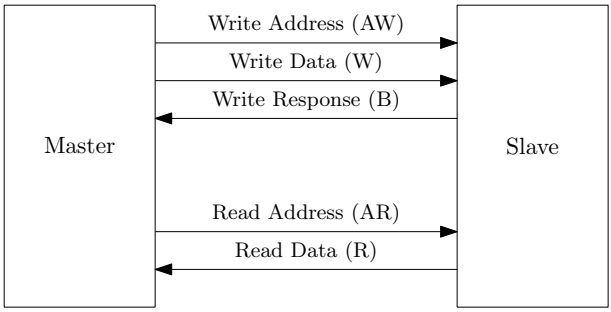


Fig. 18.6 AXI channels for communication between a master and channel

Each of these five channels contains several signals, and all these signals in each channel have the prefix as follows:

- AW for signals on the *write address* channel.
- AR for signals on the *read address* channel.
- W for signals on the *write data* channel.
- R for signals on the *Read data* channel.
- B for signals on the *write response* channel.

18.2.3 AXI Protocol Features

The AXI protocol has several key features that are designed to improve bandwidth and latency of data transfers and transactions and these features are discussed below:

- Independent read and write channels: AXI supports two different sets of channels, one for write and another for read operations. Bandwidth performances of the interfaces are improved because of two independent sets of channels.
- Multiple outstanding addresses: AXI allows for multiple outstanding addresses. This means that a master can issue transactions without waiting for earlier transactions to complete. This can improve system performance because it enables parallel processing of transactions.
- No strict timing relationship between address and data operations: There is no strict timing relationship between the address and data operations in the AXI. This means that, for example, a master could issue a write address on the *write address* channel, but there is no time requirement for when the master has to provide the corresponding data to write on the *write data* channel.
- Support for unaligned data transfers: For any burst that is made up of data transfers wider than one byte, the first bytes accessed can be unaligned with the natural address boundary. For example, a 32-bit data packet that starts at a byte address of 0x1002 is not aligned with the natural 32-bit address boundary.

- Out-of-order transaction completion: Out-of-order transaction completion is possible with AXI. The AXI protocol includes transaction identifiers, and there is no restriction on the completion of transactions with different ID values. This means that a single physical port can support out-of-order transactions by acting as several logical ports, each of which handles its transactions in order.
- Burst transactions based on start address: AXI masters only issue the starting address for the first transfer. For any following transfers, the slave will calculate the next transfer address based on the burst type.

18.3 Partial Re-configuration (PR)

There are three type re-configuration mechanisms present in the current FPGAs and they are

- Static Re-configuration—In the static re-configuration, the whole bit stream is modified and loaded o the FPGA. Execution is stopped to load the bit-stream file.
- Static PR (SPR)—In this case, only a part of the bit file is modified instead of modifying the whole bit stream. The execution is stopped for a very short time to load the bit stream.
- Dynamic PR (DPR)—In the case of DPR technique, part of the whole bit stream is modified and this part can be loaded to the FPGA without halting the execution.

18.3.1 Dynamic PR

DPR technique has become very attractive nowadays because of its run-time reconfiguration feature. Every sector is adopting DPR technique so that only a part of the whole design can be programmed without hampering the whole design.

18.3.2 Advantages of DPR

The advantages of FPGAs over ASIC ICs are discussed many times in this book. Prototype of a complex digital system can be rapidly done on FPGA and the prototype design can be reconfigured. Reconfiguration means the designer can change the architecture whenever required. But this configuration was not possible at run-time till the development of DPR. Run-time configuration justifies the advantages of FPGA more strongly. The advantages of DPR are mentioned below.

- Re-configuration time in FPGAs which having millions of logics gates is very high. This high reconfiguration time is not suitable for many applications. In such applications, DPR can be used to speed up the process. Partial reconfiguration

time is proportional to the size of the bit stream, which in turn is proportional to the area of the chip being reconfigured.
- High-density FPGAs are having high chances of failure due to single event upsets (SEU). SEUs are caused by ionizing radiation strikes that discharge the charge in storage elements, such as configuration memory cells, user memory and registers. SEUs can be detected and compensated with the help of DPR (e.g. using configuration scrubbing).
- Substantial relative increase in static power consumption is another issue in high-density FPGAs. The static power consumption is related directly to the device capacity. A system might be implemented on a smaller and consequently less power-consuming device with the help of DPR.
- DPR is also useful in applications where one part of the system should be always functional. In such case, only the re-configurable parts are configured without hampering the part which needs to be functional.

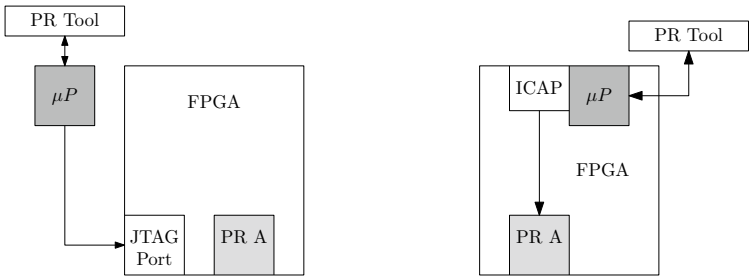
18.3.3 DPR Techniques

There are mainly two types of DPR techniques on an FPGA and they are

- Difference-based PR—Difference-based PR is used when small changes are to be done on an FPGA. For example, if some contents in the LUT or in a dedicated memory are to be changed. In this case, only the difference between the original file and the present file is loaded. Thus, the size of the new bit file is small.
- Module-based PR—This type of PR technique is required when large blocks are to be partially re-configured. Re-configurable modules are distinct portions of the main design and thus module-specific properties and layouts are needs to be changed. The design has to be partitioned and proper planning has to be done.

The dynamic PR can be achieved by some PR control logic. This PR control logic can reside either in the FPGA or in an external processor. Based on the location of the PR control logic, DPR can be done in the following two ways:

- Externally: This type of DPR uses a serial configuration port like JTAG port, processor configuration access Port (PCAP) or media configuration access port (MCAP). This is shown in Fig. 18.7.
- Internally: This type of DPR uses internal configuration access port (ICAP). Internal configuration can consist of either a custom state machine or an embedded processor such as MicroBlaze processor. This DPR technique is shown in Fig. 18.7.



(a) External DPR mechanism. (b) Internal DPR mechanism.

Fig. 18.7 Different mechanisms for DPR

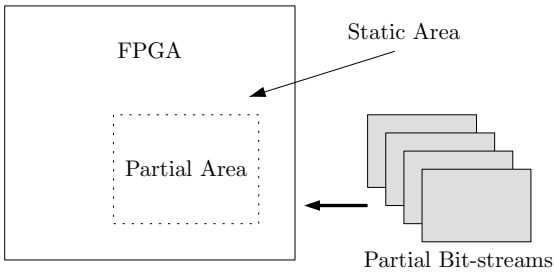
18.3.4 DPR Terminology

During DPR, the FPGA is divided into two parts, static area and partial area. The static area is fixed and is not re-configured during runtime. The area which is eligible for run-time re-configuration is called partial area. This is shown in Fig. 18.8. An FPGA may have several partial areas. In time-multiplexed manner, modules are loaded to the partial areas and every partial bit stream corresponds to a single module.

The size of a partial area should be at least the size of the most extensive module. As a consequence, there is usually a waste of logic resources that arises if modules with different resource requirements share the same island exclusively, which is called internal fragmentation. The reason is that a large module cannot be replaced by multiple smaller ones (to be hosted simultaneously). Therefore, the utilization of the partial area becomes inefficient. In Fig. 18.9, the white surfaces in the partial areas indicate the unused reconfigurable area, and thus the internal fragmentation. The reconfiguration of the partial areas can be categorized into multiple styles and these are

- Island-style.
- Slot-style.
- Grid-style.

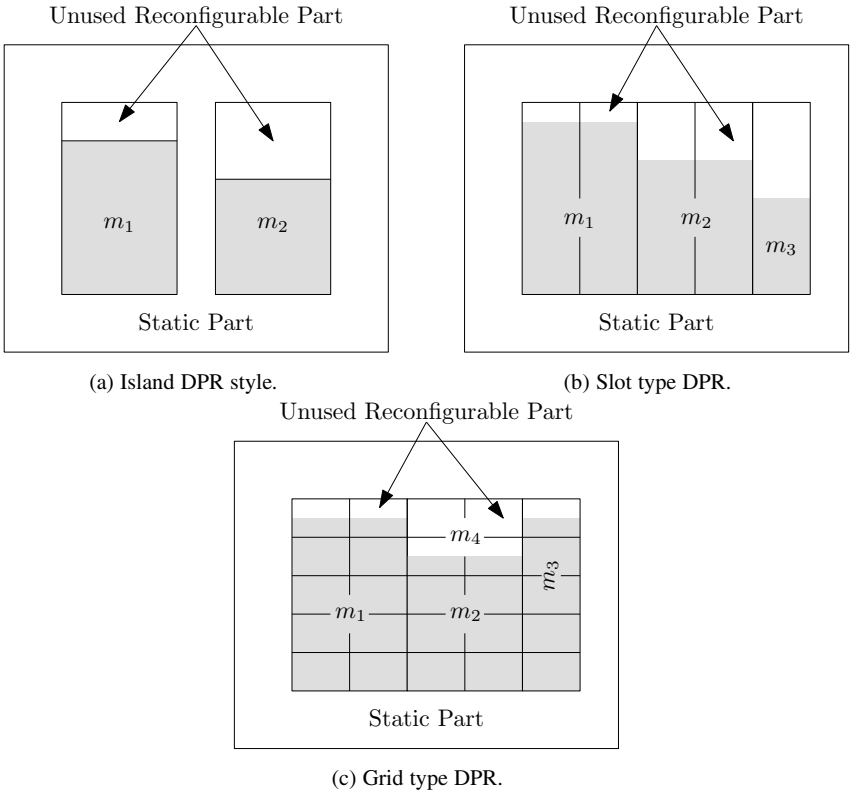
Fig. 18.8 Partitioning of FPGA in two parts for DPR



18.3.4.1 Island-Style

In island-style, only one module can reside in the partial area at the same time. This style is shown in Fig. 18.9a and here system provides multiple islands. If there is a specific island reserved for a set of modules then this style is called single island style. In multi-island style, module relocation is feasible among different islands. The use of same module at various locations on FPGA fabric is called module relocation. Module relocation is possible due to the instantiation of same module multiple times.

The size of the partial blocks is taken as the size of the module which has maximum area. This results in waste of logical resources if the modules with different sizes share the same island. This is called internal fragmentation. A large module cannot be replaced by multiple smaller ones which are to be hosted simultaneously. So, the utilization of the partial area becomes inefficient.



(a) Island DPR style.

(b) Slot type DPR.

(c) Grid type DPR.

Fig. 18.9 Different styles for DPR

18.3.4.2 Slot-Style

In slot-style, it is attempted to reduce the internal fragmentation effect. In this style, partial areas are arranged in one-dimensional slots. This style is illustrated in Fig. 18.9b. The partial area can host multiple modules at the same time, and modules can occupy the number of slots according to their resource requirements. This style is slightly complicated as communication between the modules has to be provided. Slot-style reduces the internal fragmentation but can generate external fragmentation. External fragmentation is the wastage of logic cells if any slot is left unused or partially used due to the heterogeneous size of FPGA resources.

18.3.4.3 Grid-Style

In slot-style, most of the wasted spaces are reduced in the slot-style but there may be a possibility that a slot remains unused. A more advanced style is grid-style where chances of fragmentation effect are less. If a module is using less resources, then another module should use the remaining resources. This is possible in grid-style reconfiguration which is shown in Fig. 18.9c. In this reconfiguration, the slots are arranged in a two-dimensional fashion. The implementation and management of this style is even more complex.

18.3.5 DPR Tools

DPR process is comparatively complex than the normal implementation of a digital system. Thus, EDA tool vendors launched advanced software applications to support DPR process. DPR process steps are almost similar for the two popular vendors XILINX and Intel. Xilinx provides PlanAhead and Vivado Design Suite for DPR process. On the other hand, Intel has Quartus-II and Quartus Prime to support DPR. These tools are commercial and may not be available to the academic researchers. Thus, alternative open-source tools are also available in the market. These tools are OpenPR, GoAhead, Dreams and CoPR. These tools have limited functionality but are freely available.

18.3.6 DPR Flow

In this section, DPR flow is described with the help of the Xilinx Vivado Design Suit software tool. The flow chart for the DPR process is shown in Fig. 18.10. The first step for DPR is to partition the design in terms of static and partial regions. This information is given in the constraints file. According to the partitioning, design floor planning and budgeting step is carried out. the designer has to manually floorplan

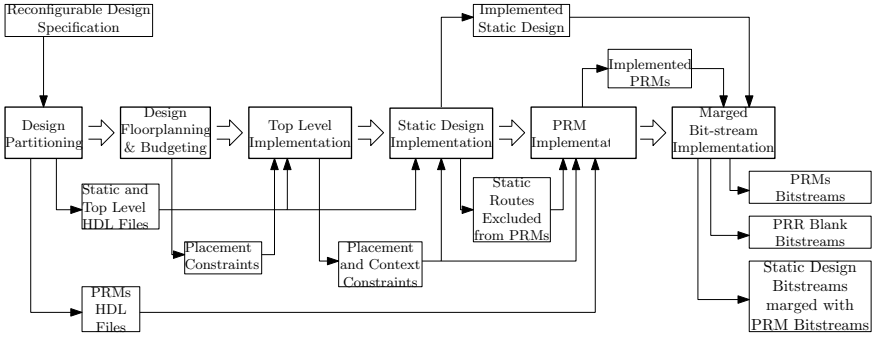


Fig. 18.10 The flow chart for the DPR process

the locations and bounding boxes of the reconfigurable regions on the FPGA fabric. Floorplanning details are written in a constraint file for incorporation in the implementation stage. The next step is to start the top-level implementation. First static design is implemented and bit file is generated. Then the PR modules (PRM) are implemented and separate bit files are generated. At the last stage, all these bit files are combined.

Communication between the static design and PRMs is an essential part of DPR design. Current vendors use proxy logic for connection between static and PRMs. Proxy logic are anchor LUTs, which are placed inside the partial area for each interface signal, as shown in Fig. 18.11. The interface signals are routed to the anchor LUTs during the implementation of the static system. The partial modules are implemented as an increment to the static system without modifying any of the already implemented static routings.

18.3.7 Communication Between Reconfigurable Modules

During the dynamic configuration, the whole FPGA is divided into mainly two parts, static and partially reconfigurable areas. There may be many reconfigurable areas according to the requirement of the design. Building communication between the static module and the configuration modules in an efficient way is a very important topic. Also, the communication between configuration modules is also important. Many techniques are proposed over the past few years for efficient communication between these modules. These techniques are outlined below.

18.3.7.1 Partial Module Linking via I/O Pins

Initially, to achieve linking between the partial modules FPGA I/O pins were used. Many FPGA projects adopted this technology where a resource slot engages a complete device and partial reconfiguration is achieved on system level by exchanging some full FPGA configurations. But DPR is meant to share a fraction of whole FPGA configurations. This technique is used in FPGA projects [9, 79] where Xilinx FPGAs are used. In [43], all interface signals of a partial module are routed to I/O pins, located inside the module bounding box and most of these pins are connected to external memory buses. In another FPGA project [4], partial modules can be directly connected to external peripherals with the help of an external crossbar chip which adds extra latency. In this scheme of connecting partial modules, modules may be relocated to different positions but they can access a private external memory bank.

18.3.7.2 Partial Module Linking via Tristate Buffers

Partial modules are linked with tristate buffers and such tristate signals have been mainly used for implementing wide input functions, such as wired-OR or wired- AND gates, as well as for global communication purposes. FPGA projects depicted in [81, 82], used tristate buffers to link the PRMs where two FPGAs are used one control and one for accelerating computations. Most Xlinx FPGAs which uses tristate buffers for module linking are based on BUS Macros. These macros define the positions of a set of tristate drivers which connects the horizontal long lines. This is due to the fact that Xilinx follows a column-wise reconfiguration of the modules. Linking partially reconfigurable modules with tristate drivers on Xilinx FPGAs can be implemented with low logic resource overhead. A few other projects which use this technique are depicted in [32, 75]. Modern FPGA architectures are no longer supporting internal tristate drivers, this was common in many older FPGA architectures.

18.3.7.3 Partial Module Linking via Logic Resources

Partial modules can also be connected via logic sources like LUT. The connection by LUTs is achieved by using one LUT at the beginning and one at the end of the wire resource. These LUTs are placed such that one LUT is placed inside the PRMs and the other one inside the static part. As a result, LUTs act as a connection point between the partial modules. This scheme is shown in Fig. 18.11. As compared to the tristate driver approach, the LUT bus macros allow a higher density of signals and a lower latency by the cost of two LUTs just to connect a static system and a partial module. Many newer Xilinx FPGAs adopted this technique to achieve DPR. In FPGA projects [36, 37], based on Xilinx Virtex-4, LUT-based linking is used. Row-based two-dimensional module linking based on LUT macros is reported in [64].

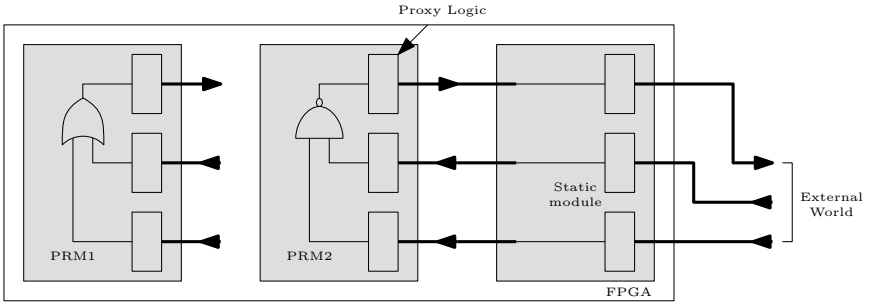


Fig. 18.11 Connection of static module with partial modules using LUT bus macros

18.3.7.4 Partial Module Linking using Proxy Logic

The LUT macro approach uses two LUTs and also it has some extra latency. This method is improved and a new approach called incremental partial design flow is proposed. In this method, a LUT which implements some proxy logic is placed in partial module for each signal which passes from partial module to static module. All these signals which cross the boundary go to another anchor LUT which is situated in the static module. This approach is shown in Fig. 18.12. This approach needs one LUT per signal and its major drawback is that routing is different for each reconfigurable island. This prevents module relocation even if the islands provide identical logic and memory layout. Changes in the static system will result different in the proxy logic. This approach is more suitable for systems having lower complexity.

18.3.7.5 Zero Logic Overhead Integration

Previous LUT-based approaches have logic overhead and thus have extra latency. As an alternative to Xilinx bus macro and proxy logic, an approach to link the par-

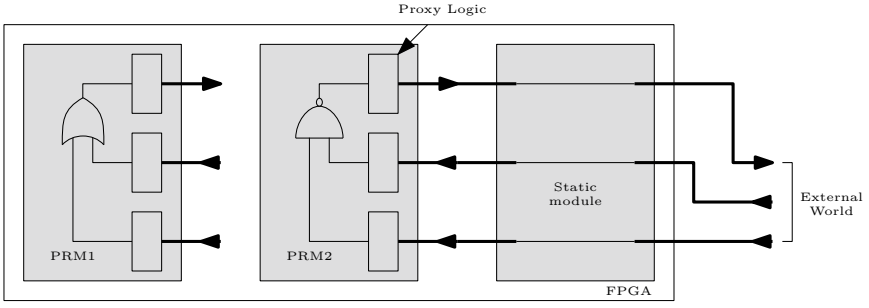


Fig. 18.12 Connection of static module with partial modules using proxy logic

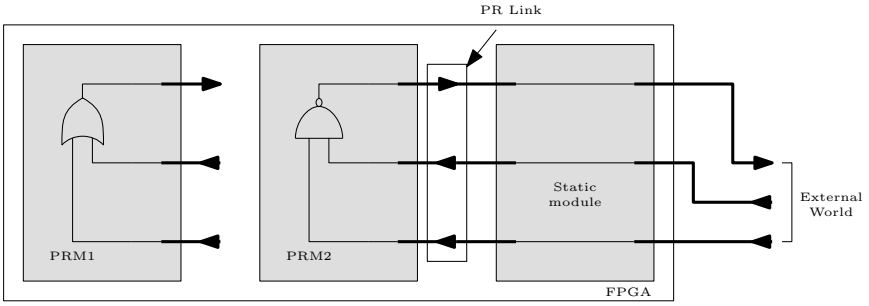
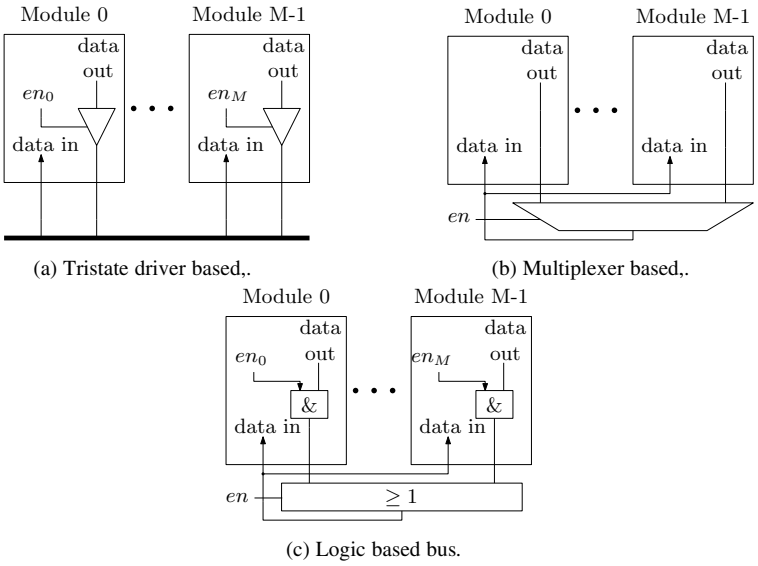


Fig. 18.13 Connection of static module with partial modules using zero logic overhead integration

tial modules without logic overhead is reported in [34]. The zero-overhead integration combines Xilinx bus macro technique with proxy logic. PRMs can be plugged directly into a reconfigurable region without additional logic resources for the communication and only the wires act as plugs, called PR link. This is shown in Fig. 18.13.

18.3.7.6 Bus-Based Communication for Reconfigurable Systems

In computer systems, buses can be easily used to establish communication between modules and memory devices. Thus, buses can also be adopted for establishing



(a) Tristate driver based,.

(b) Multiplexer based,.

(c) Logic based bus.

Fig. 18.14 Common bus implementations for inter modules communication

communication between the static module and the reconfiguration modules. A bus is a set of signals and in that set different signals have special purposes. A bus act as a shared medium for the communication between different modules connected to the bus. Modules which are connected to the bus can be of two types which are master and slave. A master module can control the bus and a slave module can respond to the instructions set by master. For example, a master can give the address to the bus. Different kinds of buses are shown in Fig. 18.14. Any of these buses can be adopted for communication between the modules.

18.4 Conclusion

In this chapter, advance programming techniques for FPGAs are discussed. SoC-based embedded systems are very important nowadays where FPGA is used to perform the tasks which have high computational complexity and processor part is used to perform the tasks which have high timing complexity. AXI protocol is generally used to connect the interfaces in an SoC design and thus a basic outline of AXI topology is discussed in this chapter. Partial reconfiguration is another important area of research nowadays. In DPR, part of the whole design can be reprogrammed, while the other part is not affected. DPR is very important in applications where the execution should not be stopped. The techniques to achieve DPR is growing day by day. Some basics of re-configuration are discussed in this chapter.

References

1. Double-dabble binary-to-bcd conversion algorithm.
2. Adiono, T., Ahmadi, N., Renardy, A. P., Fadila, A. A., & Shidqi, N. (2015). A pipelined cordic architecture and its implementation in all-digital fm modulator-demodulator. In *2015 6th Asia symposium on quality electronic design (ASQED)* (pp. 37–42).
3. Ahmed, H. M. (1989). Efficient elementary function generation with multipliers. In *Proceedings of 9th symposium on computer arithmetic* (pp. 52–59) (1989).
4. Angermeier, J., Bobda, C., Majer, M., & Teich, J. (2010). *Erlangen slot machine: An FPGA-based dynamically reconfigurable computing platform* (pp. 51–71). Netherlands, Dordrecht: Springer.
5. Antelo, E., Bruguera, J. D., & Zapata, E. L. (1996). Unified mixed radix 2–4 redundant cordic processor. *IEEE Transactions on Computers, 45*(9), 1068–1073.
6. Antelo, E., Villalba, J., Bruguera, J. D., & Zapata, E. L. (1997). High performance rotation architectures based on the radix-4 cordic algorithm. *IEEE Transactions on Computers, 46*(8), 855–870.
7. Antelo, E., Villalba, J., Bruguera, J. D., & Zapata, E. L. (1997). High performance rotation architectures based on the radix-4 cordic algorithm. *IEEE Transactions on Computers, 46*(8), 855–870.
8. Antelo, E., Villalba, J., & Zapata, E. L. (2008). A low-latency pipelined 2d and 3d cordic processors. *IEEE Transactions on Computers, 57*(3), 404–417.
9. Athanas, P., & Silverman, H. (1993). Processor reconfiguration through instruction-set metamorphosis. *Computer, 26*(3), 11–18.
10. Babu, P. R. (2009). *Digital signal processing*. SCITECH.
11. Banerjee, A., Dhar, A. S., & Banerjee, S. (2001). Fpga realization of a cordic based fft processor for biomedical signal processing. *Microprocessors and Microsystems, 25*(3), 131–142.
12. Bank, B. (2007) Direct design of parallel second-order filters for instrument body modeling. In *ICMC*.
13. Bank, B., & Smith, J. O. A delayed parallel filter structure with an fir part having improved numerical properties.
14. Barrett, P. (1987). Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko (Ed.), *Advances in cryptology - CRYPTO' 86* (pp. 311–323). Berlin: Springer.
15. Booth, A. D. (1951). A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics, 4*, 2(01), 236–240.

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7

16. Braun, T. R. (2010). An evaluation of gpu acceleration for sparse reconstruction. In *Signal processing, sensor fusion, and target recognition XIX* (2010) (Vol. 7697, p. 769715). International Society for Optics and Photonics.
17. Brent., & Kung. (1982). A regular layout for parallel adders. *IEEE Transactions on Computers C-31*(3), 260–264 (1982).
18. Bruguera, J., Antelo, E., & Zapata, E. (1993). Design of a pipelined radix 4 cordic processor. *Parallel Computing, 19*(7), 729–744.
19. Callaway, T. K., & Swartzlander, E. E. Optimizing arithmetic elements for signal processing. In *Workshop on VLSI signal processing* (pp. 91–100) (1992).
20. Cao, Z., Wei, R., & Lin, X. (2014). A fast modular reduction method. *IACR Cryptology ePrint Archive, 2014*, 40.
21. Chen, P.-Y., Lien, C.-Y., & Chuang, H.-M. (2010). A low-cost vlsi implementation for efficient removal of impulse noise. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 18*, 3, 473–481 (2010).
22. Dadda, L. (1965). Some schemes for parallel multipliers. *Alta frequenza, 34*, 349–356.
23. Dawid, H., & Meyr, H. (1996). The differential cordic algorithm: Constant scale factor redundant implementation without correcting iterations. *IEEE Transactions on Computers, 45*(3), 307–318.
24. Duprat, J., & Muller, J. (1993). The cordic algorithm: New results for fast vlsi implementation. *IEEE Transactions on Computers, 42*(2), 168–178.
25. Ercegovac, M. D., Lang, T., Muller, J., & Tisserand, A. (2000). Reciprocation, square root, inverse square root, and some elementary functions using small multipliers. *IEEE Transactions on Computers, 49*(7), 628–637.
26. Farhat, H. Finite state machine minimization and row equivalence application.
27. Francis, M. (2009). Infinite impulse response filter structures in xilinx fpgas. In *XILINX*.
28. Gisuthan, B., & Srikanthan, T. (2002). Pipelining flat cordic based trigonometric function generators. *Microelectronics Journal, 33*(01), 77–89.
29. Goldschmidt, R. Applications of division by convergence.
30. Han, T., & Carlson, D. A. (1987). Fast area-efficient vlsi adders. In *1987 IEEE 8th symposium on computer arithmetic (ARITH)* (pp. 49–56).
31. Smith, J. L. (1996). Implementing median filters in xc4000e fpgas. In *Proceedings of VII* (p. 16).
32. Kalte, H., Porrmann, M., & Ruckert, U. (2004) System-on-programmable-chip approach enabling online fine-grained 1d-placement. In *18th international parallel and distributed processing symposium, 2004. Proceedings.* (p. 141)
33. Kebbati, H., Blonde, J.-P., & Braun, F. (2006). A new semi-flat architecture for high speed and reduced area cordic chip. *Microelectronics Journal, 37*(02), 181–187.
34. Koch, D., Beckhoff, C., & Torresen, J. (2010). Zero logic overhead integration of partially reconfigurable modules. In *Proceedings of the 23rd symposium on integrated circuits and system design, SBCCI '10* (pp. 103–108). New York, NY, USA: Association for Computing Machinery.
35. Kogge, P. M., & Stone, H. S. (1973). A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers C-22, 8*, 786–793 (1973).
36. Koh, S., & Diessel, O. Comma: A communications methodology for dynamic module-based reconfiguration of fpgas, 173–182.
37. Koh, S., & Diessel, O. (2006) Comma: A communications methodology for dynamic module reconfiguration in fpgas. In *2006 14th Annual IEEE symposium on field-programmable custom computing machines* (pp. 273–274).
38. Kuhlmann, M., & Parhi, K. K. (2002). P-cordic: A precomputation based rotation cordic algorithm. *EURASIP Journal on Advances in Signal Processing, 2002*(9), 137251.
39. Kulkarni, A., & Mohsenin, T. (2015). Accelerating compressive sensing reconstruction omp algorithm with cpu, gpu, fpga and domain specific many-core. In *2015 IEEE international symposium on circuits and systems (ISCAS)* (pp. 970–973). IEEE.

40. Kulkarni, A., Shea, C., Abtahi, T., Homayoun, H., & Mohsenin, T. (2017). Low overhead cs-based heterogeneous framework for big data acceleration. *ACM Transactions on Embedded Computing Systems (TECS), 17*(1), 1–25.
41. Ladner, R. E., & Fischer, M. J. (1980). Parallel prefix computation. *Journal of the ACM (JACM), 27*(4), 831–838.
42. Ling, H. (1981). High-speed binary adder. *IBM Journal of Research and Development, 25*(3), 156–166.
43. Lockwood, J. W., Naufel, N., Turner, J. S., & Taylor, D. E. (2001). Reprogrammable network packet processing on the field programmable port extender (fpx). In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on field programmable gate arrays, FPGA '01* (pp. 87–93). New York, NY, USA: Association for Computing Machinery.
44. MacQueen, J., et al. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability* (Vol. 1, pp. 281–297). Oakland, CA, USA.
45. Macsorley, O. L. (1961). High-speed arithmetic in binary computers. *Proceedings of the IRE, 49*(1), 67–91.
46. Majithia, J. C., & Kitai, R. (1971). An iterative array for multiplication of signed binary numbers. *IEEE Transactions on Computers C-20, 2*, 214–216.
47. Martin, J. C. (1990). *Introduction to languages and the theory of computation* (1st ed.). USA: McGraw-Hill Inc.
48. Martinez, J., Cumplido, R., & Feregrino, C. (2005). An fpga-based parallel sorting architecture for the burrows wheeler transform. In *2005 international conference on reconfigurable computing and FPGAs (ReConFig'05)* (pp. 7–17).
49. Microchip. (2002). *2.7V 4-channel/8-channel 10-bit A/D converters with SPI™ Serial interface*. Microchip Technology Inc.
50. Microchip. (2010). *8/10/12-bit dual voltage output digital-to-analog converter with SPI interface*. Microchip Technology Inc.
51. Milenkovic, N. Z., Stankovic, V. V., & Milic, M. (2015). Modular design of fast leading zeros counting circuit. *Journal of Electrical Engineering, 66*, 329–333.
52. MILOS, D. E., & Lang, T. Fast cosine/sine implementation using on line cordic. In *21st annual asilomar conference on signals. system and computers*.
53. Mou, Z., & Jutand, F. (1992). 'Overturned-stairs' adder trees and multiplier design. *IEEE Transactions on Computers, 41*(8), 940–948.
54. Parhi, K. K. (2007). *VLSI digital signal processing systems: Design and implementation*. Wiley.
55. Phatak, D. S. (1998). Double step branching cordic: A new algorithm for fast sine and cosine generation. *IEEE Transactions on Computers, 47*(5), 587–602.
56. Rabah, H., Amira, A., Mohanty, B. K., Almaadeed, S., & Meher, P. K. (2015). Fpga implementation of orthogonal matching pursuit for compressive sensing reconstruction. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 23*(10), 2209–2220.
57. Gonzalez, R. C., & Woods, R. (2007). *Digital image processing*. Prentice Hall.
58. Robertson, J. E. A new class of digital division methods. *IRE Transactions on Electronic Computers EC-7*, 3 (1958), 218–222.
59. Roy, S. *Division Algorithms*. No. online http://digitalsystemdesign.in.
60. Roy, S. Parallel fpga implementation of fir filters. digitalsystemdesign.in.
61. Roy, S., Acharya, D. P., & Sahoo, A. K. (2019). Incremental gaussian elimination approach to implement omp for sparse signal measurement. *IEEE Transactions on Instrumentation and Measurement, 1*.
62. Roy, S., Acharya, D. P., & Sahoo, A. K. (2019). Low-complexity architecture of orthogonal matching pursuit based on qr decomposition. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1–10.
63. Schulte, M. J., & Stine, J. E. (1999). Approximating elementary functions with symmetric bipartite tables. *IEEE Transactions on Computers, 48*(8), 842–847.
64. Silva, M. L., & Ferreira, J. C. (2010). Creation of partial fpga configurations at run-time. In *2010 13th Euromicro conference on digital system design: Architectures, methods and tools* (pp. 80–87).

65. Sklansky, J. (1960). Conditional-sum addition logic. *IRE Transactions on Electronic Computers EC-9, 2*, 226–231.
66. Stan, M. R., & Burleson, W. P. (1994) Limited-weight codes for low-power i/o. In *International workshop on low power design* (Vol. 6, pp. 6–8). Citeseer.
67. Strollo, A. (2000). Low power flip-flop with clock gating on master and slave latches. *Electronics Letters, 36*, 294–295(1).
68. Su, C.-L., Tsui, C.-Y., & Despain, A. M. (1994). Low power architecture design and compilation techniques for high-performance processors. In *Proceedings of COMPCON'94* (pp. 489–498). (IEEE).
69. Suzuki, H., Morinaka, H., Makino, H., Nakase, Y., Mashiko, K., & Sumi, T. (1996). Leading-zero anticipatory logic for high-speed floating point addition. *IEEE Journal of Solid-State Circuits, 31*(8), 1157–1164.
70. Takagi, N., Asada, T., & Yajima, S. (1991). Redundant cordic methods with a constant scale factor for sine and cosine computation. *IEEE Transactions on Computers, 40*(9), 989–995.
71. Timmermann, D., Hahn, H., & Hosticka, B. J. (1992). Low latency time cordic algorithms. *IEEE Transactions on Computers, 41*(8), 1010–1015.
72. Tocher, K. D. (1958). Techniques of multiplication and division for automatic binary computers. *The Quarterly Journal of Mechanics and Applied Mathematics, 11*, 3(01), 364–384.
73. Juang, T.-B., Hsiao, S.-F., & Tsai, M.-Y. (2004). Para-cordic: Parallel cordic rotation algorithm. *IEEE Transactions on Circuits and Systems I: Regular Papers, 51*(8), 1515–1524.
74. Volder, J. E. (1959). The cordic trigonometric computing technique. *Electronic Computers, IRE Transactions on EC-8, 3*, 330–334.
75. Walder, H., & Platzner, M. (2004). A runtime environment for reconfigurable hardware operating systems. In J. Becker, M. Platzner, & S. Vernalde (Eds.), *Field programmable logic and application* (pp. 831–835). Berlin: Springer.
76. Wallace, C. S. (1964). A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers EC-13, 1*, 14–17.
77. Walther, J. S. (1971). A unified algorithm for elementary functions. In *Proceedings of the May 18–20, 1971, spring joint computer conference* (pp. 379–385).
78. Wang, S., Piuri, V., & Wartzlander, E. E. (1997). Hybrid cordic algorithms. *IEEE Transactions on Computers, 46*(11), 1202–1207.
79. Wazlowski, M., Agarwal, L., Lee, T., Smith, A., Lam, E., Athanas, P., Silverman, H., & Ghosh, S. (1993). Prism-ii compiler and architecture. In *[1993] Proceedings IEEE workshop on FPGAs for custom computing machines* (pp. 9–16).
80. Wimer, S., & Albahari, A. (2014). A look-ahead clock gating based on auto-gated flip-flops. *IEEE Transactions on Circuits and Systems I: Regular Papers, 61*(5), 1465–1472.
81. Wirthlin, M. J., & Hutchings, B. L. (1995). DISC: The dynamic instruction set computer. In J. Schewel (Ed.), *Field programmable gate arrays (FPGAs) for fast board development and reconfigurable computing* (Vol. 2607, pp. 92–103) . International Society for Optics and Photonics, SPIE.
82. Wirthlin, M. J., & Hutchings, B. L. (1996). Sequencing run-time reconfigured hardware with software. In *Proceedings of the 1996 ACM fourth international symposium on field-programmable gate arrays FPGA '96* (pp. 122–128). New York, NY, USA: Association for Computing Machinery.
83. XILINX. 7 series dsp48e1 slice.
84. Zuras, D., & McAllister, W. H. (1986). Balanced delay trees and combinatorial division in VLSI. *IEEE Journal of Solid-State Circuits, 21*(5), 814–819.

Index

S. Roy, *Advanced Digital System Design*,
https://doi.org/10.1007/978-3-031-41085-7