

Computer Architecture and Design Methodologies

Bekkay Hajji
Adel Mellit
Loubna Bouselham

A Practical Guide for Simulation and FPGA Implementation of Digital Design



Springer

Computer Architecture and Design Methodologies

Series Editors

Anupam Chattopadhyay, Nanyang Technological University, Singapore, Singapore

Soumitra Kumar Nandy, Indian Institute of Science, Bangalore, India

Jürgen Teich, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU),
Erlangen, Germany

Debdeep Mukhopadhyay, Indian Institute of Technology Kharagpur, Kharagpur,
West Bengal, India

Twilight zone of Moore's law is affecting computer architecture design like never before. The strongest impact on computer architecture is perhaps the move from unicore to multicore architectures, represented by commodity architectures like general purpose graphics processing units (gpgpus). Besides that, deep impact of application-specific constraints from emerging embedded applications is presenting designers with new, energy-efficient architectures like heterogeneous multi-core, accelerator-rich System-on-Chip (SoC). These effects together with the security, reliability, thermal and manufacturability challenges of nanoscale technologies are forcing computing platforms to move towards innovative solutions. Finally, the emergence of technologies beyond conventional charge-based computing has led to a series of radical new architectures and design methodologies.

The aim of this book series is to capture these diverse, emerging architectural innovations as well as the corresponding design methodologies. The scope covers the following.

- Heterogeneous multi-core SoC and their design methodology
- Domain-specific architectures and their design methodology
- Novel technology constraints, such as security, fault-tolerance and their impact on architecture design
- Novel technologies, such as resistive memory, and their impact on architecture design
- Extremely parallel architectures

More information about this series at <https://link.springer.com/bookseries/15213>

Bekkay Hajji · Adel Mellit · Loubna Bouselham

A Practical Guide for Simulation and FPGA Implementation of Digital Design



Springer

Bekkay Hajji
National School of Applied Sciences
Mohammed first University
Oujda, Morocco

Adel Mellit
Renewable Energy Laboratory
University of Jijel
Jijel, Algeria

Loubna Bouselham
National School of Architecture
Oujda, Morocco

ISSN 2367-3478 ISSN 2367-3486 (electronic)
Computer Architecture and Design Methodologies
ISBN 978-981-19-0614-5 ISBN 978-981-19-0615-2 (eBook)
<https://doi.org/10.1007/978-981-19-0615-2>

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2022

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

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

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

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

Preface

In recent years, the digital systems market has exploded in many industrial and consumer sectors such as telecommunications, satellites, medical devices, vehicles and robots. These increasingly important needs generate fierce industrial competition where factors such as cost, performance and above all “Time To Market” became essential for the success of a product. In addition, design productivity will be the real challenge for future systems. In this context, the FPGA (Field Programmable Gate Array), with its great integration and reconfiguration capacities, constitutes a key element for the rapid development of prototypes. In order to encourage the wide dissemination of this type of circuits, it is necessary to improve the development environments to make them more accessible to non-electronics experts.

This book is the result of many years of teaching a course in digital systems design and embedded systems (for undergraduate and postgraduate students) at the National School of Applied Sciences in Oujda, Mohamed Premier University, (Morocco) and at the Faculty of Science and Technology, Jijel University, (Algeria). The target audience are electrical engineering students (universities and engineering schools) as well as designers of modern digital systems.

The objective of this book is to serve as a practical guide for the FPGA simulation and implementation of digital design for all those who have already learned the classic digital design (like combinational logic and digital design). The book makes a clear contribution by offering case studies to demonstrate the book’s teachings in practice for the design and implementation of digital systems.

The book is composed of four parts. The first part of this book has two chapters and covers various aspects: FPGA architectures, ASIC versus FPGA comparison, FPGA design flow and basic VHDL concepts necessary to describe the design of digital systems. The second part of the book includes three chapters, deals with the design of digital circuits such as combinational logic circuits, sequential logic circuits and finite-state machines. It is a largely hands-on lab class for design digital circuits and then implementing their designs on an FPGA platform (Altera or Xilinx). The third part of the book is reserved to digital projects carried out on the FPGA platform,

a good number of laboratory projects have been addressed. Finally, the fourth and last part of this book is devoted to recent research work and applications carried out on FPGA including Artificial Intelligence techniques in renewable energy systems. Some real-time examples have been presented and discussed in details.

Structure of This Book

The book consists of four parts and eight chapters:

The first part of this book consists of Chaps. 1 and 2 described below:

- Chapter 1 covers an introduction to FPGA, including FPGA architectures, design metrology, design tools and a brief comparison between ASIC and FPGA.
- Chapter 2 discusses the basic VHDL concepts. First, we introduce the hardware description language, followed by VHDL design units. Next, the key VHDL constructs like concurrent functionally, sequential statement and structural statements are presented. Finally, various Test-Benches have been presented.

Three Chaps. 3–5 are described below, constitute Part II of the book:

- Chapter 3 describes combinational logic circuits in details. The chapter includes arithmetic circuits, multiplexer and demultiplexer, seven segment displays, encoders, decoders and comparators. All presented combinational logic circuits have been simulated and tested.
- Chapter 4 discusses sequential logic circuits, it includes D-latch, D-flip flop, registers and counters. Examples presented have been simulated in verify using test-bench
- Chapter 5 provides the finite-state machines (FSMs) using the VHDL. The Moore and Mealy machines and their use to code the sequence detectors and counters are described in this chapter.

Chapters 6 and 7 of the book constitute respectively the third and the last part of this book:

Various laboratory projects have been described and discussed in detail in Chap. 6. The laboratory projects are (1) Simple calculator design, (2) Digital clock (3) Traffic light control System, (4) Design and Implementation of FPGA-based vending machine, and (5) Control of a 4-phase step motor (Direction & Speed).

Chapter 7 start with a brief on the recent applications of FPGA in renewable energy system, particularly solar and wind energy systems. Followed by various case study applications: (1) FPGA-based intelligent photovoltaic module simulation, (2) FPGA-based implementation of irradiance equalization algorithm for reconfigurable photovoltaic, (3) FPGA-based implementation of an intelligent MPPT algorithm,

and (4) XSG-Based implementation control of grid-connected hybrid system (PV-WT).

Oujda, Morocco
Jijel, Algeria
Oujda, Morocco

Bekkay Hajji
Adel Mellit
Loubna Bouselham

Acknowledgments

Firstly, we would like to warmly thank our engineering students from National School of Applied Sciences (ENSA) of Oujda, First Mohamed University, Morocco, and students from the University of Jijel, Algeria, for their involvements in the development of real projects implemented into FPGA platforms.

We wish also to express my gratitude to Dr. Ramesh Nath Premnath, senior publishing editor for academic books at Springer, covering the region of South East Asia and Australasia region, for agreement for the publishing agreement of this book, as well as to Viradasarani Natarajan effort in preparation and advancement of this book and his team for its production at Scientific Publishing Services.

Contents

Part I Introduction to FPGA Technology and VHDL Language

1	Introduction to Field Programmable Gate Arrays (FPGA)	3
1.1	Introduction to FPGA	3
1.2	FPGA Architecture	4
1.2.1	General Structure of an FPGA Circuit	4
1.2.2	FPGA Routing Architectures	6
1.3	ASIC Versus FPGA Comparison	8
1.4	Design Methodology	9
1.4.1	Design Entry	9
1.4.2	Synthesis	9
1.4.3	Implementation and Programming	10
1.4.4	Design Tools	10
1.5	Summary	18
2	Basic VHDL Concepts	19
2.1	Introduction to Hardware Description Language	19
2.2	VHDL Design Units	21
2.2.1	Libraries and Packages	22
2.2.2	Entity	23
2.2.3	Architecture	25
2.3	VHDL Code Instructions	31
2.3.1	Concurrent Instructions	32
2.3.2	Sequential Instructions	34
2.4	Test Bench	38
2.5	Syntax Summary	39
2.5.1	Comments	40
2.5.2	Signal, Variables and Constants	40
2.5.3	Array	40
2.5.4	Operators	41
2.5.5	Attributes	41
2.6	Summary	42

Part II Design Digital Circuits (Simulation and Implementation on FPGA)

3 Combinational Logic Circuits	45
3.1 Introduction	45
3.2 Arithmetic Circuits	45
3.2.1 1-Bit Half Adder	46
3.2.2 1-Bit Full Adder	51
3.2.3 4-Bit Ripple Carry Adder	57
3.2.4 4-Bit Adder-Subtractor	63
3.3 Multiplexers	66
3.3.1 2-to-1 Multiplexer	66
3.3.2 4-to-1 Multiplexer	72
3.3.3 8-to-1 Multiplexer	77
3.4 Demultiplexer	80
3.4.1 1-to-4 Demultiplexer	81
3.4.2 1-to-8 Demultiplexer	86
3.5 Decoder	90
3.5.1 2-To-4 Decoder	91
3.5.2 3-to-8 Decoder	96
3.5.3 BCD-To-Decimal Decoder	101
3.6 Encoder	106
3.6.1 Binary Encoder	107
3.6.2 Priority Encoder	112
3.7 Summary	118
4 Sequential Logic Circuits	119
4.1 Introduction	119
4.2 D Flip-Flop	120
4.2.1 D Flip-Flop with Synchronous Reset	123
4.2.2 D Flip-Flop with Asynchronous Reset	125
4.2.3 Application of D Flip-Flop to Build a Clock Divider	126
4.2.4 Implementation and Validation on FPGA Platform	129
4.3 Registers	130
4.3.1 Serial-In Parallel-Out (SIPO) Shift Register	131
4.3.2 Serial-In Serial-Out (SISO) Shift Register	138
4.3.3 Parallel-In Parallel-Out (PIPO) Shift Register	141
4.3.4 Parallel-In Serial-Out (PISO) Shift Register	144
4.4 Counters	148
4.4.1 Asynchronous Counter	149
4.4.2 Synchronous Counter	157
4.4.3 Ring Counter	170
4.4.4 Johnson Counter	171
4.5 Summary	172

5 Finite State Machines	175
5.1 Introduction	175
5.2 Moore Machine	176
5.3 Mealy Machine	177
5.4 VHDL Implementation for Moore Machine	178
5.4.1 Moore Machine Described Using Three Processes	178
5.4.2 Moore Machine Described with Two Processes	180
5.5 VHDL Implementation for Mealy Machine	182
5.6 Examples: States Machines in VHDL	183
5.6.1 4-Bit BCD Counter FSM	184
5.6.2 Sequence Detector FSM	188
5.6.3 Parity Checker	198
5.7 Summary	205

Part III Laboratory Projects

6 Digital Projects Carried Out on the FPGA Platform	209
6.1 Introduction	209
6.2 Lab #1 Simple Calculator Design	210
6.2.1 Design of Arithmetic and Logic Unit (ALU)	211
6.2.2 Design of Multiplexers and Demultiplexers	214
6.2.3 Design of Registers	217
6.2.4 Design of Controller	218
6.2.5 Implementation of the Final Design of the Mini-calculator on the FPGA Platform	220
6.3 Lab #2 Digital Clock	222
6.3.1 Design of Digital Clock	224
6.3.2 Design of Frequency Divider	224
6.3.3 Design of Counters	226
6.3.4 Design of 7-Segment Display	229
6.3.5 Final Design of Digital Clock	230
6.3.6 Implementation of the Final Design of Digital Clock on the FPGA Platform	232
6.4 Lab #3 Design of Traffic Light Controller	233
6.4.1 Traffic Light Controller Operation	234
6.4.2 Design Traffic Light Controller in VHDL	235
6.4.3 Traffic Light Controller Implementation in the FPGA Platform	240
6.5 Lab #4 Vending Machine Controller	241
6.5.1 State Machine Diagram of Vending Machine	244
6.5.2 VHDL Implementation and Simulation of Vending Machine	247
6.5.3 Implementation and Validation on FPGA Platform	254

6.6	Lab #5 Control of a 4-Phase Step Motor (Speed and Position)	258
6.6.1	Laboratory Design Specifications	259
6.6.2	Stepper Motor Control: Speed and Position	260
6.7	Summary	276
Part IV FPGA Applications		
7	FPGA Applications in Renewable Energy Systems: Photovoltaic, Wind-Turbine and Hybrid Systems	279
7.1	Introduction	279
7.2	Neural Networks	280
7.3	Case Studies	281
7.3.1	Case Study 1: FPGA-Based Intelligent Photovoltaic Module Simulator	281
7.3.2	Case Study 2: FPGA-Based Implementation of Irradiance Equalization Algorithm for Reconfigurable Photovoltaic	288
7.3.3	Case Study 3: FPGA-Based Implementation of an MPPT Algorithm	291
7.3.4	Case Study 5: XSG-Based Implementation Control of Grid-Connected Hybrid System (PV-WT)	298
7.4	Summary	303
	References	303
	Appendix A: VHDL Codes of MLP Modules	305
	Appendix B: VHDL Codes of the Interconnection Reconfiguration Algorithm Between the PV Panels	311
	Appendix C: P & O Algorithm	325
	References	327

About the Authors



Dr. Bekkay Hajji is a Professor and the Director of the Renewable Energy, Embedded Systems and Information Processing Laboratory in the National School of Applied Sciences at the Mohammed 1st University. He received his DEA in microelectronics from the Institut National des Sciences Appliquées of Toulouse (INSAT, France) in 1996 and his Ph.D. from the Laboratory for Analysis and Architecture of Systems (LAAS-CNRS) Toulouse, France in 1999. After, he worked as a Process Engineer for ST-Microelectronics (2000–2003) and an Electrical Engineering Coordinator, National School of Applied Sciences Oujda (2010–2014). He is a scientific expert-evaluator affiliated with CNRST-Maroc for the period 2019–2021. During this period, he evaluated several COVID19 projects, PHC Maghreb 2021 projects and Multithematic Research & Development projects 2020. His research activities are dedicated to the development of Bio-chemical micro-sensors, embedded systems and renewable energy systems. He is editor of two Springer books and author and co-author of more than 50 papers in international refereed journals and papers in conference proceedings.



Dr. Adel Mellit is a Professor of Electronics at the Faculty of Sciences and Technology, Jijel University, Algeria. He received his M.S. Degree and PhD in Electronics from the University of Sciences Technologies (USTHB) Algiers in 2002 and 2006 respectively. Research interests of Prof. Adel Mellit focus on the application of artificial intelligence techniques, including machine learning deep learning in photovoltaic systems and microgrids. He has authored and co-authored 170 papers in international peer reviewed journals and papers in conference proceedings, six book chapters, two proceedings book and one book in Springer. He is the Director of the Renewable Energy Laboratory at Jijel University and is an Associate Member at the ICTP Trieste (Italy). He is serving as an Editor of IEEE journal of photovoltaic, a Subject Editor for the Energy Journal (Elsevier Ltd.) and he is a member of the editorial board of the Renewable Energy Journal (Elsevier Ltd.).



Dr. Loubna Bouselham received her M.Sc. and Ph.D. degrees in Electrical Engineering from National School of Applied Sciences at Mohammed first University in 2010 and 2019, respectively. Her current research interests include digital controls for energy management in smart buildings. In 2021, she joined the national school of architecture of Oujda as an assistant professor and a scientific research manager.

Part I

**Introduction to FPGA Technology
and VHDL Language**

Chapter 1

Introduction to Field Programmable Gate Arrays (FPGA)



Abstract In this chapter, the FPGA technology is introduced. First, the evolution history of FPGA circuits and the fields of its application are reviewed. Then, an overview of the basic FPGA architecture is presented. A comparison of the performances between FPGA circuits and their ASIC counterparts is then established. Finally, the whole design of FPGAs as well as the different steps required for the design and the associated tools are described.

1.1 Introduction to FPGA

FPGAs (Field Programmable Gate Arrays) are semiconductor devices based on a matrix of configurable logic blocks linked by programmable interconnections. Their particularity comes from their re-configurability. FPGAs can be reprogrammed after manufacture to meet the requirements of the desired applications or functionality.

The first commercial FPGA chip (XC2064) was introduced and delivered for the first time in 1985 by the American technology company (Xilinx). This device had a maximum capacity of 800 logic gates. The technology used was a $2\text{ }\mu\text{m}$ aluminum technology with 2 levels of metallization. Xilinx will be followed a little later, and never dropped, by its most serious competitor Altera which launched in 1992 the FPGA FLEX 8000. The maximum capacity reached 15,000 logic gates. Taking advantage of the technological evolution, FPGA circuits have not stopped developing and many manufacturers have since launched into the manufacture of FPGAs (Xilinx, Altera, Actel, Lattice, Motorola...). But the main manufacturers who hold an important place on the market are Xilinx and Altera (Altera was absorbed in 2015 by Intel).

Since the 2000s, the complexity of FPGAs has reached 6.8 billion transistors, the clock frequency exceeds one Gigahertz, and the most advanced technology today reaches 16 nm (Xilinx Virtex UltraScale+).

Given their inherent configurability and relatively low development cost, the impact of FPGA circuits on various engineering directions has grown with a rapid

pace. FPGAs are useful for a wide range of applications, such as medical imaging, aerospace and defense, automotive, video and image processing, etc.

1.2 FPGA Architecture

1.2.1 General Structure of an FPGA Circuit

An FPGA consists of a matrix of Configurable Logic Blocks (CLBs) surrounded by input/output (I/O) blocks. The set of CLBs and I/Os is connected by routing resources as depicted on Fig. 1.1.

The FPGA concept is based on the use of a LUT (Look-Up-Table) as a combinatorial element of the basic cell. This LUT can be seen as a memory that allows creating any logic function. An N -input LUT consists of 2^N configuration bits into which the required truth table is programmed during the configuration step. For area and delay purposes, the optimal number of inputs for LUTs has been found to be four. However, this number can vary depending on the targeted application by the vendor. Table 1.1 lists some devices of the XILINX FPGA Virtex family with the

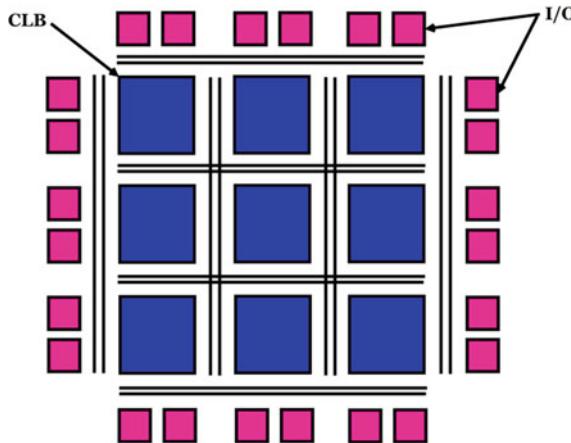


Fig. 1.1 Basic FPGA architecture

Table 1.1 XILINX FPGA Virtex family with the corresponding number of LUT cells

Family	Year	Technology	Max cells
Virtex4	2004	90 nm	200,000 LUT4
Virtex5	2007	65 nm	330,000 LUT6
Virtex6	2010	40 nm	760,000 LUT6
Virtex7	2012	28 nm	2,000,000 LUT6

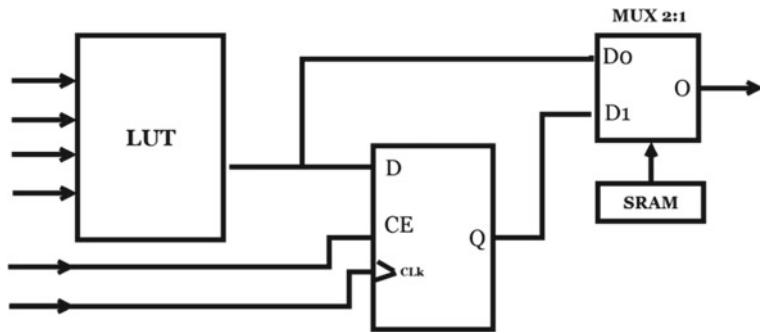


Fig. 1.2 Basic cell of an FPGA

corresponding number of LUT cells and the number of inputs for the LUTs. The following figure (Fig. 1.2) represents a basic cell of an FPGA for a LUT of 4 inputs. The D flip-flop allows the implementation of the function thanks to the memory block. The configuration of the 2 to 1 output multiplexer allows the selection of the two types of function: combinatorial or sequential.

The basic cells of an FPGA are arranged in rows and columns to form a Basic Logic Element (BLE). The BLEs are grouped in turn to form the CLB a. Each BLE can be connected to any input of the CLB or to another BLE as shown in Fig. 1.3. The logic elements are connected to each other and to the I/O through the interconnection network. This network is programmed in the same way as the logic elements with static RAMs. The set allows to completely configuring the FPGA whose CLB matrices can be organized from 8×8 to 128×128 .

In addition to reconfigurable logic, contemporary FPGAs contain numerous embedded blocks. Namely: Block RAMs used for efficient data storage and buffering. Indeed, FPGAs are composed of memory blocks that allow configuring and freezing the functionality of the circuit. There are several types of FPGAs (SRAM-based

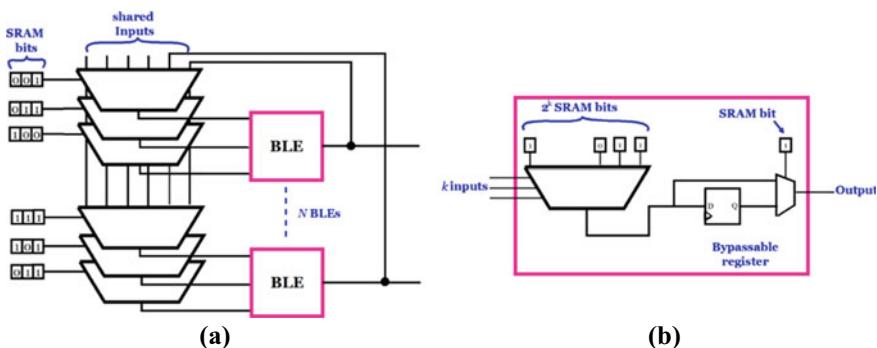
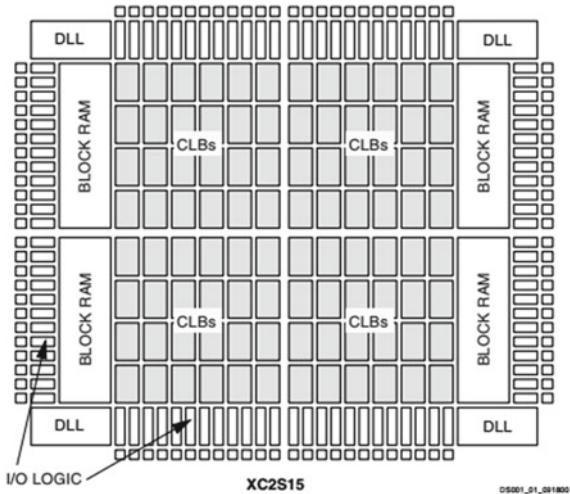


Fig. 1.3 a A configurable logic block b A basic logic element

Fig. 1.4 Spartan-II XC2S15 device architecture overview



FPGAs, Flash-based FPGAs and Antifuse-based FPGAs), classified according to the way they are programmed.

The majority of FPGAs use static memories. For example: Altera Cyclone II offer between 119 and 1,152 Kbits of on-board memory and a global clock network that includes 16 global clock lines driving the entire device. Xilinx Spartan-II (see Fig. 1.4) also includes a fully Digital Delay-Locked loop (DLL) for clock distribution delay compensation and clock domain control. It can eliminate the offset between the clock input buffer and the internal clock input pins throughout the device.

1.2.2 *FPGA Routing Architectures*

The FPGA routing interconnect provides different connections between FPGA components. Two types of interconnection topologies exist: Mesh and tree-based interconnections (see Fig. 1.5). These architectures differ in the way that the logic blocks are interconnected in the network.

1.2.2.1 **Mesh Architecture**

In the mesh architecture also called island style architecture, the CLBs are placed in a 2D grid organized in rows and columns and surrounded by interconnection resources (Fig. 1.5a). Generally, the routing interconnects in mesh architecture is composed of the following:

- Connection blocks responsible for connecting logical elements to the interconnection network (Fig. 1.6a).

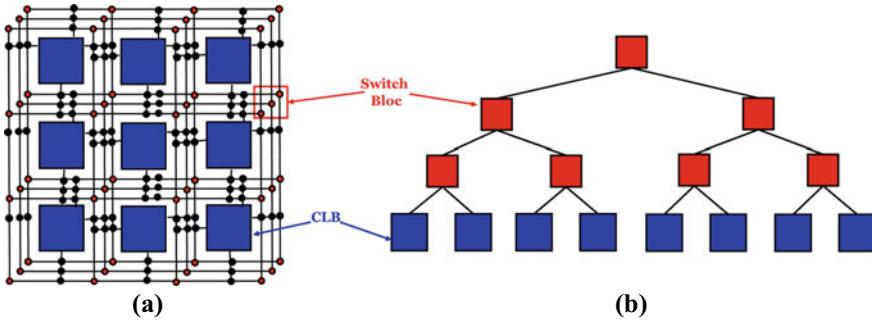


Fig. 1.5 **a** Mesh-based interconnect **b** tree-based interconnection

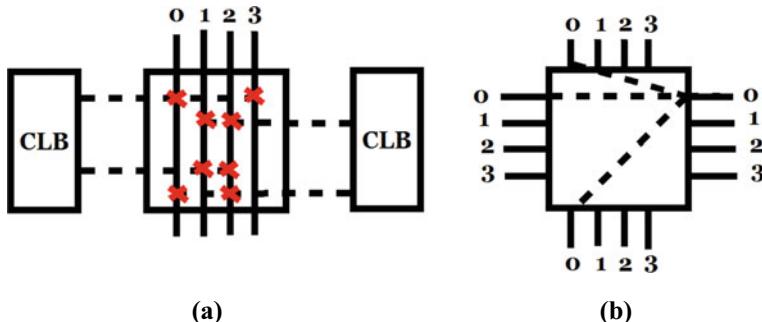


Fig. 1.6 **a** Block de connection **b** switch bloc

- Switching blocks are used to connect all the connection blocks (Fig. 1.6b).

The number of connections between the switch blocks and the connection blocks represents the routing channel of the FPGA. A routing channel is one of the most important architectural parameters. Its size (the number of wires) must be set before the manufacture of the FPGA by the designer and it must allow the routing of all the signals of the application used by the FPGA.

The mesh architecture is the most common FPGA architecture used commercially. However, 90% of the surface is occupied by the interconnection network, so the large number of switches traversed increases considerably the propagation delay.

1.2.2.2 Tree-Based FPGA

In the tree architecture, the logical blocks are grouped in separate clusters and recursively connected to form a hierarchical structure (Fig. 1.5b). Each cluster is composed of a local interconnection network in the form of a switch box. Compared to the mesh architecture, a connection between two logical blocks requires fewer switches. The

number of switches in a tree architecture increases logarithmically in contrast to the matrix architecture where the number of switches increases linearly with the Manhattan distance between two logical blocks. However, in tree architecture, the higher the number of levels in the architecture, the more difficult it is to generate the corresponding layout. In addition, the mesh model is more scalable because it has a regular structure that maximizes the use of the logic.

1.3 ASIC Versus FPGA Comparison

The ability to be reconfigured gives FPGA circuits' great flexibility for use in many designs. Their basic programming can be modified to make it specific to the circuit being used or to improve or resolve faults, which is impossible with ASIC (Application Specific Integrated Circuit) circuits.

An ASIC is a unique type of integrated circuit designed for a specific application and cannot be modified once created. Indeed, designers need to be completely sure of their design, especially if large quantities of the same ASIC are being manufactured. With the programmable nature of an FPGA, manufacturers can correct design errors and even send corrections or updates. This makes FPGAs ideal for prototyping. Manufacturers can create their prototypes in an FPGA so that they can test and revise them before moving to the production phase.

Another crucial advantage of FPGAs is their speed to market compared to ASICs. Their design flow is extremely simple: one can test an idea or concept, and then verify it on hardware without going through the long process of manufacturing an ASIC. Once the program is validated, it takes only a few minutes to implement.

Due to the above-mentioned advantages of FPGAs, they are constantly increasing their market share compared to ASICs. Although ASICs are generally less expensive than an FPGA solution, FPGAs are a worthwhile investment in the long term. This is due to their re-programmability and flexibility, which means they can be reused for other projects, despite the higher purchase price of an ASIC. However, the main advantage of FPGAs, namely their flexibility, is also the main cause of their disadvantage. The flexible nature of FPGAs makes them larger, slower and more power consumption than their ASIC counterparts. An FPGA requires 20 to 35 times more area, and a speed 3 to 4 times slower than the ASIC. Table 1.2 summarizes the main differences between FPGAs and ASICs.

Table 1.2 ASIC versus FPGA comparison

	FPGA	ASIC
Time to market	Fast	Slow
Design flow	Simple	Complex
Unit cost	High	Low
Power consumption	High	Low
Unit size	Medium	Low

1.4 Design Methodology

The FPGA design flow consists of different phases, as shown in the figure below (See Fig. 1.7). Each phase will be explored in detail in the following.

1.4.1 Design Entry

The design entry can be accomplished by two methods:

- *Use of a schematic*: the design is carried out with the help of gates and wires. Indeed, it is easily readable, understandable and they offer a better visibility of the whole system. But for large designs become difficult to maintain.
- *Use of Hardware Description Language (HDL)*: a fast language-based process that avoids the need to design lower-level hardware. It is more suitable for complex systems and is the most popular design input for FPGA designs. Two languages those are popular for FPGA designers: VHDL and Verilog (more discussed in Chap. 2).

Once the desired functionality of the circuit or system to be designed is specified, its behaviour can be modelled through functional simulation. This means that the circuit can be tested to verify its operation. If a problem is detected, the necessary changes are made to the circuit specification.

1.4.2 Synthesis

Once the design has been simulated correctly, we move on to circuit synthesis. A logic synthesis converts the design input into a netlist of basic logic gates such as LUTs, flip-flops, memories, DSP slices, etc. This phase usually goes through two different steps: logic optimization and technology mapping.

Logic optimization consists in simplifying the logic function of the design or any redundant logic is removed at this stage. The optimized circuit is then mapped while taking into consideration the speed, area or power constraints imposed by the designer.

FPGA synthesis is performed by dedicated synthesis tools such as: Cadence, Synopsys and Mentor Graphics.

1.4.3 Implementation and Programming

This phase consists of three steps: translate, map, and place & route. The first step consists of converting the input netlist into clusters of logic blocks that can be mapped into the physical logic blocks of the FPGA (Fig. 1.8). In the place phase, the encapsulation algorithm used must take into account all user-defined constraints, namely: pin assignment and position, timing requirements such as maximum delay or clock input period.

The next step is to allocate the available routing resources in the FPGA to the different connections between all logic blocks and I/O blocks. The routing algorithm in this phase should minimize the delay along the critical path and avoid congestion in the FPGA routing resources.

The placement and routing tools used in this step are provided by the FPGA vendors. At the output, a “Bitstream” file is provided that can be downloaded to the FPGA device.

The final step in the process is to upload the generated “Bitstream” file to the FPGA. Then, the resulting physical circuit undergoes final in-circuit testing.

1.4.4 Design Tools

To complete the design process, from circuit specification to simulation, synthesis and implementation, a CAD tool is used. At each stage, the CAD environment allows for simulations to be performed in order to validate each step of the implementation. At design entry, a behavioral simulation is performed to test the code and find logical errors. At synthesis, the functionality of the design can be verified using a functional simulation that ignores timing issues. At implementation, a simulation is performed to check timing performance, device resource usage, and power consumption. It takes more time and provides much more detail than previous simulations. The results of this simulation may require the designer to go back and make changes to the circuit specification, design constraints, or synthesis strategies.

In this book, we will mainly use ISE version 14.7 for Xilinx FPGAs and Quartus II 13 for Altera FPGAs. Moreover, a range of CAD tools is offered by both Xilinx and Intel and can also be used in a similar way. The underlying ideas are exactly the same, only the corresponding software environments are different.

To summarize the design flow shown in Fig. 1.7, consider all of the steps that must be completed in Quartus in order to implement a “xor” gate:

- Creation of a project

We start by creating a new project (file > new project wizard) by specifying its name, location and the name of the top-level entity (see Fig. 1.9). Please note that the name of the project and the name of the top-level entity must be the same. Then the characteristics of the target FPGA are selected (see Fig. 1.10). In our case,

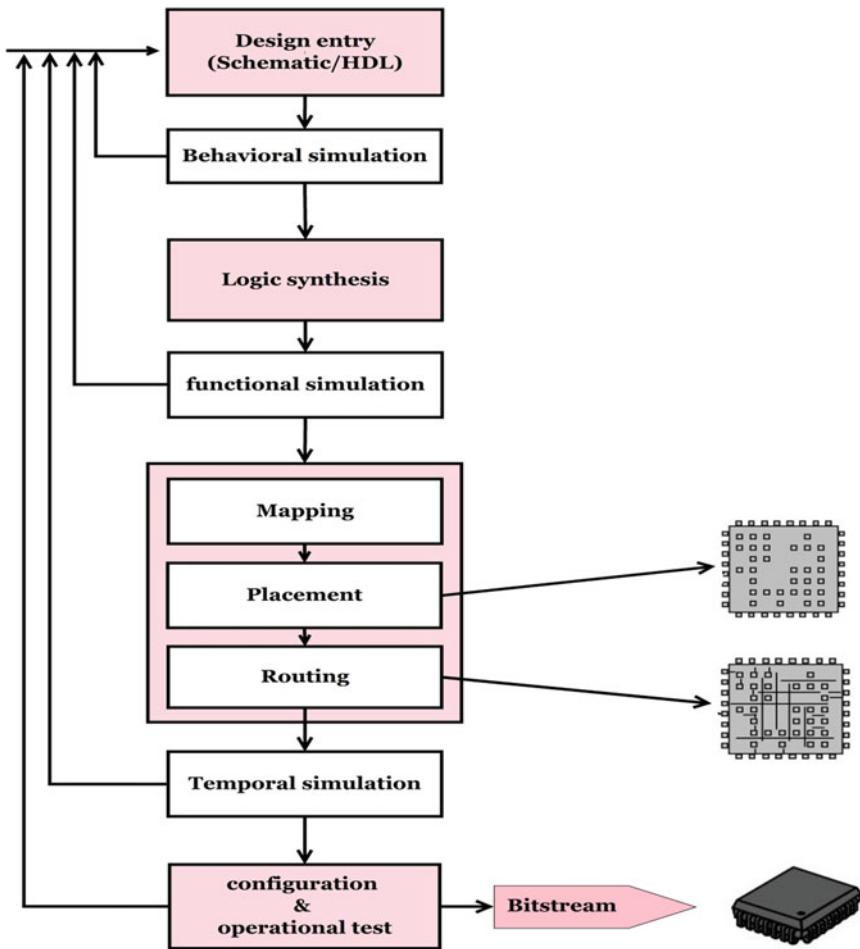


Fig. 1.7 Simplified FPGA design flow

we have selected “Cyclone II” as the FPGA family and also indicated the device EP2C70F896C6. These data can be found in the documentation that accompanies the device and are also written on the appropriate chip.

- Circuit description

Once the project is created, the circuit can be described. In the “FILE” menu, we choose “NEW”. Window offering different types of description appears. Since we are interested in VHDL language, we choose “VHDL File” (see Fig. 1.11) and we write the VHDL code in the dedicated area. The VHDL code can also be edited in any text editor except that it must be saved with the extension (.vhd) and added to the project (Project > add/Remove file in Project).

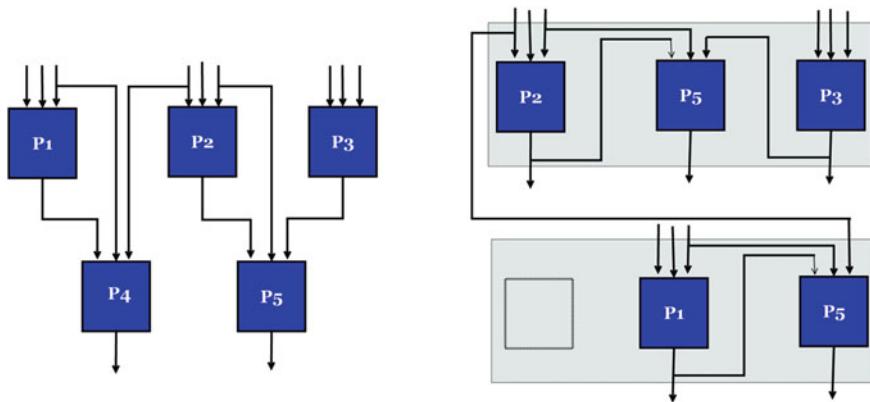


Fig. 1.8 An example of cluster packing

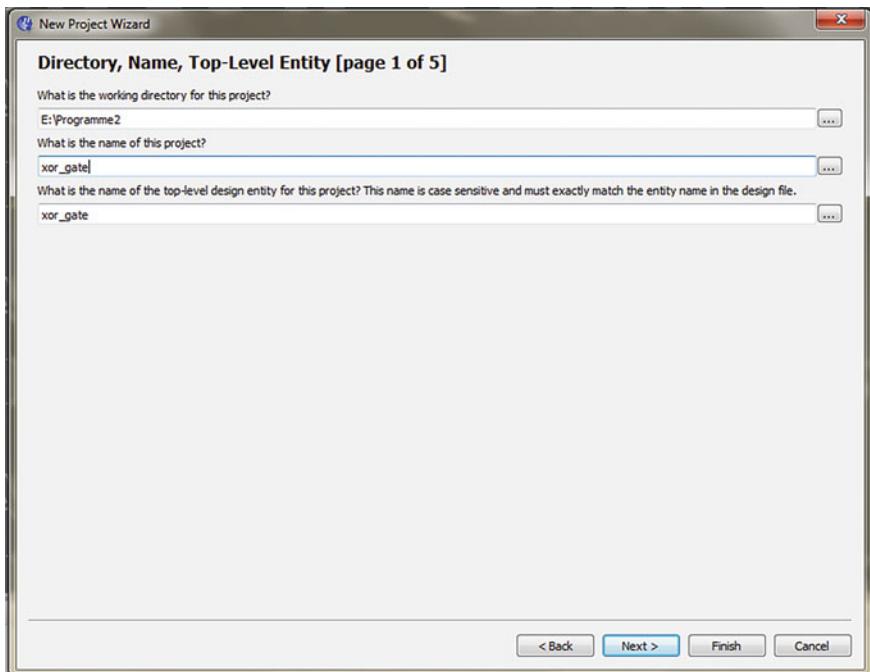


Fig. 1.9 Creation of new project under Quartus II

- Compiling the program

After saving the VHDL code in the project directory and with the same name as the Entity, the code can be compiled (processing > start compilation). The compiler generates a report indicating all errors and / or warnings found, as shown in Fig. 1.12.

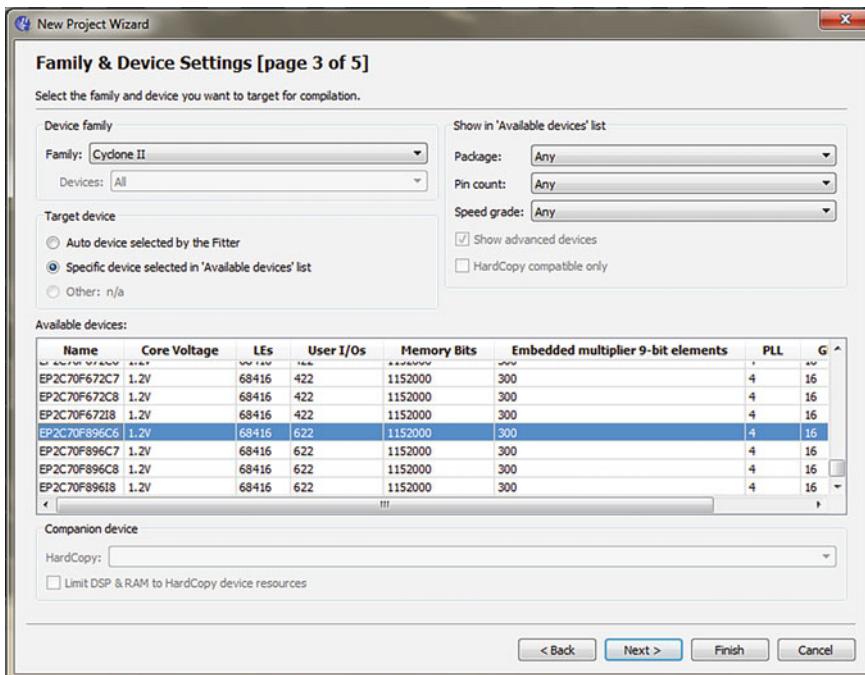


Fig. 1.10 Choice of device family under Quartus II

Once the compilation is finished without any indication of errors, the description is synthesizable in a layout that can be viewed on the menu: “Tools > NETlist Viewers > RTL Viewer” (see Fig. 1.13).

- Pin assignment

At the next step, I/O of the circuit described are assigned to the pins of the used FPGA device (Assignment > pin planner). At pin planner, the different nodes with their direction (Input/output) are listed. We choose the appropriate location of each node. In our case, the inputs take the addresses of two switches (PIN_N25, PIN_P25) and the output takes the address of an LED (PIN_AE23) on the FPGA board (see Fig. 1.14).

- Functional Simulation

To perform a simulation, the test vectors must be defined on the menu: File > New > Vector Waveform (see Fig. 1.15).

First, we introduce the I/O of the circuit by going to the menu: Edit > Insert > Insert Node or Bus. The window of Fig. 1.16 is displayed. By clicking on “node finder” another window is displayed where the different available nodes are listed. Just select the desired node for the simulation and transfer it to the right tab (Fig. 1.16).

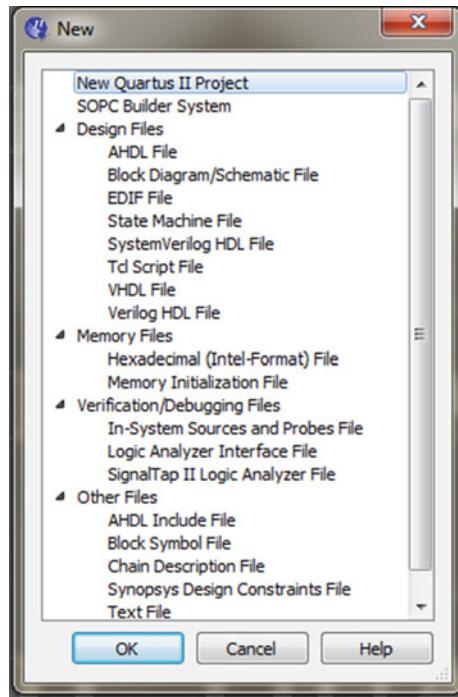


Fig. 1.11 Choice of language Quartus II description under

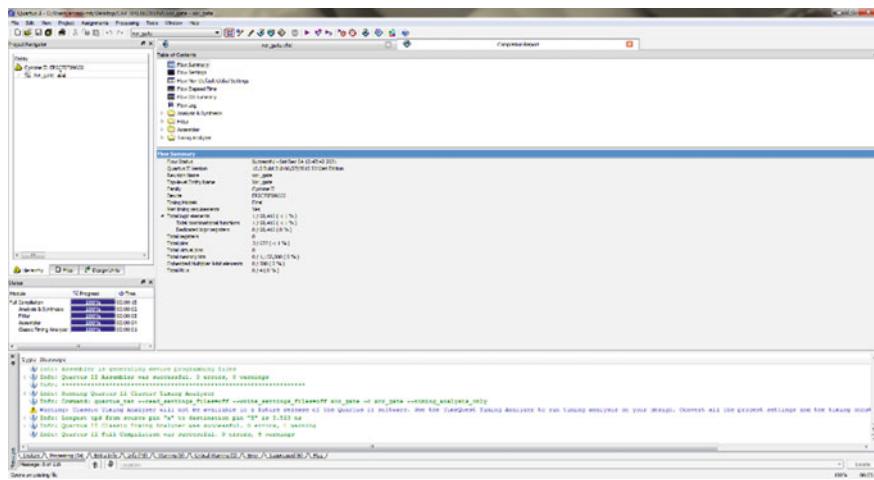


Fig. 1.12 Compilation report

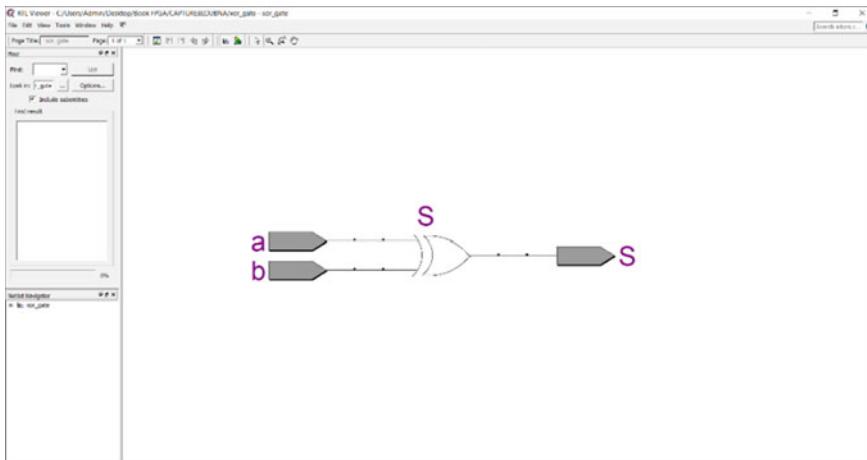


Fig. 1.13 Synthesizable layout after compilation

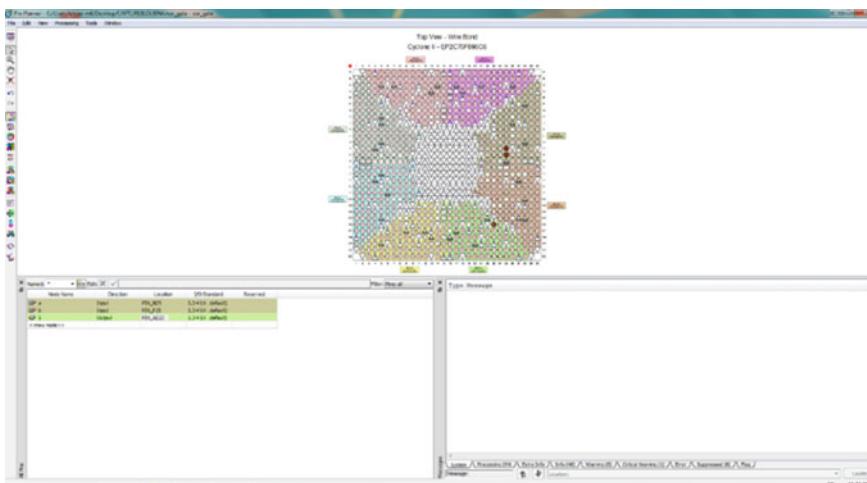


Fig. 1.14 Pin planner for assignment

Once the I/O are on the editor window, we define the test vectors by drawing up a chronogram and specifying the simulation time (Edit > End Time) (see Fig. 1.17).

Finally, the simulation can be executed by going to the: Processing > Simulator tool menu > Functional simulation mode. Then, we go to the Processing > Generate Functional Simulation Netlist > start.

The results of simulation are shown on Fig. 1.18.

Fig. 1.15 Vector waveform of Qartus II

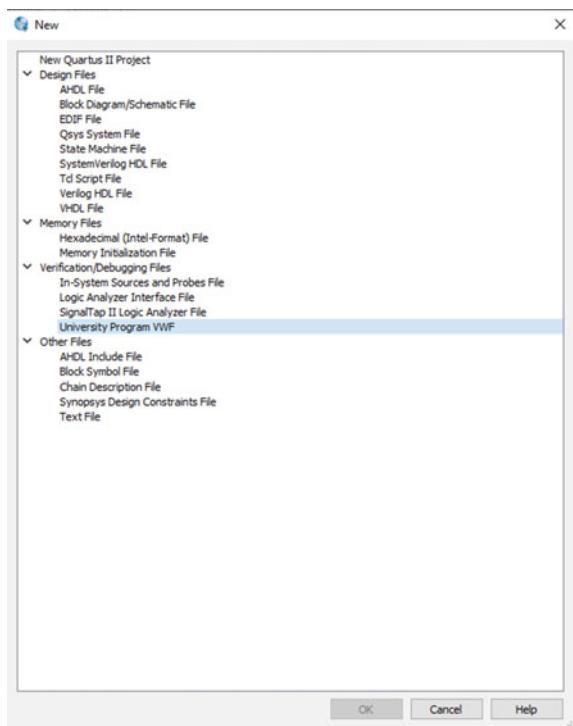
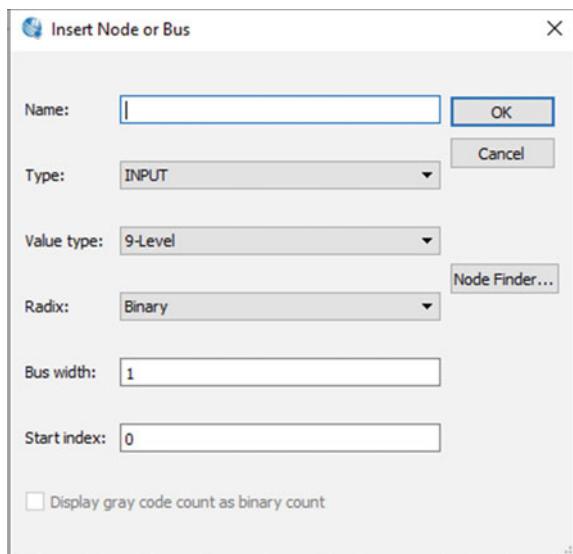


Fig. 1.16 Node insertion in Qartus II



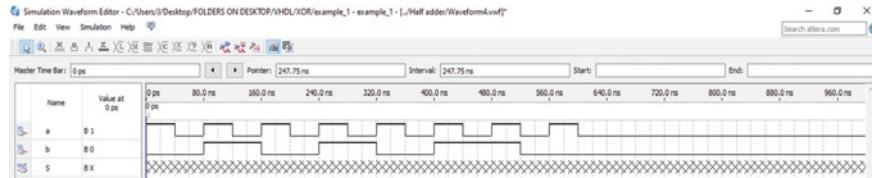


Fig. 1.17 Test vectors before simulation under Quartus II



Fig. 1.18 Simulation results under Quartus II

- Programming and configuration of the FPGA circuit

The final step is to transfer the program to the FPGA board; we start in the “Tools > Programme” menu, then the window below appears. By clicking on “start” the program is transferred to the card while viewing the progress of the transfer (see Fig. 1.19).

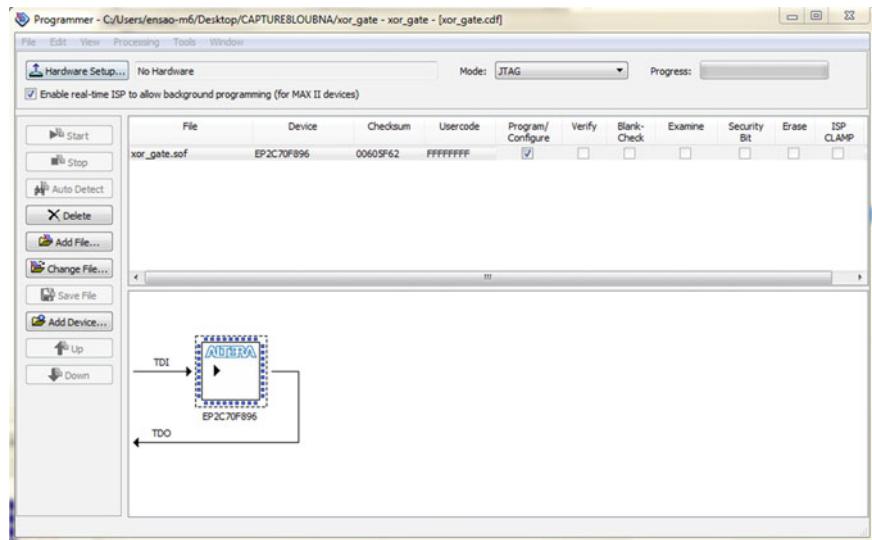


Fig. 1.19 Programming step under Quartus II

1.5 Summary

A brief overview of the different aspects related to FPGAs is presented in this chapter. In summary:

- An FPGA circuit is a flexible solution well adapted to economic constraints. Reduced time to market with performance and capabilities to integrate very complex applications.
- The internal architecture of FPGAs differs from one manufacturer to another and even between the different ranges of the same manufacturer. Generally, it is composed of a matrix of configurable logic blocks surrounded by input and output blocks.
- Design and simulation are the two main steps in the design flow of an FPGA. The design tasks allow going from one description to another to arrive at the configuration “bitstream” file.
- At each stage of the design process, the CAD environment allows simulations to be carried out in order to validate each stage of the implementation.

Chapter 2

Basic VHDL Concepts



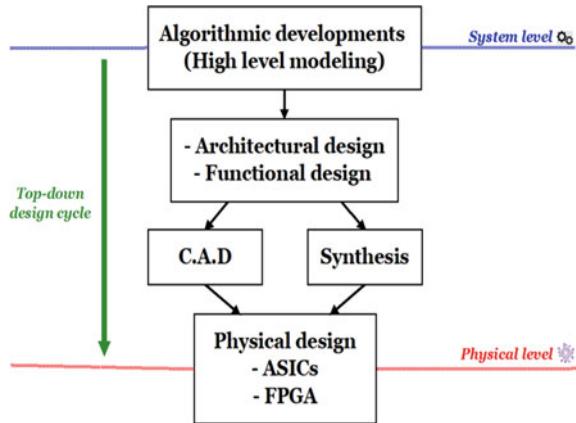
Abstract In the previous chapter, we showed that HDLs are used for both simulation and synthesis. With an HDL, the specification of digital systems is much faster than a full schematic drawing. The debugging cycle is also much faster, as modifications require code changes instead of laborious schematic rewiring. In this chapter, we will explore the modelling of digital systems with the VHDL language by introducing these basic concepts. We start with the structure of a VHDL code and then we move to data manipulation with the different instructions (combinatorial, sequential). We end this chapter with a syntax summary.

2.1 Introduction to Hardware Description Language

With the large number of logic functions integrated on the same chip, the VLSI (Very-Large-Scale Integration) design becomes more and more complex. This number of functions is yearly quadrupled according to the Moore's law. Indeed, a schematic design in this case becomes a laborious task; Boolean equations need to be simplified and finite state machines need to be translated to logic gates, which makes the operation prone to errors. Therefore, shifting from low-level to more abstract levels design was obvious, where the logical function is defined by a Hardware Description Language (HDL), while a Computer-Aided Design (CAD) tool is responsible for producing the optimized gates.

Unlike a programming language, an HDL doesn't target an execution, even if there is an associated simulator. It is applied to the different stages of the iterative design process, starting from an algorithmic description until a logical synthesis as well as a formal verification of the design accuracy. The main stages of a digital system design flow are summarized in Fig. 2.1. For industry designers, System Verilog and VHDL are the famous HDLs. Most modern CAD tools that perform simulation, synthesis and layout, support both languages. The main differences between the two languages are: RTL models, the syntax and the coding style (VHDL is derived from ADA, Verilog from C).

Fig. 2.1 Digital system design flow



This book focusses on VHDL (Very high speed integrated circuit (VHSIC) Hardware Description Language). All the codes presented in the next chapters are programmed in VHDL language. VHDL was established in 1980 following the need of the US Department of Defense (DoD) to unambiguously describe the operation of ASICs in their microelectronic equipment. Thus, the VHDL was originally used for the ASICs documentation. In order to reflect the industry evolution; the language has been revised for several times. The main VHDL revisions are summarized in Table 2.1.

Like any HDL language, VHDL allows the synthesis and simulation of circuits. During simulation, inputs are applied to a module and the outputs are checked to ensure that the module is functioning properly. During synthesis, the VHDL code is

Table 2.1 Summary of the VHDL history

1983	The definition of the VHDL was ceded by DoD to three companies: INTERMETRICS, IBM and TEXAS INSTRUMENTS
1985	First official version of VHDL (Version 7.2)
1986	VHDL is given to IEEE to make it a standard
1987	First VHDL version corresponding to IEEE standards (IEEE 1067_1987) where a considerable range of data types have been incorporated, including logic, character, arrays, string and numeric
1993	Development of the IEEE 1164 Standard which defines the std_logic type with 9 possible states
1999	Development of the IEEE 1076.1 Standard or VHDL-AMS by IEEE Automation standard committee in order to respond to various electronic problems
2007	Incorporation of the VHPI interface to allow tools to programmatically access a VHDL model before and during simulation. (VHDL project 3.0 approved by Accellera's VHDL technical committee)
2008	Approval of the VHDL-2008 version which fixed the various problems discovered during the trial period of 3.0 version

translated into a netlist describing the hardware that implements the desired functionality. In this chapter, the fundamental concepts of the VHDL are treated concisely by defining the description style (grammar and vocabulary) and the VHDL code structuring through several examples.

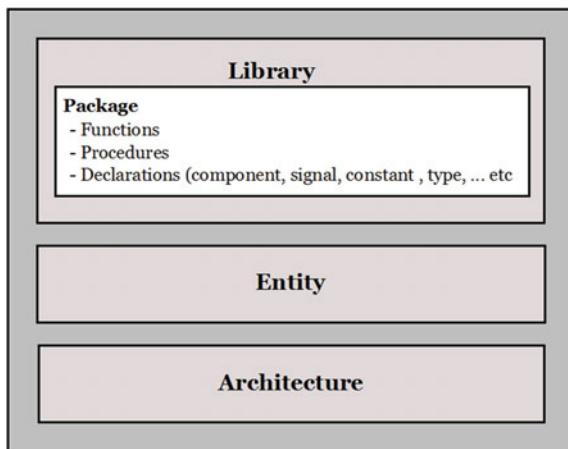
2.2 VHDL Design Units

As depicted on Fig. 2.2, the general structure of a VHDL code consists of three main units:

- Library/package needed for the design
- Entity where circuit inputs/outputs are specified
- Architecture where the operation of the circuit is described.

Put differently, the VHDL makes a distinction between the external and the internal view of the circuit, the interface is described with the entity while its implementation is defined by the architecture. Let's consider the example of Listing 2.1: the entity is declared with the name “example_1”. a and b are input signals of type “std_logic” and S is an output signal of type also “std_logic”. The architecture is called “circuit”. The first two lines identify the reference libraries used in the design. Each unit is discussed in detail in the following sections.

Fig. 2.2 Basic sections of VHDL code



2.2.1 Libraries and Packages

Any VHDL description used for synthesis needs libraries. The reference library to be used is identified with the keyword ‘LIBRARY’ followed by the library name:

Library library name;

The most frequently used libraries are “IEEE”, “std” and “work”. Each library contains design units, i.e. packages. All VHDL data types, subtypes and corresponding logical and arithmetic operations and some other functions are defined in specified packages. The access to a package is done by using the “USE” clause followed by the package name and the suffix “all”. The corresponding syntax is as follows:

```
use.library name.package name.all;
```

In the example of Listing 2.1, the 2nd line identifies the IEEE library as reference library while the 3rd line indicates that “std_logic_1164” is used as a specific package. These two lines are necessary to make a package visible to the design and accessible by an entity. Table 2.2 summarizes the fundamental libraries and packages required for basic operations.

Table 2.2 Libraries and packages

Library	Package	Specification
IEEE	Std_logic_1164	Defines the standard for the description of 9 interconnection data types used in the VHDL language including the STD_LOGIC and STD_LOGIC_VECTOR types
	numeric_std	Defines the types SIGNED and UNSIGNED and corresponding operators, having STD_LOGIC as the base type
	numeric_bit	Same as numeric_std, but with BIT as the base type
	std_logic_arith	Defines the types SIGNED and UNSIGNED and corresponding operators. This package is partially equivalent to numeric_std
	std_logic_unsigned	Introduces functions that allow arithmetic, comparison, and some shift operations with signals of type STD_LOGIC_VECTOR operating as unsigned numbers
	std_logic_signed	Same as std_logic_unsigned, but operating as signed numbers
	fixed_pkg	Defines the unsigned and signed fixed-point types UFIXED and SFIXED and related operators
	float_pkg	Defines the floating-point type FLOAT and related operators
std	standard	Contains several data type definitions (BIT, INTEGER, BOOLEAN, CHARACTER, etc.) and respective logic, arithmetic, comparison, shift, and concatenation operators
	textio	A resource package for text and files
work	‘Set by user’	Refers to the current working library, in which all of the entity and architecture body descriptions are assumed to be held

2.2.2 Entity

The entity represents the black box image of the module to be designed with the Input/Output (I/O) connections visible from the outside, as in Figs. 2.3 and 2.4. The main part of the declaration of an ENTITY is PORT. It lists the different interface ports connections and provides information about the direction (I/O) of ports, data type, and bit width (See Fig. 2.5).

The entity declaration generally takes the following form:

```
ENTITY entity_name IS PORT (
    port_name: port_mode signal_type;
    port_name: port_mode signal_type; ...);
END entity_name;
```

Fig. 2.3 Design flow



Fig. 2.4 XOR gate

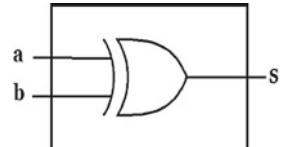
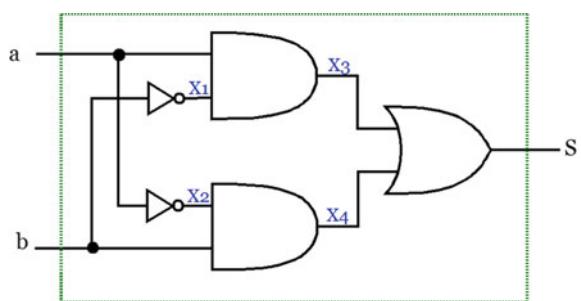


Fig. 2.5 Xor gate scheme based on the AND, OR and or logic gates



The port mode indicates the direction of the information; five modes are supported in VHDL coding:

1 2 3 4 5 6 7 8 9 10 11 12 13	<pre>-----LIBRARY----- libraryIEEE; useIEEE.std_logic_1164.all; -----ENTITY----- entity example_1 is port(a,b : instd_logic; S: outstd_logic); end example_1; -----ARCHITECTURE----- architecture circuit of example_1 is begin S <= a xor b; end circuit;</pre>
---	--

Listing 2.1 XOR gate VHDL code

- IN: the port can be read but not updated. IN ports are mainly used for control signals and unidirectional data signals.
- OUT: the port can be updated but not read. As OUT ports cannot be read, they are used as outputs of the entity, but not for internal feedback to the entity.
- INOUT: This is a bi-directional port can be read and updated. It allows internal feedback of signals within the entity.
- BUFFER: this mode is similar to the INOUT mode in that it can be read and written to the entity. BUFFER mode is used when a signal is sent but it must also be used internally.
- LINKAGE: this mode can be also read and written. It is useful when VHDL entities are connected to non-VHDL entities.

Signal_type specifies the type of data or information that can be communicated. Almost all VHDL signals and ports use the std_logic (one bit) and std_logic_vector (bus) data types. In addition, VHDL has several predefined data types. Some of the predefined types are summarized in Table 2.3. Note that VHDL is a strongly typed language: signals and variables of different types cannot be mixed in the same assignment instruction. For example, the statement ‘S <= a xor b’ is only valid if S, a and b have the same data type.

For the previous example, the ENTITY is declared under the name ‘example_1’. Its meaning is as follows: the circuit has three I/O ports, of which two are inputs (a and b, mode IN) and the other is an output (S, OUT mode). The type of these three signals is std_logic. Before the PORT clause, no declaration is allowed, unless it is a declaration of a generic component. The GENERIC clause allows the specification of generic constants, which can be easily modified or adapted to different applications. For example, the GENERIC declaration for the entity below specifies two parameters,

Table 2.3 Predefined data type in VHDL

Data type	Specification
bit	'0' or '1'
Boolean	FALSE or TRUE
Integer	an integer in the range— $(2^{31} + 1)$ to $(2^{31} - 1)$
Real	Floating point number in the range 1.0E38–1.0E38
Character	Any legal VHDL character including upper and lowercase letters, digits, and special characters
Time	An integer with units fs, ps, ns, μ s, sec, min or hr

called X and Y. The first is of type std_logic_vector and has a value of “1111”, while the second is of type integer and has the value of “9”. Therefore, whenever X and Y are encountered in the code, they are automatically assigned to the values “1111” and 9.

```
ENTITY example_2 IS
  GENERIC (X: std_logic_vector (3DOWNT00):= "1111";
  Y: integer:=9);
  PORT(...);
END example_2;
```

2.2.3 Architecture

The circuit functionality and the relationship between the ports of the entity are described by the architecture. A simplified syntax is listed below:

```
ARCHITECTURE architecture_name OF entity_name IS
  ...
  [declarative_part]
  ...
  BEGIN
  ...
  [statements_part]
  ...
  END architecture_name;
```

The declaration of the architecture starts with its name ‘architecture_name’, followed by the name of the entity ‘entity_name’ with which the architecture is associated. In the declarative part, signals and components that are used in the architecture

are declared while the statements' part contains declarations that describe how the module works. In VHDL, the architecture is part of a concurrent domain, the function of this architecture is described by a set of concurrent instructions executing asynchronously at a macroscopic level (the order of writing does not matter). This description can be structural (the black box is considered as an interconnection of other black boxes), behavioral or “data flow” (we give the algorithm that the black box performs). A hybrid architecture combining the three description styles is possible.

2.2.3.1 Data Flow

The circuit is described at the data flow level by giving functional description of Boolean equations. For the previous example, the xor gate is coded as data flow by the equation:

$S \leftarrow a \text{ xor } b$ (line 12, Listing 2.1).

This line assigns to the signal S a value ($a \text{ xor } b$). In this description, no time limit is associated. However, a timed description is also legal by using the AFTER clause:

$S \leftarrow a \text{ xor } b \text{ after } 1 \text{ ns};$

(The right-hand side of the assignment is assigned to the left-hand side after delta time interval of 1 ns).

2.2.3.2 Behavioral Description

A behavioral description is a conditional description. It provides an algorithm that models the operation of the circuit from its truth table. Listing 2.2 shows an example of behavioral architecture description for the xor gate. The difference between the dataflow and behavioral description is that the behavioral description uses the PROCESS statement. A process is a part of the description in which the instructions are executed sequentially, i.e. one after the other. The statement is triggered by one or more changes of state of the variables in the sensitivity list. The process syntax generally takes the following form:

```
Process_name PROCESS (sensitivity list)
BEGIN
    --Process instructions ;
END PROCESS Process_name;
```

Note: The Process_name is optional, but it can be very useful to identify a process among others.

2.2.3.3 Structural Description

Another way to describe the implementation of an entity is to specify how it is composed of subsystems. A structural design can be considered as a textual description of a schematic diagram that interconnects these subsystems.

Let's consider the same example above. As depicted in Table 2.4 and Fig. 2.5, the XOR gate can be decomposed into primitive gates derived from its Boolean equation

$$S = a \text{ xor } b = \bar{a}b + a\bar{b}$$

First, each logical gate used in the structural design must be described. Listings 2.3–2.5 provide a behavioral description of these logic gates. Then, these logic gates are instantiated and connected to each other to be used in the design of the XOR gate. Listing 2.6 shows the structural description of the XOR gate design. At the structural architecture level, before it's BEGIN:

- The internal signals of the architecture that link the instantiated modules are defined (line 13, Listing 2.6). The internal signals have a type but not a mode.
- Each module used in the structural description is declared (Listing 2.6, lines 15–25). The format is the same as for the entity except that the ENTITY keyword is replaced by the COMPONENT keyword

Table 2.4 Xor gate truth table

a	b	S
0	0	0
0	1	1
1	0	1
1	1	0

1	architecture behav_desc of exam-
2	ple_1 is
3	begin
4	process (a,b)
5	begin
6	if a = b then
7	S <= '0';
8	else
9	S <= '1';
10	end if;
11	end process;
	end behav_desc;

Listing 2.2 Behavioral description of XOR gate

Once components are declared, they can be instantiated (Listing 2.6, lines 28–32). Each instance of the component is unique, and a component can have multiple instances, as in the case of the not_gate and and_gate, they are instantiated twice. The connection of the ports of each component instance to the internal signals is specified by the port mapping process. For example, or_gate is an instance of the xor_gate entity; it has its x port connected to the X3 signal, its y port connected to the X4 signal and its z port connected to the output port S.

1	libraryieee;
2	use ieee.std_logic_1164.all;
3	
4	entity and_gate is port (x,y : in std_logic;
5	z : out std_logic);
6	end and_gate;
7	architecture and_behav of and_gate is
8	begin
9	z<=xandy;
10	end and_behav;

Listing 2.3 Behavioural description of AND gate

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity or_gate is port ( x,y : in std_logic;
5 z : out std_logic);
6 end or_gate;
7
8 architecture or_behav of or_gate is
9 begin
10 z <= xory;
11 end or_behav;
```

Listing 2.4 Behavioural description of OR gate

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity not_gate is port (
5 x : in std_logic;
6 z : out std_logic);
7 end not_gate;
8
9 architecture not_behav of not_gate is
10 begin
11 z <= notx;
12 end not_behav;
```

Listing 2.5 Behavioural description of NOT gate

```

1  Library ieee;
2  Useieee.std_logic_1164.all;
3  ----- ENTITY DECLARATION-----
4  entity xor_gate is port (
5    a,b : instd_logic;
6    s : outstd_logic);
7  end xor_gate;
8  ----- ARCHITECTURE DECLARATION-----
9  architecture xor_structural of xor_gate is
10
11    signal X1,X2,X3,X4: std_logic;
12    component and_gate
13      port ( x,y : instd_logic; z: out std_logic);
14    end component;
15
16    component or_gate
17      port ( x,y : instd_logic;z : outstd_logic);
18    end component;
19
20    component not_gate
21      port ( x : instd_logic;z : outstd_logic);
22    end component;
23
24    begin
25      C1: not_gate port map (a,X2);
26      C2: not_gate port map (b,X1);
27      C3: and_gate port map (a,X1,X3);
28      C4: and_gate port map (b,X2,X4);
29      C5: or_gate port map (X3,X4,s);
   end xor_structural;
```

Listing 2.6 Structural descriptions of xor gate

2.2.3.4 Hybrid Description

A hybrid design combining several design techniques is possible. Often a behavioral description is coupled with a structural description. Listing 2.7 depicted structural architecture of the XOR gate where instead of using not_gate component as in the previous example, the inverter gate is described by its Boolean equation (Listing 2.7, lines 28–29).

```

11  architecture xor_structural of xor_gate is
12
13      signal X1,X2,X3,X4: std_logic;
14
15      component and_gate
16          port ( x,y : instd_logic; z: out std_logic);
17      end component;
18
19      component or_gate
20          port ( x,y : instd_logic;z : outstd_logic);
21      end component;
22
23      component not_gate
24          port ( x : instd_logic;z : outstd_logic);
25      end component;
26
27      begin
28          X2 <= nota;
29          X1 <= notb;
30          C3: and_gate port map (a, X1, X3);
31          C4: and_gate port map (b, X2, X4);
32          C5: or_gate port map (X3, X4, s);
33      end xor_structural;

```

Listing 2.7 Hybrid description architecture of xor gate

2.3 VHDL Code Instructions

VHDL architecture is described in form of concurrent instructions or sequential instructions depending on the type of circuit to be designed.

VHDL code is inherently concurrent (parallel) where the order of instruction placement is not important. However, only instructions placed within a “PROCESS”, “FUNCTION” or “PROCEDURE” are executed sequentially.

Concurrent instructions include concurrent signal assignment, PROCESS and COMPONENT instantiations. They are intended only for the description of combinational circuits whose outputs depend only on the inputs. On the other hand, a sequential circuit conditioned by a clock and a sequence of operations to be respected, requires sequential instructions for its description in VHDL.

Note that the sequential instructions as they are used for designing sequential circuits, they are also used for designing combinatorial circuits.

2.3.1 Concurrent Instructions

As mentioned above, VHDL offers a complete set of concurrent instructions, namely:

- A simple assignment of signals using the arrow “=>”, such as the case of the description in Listing 2.1.
- A “process” is a concurrent instruction that contains only sequential instructions.
- Instantiation of a component in the case of a structural description.
- Conditional assignment and selective assignment.
- The “GENERATE” instruction.

The last two instructions (d–e) are more detailed below.

2.3.1.1 Conditional Assignment: WHEN/ELSE Statement

The “when/else” statement is one way to describe the concurrent signal assignments. In general; its syntax is as follows:

```
output_signal <= value_1 when condition_1 else
value_2 when condition_2 else
...
value_n;
```

The expressions after the “when” clauses are evaluated successively until a true expression is found. The expression after the final “else” is unconditional and will be executed if none of the preceding expressions are true. Each condition is a Boolean expression i.e., True or False. Note that multiple conditions are accepted are can be grouped using AND, OR, and NOT. The Listing 2.8 presents a concurrent description of the XOR gate using the when-else statement and based on its truth table of Fig. 2.6.

11 12 13 14 15	ARCHITECTURE Concurrent_state_1 OF Xor_gate IS BEGIN S<= '0' WHEN(a AND b)="00"OR(a AND b)="11" ELSE '1'; END Concurrent_state_1 ;
----------------------------	--



Fig. 2.6 Series of N inverter gates

Listing 2.8 XOR gate using “when/else” statement

2.3.1.2 Selective Assignment: WITH/SELECT Statement

WITH/SELECT is another concurrent statement. Based on several possible values of the input signals, a selected signal is assigned to the output signal. A simplified syntax for this statement is presented below:

```
with control_expression select
    output_signal <= value_1 when option_1,
                           value_2 when option_2,
...
                           value_n when others;
```

The WITH/SELECT statement requires that all input values be covered, for which the keyword OTHERS is often helpful. The Listing 2.9 presents a concurrent description of the XOR gate using with/select statement and based on its truth table of Fig. 2.6.

As shown in the Listing 2.9, SELECT allows the use of multiple values (instead of multiple conditions), which can be grouped with “|” (means “or”).

2.3.1.3 The GENERATE Statement

VHDL provides the GENERATE statement to have a section of code repeated a number of times. Two ways to apply the GENERATE statement:

11	ARCHITECTURE Concurrent_state_2 OF Xor_gate IS
12	BEGIN
13	WITH (aANDb) SELECT
14	S <= '0' WHEN "00" "11",
15	'I' WHEN OTHERS;
16	END Concurrent_state_2 ;

Listing 2.9 XOR gate using “with/select” statement

- Conditional GENERATE which includes an IF statement in the GENERATE loop. A simplified syntax is shown below:

```
label : IF (boolean_expression) GENERATE
concurrent_statements;
END GENERATE [label];
```

- Iterative GENERATE (FOR-GENERATE) is used to create multiple instances of a section of code.

```
label:FOR identifier IN range GENERATE
concurrent_statements;
END GENERATE [label];
```

The example below illustrates the use of this instruction. We assume that we want to connect a series of N inverter gates as shown in Fig. 2.6. In this case, using a structural description of the instantiated components as in the Listing 2.6 is not practical, especially if the N number of components is large. Therefore, the iterative form of the “generate” instruction is used, where the number N of gates to be connected is specified as described in Listing 2.10. N can be a constant or a generic parameter.

2.3.2 *Sequential Instructions*

The background of a VHDL code is usually concurrent, but it is often meaningful to use the sequential domain to design certain types of circuits. VHDL has a very complete set of sequential instructions, but these instructions can only be used within a process or subprogram (Function/Procedure). The main sequential statements are listed and described below.

2.3.2.1 **IF Statement**

The “if” statement allows the conditional execution of sequences of instructions. Its basic syntax is presented as follows:

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity not_gate_series is
5     generic( N:integer:=9);-- N=9 i.e. 8 gates are connected
6     port (
7         Input : in std_logic;
8         Output : out std_logic);
9 end not_gate_series;
10
11 architecture struct_series of not_gate_series is
12 signal G: std_logic_vector(1 to N); --signal memorizing the N nodes
13 component not_gate
14     port ( x : in std_logic; z : out std_logic);
15 end component;
16 begin
17
18     --instance of the components
19     inverter: for i in 1 to N generate
20         C1: not_gate port map (G(i),G(i+1));
21     end generate inverter;
22
23     G(1)<= Input;--the first node is connected to the input
24     Output<=G(N);--the Nth node is connected to the output
25
26 end struct_series;

```

Listing 2.10 Series of 8 inverter gates using “GENERATE” statement

```

If Boolean_condition then
    sequence_of_instructions_1
else
    sequence_of_instructions_2
end if;

```

The sequence_of_instructions_1 will be executed only when the Boolean condition is true, while the sequence_of_instructions_2 will be executed only when the Boolean condition is false. In some cases, several levels of nesting may be necessary. In order to reduce them, the “elsif” form is used. This form improves the readability of the program in structures with a high number of branches. Example syntax of the “if” statement with “elsif” form is presented below:

```

If Boolean_condition_1 then
    sequence_of_instructions_1
elsif Boolean_condition_2
    sequence_of_instructions_2
else
    sequence_of_instructions_2
end if;

```

The Listing 2.11 presents a sequential description of the XOR gate using the if statement.

```

11  ARCHITECTURE Sequential_state_1 OF Xor_gate IS
12  BEGIN
13      PROCESS (a,b)
14      if a='0'andb='0' then
15          S <='0';
16          elsif a='1'andb='1'
17              S <='0';
18              else
19                  S <='1';
20                  end if;
21      end PROCESS;
22  END Sequential_state_1 ;
23

```

Listing 2.11 XOR gate using “if” statement

2.3.2.2 CASE Statement

This instruction allows selecting a sequence of instructions among others, according to the value of an expression. It is the equivalent of the combinatorial instruction “with/select”. The syntax rules are as follows:

```

Case expression
when choice_1=>sequence_of_instructions_1
    when choice_2=>sequence_of_instructions_2
        when choice_3=>sequence_of_instructions_3
            ...
                when others => sequence_of_instructions_n
end case;

```

The value of the expression will be compared with the n possible choices, i.e., choice_1, choice_2, ..., choice_n. When a match is found, the assignment corresponding to that particular choice will be performed.

The Listing 2.12 presents a sequential description of the XOR gate using the case statement.

2.3.2.3 LOOP Statement

We use a loop statement to express a sequence of statements that is to be repeatedly executed. There are five cases involving the LOOP statements:

Unconditional LOOP: in this case, the number of iterations will be infinite

```
[label:] LOOP
sequential_statements
END LOOP [label];
```

11	ARCHITECTURE Sequential_state_2 OF Xor_gate IS
12	BEGIN
13	PROCESS
14	BEGIN
15	case (a AND b) is
16	WHEN "00"=> S <='0';
17	WHEN "11"=> S <= '0';
18	WHEN OTHERS => S <='1';
19	end case;
20	end process;
21	END Sequential_state_2 ;
22	

Listing 2.12 XOR gate using “case” statement

LOOP with FOR: the instruction sequence is repeated for a given number of times. a count variable is used to count the number of cycles.

```
[label:]FOR identifier INrange LOOP
sequential_statements
END LOOP [label];
```

LOOP with WHILE: this form loops while a given condition is true, this condition is tested before each round.

```
[label:]WHILE condition LOOP
sequential_statements
END LOOP [label];
```

LOOP with EXIT: the “EXIT” instruction allows to exit a loop and thus to interrupt all the remaining iterations of the loop

```
[loop_label:] [FOR identifier IN range] LOOP
...
[exit_label:] EXIT [loop_label] [WHEN condition];
...
END LOOP [loop_label];
```

LOOP with NEXT: the instruction next allows stopping the iteration during the loop. After this instruction, the following iteration is executed.

```
[loop_label:] [FOR identifier IN range] LOOP  
...  
[next_label:] NEXT [loop_label] [WHEN condition];  
...  
END LOOP [loop_label];
```

2.3.2.4 Assertion and Report Statements

The ASSERT instruction allows to monitor a condition and to send a message if this condition is false. Its syntax is as follow:

```
assert condition [report expression] [ severity expression];
```

If the monitored condition is true, nothing happens. However, if it is false, the “expression” message is displayed on the screen and the level of severity of the error is indicated. The message is of type STRING. In case of absence of the clause “report”, the message “assertion Violation” will be taken by default.

2.4 Test Bench

Once the VHDL model is written, it must be verified that it meets the functional specifications of the system to be designed. For this purpose, behavior is simulated by test inputs of a test bench.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.all;
3
4 -- testbench entity
5 entity testbench1 is -- no inputs or outputs
6 end;
7
8 -- testbench architecture
9 architecture sim of testbench1 is
10 component xor_gate
11     port(a, b: in STD_LOGIC;
12             s: out STD_LOGIC);
13 end component;
14 signal tb_a,tb_b,tb_s:STD_LOGIC;
15 begin
16     --instantiate device under test
17     dut:xor_gate port map(tb_a,tb_b,tb_s);
18     -- apply inputs one at a time
19     process
20         begin
21             a <= '0'; b <= '0'; wait for 10ns;
22             a<= '1'; b <= '0'; wait for 10ns;
23             a <= '1'; b <= '1'; wait for 10ns;
24             a<= '0'; b<= '1'; wait for 10ns;
25         end process;
26     end sim ;
27

```

Listing 2.13 Test bench of xor gate

A test bench is a piece of VHDL code that provides input combinations covering all the circumstances in which the final circuit will be used and the resulting outputs are verified. Simple Test bench for the Xor gate is presented in Listing 2.13. A VHDL code for the test bench is similar to an ordinary VHDL code. The particularity is that the entity is empty. At the architecture level, the object to be tested is instantiated (line 18, Listing 2.13) and then the stimuli are generated (line 22–25, Listing 2.13). Delays are used to apply the inputs in the right order. Then, the results of the simulation must be seen and the outputs produced must be verified as correct.

A Test bench is simulated in the same way as other VHDL modules. However, it is not synthesizable.

2.5 Syntax Summary

In the following section, we present a summary of the main syntax elements used in VHDL codes developed in this book.

2.5.1 *Comments*

Comments in VHDL start with two dashes “--”, they can be put anywhere in the code and they end with the line.

--it is a comment.

2.5.2 *Signal, Variables and Constants*

SIGNAL declares a signal allowing communication between concurrent states within architecture. In the definition, a type must be specified for the signal and a default value can be assigned.

```
SIGNAL sig_name {sig_name} : signal_type [:=initial_value];
```

Variables are used in a PROCESS, a PROCEDURE or a FUNCTION. They can be given an initial value. This initial value is assigned to each call of the PROCEDURE or FUNCTION.

```
VARIABLE var_name {var_name} : type [:= value];
```

Constants are used to reference a specific value or type.

```
CONSTANT const_name { const_name} : type := value;
```

2.5.3 *Array*

VHDL arrays can be multidimensional. They are defined with fixed dimensions or without specifying the dimensions.

```
TYPE identifier IS ARRAY
    [unconstrained_array_definition];
    [constrained_array_definition];
```

Example

```
TYPE data_bus IS array (0 to 31) OF bit; -- array of 32 bit
```

Table 2.5 VHDL operator classes

Class	Operators
Logical operators	And, or, nand, nor, xor
Relational operators	=, /=, <, <=, >, >=
Shift operators	Sll, srl, sla, sra, rol, ror
Addition operators	+ , −, &
Sign operator	+ , −
Multiplication operator	* , /, mod, rem
Various operator	Abs, not

2.5.4 Operators

There are six classes of operators and a unique priority level is assigned to each class. Table 2.5 summarizes the main classes and their corresponding operators in an increasing order of priority (the lowest priority class first).

2.5.5 Attributes

Attributes are used to add additional information to a signal, variable or component. Some predefined attributes in VHDL are listed in the Table 2.6.

object_name'attribut_name

Table 2.6 Predefined attributes in VHDL

Attribute	Acts on	Returned value
'left	Scalar	left element
'left(n)	Array	left terminal of the index of dimension n,
'right	Scalar	right element
'right(n)	Array	right terminal of the index of dimension n,
'high	Scalar	largest element
'high(n)	Array	maximum bound of the index of the dimension n
'low	Scalar	smallest element
'low(n)	Array	minimum bound of the index of the dimension n
'event	Signal	Boolean value “TRUE” if the signal value has just changed

2.6 Summary

In this chapter, the general background and coding techniques of VHDL are highlighted. Basic concept to programming can be summarized as follow:

- Three main units constitute a VHDL code: a working library, an entity that describes the external view of the system and an architecture that describes the behaviour or internal structure of the system.
- Sequential instructions are used into a process; the execution of this process can be suspended by the “wait” instruction.
- A process is a concurrent instruction that contains only sequential instructions.
- The concurrent signal assignment can take either a conditional or a selective form.
- The instantiation of a component allows to take a copy of a declared component and to make the correspondence between its ports and the effective ports of the circuit.
- The main purpose of the test bench is to verify the functionality of the circuit and to ensure that it corresponds to the specification previously set by the designer.

Part II

**Design Digital Circuits (Simulation
and Implementation on FPGA)**

Chapter 3

Combinational Logic Circuits



Abstract This chapter aims to describe in VHDL the design of various combinational circuits such as arithmetic circuits (adder and subtractor), multiplexers and demultiplexer, decoders and encoders. These circuits have been verified and tested on the one hand by simulations carried out using ModelSim tool and on the other hand by their implementation in the FPGA platform.

3.1 Introduction

This chapter presents a practical guide for the simulation, implementation and validation of certain combinational logic designs on the FPGA platform. A combinatorial logic is a type of digital circuit whose outputs only depend on the current value of its inputs. Thus, we refer to the system without memory.

This chapter focuses on the design of various combinatorial circuits such as arithmetic circuits, Multiplexer and Demultiplexer circuits, encoder and decoder circuits using the VHDL description language. This chapter makes the user more comfortable towards learning of design of digital circuits. For designing combinational logic circuits, several VHDL modeling approaches have been such as data flow modeling, behavioral and structural modeling.

A good part of this chapter has been devoted to the subject of testing and verifying the operation of combinatorial circuits using simulations carried out by the ModelSim tool and the validation of the design implementation on FPGA.

3.2 Arithmetic Circuits

The basic building blocks of an arithmetic unit in a digital system are adders. They are very important components due to their extensive use in other basic digital operations such as subtraction, multiplication and division. In this lab, we illustrate the implementation of arithmetic operations such as addition and subtraction based on

adder components. Several types of adders such as 1-bit half adder, 1-bit full adder, 4-bit adder, and 4-bits adder-subtractor have been implemented and discussed. Before moving to the implementation of these circuits in the FPGA platform, the validation of their VHDL design will be checked by the simulation timing diagrams using the Quartus II and ModelSim simulation tools.

We start this Lab by building the 1-bit half adder using two logic gates (XOR and AND). Then we use two half adder components to design the full adder. Next, we use four components of the 1-bit full adder to implement the 4-bit Ripple Carry Adder (RCA). Finally, we use 4 units of full adders and a separate XOR gate to design 4-bit Adder-Subtractor.

3.2.1 1-Bit Half Adder

The first basic functional block is the half adder. A 1-bit half adder is a logic circuit that adds two bits and produce a sum bit and a carry bit. Figure 3.1 shows the logic symbol of the circuit, which is called a 1-bit half adder.

The 1-bit half adder truth table is given in Table 3.1. This table indicates the values of the outputs for each possible input, and thus specifies the operation of a 1-bit half adder. The sum (S) output is 1 when either of the inputs (A or B) is 1, and the carry output (Cout) is 1 when both the inputs (A and B) are 1.

The Boolean functions of the two outputs, derived from the truth table are:

$$S = \overline{A}B + \overline{B}A = A \oplus B \quad (3.1)$$

$$Cout = A \cdot B \quad (3.2)$$

Fig. 3.1 1-bit half adder

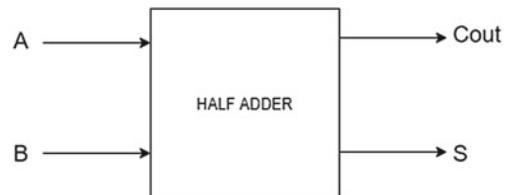
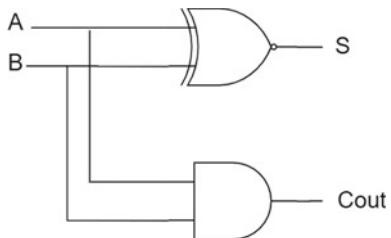


Table 3.1 1-bit half adder truth table

Inputs		Outputs	
A	B	S	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Fig. 3.2 1-bit half adder circuit



When synthesized, the logic circuit shown in Fig. 3.2 is produced.

3.2.1.1 VHDL Implementation

Using the schematic shown in Fig. 3.2 as a guide, the 1-bit half adder can be implemented using dataflow modeling approach. The VHDL program equivalent can be expressed as two concurrent statements within our architecture body (see Listing 3.1). These instructions are executed simultaneously as soon as the logic inputs (a and b) are supplied. It should be noted that the order of these statements in the architecture body is unimportant.

```

1  library ieee;
2  USE ieee.std_logic_1164.all;
3
4  ENTITY Half_adder IS
5  PORT (a,b : IN std_logic;
6         s,cout : OUT std_logic);
7  END Half_adder;
8
9
10 ARCHITECTURE dataflow OF Half_adder IS
11
12 BEGIN
13
14     s <= a XOR b;
15     cout <= a AND b;
16
17 END dataflow;

```

Listing 3.1 VHDL code of 1-bit for half adder

3.2.1.2 Simulation

The simulation for this 1-bit half adder has been performed using Quartus II and ModelSim tool. The operation of the half-adder is checked using a simulation timing diagram before programming its implementation into the FPGA platform.

An example of the VHDL test bench for the design of 1-bit half adder is shown in Listing 3.2.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity tb_half_adder is --- Empty entity
5  end tb_half_adder;
6
7  Architecture test of tb_half_adder is
8
9  -- Declaration of the component to be tested
10 Component Half_Adder
11 Port(
12   A, B:  in std_logic;
13   Sum, Carry : out std_logic);
14 end component;
15
16 -- declaration of interconnection signals
17 Signal stim_A : std_logic:= '0';
18 Signal stim_B : std_logic:= '0';
19 Signal stim_Sum : std_logic;
20 Signal stim_Carry : std_logic;
21
22 begin
23 -- Instantiation of the component to be tested
24 comp_à_test : half_adder port map(
25   A => stim_A,
26   B => stim_B,
27   sum => stim_sum,
28   Carry => stim_carry);
29
```

```

30  | --Generation of test vectors
31  | process
32  | begin
33  |   stim_A <= '0';
34  |   stim_B <= '0';
35  |   wait for 10 ns;
36  |   stim_A <= '0';
37  |   stim_B <= '1';
38  |   wait for 10 ns;
39  |   stim_A <= '1';
40  |   stim_B <= '0';
41  |   wait for 10 ns;
42  |   stim_A <= '1';
43  |   stim_B <= '1';
44  |   wait for 10 ns;
45  | end process;
46
47  | end test;

```

Listing 3.2 VHDL test bench of the 1-bit half adder

A VHDL code for the test bench is similar to an ordinary VHDL code. The particularity is that the entity is empty. At the architecture level, the object to be tested is instantiated (line 24, see Listing 3.2) and then the stimuli are generated (line 3345, see Listing 3.2). Delays are used to apply the inputs in the right order. A test bench is simulated in the same way as other VHDL modules.

The simulation is shown in Fig. 3.3. As you can see, the 1-bit half adder model is checked correctly for different combinations of inputs according to its truth table explained in Sect. 3. The sum bit S is HIGH for two inputs of different values and the carry cost is HIGH whenever two input bits are HIGH. For the $a = 1$ and $b = 1$ inputs, the outputs are $S = 0$ and $Cout = 1$, which are highlighted in Fig. 3.3.



Fig. 3.3 Simulation waveform of the half adder

3.2.1.3 Implementation and Validation on FPGA Platform (Verification and Debugging on Actual Device)

To verify the expected behavior of the designed circuit, the designer can use the basic I/O devices of the Altera DE2-70 FPGA development and education board shown in Fig. 3.4 such as light emitting diodes (LEDs), buttons and switches. The designer can perform more detailed checks by connecting measuring instruments such as an oscilloscope and logic analyzer to general purpose inputs and outputs (GPIO) available in the FPGA board.

Programming the DE2-70 board requires:

- An USB cable.
- The programming software in Quartus II to transfer the design from the PC to the FPGA.
- Joint Test Action Group (JTAG) a standard protocol for programming and debugging of printed circuit boards (PCBs).

In this example, A and B are assigned in the DE2 board to switch 0 and switch 1, respectively. The sum S and carry Cout are assigned to LED red 0 and LED red 1, respectively. The AA23, AB26, AJ6 and AK5 FPGA pins are assigned to the labels SW[0], SW[1], LEDR[0] and LEDR[1], as shown next Table 3.2.

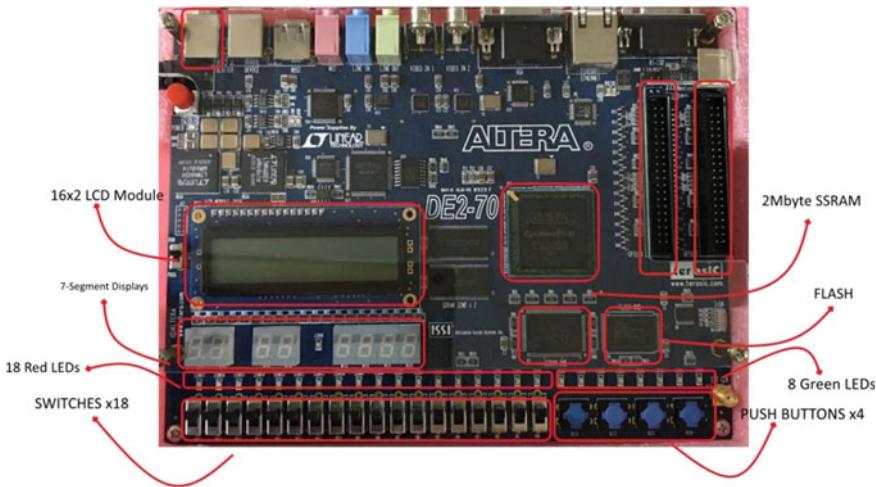


Fig. 3.4 FPGA board for education or training

Table 3.2 Pins assigned to the half adder

Inputs		Outputs	
SW[0]	PIN_AA23	LEDR[0]	LEDR[0]
SW[1]	PIN_AB26	LEDR[1]	LEDR[1]

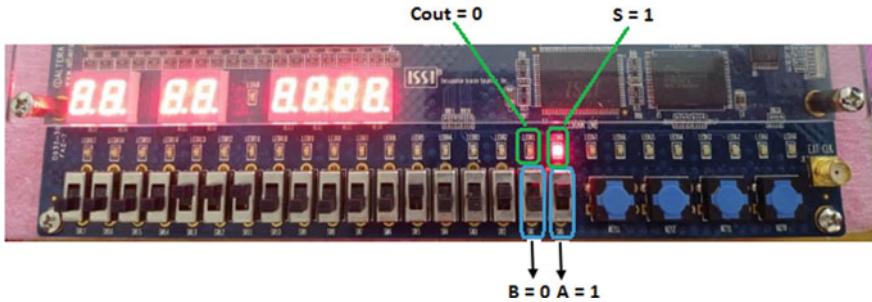


Fig. 3.5 Validation of the design of the 1-bit half adder on the FPGA board

The design of the half adder has been verified according to the circuit truth table. As shown in Fig. 3.5, the led assigned to the sum only lights up when a is different from b and while the led assigned to the retained only lights up when A is equal to B .

3.2.2 1-Bit Full Adder

The 1-bit full-adder extends the concept of the half-adder by providing an additional carry-in (Cin) input, as shown in Fig. 3.6. This is a design with three inputs (A , B , and Cin) and two outputs (S and $Cout$). This a combinational circuit adds the three binary input numbers to produce sum and carry-out terms.

The truth table of this design is shown in Table 3.3.

As shown in Table 3.3, there are eight possible input combinations for the three inputs and for each case the S and $Cout$ values are listed. The Boolean functions of the outputs, derived from the truth table are:

$$S = A \oplus B \oplus Cin \quad (3.3)$$

$$Cout = A.B + B.Cin + Cin.A \quad (3.4)$$

If you refer back to the half adder circuit in Fig. 3.2, you can see that the full adder can be constructed from two half adders and an OR gate, as shown in Fig. 3.7.

Fig. 3.6 1-bit full adder circuit

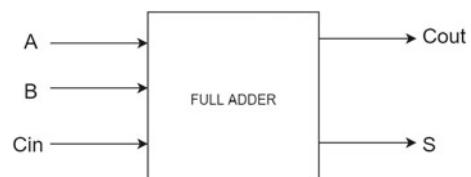
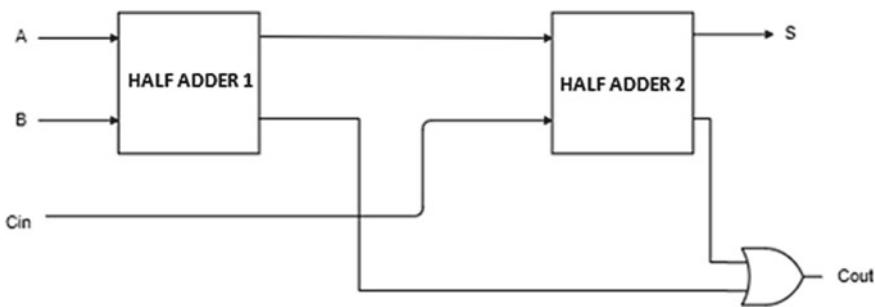


Table 3.3 1-bit full adder truth table

Inputs		Outputs		
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

**Fig. 3.7** Schematic of 1-bit full adder circuit using 2 half-adders

3.2.2.1 VHDL Implementation

We write the VHDL code for the 1-bit full adder with two approaches modeling: dataflow and structural modeling.

(a) Design a full adder using dataflow modeling

The 1-bit full adder is implemented using dataflow modeling. The VHDL code is provided in the Listing 3.3:

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Full_adder is port(
5      A,B, Cin : in std_logic;
6      S, Cout : out std_logic);
7
8  end Full_adder;
9
10
11 architecture dataflow of Full_adder is
12
13 begin
14
15 begin
16     S <= (A XOR B) XOR Cin;
17     Cout <= (A and B) or (B and Cin) or (Cin and A);
18
19 end dataflow ;

```

Listing 3.3 VHDL code of the 1-bit full adder using dataflow modeling

Two Eqs. (3.3 and 3.4) are in the architecture of the code depicting the Boolean equation for the sum and carry. These are called concurrent statements because they synthesize two logic circuits that are executed concurrently (at the same time), as soon as the inputs to the logic (A, B, and Cin) are provided.

(b) Design a full adder using structural modeling

The 1-bit full adder can be implemented using two half adders and an OR gate as shown in Fig. 3.7. Using structural modeling, The VHDL code is provided below (Listing 3.4):

```

1  Library ieee;
2  Use ieee.std_logic_1164.all;
3
4  Entity Full_adder is port(
5    A,B,Cin: in std_logic;
6    S, Cout: Out std_logic);
7  end Full_adder;
8
9  Architecture struct of Full_adder is
10
11  -- Component declaration
12  component Half_adder
13  Port (
14    A,B: in std_logic;
15    S, Cout: out std_logic);
16  end component;
17
18  -- Signal declaration
19  Signal I1, I2, I3: std_logic;
20
21  begin
22  -- Generation of the output carryover
23  Cout <= I2 or I3;
24
25  -- Instantiation of the first Half adder
26  HA1 : Half_adder port map(A, B, I1, I2);
27
28  -- Instantiation of the second half-adder
29  HA2 : Half_adder port map(I1, Cin, S, I3);
30
31  end struct;

```

Listing 3.4 VHDL code of the 1-bit full adder using structural modeling

3.2.2.2 Simulation of the 1-Bit Full Adder

The VHDL test bench for the design of the 1-bit full adder is shown in Listing 3.5. As can be seen in the simulation of Fig. 3.8, the results of this full adder are exactly

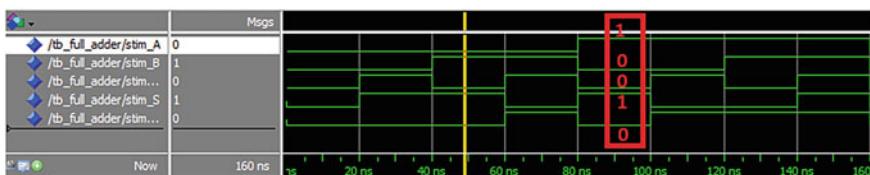


Fig. 3.8 Simulation waveform of the 1-bit full adder

the same as shown in Table 3.3. Note that the sum and carry of A, B, and Cin are highlighted in red, and verify clearly the truth table.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity tb_full_adder is --- Empty entity
5  end tb_full_adder;
6
7  Architecture test of tb_full_adder is
8
9  -- Declaration of the component to be tested
10 Component full_Adder
11 Port(
12   A, B, Cin:  in std_logic;
13   S, Cout : out std_logic);
14 end component;
15
16 -- declaration of interconnection signals
17 Signal stim_A : std_logic:= '0';
18 Signal stim_B : std_logic:= '0';
19 Signal stim_Cin : std_logic:= '0';
20 Signal stim_S : std_logic;
21 Signal stim_Cout : std_logic;
22
23 begin
24 -- Instantiation of the component to be tested
25 FA : full_adder port map(
26   A => stim_A,
27   B => stim_B,
28   Cin => stim_Cin,
29   S => stim_S,
30   Cout => stim_Cout);
31
```

```

32 | --Generation of test vectors
33 | process
34 | begin
35 |   stim_A <= '0';
36 |   stim_B <= '0';
37 |   stim_cin <= '0';
38 |   wait for 20 ns;
39 |   stim_A <= '0';
40 |   stim_B <= '0';
41 |   stim_cin <= '1';
42 |   wait for 20 ns;
43 |   stim_A <= '0';
44 |   stim_B <= '1';
45 |   stim_cin <= '0';
46 |   wait for 20 ns;
47 |   stim_A <= '0';
48 |   stim_B <= '1';
49 |   stim_cin <= '1';
50 |   wait for 20 ns;
51 |   stim_A <= '1';
52 |   stim_B <= '0';
53 |   stim_cin <= '0';
54 |   wait for 20 ns;
55 |   stim_A <= '1';
56 |   stim_B <= '0';
57 |   stim_cin <= '1';
58 |   wait for 20 ns;
59 |   stim_A <= '1';
60 |   stim_B <= '1';
61 |   stim_cin <= '0';
62 |   wait for 20 ns;

63 |   stim_A <= '1';
64 |   stim_B <= '1';
65 |   stim_cin <= '1';
66 |   wait for 20 ns;
67 |
68 | end process;
69 |
70 | end test;

```

Listing 3.5 VHDL test bench of the 1-bit full adder

3.2.2.3 Implementation and Validation on FPGA Platform

In this example, A, B and Cin are assigned in the DE2 board to switch 0, switch 1 and switch 2, respectively. The sum and carry are assigned to LED red 0 and LED

Table 3.4 Pins assigned of 1-bit full adder

Inputs		Outputs	
SW[0]	PIN_AA23	LED[0]	PIN_AJ6
SW[1]	PIN_AB26	LED[1]	PIN_AK5
SW[2]	PIN_AB25	LED[2]	PIN_AJ5

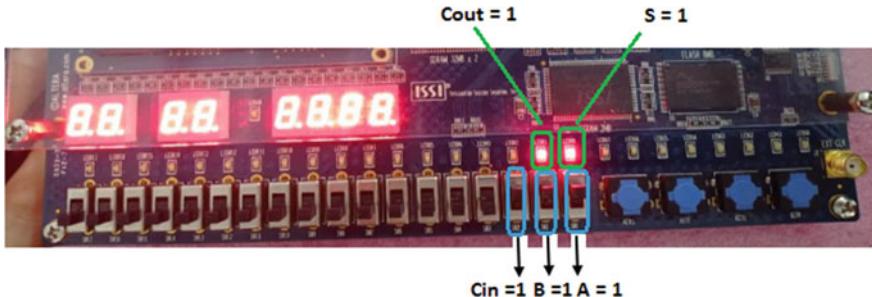


Fig. 3.9 Validation of the design of the 1-bit full adder on the FPGA board

red 1, respectively. The AA23, AB26, AJ6 and AK5 FPGA pins are assigned to the labels SW[0], SW[1], LEDR[0] and LEDR[1], as shown next Table 3.4.

The design of the 1-bit full adder has been verified according to the circuit truth table. As shown in Fig. 3.9, the LEDs assigned to the sum and to the carry forward only light up at the same time when a, b and Cin are equal to 1. Otherwise, i.e. if at least one input is different from 1, only one output LED lights up.

3.2.3 4-Bit Ripple Carry Adder

The addition of multibit numbers can be accomplished using several full-adders. As shown in Fig. 3.10, the design of a 4-bit ripple-carry adder (RCA) can easily be constructed from 4 instances of the 1-bit full-adder connected in series. The input to each full-adder are A_i , B_i and C_i , and the outputs are S_i and C_{i+1} , where 'i' varies from 0 to 3. Also, the carry of each stage is connected to the next unit as the carry in (i.e. the third input). The 4-bit RCA has two 4-bit input ports $A[3:0]$ ($A_3\ A_2\ A_1\ A_0$) and $B[3:0]$ ($B_3\ B_2\ B_1\ B_0$) and one carry in input, Cin set to 0. The 4-bit RCA outputs are depicted as $S[3:0]$ ($S_3\ S_2\ S_1\ S_0$) and carry out, $Cout$.

The sum for the 4-bit RCA is provided by four full adders and the carry is propagated from the first to the fourth full adder at the carry out port. It should be noted that the addition is described as " $a \oplus b \oplus cin$ ", which means that the generation of carry over between the additions is taken into consideration at each step of the addition operation. To illustrates the addition operation with a specific example, consider the

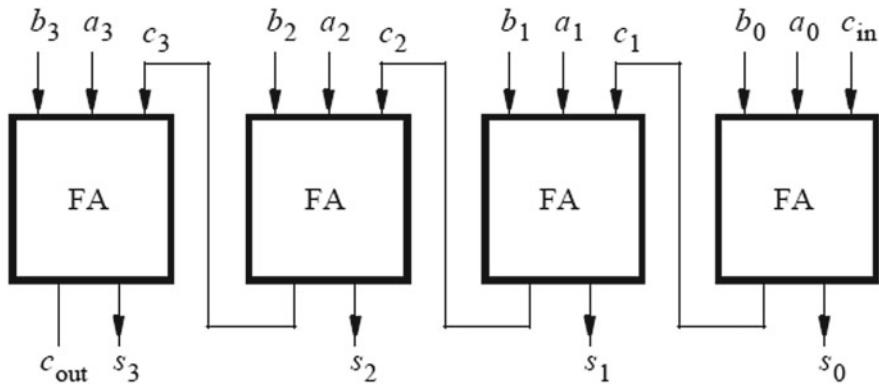


Fig. 3.10 4-bit Ripple Carry Adder

Table 3.5 Example

Subscript i	3	2	1	0	
Input carry	0	1	1	0	C_i
Augend word	1	0	1	1	A_i
Addend word	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

two binary numbers $A = 1011$ and $B = 0011$. Their sum $S = 1110$ is formed with the four-bit adder as follows (see Table 3.5).

The 4-bit adder can be used in many applications involving arithmetic operations.

3.2.3.1 VHDL Implementation

We write the VHDL description for the 4-bit Ripple Carry Adder using two modeling approaches: structural modeling and the generate statement.

(a) Design a 4-bit Ripple Carry Adder using structural modeling

The structural modeling approach for the 4-bit adder shown in Fig. 3.10 is based on the following points:

- Declaration of 1-bit full adder component. This block is used as a component in our code, we can save his VHDL code as a separate file, namely *Full_adder.vhd*, in our project
- Declaration of component interconnection signals: C1, C2, C3.
- Instantiating four copies of the 1-bit Full adder component using the Port mapping command. The first adder's carry in is set to '0' as shown above. For the rest of the

full adders, the carry input is the carry output of the previous full adder. Hence, the carries ripple up in this circuit, which gives it the name, ripple-carry adder.

Using this structural modeling, the VHDL code is provided below (see Listing 3.6):

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity Ripple_Adder is
5  Port ( A, B : in STD_LOGIC_VECTOR (3 downto 0);
6  Cin : in STD_LOGIC;
7  S : out STD_LOGIC_VECTOR (3 downto 0);
8  Cout : out STD_LOGIC);
9  end Ripple_Adder;
10
11 architecture Structural of Ripple_Adder is
12
13  --Full Adder Component Declaration
14  component full_adder
15  Port ( A, B, Cin : in STD_LOGIC;
16  S, Cout : out STD_LOGIC);
17  end component;
18
19  --Intermediate Carry declaration
20  signal c1,c2,c3: STD_LOGIC;
21
22 begin
23
24  --Instantiating four copies of the 1-bit Full adder component by Port mapping
25  FA1: full_adder port map( A(0), B(0), Cin, S(0), c1);
26  FA2: full_adder port map( A(1), B(1), c1, S(1), c2);
27  FA3: full_adder port map( A(2), B(2), c2, S(2), c3);
28  FA4: full_adder port map( A(3), B(3), c3, S(3), Cout);
29
30 end Structural;

```

Listing 3.6 VHDL code of the 4-bit Ripple Carry Adder using structural modeling

(b) Design a 4-bit RCA using generate statement

We can use the “*for-generate*” statement to considerably simplify the code in Listing 3.7. The four component instantiation statements shown previously can be written in a more general form:

adder(i): full_add PORT MAP (a(i), b(i), c(i-1), S(i), c(i));

A statement that can be written in this indexed form can be implemented using a GENERATE statement, which has the form:

```

label:
FOR index IN range GENERATE
statements;
END GENERATE;
```

The VHDL code is given in Listing 3.7 shows how to use the generate statement to create a 4-bit Ripple Carry Adder:

```

1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity Ripple_Adder is
5   Port ( A, B : in STD_LOGIC_VECTOR (3 downto 0);
6   Cin : in STD_LOGIC;
7   S : out STD_LOGIC_VECTOR (3 downto 0);
8   Cout : out STD_LOGIC);
9   end Ripple_Adder;
10
11  architecture Structural of Ripple_Adder is
12
13  --Full Adder Component Declaration
14  component full_adder
15  Port ( A, B, Cin : in STD_LOGIC;
16  S, Cout : out STD_LOGIC);
17  end component;
18
19  --Intermediate Carry declaration
20  signal c : std_logicvector(4 downto 0);
21
22 begin
23
24  c(0) <= cin;
25  adders:
26  FOR i IN 1 to 4 GENERATE
27  adder: full_add PORT MAP (a(i),b(i),c(i-1),S(i),c(i));
28  -END GENERATE;
29  cout <= c(4);
30
31 end Structural;
```

Listing 3.7 VHDL code of the 4-bit Ripple Carry Adder using generate statement

When the variable index ‘i’ goes from 1 to 4, the instruction therefore instantiates four instances of the full adder. Since we have an input carry, an output carry and three internal carries, we must use a 5-bit signal (BIT_VECTOR (4 downto 0)) if we include all carry bits in indexed form. The input carry, Cin, defined in the entity declaration, is assigned to the vector element C(0). Similarly, the output, Cout, is assigned the value of the element C(4).

3.2.3.2 Simulation

The test bench required to simulate 4-bit RCA is shown in the Listing 3.8.

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  ENTITY Tb_Ripple_Adder IS
5  END Tb_Ripple_Adder;
6
7  ARCHITECTURE behavior OF Tb_Ripple_Adder IS
8
9  --Component Declaration for the Unit Under Test (UUT)
10
11 COMPONENT Ripple_Adder
12 PORT(
13   A, B : IN std_logic_vector(3 downto 0);
14   Cin : IN std_logic;
15   S : OUT std_logic_vector(3 downto 0);
16   Cout : OUT std_logic);
17 END COMPONENT;
18
19 --Inputs
20 signal stim_A : std_logic_vector(3 downto 0) := (others => '0');
21 signal stim_B : std_logic_vector(3 downto 0) := (others => '0');
22 signal stim_Cin : std_logic := '0';
23
24 --Outputs
25 signal stim_S : std_logic_vector(3 downto 0);
26 signal stim_Cout : std_logic;
27
28 BEGIN
29
30  --Instantiate the Unit Under Test (UUT)
31  uut: Ripple_Adder PORT MAP (A => A,B => B,Cin => Cin,S => S,Cout =>; Cout);
32

```

```

33  -- Stimulus process
34  stim_proc: process
35    begin
36
37    A <= "0110";
38    B <= "1100";
39    wait for 100 ns;
40
41    A <= "1111";
42    B <= "1100";
43    wait for 10 ns;
44
45    A <= "0110";
46    B <= "0111";
47    wait for 10 ns;
48
49    A <= "0110";
50    B <= "1110";
51    wait for 10 ns;
52
53    A <= "1111";
54    B <= "1111";
55    wait for 10 ns;
56
57  end process;
58
59 END behavior;

```

Listing 3.8 VHDL test bench of the 4-bit Ripple Carry Adder

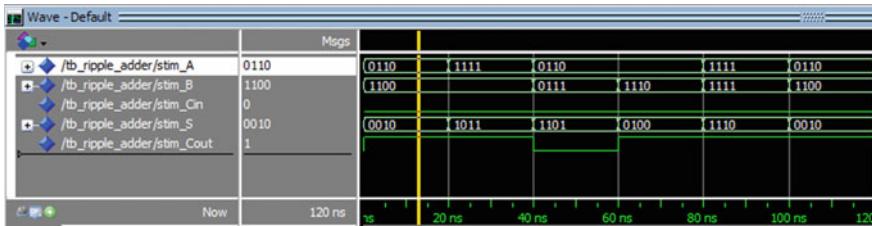


Fig. 3.11 Output waveform of the 4-bit Ripple Carry Adder

Figure 3.11 shows the simulation results for the 4-bit RCA. These results demonstrate that the outputs are correct with all possible combinations of the 4-bit inputs of A and B. Cin, the carry in of the previous adder is taken 0.

3.2.3.3 Implementation and Validation on FPGA Platform

The inputs A and B are assigned in the DE2 board to Switches SW[0]–SW[3] and Switches SW[4]–SW[7], respectively. The Sum (S) bits is assigned to LEDs red LEDR[0]–LEDR[3]. The carry-out bit (C4 = Cout) is displayed on LEDR[4].

Table 3.6 shows the pins assigned for the 4-bit Ripple Carry adder.

When the switch is turn ON, the input is set to 1 and when it goes to the OF position, the input is set to 0. The 4-bit Ripple Carry adder circuit has been tested with different combinations of inputs. As shown in Fig. 3.12, when A = 1111, B = 1111 and Cin = 0, the LEDs assigned to the sum S and to the Cout report light up. In short, the circuit works correctly for all possible combinations of A and B.

Table 3.6 Pins assigned of 4-bit Ripple Carry adder

Inputs			Outputs		
A[3]	SW[0]	PIN_AA23	S[3]	LED[4]	PIN_AK3
A[1]	SW[1]	PIN_AB26	S[2]	LED[3]	PIN_AJ4
A[2]	SW[2]	PIN_AB25	S[1]	LED[2]	PIN_AJ5
A[0]	SW[3]	PIN_AC27	S[0]	LED[1]	PIN_AK5
B[3]	SW[4]	PIN_AC26	C4 = Cout	LED[0]	PIN_AJ6
B[2]	SW[5]	PIN_AC24			
B[1]	SW[6]	PIN_AC23			
B[0]	SW[7]	PIN_AD25			

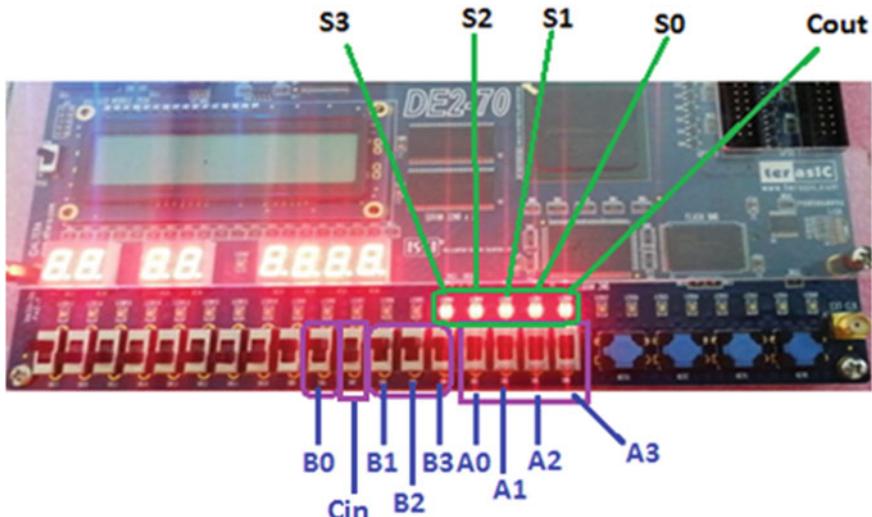


Fig. 3.12 Validation of the design of the 4-bit Ripple Carry adder on the FPGA board

3.2.4 4-Bit Adder-Subtractor

The purpose of this subsection is to design a binary adder-subtractor by simply extending the full 1-bit adder which is explained in Sect. 2.2. The adder-subtractor unit is perhaps the most important element of an Arithmetic and Logic Unit (ALU) of a microprocessor. It can be added/subtracted two N-bit binary numbers and output their N-bit binary sum/difference, a carry/borrow status bit, and if needed an overflow status bit. If we choose to represent signed numbers using two's complement, then we can build an adder/subtractor from a basic adder circuit, e.g. a ripple carry adder.

Figure 3.13 illustrates the 4-bit adder-subtractor circuit. The circuit for subtracting $A - B$ consists of a parallel adder with inverters placed between each B terminal and the corresponding full-adder input.

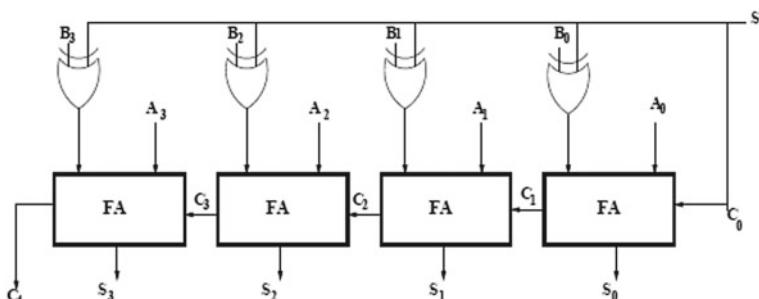


Fig. 3.13 4-bit adder-subtractor

The two primary inputs are numbers A_0, A_1, A_2, A_3 and B_0, B_1, B_2, B_3 , the primary output is S_0, S_1, S_2, S_3 . The input carry C_0 must be equal to 1. Another input is the Sub control of the operation:

- When $\text{Sub} = 0$, the circuit is an adder
- When $\text{Sub} = 1$, the circuit becomes a subtractor

Each exclusive-OR gate receives input S and one of the inputs of B , B_i . When $\text{Sub} = 0$, we have $B_i \oplus 0 = B_i$. If the full adders receive the value of B , and the input carry is 0, the circuit performs $A + B$.

When $\text{Sub} = 1$, we have $B_i \oplus 1 = \overline{B_i}$ and $C_0 = 1$. The circuit performs the operation A plus the 2's complement of B .

3.2.4.1 VHDL Implementation

In this part, we use the structural modeling style combined with generate statement to build the 4-bit adder-subtractor. The structural approach is based on full adders and logic gates. The VHDL code is provided in the Listing 3.9.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity addsub is
5  port( Sub: in std_logic;
6        A,B  : in std_logic_vector(3 downto 0);
7        Sout : out std_logic_vector(3 downto 0);
8        Cout : out std_logic);
9  end addsub;
10
11 architecture struct of addsub is
12
13 component Full_Adder is
14 port( A, B, Cin : in std_logic;
15       S, Cout : out std_logic);
16 end component;
17
18 -- Define a signal for internal carry bits
19 signal C : STD_LOGIC_VECTOR (4 downto 0);
20 signal TMP: std_logic_vector(3 downto 0);

```

```

21
22 begin
23 C(0)<= Sub;
24
25 subtraction: for i in 0 to 3 generate
26     TMP(i) <= b(i) xor sub;
27 end generate subtraction;
28
29 RCA: for i in 0 to 3 generate
30     adder: full_adder port map(
31         a => a(i),
32         b => tmp(i),
33         cin => c(i),
34         S => Sout(i),
35         cout => c(i+1));
36     end generate;
37
38 Cout <= C(4);
39 end struct;

```

Listing 3.9 VHDL code for 4-bit adder-subtractor using structural modeling

3.2.4.2 Simulation

Figure 3.14 shows the simulation results for the 4-bit adder-subtractor. When Sub = 0, the circuit is an adder and when Sub = 1, the circuit becomes a subtractor. Table 3.7 shows the operations included in the simulation in both hexadecimal and binary form. Note that the sums are interpreted as unsigned operations and the differences are interpreted as signed operations. Any sum (+) or subtract (-) can be interpreted

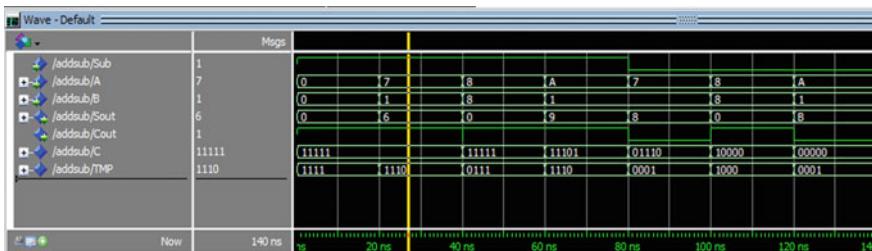


Fig. 3.14 Simulation results of a 4-bit adder-subtractor

Table 3.7 Add/subtract results

Hexadecimal sum/difference	Binary equivalent
$7 + 1 = 8$	$0111 + 0001 = 0\ 1000$ (Unsigned)
$8 + 8 = 0$	$1000 + 1000 = 1\ 0000$ (Unsigned)
$A + 1 = B$	$1010 + 0001 = 0\ 1011$ (Unsigned)

either way, but this sometimes result in a sign bit overflow. (e.g., the sums $8 + 8 = 10$ indicate an overflow if they are interpreted as signed additions).

3.2.4.3 Implementation and Validation on FPGA Platform

We use eight slide switches as inputs for A and B and one push button for the Sub as “control” input. When Sub = 0, the circuit is an adder and when Sub = 1, the circuit becomes a subtractor. To check if the circuit is correct, we use the LEDs to display the outputs as shown in the Fig. 3.15.

3.3 Multiplexers

A multiplexer (abbreviated MUX) is defined as a combinational circuit that selects one of several data inputs and to directs it to the output based on the value of a select signal. In general, a multiplexer with N select inputs has $m = 2^N$ data inputs.

In this lab, we illustrate the process of designing of 2-to-1, 4-to-1 and 8-to-1 Multiplexers using dataflow, behavioral and structural modeling in VHDL. Also, their designs have been tested by simulation and implemented in the FPGA platform.

3.3.1 2-to-1 Multiplexer

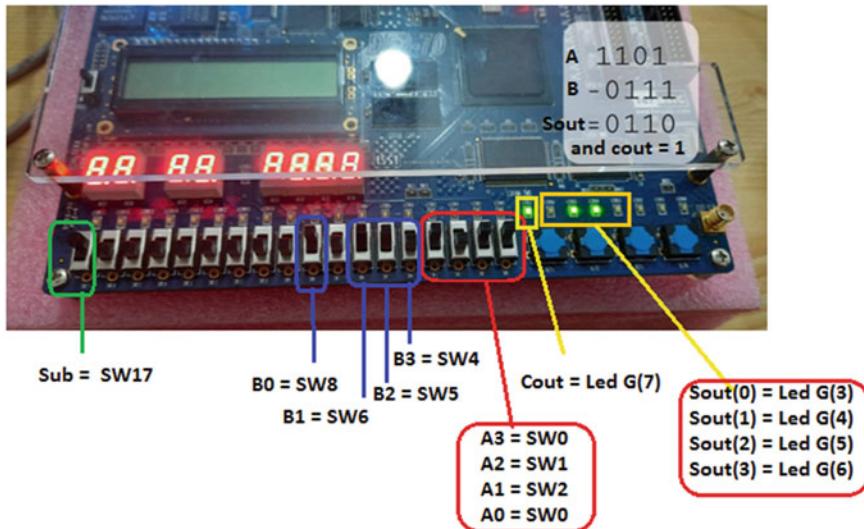
Figure 3.16 shows the symbol and truth table for a 2-to-1 multiplexer. The multiplexer has two data inputs A and B, a select input S, and one output Y. Based on the select signal S, the multiplexer chooses between the two data inputs: if $S = 0$, $Y = A$, and if $S = 1$, $Y = B$ (See Table 3.8).

The Boolean equation Y for the 2-to-1 multiplexer may be derived with a Karnaugh map or read off by inspection ($Y = 1$ if $S = 0$ AND $A = 1$ OR if $S = 1$ AND $B = 1$).

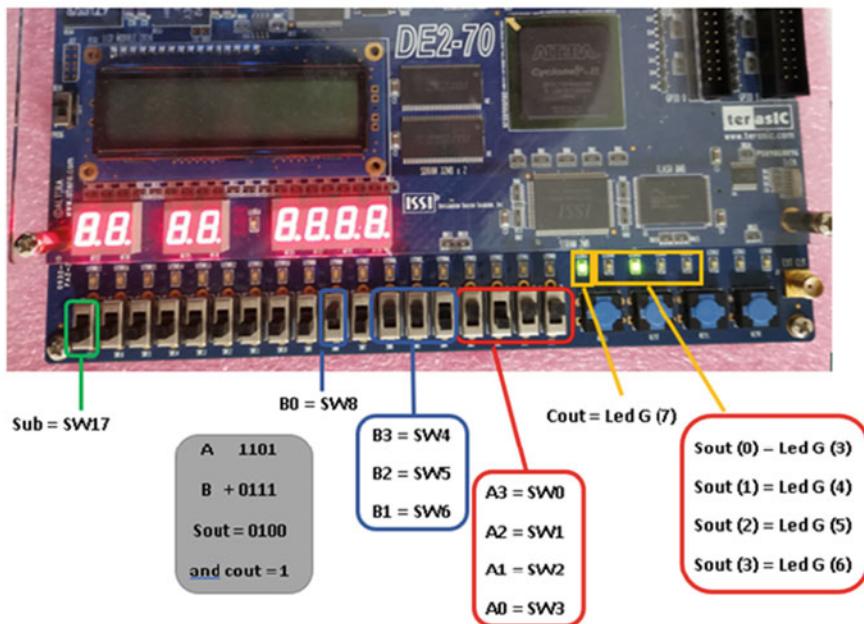
		00	01	11	10
		0	1	0	0
Sel A B	0	0	1	0	0
	1	0	1	1	1

$$Y = A \cdot \bar{S} + B \cdot S \quad (3.5)$$

From the Eq. (3.5), a 2-to-1 multiplexer can be built from sum-of-products logic as shown in Fig. 3.17.



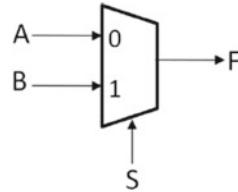
(a) 4-bit adder-subtractor design when Sub = 1



(b) 4-bit adder-subtractor design when Sub = 1

Fig. 3.15 Validation of the adder-subtractor design: **a** 4-bit adder-subtractor when Sub = 1, **b** 4-bit adder-subtractor when Sub = 1

Fig. 3.16 2-to-1 multiplexer: **a** symbol; **b** truth table

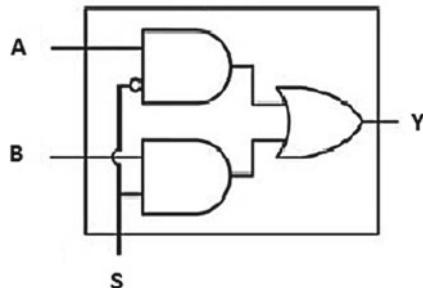


(a): Symbol for a 2-to-1 multiplexer

Table 3.8 Truth table for a 2-to-1 multiplexer

	S	A	B	Y
When S = 0, the Output is A	0	0	0	0
	0	0	1	0
	0	1	0	1
	0	1	1	1
When S = 1, the Output is B	1	0	0	0
	1	0	1	1
	1	1	0	0
	1	1	1	1

Fig. 3.17 2-to-1 multiplexer implementation using two-level logic



3.3.1.1 VHDL Implementation

Several different VHDL modeling approaches can be used to define a multiplexer. We can use data flow, behavioral and structural modeling.

(a) Design a 2-to-1 multiplexer using dataflow modeling:

The VHDL code (see Listing 3.10) shows the description of the 2-to-1 multiplexer using data flow modeling.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity Mux2_1 is
5    port( A,B,S: in std_logic;
6          F: out std_logic);
7  end Mux2_1;
8
9  architecture Dataflow of Mux2_1 is
10 begin
11
12   F <= ((not S) and A) or (S and B);
13
14 end Dataflow;
```

Listing 3.10 VHDL code for 2-to-1 multiplexer using data flow modeling

(b) **Design a 2-to-1 multiplexer using behavioral modeling:**

In behavioral modeling, we describe the behavior of the 2-to-1 multiplexer using sequential statements contained within a process like IF-THEE ELSE or CASE. It should be noted that the sequential statements are statements that execute serially one after the other. The Listing 3.11 and 3.12 illustrate a 2-to-1 multiplexer using IF-THEE ELSE and CASE sequential statements.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity Mux2_1 is
5    port( A,B,S: in std_logic;
6          F: out std_logic);
7  end Mux2_1;
8
9  architecture Behavioral of Mux2_1 is
10 begin
11
12   Process(A,B,S)
13   | Begin
14   | if(S='0')then
15   |   | F<=A;
16   | else
17   |   | F<=B;
18   | end if;
19   | end Process;
20
21   end Behavioral;
```

Listing 3.11 VHDL code of the 2-to-1 multiplexer using if-then else statement

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity Mux2_1 is
5  port( A,B,S: in std_logic;
6        F: out std_logic);
7  end Mux2_1;
8
9  architecture Dataflow of Mux2_1 is
10 begin
11
12 process (A, B, S)
13 begin
14 Case S is
15 when '0'=> F<= A;
16 When others => F<= B;
17 end case;
18 end process;
19
20 end Dataflow;

```

Listing 3.12 VHDL code of the 2-to-1 multiplexor using case statement

3.3.1.2 Simulation

The test bench code to simulate the 2-to-1 multiplexer is shown in Listing 3.13. Simulation scenario covers all the possible outcomes, of the truth table of 2-to-1 multiplexer, Fig. 3.18. 20 ns delays are added at each time a change has done in between inputs, for better observation. It should be noticed that simulation steps are covered in the “*stim_proc*” process.

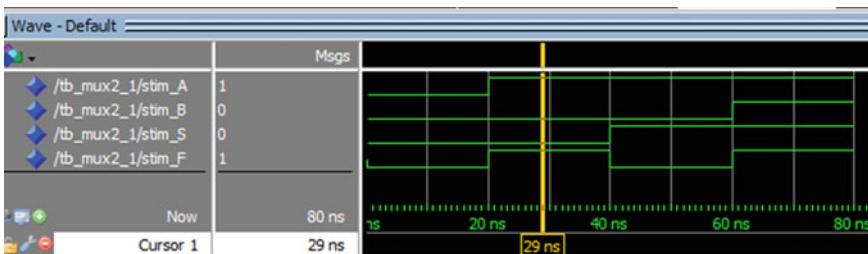


Fig. 3.18 Simulation waveform of a 2-to-1 multiplexer

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  ENTITY tb_Mux2_1 IS
5  END tb_Mux2_1;
6
7  ARCHITECTURE test OF tb_Mux2_1 IS
8
9    -- Component Declaration for the Unit Under Test (UUT)
10
11   COMPONENT Mux2_1
12   PORT(
13     A : IN bit;
14     B : IN bit;
15     S : IN bit;
16     F : OUT bit
17   );
18  END COMPONENT Mux2_1;
19
20
21  --Inputs
22  signal stim_A : std_logic := '0';
23  signal stim_B : std_logic := '0';
24  signal stim_S : std_logic := '0';
25
26  --Outputs
27  signal stim_F : std_logic;

```

```

28  BEGIN
29    -- Instantiate the Unit Under Test (UUT)
30    uut: Mux2_1 PORT MAP (
31      a => stim_A,
32      B => stim_B,
33      S => stim_S,
34      F => stim_F
35    );
36
37    -- Stimulus process
38    stimulus: process
39    begin
40      wait for 20 ns;
41      stim_A <= '1';
42      wait for 20 ns;
43      stim_S <= '1';
44      wait for 20 ns;
45      stim_B <= '1';
46      wait;
47    end process;
48  END test ;

```

Listing 3.13 VHDL test bench for 2-to-1 multiplexer

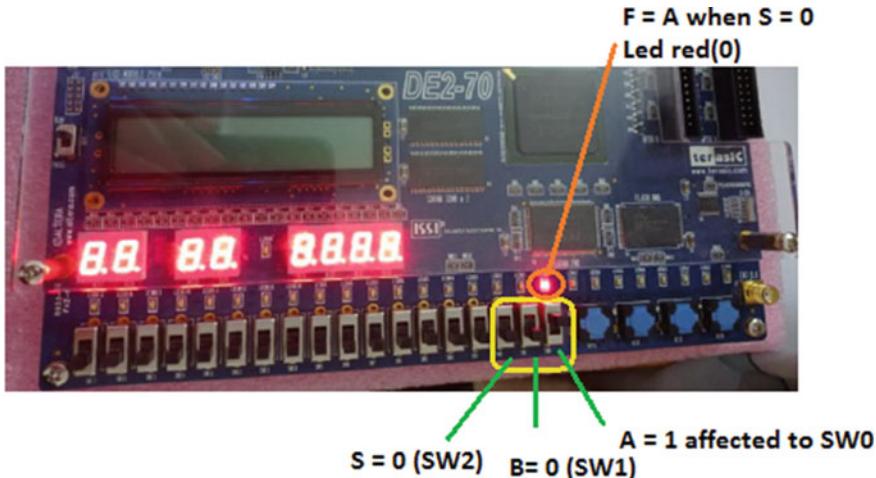
3.3.1.3 Implementation and Validation on FPGA Platform

We assign the inputs A and B in the DE2 Board to switch SW[0] and SW[1], the select input S to SW[2], and the output F to led LEDR[0].

Table 3.9 gives the pins assigned for 2-to-1 multiplexer.

Table 3.9 Pins assigned of 2-to1 multiplexer

Inputs: A and B		Input select: S	Outputs: F		
SW[0]	PIN_AA23	SW[2]	PIN_AB25	LED[0]	PIN_AJ6
SW[1]	PIN_AB26				

**Fig. 3.19** Validation of the design of the 2-to-1 multiplexer on the FPGA board

The 2-to-1 multiplexer circuit has been tested with different combinations of inputs. The design of the 2-to-1 multiplexer has been verified according to the circuit truth table. As shown in Fig. 3.19, when the selector signal switch is low, i.e. $S = 0$, the multiplexer switches to output F input A and the red LED lights up. Otherwise, i.e. when the switch goes high $S = 1$, output F receives input B and the LED lights up.

3.3.2 4-to-1 Multiplexer

A 4-to-1 multiplexer has four data inputs (A, B, C and D) and one output F, as shown in Fig. 3.20. Two select signals (S_1 and S_0) are needed to choose among the four data inputs.

The 4-to-1 multiplexer Eq. (3.6) can be described by a truth table as in Table 3.10.

$$Y = A\bar{S}_1\bar{S}_0 + B.\bar{S}_1S_0 + CS_1\bar{S}_0 + DS_1S_0 \quad (3.6)$$

4-to-1 multiplexer can be built using multiple 2-to-1 multiplexers, as shown in Fig. 3.21.

Fig. 3.20 Symbol of a 4-to-1 multiplexer

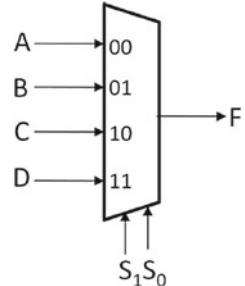
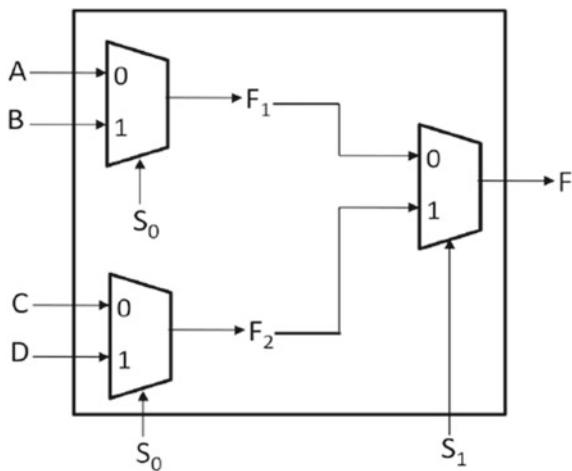


Table 3.10 4-to-1 multiplexer truth table

S_1	S_0	Y
0	0	A
0	1	B
1	0	C
1	1	D

Fig. 3.21 4-to-1 multiplexer implementation using multiple 2-to-1 multiplexers



3.3.2.1 VHDL Implementation

(a) Design a 4-to-1 multiplexer using structural modeling:

First, we examine the structural modeling, describing a module in terms of how it is composed of simpler modules. Listing 3.14 shows how to assemble a 4-to-1 multiplexer from three 2-to-1 multiplexers already defined. In the architecture:

- We declare Mux2_1 (2-to-1 multiplexer) as component using the component declaration statement,

- We declare two interconnection signals F1 and F2,
- We instantiate three copies of 2-to-1 multiplexers already defined using the Port mapping command.

```

1   library IEEE;
2   use IEEE.STD_LOGIC_1164.all;
3
4   entity Mux4_1 is
5   port( A,B,C,D: in std_logic;
6       S: in std_logic_vector(1 downto 0);
7       F: out std_logic);
8   end Mux4_1;
9
10  architecture Struct of Mux4_1 is
11
12  component Mux2_1 is
13
14  port( A,B,S: in std_logic;
15      F: out std_logic);
16  end component;
17
18  Signal F1, F2: std_logic;
19
20  Begin
21  M1: Mux2_1 port map(A,B,S(0),F1);
22  M2: Mux2_1 port map(C,D,S(0),F2);
23  M3: Mux2_1 port map(F1,F2,S(1),F);
24
25  end Struct;
```

Listing 3.14 VHDL code of the 4-to-1 multiplexer using structural modeling

(b) **Design a 4-to-1 multiplexer using conditional assignment:**

Conditional assignments select the output from among alternatives based on an input called the condition. As shown in Listing 3.15, A 4-to-1 multiplexer can select one of four inputs using multiple else clauses in the conditional signal assignment.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity Mux4_1 is
5  port(A, B,C, D: in STD_LOGIC;
6    s: in STD_LOGIC_VECTOR(1 downto 0);
7    F: out STD_LOGIC);
8  end;
9
10 architecture synth of Mx4_1 is
11 begin
12   F <= A when s = "00" else
13   B when s = "01" else
14   C when s = "10" else
15   D;
16   end synth;

```

Listing 3.15 VHDL code of the 4-to-1 multiplexer using When-else statement

(c) Design a 4-to-1 multiplexer using select with statement:

The 4-to-1 multiplexer can be rewritten with selected signal assignment as follows (Listing 3.16):

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity Mux4_1 is
5  port(A, B,C, D: in STD_LOGIC;
6    s: in STD_LOGIC_VECTOR(1 downto 0);
7    F: out STD_LOGIC);
8  end;
9
10 architecture synth of Mx4_1 is
11 begin
12   with s select F <=
13   A when "00",
14   B when "01",
15   C when "10",
16   D when others;
17
18   end synth;

```

Listing 3.16 VHDL code of the 4-to-1 multiplexer using When-select statement

3.3.2.2 Simulation

The simulation waveforms of the 4-to-1 multiplexer are shown in Fig. 3.22. The waveforms are staggered in their frequencies so as to make the largest contrast between



Fig. 3.22 Simulation waveform of the 4-to-1 multiplexer

adjacent waveforms. The results clearly show that the multiplexer transmits the data to the F output according to the select command.

3.3.2.3 Implementation and Validation on FPGA Platform

To test the design of the 4-to-1 multiplexer in the FPGA board, we use the switches SW0, SW1, SW2, SW3 as inputs, switch SW4 and SW5 as selection bits, and observing the result in the LED green [7]. Table 3.11 shows the pins assigned for 4-to-1 multiplexer.

The design of the 4-to-1 multiplexer has been verified according to the select command. As shown in Fig. 3.23, when the signal selector S1S0 is at 01, the input C switches to the output F and the associated green LED lights up.

Table 3.11 Pins assigned of the 4-to1 multiplexer

Inputs: A, B, C and D		Input select: S		Outputs: F	
SW[0]	PIN_AA23	SW[4]	PIN_AC26	LEDG[7]	PIN_AA24
SW[1]	PIN_AB26				
SW[2]	PIN_AB25		PIN_AC24		
SW[3]	PIN_AC27				

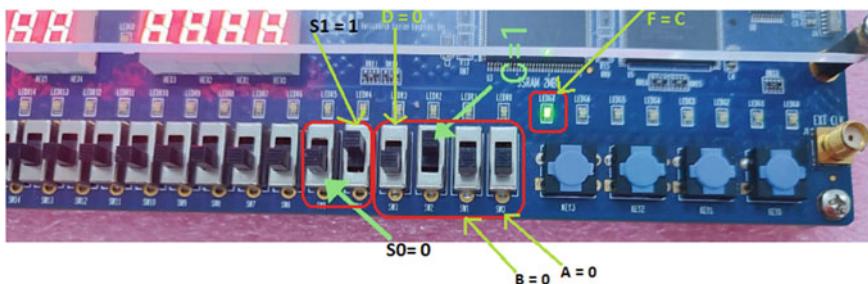


Fig. 3.23 Validation of the design of the 4-to-1 multiplexer on the FPGA board

3.3.3 8-to-1 Multiplexer

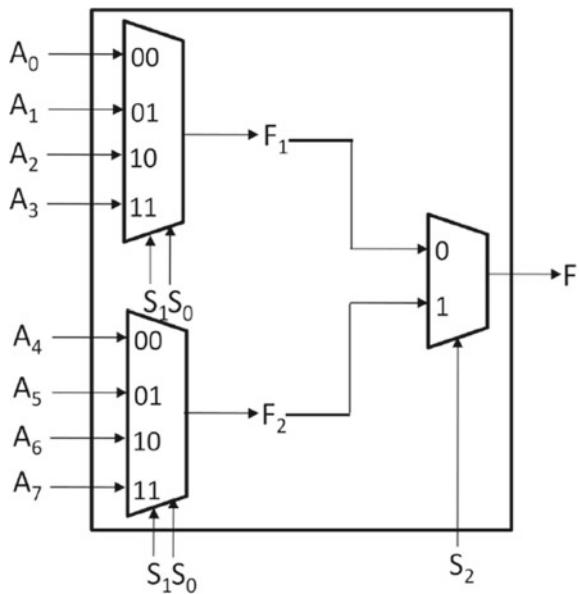
The 8-to-1 multiplexer has eight data inputs A[8:0], a three select input S₂, S₁, S₀, and one output F. The truth table of an 8-to-1 multiplexer is illustrated in Table 3.12.

Figure 3.24 shows how to assemble an 8-to-1 multiplexer from two 4-to-1 multiplexers together with one 2-to-1 multiplexer.

Table 3.12 8-to-1 multiplexer truth table

S ₂	S ₁	S ₀	F
0	0	0	A ₀
0	0	1	A ₁
0	1	0	A ₂
0	1	1	A ₃
1	0	0	A ₄
1	0	1	A ₅
1	1	0	A ₆
1	1	1	A ₇

Fig. 3.24 8-to-1 multiplexer implementation using 4-to-1 multiplexer together with 2-to-1 multiplexer



3.3.3.1 VHDL Implementation

As shown in Listing 3.17, we employ the VHDL structural modeling approach to build the 8-to-1 multiplexer from two copies of 4-to-1 multiplexers together with one 2-to-1 multiplexer.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity Mux8_1 is
5  port( A: in std_logic_vector(7 downto 0);
6    S: in std_logic_vector(2 downto 0);
7    F: out std_logic);
8  end Mux8_1;
9
10 architecture Structural of Mux8_1 is
11
12 component Mux2_1 is
13 port( A,B,S: in std_logic;
14   F: out std_logic);
15 end component;
16
17 component Mux4_1 is
18 port( A,B,C,D: in std_logic;
19   S: in std_logic_vector(1 downto 0);
20   F: out std_logic);
21 end component;
22
23 Signal F1, F2: std_logic;
24
25 Begin
26 M2: Mux4_1 port map(A(0),A(1),A(2),A(3),S(1 downto 0),F1);
27 M3: Mux4_1 port map(A(4),A(5),A(6),A(7),S(1 downto 0),F2);
28 M1: Mux2_1 port map(F1,F2,S(2),F);
29 end Structural;
```

Listing 3.17 VHDL code of the 8-to-1 multiplexer using structural modeling approach

3.3.3.2 Simulation

An example VHDL test bench for the design of 8-to-1 multiplexers shown in Listing 3.18. Here, all possible input combinations are applied changing every 20 ns. As shown in Fig. 3.25, for the selection lines (S2, S1, S0) “011” and “100”, the inputs A3 = 1 and A4 = 0 are available at the output F as it is clearly indicated in the truth Table 3.11 of 8-to-1 multiplexer. Likewise, other combinations of select lines from the truth table can be verified.



Fig. 3.25 Simulation waveforms of the 8-to-1 multiplexer

```

30   process
31   begin
32     stim_S <= "000";
33     stim_A <= "00000001";
34     wait for 20 ns;
35     stim_S <= "001";
36     stim_A <= "00000010";
37     wait for 20 ns;
38
39     stim_S <= "010";
40     stim_A <= "00000100";
41     wait for 20 ns;
42
43     stim_S <= "011";
44     stim_A <= "00001000";
45     wait for 20 ns;
46
47     stim_S <= "100";
48     stim_A <= "00010000";
49     wait for 20 ns;
50
51     stim_S <= "101";
52     stim_A <= "00100000";
53     wait for 20 ns;
54
55     stim_S <= "110";
56     stim_A <= "01000000";
57     wait for 20 ns;
58
59     stim_S <= "111";
60     stim_A <= "10000000";
61     wait for 20 ns;
62
63   end process;

```

Listing 3.18 Test bench code of the 8-to-1 multiplexer

3.3.3.3 Implementation and Validation on FPGA Platform

Figure 3.26 shows the validation of the design of the 8-to-1 Multiplexer on the FPGA board. When the selection lines S2S1S0 is equal to 011, the 8 to 1 multiplexer transmits the input A4 = 0 to the output F as clearly shown in Fig. 3.26. Likewise, other combinations of rows selected from the truth table can be verified.

3.4 Demultiplexer

A Demultiplexer (Demux) is a combinatorial circuit that operates in reverse of the multiplexer. It has one input and many outputs, so by applying control signal (select line S), we can route one input to one of its outputs. Unselected outputs produce a logic value “0”. A Demultiplexer of 2^N outputs has N select lines. In this lab, we illustrate the process of designing of 1-to-4 and 1-to-8 Demultiplexers using dataflow, behavioral and structural modeling in VHDL. Also, their designs are tested by simulation and implemented in the FPGA platform.

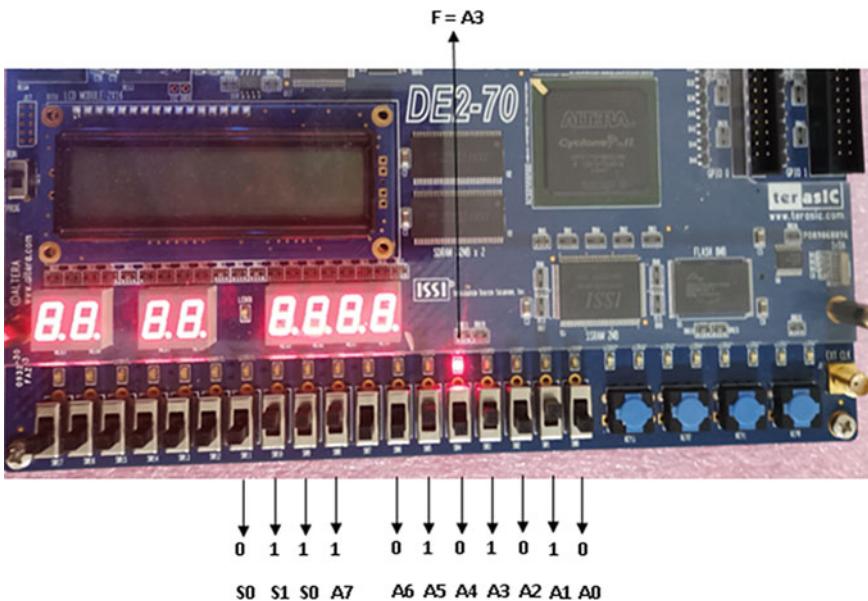


Fig. 3.26 Validation of the design of the 8-to-1 multiplexer on the FPGA board

3.4.1 1-to-4 Demultiplexer

A 1-to-4 demultiplexer has one input D, two selection inputs (S1 and S0) and four outputs (Y0 to Y3). Its symbol is shown in Fig. 3.27.

The operation of this Demultiplexer is illustrated in the truth Table 3.13. From this table, it is clearly shown that when S1 = 0 and S0 = 0, the data input is routed to output Y0 and when S1 = 0 and S0 = 1, then data output is routed to output Y1. Similarly, other outputs are connected to the input for other two combinations of the selected lines.

Figure 3.28 shows the logic circuit for a 1-to-4 Demultiplexer (four 3-input AND gates and two NOT gates) that distributes one input line to four output lines. Each AND gate in the Demultiplexer enables or inhibits the signal output according to the state of the select inputs, thus directing the data to one of the output lines. For instance, S1S0 = “10” directs incoming digital data to output Y2.

Fig. 3.27 Symbol of 1-to-4 demultiplexer

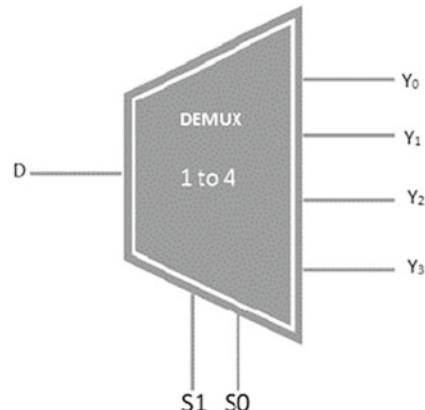
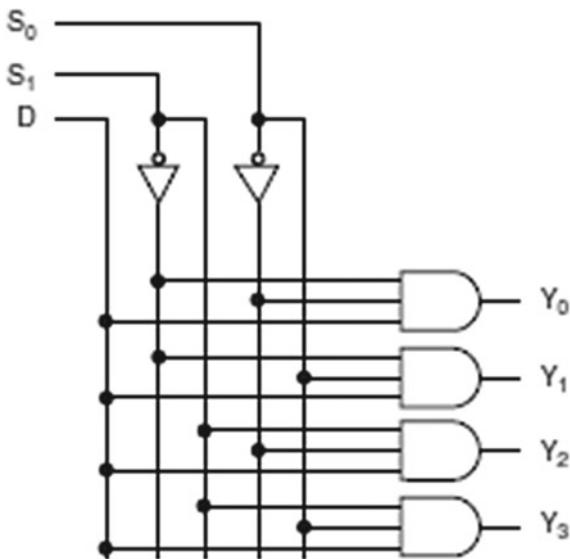


Table 3.13 Demultiplexer 1-to-4 truth table

Data input	Select input	Outputs				
D	S ₁	S ₀	Y ₀	Y ₁	Y ₂	Y ₃
D	0	0	D	0	0	0
D	0	1	0	D	0	0
D	1	0	0	0	D	0
D	1	1	0	0	0	D

Fig. 3.28 Logic circuit for a 1-to-4 demultiplexer



3.4.1.1 VHDL Implementation

The design of a 1-to-4 Demultiplexer in VHDL can be implemented using different modeling approaches such as logical operators, conditional signal assignments, and selected signal assignments.

The entity declaration for this design is defined as follows, Listing 3.19:

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity demux1_4 is
5    port(
6      D : in STD_LOGIC;
7      S: in STD_LOGIC_VECTOR(1 downto 0);
8      Y: out STD_LOGIC_VECTOR(3 downto 0));
9
10 end demux1_4;
11

```

Listing 3.19 Entity declaration for the 1-to-4 demultiplexer

The behavior of `demux1_4` can be modeled using the following three modeling approaches as shown Listing 3.20:

- `bhv1`: Behavioral modeling using the process statement. in this process, we define the operation of Demultiplexer 1-to-4 with two sequential statements: if-then-else and case-when

- **bhv2:** Data flow modeling using conditional signal assignment statements and Selected signal assignment statements.

Bhv1:

IF-THEN-ELSE statement:

```

12  ┌─ architecture bhv1 of demux1_4 is
13  ┌─ begin
14  ┌─ process (D,S) is
15  |─ begin
16  ┌─ if (S = "00") then
17  |─ Y(0) <= D; Y(1) <= '0'; Y(2) <= '0';Y(3) <= '0';
18  ┌─ elsif (S = "01") then
19  |─ Y(1) <= D; Y(0) <= '0'; Y(2) <= '0'; Y(3) <= '0';
20  ┌─ elsif (S = "10") then
21  |─ Y(2) <= D; Y(0) <= '0'; Y(1) <= '0';Y(3) <= '0';
22  ┌─ else
23  |─ Y(3) <= D; Y(0) <= '0'; Y(1) <= '0';Y(2) <= '0';
24  ┌─ end if;
25  ┌─ end process;
26  ┌─ end bhv1;
```

CASE-WHEN statement:

```

12  ┌─ architecture bhv1 of demux1_4 is
13  ┌─ begin
14  ┌─ process (D,S) is
15  |─ begin
16  ┌─ case (S) is
17  |─ when "00" => Y(0)<= D; Y(1) <= '0'; Y(2) <= '0'; Y(3) <= '0';
18  |─ when "01" => Y(0) <='0' ;Y(1) <= D; Y(2) <= '0'; Y(3) <= '0';
19  |─ when "10" => Y(0) <='0' ; Y(1) <= '0';Y(2) <= D; Y(3) <= '0';
20  |─ when "11" => Y(0) <='0' ; Y(1) <= '0'; Y(2) <= '0'; Y(3) <= D;
21  |─ when others =>null;
22  ┌─ end case;
23  ┌─ end process;
24  ┌─ end bhv1;
```

Bhv2:

Conditional assignment of the signal using when-else

```

12  architecture bhv2 of demux1_4 is
13  begin
14  | Y(0)<= D when "00" else '0';
15  | Y(1)<= D when "01" else '0';
16  | Y(2)<= D when "10" else '0';
17  | Y(3)<= D when "11" else '0';
18  | end bhv2;
19

```

Assignment of the selected signal using select-with:

```

12  architecture bhv2 of demux1_4 is
13  begin
14
15  | with (S) select
16  | Y(0)<= D when "00", '0' when others;
17  | with (S) select
18  | Y(1)<= D when "01", '0' when others;
19  | with (S) select
20  | Y(2)<= D when "10", '0' when others;
21  | with (S) select
22  | Y(3)<= D when "11", '0' when others;
23
24  | end bhv2;

```

Listing 3.20 VHDL code for 1-to-4 demultiplexer

3.4.1.2 Simulation

The VHDL test bench of Demultiplexer 1 to 4 and the simulation timing diagrams are respectively illustrated in Listing 3.21 and Fig. 3.29. According to the state of the select inputs, this circuit correctly performs the routing of one input to the four outputs. For example, $S_1 S_0 = 10$ directs digital data $D = 1$ to the active output $Y(2) = 1$, the other outputs remain inactive, $Y(0) = 0$, $Y(1) = 0$ and $Y(3) = 0$.

Likewise, the data item D is distributed to the other outputs $Y(0)$, $Y(1)$ and $Y(3)$ according to the state of the selection inputs $S_1 S_0$ presented.

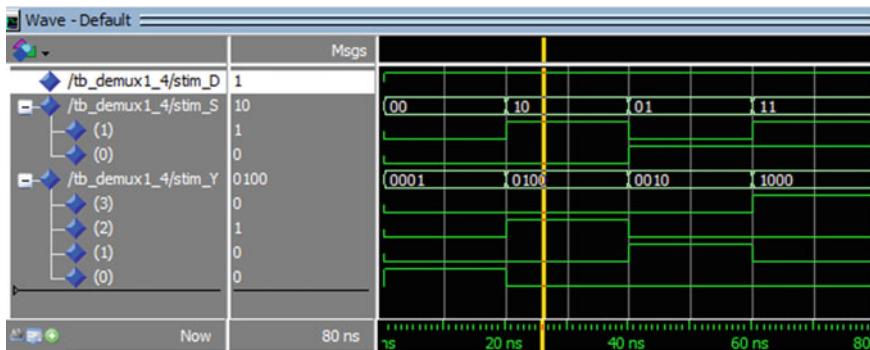


Fig. 3.29 Simulation waveforms of the 1-to-4 demultiplexer

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  ENTITY tb_Demux1_4 IS
5  END tb_Demux1_4;
6
7  ARCHITECTURE behavior OF tb_Demux1_4 IS
8
9  --Component Declaration for the Unit Under Test (UUT)
10
11 COMPONENT Demux1_4
12 PORT(
13   D : IN std_logic;
14   S : IN std_logic_vector (1 downto 0);
15   Y : OUT std_logic_vector(3 downto 0));
16 END COMPONENT;
17
18 --Inputs
19 signal stim_D : std_logic;
20 signal stim_S : std_logic_vector(1 downto 0);
21
22 --Outputs
23 signal stim_Y : std_logic_vector(3 downto 0);
24
25 BEGIN
26
27 --Instantiate the Unit Under Test (UUT)
28 uut: Demux1_4 PORT MAP (
29   D => stim_D,
30   S => stim_S,
31   Y => stim_Y);
32 
```

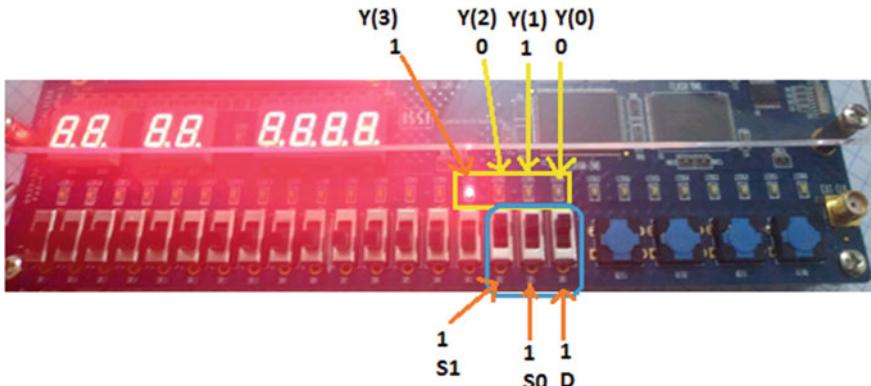
Listing 3.21 Test bench code of the 1-to-4 demultiplexer

3.4.1.3 Implementation and Validation on FPGA Platform

The design of 1-to-4 Demultiplexer has been implemented in the FPGA board, type DE2-70 Altera. To test its operation, we use switch SW0 as input, switches SW1 and

Table 3.14 Pins assigned of the 1-to-4 demultiplexer

Inputs: D		Input select: S (S_1S_0)		Outputs: Y(0), Y(1), Y(2), Y(3)	
SW[0]	PIN_AA23	SW[1]	PIN_AB26	LEDR[0]	PIN_AJ6
		SW[2]	PIN_AB25	LEDR[1]	PIN_AK5
				LEDR[2]	PIN_AJ5
				LEDR[3]	PIN_AJ4

**Fig. 3.30** Validation of the design of the 1-to-4 demultiplexer on the FPGA board

SW2 as select bits and observe the results in leads LEDR0, LEDR1, LEDR2 and LEDR3. Table 3.14 shows the pins assigned for 1-to-4 demultiplexer.

Depending on the state of the select command, the 1-to-4 Demultiplexer circuit performs the routing from input D to the four outputs. As shown in Fig. 3.30, when the signal selector is at “11”, the Demultiplexer switches to output Y(3), input D and the red LED[3] lights up, the other outputs remain inactive, Y(0) = 0, Y(1) = 0 and Y(2) = 0 and the corresponding LEDs remain off.

3.4.2 1-to-8 Demultiplexer

The 1-to-8 Demultiplexer has a single D data input and eight outputs. It has three select lines S_2, S_1, S_0 which are used as the control signals. Figure 3.31 shows the logic diagram for this Demultiplexer. The data bit at input D is transferred to one of the eight outputs depending upon the states of the select lines. The truth Table 3.15 summarizes the operation of this Demultiplexer. As shown in this table, when $S_2S_1S_0 = "000"$, only the first AND gate is enabled, and data input D has been appeared at the output Y_0 . Other SELECT codes cause input D to reach the other outputs.

Fig. 3.31 Logic circuit for a 1-to-8 demultiplexer

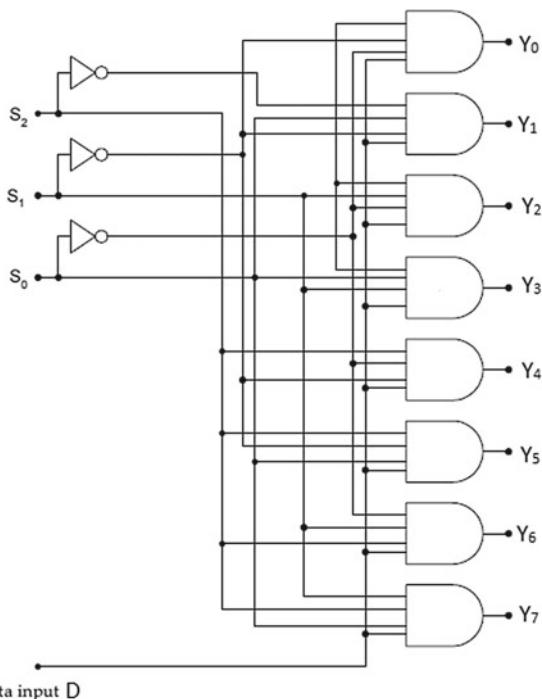


Table 3.15 Demultiplexer 1-to-8 truth table

Data input	Select input			Outputs								
	D	S ₂	S ₁	S ₀	Y ₇	Y ₆	Y ₅	Y ₄	Y ₃	Y ₂	Y ₁	Y ₀
D	0	0	0	0	0	0	0	0	0	0	0	D
D	0	0	1	0	0	0	0	0	0	0	D	0
D	0	1	0	0	0	0	0	0	0	D	0	0
D	0	1	1	0	0	0	0	0	D	0	0	0
D	1	0	0	0	0	0	0	D	0	0	0	0
D	1	0	1	0	0	0	D	0	0	0	0	0
D	1	1	0	0	D	0	0	0	0	0	0	0
D	1	1	1	D	0	0	0	0	0	0	0	0

3.4.2.1 VHDL Implementation

To build the 1-to-8 Demultiplexer, we chose the behavioral modeling approach using the CASE statement, but this statement must be placed inside a process statement. Listing 3.22 shows the VHDL code for the 1-to-8 Demultiplexer.

```

1      library IEEE;
2      use IEEE.STD_LOGIC_1164.ALL;
3
4      entity Demux1_8 is
5          port(
6              D:in std_logic;
7              S:in std_logic_vector(2 downto 0);
8              Y:out std_logic_vector(7 downto 0));
9          end Demux1_8;
10
11     architecture behavioral of Demux1_8 is
12     begin
13         process (D, S)
14         begin
15             Y <= "00000000";
16             case S is
17                 when "000" => Y(0) <= D;
18                 when "001" => Y(1) <= D;
19                 when "010" => Y(2) <= D;
20                 when "011" => Y(3) <= D;
21                 when "100" => Y(4) <= D;
22                 when "101" => Y(5) <= D;
23                 when "110" => Y(6) <= D;
24                 when "111" => Y(7) <= D;
25                 when others => Y <= "00000000";
26             end case;
27         end process;
28     end behavioral;

```

Listing 3.22 VHDL code for 1-to-8 demultiplexer

Simulation

The VHDL test bench code for the 1-to-8 Demultiplexer is shown in the Listing 3.23. Figure 3.32 shows that the data D at the input of the Demultiplexer 1–8 is transferred to one of its eight outputs according to the states of the selection lines. For example, when $S_2S_1S_0 = "101"$, data input D has been appeared at the output Y(5), other outputs remain inactive. Other SELECT codes cause input D to reach the other outputs.

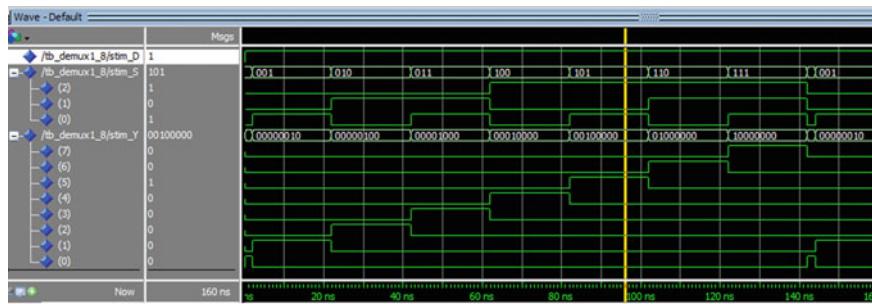


Fig. 3.32 Simulation waveforms of the 1-to-8 demultiplexer

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  ENTITY tb_Demux1_8 IS
5  END tb_Demux1_8;
6
7  ARCHITECTURE behavior OF tb_Demux1_8 IS
8
9    --Component Declaration for the Unit Under Test (UUT)
10
11   COMPONENT Demux1_8
12   PORT(
13     D : IN std_logic;
14     S : IN std_logic_vector (2 downto 0);
15     Y : OUT std_logic_vector(7 downto 0));
16   END COMPONENT;
17
18   --Inputs
19   signal stim_D : std_logic := '0';
20   signal stim_S : std_logic_vector (2 downto 0) := "000";
21
22   --Outputs
23   signal stim_Y : std_logic_vector(7 downto 0);
24
25 BEGIN
26
27   --Instantiate the Unit Under Test (UUT)
28   uut: Demux1_8 PORT MAP (
29     D => stim_D,
30     S => stim_S,
31     Y => stim_Y);
32

```

```

33  | --Stimulus process
34  |  stim_proc: process
35  |    begin
36
37    stim_D <= '1';
38
39    stim_S <= "000";
40    wait for 2 ns;
41
42    stim_S <= "001";
43    wait for 20 ns;
44
45    stim_S <= "010";
46    wait for 20 ns;
47
48    stim_S <= "011";
49    wait for 20 ns;
50
51    stim_S <= "100";
52    wait for 20 ns;
53
54    stim_S <= "101";
55    wait for 20 ns;
56
57    stim_S <= "110";
58    wait for 20 ns;
59
60    stim_S <= "111";
61    wait for 20 ns;
62
63  end process;
64  END behavior;

```

Listing 3.23 Test bench code of the 1-to-8 demultiplexer

Implementation and Validation on FPGA Platform

The design of the 8-to-1 Demultiplexer has been tested and validated on an FPGA platform, type DE2-70 Altera. As shown in Fig. 3.33, when the signal selector S2S1S0 is at “011”, the data input D has been appeared at the output Y(3) and the red LED[3] lights up, the other outputs remain inactive and the corresponding LEDs remain off.

3.5 Decoder

A Decoder is a digital circuit used to decode the coded digital information. As shown in Fig. 3.34, a decoder has N inputs and 2^N outputs. Since each of the N inputs can be 0 or 1, there are $2N$ possible input combinations or codes. It asserts exactly one of its outputs depending on the input combination. They are widely used in

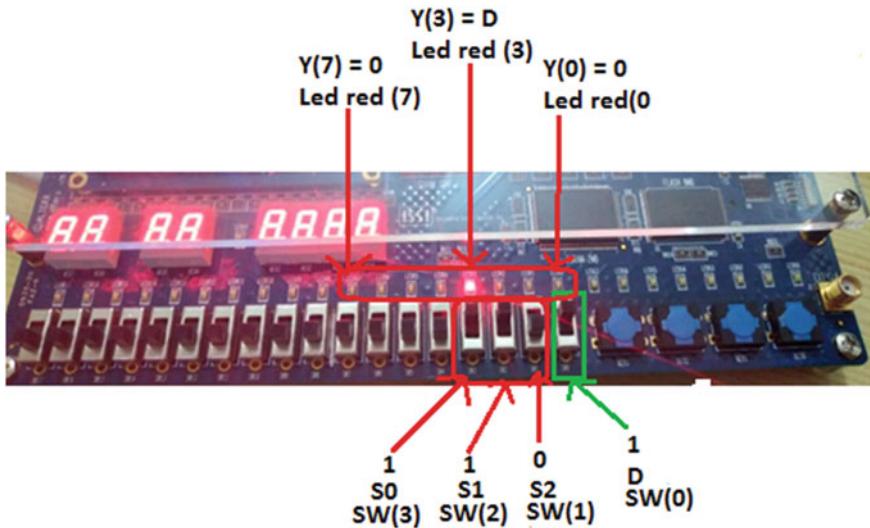


Fig. 3.33 Validation of the design of the 1-to-8 demultiplexer on the FPGA board

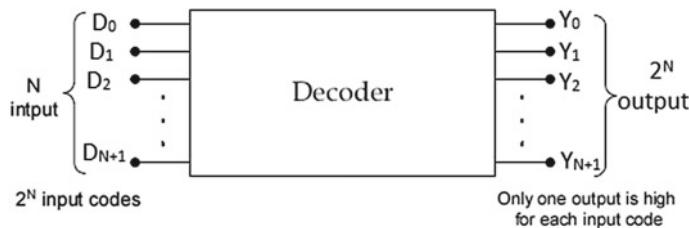


Fig. 3.34 Decoder diagram

many applications like data multiplexing, 7-segment display and memory address decoding.

3.5.1 2-To-4 Decoder

Figure 3.35 shows the logic circuit of a 2-to-4 decoder. The circuit detects the presence of a particular binary code and makes one and only one output HIGH, depending on the value of the 2-bit number D1D0, as shown by the truth Table 3.16. For example, if D1D0 = “10”, the output Y2 is active since 10 (binary) = 2 (decimal).

Fig. 3.35 Logic circuit of 2-to-4 decoder

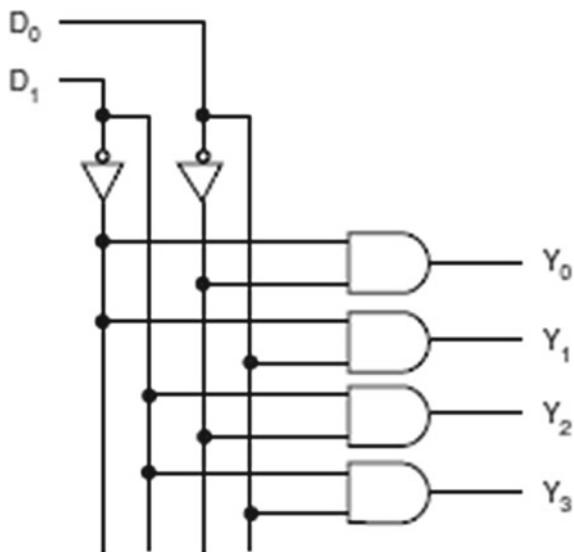


Table 3.16 Truth table of 2-to-4 decoder

Inputs		Outputs			
D ₁	D ₀	Y ₀	Y ₁	Y ₂	Y ₃
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

3.5.1.1 VHDL Implementation

To describe the behavior of the decoder circuit in VHDL, we can use the two types of simultaneous signal assignments: the selected signal assignment statement and the conditional signal assignment statement. The following VHDL code (see Listing 3.24) implements a 2-to-4 decoder using a selected signal assignment statement: “*With/ Select*”.

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  ENTITY Decoder2_4 IS
5  PORT(
6    d : IN STD_LOGIC_VECTOR (1 downto 0);
7    y : OUT STD_LOGIC_VECTOR (3 downto 0));
8  END Decoder2_4;
9
10 ARCHITECTURE behavioral OF Decoder2_4 IS
11
12 BEGIN
13
14  WITH d SELECT
15  y <= "0001" WHEN "000",
16    "0010" WHEN "001",
17    "0100" WHEN "010",
18    "1000" WHEN "011",
19    "0000" WHEN others;
20
21 END behavioral;
22

```

Listing 3.24 VHDL code for 2-to-4 decoder using a selected signal assignment statement: *With/Select*

The “*when/else*” conditional statement is another way to describe in VHDL the 2-to-4 decoder, see Listing 3.25. Note that only one decoder output is activated, depending on the value of inputs D.

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3
4  ENTITY Decoder2_4 IS
5  PORT(
6    d : IN STD_LOGIC_VECTOR (0 to 1);
7    y : OUT STD_LOGIC_VECTOR (0 to 3));
8  END Decoder2_4;
9
10 ARCHITECTURE behavioral OF Decoder2_4 IS
11
12 BEGIN
13
14  y <= "1000" WHEN (d= "00" ) ELSE
15    "0100" WHEN (d= "01") ELSE
16    "0010" WHEN (d= "10") ELSE
17    "0001" WHEN (d= "11") ELSE
18    "0000";
19
20 END behavioral;
21

```

Listing 3.25 VHDL code for 2-to-4 decoder using a conditional signal assignment statement: *When/Else*

3.5.1.2 VHDL Implementation

The test bench code for the 2-to-4 Decoder is illustrated in the Listing 3.26. Figure 3.36 shows the simulation waveforms for the 2-line to 4-line decoder. The inputs D1 and D0 are combined as a single 2-bit D value, to which an increasing binary count is applied as a stimulus. The decoder output waveforms are observed individually to determine the Decoder's response. For example, when D1D0 = "01", the output Y1 is active and the values assigned to Y are: Y(3) = 0, Y(2) = 0, Y(1) = 1, Y(0) = 0. Similarly, the operation of 2-to-4 decoder is checked for the other combinations of inputs as indicated in the truth Table 3.17.

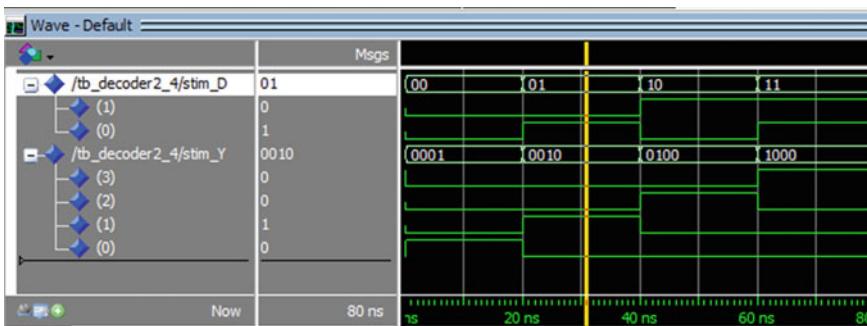


Fig. 3.36 Simulation waveforms of 2-to-4 decoder

Table 3.17 Pins assigned of the 2-to-4 decoder

Inputs: D1, D0		Outputs: Y(0), Y(1), Y(2), Y(3)	
SW[0]	PIN_AA23	LEDG[4]	PIN_Y24
SW[1]	PIN_AB26	LEDG[5]	PIN_Y23
		LEDR[6]	PIN_AA27
		LEDR[7]	PIN_AA24

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity tb_Decoder2_4 is
5  end tb_Decoder2_4;
6
7  architecture behavior of tb_Decoder2_4 is
8
9  --Component Declaration for the Unit Under Test (UUT)
10 component Decoder2_4 is
11  Port ( D : in STD_LOGIC_VECTOR (1 downto 0);
12    Y : out STD_LOGIC_VECTOR (3 downto 0));
13  end component;
14
15  -- Inputs
16  signal stim_D: STD_LOGIC_VECTOR(1 downto 0);
17  -- Outputs
18  signal stim_Y: STD_LOGIC_VECTOR(3 downto 0);
19
20 begin
21
22  --Instantiate the Unit Under Test (UUT)
23  uut: Decoder2_4 port map(
24    D => stim_D, Y => stim_Y);
25
```

```
26  -- Stimulus process
27  stim: process
28  begin
29
30    stim_D <= "00";
31    wait for 20 ns;
32
33    stim_D <= "01";
34    wait for 20 ns;
35
36    stim_D <= "10";
37    wait for 20 ns;
38
39    stim_D <= "11";
40    wait for 20 ns;
41    wait;
42
43  end process;
44
45  end behavior;
```

Listing 3.26 Test bench code for the 2-to-4 decoder

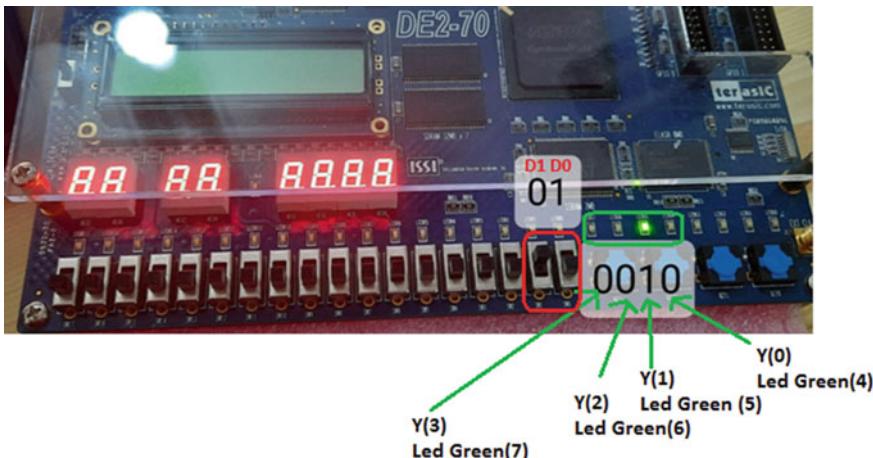


Fig. 3.37 Validation of the design of the 2-to-4 decoder on the FPGA board

Implementation and Validation on FPGA Platform

Before we can upload the 2-to-4 Decoder circuit into the hardware, we need to assign the input and output pin numbers on the target FPGA. At this point, we can recompile the design file and program the FPGA. Table 3.17 shows the pins assigned for 2-to-4 Decoder.

As shown in Fig. 3.37, when $D1D0 = 01$, output $Y2$ is active, the Green LED [5] lights up, the other outputs remain inactive, $Y(0) = 0$, $Y(1) = 0$ and $Y(3) = 0$ and the corresponding LEDs remain off.

3.5.2 3-to-8 Decoder

Figure 3.38 depicts the circuit for a 3-line-to-8-line Decoder, again with an active-LOW enable, G. In this case, the decoder outputs are active LOW. One and only one output is active for any given combination of $D2D1D0$. Table 3.18 shows the truth table for this decoder. Again if the enable line is HIGH, no output is active.

3.5.2.1 VHDL Implementation

There are other ways to write the 3-to-8 Decoder in VHDL using a case statement and selected signal assignment. The following VHDL code (Listing 3.27) is written for the 3-to-8 decoder using case statement:

Fig. 3.38 Logic circuit of 3-to-8 decoder

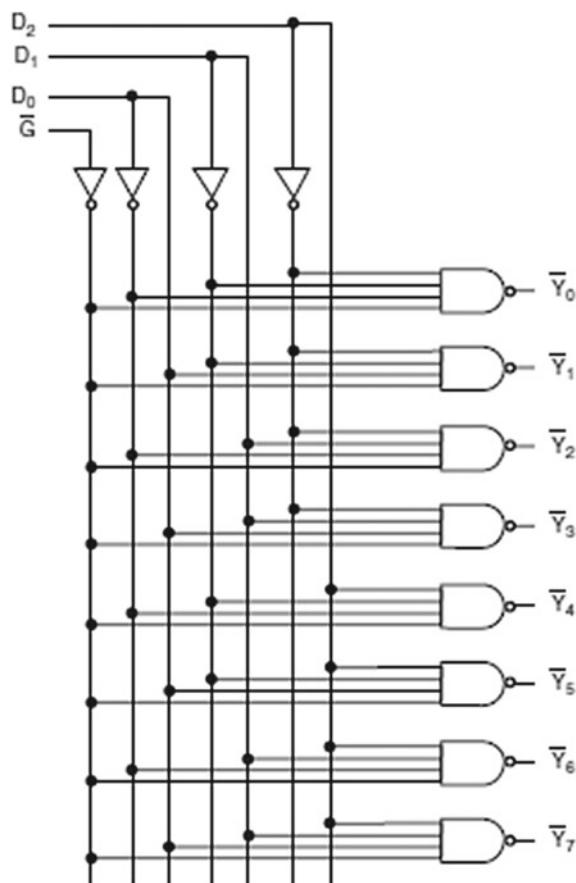


Table 3.18 Truth table of a 3-to-8 decoder with enable

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity Decoder3_8 is
5    Port ( D : in std_logic_vector(2 downto 0);
6           Y : out std_logic_vector(7 downto 0));
7  end Decoder3_8;
8
9  architecture behavioral of Decoder3_8 is
10
11 begin
12
13 process (D)
14
15 begin
16 case D is
17 when "000" => Y <= "00000001";
18 when "001" => Y <= "00000010";
19 when "010" => Y <= "00000100";
20 when "011" => Y <= "00001000";
21 when "100" => Y <= "00010000";
22 when "101" => Y <= "00100000";
23 when "110" => Y <= "01000000";
24 when "111" => Y <= "10000000";
25 when others => Y <= "XXXXXXXX";
26 end case;
27
28 end process ;
29
30 end behavioral;
31
```

Listing 3.27 VHDL code for 3-to-8 decoder using a CASE statement

The following VHDL code is written for 3-to-8 decoder using with-select construct (see Listing 3.28):

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all, IEEE.NUMERIC_STD.all;
3  use IEEE.STD_LOGIC_ARITH.all;
4  use IEEE.STD_LOGIC_UNSIGNED.all;
5
6  entity Decoder3_8 is
7  Port ( D : in std_logic_vector(2 downto 0);
8  Y : out std_logic_vector(7 downto 0));
9  end Decoder3_8;
10
11 architecture behavioral of Decoder3_8 is
12
13 begin
14
15  with D select
16
17  Y <= "00000001" when "000",
18  "00000010" when "001",
19  "00000100" when "010",
20  "00001000" when "011",
21  "00010000" when "100",
22  "00100000" when "101",
23  "01000000" when "110",
24  "10000000" when "111",
25  "XXXXXXXX" when others;
26
27 end behavioral;

```

Listing 3.28 VHDL code for 3-to-8 decoder using with-select construct

The conditional signal assignment instruction assigns a value to all bits of the Y vector. Note that only one output is enabled for any given combination of D2D1D0.

3.5.2.2 Simulation

The timing diagrams in Fig. 3.39 confirms the correct operation of 3-to-8 Decoder as indicated in its truth Table 3.17. For example, if D2D1D0 = “001”, the 3-to-8

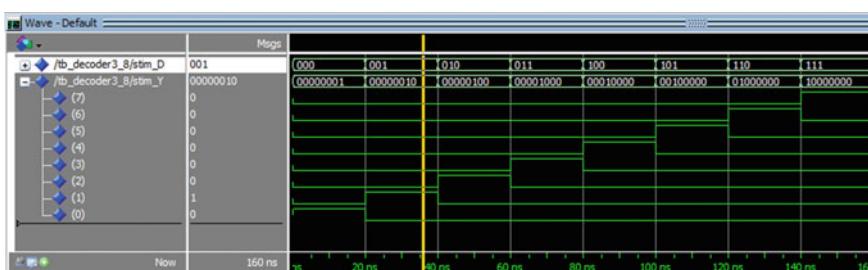


Fig. 3.39 Simulation waveforms for 3-to-8 decoder

decoder activates the output $Y_2 = 1$ and the output $Y = "00000010"$. Similarly, the correct functioning of the decoder is checked for the other combinations of the D data. Listing 3.29 shows the Test Bench code for the 3-to-8 Decoder.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity tb_Decoder3_8 is
5  end tb_Decoder3_8;
6
7  architecture beh of tb_Decoder3_8 is
8
9  component Decoder3_8
10 end component;
11
12 port (
13     D: in std_logic_vector(2 downto 0);
14     Y: out std_logic_vector(7 downto 0));
15
16 -- inputs:
17 signal stim_D : std_logic_vector(2 downto 0);
18 -- Outputs:
19 signal stim_Y : std_logic_vector(7 downto 0);
20
21 begin
22
23 -- Instantiate the Unit Under Test (UUT)
24 uut : Decoder3_8 port map (
25     D => stim_D,
26     Y => stim_Y);
27

```

```

27  Stim: process
28
29 begin
30     stim_D <="000"; --input = 0.
31     wait for 20 ns;
32     stim_D <="001"; --input = 1.
33     wait for 20 ns;
34     stim_D <="010"; --input = 2.
35     wait for 20 ns;
36     stim_D <="011"; --input = 3.
37     wait for 20 ns;
38     stim_D <="100"; --input = 4.
39     wait for 20 ns;
40     stim_D <="101"; --input = 5.
41     wait for 20 ns;
42     stim_D <="110"; --input = 6.
43     wait for 20 ns;
44     stim_D <="111"; --input = 7.
45     wait for 20 ns;
46
47 end process;
48
49 end beh;

```

Listing 3.29 Test bench code for the 3-to-8 decoder

3.5.3 BCD-To-Decimal Decoder

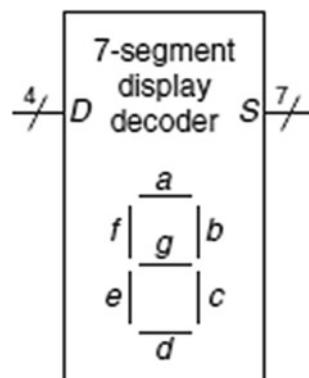
A BCD-to-seven-segment decoder is a combinational circuit that accepts a 4-bit BCD input and produces seven outputs to control the light emitting diodes to display a digit from 0 to 9. For example, the input digit = “0000” must illuminate segments a, b, c, d, e, and f to display the digit 0. We can make a truth Table 3.19 for each of the outputs, showing which segment must be active for every digit we wish to display.

Figure 3.40 shows the seven outputs which are often referred to as segments a through g. The numeric display chosen to represent the decimal digit is shown Fig. 3.41.

Table 3.19 Truth table for BCD-to-seven segment decoder

BCD input				Seven-segment decoder						
A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
All other inputs				0	0	0	0	0	0	0

Fig. 3.40 Seven segment display decoder



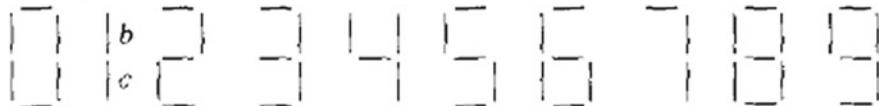


Fig. 3.41 Numeric designation for display

3.5.3.1 VHDL Implementation

In this lab, we use two approaches to describe in VHDL a BCD to 7 segment display decoder based on its truth table such as when-else statement and case-when statement.

Approach N°1: Case statement

The following VHDL code (see Listing 3.30) uses case statement to describe a BCD to 7 segment display decoder. The case statement performs different actions depending on the value of its input. A case statement implies combinational logic if all possible input combinations are considered; otherwise it implies sequential logic because the output keep its old value in the undefined cases.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity bcd_7segment is
5    Port ( BCD : in STD_LOGIC_VECTOR (3 downto 0);
6           Seven_Segment : out STD_LOGIC_VECTOR (6 downto 0));
7  end bcd_7segment;
8
9  architecture Behavioral of bcd_7segment is
10
11 begin
12
13 process(BCD)
14 begin
15
16 case BCD is
17   -----
18   when "0000" => Seven_Segment <= "0000001"; -- 0
19   when "0001" => Seven_Segment <= "1001111"; -- 1
20   when "0010" => Seven_Segment <= "0011001"; -- 2
21   when "0011" => Seven_Segment <= "0000110"; -- 3
22   when "0100" => Seven_Segment <= "1001100"; -- 4
23   when "0101" => Seven_Segment <= "0100100"; -- 5
24   when "0110" => Seven_Segment <= "0100000"; -- 6
25   when "0111" => Seven_Segment <= "0001111"; -- 7
26   when "1000" => Seven_Segment <= "0000000"; -- 8
27   when "1001" => Seven_Segment <= "0000100"; -- 9
28   when others => Seven_Segment <= "1111111"; --
29   end case;
30
31 end process;
32
33 end Behavioral;
```

Listing 3.30 VHDL code for BCD to 7 segment display using CASE statement

The case statement checks the value of data. When data is 0, the statement performs the action after the \Rightarrow , setting segments to “1111110”. The case statement similarly checks other data values up to 9 (note the use of X for hexadecimal numbers). The others clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.

Approach N 2: When else statement.

The following VHDL code (see Listing 3.31) uses *When-else* statement to design a BCD to 7 segment display decoder.

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  ENTITY bcd_7segment IS
5  PORT (BCD :IN std_logic_vector(3 downto 0);
6        Seven_Segment :OUT std_logic_vector(6 downto 0) );
7  END bcd_7segment;
8
9  ARCHITECTURE behavior OF bcd_7segment IS
10
11 BEGIN
12
13    Seven_Segment <= "1000000" when BCD="0000" else
14    "1111001" when BCD ="0001" else
15    "0100100" when BCD ="0010" else
16    "0011000" when BCD ="0011" else
17    "0011001" when BCD ="0100" else
18    "0010010" when BCD ="0101" else
19    "0000010" when BCD ="0110" else
20    "1111000" when BCD ="0111" else
21    "0000000" when BCD ="1000" else
22    "0010000" ;
23
24  END behavior;
25

```

Listing 3.31 VHDL code for BCD to 7 segment display decoder using WHEN-ELSE statement

3.5.3.2 Simulation

Figure 3.42 depicts an example of the timing diagram of BCD-to-7 segment display. In this example, the decoder takes a 4-bit number and displays its decimal value on the segments. For example, the number “0111” = 7 should turn on segments a, b, and c: the number displayed as output is “0001111”. Once the BCD input number



Fig. 3.42 Waveform for BCD to 7-segment display decoder

changes, the decoder performs the necessary conversions again and maps the result to the appropriate 7-segment outputs.

The following VHDL code (see Listing 3.32) shows the test bench for BCD to 7-Segment Display Decoder.

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  ENTITY tb_bcd_7segment IS
5  END tb_bcd_7segment;
6
7  ARCHITECTURE behavior OF tb_bcd_7segment IS
8
9  --Component Declaration for the Unit Under Test (UUT)
10
11 COMPONENT bcd_7segment
12 PORT(
13   BCD : IN std_logic_vector(3 downto 0);
14   Seven_Segment : OUT std_logic_vector(6 downto 0)
15 );
16 END COMPONENT;
17
18 --Inputs
19 signal stim_BCD : std_logic_vector(3 downto 0) := (others => '0');
20
21 --Outputs
22 signal stim_Seven_Segment : std_logic_vector(6 downto 0);
23
24 BEGIN
25
26 -- Instantiate the Unit Under Test (UUT)
27 uut: bcd_7segment PORT MAP (
28   BCD => stim_BCD,
29   Seven_Segment => stim_Seven_Segment
30 );
31

```

```

32  |-- Stimulus process
33  |stim_proc: process
34  |begin
35  |  stim_BCD <= "0000";
36  |  wait for 100 ns;
37
38  |  stim_BCD <= "0001";
39  |  wait for 100 ns;
40
41  |  stim_BCD <= "0010";
42  |  wait for 100 ns;
43
44  |  stim_BCD <= "0011";
45  |  wait for 100 ns;
46
47  |  stim_BCD <= "0100";
48  |  wait for 100 ns;
49  |  stim_BCD <= "0101";
50  |  wait for 100 ns;
51
52  |  stim_BCD <= "0110";
53  |  wait for 100 ns;
54
55  |  stim_BCD <= "0111";
56  |  wait for 100 ns;
57
58  |  stim_BCD <= "1000";
59  |  wait for 100 ns;
60
61  |  stim_BCD <= "1001";
62  |  wait for 100 ns;
63  |end process;
64  |END behavior;

```

Listing 3.32 Test bench code for BCD to 7 Segment Display Decoder

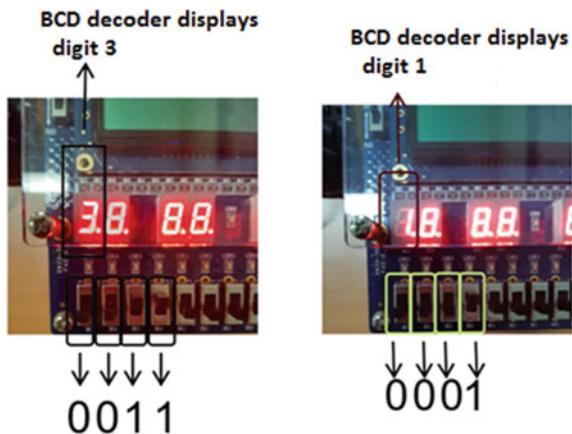
3.5.3.3 Implementation and Validation on FPGA Platform

To test the operation of the BCD-to-7 segment decoder on the FPGA, DE2 board, we connect the BCD inputs to switches SW3, SW2, SW1, SW0, and connect the outputs of the decoder to the HEX0 display. The segments in this display are called HEX0_D[0], HEX0_D[1]–HEX0_D[6], corresponding to the Fig. 3.20. Table 3.20 shows the pins assigned for BCD-to-7 segment decoder.

As shown in Fig. 3.43, when the input digit = “0111”, illuminate segments d, e, f and g to display the digit 7 and when input digit = “0001”, BCD decoder displays digit 1.

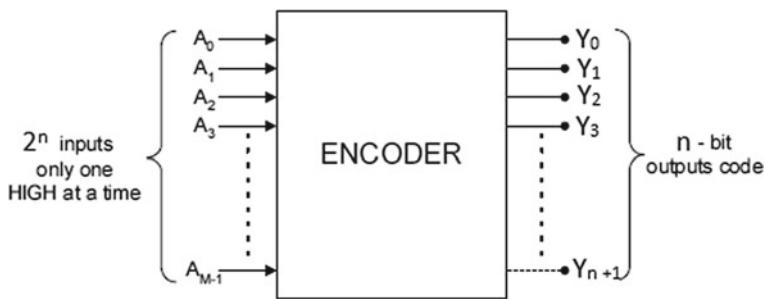
Table 3.20 Pins assigned of BCD-to-7 segment decoder

Inputs BCD: BCD[0], BCD[1], BCD[2], BCD[3]		Outputs: Seven_segment(0), Seven_segment (1), Seven_segment (2), Seven_segment (3), Seven_segment (4), Seven_segment (5), Seven_segment (6)	
SW[0]	PIN_AA23	HEX0_D[0]	PIN_AE8
SW[1]	PIN_AB26	HEX0_D[1]	PIN_AF9
		HEX0_D[2]	PIN_AH9
		HEX0_D[3]	PIN_AD10
SW[2]	PIN_AC27	HEX0_D[4]	PIN_AF10
		HEX0_D[5]	PIN_AD11
		HEX0_D[6]	PIN_AD12

Fig. 3.43 Validation of the design of the BCD-to-7 segment decoder on the FPGA board

3.6 Encoder

An encoder is a digital circuit that performs the inverse operation of a decoder. As shown in Fig. 3.44, an encoder has 2^N inputs and N outputs. It activates a specified output for a unique digital input code. Encoders are used to reduce the number of bits needed to represent given information. A practical use of encoders is for transmitting information in a digital system. Encoding the information allows the transmission link to be built using fewer wires. Encoding is also useful if information is being stored for later use, because fewer bits need to be stored.

**Fig. 3.44** Block diagram of an encoder

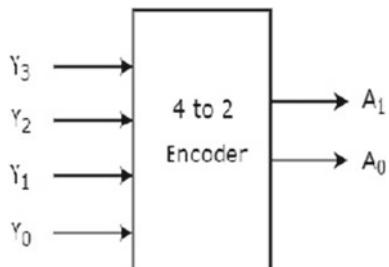
3.6.1 Binary Encoder

Figure 3.45 shows a 4-to-2 binary encoder. This device has four inputs \$Y_3, Y_2, Y_1\$ and \$Y_0\$ and two outputs \$A_1\$ and \$A_0\$.

At any time, only one of these 4 inputs can be '1' in order to generate the respective binary code at the output. The truth Table 3.21 shows the allowable input states, which yield the Boolean equations used to design the encoder.

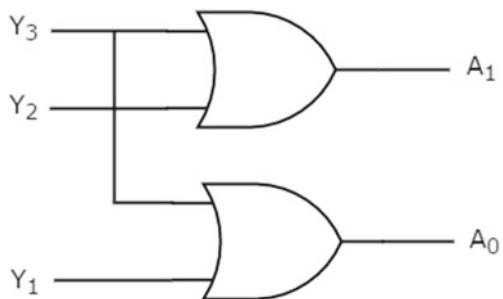
These Boolean equations are:

$$A_1 = Y_3 + Y_2 \quad (3.7)$$

Fig. 3.45 Block diagram of 4-to-2 encoder**Table 3.21** 4-to-2 encoder truth table

Inputs				Outputs	
\$Y_3\$	\$Y_2\$	\$Y_1\$	\$Y_0\$	\$A_1\$	\$A_0\$
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Fig. 3.46 Logic circuit for a 4-to-2 encoder



$$A_0 = Y_3 + Y_1 \quad (3.8)$$

The 4-to-2 Encoder can be implemented with two input OR gates, as shown in the Fig. 3.46.

3.6.1.1 VHDL Implementation

VHDL Code for 4-to-2 encoder can be done in different methods like using case statement, using if-else statement and using logic gates.

Method 1: Case statement (see Listing 3.33)

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity Encoder4_2 is
5  port(
6  Y : in STD_LOGIC_VECTOR(3 downto 0);
7  A : out STD_LOGIC_VECTOR(1 downto 0)
8  );
9  end Encoder4_2;
10
11 architecture bhv of Encoder4_2 is
12 begin
13
14 process(Y)
15 begin
16 case Y is
17 when "1000" => A <= "00";
18 when "0100" => A <= "01";
19 when "0010" => A <= "10";
20 when "0001" => A <= "11";
21 when others => A <= "ZZ";
22 end case;
23 end process;
24
25 end bhv;
  
```

Listing 3.33 VHDL code for 4-to-2 encoder using a case statement

Method 2: If-then-else statement (see listing 3.34)

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity Encoder4_2 is
5    port(
6      Y : in STD_LOGIC_VECTOR(3 downto 0);
7      A : out STD_LOGIC_VECTOR(1 downto 0)
8    );
9  end Encoder4_2;
10
11 architecture bhv of Encoder4_2 is
12 begin
13
14 process(Y)
15 begin
16   if (Y="1000") then
17     A <= "00";
18   elsif (Y="0100") then
19     A <= "01";
20   elsif (Y="0010") then
21     A <= "10";
22   elsif (Y="0001") then
23     A <= "11";
24   else
25     A <= "ZZ";
26   end if;
27 end process;
28
29 end bhv;

```

Listing 3.34 VHDL code for 4-to-2 encoder using if-then-else statement

Method 3: Logic gates statement (see Listing 3.35).

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity Encoder4_2 is
5    port(
6      Y : in STD_LOGIC_VECTOR(3 downto 0);
7      A : out STD_LOGIC_VECTOR(1 downto 0)
8    );
9  end Encoder4_2;
10
11 architecture bhv of Encoder4_2 is
12 begin
13
14   A(0) <= Y(1) or Y(2);
15   A(1) <= Y(1) or Y(3);
16
17 end bhv;

```

Listing 3.35 VHDL code for 4-to-2 encoder using a logic gates statement

3.6.1.2 Simulation

A test bench VHDL code for the 4-to-2 Encoder is shown in Listing 3.36. Here, all possible input combinations are applied changing every 20 ns. The simulations result in Fig. 3.47 confirm clearly the correct operation of the 4-to-2 Encoder.

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  ENTITY tb_encoder4_2 IS
5  END tb_encoder;
6
7  ARCHITECTURE behavior OF tb_encoder4_2 IS
8
9  --Component Declaration for the Unit Under Test (UUT)
10
11 COMPONENT encoder4_2
12 PORT(
13   Y : IN std_logic_vector(3 downto 0);
14   A : OUT std_logic_vector(1 downto 0)
15 );
16 END COMPONENT;
17
18 --Inputs
19 signal stim_Y : std_logic_vector(3 downto 0) := (others => '0');
20
21 --Outputs
22 signal stim_A : std_logic_vector(1 downto 0);
23
24 BEGIN
25
26 --Instantiate the Unit Under Test (UUT)
27 uut: encoder PORT MAP (
28   Y => stim_Y,
29   A => stim_A
30 );
31

```

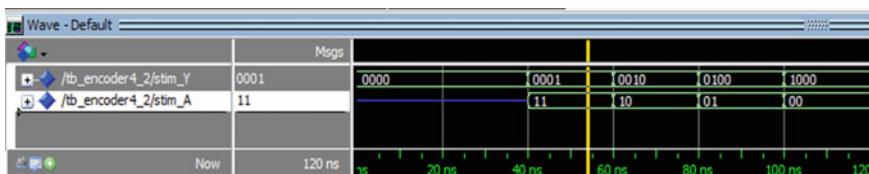


Fig. 3.47 Simulation waveforms of the 4-to-2 encoder

```

32  | --Stimulus process
33  | @stim_proc: process
34  | begin
35  |   --hold reset state for 100 ns.
36  |   wait for 100 ns;
37  |   stim_Y <= "0000";
38
39  |   wait for 100 ns;
40  |   stim_Y <= "0001";
41
42  |   wait for 100 ns;
43  |   stim_Y <= "0010";
44
45  |   wait for 100 ns;
46  |   stim_Y <= "0100";
47
48  |   wait for 100 ns;
49  |   stim_Y <= "1000";
50
51  |   wait;
52  | end process;
53
54  | END behavior;

```

Listing 3.36 Test bench code of the 4-to-2 encoder

3.6.1.3 Implementation and Validation on FPGA Platform

To test the design of 4-to-2 encoder in the FPGA Board, we use the switches SW0, SW1, SW2, SW3 as inputs, and observing the result in the LEDR0 and LEDR1. Table 3.22 shows the pins assigned for 4-to-2 Encoder.

All possible input combinations have been tested to verify that the 4-to-2 encoder design is working properly. The observed results specify that the Leads display a correct binary code at the output according to the values of the 4 inputs indicated in the truth table of the 4-to-2 encoder: for example, if $Y = "0001"$, the LEDR (0) and LEDR(1) light up corresponding to the output values $A(0) = 1$ and $A(1) = 1$, as shown in the Fig. 3.48. At any time, only one of these 4 inputs can be ‘1’.

Table 3.22 Pins assigned of 4-to-2 encoder

Inputs: Y_0 , Y_1 , Y_2 and Y_3		Outputs: A_0 , A_1	
SW[0]	PIN_AA23	LEDR[1]	PIN_AK5
SW[1]	PIN_AB26		
SW[2]	PIN_AB25		
SW[3]	PIN_AE14		

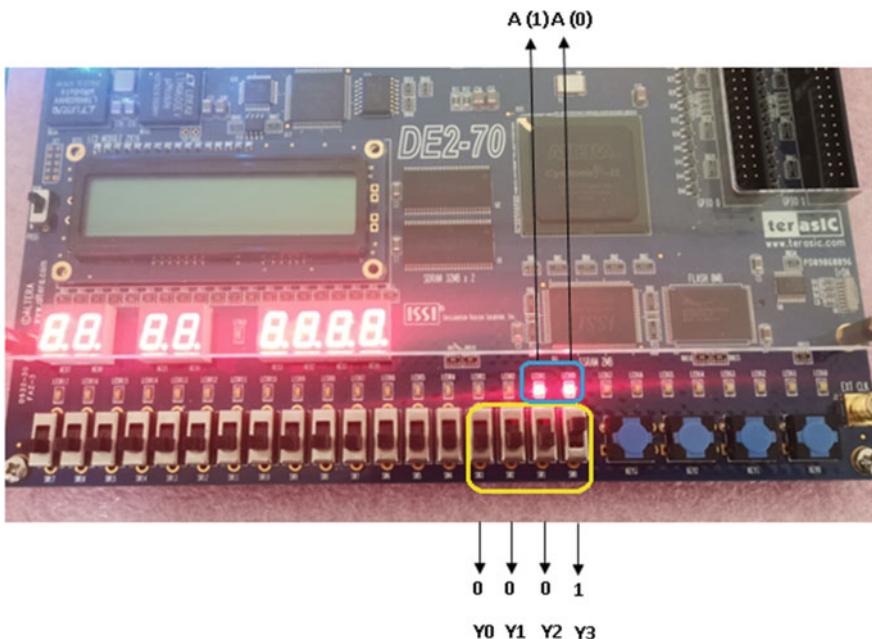


Fig. 3.48 Validation of the design of the 4-to-2 encoder on the FPGA board

3.6.2 Priority Encoder

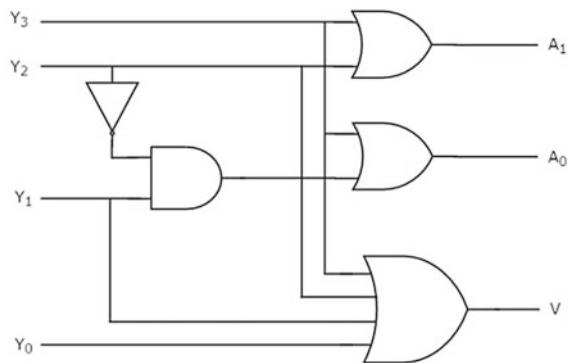
One of the main problems of the binary encoders is that if more than one of the input bits is high at the same time, then it produces an error at the output. While priority encoders were designed to solve this issue. It gives an output by considering only the highest priority bit. Because of this fact and characteristic, priority encoders are excellent for handling interrupt requests for a microprocessor.

The truth table of a 4-to-2 priority encoder is given in Table 3.23. In addition to the two outputs Y₁ and Y₀, the circuit has a third output designated by V; this is a valid bit indicator that is set to 1 when one or more inputs are equal to 1.

Table 3.23 Truth table of the 4-to-2 priority encoder

Inputs				Outputs		
A ₃	A ₂	A ₁	A ₀	Y ₁	Y ₀	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

Fig. 3.49 Logic circuit of the 4-to-2 priority encoder



From the above truth table, if all inputs are 0, there is no valid input and V is equal to 0. The X's in the table show the don't care condition, i.e., it may either be 0 or 1.

According to Table 3.22, the highest priority is assigned to the input whose subscript has the largest numerical value. Input A_3 has the highest priority, so, regardless of the values of the other inputs, when this input is 1, the output for $Y_1 Y_0$ is "11". A_2 has the next priority level. The output is "10" if $A_2 = 1$, provided that $A_3 = 0$, regardless of the values of the other two lower priority inputs. The output for A_1 is generated only if higher priority inputs are 0, and so on down the priority levels. The Boolean equations for 4-to-2 priority encoder, derived from the truth Table 3.22 are:

$$V = Y_0 + Y_1 + Y_2 + Y_3 \quad (3.9)$$

$$A_1 = Y_2 + Y_3 \quad (3.10)$$

$$A_0 = Y_1 \overline{Y_2} + Y_3 \quad (3.11)$$

When synthesized, the logic circuit for the 4-to-2 priority encoder is shown in Fig. 3.49.

3.6.2.1 VHDL Implementation

In VHDL, several methods can be used to describe the behavior of a priority encoder. The most obvious way is to use the encoder specific Boolean equations. However, these equations become more cumbersome and susceptible to typing errors when the encoder output has multiple bits. Faced with this situation, another alternative presents itself. It consists of using two solutions, one using *When/else*, and the other with *with/select/when*.

Solution N 1: When-else

The conditional assignment statement is an ideal alternative for use in a priority encoder circuit. A VHDL code using this format is shown below (see Listing 3.37).

```

1  LIBRARY ieee ;
2  USE ieee.std_logic_1164.all ;
3
4  ENTITY priority_encoder4_2 IS
5    PORT(
6      A : in std_logic_vector (3 downto 0);
7      Y : out std_logic_vector (1 downto 0);
8      V: out std_logic);
9    END priority_encoder4_2;
10
11 ARCHITECTURE Behavioral OF priority_encoder4_2 IS
12
13 BEGIN
14   --Conditional Signal Assignment
15
16   Y <= "11" WHEN A(3)= '1' ELSE
17     "10" WHEN A(2)= '1' ELSE
18     "01" WHEN A(1)= '1' ELSE
19     "00";
20   V <= '0' WHEN A = "0000" ELSE '1' ;
21
22 END Behavioral;
```

Listing 3.37 VHDL code of a 4-to-2 priority encoder using When-Else statement

In the conditional signal assignment, the highest-priority condition is examined first. If it is true ($A(3) = '1'$), the output is assigned according to that condition ($Y = "11"$) and no further conditions are evaluated. If the first condition is false, the condition of next priority is evaluated, and so on until the end. Thus, a low-priority input cannot alter the code resulting from an input of higher priority, as required by the priority encoding principle. If no clause is true, then the default value "00" is assigned to the output.

Solution N 2: if-then-else

The effect is similar to that of an IF statement, where a sequence of conditions is evaluated, but only one output assignment is made. However, an IF statement must be used within a PROCESS statement. The VHDL code for a priority encoder that uses the IF statement is shown below (see Listing 3.38).

```
1 LIBRARY ieee ;
2 USE ieee.std_logic_1164.all ;
3
4 ENTITY priority_encoder4_2 IS
5 PORT(
6   A : in std_logic_vector (3 downto 0);
7   Y : out std_logic_vector (1 downto 0);
8   V: out std_logic);
9 END priority_encoder4_2;
10
11 ARCHITECTURE Behavioral OF priority_encoder4_2 IS
12
13 BEGIN
14   --if-then-else statements
15
16   Process (A)
17
18   begin
19
20   If (A(3) = '1') then Y <= "11";
21   Elsif (A(2) = '1') then Y <= "10";
22   Elsif (A(1) = '1') then Y <= "01";
23   Elsif (A(0) = '1') then Y <= "00";
24   Else
25     V <= '0';
26     Y <= "00";
27   end if ;
28
29   end process ;
30
31 END Behavioral;
32
```

Listing 3.38 VHDL code for a 4-to-2 priority encoder using IF-Else Statement

3.6.2.2 Simulation

The VHDL test bench code for 4-to-2 priority encoder is shown in Listing 3.39. This test bench is created to verify the logic/operation of the entity and the architecture. It works on the simple principle where a series of inputs are given to the entity and the architecture, and the outputs can be displayed as a simple square wave.

```
1  LIBRARY ieee ;
2  USE ieee.std_logic_1164.all ;
3
4  entity tb_priority_encoder4_2 is
5  end tb_priority_encoder4_2;
6
7  architecture bench of tb_priority_encoder4_2 is
8
9    -- Component Declaration for the Unit Under Test (UUT).
10
11 component priority_encoder4_2
12 Port (
13   A : in STD_LOGIC_VECTOR (3 downto 0);
14   Y : out STD_LOGIC_VECTOR (1 downto 0));
15 end component;
16
17 --Inputs
18 signal stim_A: STD_LOGIC_VECTOR (3 downto 0):="0000";
19
20 -- Outputs
21 signal stim_Y: STD_LOGIC_VECTOR (1 downto 0):="00";
22 begin
23
24   --Instantiate the Unit Under Test (UUT).
25
26   uut: priority_encoder4_2 port map (
27     A => stim_A,
28     Y => stim_Y );
29
30   -- Stimulus process
31
```

```

32  stim_proc: process
33  begin
34      --test bench stimulus code
35      wait for 20ns;
36          stim_A <= "0000";
37      wait for 20ns;
38          stim_A <= "0001";
39      wait for 20ns;
40          stim_A <= "0010";
41      wait for 20ns;
42          stim_A <= "0011";
43      wait for 20ns;
44          stim_A <= "0100";
45      wait for 20ns;
46          stim_A <= "0101";
47      wait for 20ns;
48          stim_A <= "0110";
49      wait for 20ns;
50          stim_A <= "0111";
51      wait for 20ns;
52          stim_A <= "1000";
53      wait for 20ns;
54          stim_A <= "1001";
55      wait for 20ns;
56          stim_A <= "1010";
57      wait for 20ns;
58          stim_A <= "1011";
59      wait for 20ns;
60          stim_A <= "1100";
61      wait for 20ns;
62          stim_A <= "1101";
63      wait for 20ns;
64          stim_A <= "1110";
65      wait for 20ns;
66          stim_A <= "1111";
67      wait for 20ns;
68          stim_A <= "0000";
69      wait;
70
71      wait;
72  end process;
73  end bench;

```

Listing 3.39 VHDL test bench code for a 4-to-2 priority encoder

Figure 3.50 shows the simulation of the 4-to-2 priority encoder. The encoder is working properly as required by the priority encoding principle. The highest priority is assigned to the entry whose index has the highest numerical value. Initially, the default value “00” is assigned to the output for an input of “0000”.

The following input “0001” could not modify the result of this output because it was considered a low priority entry. When $A(1) = 1$, the output goes to 11 and remains held at this value until the arrival of a higher level of $A(2) = 1$ to go to “10”.

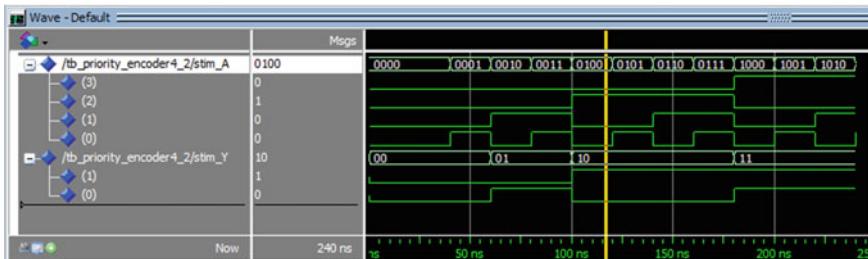


Fig. 3.50 Simulation waveforms of the 4-to-2 priority encoder

When the input has more than one level is activated, the output will not be changed: This is the case with inputs “0101”, “0110”, “0111”. Finally, the output switches to 11 at the instant 180 ns when A(3) = 1 (input = “1000”).

3.7 Summary

In this chapter, the design of various combinatorial circuits such as arithmetic circuits, Multiplexer and Demultiplexer circuits, Encoder and Decoder circuits has been described and implemented in VHDL language. To do this, we started with the design of the arithmetic circuits. Several types of adder such as 1-bit half adder, 1-bit full adder, 4-bit adder, and 4-bits adder-subtractor have been described and implemented in VHDL with different architectures like structural, data flow and behavioral descriptions. Then, we have illustrated the design process for multiplexers (2-to-1, 4-to-1 and 8-to-1) and Demultiplexers (1-to-4 and 1-to-8) using the flow of data, behavioral and structural modeling architectures. The design of the encoder (Binary Encoder, Priority Encoder) and decoder (2-to-4, 3-to-8, BCD-to-Decimal) have been presented at the end of this chapter. The operation of these circuits is verified on the one hand by simulations carried out using ModelSim tool and on the other hand by their implementation into an FPGA platform.

Chapter 4

Sequential Logic Circuits



Abstract The objective of this chapter is to describe in VHDL the design of sequential logic circuits such as latches, flip-flops, shift registers and various counters. The operation of each circuit is verified through the simulation results obtained by using the ModelSim tool. This chapter also cover the implementation and validation of the operation of sequential logic circuits in the FPGA platform.

4.1 Introduction

Unlike combinatorial circuits whose output is a function only of the current input, sequential circuits are logic circuits whose output depends not only on the current value of its inputs, but also on the values of past inputs. This type of circuits which exhibit this behavior allow to create more sophisticated and intelligent systems. In other words, combinational circuits can be built based on Boolean equations only, while sequential circuits must employ also storage resources as latches and flip-flops.

Sequential circuits are broadly classified into two main categories, called as synchronous and asynchronous. In the synchronous design, all registers in this circuit are clocked by the same clock signal. The sequential logic is designed to occur either on the rising edge of the clock signal when flip-flops are used or at a particular logic level when latches are used. In sequential logic, the clock signal is not used to control the timing of the circuit. This form of design allows for very fast operation of the sequential logic, but its operation is subject to timing issues where uneven delays in the logic gates can cause the circuit to operate improperly.

The objective of this chapter is to describe in VHDL the design of sequential circuits such registers and counters through the process statement. The operation of these circuits is verified on the one hand by simulations carried out using ModelSim tool and on the other hand by their implementation in the FPGA platform.

We start by examining the design of latches and flip-flops in VHDL that are used as logic elements to design the sequential logic circuits. This is followed by a study of registers which are fundamental component of microprocessors. There are four types of registers called shift registers like Serial-In Parallel-Out (SIPO),

Serial-In Serial-Out (SISO), Parallel-In Parallel- Out (PIPO) and Parallel-In Serial-Out (PISO). Finally, we look at the counters which are the most important logic circuits in digital systems. Counting circuits can be designed using D flip-flops and they are generally classified into two categories synchronous and asynchronous.

4.2 D Flip-Flop

There are two types of digital circuits, D-latch and D-flip-flop, which are used to store data. The main difference between these circuits is the conditions under which the data is stored. In a D latch, which is transparent, the Q output takes the value of the D input only when a validation input (EN) is in the high state. Unlike latch, the Q output of the flip-flop also follows D, but only when there is a transition on an enable input called the clock (CLK). Therefore, the D flip-flop is said to be controlled by “edge”, while the D latch is controlled by “level”. Figure 4.1 shows the D-type latch and D-type flip-flop.

There are many different ways to implement latches and flip-flops in VHDL. For a behavioral description of a D latch, an “*If–then statement*” placed within a PROCESS on line 12 of the Listing 4.1, which is written to indicate the “Q follows D” property of the latch according to its truth table, i.e., when EN is ‘1’, the input data provided in D is stored. Otherwise, when EN is ‘0’, the circuit holds the value provided in D for the last time.

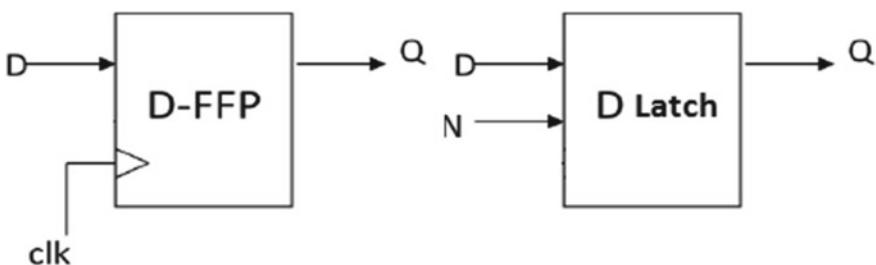


Fig. 4.1 D flip-flop and D-latch

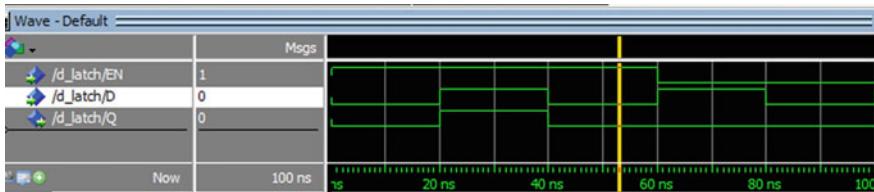


Fig. 4.2 Simulation waveform of D-latch

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity D_latch is
5  port ( D, EN : in std_logic;
6        Q: out std_logic);
7  end D_latch;
8
9  architecture behavior of D_latch is
10 begin
11
12 process (EN, D)
13 begin
14 IF (EN = '1') then
15   Q <= D;
16 end if;
17 end process;
18
19 end behavior;
```

Listing 4.1 VHDL implementation for a D latch

Figure 4.2 shows the waveform of the D latch behavior according to EN and D inputs. Input data (D) passes to output (Q) as long as the EN input is held at level “1”. This behavior is called “transparent”. If EN is not high, the output retains its previous value.

The advantage of the D latch is its low cost on circuitry, while the main drawback is that it has no precise control of the input D to output Q. For instance, if a circuit has several D latches in cascade and if EN = ‘1’ for a long period of time, the data at the D input might be propagated through many latches. The D flip-flop is designed to address the problem of imprecise control from input D to output Q presented by D-latch. A D flip-flop can be implemented in a similar way using behavioral modeling, as shown in Listing 4.2.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity D_Flip_Flop is
5  port ( D, clk : in std_logic;
6        Q: out std_logic);
7  end D_Flip_Flop;
8
9  architecture behavior of D_Flip_Flop is
10 begin
11
12 process (clk)
13 begin
14 IF (clk'event and clk = '1') then
15   Q <= D;
16 end if;
17 end process;
18
19 end behavior;

```

Listing 4.2 VHDL code of a D flip-flop using If-then statement

The construct “*clk’event*” (pronounced “clock tick event”) indicates that a change has just happened on the clock. This, combined with the test for a logic HIGH on the clock, indicates that a positive clock edge has just occurred. The PROCESS statement tests only for a change in CLK, since no change of Q is possible without a positive clock edge. Input D is not tested because it cannot make Q change on its own.

The second method shown below (Listing 4.3) uses the *wait* statement to describe the D flip-flop. The evaluation is suspended by a *wait-until statement* (over time) until the expression evaluates to true.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity D_Flip_Flop is
5  port ( D, clk : in std_logic;
6        Q: out std_logic);
7  end D_Flip_Flop;
8
9  architecture behavior of D_Flip_Flop is
10 begin
11
12 process (clk)
13 begin
14 wait until (clk'event and clk = '1') then
15   Q <= D;
16 end process;
17
18 end behavior;

```

Listing 4.3 VHDL code of a D flip-flop using wait statement

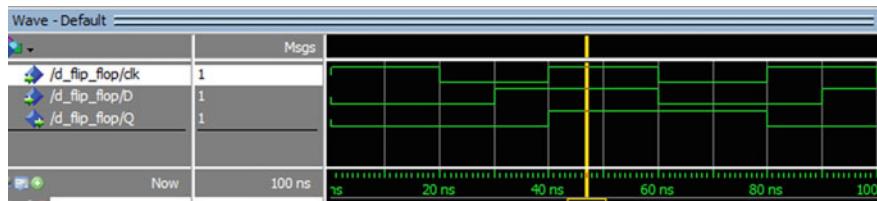


Fig. 4.3 Simulation waveform of D flip-flop

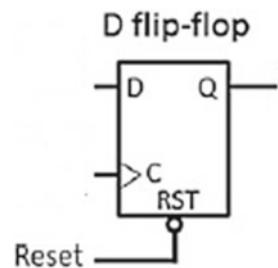
The simulation waveform of the D flip-flop is shown in Fig. 4.3. As we can see, at each positive clock edge of CLK (positive edge triggered), the Q output is assigned the value of the D input. When the CLK input is steady at a logic 0 or a 1, the Q output holds its current value even when the D input changes.

4.2.1 D Flip-Flop with Synchronous Reset

There are several variations for the D flip-flop implementation. In Fig. 4.4, a reset control signal has been added to the circuit. This reset can either be asynchronous or synchronous. The D flip-flop is known as D flip-flop with synchronous reset if the changes in the output Q occur in synchronization with the clock (on the active edge of the clock).

The following VHDL code (see Listing 4.4) describes the behavior of D flip-flop with asynchronous reset.

Fig. 4.4 D flip-flop with synchronous reset



```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity D_Flip_Flop is
5  port ( D, clk, reset : in std_logic;
6        Q: out std_logic);
7  end D_Flip_Flop;
8
9  architecture behavior of D_Flip_Flop is
10 begin
11
12 process (clk)
13 begin
14 if rising_edge(clk) then
15   if (reset = '1') then
16     Q <= '0';
17   else
18     Q <= D;
19   end if;
20 end if;
21 end process;
22
23 end behavior;

```

Listing 4.4 VHDL code of a D flip-flop with synchronous reset

In the “*Ifloop*”, the synchronous reset implementation, line 14, Listing 4.4, should be placed after “*rising_edge(clk)*”. The process is activated only on a rising clock edge, at which time either a reset or the normal operation is executed. Another way to code a synchronous reset is shown below, Listing 4.5. At the clock edge, normal operation is performed. If the reset is active, the result of normal operation is canceled by the reset action.

```

12 process (clk) begin
13   if rising_edge(clk) begin
14     Q <= D; -- normal operation
15   if (reset = '1') then
16     Q <= '0'; -- do reset
17   end if;
18 end if;
19 end process;
20

```

Listing 4.5 An alternative way of coding a D flip-flop with synchronous reset

The waveform in Fig. 4.5 shows the simulation results for the D flip-flop with synchronous reset.

At the start of the 2th clock cycle, the reset signal (RST) = “0”, normal operation takes place and the Q output copies the D input which is at the low level. Next, at the start of the 3th clock cycle, the reset signal (RST) remains low and the Q output follows the D input which goes high. During the rising edge of the 4th clock cycle, input D remains high and the reset signal (RST) goes to level 1. In this case, normal operation is canceled by the reset action and the output Q is equal to zero.



Fig. 4.5 Waveform simulation results for the D flip-flop with synchronous reset

4.2.2 *D Flip-Flop with Asynchronous Reset*

The D Flip-Flop with asynchronous reset has three inputs:

- data input (D, value to store);
- clock input (clk), and asynchronous reset input (rst, active high);
- and one output: data output (Q).

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity D_Flip_Flop is
5  port ( D, clk, reset : in std_logic;
6        Q: out std_logic);
7  end D_Flip_Flop;
8
9  architecture behavior of D_Flip_Flop is
10 begin
11
12  process (clk, reset)
13  begin
14    if (reset = '1') then
15      Q <= '0';
16    elsif rising_edge(clk) then
17      Q <= D;
18    end if;
19  end process;
20
21 end behavior;
```

Listing 4.6 VHDL code for a D flip-flop with asynchronous reset

To describe the behavior of the D flip-flop with asynchronous reset, we use a “Process (Clk, reset)” block sensitive to the clock signal CLK and to the reset signal. The output Q of D flip-flop changes when a rising edge of clock occurs or the reset is asserted. In the “If loop”, asynchronous items (line 14, Listing 4.6) are before the rising_edge (Clk) statement. The behavior of this flip-flop is described as follows: If reset signal (Reset = ‘1’) is asserted, Q is driven to logic ‘0’, else Q is driven by D (the input data is stored $Q \leq D$) at rising edge of clk. As can be seen in Fig. 4.6,

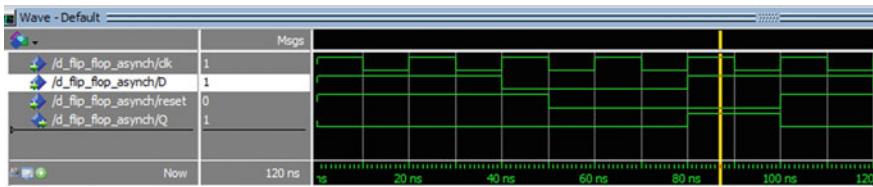


Fig. 4.6 Waveform simulation results for the D flip-flop with asynchronous reset

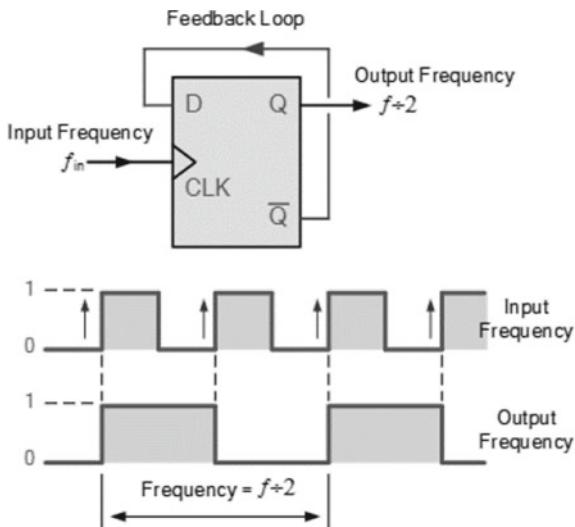
during the first 3 clock cycles, the reset signal is held at “1” and the Q output is driven to logic “0” and remains in this state. At the 4th clock cycle and at the rising edge, the reset signal = ‘0’ which will lead to the input D = ‘0’ being stored in the Q output. At the 5th clock cycle, the reset signal always remains at level ‘0’, but input D goes to level 1 and the output Q copies this state. Finally, at the 6th clock cycle, the input D remains at level 1 but the reset signal is activated at 1 and forces the output Q to go to level ‘0’.

4.2.3 Application of D Flip-Flop to Build a Clock Divider

Usually the clock signal comes from a crystal oscillator on-board. Two oscillators are embedded in the FPGA board, Altera DE2-70, which produce clock signals of 28.86 and 50 MHz. However, some devices like LEDs do not need such a high frequency to blink. The objective of this part is to validate experimentally on the FPGA platform, the design of the clock divider, created from several D flip-flops. This frequency divider will flash the LEDs on the FPGA card. A flip-flop with its inverted output fed back to its input serves as a divide-by-2 circuit. This circuit is shown in Fig. 4.7.

The following Listing 4.7 shows the VHDL code for a frequency divider by 2.

Fig. 4.7 Frequency division by 2



```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity frequency_divider_2 is
5  port (
6    reset : in std_logic;
7    clk : in std_logic;
8    Q : out std_logic
9    );
10 end frequency_divider_2;
11
12 architecture behavior of frequency_divider_2 is
13
14  signal temp : std_logic;
15
16 begin
17 process (clk,reset)
18 begin
19 if reset = '1' then
20   temp <= '0';
21 elsif clk'event and clk = '1' then
22   temp <= not temp;
23 end if;
24 end process;
25
26 Q <= temp;
27
28 end architecture;

```

Listing 4.7 VHDL code of the frequency divider by 2

It can be seen from the waveforms in Fig. 4.8 that the period of the output pulses Q doubles the period of the input clock CLK (i.e. the output frequency Q is half frequency of CLK).

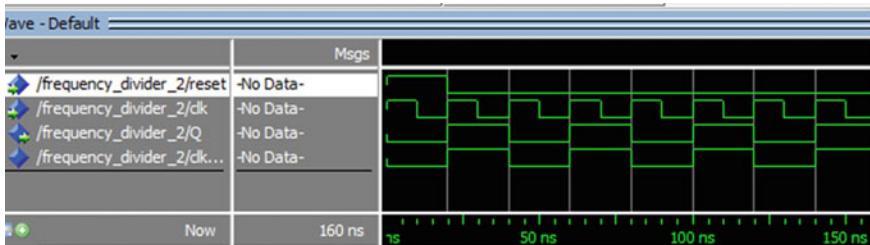


Fig. 4.8 Simulation waveform of frequency divider by 2

With this circuit, we can actually divide the clock by cascading the previous circuit, as displayed in Fig. 4.9.

Each stage divided the frequency by 2. Suppose that the input clock frequency to the first stage is 50 MHz (50,000,000 Hz). After n stage, CLK must be divided by a power-of-2 in order to generate the slow clock, CLK_div of 1 Hz. The frequency became: $\text{CLK_div} = \text{CLK}/2^n$. The VHDL program for a frequency divider of 1 Hz as well as the simulation results are shown in Listing 4.8 and in Fig. 4.10, respectively.



Fig. 4.9 Block diagram of the clock divider of 1 Hz

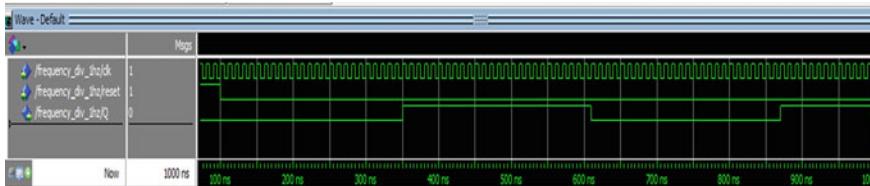


Fig. 4.10 Simulation waveform of frequency divider of 1 Hz

```

1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_arith.all;
4   use ieee.std_logic_unsigned.all;
5
6   entity frequency_div_1hz is
7   | generic (N: integer:= 25999999);
8   port (
9   | clk, reset : in std_logic;
10  | Q : out std_logic );
11 end frequency_div_1hz;
12
13 architecture behavior of frequency_div_1hz is
14 signal temp : std_logic;
15
16 begin
17 process (clk,reset, temp)
18 variable s: integer range 0 to N;
19 begin
20 if reset = '1' then
21   s := 0;
22   temp <= '0';
23 elsif clk'event and clk = '1' then
24   if s = N then
25     s := 0;
26     temp <= not temp;
27   else
28     s := s + 1;
29   end if;
30 end if;
31 end process;
32 Q <= temp;
33 end behavior;

```

Listing 4.8 VHDL code for a frequency divider of 1 MHz

4.2.4 Implementation and Validation on FPGA Platform

The frequency divider has two inputs Clk (clock signal), reset (control signal) and one output Q, representing the slow clock signal of 1 Hz. The clock signal is supplied by the DE2-70 board; it is 50 MHz. To test the design of the 1 Hz frequency divider in the FPGA board, we assign the 50 MHz clock signal to the PAD_15 pin, the reset to the switch SW0 and the Q output to assigned to LED green [7]. Table 4.1 shows the pins assigned for frequency divider of 1 Hz.

Table 4.1 Pins assigned of frequency divider of 1 Hz

Inputs: CLK and reset	Outputs: Q		
CLK_50	PIN_AD15	Q = LedG[7]	PIN_AA24
Reset = SW[0]	PIN_AA23		

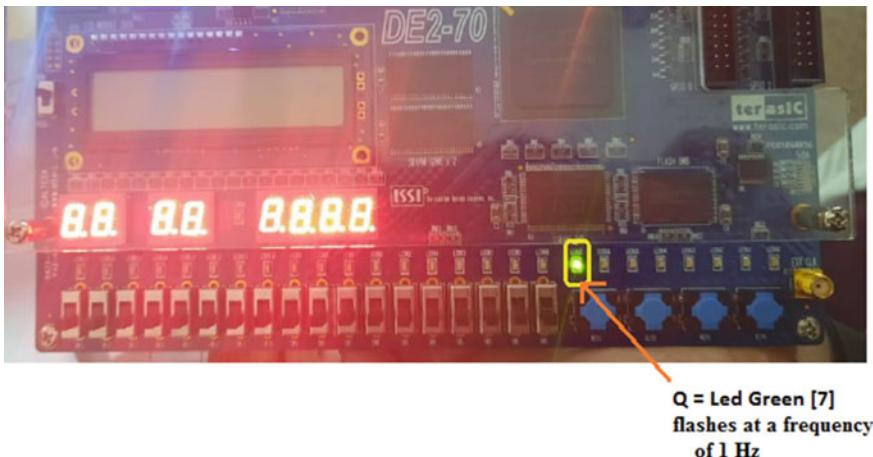


Fig. 4.11 Validation of the design of the frequency divider of 1 Hz on the FPGA board

As shown in Fig. 4.11, the green Led [7] of the FPGA platform flashes at a frequency of 1 Hz, demonstrating the correct operation of the 1 Hz frequency divider.

4.3 Registers

A flip-flop can only store a one bit of information 0 or 1. When it comes to storing multi-bits of information, a register made up of several flip-flops is used for this purpose. These flip-flops share the same clock and control signals for joint operation. Registers are fundamental component of microprocessors. Remember that a microprocessor does not process digital signals limited to a single bit, but words at 4, 8, 16 or 64 bits, in parallel mode, means that all the bits are processed at the same times.

There are other kinds of registers called shift registers. A shift register is a register in which binary data can be stored and then shifted left or right when the control signal is asserted. Shift registers can be classified into four distinct groups.

- Serial-in parallel-out (SIPO), in which the register is loaded serially, one bit at a time, and when an output is required the data stored in the register can be read in parallel form.
- Serial-in serial-out (SISO), in which data can be moved serially in and out of the register, one bit at a time.
- Parallel-in parallel-out (PIPO), in which all the flip-flops in the register are loaded simultaneously, and when an output is required the flip-flops are read simultaneously.

- Parallel-in serial-out (PISO), in which all the flip-flops are loaded simultaneously and when an output is required, the data stored is removed serially from the register one bit at a time under clock control.

4.3.1 Serial-In Parallel-Out (SIPO) Shift Register

Figure 4.12 shows a 4-bit Serial In-Parallel-Out (SIPO) shift register, sensitive to positive edge of the clock pulse. In this kind of shift register, the content of all flip-flops can be read in parallel (hence “parallel-out”) at the output pins of the flip-flops (Q_1 to Q_4). However, the data bits are loaded into the shift register in a serial fashion using the Dinput. The contents of each flip-flop are transferred to the next flip-flop at each positive edge of the clock. Thus, it requires four clock cycles to input (shift in) a 4-bit data into such a register.

4.3.1.1 VHDL Implementation and Simulation

Several approaches can be used to program shift registers in VHDL, such as structural, data flow, and behavioral descriptions. We can use the behavioral description to implement a four-bit SIPO shift register, as shown in Listing 4.9. It is very similar to the description of an ordinary register. A PROCESS (lines 15–25), with the “If

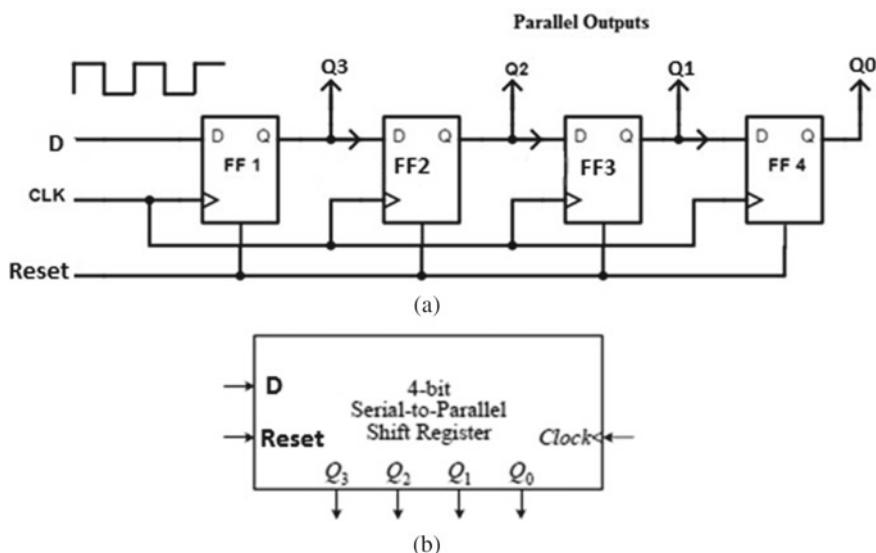


Fig. 4.12 4-bit serial-in parallel-output (SIPO) shift register: **a** circuit; **b** logic symbol

instruction" playing the central role, is used to build this circuit. The process is sensitive to the CLK and reset signals (line 15). The statements below:

21	temp(3 downto 1) <= temp(2 downto 0);
22	temp(0) <= D;

Indicates that when the shift register is activated, the whole data vector "temp" shifts one position to the right at every positive clock transition, with the rightmost value discarded and the leftmost position taken by serial input D.

The line 21 is equivalent to the three separate assignment statements, as follows:

21	temp(1) <= temp(0);
22	temp(2) <= temp(1);
23	temp(3) <= temp(2);

1	library ieee;
2	use ieee.std_logic_1164.all;
3	
4	entity Register_SIPO is
5	port(
6	clk_1hz, reset : in std_logic;
7	D: in std_logic;
8	Q: out std_logic_vector(3 downto 0));
9	end Register_SIPO;
10	
11	architecture beh of Register_SIPO is
12	signal temp: std_logic_vector(3 downto 0);
13	begin
14	
15	process (clk_1hz)
16	begin
17	if (clk_1hz'event and clk_1hz='1') then
18	if reset = '1' then
19	temp <= (others => '0');
20	else
21	temp(3 downto 1) <= temp(2 downto 0);
22	temp(0) <= D;
23	end if;
24	end if;
25	end process;
26	Q <= temp;
27	end beh;

Listing. 4.9 VHDL code of 4-bit SIPO shift register

Simulation results for 4-bit SIPO shift register are shown in Fig. 4.13. As can be seen, the content of each flip-flop moves one position to the right at every rising edge of the clock. Thus, it requires four clock cycles to input (D) a 4-bit data into such a register.

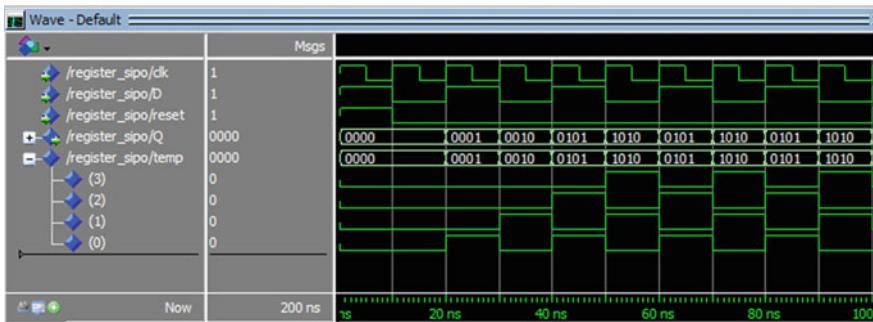


Fig. 4.13 Waveform simulation results for the 4-bit SIPO Shift register

4.3.1.2 Implementation and Validation on FPGA Platform

To test the design of the 4-bit SIPO shift register on the FPGA platform, a frequency divider module (see Fig. 4.14) is needed to produce a slowed-down clock signal of 1 Hz frequency from clock signal of 50 Hz provided by the FPGA board.

Listing 4.5 shows the VHDL code of the frequency divider module of 1 Hz.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_arith.all;
4  use ieee.std_logic_unsigned.all;
5
6  entity frequency_div_1hz is
7  | generic (N: integer:= 25999999);
8  port (
9  | clk_50MHz, reset : in std_logic;
10 | clk_1Hz : out std_logic );
11 end frequency_div_1hz;
12
13 architecture behavior of frequency_div_1hz is
14 signal temp : std_logic;
15
16 begin
17 process (clk_50MHz,reset, temp)
18 variable s: integer range 0 to N;
19 begin
20 if reset = '1' then
21 | s := 0;

```

Listing 4.10 VHDL code for the frequency divider module of 1 Hz

Fig. 4.14 Frequency divider module of 1 Hz

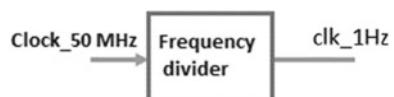
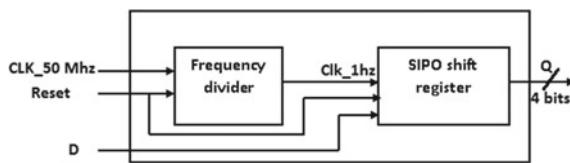


Fig. 4.15 Association of the frequency divider 1 Hz with the 4-bit SIPO shift register



```

22      temp <= '0';
23  elsif clk_50MHz'event and clk_50MHz = '1' then
24    if s = N then
25      s := 0;
26      temp <= not temp;
27    else
28      s := s + 1;
29    end if;
30  end if;
31  end process;
32  Clk_1Hz <= temp;
33  end behavior;
  
```

Listing. 4.10 VHDL code for the frequency divider module of 1 Hz

Figure 4.15 represents the association of the frequency divider 1 Hz with the 4-bit SIPO shift register.

With the structural method, we develop the VHDL program of the SIPO shift register (Listing 4.11) in order to be able to shift the contents of each flip-flop to the right at each rising edge of the clock.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Register_SIPO_struct is
5  port(
6    D, clk, reset : in std_logic;
7    Q: out std_logic_vector(3 downto 0) );
8  end Register_SIPO_struct;
9
10 architecture struct of Register_SIPO_struct is
11
12 component frequency_div_1hz
13 port ( clk_50Mhz, reset: in std_logic;
14   clk_1hz : out std_logic);
15 end component;
16
17 component Register_SIPO
18 port ( clk_1hz, reset, D: in std_logic;
19   Q : out std_logic_vector(3 downto 0));
20 end component;
21
22 Signal S: std_logic;
23
24 begin
25 Freq_div: frequency_div_1hz port map( clk, reset, S);
26 SIPO_register: Register_SIPO port map(S, reset,D, Q);
27
28 end struct;

```

Listing 4.11 VHDL code of the association of the frequency divider 1 Hz with the 4-bit SIPO shift register

To test the design of the SIPO shift register in the FPGA board, we assign the D input and the reset signal in the DE2 board to the switches SW [0] and SW [1] and observing the result in the Red LED [0], Red LED [1], Red LED [2] and Red LED [3]. Table 4.2 shows the pins assigned to the 4-bit SIPO shift register.

Figure 4.16 shows the validation of the design of the SIPO shift register on the FPGA platform: the input data D goes from Q(0) to Q(3) via Q(1) and Q(2) at each rising edge of the clock.

Table 4.2 Pins assigned to the 4-bit SIPO shift register

Inputs: CLK and reset	Outputs: Q(3), Q(2), Q(1), Q(0)		
CLK_50	PIN_AD15	LedR[0]	PIN_AJ6
Reset = SW[0]	PIN_AA23	LedR[1]	PIN_AK5
		LedR[2]	PIN_AJ5
		LedR[3]	PIN_AJ4

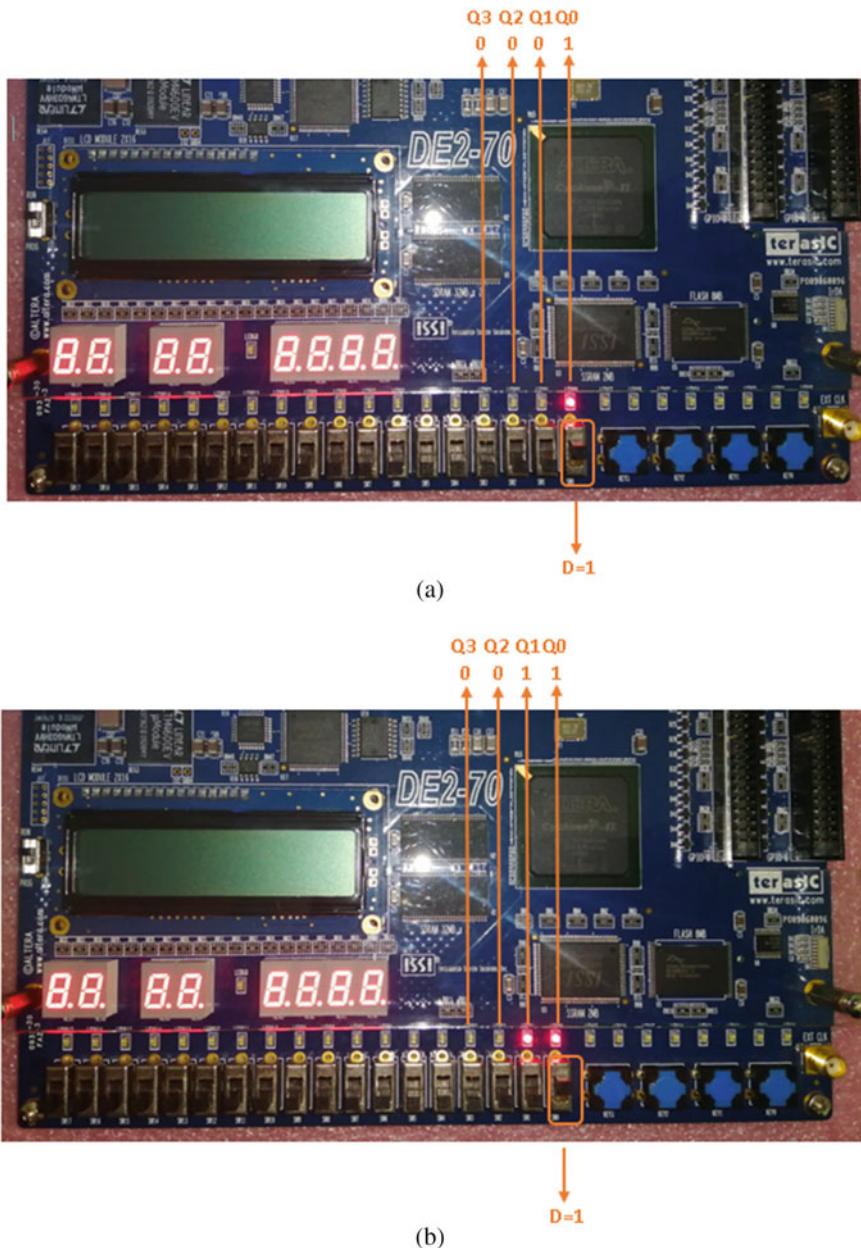


Fig. 4.16 Validation of the design the SIPO shift register on the FPGA platform: the input data D passes from Q(0) to Q(3) via Q(1) and Q(2) at each rising edge of the clock. **a** At the first clock cycle, Q(0) of the SIPO shift register receives data D; **b** At the second clock cycle, the content of Q(0) moves to Q(1); **c** At the third clock cycle, the content of Q(1) moves to Q(2); **d** At the fourth clock cycle, the content of Q(2) moves to Q(3)

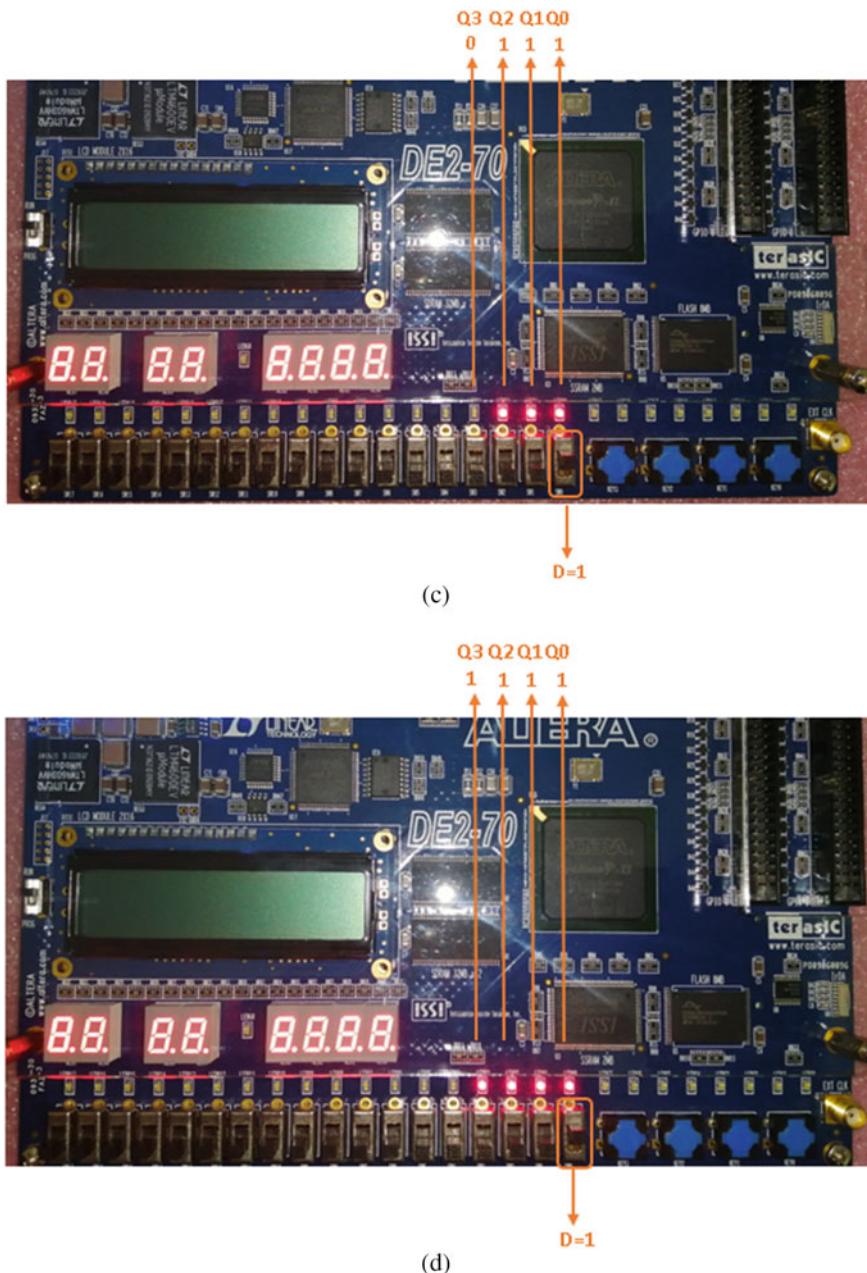


Fig. 4.16 (continued)

4.3.2 Serial-In Serial-Out (SISO) Shift Register

In this type of register, data D is serial shifted, i.e., one bit at a time under clock control. This data appears at the serial output Q, after 4 clock cycles and is lost one clock cycle later. Figure 4.17 shows the structure of a 4-bit serial-in, serial-out right-shift (SISO) register. It consists of four D flip-flops which are connected in a serial manner. All these flip-flops are synchronous with each other since the same clock signal is applied to each flip flop. The SISO shift register has only three connections, the serial input (D) which determines what enters the left hand flip-flop, the serial output (Q) which is taken from the output of the right hand flip-flop and the sequencing clock signal (Clock).

4.3.2.1 VHDL Implementation and Simulation

A signal assignment statement is used to implement the Boolean equations for the SISO shift register as shown in Listing 4.12. It is written as a single statement for efficiency:

```
21 | temp <= D & temp(3 downto 1);
```

but it could also be written as four separate assignment statements, as follows:

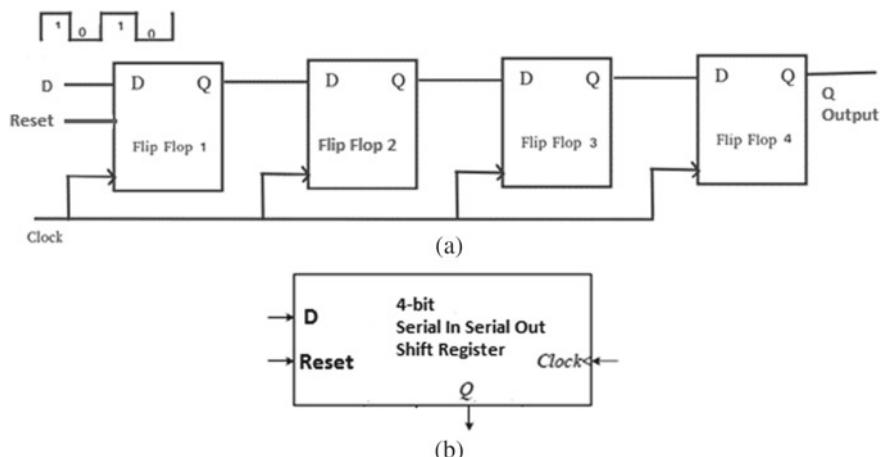


Fig. 4.17 Serial-in serial-output (SISO) shift register: **a** circuit; **b** logic symbol

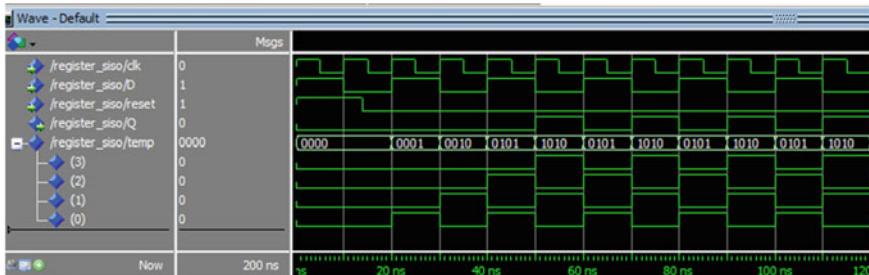


Fig. 4.18 Waveform simulation results for the SISO shift register

```

21      temp(0) <= D;
22      temp(1) <= temp(0);
23      temp(2) <= temp(1);
24      temp(3) <= temp(2);

```

For a D flip-flop, Q follows D. When a clock pulse is applied to the circuit, the contents of the flip-flops move one position to the right and the bit at the circuit input is shifted into Q1. The bit stored in Q3 is overwritten by the former value of Q2 and is lost. Since the data move from left to right, we say that the shift register implements a right shift function.

Figure 4.18 shows the simulation waveforms of the SISO shift register. It can be seen that the data D is shifted in series at the rate of one bit at each rising edge of the clock.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Register_SISO is
5  port(
6    clk, D, reset : in std_logic;
7    Q : out std_logic);
8  end Register_SISO;
9
10 architecture behavior of Register_SISO is
11
12  signal temp: std_logic_vector(3 downto 0);
13
14 begin
15  process(clk,reset)
16  begin
17    if rising_edge(clk) then
18      if(reset='1') then
19        temp <= "0000";
20      else
21        temp <= D & temp(3 downto 1) ;
22      end if;
23    end if;
24  end process;
25  Q <= temp(0);
26
27 end behavior;

```

Listing 4.12 VHDL code of SISO shift register

We can also describe the design of the SISO shift register using for loop iterative statement as indicated in Listing 4.13.

```

15  process(clk,reset)
16
17  begin
18  if(reset='1') then
19    temp<= "0000";
20  elsif rising_edge(clk) then
21    for i in 0 to 2 loop
22      temp(i+1) <= temp(i); end loop;
23      temp(0) <= D;
24    end if;
25
26  end process;
27  Q <= temp(3);

```

Listing 4.13 Design of the SISO shift register using for loop iterative statement

4.3.2.2 Implementation and Validation on FPGA Platform

To test the design of the SISO shift register in the FPGA board, the pins assigned to the 4-bit SIPO shift register are shown in Table 4.3. Figure 4.19 shows the validation of the design of the SISO shift register on the FPGA platform: the data D is shifted in series at the rate of one bit at each rising edge of the clock.

4.3.3 Parallel-In Parallel-Out (PIPO) Shift Register

The following circuit (see Fig. 4.20) is a four-bit parallel in parallel out shift register constructed by four D flip-flops. In this type of register, data is given as input separately for each flip flop. There are no interconnections between these individual flip-flops because no serial data shift is required. Similarly, the output is also collected individually from each flip-flop. Once the register is clocked, all the data at the D inputs appear at the corresponding Q outputs simultaneously.

Table 4.3 Pins assigned to the SISO shift register

Input: CLK and reset	Output: Q		
CLK_50	PIN_AD15	LedR[0]	PIN_AJ6
Reset = SW[0]	PIN_AA23		

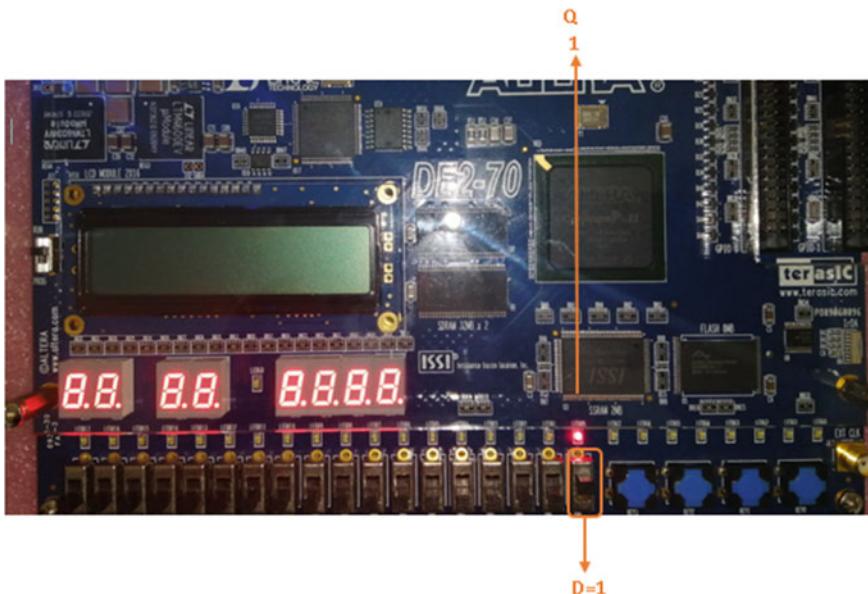


Fig. 4.19 Validation of the design the SISO shift register on the FPGA platform

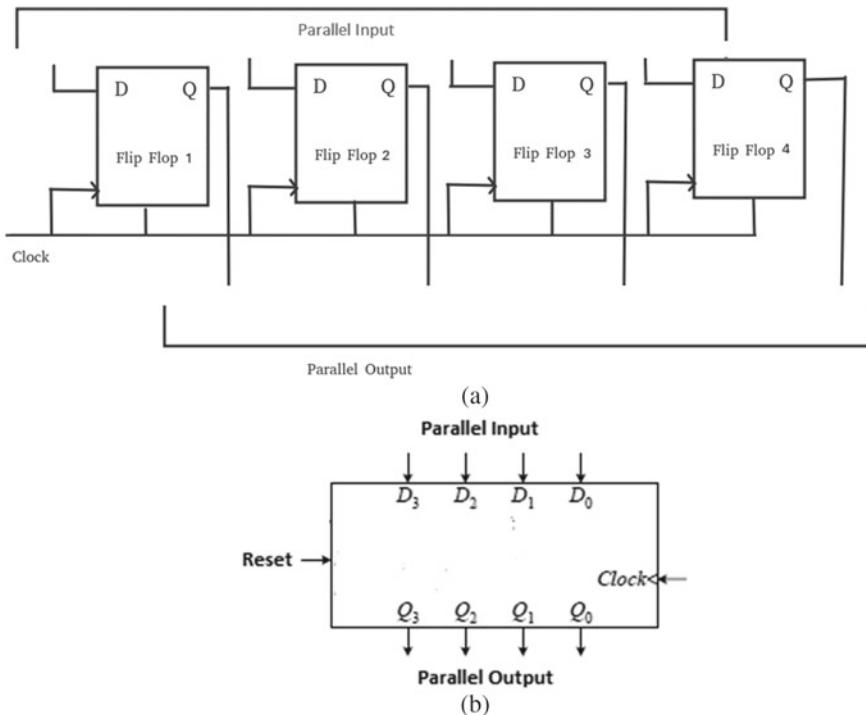


Fig. 4.20 4-bit parallel-in parallel-output (PIPO) shift register: **a** circuit; **b** logic symbol

4.3.3.1 VHDL Implementation and Simulation

A VHDL description of 4-bit Parallel-In Parallel-Output (PIPO) shift register is shown in Listing 4.14. By using the “*If statement*”, we can easily implement the operation of the 4-bit PIPO shift register in VHDL. The “*If statement*” is inside a process (lines 17–26), sensitive to reset signal, data D and the clock signal CLK. When the shift register is activated, all the data of the D inputs appear simultaneously on the corresponding Q outputs.

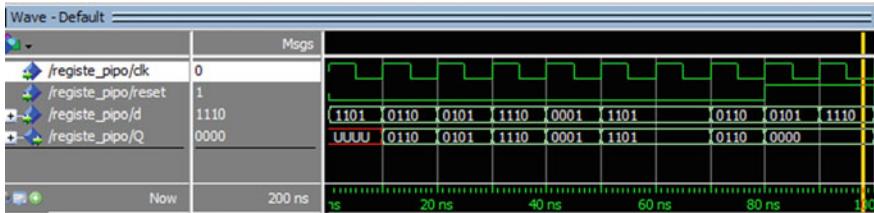


Fig. 4.21 Waveform simulation results for the 4-bit PIPO shift register

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all;
3
4  entity Registe_PIPO  is
5    port(
6      clk, reset : in STD_LOGIC;
7      d : in STD_LOGIC_VECTOR(3 downto 0);
8      Q : out STD_LOGIC_VECTOR(3 downto 0)
9      );
10 end Registe_PIPO;
11
12 architecture behavior of Registe_PIPO is
13 begin
14
15   process (clk,d,reset) is
16   begin
17     if (reset='1') then
18       Q <= "0000";
19     elsif (rising_edge (clk)) then
20       Q <= D;
21     end if;
22   end process;
23
24 end behavior;

```

Listing 4.14 VHDL code of the 4-bit PIPO shift register

Figure 4.21 illustrates the simulation results of the 4-bit PIPO shift register. For the first 6 clocks, the reset signal = 0, the data arriving in parallel on the inputs of the 4 flip-flops, they were collected simultaneously and in parallel format. When the reset signal goes high, the process cancels the output Q = “0000” even if the data arrives in parallel because the priority is given to the synchronous reset of the PIPO register.

4.3.3.2 Implementation and Validation on FPGA Platform

To test the design of the PIPO shift register in the FPGA board, the pins assigned to the 4-bit SIPO shift register are shown in Table 4.4.

Figure 4.22 shows the validation of the design of the PIPO shift register on the FPGA platform: all data at the D inputs appear simultaneously at the corresponding Q outputs once the register is clocked.

Table 4.4 Pins assigned to the PIPO shift register

Inputs: CLK and reset	Outputs: Q(3), Q(2), Q(1), Q(0)		
CLK_50	PIN_AD15	LedR[0]	PIN_AJ6
Reset = SW[4]	PIN_AC26	LedR[1]	PIN_AK5
D[0] = SW[0]	PIN_AA23	LedR[2]	PIN_AJ5
D[1] = SW[1]	PIN_AB26	LedR[3]	PIN_AJ4
D[1] = SW[2]	PIN_AB25		
D[3] = SW[3]	PIN_AC27		

4.3.4 Parallel-In Serial-Out (PISO) Shift Register

The block diagram of 4-bit parallel-in-serial-out shift register (PISO) shift register is shown in the following Fig. 4.23. It consists of four D flip-flops, which are cascaded and synchronized by the same clock signal CLK. As this type of shift register converts parallel data, such as an 8-bit data word into serial format, it can be used to multiplex many different input lines into a single serial DATA stream, which can be sent directly to a computer or transmitted over a communications line.

The input data is individually connected to each latch via a multiplexer at the input of each latch. The output of the previous latch and the parallel data input are connected to the input of the MUX and the output of the MUX is connected to the next flip-flop.

4.3.4.1 VHDL Implementation and Simulation

The code (Listing 4.15) shows VHDL description of Parallel-In Serial-Out (PISO) shift register. This register can shift the parallel in data to the right in series. The simulation results of the 4-bit PIPO shift register of Fig. 4.24 demonstrate the correct operation of this register.

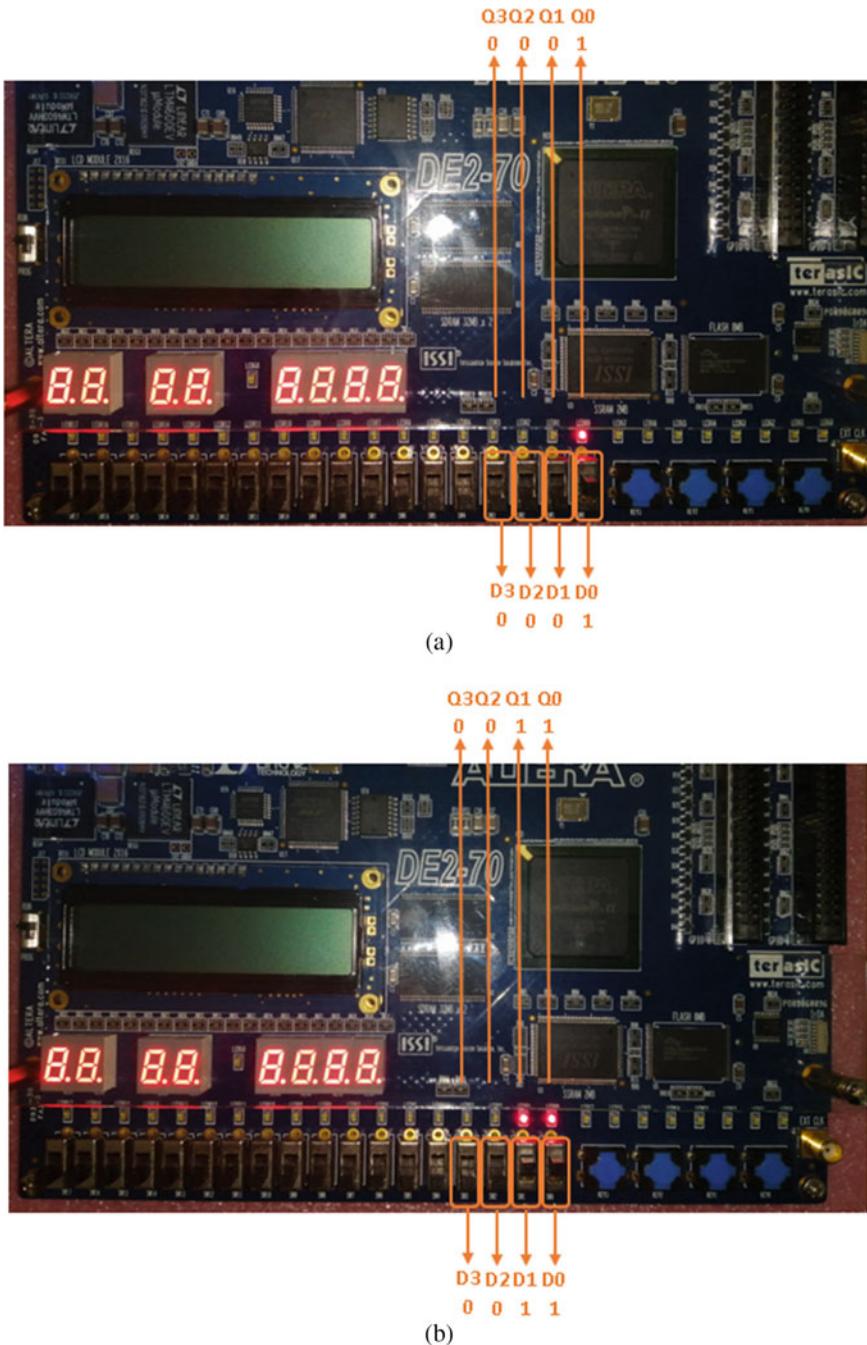


Fig. 4.22 Validation of the design the PIPO shift register on the FPGA platform. **a** Case for input data $D = 0001$, corresponding Q outputs of PIPO shift register are: “0001”; **b** Case for input data $D = 0011$, corresponding Q outputs of PIPO shift register are: 0011; **c** Case for input data $D = “1111”$, corresponding Q outputs of PIPO shift register are: “0011”

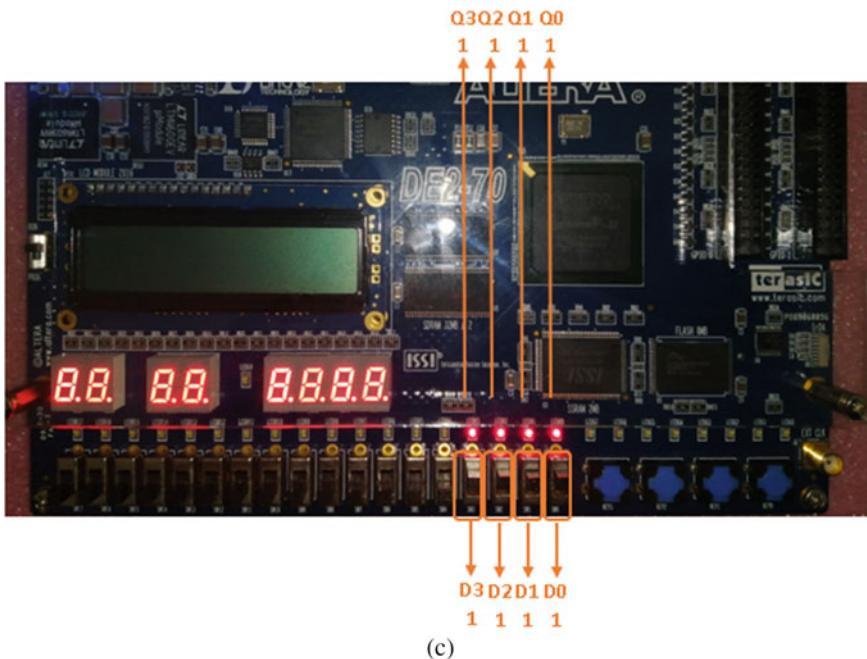


Fig. 4.22 (continued)

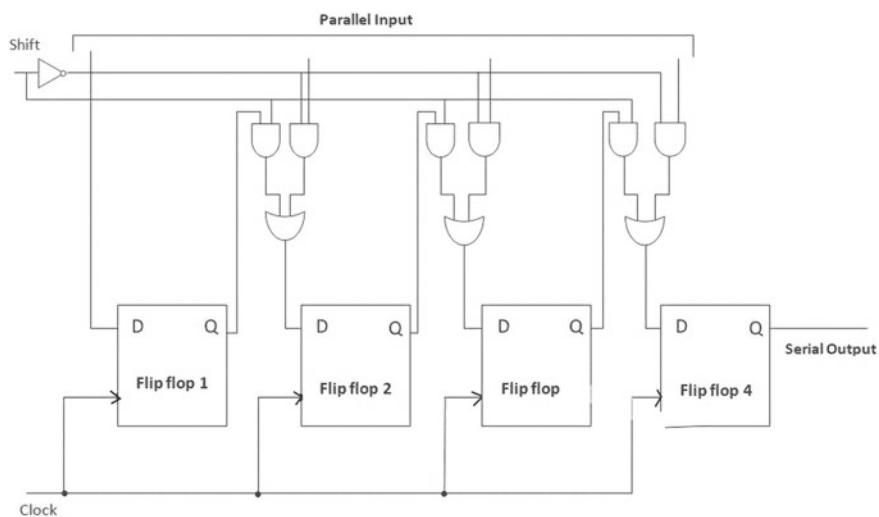


Fig. 4.23 The logic circuit of the PISO shift register



Fig. 4.24 Waveform simulation results of the 4-bit PISO shift register

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Register_PISO is
5  port(
6    clk, load, shift : in std_logic;
7    D: in std_logic_vector(3 downto 0);
8    Q: out std_logic );
9  end Register_PISO;
10
11 architecture beh of Register_PISO is
12
13  Signal temp: std_logic_vector(3 downto 0);
14 begin
15
16  process (clk, load)
17  begin
18    if (CLK'event and CLK='1') then
19      elsif load = '1' then
20        temp <= D;
21      elsif shift = '1' then
22        temp(0)<=D(0);
23        temp(1)<=D(1);
24        temp(2)<=D(2);
25        temp(3) <= D(3);
26      end if;
27
28    end process;
29    Q<= temp(3);
30  end beh;
```

Listing 4.15 VHDL code of 4-bit PISO shift register

4.3.4.2 Implementation and Validation on FPGA Platform

To test the design of the PISO shift register in the FPGA board, the pins assigned to the 4-bit SIPO shift register are shown in Table 4.5.

Figure 4.25 shows the validation of the design of the PISO shift register on the FPGA platform: the register can shift the parallel in data to the right in series.

Table 4.5 Pins assigned to the PISO shift register

Inputs: CLK and reset	Output: Q		
CLK_50	PIN_AD15	LedR[0]	PIN_AJ6
D[0] = SW[0]	PIN_AA23		
D[1] = SW[1]	PIN_AB26		
D[1] = SW[2]	PIN_AB25		
D[3] = SW[3]	PIN_AC27		
Load = SW[4]	PIN_AC26		
Shift = SW[5]	PIN_AC24		

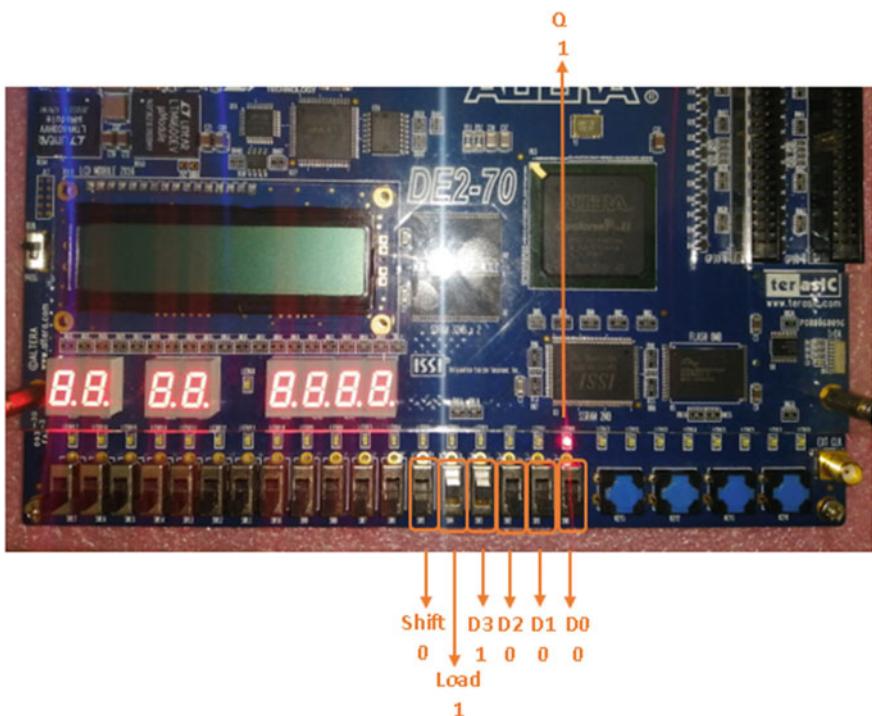


Fig. 4.25 Validation of the design of the PISO shift register, input data D = “1000”, shift = 0, Load = 1 and corresponding Q output of PISO shift register is equal to 1

4.4 Counters

Counters are important circuits of digital systems. They are used for many digital applications such as the counting events, measuring frequency and time, and increment memory addresses. It also used to calculate the total number of clock pulses and distance application devices. Counters can be implemented using registers such

as flip-flops, discussed in Sect. 4.2 of this chapter. They are generally classified into two categories: synchronous and asynchronous.

- A synchronous counter has all flip-flops change state synchronously with the clock input whether a periodic clock or a periodic pulse occurs.
- An asynchronous counter is made up of flip-flops that do not change state simultaneously with the clock input.

We will show how the counter circuits can be designed using D flip-flops.

4.4.1 Asynchronous Counter

A ripple counter is an asynchronous counter whose state changes are not controlled by a synchronization clock pulse. The structure of a 4-bit binary ripple counter is shown in Fig. 4.26. It consists of a series connection of D flip-flops, with the first flip-flop is clocked by the external clock pulse and then each successive flip-flop is clocked by the output of the preceding flip-flop.

The 4-bit binary ripple counter has two inputs, Clk, Reset, and a 4-bit outputs Q, represented by Q_3, Q_2, Q_1, Q_0 .

4.4.1.1 VHDL Implementation and Simulation

We use two ways to describe a 4-bit binary ripple counter in VHDL.

(a) First method

The first method uses behavioral modeling and it is based on the following points:

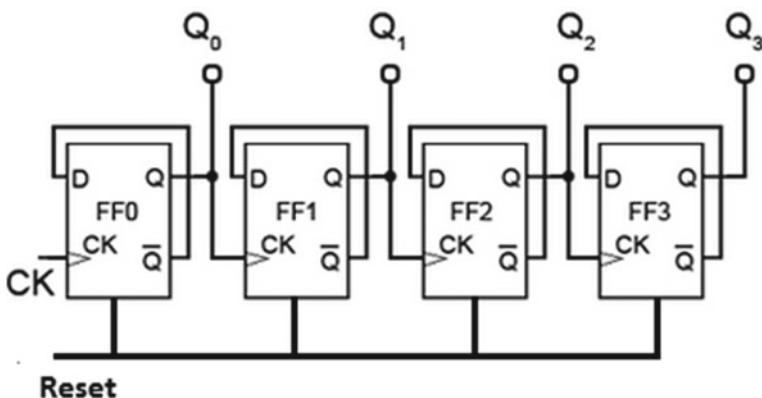


Fig. 4.26 Structure of a 4-bit binary ripple counter

- Generation of flip-flops control clock signals using simple assignments. The output of each flip-flop becomes the clock of the next flip-flop except for the first flip-flop;
- Describe the operation of the flip-flops using the If–then statement, executed inside the process;
- Generation of the ripple counter output Q by a simple assignment.

Listing 4.16 illustrates the 4-bit ripple counter VHDL program. During operation, if Reset is 1, the counter must reset its count value to zero (0000). Otherwise, the counter starts with binary 0 and increments by 1 with each CLK clock cycle. Once it reaches the number 15 (“1111”), it resets to “0000” and starts again.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4
5  entity Counter_Ripple is
6  port (
7      clk    : in std_logic;
8      reset : in std_logic;
9      Q : out std_logic_vector(3 downto 0)
10 );
11 end Counter_Ripple;
12
13 architecture Behavioral of Counter_Ripple is
14 -- signals declaration
15 signal clk_i, temp : std_logic_vector(3 downto 0);
16
17 begin
18     -- clocks
19     clk_i(0) <= clk;
20     clk_i(1)<=temp(0);
21     clk_i(2)<=temp(1);
22     clk_i(3)<=temp(2);
23
24     -- flip-flops
25 gen_cnt: for i in 0 to 3 generate
26     dff: process(reset, clk_i)
27     begin
28         if (reset = '1') then
29             temp(i) <= '1';
30         elsif (clk_i(i)'event and clk_i(i) = '1') then
31             temp(i) <= not temp(i);
32         end if;
33     end process;
34 end generate;
35
36     -- Output
37     Q <= not temp;
38
39 end Behavioral;
40

```

Listing 4.16 VHDL code for 4-bit ripple counter using behavioral modeling



Fig. 4.27 Simulation waveform of 4-bit ripple counter

The test bench code written in VHDL appears in Listing 4.17, provides a stimulus for simulating and verifying the functionality of the 4-bit ripple counter. The waveforms obtained from this simulation are shown in Fig. 4.27. The counter is functioning correctly and increments by 1 at each rising edge of the CLK clock. Thus, when the counter has reached its maximum value 15, the next clock edge will cause the counter to wind up and its next value will be zero.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity tb_Counter_Ripple is
5 end tb_Counter_Ripple;
6
7 architecture Behavioral of tb_Counter_Ripple is
8
9 component Counter_Ripple
10 Port(
11     clk: in std_logic; -- clock input
12     reset: in std_logic; -- reset input
13     Q: out std_logic_vector(3 downto 0) -- output 4-bit counter
14 );
15 end component;
16
17 signal stim_reset,stim_clk: std_logic;
18 signal stim_Q:std_logic_vector(3 downto 0);
19
20 begin
21     uut: Counter_Ripple port map(
22         clk => stim_clk,
23         reset=>stim_reset,
24         Q => stim_Q
25     );
26
27     -- Clock process definitions
28     clock_process :process
29     begin
30         stim_clk <= '0';
31         wait for 10 ns;
32         stim_clk <= '1';
33         wait for 10 ns;
34     end process;
35
36     -- Stimulus process
37     stim_proc: process
38     begin
39         stim_reset <= '1';
40         wait for 20 ns;
41         stim_reset <= '0';
42         wait;
43     end process;
44 end Behavioral;

```

Listing 4.17 Test bench VHDL code of the 4-bit ripple counter

(b) Second method

The second method to describe a 4-bit ripple counter uses structural modeling. It based on the Instantiating four copies of the flip-flops using the Port mapping command. The VHDL program of this structural modeling is provided in the following Listing 4.19. The D flip-flop VHDL program, shown below (see Listing 4.18), should be in the same directory as the one containing the ripple counter VHDL program file. This file is necessary for the correct execution of the ripple counter.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use ieee.std_logic_unsigned.all;
4
5  entity D_FF is
6    port (D, clk, reset : in std_logic;
7          Q, Qn : out std_logic);
8  end D_FF;
9
10 architecture behavioral of D_FF is
11 begin
12
13 process (clk, reset)
14 begin
15 if reset = '1' then
16   Q<='0';
17   Qn<= '1';
18 elsif clk'event and clk = '1' then
19   Q<= D;
20   Qn <= not D;
21 end if;
22 end process;
23
24 end behavioral;
```

Listing 4.18 VHDL code of the D flip-flop

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use ieee.std_logic_unsigned.all;
4
5  entity Counter_Ripple_struct is
6    port ( clk, reset : in std_logic;
7           Q : out std_logic_vector(3 downto 0));
8  end Counter_Ripple_struct;
9
10 architecture Structural of Counter_Ripple_struct is
11
12   --Flip_flop Component Declaration
13   component D_FF
14     Port ( Clk, D, reset: in STD_LOGIC;
15            Q, Qn : out STD_LOGIC);
16   end component;
17
18   --Signals declaration
19   signal temp: std_logic_vector(3 downto 0) := "0000";
20   signal qn0, qn1, qn2, qn3: std_logic;
21
22 begin
23   --Instantiating four copies of flip-flop component by Port mapping
24   FF0: D_FF port map(
25     reset => reset,
26     clk => clk,
27     D => qn0,
28     Q => temp(0),
29     -Qn => qn0);
30   FF1: D_FF port map(
31     reset => reset,
32     clk => temp(0),
33     D => qn1,
34     Q => temp(1),
35     -Qn => qn1);
36   FF2: D_FF port map(
37     reset => reset,
38     clk => temp(1),
39     D => qn2,
40     Q => temp(2),
41     -Qn => qn2);
42   FF3: D_FF port map(
43     reset => reset,
44     clk => temp(2),
45     D => qn3,
46     Q => temp(3),
47     -Qn => qn3);
48
49   Q<= temp;
50 end Structural;

```

Listing 4.19 VHDL code of the 4-bit ripple counter using structural modeling

4.4.1.2 Implementation and Validation on FPGA Platform

The considered ripple counter has two inputs Clk (clock signal), reset (control signal) and one output Q, represented by four outputs: Q₀, Q₁, Q₂ and Q₃. The clock signal is provided by the DE2-70 board. This board includes two oscillators that produce 28.86 and 50 MHz clock signals.

To test the design of the 4-bit ripple counter in the FPGA board, we need to:

- slow down the clock signal from 50 MHz to 1 Hz through a frequency divider (see Listing 4.10)
- use a BCD decoder for 7-segment display (see Listing 4.20).

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity bcd_7segment is
5    Port ( BCD : in STD_LOGIC_VECTOR (3 downto 0);
6           Seven_Segment : out STD_LOGIC_VECTOR (13 downto 0));
7  end bcd_7segment;
8
9  architecture one of bcd_7segment is
10 begin
11
12    Seven_Segment <= "11111111000000" when BCD = "0000" else
13    Seven_Segment <= "1111111111001" when BCD = "0001" else
14    Seven_Segment <= "11111110100100" when BCD = "0010" else
15    Seven_Segment <= "11111110110000" when BCD = "0011" else
16    Seven_Segment <= "11111110011001" when BCD = "0100" else
17    Seven_Segment <= "11111100100010" when BCD = "0101" else
18    Seven_Segment <= "1111110000010" when BCD = "0110" else
19    Seven_Segment <= "1111111111100" when BCD = "0111" else
20    Seven_Segment <= "11111110000000" when BCD = "1000" else
21    Seven_Segment <= "11111110010000" when BCD = "1001" else
22    Seven_Segment <= "11110011000000" when BCD = "1010" else
23    Seven_Segment <= "11110011111001" when BCD = "1011" else
24    Seven_Segment <= "11110010100100" when BCD = "1100" else
25    Seven_Segment <= "11110010110000" when BCD = "1101" else
26    Seven_Segment <= "11110010011001" when BCD = "1110" else
27    Seven_Segment <= "11110010010010" when BCD = "1111";
28  end one;

```

Listing 4.20 VHDL code of the BCD to 7-segment display decoder

Listing 4.21 shows the VHDL program of the ripple counter associated with the frequency divider and the BCD decoder for a 7-segment display. This program will display the counter outputs at a rate of 1 Hz on the two 7-segment displays of the FPGA card. The 50 MHz clock signal is assigned to pin PAD_15 and the switch SW0 used as a counter reset. The 4-bit output of the ripple counter is used to drive the inputs of the two 7-segment decoders, HEX0_D and HEX1_D of the FPGA board. Table 4.6 shows the pins assigned to the 4-bit ripple counter.

Table 4.6 Pins assigned of 4-bit ripple counter

Inputs: CLK and reset		Outputs: Q	
CLK_50	PIN_AD15	HEX0_D[0]	PIN_AE8
Reset = SW[0]	PIN_AA23	HEX0_D[1]	PIN_AF9
		HEX0_D[2]	PIN_AH9
		HEX0_D[3]	PIN_AD10
		HEX0_D[4]	PIN_AF10
		HEX0_D[5]	PIN_AD11
		HEX1_D[0]	PIN_AG13
		HEX1_D[1]	PIN_AE16
		HEX1_D[2]	PIN_AF16
		HEX1_D[3]	PIN_AG16
		HEX1_D[4]	PIN_AE17
		HEX1_D[5]	PIN_AF17

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity Ripple_counter_7segment is
5   Port ( clk, reset : in std_logic;
6         Seven_Segment : out STD_LOGIC_VECTOR (13 downto 0));
7   end Ripple_counter_7segment;
8
9 architecture one of Ripple_counter_7segment is
10
11 component frequency_div_1hz
12   Port (clk, reset: in std_logic;
13        clk_1hz: out std_logic);
14   end component;
15
16 component counter_Ripple
17   Port (clk, reset : in std_logic;
18        Q : out STD_LOGIC_VECTOR (3 downto 0));
19   end component;
20
21 component BCD_7segment
22   Port (BCD : in STD_LOGIC_VECTOR (3 downto 0);
23         Seven_Segment : out STD_LOGIC_VECTOR (13 downto 0));
24   end component;
25
26 signal S1: std_logic_vector (3 downto 0);
27 signal S2: std_logic_vector ;
28 begin
29   X1: frequency_div_1hz port map (clk, reset, S2);
30   X2: counter_Ripple port map (S2, reset,S1);
31   X3: BCD_7segment port map (S1, Seven_Segment);
32
33 end one;

```

Listing 4.21 VHDL code of the ripple counter associated with the frequency divider

The results observed on the 7-segment displays (Fig. 4.28) clearly show that the 4bit counter is functioning correctly. if Reset is 1, the counter output is zero (“0000”). Otherwise, the counter starts counting from zero to the number 15 (“1111”). Once this number is reached, it resets to “0000” and starts again.

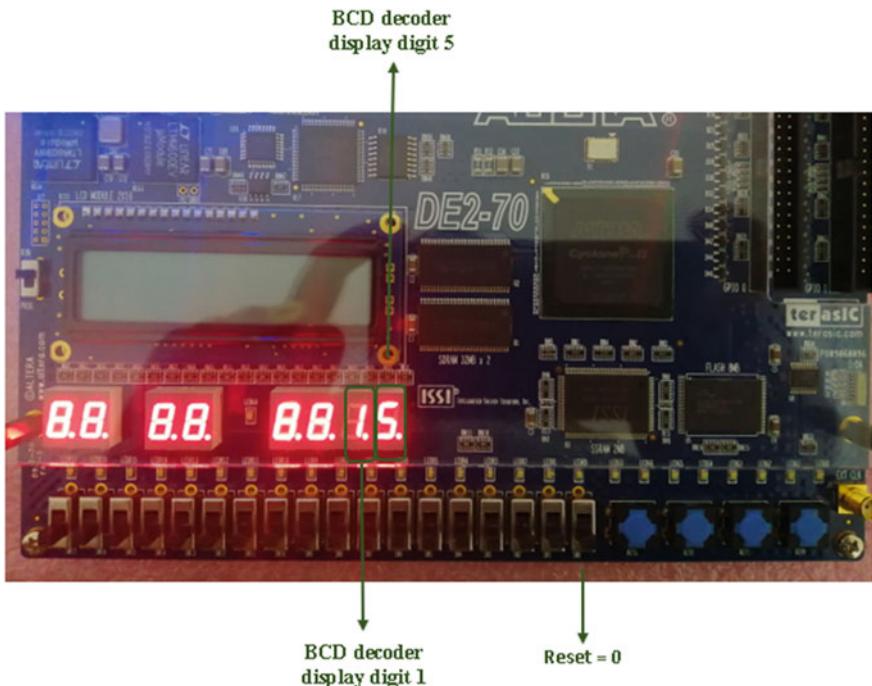


Fig. 4.28 Validation of the design of the 4-bit ripple counter on the FPGA board

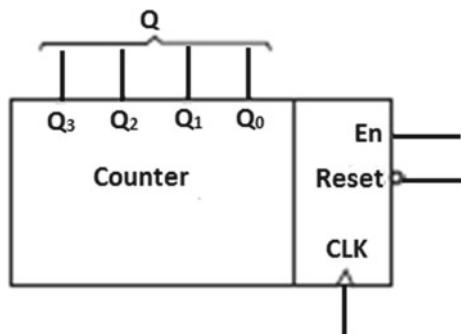
4.4.2 Synchronous Counter

In this type of counter, the “*clock*” pulses are applied to all the flip-flops of a counter simultaneously. So that all flip-flop outputs change at the same time, after only T_p (ns) of propagation delay from each D flip-flop. Therefore, this type of counter is faster than the ripple counter which requires nT_p (ns) after the rising edge of CLK for its output to be valid. Different types of synchronous counters namely, Binary counter Down count, Binary counter Up/Down count, BCD counter, Ring counter and Johnson counter will be discussed in this part. Their descriptions in VHDL as well as their simulation and implementation in FPGA platform will be illustrated and presented in detail.

4.4.2.1 Up Binary Counter

Figure 4.29 shows a simple synchronous binary counter. This counter accepts three inputs, clock input (CLK), reset control (Reset) and signal enable (EN), and one output Q represents the 4-bit values stored in the counter. Since addition is not defined in the *std_logic_1164* package, it must be declared as an unsigned type.

Fig. 4.29 Synchronous up binary counter



All operations are synchronized by the clock, and all state changes take place following the rising edge of the clock input. Operation of the counter is as follows:

On the rising edge of the clock:

- The counter is cleared when Reset = ‘0’,
- The counter is incremented when En = ‘1’. The statement $\text{temp} \leq \text{temp} + 1$ (Line 22 in Listing 4.20) increments the counter. “**temp**” is used as an internal signal to implement the counter functionality. When the counter is in state “1111”, the next increment takes it back to state “0000”.
- Outside of the counter process, the value **temp** is assigned to output **Q** using a signal assignment statement: $\text{Q} \leq \text{temp}$.
- VHDL implementation and simulation

The following code (Listing 4.22) shows a VHDL program which describes a 4-bit synchronous binary counter with synchronous reset, Enable signal and up count. It is modeled using a single process and with arithmetic operators (i.e. $+$). Remember that the “ $+$ ” operator is not defined in the *std_logic_1164* package. We need to include the “*numeric_std package*” in order to add this capability.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4
5
6  entity Counter_binary is
7    Port ( CLK,Reset, En : in STD_LOGIC;
8          Q : out unsigned(3 downto 0));
9  end Counter_binary;
10
11 architecture Behavioral of Counter_binary is
12   signal temp: unsigned(3 downto 0);
13
14 begin
15
16 process (CLK)
17 begin
18   if CLK'event and CLK = '1' then
19     if Reset = '0' then
20       temp <= "0000";
21     elsif En = '1' then
22       temp <= temp + 1;
23     end if;
24   end if;
25 end process;
26 Q<= temp;
27
28 end Behavioral;

```

Listing 4.22 VHDL program for 4-bit Up binary counter

The simulation timing diagrams in Fig. 4.30 illustrate the correct operation of the counter. When reset = 0, the output of the counter is equal to 0. Otherwise, when the enable signal En = 1, the counter counts up to 15 and resets to 0. Then it starts again from 0.

(b) Implementation and validation on FPGA platform

Figure 4.31 shows the validation of the design of the Up 4-bit binary counter on the LEDs and on the 7-segment displays of the FPGA board.

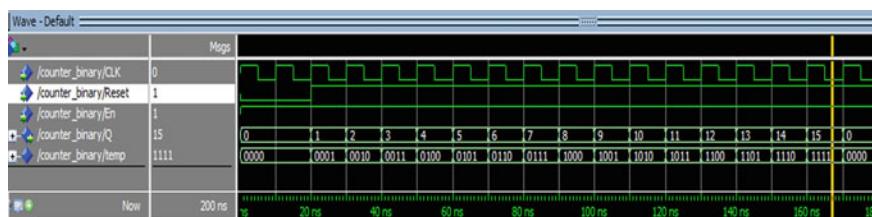


Fig. 4.30 Simulation waveform of 4-bit Up binary counter

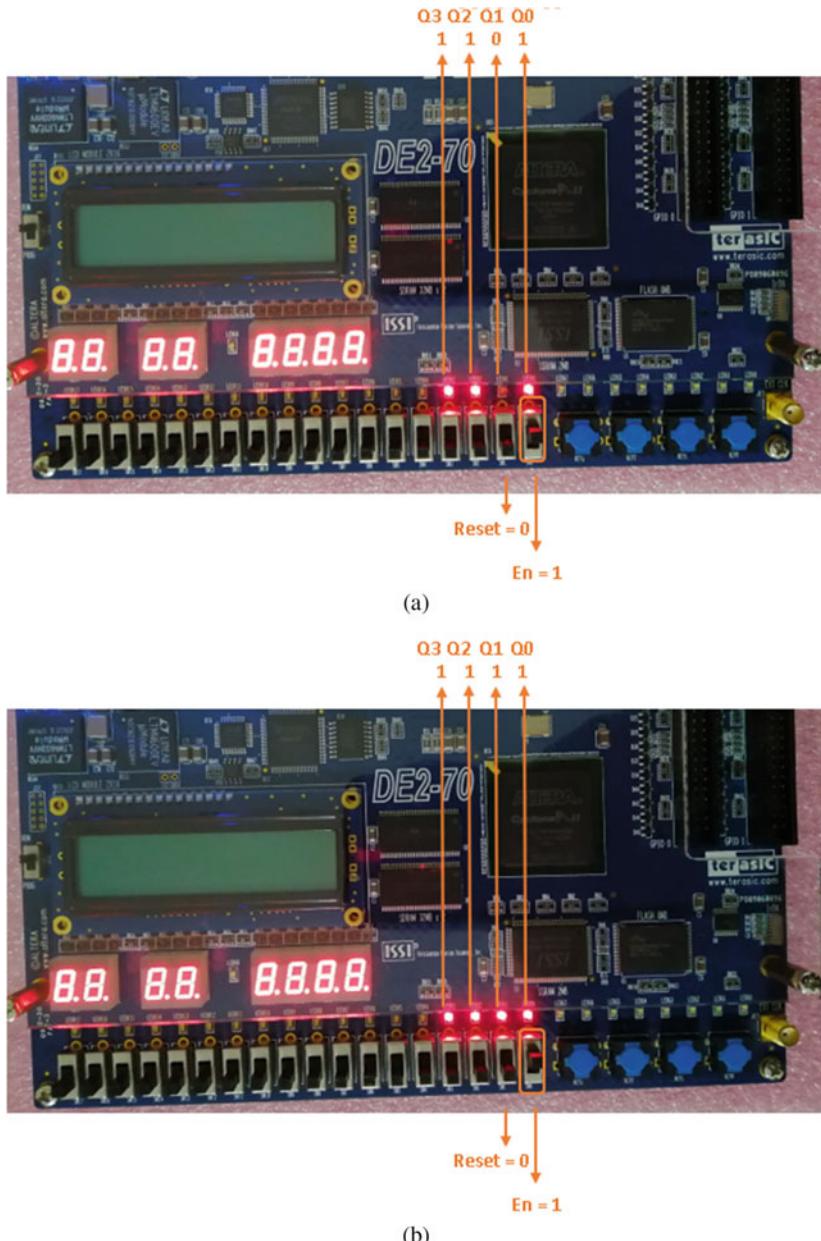


Fig. 4.31 Validation of the design of the 4-bit binary counter up on the FPGA board. **a** 4-bit binary counter when it reaches the digit 14: $Q = "1101"$; **b** 4-bit binary counter when it reaches the digit 15: $Q = "1111"$; **c** Display of the 4-bit binary counter outputs on 7 segments when it reaches the number 11; **d** Display of the 4-bit binary counter outputs on 7 segments when it reaches the number 13

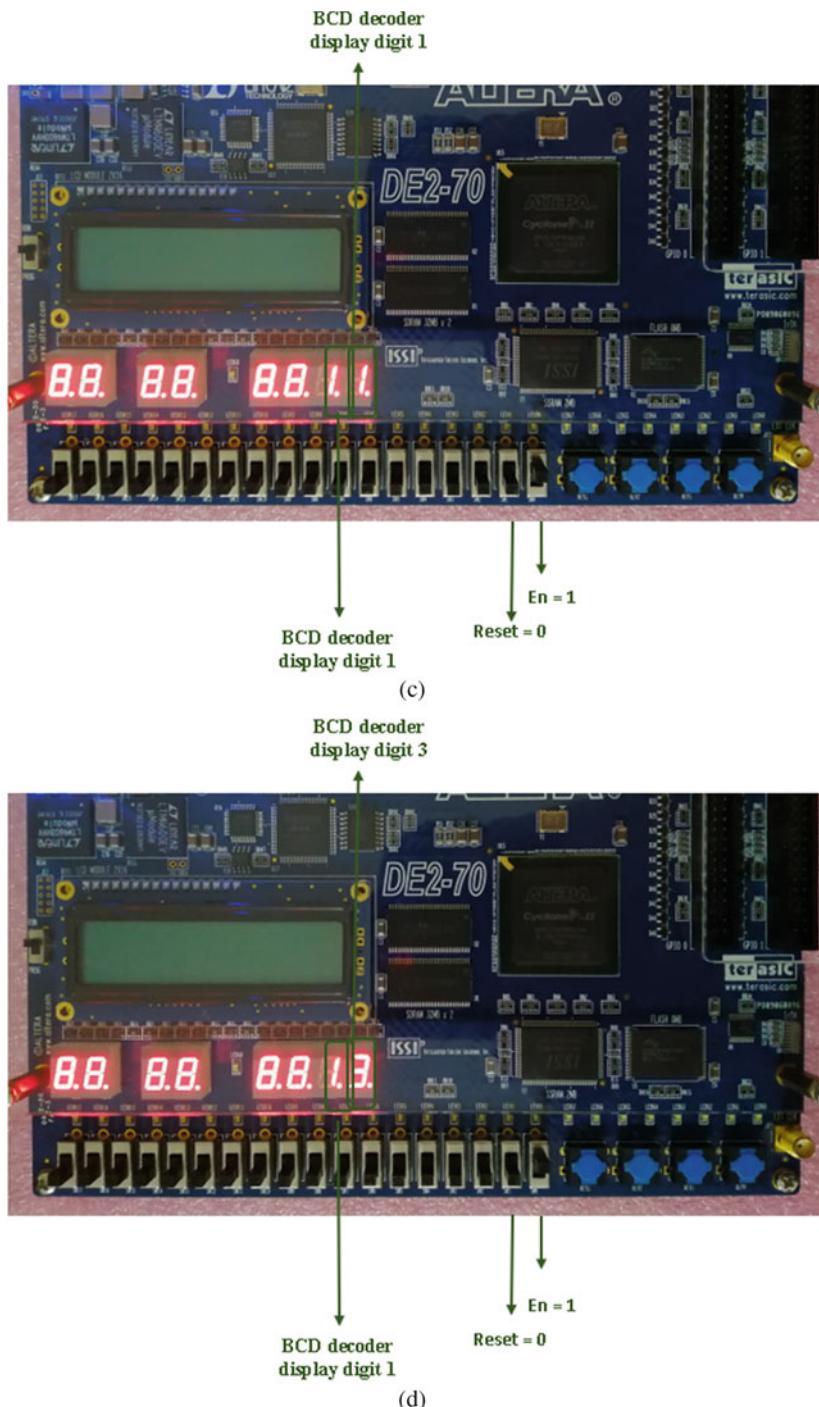


Fig. 4.31 (continued)

Figure 4.31a, b show the states of the counter when it reaches the digits 14 ($Q = "1101"$) and 15 ($Q = "1111"$) respectively. Whereas, Fig. 4.31c, d show the display of the counter outputs when it reaches digits 11 and 13, respectively.

4.4.2.2 Down Binary Counter

Until now, we have dealt with simple progressive counters whose binary value increases with each clock. To make a down counter, we need to use the statement $\text{temp} \leq \text{temp} - 1$ (line 21 of Listing 4.23) which decrements the count. To make a down counter, the statement $\text{temp} \leq \text{temp} - 1$ (line 21 of Listing 4.23), decrements the counter by 1, will replace the instruction $\text{temp} \leq \text{temp} - 1$ of the 4-bit counter up count program as shown in Listing 4.22. The example below shows the VHDL program for a 4-bit binary counter down count using an integer type.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.numeric_std.all;
4
5  entity Counter_down_count is
6    Port ( CLK,Reset, En : in STD_LOGIC;
7           Q : out unsigned(3 downto 0));
8  end Counter_down_count;
9
10 architecture Behavioral of Counter_down_count is
11   signal temp: integer;
12
13 begin
14
15 process (CLK)
16 begin
17   if CLK'event and CLK = '1' then
18     if Reset = '0' then
19       temp <= 15;
20     elsif En = '1' then
21       temp <= temp - 1;
22     end if;
23   end if;
24 end process;
25 Q<= to_unsigned(temp, 4);
26
27 end Behavioral;

```

Listing 4.23 VHDL code for 4-bit down binary counter

The simulation timing diagrams in Fig. 4.32 illustrate the correct operation of the counter. At each rising edge, the counter counts down from 1 from the number 15. Then it starts counting again until it reaches the number 15.

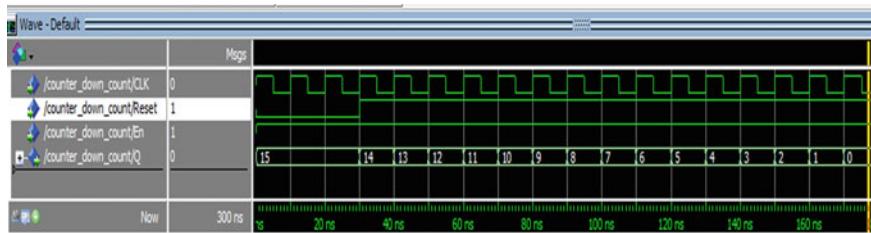


Fig. 4.32 Simulation waveform of 4-bit down binary counter

4.4.2.3 Up/Down Binary Counter

Figure 4.33 shows the logic symbol of a 4-bit synchronous Up/Down binary counter. The inputs are, CLK, Reset and control signal Up. The VHDL description of a 4-bit counter Up/Down count is shown in the Listing 4.24. When Reset is 1, the outputs are all 0. Otherwise, when the control input Up = 1, the circuit counts up, and when it is 0, the counter counts down.

The simulation results for this counter are shown in Fig. 4.34. After the reset (RST) signal has gone down and the control signal UP (Up = 1) remains high, the counter counts by 1 at each clock pulse, starting from the start value 0. When the control signal UP (Up = 0) goes low, the counter counts down by 1 at each rising edge of the clock. These simulations demonstrate the correct functioning of the meter.

Fig. 4.33 4-bit Up/Down counter

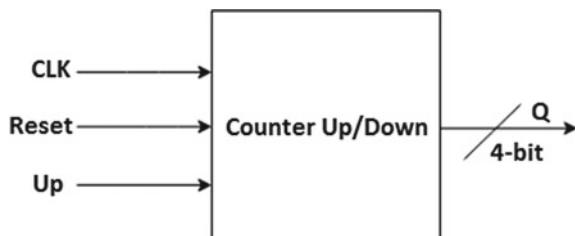


Fig. 4.34 Simulation waveform of 4-bit Up/Down binary counter

```

1   library IEEE;
2   use IEEE.std_logic_1164.all;
3   use IEEE.numeric_std.all;
4
5   entity Counter_up_down is
6   | generic(N : POSITIVE := 4);
7   | port(Clk, Reset, Up : in std_logic;
8   | Q : out std_logic_vector(N-1 downto 0));
9   end entity Counter_up_down;
10
11 architecture Rtl of Counter_up_down is
12   signal temp : unsigned(N-1 downto 0);
13   constant Cmax : unsigned(N-1 downto 0) := (others => '1');
14   constant Cmin : unsigned(N-1 downto 0) := (others => '0');
15
16 begin
17
18 process(Clk, Reset) is
19 begin
20 if Reset = '1' then
21 temp <= (others => '0');
22 elsif rising_edge(Clk) then
23 if Up = '1' and temp < Cmax then
24 temp <= temp + 1;
25 elsif Up = '0' and temp > Cmin then
26 temp <= temp - 1;
27 end if;
28 end if;
29 end process;
30 Q <= std_logic_vector(temp);
31
32 end architecture Rtl;

```

Listing 4.24 VHDL code for 4-bit Up/Down binary counter

4.4.2.4 BCD Counter

A Binary Coded Decimal Counter (BCD) is a serial digital counter used to count ten digits, from “0000” to “1001” and back to “0000”. A circuit for this counter is given in Fig. 4.35. The Clear input is used to provide an asynchronous reset for four digits in the counter. If EN = 1, the count value is incremented on the positive clock edge, and if EN = 0, the count value is unchanged. Each digit can take the values from “0000” to “1001”.

Fig. 4.35 A circuit of the 4-bit BCD counter



(a) VHDL implementation and simulation:

Listing 4.25 gives the VHDL code for the 4-bit BCD counter.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4  use IEEE.STD_LOGIC_ARITH.ALL;
5
6  entity counter_BCD is
7    port( EN, clk, reset: in std_logic;
8          BCD: out std_logic_vector(0 to 3));
9  end counter_BCD;
10
11 architecture Behavioral of counter_BCD is
12
13   signal temp: std_logic_vector(0 to 3);
14
15 begin
16
17   process(Clk,Reset)
18     begin
19     if Reset='1' then
20       temp <= "0000";
21     elsif(rising_edge(Clk)) then
22       if EN = '1' then
23         if temp="1001" then
24           temp<="0000";
25         else
26           temp <= temp + 1;
27         end if;
28       end if;
29     end process;
30   BCD <= temp;
31 end Behavioral;

```

Listing 4.25 VHDL code for the 4-bit BCD counter

Figure 4.36 depicts the simulations waveform for 4-bit BCD counter. As can be seen, the BCD counter count 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001. When it reaches digit 9, it returns to 0.

(b) Implementation and validation on FPGA platform:

Now we try to implement the design of the BCD counter on the FPGA platform and use the 7 segment display to visualize the counter values. The clock available on the

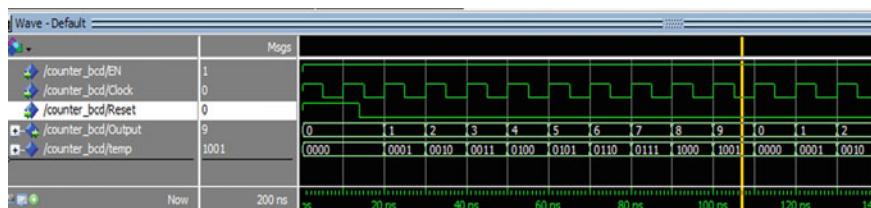


Fig. 4.36 Simulation waveform of 4-bit BCD counter

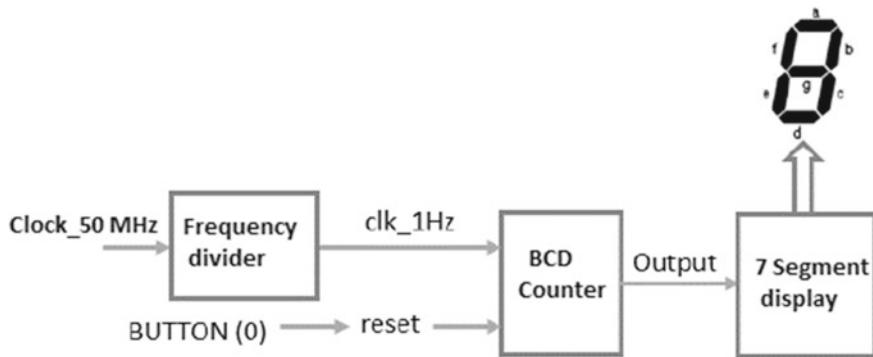


Fig. 4.37 Design of the 4-bit BCD counter on the FPGA board and 7 segment display

DE2_70 Altera card is a 50 MHz clock. This clock must therefore be considerably slowed down in order to be able to watch the counter values scroll. The application to be carried out can be represented by the diagram below, Fig. 4.37.

The counter will be controlled by a 1 Hz clock (clk_1Hz) and reset to zero by the reset signal controlled by the BUTTON (0) pushbutton. The value of the counter (Output) will be displayed on the 1st 7-segment display of the FPGA board. Each block of the diagram will be represented in VHDL by a process statement. In total, there are 3 process blocks:

Process N°1: Frequency divider of 1 Hz (Listing 4.26).

```

18  process (clk,reset)
19  begin
20  if rising_edge(clk) then
21  if clk_count > 50000000 then
22  clk_count<= (others => '0');
23  else
24  clk_count <= clk_count + 1;
25  end if;
26
27  if clk_count < 25000000 then
28  clk_1hz <= '1';
29  else
30  clk_1hz <= '0';
31  end if;
32  end if;
33  end process;

```

Listing 4.26 VHDL code for frequency divider

Process N°2: BCD counter (Listing 4.27).

```

36  process (clk_1hz,reset)
37  begin
38  if reset = '1' then
39  |temp <= "0000";
40  elsif rising_edge(clk) then
41  |if temp = "1010" then
42  | |temp <= "0000";
43  |else temp <= temp + 1;
44  |end if;
45  |end if;
46  end process;

```

Listing 4.27 VHDL code for BCD counter

Process N°3: Display on 7-Segment (Listing 4.28).

```

48  process (temp)
49  begin
50  case temp is
51  when "0000" => hex <= "1000000";
52  when "0001" => hex <= "1111001";
53  when "0010" => hex <= "0100100";
54  when "0011" => hex <= "0110000";
55  when "0100" => hex <= "0001101";
56  when "0101" => hex <= "0010010";
57  when "0110" => hex <= "0000010";
58  when "0111" => hex <= "0111000";
59  when "1000" => hex <= "0000000";
60  when others => hex <= "0010000";
61  end case;
62  end process;

```

Listing 4.28 VHDL code for Display on 7-Segment

For implementation on the FPGA platform and 7-segment display, Listing 4.29 shows the VHDL code of the BCD counter. As can be seen, the three processes are running in parallel simultaneously. In addition, these processes are interconnected by intermediate signals declared in the architecture.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_arith.all;
4  use ieee.std_logic_unsigned.all;
5  use ieee.numeric_std.all;
6
7  entity BCD_counter_7Segment is
8  port ( clk, reset : in std_logic;
9  hex : out std_logic_vector( 6 downto 0 );
10 end BCD_counter_7Segment;
11
12 architecture behavior of BCD_counter_7Segment is
13 signal temp: std_logic_vector(3 downto 0);
14 signal clk_count : std_logic_vector(25 downto 0);
15 signal clk_1hz: std_logic;
16
17 begin
18 process (clk,reset)
19 begin
20 if rising_edge(clk) then
21 if clk_count > 50000000 then
22 clk_count<= (others => '0');
23 else
24 clk_count <= clk_count + 1;
25 end if;
26
27 if clk_count < 25000000 then
28 clk_1hz <= '1';
29 else
30 clk_1hz <= '0';
31 end if;
32 end if;
33 end process;
34
35 process (clk_1hz,reset)
36 begin
37 if reset = '1' then
38 temp <= "0000";
39 elsif rising_edge(clk) then
40 if temp = "1010" then
41 temp <= "0000";
42 else temp <= temp + 1;
43 end if;
44 end if;
45 end process;
46
47 process (temp)
48 begin
49 case temp is
50 when "0000" => hex <= "1000000";
51 when "0001" => hex <= "1111001";
52 when "0010" => hex <= "0100100";
53 when "0011" => hex <= "0110000";
54 when "0100" => hex <= "0001101";
55 when "0101" => hex <= "0010010";
56 when "0110" => hex <= "0000010";
57 when "0111" => hex <= "0111000";
58 when "1000" => hex <= "0000000";
59 when others => hex <= "0010000";
60 end case;
61 end process;
62
63 end behavior;

```

Listing 4.29 VHDL code of the BCD counter for the 7-Segment display of the FPGA board

Table 4.7 Pins assigned of the 4-bit BCD counter (inputs and output)

Inputs: CLK and reset	
CLK_50	PIN_E16
Reset	PIN_AA23
Output	
HEX0_D[0]	PIN_AE8
HEX0_D[1]	PIN_AF9
HEX0_D[2]	PIN_AH9
HEX0_D[3]	PIN_AD10
HEX0_D[4]	PIN_AF10
HEX0_D[5]	PIN_AD11
HEX0_D[6]	PIN_AD12

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity ring_counter is
5  port(clk : in std_logic;
6  |   rst : in std_logic;
7  |   op : out std_logic_vector(3 downto 0));
8  end entity;
9
10 architecture beh of ring_counter is
11 signal opt : std_logic_vector(3 downto 0);
12
13 begin
14 process (clk,rst)
15 begin
16 if (rst = '1') then
17 opt <= "1000";
18 elsif (rising_edge(clk)) then
19 opt <= opt(0) & opt(3 downto 1);
20 end if;
21 end process;
22 op <= opt;
23 end beh;
```

Listing 4.30 VHDL code for 4-bit Ring counter

Table 4.7 shows the pins assigned for 4-bit BCD counter.

The results observed on the 7-segment displays (Fig. 4.38) clearly show that the 4bit BCD counter is functioning correctly. if Reset is 1, the counter output is zero (0000). Otherwise, the counter starts counting from “0000” to “1001” and back to “0000”.

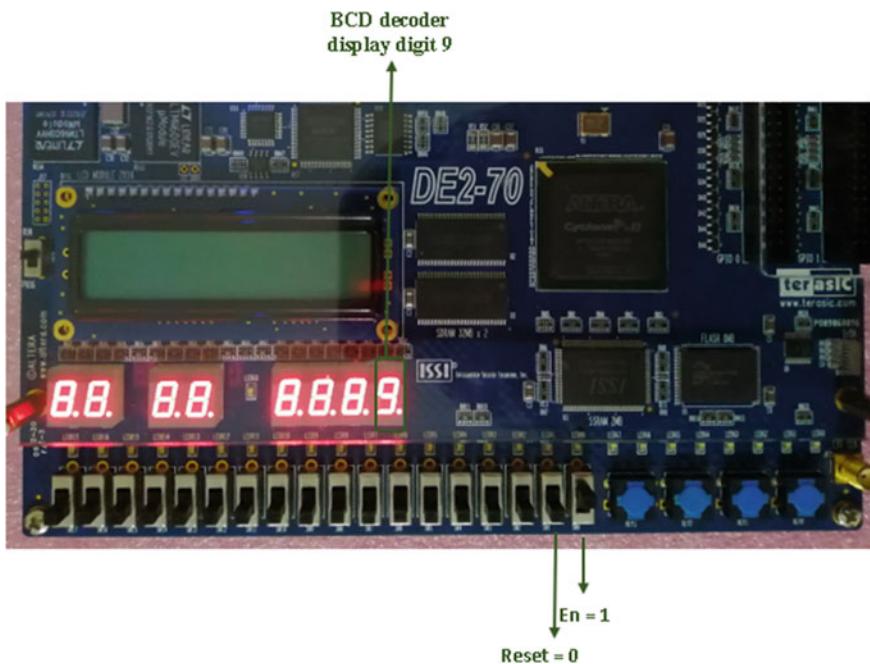


Fig. 4.38 Validation of the design of the 4-bit BCD counter on the FPGA board

4.4.3 Ring Counter

Ring counter is a typical application of shift register. The output of the last flip-flop of shift register is feed back to the input of first flip-flop. Such a shift register is also called a ring counter. For a 4-bit Ring Counter, the data pattern will repeat every four clock pulses. If pattern is “1000”, then it will generate “0100”, “0010”, “0001”, “1000” and so on. Logic diagram of 4-bit ring counter using D flip-flops is shown Fig. 4.39.

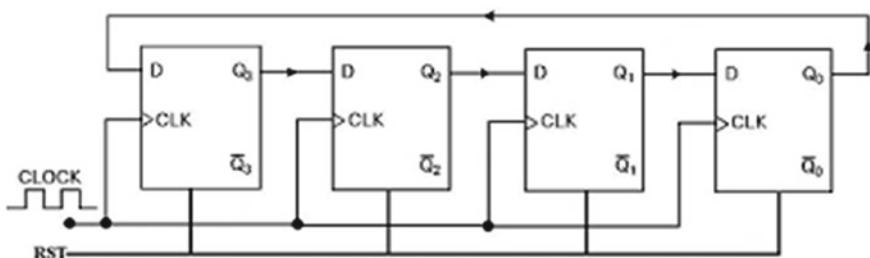


Fig. 4.39 Logic diagram of 4-bit ring counter using D flip-flops

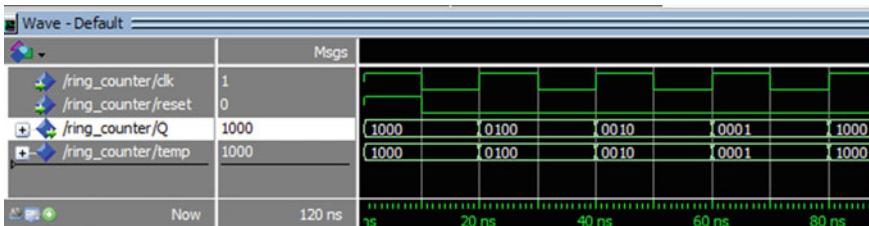


Fig. 4.40 Simulation waveform of 4-bit Ring counter

The VHDL description of the 4-bit ring counter is shown in Listing 4.30. When the reset signal is asserted (line 16, Listing 4.30), the counter is set to the value “1000”. At each rising clock edge, the counter value is rotated to the left. The simulation results for this counter are shown in Fig. 4.40. The counter circulates through “0100”, “0010”, “0001” and “1000” states, and then repeats.

4.4.4 Johnson Counter

Figure 4.41 shows a 4-bit Johnson counter, known as the Twisted-Ring counter. It is exactly the same as the ring counter except that the inverted output of the last flip-flop is connected to the input of the first flip-flop. The VHDL description of a 4-bit Johnson counter is shown in Listing 4.31. The simulation results are shown in Fig. 4.42. A 4-bit Johnson counter passes blocks of four logics “0” and then passes four logics “1”. So it will produce 8-bit pattern. For example, “1000” is the initial output then it will generate “1100”, “1110”, “1111”, “0111”, “0011”, “0001”, “0000” and this patterns will be repeated.

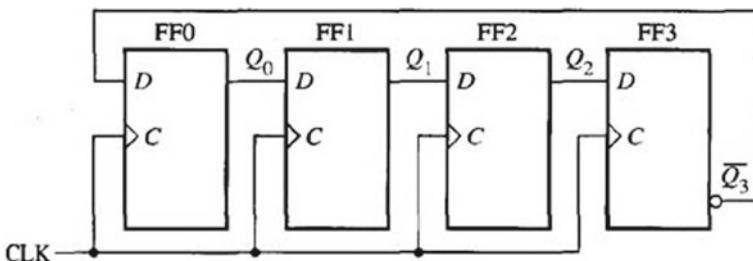


Fig. 4.41 4-bit Johnson counter using D flip-flop

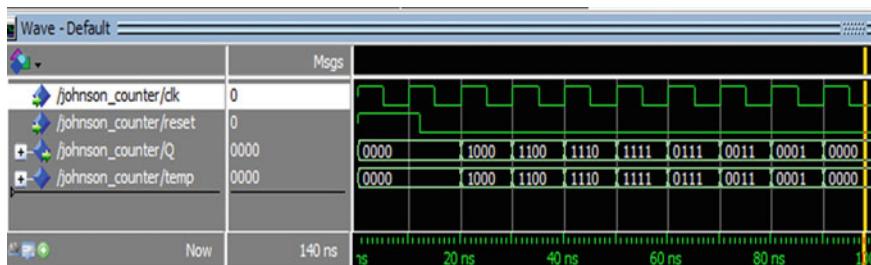


Fig. 4.42 Simulation waveform of 4-bit Johnson counter

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity johnson_counter is
5  port(clk : in std_logic;
6       reset : in std_logic;
7       Q : out std_logic_vector(3 downto 0));
8  end entity;
9
10 architecture beh of johnson_counter is
11 signal temp : std_logic_vector(3 downto 0);
12
13 begin
14 process (clk,reset)
15 begin
16 if (reset = '1') then
17   temp <= "0000";
18 elsif (rising_edge(clk)) then
19   temp <= (not temp(0)) & temp(3 downto 1);
20 end if;
21 end process;
22 Q <= temp;
23 end beh;
```

Listing 4.31 VHDL code for 4-bit Johnson counter

4.5 Summary

In this chapter, the basic memory elements for storing information such as latches and D Flip-flops are described in VHDL. They are fundamental building blocks for all sequential circuits. The main difference between a latch and a flip-flop has been discussed and explained. Afterward, several approaches to design a shift register in VHDL, such as structural, data flow and behavioral descriptions have been used. Here we have focused on four types of shift registers like SIPO, SISO, PIPO and PISO. Finally, we have examined the counters as popular sequential circuit families. These counters can be designed using D flip-flops and they are generally classified

into two categories synchronous and asynchronous. The operation of these circuits is verified on the one hand by simulations carried out using ModelSim tool and on the other hand by their implementation into an FPGA platform.

Chapter 5

Finite State Machines



Abstract This chapter deals with the implementation of Finite State Machines (FSM) in VHDL for the modeling and design of sequential digital circuits. State machines can be classified into two types: Moore and Mealy. The state transition diagram is an efficient design tool that can be used to describe finite state machines represented as a set of transitions, which may or may not be labeled. The main design guidelines for FSM as well as the importance of multi-process FSM will be discussed in this chapter. Additionally, this chapter illustrates practical examples of FSMs such as the 4-bit BCD counter, sequence detector and parity checker that are useful in actual practice. Finally, test and validation of these practical examples have been done on the FPGA platform.

5.1 Introduction

Finite State Machine (FSM) is a powerful technique used to design sequential circuits from a written specification. This technique can be utilized in many fields of study e.g. neural networks, artificial intelligence, mathematics, games, robotics and sequential flow of data.

There are two most popular types of FSMs namely the Mealy machine and the Moore machine. The main difference between these machines is in the calculation of the value of the following state. In a Moore circuit, the outputs depend only on the current state values while in a Mealy machine, the outputs depend on both the present states and external inputs.

In what follows, we will illustrate with explanatory diagrams the architectures of each type of machine. A focus will be made on the VHDL implementation of FSM. The three separate processes or two processes approaches to describing a FSM in VHDL will be detailed and the difference between the two Moore and Mealy styles will be clarified. Finally, we finish this chapter by presenting some examples of FSM applications.

5.2 Moore Machine

Figure 5.1 shows the block diagram of Moore machine. It consists of the following three main blocks:

- Combinational logic for next state:

This block (combinatorial circuit) receives the inputs and the current state “current_state” to generate the next state as input to the flip-flops (Current state register block) to store it.

- Current state register:

This block (sequential block) is responsible for updating the “current_state” of FSM according to the valid input at each active edge of the clock (clk). The synchronous or asynchronous reset input is used to initialize the state register.

- Output logic:

The outputs of the Moore FSM are calculated by a combinatorial logic block whose only inputs are the current states, outputs of the flip-flops.

In the Moore model, the outputs of the sequential circuit are synchronized with the clock because they depend only on flip-flop outputs that are synchronized with the clock.

The Moore machine state diagram is shown in Fig. 5.2. It has two states, states 1 and state 2, represented by a circle or a “bubble”. On each state is indicated its name as well as the value of the output. Since the outputs in a Moore FSM depend only on the current stat, the outputs are shown inside the state bubble: state 1/output 1 and state 2/output 2.

The transition from one state to another is indicated by the transition arc and transition condition. Depending on the changes in the input or transition condition, the state transition occurs.

Fig. 5.1 Block diagram of Moore machine

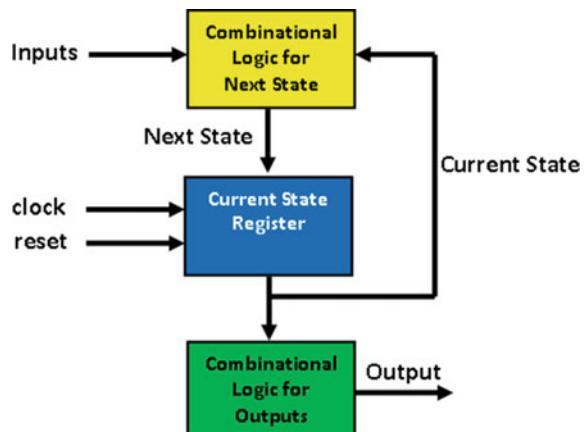
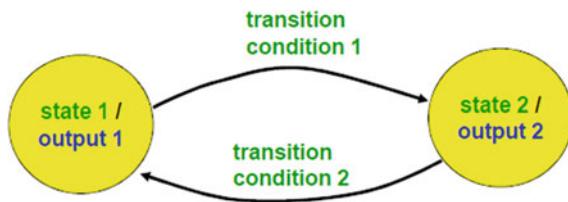


Fig. 5.2 State diagram representation of Moore machine



5.3 Mealy Machine

Unlike the Moore machine, in Mealy machine the outputs depend on both the current state (current_state) and present input (input). As shown in Fig. 5.3, the Mealy FSM architecture has three key blocks:

- Combinational logic for next state,
- Current state register, and
- Combinational logic for outputs.

The outputs of the Mealy FSM may change if the inputs change during the clock pulse period. Because of this, the outputs can have momentary false values due to the delay encountered between the moment the inputs change and the moment the flip-flop outputs change.

In order to synchronize an asynchronous Mealy machine, the inputs of the sequential circuit must be synchronized with the clock and the outputs must be sampled only during the clock-pulse transition.

The state diagram of the Mealy machine shown in Fig. 5.4 has two states: state 1 and state 2. Bubble indicates the state, and the transition from one state to other is indicated by the transition arc. Since the output is a function of the current state and the input, the transition arc displayed in the state diagram indicates the transition/output condition.

Fig. 5.3 Block diagram of Mealy machine

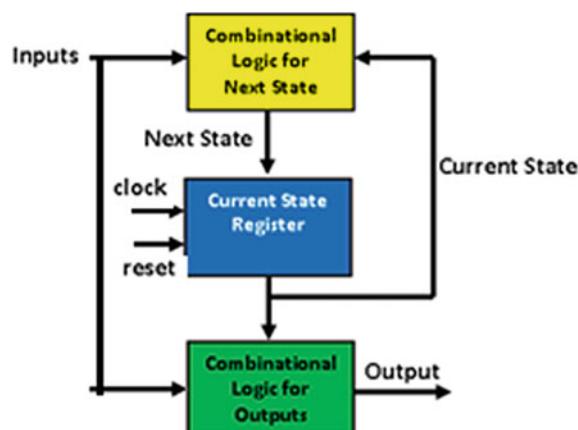
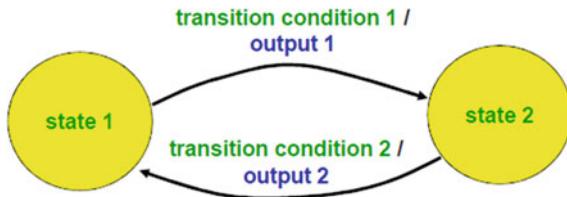


Fig. 5.4 State diagram representation of Mealy machine



5.4 VHDL Implementation for Moore Machine

There are many methods that can be used to describe FSMs using VHDL. In this section, we use the approach of a three separate process blocks or using two process blocks to describe an FSM.

5.4.1 Moore Machine Described Using Three Processes

The VHDL modeling procedure for Moore machine is described as follows:

- We use enumeration data type to represent the states.
type state_type is (state0, state1, state2, state3);
- We declare signals for the current state and next state.
signal current_state, next_state:state_type;
- We use one process for every component in the FSM; process for the current state register, process for the next state logic and process for output logic.

Listing 5.1 shows the VHDL description of the Moore machine with three processes. In the declarative part of the architecture, we declare an enumeration type to specify the set of states for the FSM, for example:

12	type state is (state1, state2, state3, state4);
13	signal current_state, next_state: state;

Each value in the enumeration type represents one of the FSM states.

This machine can be described with three separate processes integrated into the body of the architecture, as shown by the model described in the Listing 5.1. One is used to update the current state, the second process is used for the next state, and finally the last process is used to determine the outputs of the circuit.

```

1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity Moore_FSM is
5   port(clk, rst: in std_logic;
6       inp1, inp2,inp3,inp4: in std_logic;
7       outp1, outp2,outp3,outp4: out std_logic );
8   end Moore_FSM;
9   -----
10
11  architecture beh of Moore_FSM is
12  type state is (state1, state2, state3,state4);
13  signal current_state, next_state: state; FSM has four states state1,  
state2, state3 and state4  
Current_state and next_state are  
declared of type of state
14
15  begin
16  -----
17  state_register: process(clk, rst)
18  begin
19  if(rst='1') then
20  current_state<=initial_state;
21  elsif(clk'event and clk='1') then
22  current_state<=next_state; This process is used to update  
the current_state
23  end if;
24  end process; On rising edge of clock, the next_state  
is assigned to the current_state
25
26  next_state_logic : process (current_state,inp1, input2,...)is
27  begin
28  case current_state is
29  when state1 =>
30  if condition1 then
31  next_state <= state_value;
32  elsif condition2 then
33  next_state <= state_value;
34  else
35  next_state <= state_value;
36  end if;
37  when state2 =>
38  ...
39  end case;
40  end process ;
41

```

```

42  output_logic : process (current_state,input1, input2, ...) is
43  begin
44  case current_state is
45  when state1 =>
46  | output1 <= value; output2 <= value; ...
47  | if condition1 then
48  | | output1 <= value; output2 <= value; ...
49  | elsif condition2 then
50  | | output1 <= value; output2 <= value; ...
51  | |
52  | else
53  | | output1 <= value; output2 <= value; ...
54  | end if;
55  |
56  when state2 =>
57  ...
58  end case;
59  end process;
60
61 end beh;

```

This process is used for the output logic for Moore machine

Listing 5.1 FSM design template for Moore machine described with three processes

5.4.2 Moore Machine Described with Two Processes

There is another approach using two processes to describe the Moore machine. One of them is used for updating the current state while the other process is used for the determining the outputs of the circuit and next states. Often, the processes representing the two combinational logic blocks may be combined.

As shown in Listing 5.2, the next state logic and output logic processes are merged into one process.

```

25  --
26  -- One process used for the next states and the output logic for Moore machines
27  next_state_output_logic : process (current_state,input1, input2,...)is
28  begin
29  case current_state is
30  | when state1 =>
31  | if condition1 then
32  | | next_state <= state_value;
33  | elsif condition2 then
34  | | next_state <= state_value;
35  | else
36  | | next_state <= state_value;
37  | end if;
38  | output1 <= value; output2 <= value; ...
39
40  | when state2 =>
41  | ...
42  end case;
43  end process ;
44

```

If-then statement

Unlike the Mealy style, Moore FSM outputs are declared outside of the If-then statement.

Listing 5.2 One process unit for the determination of circuit outputs and the next states

In some designs, if Moore style FSM outputs are only active for one or two states, they can be expressed using simple concurrent assignments rather than using a process, for example:

```

39     output1 <= '1' when current_state = state1 else '0';
40     output2 <= '1' when current_state = state3
41     or current_state = state4 else
42     '0';

```

The template used for the process unit responsible to update the current state is given in Listing 5.3.

```

16
17 state_register: process(clk, rst)
18 begin
19 if(rst='1') then
20   current_state<=initial_state;
21 elsif(clk'event and clk='1') then
22   current_state<=next_state;
23 end if;
24 end process;
25

```



Process responsible for updating the current_state

Listing 5.3 Process unit for updating the current state

When all the parts are integrated, the VHDL code for Moore state machine happens to be as in Listing 5.4.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Moore_FSM is
5  port(clk, rst: in std_logic;
6       inp1, inp2,inp3,inp4: in std_logic;
7       outp1, outp2,outp3,outp4: out std_logic );
8  end Moore_FSM;
9
10 architecture beh of Moore_FSM is
11
12 type state is (state1, state2, state3,state4);
13 signal current_state, next_state: state;
14
15 begin
16
17 state_register: process(clk, rst)
18 begin
19 if(rst='1') then
20   current_state<=initial_state;
21 elsif(clk'event and clk='1') then
22   current_state<=next_state;
23 end if;
24 end process;
25

```

```

25  --
26  |-- One process used for the next states and the output_logic for Moore machines
27  next_state_output_logic : process (current_state,input1, input2,...)is
28  begin
29  case current_state is
30  when state1 =>
31  if condition1 then
32  next_state <= state_value;
33  elsif condition2 then
34  next_state <= state_value;
35  else
36  next_state <= state_value;
37  end if;
38  output1 <= value; output2 <= value; ...
39
40  when state2 =>
41  ...
42  end case;
43  end process ;
44
45  end beh;

```

Listing 5.4 FSM design template for Moore machine described with two processes

5.5 VHDL Implementation for Mealy Machine

To implement the Mealy machine in VHDL, the same strategy presented above is adopted. It consists to use either two or three processes in order to describe all the blocks of Mealy. In this example, we describe the Mealy machine using three processes. The principle remains the same for the case of two processes. Listing 5.5 depicts the template used to code the Mealy FSM with two processes.

As can be seen, in the entity, the declarative part of the architecture of Mealy state machines as well as the process for updating the current state are the same as those written for Moore state machines (see Listing 5.4). The only difference occurs in the implementation of the process written for the determination of circuit outputs and next states. In Mealy machines, the circuit outputs are determined considering the values of external inputs. For this reason, we should determine the outputs of the circuits after checking the values of external inputs by an if statement as in Listing 5.5.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Mealy_FSM is
5  port(clk, rst: in std_logic;
6  inp1, inp2,inp3,inp4: in std_logic;
7  outp1, outp2,outp3,outp4: out std_logic );
8  end Mealy_FSM;
9
10
11
12  type state is (state1, state2, state3,state4);
13  signal current_state, next_state: state;
14 begin
15
16  -- Process used to update the current state
17  p1_state_register: process(clk, rst)
18  begin
19  if(rst='1') then
20  current_state<=initial_state;
21  elsif(clk'event and clk='1') then
22  current_state<=next_state;
23  end if;
24  end process;
25
26  --Process used for the next states and the output logic for Mealy machines
27  p2_next_state_output_logic : process (current_state,inp1, input2,...)is
28  begin
29  case current_state is
30  when state1 =>
31  if condition then
32  next state <= state-value;
33  output1 <= value; output2 <= value; ...
34  elsif condition2 then
35  next state <= state-value;
36  output1 <= value; output2 <= value; ...
37  ...
38  else
39  next state <= state-value;
40  output1 <= value; output2 <= value; ...
41  end if;
42
43  when state2 =>
44  ...
45  end case;
46  end process;
47
48 end beh;
```

If-then statement

Mealy FSM outputs are declared inside the IF-then statement.

Listing 5.5 FSM design template for Mealy machine described with two processes

5.6 Examples: States Machines in VHDL

This section presents a number of examples of FSMs described by using the VHDL constructs. All of these examples have been modeled using the two-process approach: process “present_state_register” for updating of the “present_state” and another process for the “next_state_function and output_function”.

5.6.1 4-Bit BCD Counter FSM

Figure 5.5 displays a diagram for a 4-bit BCD counter. It has two inputs, the clock signal (clk) and the asynchronous reset signal (rst), and a single output (count) coded on 4 bits. The BCD counter counts up from “0000”, “0001”, ..., “1001” and then go back to “0000” and so on.

BCD counter can be easily implemented using Moore state machines. For a 4-bit counter, we have 4 flip-flops, and therefore 10 states in total. Moore state diagram of the 4-bit binary counter is shown in Fig. 5.6. The states are called S0, S1, ..., S9, each name corresponding to the binary value of the output.

5.6.1.1 VHDL Implementation and Simulation

The entity and the declarative part of the architecture of the 4-bit BCD counter are described in Listing 5.6. An enumerated type to specify the set of states for the FSM machine appears in lines 10–11.

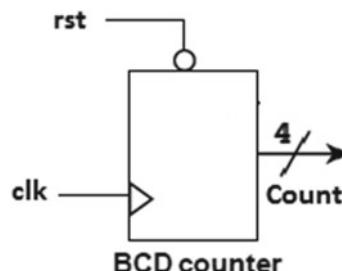


Fig. 5.5 Block diagram of a 4-bit BCD counter

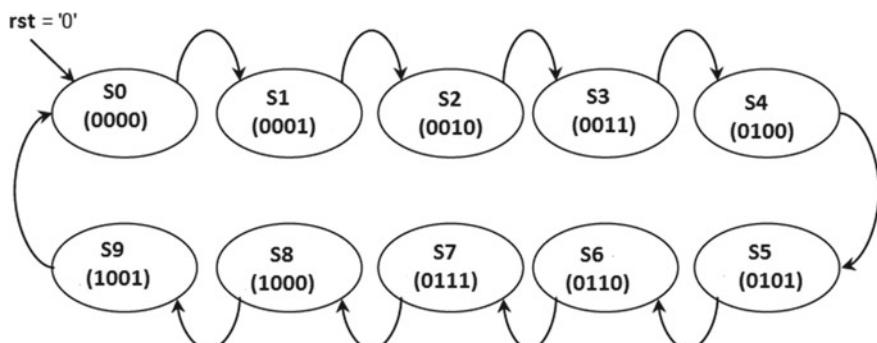


Fig. 5.6 Moore state diagram for 4-bit BCD counter

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity FSM_4bit_counter is
5  port ( clk, rst: in std_logic;
6        count: out std_logic_vector (3 downto 0));
7  end FSM_4bit_counter;
8
9  architecture state machine of FSM 4bit counter is
10 type state is (S0,S1,S2,S3,S4,S5,S6,S7,S8,S9);
11 signal pr_state, nx_state: state;
12 begin
13
```

Listing 5.6 Entity and the declarative part of the architecture of the 4-bit counter

The design of the first process (P1) for the present state update operation is shown in Listing 5.7, lines 14–21.

```
14 P1: process (rst, clk)
15 begin
16 if (rst='1') then
17   pr_state <= S0;
18 elsif (clk'event and clk='1') then
19   pr_state <= nx_state;
20 end if;
21 end process;
22
```

Listing 5.7 VHDL code for process (P1) responsible for the present state update operation

The second process (P2) used to determine the circuit outputs and next states is described in Listing 5.8, lines 23–57. This process is sensitive to the “present_state” signal. It is therefore only active after the completion of the first process assuming that the current state is updated in the first process.

```

23  P2: process (pr_state)
24  begin
25  case pr_state is
26    when S0 =>
27      count <= "0000";
28      nx_state <= S1;
29    when S1 =>
30      count <= "0001";
31      nx_state <= S2;
32    when S2 =>
33      count <= "0010";
34      nx_state <= S3;
35    when S3 =>
36      count <= "0011";
37      nx_state <= S4;
38    when S4 =>
39      count <= "0100";
40      nx_state <= S5;
41    when S5 =>
42      count <= "0101";
43      nx_state <= S6;
44    when S6 =>
45      count <= "0110";
46      nx_state <= S7;
47    when S7 =>
48      count <= "0111";
49      nx_state <= S8;
50    when S8 =>
51      count <= "1000";
52      nx_state <= S9;
53    when S9 =>
54      count <= "1001";
55      nx_state <= S0;
56  end case;
57  end process;
58

```

Listing 5.8 Second process (P2) for the determination of circuit outputs and next states

When all parts of the VHDL code are integrated, the complete 4-bit counter program is found to be as in Listing 5.9.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity FSM_4bit_counter is
5  port ( clk, rst: in std_logic;
6        count: out std_logic_vector (3 downto 0));
7  end FSM_4bit_counter;
8
9  architecture state_machine of FSM_4bit_counter is
10 type state is (S0,S1,S2,S3,S4,S5,S6,S7,S8,S9);
11 signal pr_state, nx_state: state;
12 begin
13
14  P1: process (rst, clk)
15  begin
16    if (rst='1') then
17      pr_state <= S0;
18    elsif (clk'event and clk='1') then
19      pr_state <= nx_state;
20    end if;
21  end process;
22
23  P2: process (pr_state)
24  begin
25    case pr_state is
26    when S0 =>
27      count <= "0000";
28      nx_state <= S1;
29    when S1 =>
30      count <= "0001";
31      nx_state <= S2;
32    when S2 =>
33      count <= "0010";
34      nx_state <= S3;
35    when S3 =>
36      count <= "0011";
37      nx_state <= S4;
38    when S4 =>
39      count <= "0100";
40      nx_state <= S5;
41    when S5 =>
42      count <= "0101";
43      nx_state <= S6;
44    when S6 =>
45      count <= "0110";
46      nx_state <= S7;
47    when S7 =>
48      count <= "0111";
49      nx_state <= S8;
50    when S8 =>
51      count <= "1000";
52      nx_state <= S9;
53    when S9 =>
54      count <= "1001";
55      nx_state <= S0;
56    end case;
57  end process;
58
59 end state_machine;

```

Listing 5.9 The complete 4-bit counter code using Moore machine

Simulation results for the 4-bit BCD counter are depicted in Fig. 5.7. As can be seen, the output (count) increment from 0 to 9, and then restarts from 0 again. The

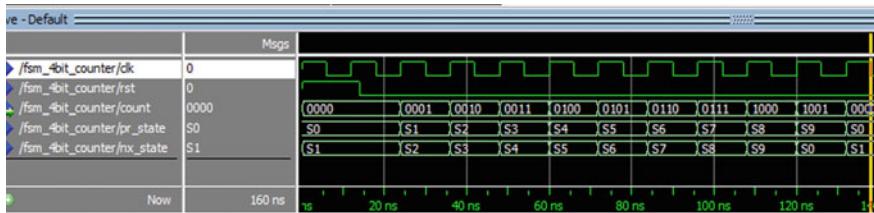


Fig. 5.7 Simulation results for the 4-bit BCD counter

system has ten states and switches from one state to the other at each rising edge of the clock.

5.6.1.2 Implementation and Validation on FPGA Platform

Figure 5.8 shows the validation of the design of 4-bit BCD counter on the LEDs and on the 7-segment displays of the FPGA board.

5.6.2 Sequence Detector FSM

In this example, we develop a sequence detector for bit sequence “1011” by using the concept of the FSM state machine. At each clock cycle, a value will be sampled, if the “1011” sequence is detected, a “1” will be produced at the output for 1 clock cycle. There are two methods to design state machines, first is Mealy and second is Moore style. In this subsection, we implement the sequence detector with both styles.

5.6.2.1 Mealy Machine Style

Figure 5.9 shows the Mealy state diagram for detecting a sequence of the “1011”.

As shown in Fig. 5.9:

- When the machine receives input ‘1’ and the system is in the initial state (s0), it goes to the next state with the output equal to ‘0’. Otherwise, i.e., if the entry is ‘0’, the system remains in the same state.
- When it is in the second state (s1) and the machine receives an input of ‘0’, it goes to the third state (s2) with the output equal to ‘0’. If it receives an entry of ‘1’, it remains in the same state.
- When it is in the third state (s2), the machine receives an input of ‘1’, the detector goes to the fourth state (s3) with the output equal to ‘0’. If the received input is ‘0’, it returns to the initial state.

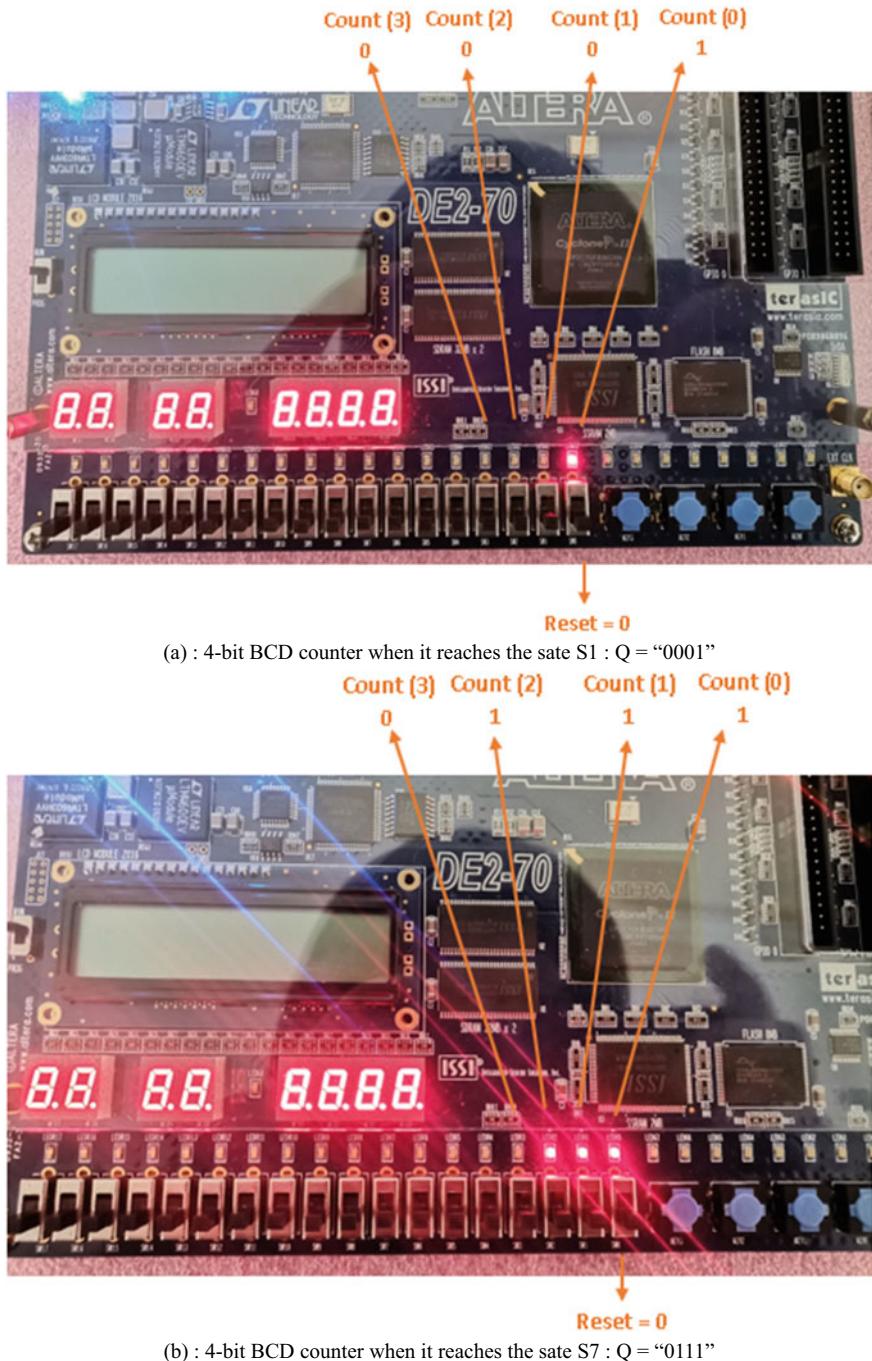
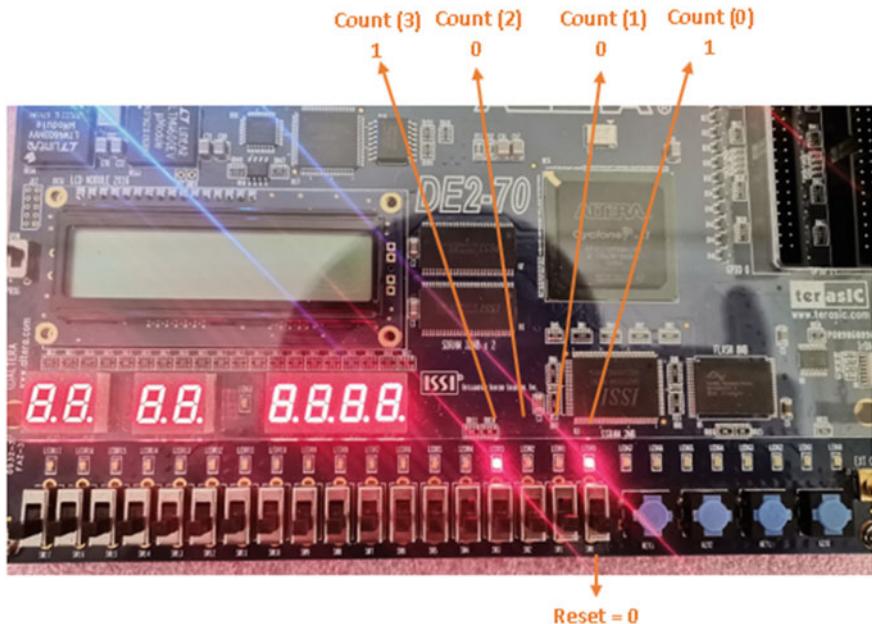
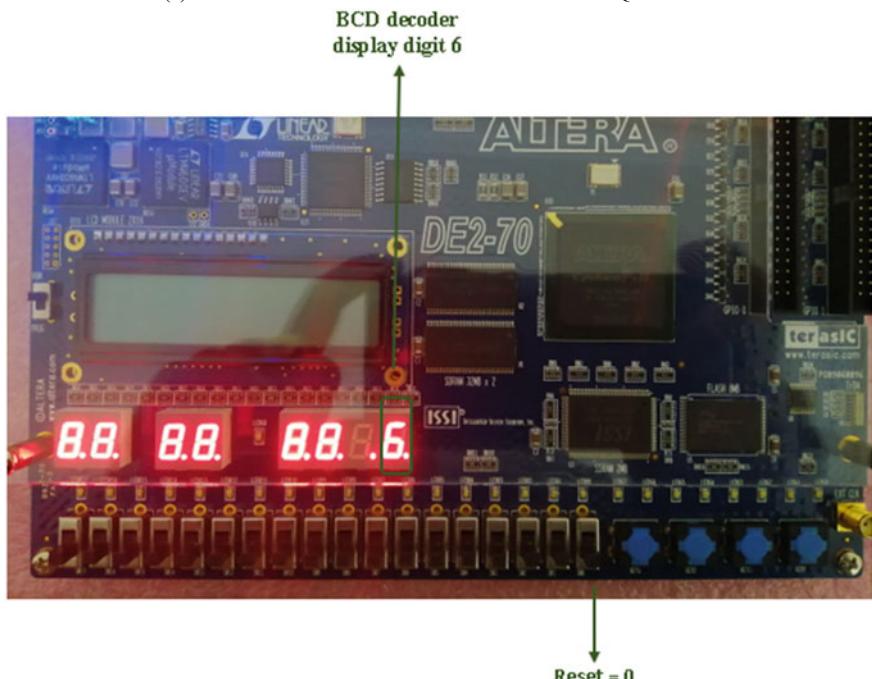


Fig. 5.8 Validation of the design of 4-bit BCD counter on the LEDs and on the 7-segment displays of the FPGA board (a-d)



(c) : 4-bit BCD counter when it reaches the state S9 : Q = "1001"



(d) : 4-bit BCD counter when it reaches the state S6 : Q = "0110", 7-segment display

Fig. 5.8 (continued)

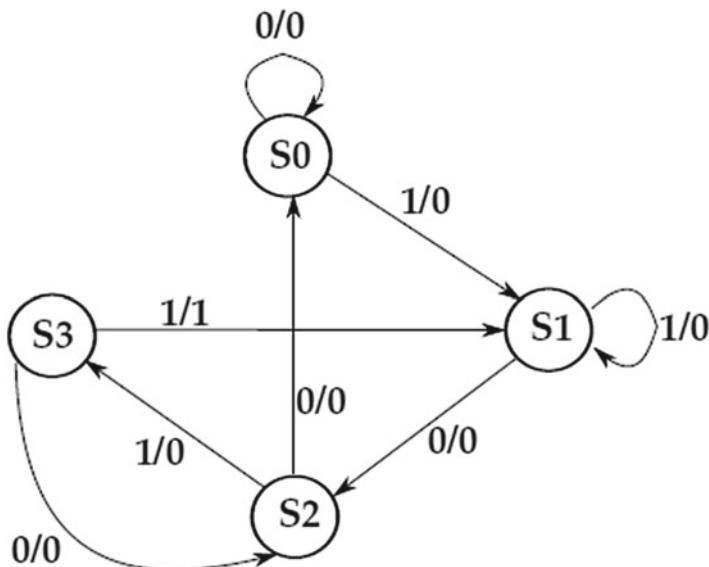


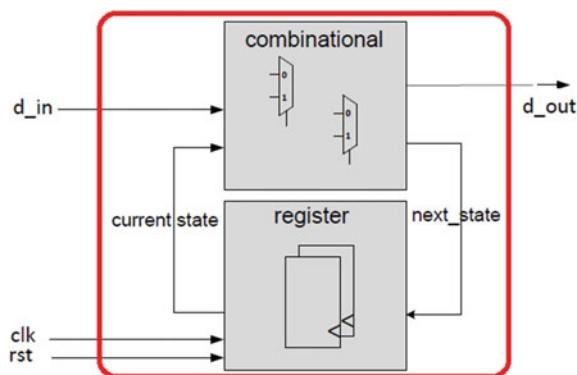
Fig. 5.9 Mealy state machine for detecting a sequence of “1011”

Finally, when it is in the fourth state (s_3) and the machine receives an input of ‘1’, the detector returns to the second state (s_1), with the output equal to ‘1’. If the input receive is ‘0’ it goes back to the third state (s_2).

Figure 5.10 shows the entity for the sequence detector. It has d_{in} , clk and rst as inputs and d_{out} as output.

We use the two-process approach to implement the Mealy machine for the sequence detector. One process is used for the “state_register” block, its role is to update the current state of the FSM. The other process decides the next state of the FSM based on the current state and the input and drives the output based on the state

Fig. 5.10 Entity for the sequence detector



(and the input for the Mealy implementation). The VHDL code for the two-process approach to implement the Mealy machine is shown in Listing 5.10.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity sequence_detector_Mealy is
5  port (
6    clk : in std_logic;
7    rst : in std_logic;
8    d_in : in std_logic;
9    d_out : out std_logic);
10 end sequence_detector_Mealy;
11
12 architecture beh of sequence_detector_Mealy is
13
14  -- Define a enumeration type for the states
15  type state_type is (s0, s1, s2, s3);
16  -- Define the needed internal signals
17  signal current_state, next_state : state_type;
18
19 begin
20
21  reg: process (clk, rst) -- The register process
22  begin -- process register
23  if rst = '1' then -- asynchronous reset (active high)
24  | current_state <= s0;
25  elsif clk'event and clk = '1' then -- rising clock edge
26  | current_state <= next_state;
27  end if;
28  end process;
29
30  comb: process (d_in, current_state) -- The combinational process
31  begin
32  -- set default value
33  d_out <= '0';
34  next_state <= current_state;
35  -- next-state logic
36  case current_state is
37  when s0 =>
38  if d_in = '1' then
39  | next_state <= s1;
40  | d_out <= '0';
41  else
42  | next_state <= s0;
43  | d_out <= '0';
44  end if;
45
46  when s1 =>
47  if d_in = '0' then
48  | next_state <= s2;
49  | d_out <= '0';
50  else
51  | next_state <= s1;
52  | d_out <= '0';
53  end if;
54

```

```

55  when s2 =>
56  if d_in = '1' then
57    next_state <= s3;
58    d_out <= '0';
59  else
60    next_state <= s0;
61    d_out <= '0';
62  end if;
63
64  when s3 =>
65  if d_in = '1' then
66    next_state <= s1;
67    d_out <= '1';
68  else
69    next_state <= s2;
70    d_out <= '0';
71  end if;
72
73  when others => null;
74  end case;
75  end process;
76
77 end beh;
```

Listing 5.10 VHDL code for the implementation of the Mealy machine for the sequence detector

The simulation results of the sequence detector for bit sequence “1011” are shown in Fig. 5.10. We notice that every time the sequence “1011” is detected, a “1” occurs at the output of the system. For example, the data sequence $d_{in} = “01,011,001,011”$ applied to the circuit, the result obtained at the output $d_{out} = “00,001,000,001”$ at the output.

In order to implement the design of the sequence detector on the FPGA platform, we assign the reset signal (rst) and the data sequence d_{in} to the switches SW [0] and SW [1] of the DE2 board, the clock signal (clk) to pin_AD15, and d_{out} output to LEDR [0]. Figure 5.12 shows the validation of the sequence detector on the LEDs of the FPGA board. We notice that at the fourth rising edge of the clock, the sequence “1011” is detected and a “1” appears at the output of the system and the red led lights up. This result is in good agreement with that obtained in simulation, see Fig. 5.11.

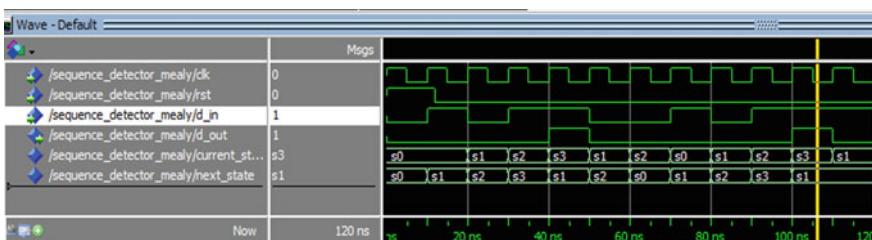


Fig. 5.11 Simulation results of Mealy machine for the sequence detector

At Fourth rising_edge:
Current_state <= S3; next_state <= S1

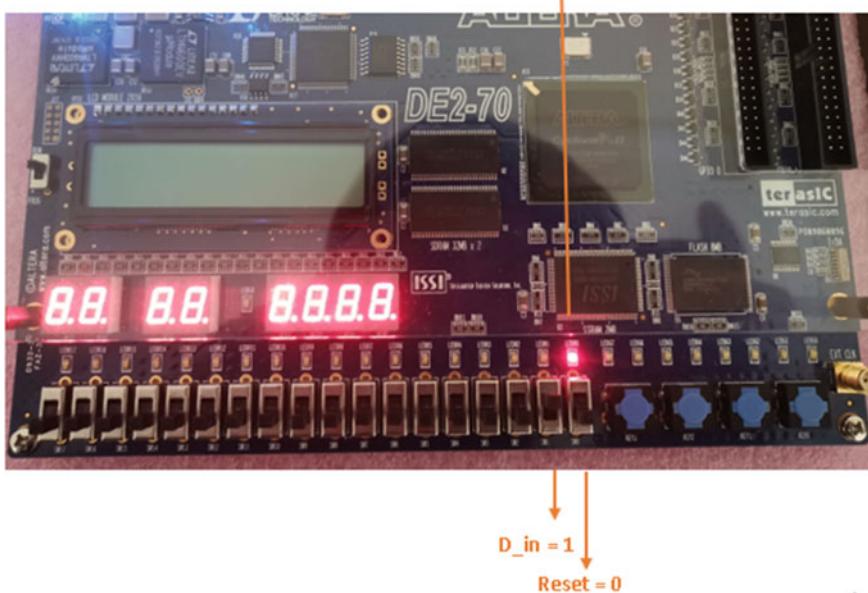


Fig. 5.12 Validation of the sequence detector on the LEDs of the FPGA board

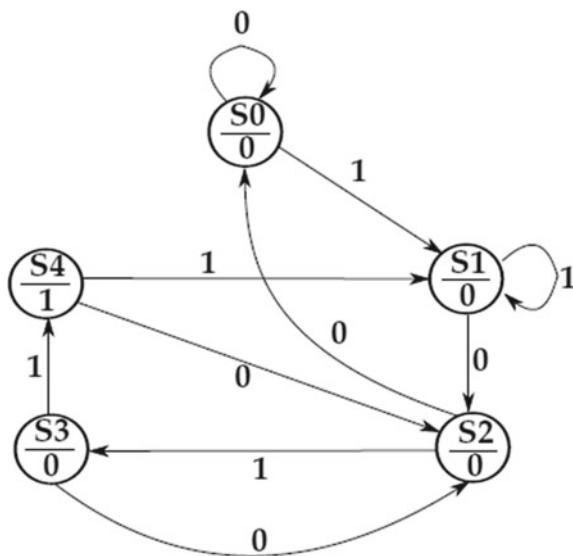
5.6.2.2 Moore Machine Style

Figure 5.13 gives a Moore state diagram for implementing the detecting a sequence of “1011”.

As shown in this figure:

- In the initial state (s_0), the output of the detector is “0”. When the machine receives input “1”, it goes to the next state. If the input is “0”, it remains in the same state.
- In the second state (s_1), the detector output is “0”. When the machine receives an input of “0”, it goes into the third state. If it receives an input of “1”, it remains in the same state.
- In the third state (s_2), the detector output is “0”. When the machine receives an input of “1”, it goes into the fourth state. If the received input is “0”, it returns to the initial state.
- In the fourth state (s_3), the detector output is “0”. When the machine receives an input of “1”, it goes into the fifth state. If the received input is “0”, it reverts to the third state.
- In the fifth state, the output of the detector is “1”. When the machine receives an input of “0”, it goes to the third state, otherwise it goes to the second state.

Fig. 5.13 Moore state diagram for detecting a sequence of “1011”



The VHDL code for the Moore-style implementation of the sequencer is shown in Listing 5.11.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity sequence_detector_Moore is
5  port (
6    clk : in std_logic;
7    rst : in std_logic;
8    d_in : in std_logic;
9    d_out : out std_logic);
10   end sequence_detector_Moore;
11
12 architecture beh of sequence_detector_Moore is
13
14   -- Define a enumeration type for the states
15   type state_type is (s0, s1, s2, s3, s4);
16   -- Define the needed internal signals
17   signal current_state, next_state : state_type;
18
19 begin
20

```

```
21  reg: process (clk, rst) -- The register process
22  | begin -- process register
23  | if rst = '1' then -- asynchronous reset (active high)
24  | | current_state <= s0;
25  | elsif clk'event and clk = '1' then -- rising clock edge
26  | | current_state <= next_state;
27  | end if;
28  | end process;
29 -----
30  comb: process (d_in, current_state) -- The combinational process
31  begin
32  | -- set default value
33  | d_out <= '0';
34  | next_state <= current_state;
35  | -- next-state logic
36  | case current_state is
37  | when s0 =>
38  | | if d_in = '1' then
39  | | | next_state <= s1;
40  | | else
41  | | | next_state <= s0;
42  | | end if;
43  | | d_out <= '0';
44  |
45  | when s1 =>
46  | | if d_in = '0' then
47  | | | next_state <= s2;
48  | | else
49  | | | next_state <= s1;
50  | | end if;
51  | | d_out <= '0';
52  |
```

```

53  |      when s2 =>
54  |      if d_in = '1' then
55  |          next_state <= s3;
56  |      else
57  |          next_state <= s0;
58  |      end if;
59  |      d_out <= '0';
60
61  |      when s3 =>
62  |      if d_in = '1' then
63  |          next_state <= s4;
64  |      else
65  |          next_state <= s2;
66  |      end if;
67  |      d_out <= '0';
68
69  |      when s4 =>
70  |      if d_in = '1' then
71  |          next_state <= s1;
72  |      else
73  |          next_state <= s2;
74  |      end if;
75  |      d_out <= '1';
76
77  |      when others => null;
78  |  end case;
79  | end process;
80
81 end beh;

```

Listing 5.11 VHDL code of the Moore-style implementation of the sequencer detector

The simulation results of the Moore style implementation of the sequencer is shown in Fig. 5.14. Similar results are obtained as the ones shown in Fig. 5.11 (Mealy machine approach of the sequencer). For example, the data sequence $d_{in} = "01,011,001,011"$ applied to the circuit, the result obtained at the output $d_{out} = "00,001,000,001"$. In conclusion, the two approaches, Moore or mealy machine, make it possible to implement the sequence detector in VHDL.

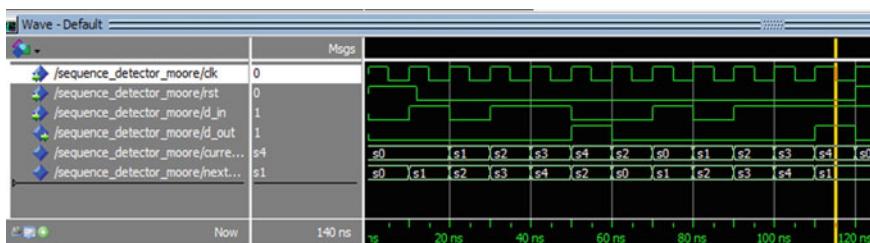


Fig. 5.14 Simulation results of Moore machine for the sequence detector

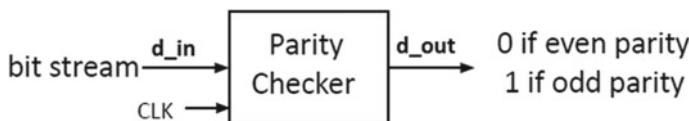


Fig. 5.15 Parity checker

5.6.3 Parity Checker

In this example, we illustrate the importance of using Moore or Mealy machines for parity detection in a real practical case. The parity checker of Fig. 5.15 counts the number of 1's in a bit-serial input stream (d_{in}). If the number of '1's in the input bits (d_{in}) is odd, the parity checker produces '1' on its output (parity) or '0' otherwise.

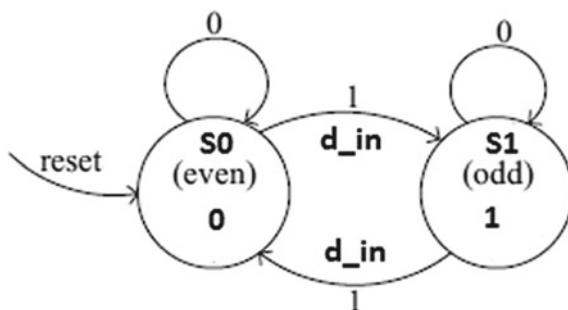
5.6.3.1 Parity Checker Using Moore Machine Style

The state diagram of the odd parity checker is shown in Fig. 5.16. The circuit has two states (S_0 and S_1) and switches from one to the other whenever $d_{in} = '1'$ is received.

- State S_0 : the output is '0' if an even number of 1's have been detected.
- State S_1 : the output is '1' if an odd number of 1's detected.

The odd parity checker for the state diagram shown in Fig. 5.16 can be described by two or three processes Moore FSM. The VHDL code of the two-process Moore approach to implement the parity checker is shown in Listing 5.12. The process (P1) is used to update the value "current_state" while the process (P2) is used to update the "next_state" and assign the outputs of the circuit "parity".

Fig. 5.16 State diagram of odd parity checker for Moore machine



```

1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity parity_checker_Moore is
5   port(clk, rst: in std_logic;
6       d_in: in std_logic;
7       parity: out std_logic);
8   end parity_checker_Moore;
9
10  architecture logic_flow of parity_checker_Moore is
11  type state is (s0, s1);
12  signal current_state, next_state: state;
13
14  begin
15
16  p1: process(clk, rst) --current state update
17  begin
18  if(rst='1') then
19      current_state<=s0; --default state on reset.
20  elsif(rising_edge(clk)) then
21      current_state<=next_state; --state change.
22  end if;
23  end process;
24
25  p2: process (current_state, d_in)
26  begin
27
28  case current_state is
29  when s0 => --when current state is "s0"
30  if(d_in ='0') then
31      next_state<=s0;
32  else
33      next_state<=s1;
34  end if;
35  parity<='0';
36
37  when s1 => --when current state is "s1"
38  if(d_in ='0') then
39      next_state<=s1;
40  else
41      next_state<=s0;
42  end if;
43  parity<='1';
44
45  end case;
46  end process;
47
48  end logic_flow;

```

Listing 5.12 VHDL code for parity checker using two-process Moore machine

Simulation results relative to the VHDL code in Listing 5.12 are shown in Fig. 5.17. We notice that the circuit is working as expected. If the number of ‘1’s in



Fig. 5.17 Simulation results for parity checker using two-process Moore machine

the input bits (d_{in}) is odd, the parity checker produces ‘1’ on its output (parity) or ‘0’ otherwise.

The VHDL code for the parity checker described using the three-process Moore machine is presented in Listing 5.13. The process (P1) is used to update the value “current_state”, while the process (P2) takes care of the update of the “next_state”, and the process (P3) is in charge for the assignment of the output (parity).

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity parity_checker_three_process_Moore is
5  port(clk, rst: in std_logic;
6  d_in: in std_logic;
7  parity: out std_logic);
8  end parity_checker_three_process_Moore;
9
10 architecture logic_flow of parity_checker_three_process_Moore is
11 type state is (s0, s1);
12 signal current_state, next_state: state;
13
14 begin
15
16 p1: process(clk, rst) --current state update
17 begin
18 if(rst='1') then
19 | current_state<=s0; --default state on reset.
20 elsif(rising_edge(clk)) then
21 | current_state<=next_state; --state change.
22 end if;
23 end process;
24
25 p2: process (current_state, d_in)-- --nexte state update
26 begin
27 case current_state is
28

```

```

29  | when s0 => --when current state is "s0"
30  | if(d_in ='0') then
31  |   next_state<=s0;
32  | else
33  |   next_state<=s1;
34  | end if;
35
36  | when s1 => --when current state is "s1"
37  | if(d_in ='0') then
38  |   next_state<=s1;
39  | else
40  |   next_state<=s0;
41  | end if;
42
43  | end case;
44  | end process;
45
46  p3: process (current_state)-- assignment of the output
47  begin
48
49  case current_state is
50  when s0 => --when current state is "s0"
51  parity <= '0';
52
53  when s1 => --when current state is "s1"
54  parity <= '1';
55
56  when others => parity <= '0';
57
58  end case;
59  end process;
60
61
62  end logic_flow;

```

Listing 5.13 VHDL code for parity checker using three-process Moore machine



Fig. 5.18 Simulation results for parity checker using three-process Moore machine

At Fifth rising_edge:
Current_state <= S1, next_state <= S0

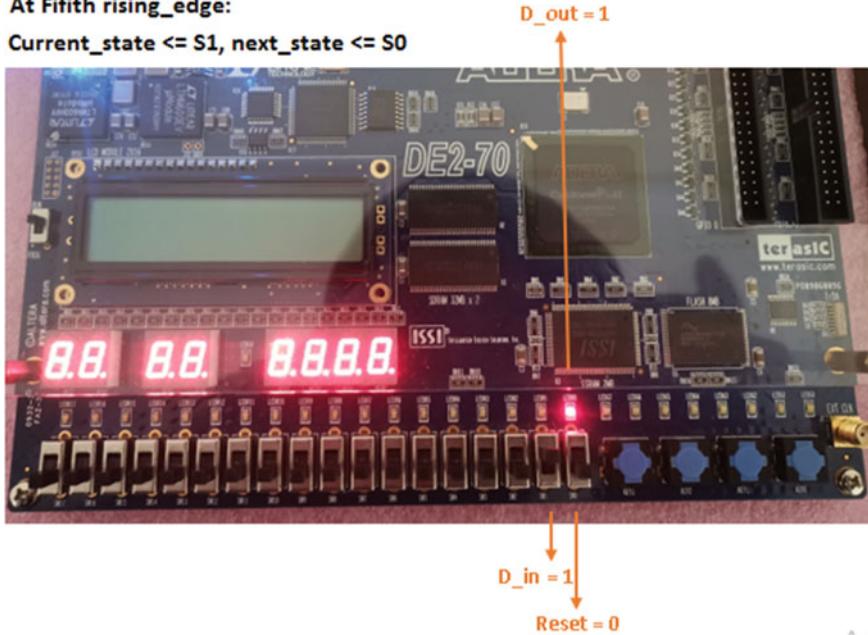


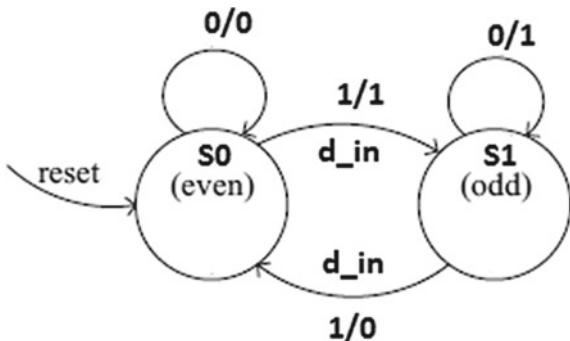
Fig. 5.19 Validation of the parity checker on the LEDs of the FPGA board

Figure 5.18 shows the simulation results for parity checker using three-process Moore machine. These results are identical to those obtained previously using Moore's approach with two processes.

In order to implement the design of the parity checker on the FPGA platform, we assign the reset signal (rst) and the data sequence d_in to the switches SW [0] and SW [1] of the DE2 board, the clock signal (clk) to pin_AD15, and d_out output to LEDR [0]. Figure 5.19 shows the validation of the parity checker on the LEDs of the FPGA board. We observe that at the fifth rising edge of the clock, the parity checker produces “1” on its output specifying that the number of “1” detected in the input bits (d_in) is odd. This result demonstrates the correct functioning of the circuit and it agrees well with that obtained in simulation, see Fig. 5.18.

5.6.3.2 Parity Checker Using Mealy Machine Style

Unlike the Moore machine, the output of the Mealy machine is a function of the current state and data input (d_in). The parity checker can be described by two or three processes of the Mealy machine. For better readability, it is recommended to use the three-process FSM. Listing 5.14 shows VHDL code using a three-process Mealy machine to implement the parity checker. The Mealy machine state diagram for parity checker is shown in Fig. 5.20.

**Fig. 5.20** State diagram of odd parity checker for Mealy machine

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity parity_checker_Mealy is
5  port(clk, rst: in std_logic;
6        d_in: in std_logic;
7        parity: out std_logic);
8  end parity_checker_Mealy;
9
10 architecture logic_flow of parity_checker_Mealy is
11 type state is (s0, s1);
12 signal current_state, next_state: state;
13
14 begin
15
16 p1: process(clk, rst) --current state update
17 begin
18 if(rst='1') then
19   current_state<=s0; --default state on reset.
20 elsif(rising_edge(clk)) then
21   current_state<=next_state; --state change.
22 end if;
23 end process;
24
  
```

```

25  p2: process (current_state, d_in)-- --nexte state update
26  begin
27  case current_state is
28
29  when s0 => --when current state is "s0"
30  if(d_in ='1') then
31  |next_state<=s1;
32  else
33  |next_state<=s0;
34  end if;
35
36  when s1 => --when current state is "s1"
37  if(d_in ='1') then
38  |next_state<=s0;
39  else
40  |next_state<=s1;
41  end if;
42
43  end case;
44  end process;
45
46  -----
47  p3: process (current_state)-- assignment of the output
48  begin
49  case current_state is
50  when s0 => --when current state is "s0"
51  if (d_in = '1') then
52  |parity <= '1';
53  else
54  |parity <= '0';
55  end if;
56
57  when s1 => --when current state is "s1"
58  if (d_in = '1') then
59  |parity <= '0';
60  else
61  |parity <= '1';
62  end if;
63
64  when others => parity <= '0';
65
66  end case;
67  end process;
68
69  end logic_flow;

```

Listing 5.14 VHDL code using a three-process Mealy machine to implement the parity checker

The simulation results shown in Fig. 5.21 demonstrate the correct functioning of the parity checker design using the Mealy machine with three processes.



Fig. 5.21 Simulation results of parity checker using three-process Mealy machine

5.7 Summary

Finite state machines are of particular interest for the design of digital systems. FSM can be grouped into two types Moore (where the outputs depend only on the current state values) and Mealy (where outputs depend on both the present states and external inputs).

This chapter attempts to familiarize the reader with the key concepts of the finite state machine. The key blocks of each state machine (Moore or Mealy) have been explained and described in VHDL. Typically, an FSM is implemented using a register to store the current state and the combination logic for the next and output state functions. To describe these elements in VHDL, we used the FSM approach with three separate processes or two processes.

This chapter ended with the presentation of some examples of state machine applications, namely the 4-bit BCD counter, sequence detector and parity checker.

Part III

Laboratory Projects

Chapter 6

Digital Projects Carried Out on the FPGA Platform



Abstract In this chapter, five laboratory projects have been presented, namely: simple calculator design (Arithmetic Logic Unit), digital clock, traffic light control system, design and implementation of vending machine and control of a 4-phase step motor (Direction and Speed). The operation of the projects has been verified by using ModelSim simulation and subsequently implemented into an FPGA platform.

6.1 Introduction

This part of the book is devoted to the laboratory's projects on the real-time implementation of digital systems in FPGAs, five laboratory projects have been addressed and presented in detail. We start this chapter with the first project related to the design, simulation and implementation on the FPGA Altera DE2_70 platforms of a mini-calculator based on an arithmetic and logic unit (ALU), to perform arithmetic operations on 4-bit numbers. This is followed by the second project devoted to the development of a digital clock using counters. This clock will display the hours (from 0 to 23) on the HEX7-6 7-segment displays, the minutes (from 0 to 59) on HEX5-4 and the seconds (from 0 to 59) on HEX3-2. The third project concerns the development of a traffic light controller for the intersection of three main roads A, B and C. For this, we used the finite state diagram (FSM) of a Mealy machine based on the guidelines provided. The traffic light controller is tested using Quartus development tools and the DE-70 Altera FPGA board. The vending machine controller was discussed and presented in Lab #4. This controller allows customers to access the contents of the vending machine and choose the desired beverage from the many varieties offered. In addition, the customer can purchase a combination of drinks depending on how much they want to spend. The machine returns the balance amount if it provides excess money and also return the change. We close this chapter with the last project dedicated to the development of a stepper motor controller allowing to control the position and the speed of the motor. The VHDL code which provides different operating modes of the motor, namely: step by step mode, maximum torque

or half step. Finally, the design of this controller is implemented and validated on the DE-70 Altera FPGA platform.

6.2 Lab #1 Simple Calculator Design

ALU (Arithmetic logic unit) is a critical component of a microprocessor and is the core component of central processing unit. Furthermore, it is the heart of the instruction execution portion of every computer. ALU's comprise the combinational logic that implements logic operations, such as 'AND' and 'OR', and arithmetic operations, such as 'ADD' and 'SUBTRACT'. The purpose of this laboratory is to design, simulate and implement on the FPGA platform Altera DE2_70, a mini-calculator based on an arithmetic and logic unit (ALU) to perform arithmetic operations on 4-bit numbers which are defined at using switches. The block diagram of the mini-calculator is shown in Fig. 6.1. It consists of an ALU, a 4 registers of 4 bits (R0, R1, R2 and R3), a 4-bit multiplexer with 2 inputs (Mux 1), a 4-bit multiplexer with 4 input (Mux 2), a 4-bit demultiplexer with 4 outputs (Demux) and controller.

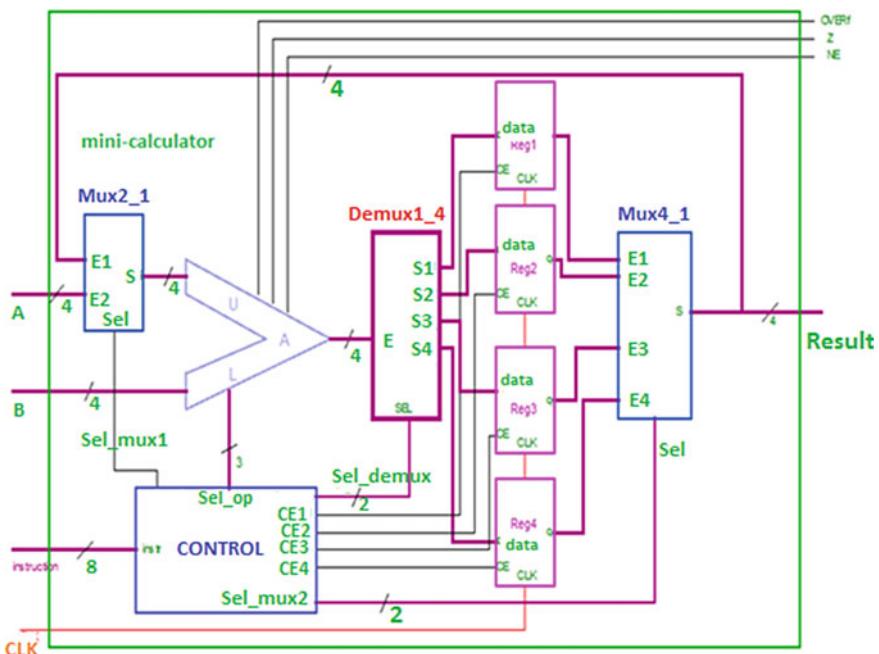


Fig. 6.1 Block diagram of the mini-calculator

6.2.1 Design of Arithmetic and Logic Unit (ALU)

The circuit of Fig. 6.2 (ALU) has as inputs A, B, and OpCode (operation code), and RES as output. It is used to perform several logical and arithmetic operations such as: addition, subtraction, and other logical operations such as AND, NAND, OR, XOR, and complement A. Then, the results of the ALU will be stored in four 4-bit registers (R_0, R_1, R_2, R_3).

The ALU truth table is shown in Table 6.1, where each function is selected by a different opcode value. Note that the upper six instructions are logical, while the lower two instructions are arithmetic.

As can be seen in Fig. 6.2, the ALU has three output signals $A_{_SUB_B}$, $A_{_INF_B}$ and $A_{_EGAL_B}$ coming from the 4-bit comparator, as well as three signals indicating the state of the result (RES Output) obtained, defined by:

- OVFL: indicates an integer overflow of the addition and subtraction functions.
- ZERO: indicate if the result of an operation is null.
- NEG: indicate if the result of an operation is negative.

Fig. 6.2 Basic structure of an Arithmetic and Logic Unit (ALU)

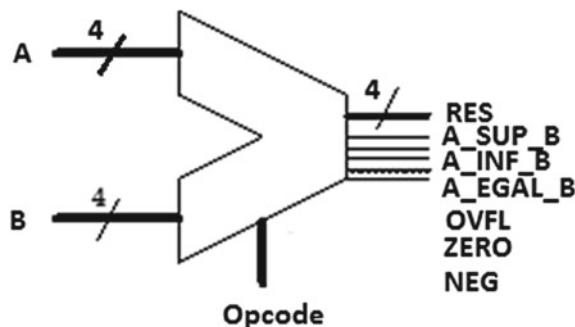


Table 6.1 Truth table of the ALU

Opcode	Instruction	Operation	Unit
000	AND	$RES \Leftarrow A \text{ AND } B$	Logic
001	NAND	$RES \Leftarrow A \text{ NAND } B$	
010	OR	$RES \Leftarrow A \text{ OR } B$	
011	NOR	$RES \Leftarrow A \text{ NOR } B$	
100	XOR	$RES \Leftarrow A \text{ XOR } B$	
101	Complement A	$RES \Leftarrow \text{NOT } A$	
110	Add A and B	$RES \Leftarrow A + B$	Arithmetic
111	Subtract A and B	$RES \Leftarrow A - B$	

The VHDL code of the ALU is presented below (see the Listing 6.1). This code contains three processes, named respectively, “Comp” for the comparison of two 4-bit bits, “Op” to describe the basic operations of the ALU and “Indic” to inform us of the status of the result (RES) got. In the “Op” process (lines 34–48), with the CASE statement, we describe in VHDL the arithmetic and logic operations of the ALU. This piece of VHDL contains two main sections, called logic (Lines 38–42) and arithmetic units (Lines 44–48), each controlled by the same three LSBs of OpCode.

```

1  Library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use ieee.std_logic_unsigned.all;
5
6  entity ALU_minicalculator is
7  port (A, B: in unsigned(3 downto 0);
8    OpCode: in std_logic_vector (2 downto 0);
9    RES: out unsigned(3 downto 0);
10   A_SUB_B, A_INF_B, A_EGAL_B, OVFL, NEG, ZERO: out boolean);
11 end ALU_minicalculator;
12
13 architecture behavioral of ALU_minicalculator is
14 signal S: unsigned(3 downto 0);
15 begin
16
17 Comp: Process(A, B)
18 begin
19 if A < B then
20   A_SUB_B <= false;
21   A_INF_B <= true;
22   A_EGAL_B <= false;
23 elsif A > B then
24   A_SUB_B <= true;
25   A_INF_B <= false;
26   A_EGAL_B <= false;
27 elsif A = B then
28   A_SUB_B <= false;
29   A_INF_B <= false;
30   A_EGAL_B <= true;
31 end if;
32 end process;
33

```

```

34  Op: process(A, B, opCode)
35  begin
36  case OpCode is
37    ----- Logic unit -----
38    when "000" => S <= A and B;
39    when "001" => S <= A nand B;
40    when "010" => S <= A or B;
41    when "100" => S <= A xor B;
42    when "101" => S <= not A;
43    ----- Arithmetic unit -----
44    when "110" => S <= A + B;
45    when "111" => S <= A - B;
46    when others => S <= "XXXX";
47  end case;
48  end process;
49

```

```

50  indic: process(S)
51  begin
52  if S > "1111" then
53    OVFL <= true;
54  else
55    OVFL <= false;
56  end if;
57  if S < 0 then
58    NEG <= true;
59  else
60    NEG <= false;
61  end if;
62  if S = 0 then
63    ZERO <= true;
64  else
65    ZERO <= false;
66  end if;
67  RES <= S;
68  end process;
69  end behavioral;

```

Listing 6.1 VHDL code of the arithmetic and logic unit (ALU)

The simulation results, shown in Fig. 6.3, demonstrate the correct operation of the mini-calculator. Depending on the selected code “OpCode”, the ALU performs logical and arithmetic functions correctly.

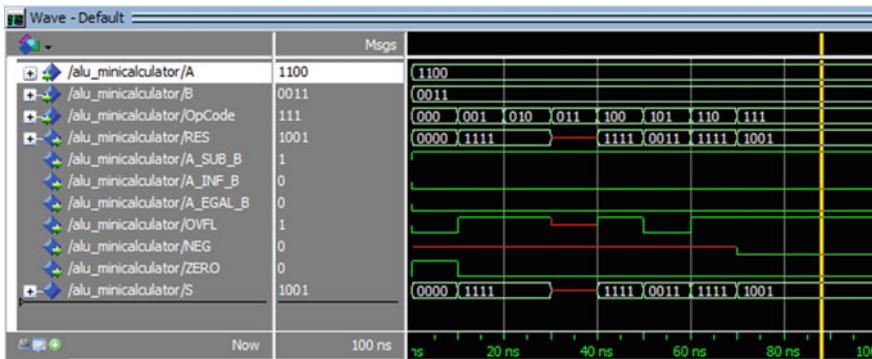


Fig. 6.3 Simulation results of the ALU

6.2.2 Design of Multiplexers and Demultiplexers

According to the architecture of the mini-calculator presented in the Fig. 6.4a, two multiplexers are essential: 2-to-1 Multiplexer (Mux2_1) and 4-to-1 Multiplexer (Mux4_1). The first multiplexer (Mux2_1) is located before the first input of the UAL and which is intended to select either the operand A or a data item stored in one of the registers according to the select input (Sel). The symbol and the truth table of 2-to-1 multiplexer are depicted in the Fig. 6.4b.

Using When-Else statement, a VHDL code for this circuit is given below (Listing 6.2).

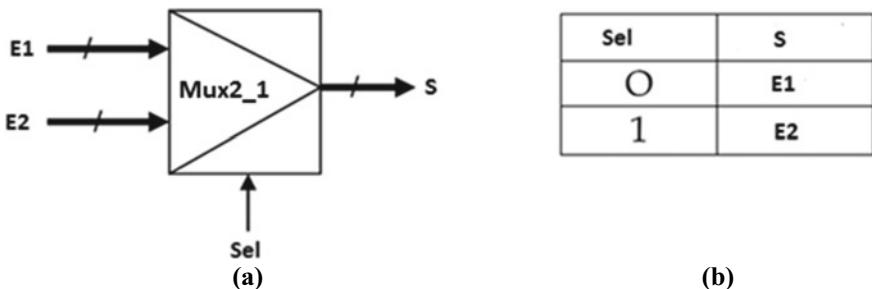


Fig. 6.4 2-to-1 multiplexer: **a** Symbol; **b** Truth table

```

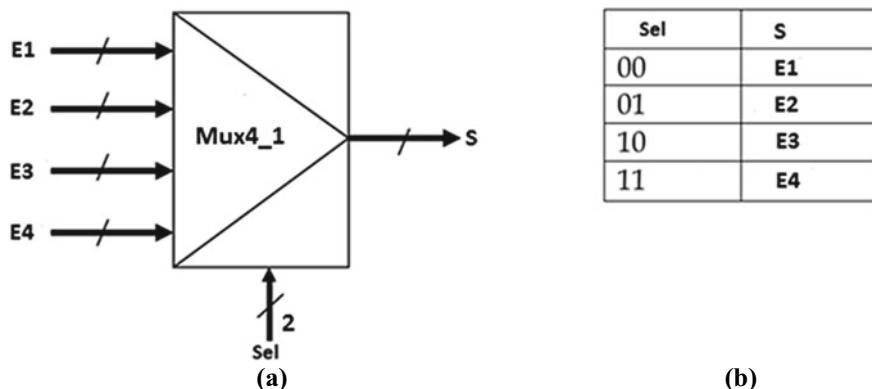
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3
4 ENTITY Mux2_1_minicalculator IS
5 PORT (
6 E1,E2: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
7 sel: IN STD_LOGIC;
8 S: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
9 END Mux2_1_minicalculator;
10
11 ARCHITECTURE RTL OF Mux2_1_minicalculator IS
12 BEGIN
13 PROCESS(E1, E2, sel)
14 BEGIN
15 IF sel = '1' THEN
16 S<= E1;
17 ELSE
18 S<= E2;
19 END IF;
20 END PROCESS;
21
22 END RTL;

```

Listing 6.2 VHDL code of the 2-to-1 multiplexer

The second multiplexer (Mux4_1) is located after the outputs of the four 4-bit registers (E1, E2, E3 and E4). Its role is to choose the source register where the data are stored as a function of bits 6 and 7 of the instruction. The symbol and the truth table of 4-to-1 multiplexer are depicted in the Fig. 6.5.

A VHDL code for 4-to-1 multiplexer using a Case statement (lines 15–20) is shown below in Listing 6.3.

**Fig. 6.5** 4-to-1 multiplexer: **a** Symbol; **b** Truth table

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  ENTITY Mux4_1_minicalculator is
5
6  PORT (
7    E1,E2,E3,E4: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
8    sel:IN STD_LOGIC_VECTOR(1 DOWNTO 0);
9    S: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
10   END Mux4_1_minicalculator;
11
12  ARCHITECTURE behv OF Mux4_1_minicalculator IS
13  BEGIN
14    BEGIN
15      PROCESS (sel)
16      BEGIN
17        CASE sel IS
18          WHEN "00" => S<= E1;
19          WHEN "01" => S<= E2;
20          WHEN "10" => S<= E3;
21          WHEN OTHERS => S<= E4;
22        END CASE;
23      END PROCESS;
24
25  END behv;

```

Listing 6.3 VHDL code of the 4-to-1 multiplexer

The task of the demultiplexer is to choose one of the destination registers according to the command of the controller. Its truth table and its symbol are represented in Fig. 6.6. To describe in VHDL the behavior of multiplexer 1-to-4, we adopted behavioral modeling using the sequential CASE statement as shown in the Listing 6.4.

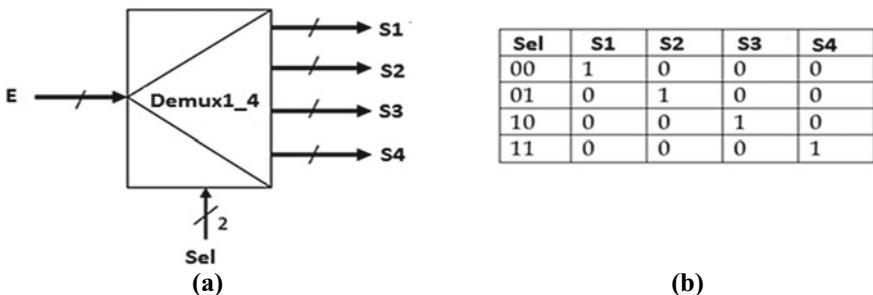


Fig. 6.6 1-to-4 demultiplexer: **a** Symbol; **b** Truth table

```

1 LIBRARY ieee;
2 USE ieee.std_logic_1164.ALL;
3
4 ENTITY Demux1_4_minicalculator IS
5 PORT(
6   E : IN std_logic_vector (3 DOWNTO 0);
7   Sel : IN std_logic_VECTOR (1 DOWNTO 0);
8   S1 , S2 , S3 ,S4 : OUT std_logic_VECTOR (3 DOWNTO 0));
9 END Demux1_4_minicalculator;
10
11 ARCHITECTURE one OF Demux1_4_minicalculator IS
12 BEGIN
13
14 PROCESS(E, Sel)
15 BEGIN
16 CASE Sel IS
17   when "00" => S1<= E; S2<= "0000"; S3<= "0000"; S4<= "0000";
18   when "01" => S2<= E; S1<= "0000"; S3<= "0000"; S4<= "0000";
19   when "10" => S3<= E; S1<= "0000"; S2<= "0000"; S4<= "0000";
20   when others => S4<= E; S1<= "0000"; S2<= "0000"; S3<= "0000";
21 END CASE;
22 END PROCESS;
23 END one;

```

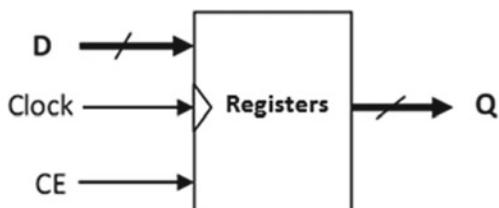
Listing 6.4 VHDL code of the 1-to-4 demultiplexer

6.2.3 Design of Registers

Figure 6.7 shows the formed register based on a set of D flip-flops. This register accepts three inputs D (data: of 4-bits), Clock (clk), CE and one output Q of 4-bits. If CE = 1, the output of the register will be a copy of the D input, otherwise the output is a highly unknown state.

With the If-else statement, the VHDL code for this register is given in the Listing 6.5.

Fig. 6.7 Registers of the ALU



```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Registre_minicalculator is
5  port(
6    clk,CE  : in std_logic;
7    D :  in std_logic_vector(0 downto 3);
8    Q : out std_logic_vector(0 downto 3));
9  end Registre_minicalculator;
10
11 architecture behv of Registre_minicalculator is
12 begin
13
14 process(clk,CE)
15 begin
16 if CE = '0' then
17   Q <= "0000"
18 elsif CE = '1' and clk'event and clk='1' then
19   Q <= D;
20 end if;
21 end process;
22
23 end behv;

```

Listing 6.5 VHDL code of the register

6.2.4 Design of Controller

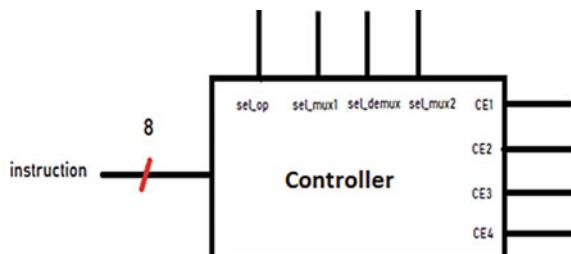
While the UAL is doing its task, the controller sends control signals that allow data to flow without blocking the operation, based on the data read from the instruction.

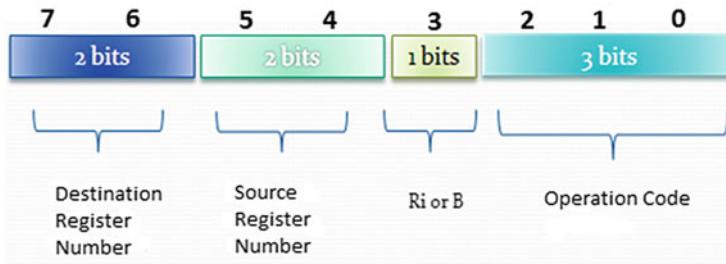
Figure 6.8 shows the block diagram of the controller for the mini-calculator.

The instructions of the mini-calculator consist of a string of numbers which are stored as binary data words (8-bit), as shown in Fig. 6.9.

The operation code or OpCode is coded on 3 bits and is used to defines the type of operation in the mini-calculator has to perform when the instruction is executed. Bit 3 of the instruction set is used to specify whether you want to perform an operation on inputs A and B, or on B and data from the register whose address is specified

Fig. 6.8 Block diagram of the controller of the mini-calculator



**Fig. 6.9** Instruction format of the mini-calculator

in the Source register number field. Bit 5 and Bit 4 are used to select the register in which data will be used to perform a calculation with the operand B => This is the multiplexer selection signal 4 to 1. Bits 7 and 6 are used to select the register in which the UAL result will be stored => this is the selection signal of the 1 to 4 demultiplexer. The VHDL code for the controller is indicated in the Listing 6.6.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity controller_minicalculator is
5  port(
6    instr : in std_logic_vector(7 downto 0);
7    Sel_op: out std_logic_vector(2 downto 0);
8    Sel_mux1:out std_logic;
9    sel_demux, sel_mux2: out std_logic_vector (1 downto 0);
10   -CE1,CE2,CE3,CE4:out std_logic);
11  end controller_minicalculator;
12
13  Architecture simple of controller_minicalculator is
14
15  signal S :std_logic_vector(1 downto 0);
16
17  begin
18    Sel_op <= instr(2) & instr(1) & instr(0) ;
19    sel_mux1 <= instr(3);
20    sel_mux2 <= instr(5) & instr(4);
21    sel_demux <= instr(7) & instr(6);
22

```

```

23  |  Process (s)
24  |  |
25  |  begin
26  |  |
27  |  case
28  |  |  when s is
29  |  |  |  when "00" => CE1<= '1'; CE2<= '0', CE3<= '0'; CE4<= '0';
30  |  |  |  when "01" => CE1<= '0'; CE2<= '1', CE3<= '0'; CE4<= '0';
31  |  |  |  when "10" => CE1<= '0', CE2<= '0', CE3<= '1'; CE4<= '0';
32  |  |  |  when others => CE1<= '0'; CE2<= '0', CE3<= '0'; CE4<= '1';
33  |  |  end case;
34  |  end process;
35  |  end simple;

```

Listing 6.6 VHDL code controller

6.2.5 Implementation of the Final Design of the Mini-calculator on the FPGA Platform

To implement the final design of the mini-calculator in the FPGA platform, we present here the method which is based on the instantiation of the components using the structural description. This approach is based on the following points:

- In the declarative part of the architecture, we declare the different components forming the mini-calculator, namely, 2-to-1 Multiplexer, ALU, 1-to-4 Demultiplexer, controller, Register and 4-to-1 Multiplexer;
- Declaration of component interconnection signals;
- Instantiating the different components forming the mini-calculator using the Port mapping command.

Using this method, The VHDL program for mini-calculator is provided below (Listing 6.7).

```

1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.numeric_std.all;
4  USE ieee.std_logic_unsigned.ALL;
5
6  entity minicalculator is port(
7  |  A,B : in std_logic_vector(3 downto 0);
8  |  clk : in std_logic;
9  |  instruction : in std_logic_vector(7 downto 0);
10 |  Result : buffer std_logic_vector(3 downto 0);
11 |  S_A_SUP_B,S_A_INF_B,S_A_EGAL_B,S_NEG,S_ZERO,S_OVFL: out std_logic);
12 end minicalculator;
13

```

```

14  --Architecture final of minicalculator is
15
16  ----- Interconnection signals -----
17  Signal v1,v2,v3,v4,v5,v6,v7,v8,v9,v10: Std_logic_vector (3 downto 0);
18  Signal C1:std_logic_vector(2 downto 0);
19  Signal C2:std_logic;
20  Signal C3,C4:std_logic_vector(1 downto 0);
21  Signal C5, C6, C7, C8:std_logic;
22
23  ----Multiplexer2_1
24  component mux2_1_minicalculator
25  port(
26    E1, E2: in std_logic_vector (3 downto 0);
27    Sel: in std_logic;
28    S: OUT std_logic_vector (3 downto 0));
29  end component;
30
31  ----- ALU
32  component ALU_minicalculator
33  port(
34    A, B: in std_logic_vector (3 downto 0);
35    OpCode: in std_logic_vector (2 downto 0);
36    Res : out std_logic_vector(3 downto 0);
37    S_A_SUP_B,S_A_INF_B,S_A_EGAL_B,S_NEG,S_ZERO,S_OVFL: out std_logic);
38  end component;
39

```

```

40  -- Demultiplexer--
41  component demux1_4_minicalculator
42  port(
43    E:in std_logic_VECTOR (3 DOWNTO 0);
44    Sel : in std_logic_VECTOR (1 DOWNTO 0);
45    S1, S2, S3, S4 : out std_logic_VECTOR (3 DOWNTO 0));
46  end component;
47
48  --Controller--
49  component Controller_minicalculator
50  port(
51
52    instr: in std_logic_vector (7 downto 0);
53    Sel_op: out std_logic_vector(2 downto 0);
54    Sel_mux1: out std_logic;
55    Sel_demux, sel_mux2: out std_logic_vector (1 downto 0);
56    CE1, CE2, CE3, CE4 : out std_logic);
57  end component;
58
59  --Registre--
60  component register_minicalculator
61  port(
62    clk, CE: in std_logic;
63    D :in std_logic_vector(3 downto 0);
64    Q: out std_logic_vector(3 downto 0));
65  end component;
66

```

```

67  |--Multiplexer4_1--
68  |component mux4_1_minicalculator
69  |port(
70  |  E1,E2,E3,E4 : in std_logic_vector(3 downto 0);
71  |  Sel: in std_logic_vector(1 downto 0);
72  |  S : OUT std_logic_vector(3 downto 0));
73  |END component;
74  |
75  begin
76
77  |U0:Controller_minicalculator port map (
78  |  instruction, C1, C2, C3, C4, C5, C6, C7, C8) ;
79  |  U1:mux2_1_minicalculator port map (A, Result, C2,v1) ;
80  |  U2:ALU_minicalculator port map (
81  |    v1, B, C1, v2, S_A_SUP_B, S_A_INF_B, S_A_EGAL_B, S_OVFL, S_NEG, S_ZERO) ;
82  |  U3:demux1_4_minicalculator port map (v2, C3, v3, v4, v5, v6) ;
83  |  U4:register_minicalculator port map (Clk, C5, v3, v7) ;
84  |  U5:register_minicalculator port map (Clk, C6, v4, v8) ;
85  |  U6:register_minicalculator port map (Clk, C7, v5, v7) ;
86  |  U7:register_minicalculator port map (Clk, C8, v6, v10) ;
87  |  U8:mux4_1_minicalculator port map (v7, v8, v9, v10, c4, Result) ;
88
89  end final;

```

Listing 6.7 VHDL code of the mini-calculator

Figure 6.10 shows the implementation of the final design of the mini-calculator on the FPGA platform. We consider the following example: A = (1100), B = (1011) and the instruction is “10100001”, the operation performed by the mini-calculator is: A nand B = 0111 as shown in Fig. 6.10a. If the instruction is “10100010”, the operation performed by the mini-calculator is: A OR B = “1111”, as shown in Fig. 6.10b. By changing the instruction code, we can illustrate all the operations of the mini-calculator. Figure 6.10c and d represent the operations A XOR B and A add B, respectively.

6.3 Lab #2 Digital Clock

In this lab we try to design and implement on the FPGA platform using the concept of counters. This clock will display the hours (from 0 to 23) on the HEX7-6 7-segment displays, the minutes (from 0 to 59) on HEX5-4 and the seconds (from 0 to 59) on HEX3-2. Figure 6.11 illustrates the general diagram of the digital clock. It accepts the clock signal and the reset signal as inputs and produces six outputs s0x, s1x, m0x, m1x, h0x and h1x with:

- s0, s1: to count the units and tens of SECONDS.
- m0, m1: to count the units and tens of MINUTES.
- h0, h1: to count units and tens of HOURS.
- The reset input is common for all modules of the system and its role is to reset all counters.
- If Z1 is active and its counter reaches the value “1001”, i.e. From 9 in decimal, the units of seconds (Z2) are automatically incremented.

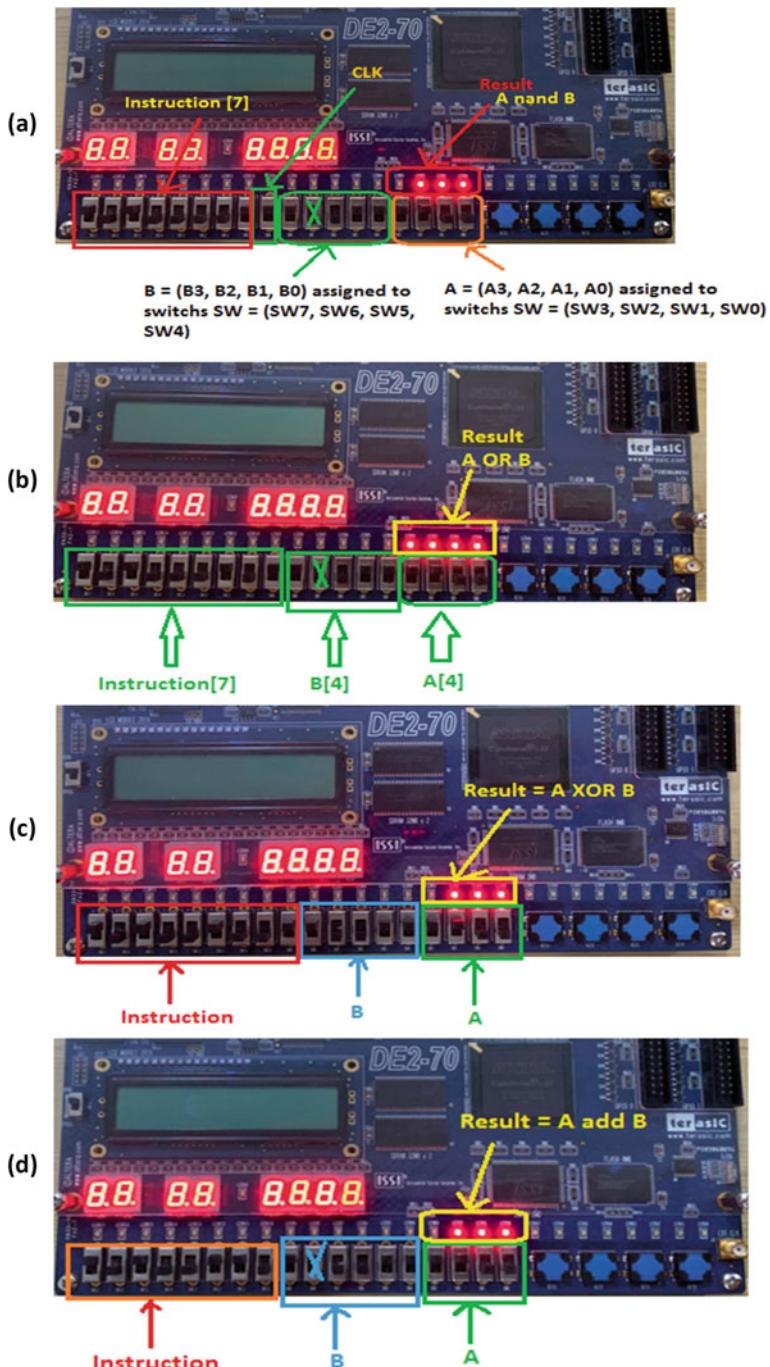


Fig. 6.10 Implementation of the final design of the mini-calculator on the FPGA platform. **a** A = (1100), B = (1011), instruction code = 10100001, operation performed: A nand B. **b** A = (1100), B = (1011), Instruction code = 10100010, operation performed: A OR B. **c** A = (1100), B = (1011), Instruction code = 10100100, operation performed: A XOR B. **d** A = (1100), B = (1011), Instruction code = 10100111, operation performed: A add B

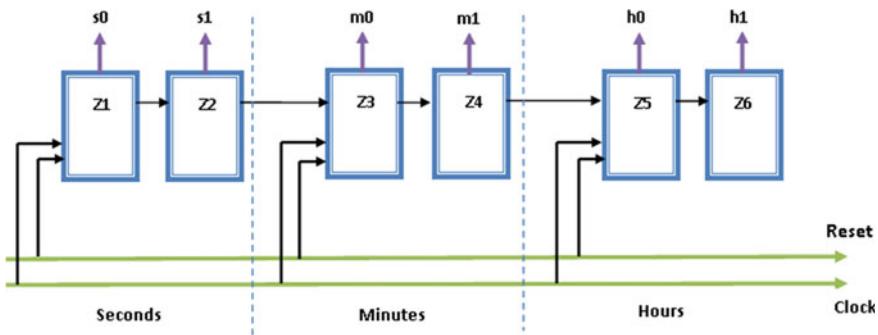


Fig. 6.11 General diagram of the digital clock

- If Z2 is active and its counter reaches the value “0110”, the units of the minutes are automatically incremented up to the value “1001”.
- Same principle for Z 3, Z4, Z5 and Z6.
- Note that the tens of hours have only three values “00”, “01”, “10”.

In addition, we need to add a function named “zero”, in order to access to the counters. When enabled, all counters will be reset to the value “00”. This function will make our digital clock works like a timer.

6.3.1 Design of Digital Clock

Figure 6.12 shows the block diagram of the digital clock. It consists of three blocks such as the frequency divider, counters and 7 segment.

6.3.2 Design of Frequency Divider

The objective of this frequency divider block (see Fig. 6.13) is to obtain a slow clock signal with a period of 1 s from the 50 MHz clock signal available on the DE2-70 Altera FPGA board.

Listing 6.8 shows the VHDL code for the frequency divider. This program makes it possible to obtain a clock signal of 1 s period from a 50 MHz clock available on the FPGA board. So to achieve this, we modify the signal generated by the FPGA board to a generic $N = 25,000,000$ in order to obtain the desired clock signal of **1 Hz** frequency. At each rising edge of the clock signal, a signal D changes state, and the series of changes of state of the variable D constitutes the output signal of 1 Hz frequency.

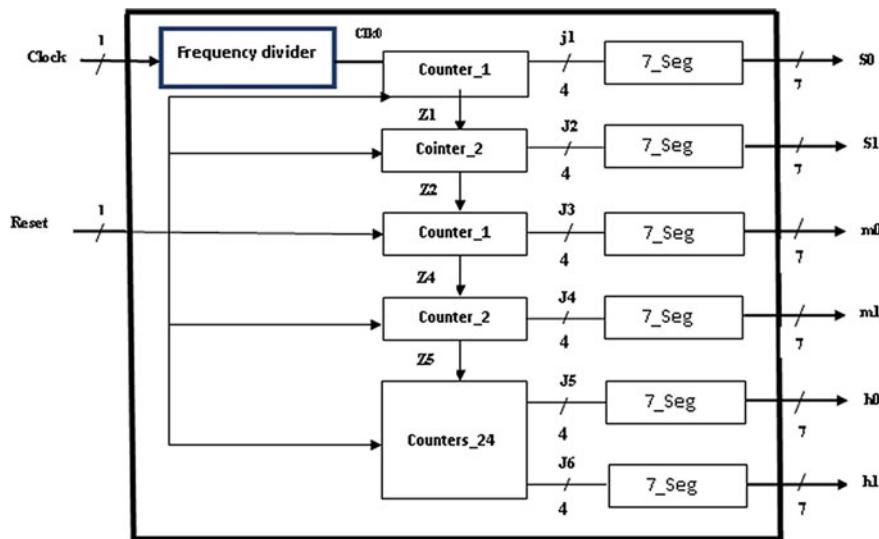


Fig. 6.12 Block diagram of the digital clock



Fig. 6.13 Frequency divider

```

entity freq_div is
  generic
    ( N : natural := 25000000 );
  port
    ( clk : in std_logic;
      reset : in std_logic;
      q : out std_logic );
end entity;
architecture rtl of freq_div is
begin
  process (clk)
    variable cnt : integer range 0 to N;
    variable D :std_logic;
  begin
    if (rising_edge(clk)) then
      if reset = '0' then
        cnt := 0;
        D:='0';
      elsif cnt=N then
        D:=not D;
        cnt:=0;
      else
        cnt := cnt + 1;
      end if;
    end if;
    q <= D;
  end process;
end rtl;

```

N global variable used for a frequency divider

cnt local variable which increments up to N

The variable D changes state each time cnt reaches the value

Output D is a square wave with a period of 1 second

Listing 6.8 VHDL code for frequency divider

6.3.3 Design of Counters

Three type of counters should be designed:

- The “counter_1” for the seconds and minutes units.
- The second counter “counter_2” for the tens of seconds and minutes.
- The last counter will take care of the units and tens of hours, block “counters_24”.

6.3.3.1 Design of counter_1

The VHDL code of the first counter (counter_1) is presented in the Listing 6.9. This program allows each rising edge of the clock signal coming from the frequency divider, to increment a variable “cnt”, and when this variable reaches the value “1001”, the output “clk12” goes high for activate the 2nd tens counter. The change of state of the variable “t” is transferred directly to the output “q1”.

```

entity counter_1 is
  port
    ( clk1      : in std_logic;
      reset1    : in std_logic;
      clk12     : out std_logic;
      q1        : out std_logic_vector(3 downto 0));
end entity;
architecture rtl of counter_1 is
begin
  process (clk1)
    variable cnt : std_logic_vector(3 downto 0);
    variable t   : std_logic;
  begin
    if (rising_edge(clk1)) then
      if reset1 = '0' then
        cnt := "0000";
        t:='0';
      elsif cnt="1001" then
        cnt:="0000";
        t:='1';
      else
        cnt := cnt + 1;
        t:='0';
      end if;
    end if;
    q1 <= cnt;
    clk12<=t;
  end process;
end rtl;

```

The variable t changes to 1 each time cnt reaches the value 9

The cnt variable is incremented up to 9 and then it is initialized.

Q1 contains BCD code
clk12 square signal lasts tens of seconds

Listing 6.9 VHDL code for counter_1

6.3.3.2 Design of counter_2

The VHDL program for the second counter (counter_2) is detailed in Listing 6.10. This second counter works in the same way as the first counter, except that the “cnt” variable is incremented on each rising edge of the “clk12” variable from the 1st counter. The variable “cnt” increments up to the value “0101” then goes back to zero, and the variable “t” goes high thus forming the clock signal for the MINUTES block.

```

entity counter_2 is
  port
    ( clk2      : in std_logic;
      reset2   : in std_logic;
      clk22     : out std_logic;
      q2        : out std_logic_vector(3 downto 0));
end entity;
architecture rtl of counter_2 is
begin
  process (clk2)
    variable cnt: std_logic_vector(3 downto 0);
    variable t :std_logic;
  begin
    if (rising_edge(clk2)) then
      if reset2 = '0' then
        cnt := "0000";
        t:='0';
      elsif cnt="0101" then
        cnt:="0000";
        t:='1';
      else
        cnt := cnt + 1;
        t:='0';
      end if;
    end if;
    q2 <= cnt;
    clk22<=t;
  end process;
end rtl;

```

The variable t changes to 1 each time cnt reaches the value 5

The cnt variable is incremented up to the value 5 then it is initialized

Q1 contains BCD code
clk12 square signal attacks
units of minutes

Listing 6.10 VHDL code for counter_2

6.3.3.3 Design of counter_24

The third block “counters24”, allows to count through variables (cnt2, cnt1) up to (2, 3) then go back to (0, 0). It therefore counts up to the value 23 h corresponding to a day and then it goes back to zeros. To do this we use the following portion of the program:

```

elsif((cnt2="0000" or cnt2="0001")and(cnt1="1001"))then

  cnt1:="0000";
  cnt2 := cnt2 + 1;

  elsif ((cnt2="0010") and(cnt1="0011")) then
  cnt1:="0000";
  cnt2:="0000";

```

The VHDL program of the third counter 24 is shown in Listing 6.11.

```

entity counter24  is
  port
    ( clk3      : in std_logic;
      reset3    : in std_logic;
      q3        : out std_logic_vector(3 downto 0);
      q4        : out std_logic_vector(3 downto 0));
end entity;
architecture rtl of counter24  is
begin
  process (clk3)
    variable cnt1: std_logic_vector(3 downto 0);
    variable cnt2: std_logic_vector(3 downto 0);
  begin
    if (rising_edge(clk3)) then
      if reset3 = '0' then
        cnt1 := "0000"; cnt2 := "0000";
      elsif ((cnt2="0000" or cnt2="0001") and (cnt1="1001")) then
        cnt1:="0000";
        cnt2 := cnt2 + 1;           → cnt2 increments up to the value 2
      elsif ((cnt2="0010" and(cnt1="0011")) then          each time cnt1 reaches the value
        cnt1:="0000"; cnt2:="0000";                      9 and it is initialized when cnt2 = 2
      else
        cnt1 := cnt1 + 1;           → cnt increments to the value 9
        end if;
      end if;
      q3<= cnt1;
      q4<= cnt2;
    end process;
  end rtl;

```

Q3 and Q4 receive the BCD codes to be transmitted to the Decoder block

cnt2 increments up to the value 2
each time cnt1 reaches the value 9 and it is initialized when cnt2 = 2 and cnt1 = 9

cnt increments to the value 9
then it initializes

Listing 6.11 VHDL code of the counter_24

6.3.4 Design of 7-Segment Display

The function of the “7_segment” block is to decode the 4-bit input of the counters, into a 7-bit output to drive the 7-segment displays. The “7_segment” block will be the same for the whole project regardless of the type of counter. The VHDL code of this circuit is presented in the Listing 6.12.

```

entity segment is
  port
  (
    a : in std_logic_vector(3 downto 0);
    b: out std_logic_vector(6 downto 0)
  );
end entity;
architecture rtl of segment is
begin
process(a)
begin
  case a is
    when"0000"=>b<="1000000";
    when"0001"=>b<="1111001";
    when"0010"=>b<="0100100";
    when"0011"=>b<="0110000";
    when"0100"=>b<="0011001";
    when"0101"=>b<="0010010";
    when"0110"=>b<="0000010";
    when"0111"=>b<="1111000";
    when"1000"=>b<="0000000";
    when"1001"=>b<="0010000";
    when others=>b<="ZZZZZZZ";
  end case;
end process;
end rtl;

```

Decoder
BCD / 7 segments

Listing 6.12 VHDL code of the 7-segment display

6.3.5 Final Design of Digital Clock

Now we put all the blocks together to form the overall digital clock project. To achieve this, we first declare the components already designed in the declarative part of the architecture, which are.

- Freq_div: the frequency divider.
- Segment: the BCD/7 segment decoder block.
- Counter1: the units counter for seconds and minutes.
- Counter2: the tens counter for seconds and minutes.
- Counter24: the units and tens of hours counter.

This step is to assemble the components previously designed in VHDL for the digital clock. To achieve this, we first declare these components in the declarative part of the architecture and then we will instantiate them using the **Port_map** command (see Listing 6.13).

```

entity digital_clock is
  port
    ( clkx,resetx: in std_logic ;
      m0x,six,m0x,mix,h0x,h1x: out std_logic_vector(6 downto 0));
end entity;
architecture has of digital_clock is
signal clk0:std_logic;
signal j1,j2,j3,j4,j5,j6:std_logic_vector(3 downto 0);
signal h1,h2,h3,h4:std_logic;
component freq_dev
port (clk : in std_logic;
      reset : in std_logic;
      q : out std_logic);
end component;
component Segment
port (a : in std_logic_vector(3 downto 0);
      b: out std_logic_vector(6 downto 0));
end component;
component Counter1
port (clk1 : in std_logic;
      reset1 : in std_logic;
      clk12 : out std_logic;
      q1 : out std_logic_vector(3 downto 0)
    );
end component;

```

Declaration of signals to interconnect the different blocks

Declaration of components: freq_dev, Deoder BCD / 7 segments, counter1

```

component counter2
port(clk2 : in std_logic;
      reset2 : in std_logic;
      clk22 : out std_logic;
      q2 : out std_logic_vector(3 downto 0));
end component;
component counter24
port(clk3 : in std_logic;
      reset3 : in std_logic;
      q3 : out std_logic_vector(3 downto 0);
      q4 : out std_logic_vector(3 downto 0));
end component;
begin
assign1:freq_dev port map(clkx,resetx,clk0);
assign2:counter1 port map(clk0,resetx,h1,j1);
assign3:counter2 port map(h1,resetx,h2,j2);
assign4:counter1 port map(h2,resetx,h3,j3);
assign5:counter2 port map(h3,resetx,h4,j4);
assign6:segment port map(j1,m0x);
assign7:segment port map(j2,six);
assign8:segment port map(j3,m0x);
assign9:segment port map(j4,mix);
assign10:counter24 port map(h4,resetx,j5,j6);
assign11:segment port map(j5,h0x);
assign12:segment port map(j6,h1x);
end has;

```

Declaration of components: counter2, counter24

Instantiation of components

Listing 6.13 VHDL of the final design of digital clock

6.3.6 Implementation of the Final Design of Digital Clock on the FPGA Platform

Figure 6.14 shows the test results of the digital clock on the DE2-70 board. We use two switches SW1 and SW2 for the control of the digital clock. The digital clock correctly counts the hours, minutes, and seconds as expected when switch SW(1) is high. The digital clock is initialized when the switch SW (2) is active.

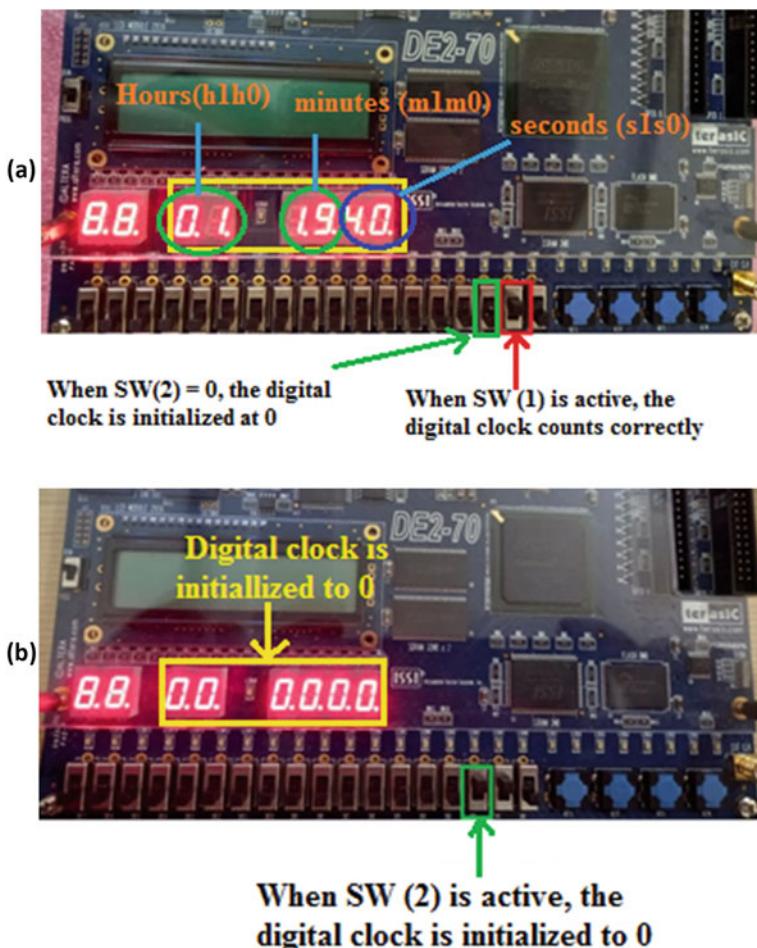


Fig. 6.14 Digital clock test results on the DE2-70 board ((a) and (b)). **a** When SW(1) is active, the digital clock counts correctly the hours, minutes and the seconds. **b** When SW(2) is active, the digital clock is initialized to 0

6.4 Lab #3 Design of Traffic Light Controller

The objective of this laboratory is to design in VHDL a traffic light controller for the intersection of four main roads A, B, C and D shown in Fig. 6.15. For this, we use the state diagram (FSM) of a Mealy machine based on the guidelines provided. We will then use this state diagram to write the behavioral VHDL description of the traffic light controller. The traffic light controller test is done using the Quartus development tools and the DE-70 Altera board.

The block diagram of the traffic light controller is shown in Fig. 6.16. It has two inputs: clock signal (CLK) and reset signal (reset). There are twelve outputs: G3 (green signal for road A), R3 (red signal for road A), O3 (orange signal for road A), V4 (green signal for road B), R4 (red signal for road B), O4 (orange signal for road B), V1 (green signal for road C), R1 (red signal for road C), O1 (orange signal for

Fig. 6.15 Traffic light intersection

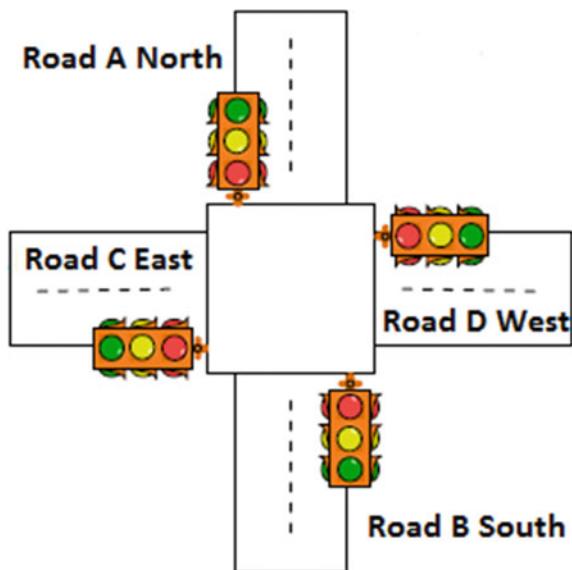
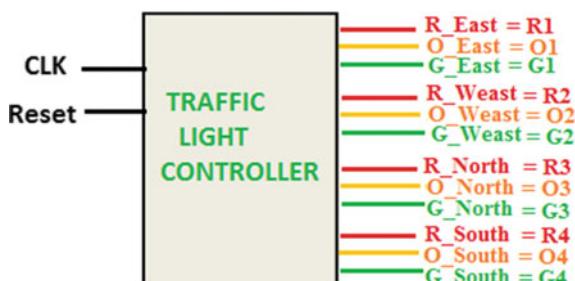


Fig. 6.16 Block diagram of the traffic light controller



road C), V2 (green signal for road D), R2 (red signal for road D), O2 (orange signal for road 4).

6.4.1 Traffic Light Controller Operation

The traffic light controller operates in six phases grouped together in Table 6.2. As shown in Table 6.2, at **state 0**, the system is in the idle state and the lights of roads A, B, C and D are in red (R1, R2, R3, R4). After 2 s, the lights on roads A and B will turn green (G3, G4) for 10 s while the lights on roads C and D (R1, R2) will turn red for the same time and the system traffic controller will go to **state 1**.

When the lights on roads A and B change from green (G3, G4) to orange (O3, O4), the lights on roads C and D will remain red (R1, R2) for 10 s and the controller is in the **state 2**. After a waiting time of 2 s, the controller will switch to **state 3**. The traffic light A and B will turn orange (O3, O4) for 10 s while the lights on roads C and D will turn green (G1, G2). After 2 s the system traffic controller transits to **state 4**, the cars on roads A and B stop (R3, R4) for 2 s and cars on roads C and D prepare to stop since the orange light has activated (O1, O2).

Finally, after 2 s, the system traffic controller changes to state 5, cars on roads A and B are allowed to pass (G3, G4), and cars on roads C and D must stop (red R1, R2) for 10 s.

The delay columns of Table 6.2 represent the values that the counter must reach in each state before moving on to the next state. Obviously, to avoid accidents, the lights on all roads A, B, C and D should not be green at the same time. Only the roads A and B or C and D are complementary. The green and red lights stay on for

Table 6.2 Different phases of the traffic light controller

States	Road A North			Road B South			Road C East			Road D West			Delay (s)
	R3	O3	G3	R4	O4	G4	R1	O1	G1	R2	O2	G2	
State 0: (R1, R2, R3, R4)	■	0	0	■			■	0	0	■	0	0	2
State 1: (G3, G4, R1, R2)	0	0	■	0	0	■		0	0	■	0	0	10
State 2: (O3,O4,R1,R2)	0	■	0		■		■			0	0	0	2
State 3: (R3, R4, G1, G2)	■	0	0	■	0	0	0	■	0	0	■	■	10
State 4: (R3, R4, O1, O2)	■	0	0			0	■	0	0	■	0	0	2
State 5: (G3, G4, R1, R2)	0	0	■		■	■	■	0	0	■	0	0	10

an exact time. In addition, the green transitions on each road must be followed by 2 s of orange.

6.4.2 Design Traffic Light Controller in VHDL

To describe the behavior of the traffic light controller in VHDL, we use the FSM whose state diagram is shown in Fig. 6.17. Time line in seconds of FSM design is illustrated in Table 6.2.

The following VHDL code (see Listing 6.14) illustrates the operation of the FSM traffic light controller.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4
5  -----Entity of the traffic light controller -----
6  entity Traffic_light_controller is
7  port (
8    clk:in std_logic; --50 MHz FPGA board clock
9    clr: in std_logic; --clear or reset mode
10   seven_seg : out std_logic_vector(13 downto 0);--7segment display shower
11   --East road traffic lights:
12   O_EAST: out std_logic; --Orange east
13   G_EAST: out std_logic; -- green east
14   R_EAST: out std_logic; --red east
15   --West road traffic lights:
16   O_WEST: out std_logic; --Orange west
17   G_WEST: out std_logic; -- green west
18   R_WEST: out std_logic; -- red west
19   --North road traffic lights:
20   O_NORTH: out std_logic; -- Orange north
21   G_NORTH: out std_logic; -- green north
22   R_NORTH: out std_logic; -- red north
23   --South road traffic lights:
24   O_SOUTH: out std_logic;-- Orange south
25   G_SOUTH: out std_logic;--green south
26   R_SOUTH: out std_logic --red south
27   );
28 end Traffic_light_controller;
29

```

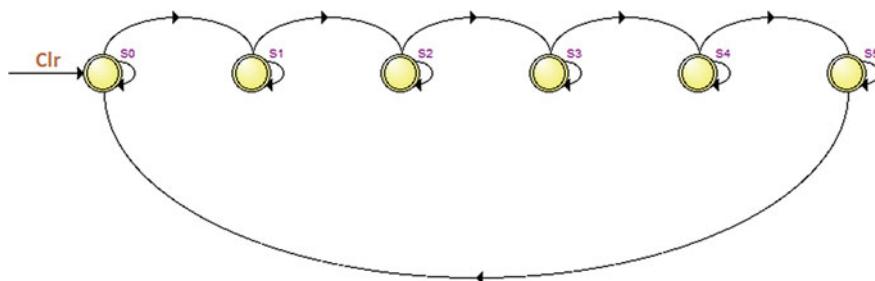


Fig. 6.17 FSM design of the traffic light controller

```

30 ----- Traffic light controller architecture-----
31  architecture traffic of Traffic_light_controller is
32  type etat_type is (S0, S1, S2, S3, S4, S5); --state type declaration
33  signal etat: etat_type; --signal state
34  signal count: std_logic_vector( 3 downto 0); --Signal counter
35  constant SEC1 : std_logic_vector (3 downto 0):= "1011";
36  constant SEC2 : std_logic_vector(3 downto 0):= "0110";
37 begin
38
39 process (clk, clr)
40 begin
41 if (clr ='1') then --first mode when reset=1
42     etat <= S0;
43     count <= X"0";
44 elsif clk'event and (clk ='1')then -- normal mode when reset=0
45 case (etat) is
46 when S0 => --first case
47     if count < SEC1 then
48         etat <= S0;
49         count<= count + 1;
50 else
51     etat <= S1;
52     count <= X"0";
53 end if;
54 when S1 => --second case
55     if count < SEC2 then
56         etat <= S1;
57         count <= count + 1;
58 else
59     etat <= S2; --third case
60     count <= X"0";
61 end if ;
62

```

```

63 when S2 =>
64     if count < SEC1 then
65         etat <= S2;
66         count <= count + 1;
67 else
68     etat <= S3;
69     count <= X"0";
70 end if;
71
72 when S3 => -- case four
73     if count < SEC2 then
74         etat <= S3;
75         count <= count + 1;
76 else
77     etat <= S4;
78     count <= X"0";
79 end if;
80
81 when S4 => --case five
82     if count < SEC1 then
83         etat <= S4;
84         count <= count + 1;
85 else
86     etat <= S5;
87     count <= X"0";
88 end if;
89

```

```

90  when S5 => --case six
91  |   if count < SEC2 then
92  |       etat <= S5;
93  |       count <= count + 1;
94  |   else
95  |       etat <= S0;
96  |       count <= X"0";
97  |   end if;
98
99  when others => --the others case
100 |   etat <= S0;
101
102 end case;
103 end if;
104
105 end process;
106
107 --Process for displaying counter values on the 7-segment display
108 C1: process (count)
109 begin
110 case count is
111 when "0000" =>
112     seven_seg <= "1111111000000"; --0
113 when "0001" =>
114     seven_seg <= "1111111111001"; --1
115 when "0010"=>
116     seven_seg <= "11111110100100"; --2
117 when "0011" =>
118     seven_seg <= "11111110110000";--3
119 when "0100" =>
120     seven_seg <= "11111110011001";--4
121 when "0101" =>

```

```

122     seven_seg <= "11111110010010";--5
123 when "0110" =>
124     seven_seg <= "11111110000010";--6
125 when "0111" =>
126     seven_seg <= "11111111111000";--7
127 when "1000" =>
128     seven_seg <= "11111110000000";--8
129 when "1001" =>
130     seven_seg <= "11111110010000";--9
131 when "1010" =>
132     seven_seg <= "11110011000000";--10
133 when "1011" =>
134     seven_seg <= "11110011111001";--11
135 when "1100" =>
136     seven_seg <= "11110010100100";--12
137 when "1101" =>
138     seven_seg <= "11110010110000"; --13
139 when "1110"=>
140     seven_seg <= "11110010011001" ;--14
141 when "1111"=>
142     seven_seg <= "11110010010010";--15
143 end case;
144 end process;
145

```

```

146 | -- Process to enumarate the differents state of lights
147 | C2: process (etat)
148 | begin
149 | case etat is
150 |   when S0 =>
151 |     R_EAST <= '1';
152 |     G_EAST <='0';
153 |     R_NORTH <='1';
154 |     G_NORTH <='0';
155 |     O_NORTH <='0';
156 |     O_EAST <='0';
157 |     R_WEST <= '1';
158 |     G_WEST <='0';
159 |     R_SOUTH <='1';
160 |     G_SOUTH <='0';
161 |     O_SOUTH <='0';
162 |     O_WEST <='0';
163 |
164 |   when S1 =>
165 |     R_EAST <= '1';
166 |     R_NORTH <='0';
167 |     G_NORTH <='1';
168 |     G_EAST <='0';
169 |     O_NORTH <='0';
170 |     O_EAST <='0';
171 |     R_WEST <= '1';
172 |     R_SOUTH <='0';
173 |     G_SOUTH <='1';
174 |     G_WEST <='0';
175 |     O_SOUTH <='0';
176 |     O_WEST <='0';
177 |
178 |   when S2 =>
179 |     R_EAST <= '1';
180 |     R_NORTH <='0';
181 |     G_NORTH <='0';
182 |     G_EAST <='0';
183 |     O_NORTH <='1';
184 |     O_EAST <='0';
185 |     R_WEST <= '1';
186 |     R_SOUTH <='0';
187 |     G_SOUTH <='0';
188 |     G_WEST <='0';
189 |     O_SOUTH <='1';
190 |     O_WEST <='0';
191

```

192	when S3 =>
193	R_EAST <= '0';
194	R_NORTH <='1';
195	G_NORTH <='0';
196	G_EAST <='1';
197	O_NORTH <='0';
198	O_EAST <='0';
199	R_WEST <= '0';
200	R_SOUTH <='1';
201	G_SOUTH <='0';
202	G_WEST <='1';
203	O_SOUTH <='0';
204	O_WEST <='0';
205	

```

206      when S4 =>
207          R_EAST <= '0';
208          R_NORTH <='1';
209          G_NORTH <='0';
210          G_EAST <='0';
211          O_NORTH <='0';
212          O_EAST <='1';
213          R_WEST <= '0';
214          R_SOUTH <='1';
215          G_SOUTH <='0';
216          G_WEST <='0';
217          O_SOUTH <='0';
218          O_WEST <='1';
219
220      when S5 =>
221          R_EAST <= '1';
222          R_NORTH <='0';
223          G_NORTH <='1';
224          G_EAST <='0';
225          O_NORTH <='0';
226          O_EAST <='0';
227          R_WEST <= '1';
228          R_SOUTH <='0';
229          G_SOUTH <='1';
230          G_WEST <='0';
231          O_SOUTH <='0';
232          O_WEST <='0';
233
234      when others =>
235          R_EAST <= '1';
236          R_NORTH <='1';
237          G_NORTH <='0';
238          G_EAST <='0';
239          O_NORTH <='0';
240          O_EAST <='0';
241          R_WEST <= '1';
242          R_SOUTH <='1';
243          G_SOUTH <='0';
244          G_WEST <='0';
245          O_SOUTH <='0';
246          O_WEST <='0';
247
248      end case;
249
250  end process;
end traffic;

```

Listing 6.14 VHDL code of the FSM traffic light controller

The simulation results of Listing 6.14 demonstrate the correct operation of the traffic light controller (see Fig. 6.18). When the reset is disable ($\text{clr} = 0$), the traffic light controller is in **state 0** (RED east–west/RED north–south) for 2 s. In this state, no authorization is granted for the passage of cars from roads A, B, C and D.

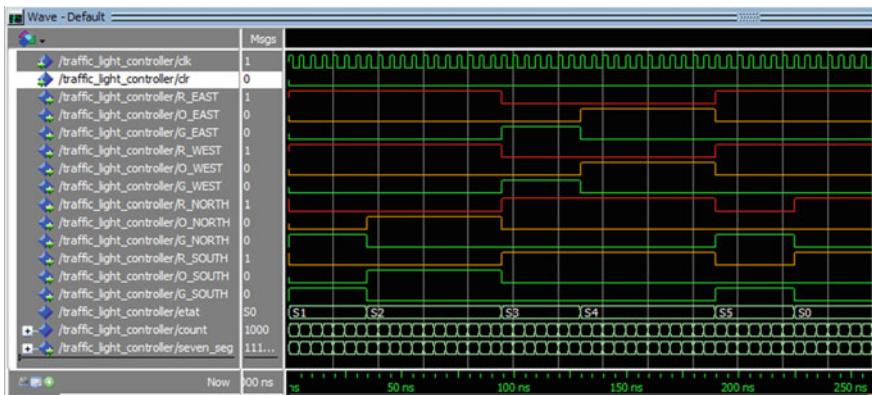


Fig. 6.18 Waveform simulation of traffic light controller

After a waiting time of 2 s, the controller transits to **state 1** (RED east–west/GREEN north–south). In this state, the lights on roads A and B are green while the lights on roads C and D are red. After a waiting time of 10 s, the controller transits to **state 2** (RED east–west/YELLOW north–south). In this state, the lights on roads A and B are yellow while the lights on roads D and C are red. This state takes 2 s after that, the transit traffic controller in **state 3** (RED north–south/GREEN east–west) to stay for 10 s. Than the system moves to the **state 4** (RED north–south/YELLOW east–west). After a time of 2 s, the traffic light controller switch to the last state: **state 5** (RED east–west/GREEN north–south) for 10 s.

- The lasts 2 lines with yellow are the binary code for the two 7-segments display.
- The time of the clock is not a real time with seconds is with nanoseconds to see all the cycle of the simulation.

6.4.3 Traffic Light Controller Implementation in the FPGA Platform

To test and validate the design of the traffic light controller on the FPGA platform, we need to slow down the 50 MHz clock signal from the FPGA board to a frequency of 1 Hz. Listing 6.15 shows the VHDL code of the frequency divider.

```

1  library ieee;
2  use ieee.STD_LOGIC_1164.ALL;
3  USE IEEE.STD_LOGIC_UNSIGNED.ALL;
4  entity frequency_div is
5    generic (N: integer:=25999999);
6  PORT (
7    clk_50, reset: in std_logic;
8    clk_1h : out std_logic);
9  end frequency_div;
10
11 architecture bhv of frequency_div is
12   signal temp: std_logic;
13 begin
14   process (clk_50, reset,temp)
15   variable s:integer range 0 to N;
16   begin
17     if reset= '1' then
18       s:=0;
19       temp <= '0';
20     elsif clk_50'event and clk_50 = '1' then
21       if s=N then
22         s:=0;
23         temp <= not temp;
24       else
25         s:=s+1;
26       end if;
27     end if;
28   end process;
29   clk_1h <=temp;
30 end bhv;

```

Listing 6.15 VHDL code of the frequency divider

We assign the inputs and outputs of this controller to the pins listed in Table 6.3.

Figure 6.19 shows the experimental results of the validation of the traffic controller on the FPGA platform. The traffic controller transits from state S0 to S5 via states S1, S2, S3 and S4. These different states are illustrated in Fig. 6.19a–d.

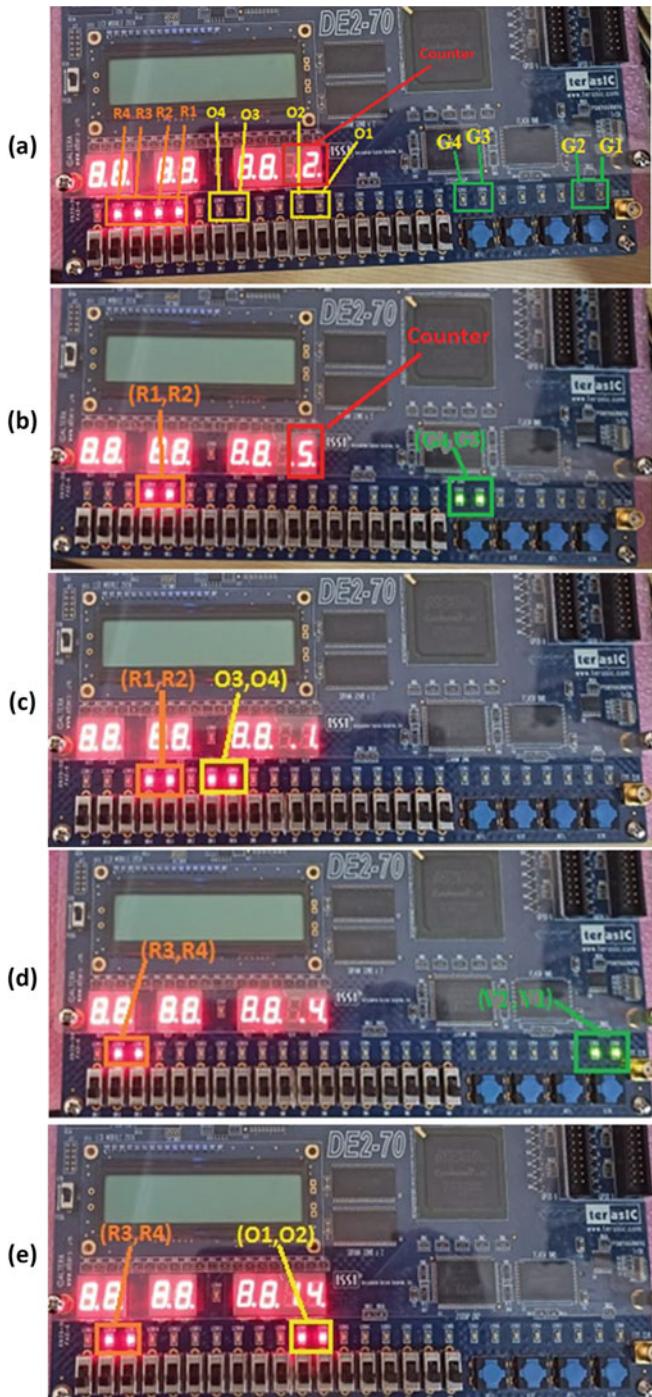
6.5 Lab #4 Vending Machine Controller

A vending machine is a machine that automatically dispenses items such as drinks, coffee, chips, soda, and other goods, after the customer has inserted some currency into the machine. In this project, we will design in VHDL a controller for a vending machine. As can be seen in Fig. 6.20, this controller has seven inputs: One_in, Two_in, Five_in, Coffee_in, chips_in, soda_in, Return_All_money_in, and two additional inputs clk (clock) and reset, which are also needed. The controller responds with five outputs: Coffee_out, Chips_out, Soda_out, Change, All_money_out.

It is possible for the customer to access the contents of the dispenser and choose the desired drink among the many varieties offered. Also, he can buy a combination

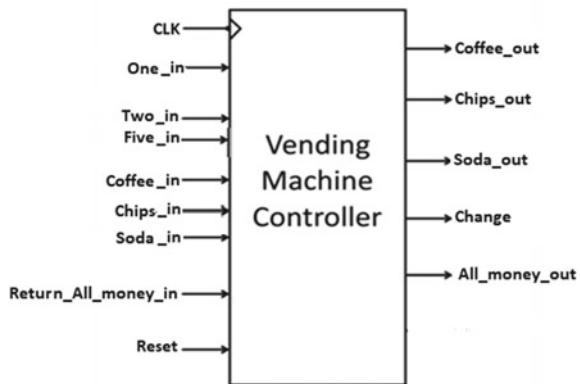
Table 6.3 Inputs and outputs pins of traffic light controller

Code vhdl	Clk	Reset	RED East	Yellow East	GREEN East	RED North	Yellow North	Green North	RED West	Yellow West	Green West	Red South	Yellow South	Green South
FPGA board	CLK_15	SWO	LEDR_15	LEDR_16	LEDR_17	LEDR_0	LEDR_1	LEDR_2	LEDR_0	LEDR_1	LEDR_2	LEDG_7	LEDG_6	LEDG_5



◀Fig. 6.19 Validation of the traffic controller on FPGA platform. **a** System is in the **state S0**, the lights on roads A, B, C and D are red (R1, R2, R3, R4), the LEDs assigned for the green and orange lights are off. **b** System is in the **state S1**, the lights on roads A and B will turn green (V3, V4) for 10 s while the lights on roads C and D (R1, R2) will turn red for the same time. **c** System is in the **state S2**, the traffic light A and B will turn orange (O3, O4) for 2 s while the lights on roads C and D remain red (R1, R2). **d** System is in the **state S3**: the cars on roads C and D will turn red (R3, R4) for 10 s while the lights on roads A and B will turn green (V1, V2). **e** System is in the **state S4**: the cars on roads A and B stop (R3, R4) for 2 s and cars on roads C and D prepare to stop since the orange light has activated (O1, O2)

Fig. 6.20 Vending machine controller



of drinks depending on how much money he wants to spend. The machine will return the balance amount if it provides excess money and also return the change.

The operating logic of the automatic distributor is detailed by the points below:

- The machine accepts 3 types of Money: 1dh, 2dh, 5dh.
- The machine can dispense:
 - Coffee: which costs 2 DH
 - Chips: which costs 3 DH
 - Soda: which costs 5 DH.
- The machine has a Coin Return button; it gives back all the money inserted (all_Money_Out).

6.5.1 State Machine Diagram of Vending Machine

The state transition diagram for the vending machine is illustrated in Fig. 6.21. The machine has twelve states (st0, st1, st2, st3, st4, st5, st6, st7, st8, st9, st10 and st11) indicating the total amount of credit and the product to choose from.

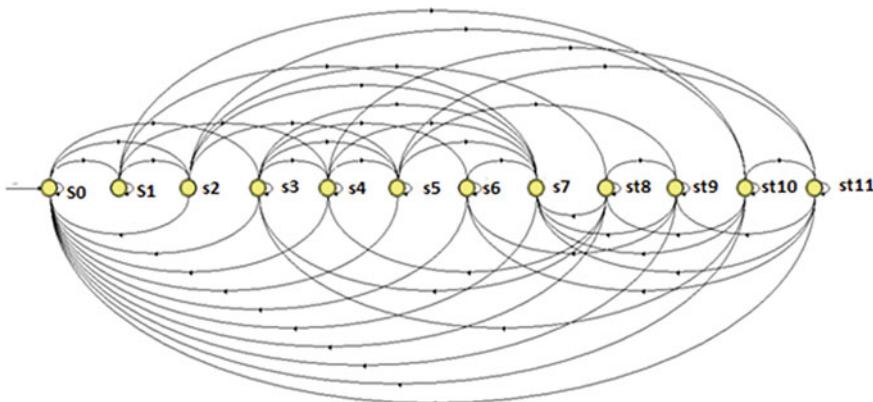


Fig. 6.21 State machine diagram (FSM) of vending machine

State 0 is the state s_0 . If nothing has happened, or if the machine has been reset, the machine is in state “ s_0 ”. From it, if 1dh is inserted, the machine goes to state s_1 . In this state, the user has six possibilities: either he does nothing and the machine remains in state s_1 ; either he activates “Return_All_Money” and the machine returns his money to him; or he inserts another 1 dh and the machine will go to state s_2 . Or he inserts 2dh and the machine will go to state s_3 , or he inserts 5dh and the machine will go to state s_6 .

State s_2 indicates that 2 dh is inserted, and we choose coffee Then, the coffee is dispensed, with no change. When the user selects the Chips or Soda, we received nothing, when he activates “Return_All_Money”, the machine gives him back his money, if the user does nothing, the machine remains in state s_2 . In this state s_2 if the user inserted another 1dh, the machine will go to state s_3 , or if the user inserted another 2 dh, the machine will go to state s_4 and if we inserted 5 dh, the machine will go to state s_7 .

The state s_3 indicates that 3 dh is inserted. If the user chooses the coffee, the coffee is dispensed, with a change of 1dh, because the coffee cost 2dh, or if the user chooses the chips, the chips is dispensed, with no change, because the chips cost 3dh, when he activates “Return_All_Money”, the machine gives him back his money, if the user does nothing, the machine remains in state s_3 .

In this state s_3 , if the user inserted another 1dh, the machine will go to state s_4 , or if the user inserted 2dh, the machine will go to state s_5 , and if the user inserted 5dh, the machine will go to state s_8 .

The state s_4 indicates that 4dh dh is inserted. If the user chooses the coffee, the coffee is dispensed with change of 2dh, or if the user chooses the chips, the chips is dispensed, with change of 1dh, if the user does nothing, the machine remains in state s_4 .

In this state, if the user inserted 1dh, the machine will go to state s_5 , or if the user inserted 2dh, the machine will move to state s_6 , and if the user inserted 5 dh, the machine will go to state s_9 .

The state st5indicates that 5dh dh is inserted, same as previous states; The user could do no action and stay at this state, or if he chooses coffee, the coffee is dispensed with change of 3dh, or if he chooses chips, the chips is dispensed with change of 2dh, and if he choose Soda, the soda is dispensed with no change. If the user does nothing, the machine remains in state st5.

In this state st5, if we inserted another 1dh, the machine will move to state st6, or if the user inserted 2dh, the machine will move to state st7, and if the user inserted 5dh the machine will go to state st10.

The state st6indicates that 6dh dh is inserted, or if he chooses coffee, the coffee is dispensed with change of 4dh, or if he chooses chips, the chips is dispensed with change of 3dh, and if he choose Soda, the soda is dispensed with change for 1dh.

In this state st6, if we inserted another 1dh, the machine will go to state st7, or if the user inserted 2dh, the machine will go to state st8.if the user does nothing, the machine remains in state st6.

The state st7 indicates that 7dh dh is inserted, or if he chooses coffee, the coffee is dispensed with change of 5dh, or if he chooses chips, the chips is dispensed with change of 4dh, and if he choose Soda, the soda is dispensed with change for 2dh.if the user does nothing, the machine remains in state st7.

In this state st7, if we inserted another 1dh, the machine will go to state st8, or if the user inserted 2dh, the machine will go to state st9, if the user does nothing, the machine remains in state st7.

The state st8 indicates that 8dh dh is inserted, or if he chooses coffee, the coffee is dispensed with change of 6dh, or if he chooses chips, the chips is dispensed with change of 5dh, and if he choose Soda, the soda is dispensed with change for 3dh.

In this state st8, if we inserted another 1dh, the machine will go to state st9, or if the user inserted 2dh, the machine will go to state st10, if the user does nothing, the machine remains in state st8.

The state st9 indicates that 9dh dh is inserted, or if he chooses coffee, the coffee is dispensed with change of 7dh, or if he chooses chips, the chips is dispensed with change of 6dh, and if he choose Soda, the soda is dispensed with change for 4dh.In this state st9, if we inserted another 1dh, the machine will go to state st10, if the user does nothing, the machine remains in state st9.

The state st10 indicates that 10dh dh is inserted, or if he chooses coffee, the coffee is dispensed with change of 8dh, or if he chooses chips, the chips is dispensed with change of 7dh, and if he choose Soda, the soda is dispensed with change for 5dh.if the user does nothing, the machine remains in state st10. The **state st11** is reserved for All_money_out, if the user input retuen_all_money in the different state, the machine will go to state st11, and the machine gives him back his moneys.

6.5.2 VHDL Implementation and Simulation of Vending Machine

A VHDL code for the vending machine is shown below in Listing 6.16. In the declarative part of architecture of vending machine (line 19), the enumerated data type, called state, contains all states of the state transition diagram of Fig. 6.21. There are twelve states: st0, st1, st2, st3, st4, st5, st6, st7, st8, st9, st10 and st11. The VHDL code is divided in two parts. The first part concerns the activation of the reset. This is an asynchronous reset, which determines the initial state of the machine “st0”, lines 24–31. The second part of the VHDL code specifies the different states through which the machine passes. For this, a “Case statement” is employed, lines 35–377.

```

1  library      IEEE;
2  use          IEEE.STD_LOGIC_1164.ALL;
3  use          IEEE.STD_LOGIC_UNSIGNED.ALL;
4  use          IEEE.STD_LOGIC_ARITH.ALL;
5
6  entity Vending is
7  Port (
8    clk, rst
9    one_in,two_in, five_in
10   return_All_money
11   coffee_in, chips_in,soda_in
12   coffee_out, chips_out,soda_out
13   change
14   All_money_out
15   bcd
16   end Vending;
17
18  architecture Behavioral of Vending is
19  TYPE State_type IS (st0, st1, st2, st3, st4, st5, st6, st7, st8, st9,st10,st11) ;
20  SIGNAL present_state, next_state : State_type ;
21
22  Begin
23
24  P1: process (clk, rst) is
25  begin
26  if (rst = '1') then
27  present_state <= st0;
28  elsif (rising_edge(clk)) then
29  present_state <= next_state;
30  end if;
31  end process;
32
33  P2: process (present_state,one_in, two_in, five_in,coffee_in,chips_in,soda_in,return_all_money) is
34  begin

```

```

34 begin
35 case present_state is
36
37 when st0 =>
38 coffee_out <= '0';
39 chips_out <= '0';
40 soda_out <= '0';
41 change <="0000";
42
43 if one_in = '1' then
44 next_state <= st1;
45 elsif two_in = '1' then
46 next_state <= st2;
47 elsif five_in = '1' then
48 next_state <= st5;
49 else
50 next_state <= st0;
51 end if;
52
53 --INSERTION OF 1DN:
54 WHEN st1 =>
55 coffee_out<='0';
56 chips_out <='0';
57 soda_out <='0';
58 change <="0000";
59 bcd <= "0000";
60 next_state <= st0;
61
62 if one_in = '1' then
63 next_state <= st2;
64 elsif two_in = '1' then
65 next_state <= st3;
66 elsif five_in = '1' then
67 next_state <= st6;
68
69 elsif return_All_money = '1' then
70 next_state <= st11;
71 else
72 next_state <= st1;
73 end if;
74
75 --INSERTION of 2DN:
76 WHEN st2 =>
77 if coffee_in ='1' then
78 coffee_out <='1';
79 chips_out <='0';
80 soda_out <='0';
81 change <= "0000";
82 bcd <= "0001";
83 next_state <= st0;
84
85 elsif one_in = '1' then
86 next_state <= st3;
87 elsif two_in = '1' then
88 next_state <= st4;
89 elsif five_in='1' then
90 next_state <= st7;
91 elsif return_All_money <= '1' then
92 next_state <= st11;
93 else
94 next_state <= st2;
95 end if;
96
97 --INSERTION OF 3DN:
98 WHEN st3 =>
99 if coffee_in ='1' then
100 coffee_out <= '0';
101 soda_out <= '0';

```

```

102 change <= "0001";
103 bcd <= "0001";
104 next_state <= st0;
105
106 elsif chips_in = '1' then
107   coffee_out <='0';
108   chips_out <= '1';
109   soda_out <= '0';
110   change <= "0000";
111   bcd <= "0010";
112   next_state <= st0;
113
114 elsif one_in = '1' then
115   next_state <= st4;
116 elsif two_in = '1' then
117   next_state <= st5;
118 elsif five_in = '1' then
119   next_state <= st8;
120 elsif returnAll_money = '1' then
121   next_state <= st11;
122 else
123   next_state <= st3;
124 end if;
125
126 --INSERTION OF 4DH:
127 WHEN st4 =>
128 if coffee_in = '1' then
129   coffee_out <='1';
130   chips_out <= '0';
131   soda_out <= '0';
132   change <= "0010";
133   bcd <= "0001";
134   next_state <= st0;
135

```

```

136 elsif chips_in = '1' then
137   coffee_out <='0';
138   chips_out <= '1';
139   soda_out <= '0';
140   change <= "0001";
141   bcd <= "0001";
142   next_state <= st0;
143
144 elsif one_in = '1' then
145   next_state <= st5;
146 elsif two_in = '1' then
147   next_state <= st6;
148 elsif five_in = '1' then
149   next_state <= st9;
150 elsif returnAll_money = '1' then
151   next_state <= st11;
152 else
153   next_state <= st4;
154 end if;
155
156 --INSERTION OF 5DH:
157 WHEN st5=>
158 if coffee_in = '1' then
159   coffee_out <='1';
160   chips_out <= '0';
161   soda_out <= '0';
162   change <= "0011";
163   bcd <= "0001";
164   next_state <= st0;
165
166 elsif chips_in = '1' then
167   coffee_out <='0';
168   chips_out <= '1';
169   soda_out <= '0';

```

```

170  change <="0010";
171  bcd <= "0010";
172  next_state <= st0;
173
174  elsif soda_in ='1' then
175      coffee_out <='0';
176      chips_out <= '0';
177      soda_out <= '1';
178      change <="0000";
179      bcd <= "0100";
180      next_state <= st0;
181
182  elsif one_in ='1' then
183      next_state <= st6 ;
184  elsif two_in = '1' then
185      next_state <= st7;
186  elsif five_in = '1' then
187      next_state <= st10;
188  elsif return_All_money = '1' then
189      next_state <= st11;
190  else
191      next_state<=st5;
192  end if ;
193
194 --INSERTION OF 6DH:
195 WHEN st6>
196  if coffee_in='1' then
197      coffee_out <='1';
198      chips_out <= '0';
199      soda_out <= '0';
200      change <="0100";
201      bcd <= "0001";
202      next_state <= st0;
203
204  elsif chips_in ='1' then
205      coffee_out='0';
206      chips_out<='1';
207      soda_out<='0';
208      change="0011";
209      bcd <= "0010";
210      next_state <= st0;
211
212  elsif soda_in ='1' then
213      coffee_out <='0';
214      chips_out <= '0';
215      soda_out <='1';
216      change <="0001";
217      bcd <= "0100";
218      next_state <= st0;
219

```

```
220 elsif one_in='1' then
221     next_state <= st7;
222 elsif two_in='1' then
223     next_state <= st8;
224 elsif return_All_money = '1' then
225     next_state <= st11;
226 else
227     next_state <= st6;
228 end if;
229 --
--INSERTION OF 7DH:
230 WHEN st7 =>
231 if coffee_in ='1' then
232     coffee_out <= '1';
233     chips_out <= '0';
234     soda_out <= '0';
235     change <= "0101";
236     bcd <= "0001";
237
238 next_state <= st0;
239
240 elsif chips_in ='1' then
241     coffee_out<'0';
242     chips_out<='1';
243     soda_out<='0';
244     change<="0100";
245     bcd <= "0010";
246     next_state <= st0;
247
248 elsif soda_in ='1' then
249     coffee_out <='0';
250     chips_out <= '0';
251     soda_out <= '1';
252     change <="0010";
253     bcd <= "0100";
254     next_state <= st0;
255
256 elsif one_in ='1' then
257     next_state<=st8;
258 elsif two_in ='1' then
259     next_state <=st9;
260 elsif return_All_money = '1' then
261     next_state <= st11;
262 else
263     next_state <=st7;
264 end if ;
265 --
--INSERTION OF 8DH:
266 WHEN st8 =>
267 if coffee_in ='1' then
268     coffee_out <= '1';
269     chips_out <= '0';
270     soda_out <= '0';
```

```

272     change <= "0110";
273     bcd <= "0001";
274     next_state <= st0;
275
276     elsif chips_in ='1' then
277         coffee_out<='0';
278         chips_out<='1';
279         soda_out<='0';
280         change<="0101";
281         bcd <= "0010";
282         next_state <= st0;
283
284     elsif soda_in ='1' then
285         coffee_out <='0';
286         chips_out <= '0';
287         soda_out <= '1';
288         change <="0011";
289         bcd <= "0100";
290         next_state <= st0;
291
292     elsif one_in ='1' then
293         next_state <=st9;
294     elsif two_in ='1' then
295         next_state <= st10;
296     elsif return_All_money='1' then
297         next_state <=st11;
298     else
299         next_state <=st8;
300     end if ;
301
302     --INSERTION OF 9DH:
303     WHEN st9 =>
304     if coffee_in ='1' then
305         coffee_out <= '1';

306     chips_out <= '0';
307     soda_out <='0';
308     change <= "0111";
309     bcd <= "0001";
310     next_state <= st0;
311
312     elsif chips_in ='1' then
313         coffee_out<='0';
314         chips_out<='1';
315         soda_out<='0';
316         change<="0110";
317         bcd <= "0010";
318         next_state <= st0;
319
320     elsif soda_in ='1' then
321         coffee_out <='0';
322         chips_out <= '0';
323         soda_out <= '1';
324         change <="0100";
325         bcd <= "0100";
326         next_state <= st0;
327
328     elsif one_in='1' then
329         next_state <= st10;
330     elsif return_All_money='1' then
331         next_state <= st11;
332     else
333         next_state <= st5;
334     end if ;
335
336     --INSERTION OF 10DH:
337     WHEN st10 =>
338     if coffee_in ='1' then
339         coffee_out <= '1';

```

```

340 chips_out <= '0';
341 soda_out <= '0';
342 change <= "1000";
343 bcd <= "0001";
344 next_state <= st0;
345
346 elsif chips_in = '1' then
347   coffee_out<='0';
348   chips_out<='1';
349   soda_out<='0';
350   change<="0111";
351   bcd <= "0010";
352   next_state <= st0;
353
354 elsif soda_in = '1' then
355   coffee_out <='0';
356   chips_out <='0';
357   soda_out <='1';
358   change <="0101";
359   bcd <= "0100";
360   next_state <= st0;
361
362 elsif return_All_money = '1' then
363   next_state<= still;
364 else
365   next_state <=st10;
366 end if ;
367
368 --THIS STATE INDICATE THAT WE WANT TO RETURN ALL MONEY.
369 WHEN still =>
370 if return_all_money = '1' then
371   All_money_out <= '1';
372   next_state <= st0;
373 else
374   all_money_out <= '0';
375 end if;
376
377 END CASE ;
378 end process;
379
380 end Behavioral;
381

```

Listing 6.16 VHDL code of vending marching

The simulation results of vending machine are depicted in Fig. 6.22. When the customer inserts the 2 dh coin and chooses the drink of the coffee, the machine dispenses the coffee (costs 2 dh) without returning the coin. When the customer inserts the 5 dh coin and chooses the drink of the soda, the machine dispenses the

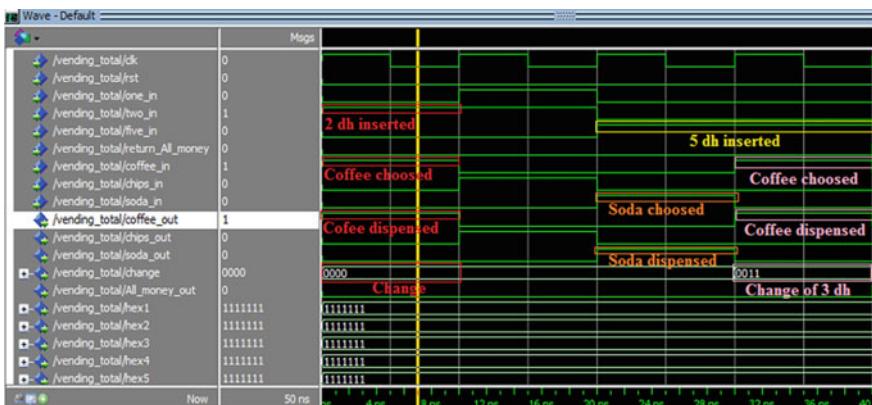


Fig. 6.22 Simulation results of vending machine

soda (costs 2 dh) without returning the coin. The machine gives back the currency of 3 dh when the customer inserts 5 dh and chooses the coffee (cost 2 dh).

6.5.3 Implementation and Validation on FPGA Platform

The Altera FPGA board includes two oscillators producing 28 MHz and 50 MHz clock signals. As a result, we did not have time to act on the switch of the FPGA board to be able to validate the inserted currency. In order to solve this problem, we have made a clock divider of 1 Hz. Listing 6.17 shows the VHDL code of clock divider of 1 Hz.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_arith.all;
4  use ieee.std_logic_unsigned.all;
5
6  entity frequency_div is
7    generic (N: integer:= 25999999);
8  port (
9    clk, rst : in std_logic;
10   clk_lhz : out std_logic );
11 end frequency_div;
12 architecture bhv of frequency_div is
13 signal temp:std_logic;
14 begin
15 process (clk,rst,temp)
16 variable s:integer range 0 to N;
17 begin
18 if rst= '1' then
19   s:= 0 ;
20   temp <= '0';
21 elsif clk'event and clk = '1' then
22 if s= N then
23   s:= 0;
24   temp <= not temp;
25 else
26   s:=s+1;
27 end if ;
28 end if;
29 end process;
30 clk_lhz <= temp ;
31 end bhv;
```

Listing 6.17 VHDL code of clock divider of 1 Hz

The diagram in Fig. 6.23 shows the association of the clock divider with the vending machine.

The total vending machine is formed by the association of the “vending machine” module and the “frequency divider” module using the structural description. Listing 6.18 shows the VHDL code for the link between the clock divider and the main code of the vending machine.

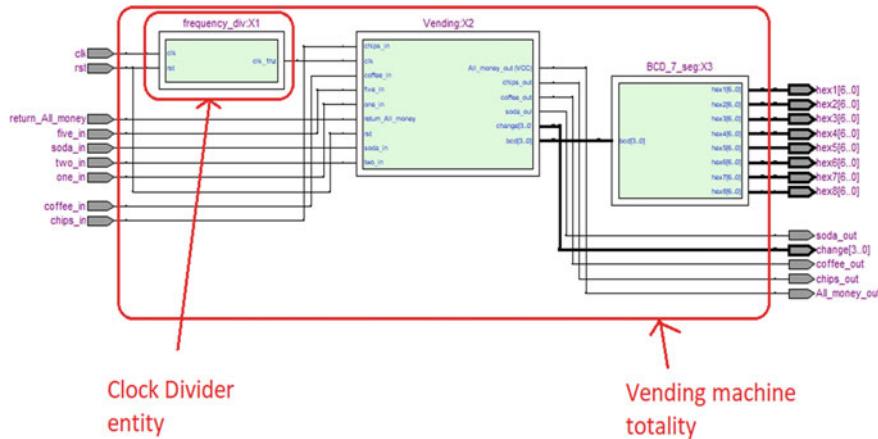


Fig. 6.23 Final vending machine: association of the clock divider with the vending machine

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Vending_total is
5    port (clk
6          ,rst
7
8          ,one_in,two_in, five_in
9          ,return_All_money
10         ,coffee_in,Chips_in,soda_in
11         ,coffee_out,chips_out,soda_out
12         ,change
13         ,All_money_out
14         ,hex1,hex2,hex3,hex4,hex5,hex6,hex7,hex8
15
16      end Vending_total;
17
18  architecture one of Vending_total is
19
20  component frequency_div
21  port (clk,rstin std_logic ;
22        clk_lhz : out std_logic);
23
24  end component;
25
26  component Vending
27  port ( clk, rst
28        ,one_in,two_in, five_in
29        ,return_All_money
30        ,coffee_in,Chips_in,soda_in
31        ,coffee_out,chips_out,soda_out
32        ,change
33        ,All_money_out
34        ,bcd

```

Table 6.4 Pins assigned of vending machine (inputs and output)

Inputs		Outputs	
Clock	PIN AD15	All_Money_Out	LEDR16
Reset	SW17	Coffee_Out	LEDR0
Return_All_Money	SW16	Chips_Out	LEDR1
One_in	SW0	Soda_Out	LEDR2
Two_in	SW1	Change[0]	LEDR4
Five_in	SW2	Change[1]	LEDR5
Coffee_in	SW3	Change[2]	LEDR6
Chips_in	SW4	Change[3]	LEDR7
Soda_in	SW5	Change[4]	LEDR8

```

35
36
37
38
39      end component;
40  component BCD_7_seg
41    port ( bcd : in std_logic_vector(3 downto 0);
42            hex1,hex2,hex3,hex4,hex5,hex6,hex7,hex8 : out STD_LOGIC_VECTOR (6 downto 0));
43  end component;
44
45
46
47  signal S: std_logic;
48  signal bcd : std_logic_vector(3 downto 0);
49  begin
50    X1: frequency_div port map (clk,rst,S);
51    X2: Vending port map (S,rst,one_in,two_in,five_in,return_all_money,coffee_in,chips_in,soda_in,
52                           coffee_out,chips_out,soda_out,change,all_money_out,bcd);
53    X3: bcd_7_seg port map ( bcd,hex1,hex2,hex3,hex4,hex5,hex6,hex7,hex8);
54  end one;
55

```

Listing 6.18 VHDL code for the total vending machine module

In order to validate the design of vending machine in the FPGA platform, the pins assigned for vending machine are listed in the Table 6.4. In addition, we have reserved the display 7 segment HEX8 for the display of the price of the products.

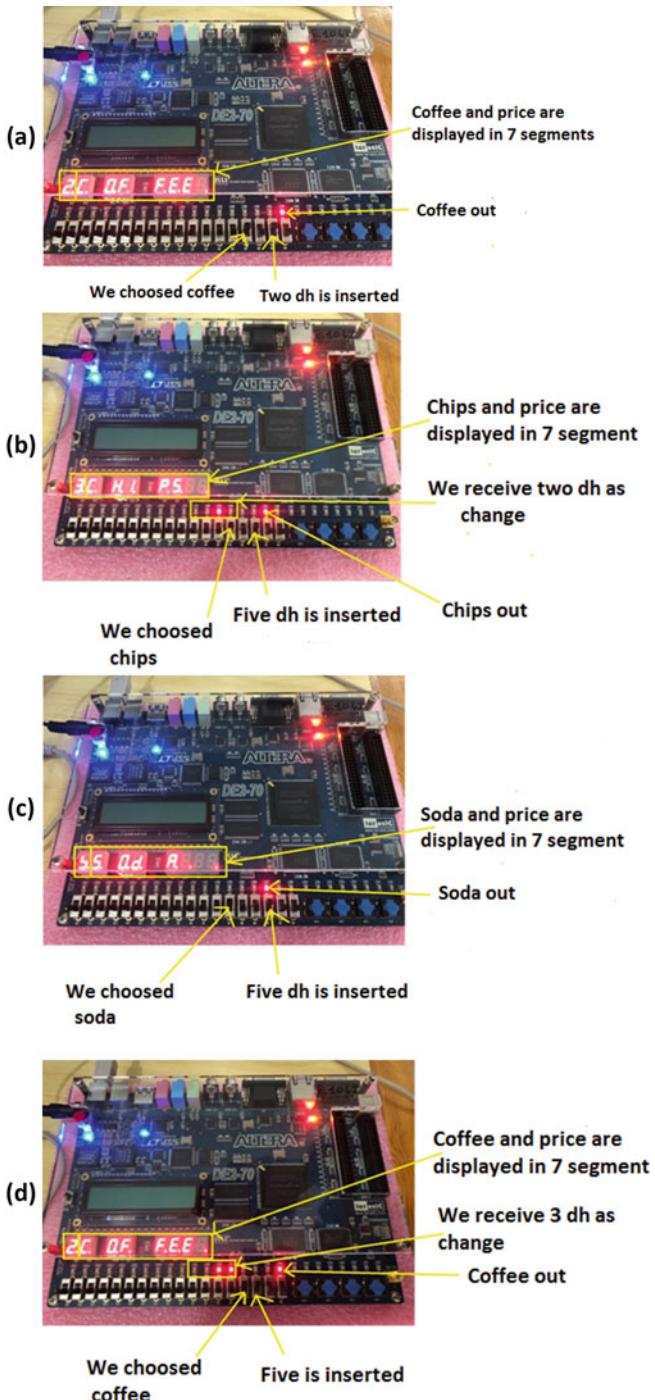
Different scenarios were tested in order to validate the design of the vending machine, as indicated below:

Case #1: the user inserted 2 dh, and he choose coffee, the coffee dispensed with no change (see Fig. 6.24a).

Case #2: 5dh is inserted 5dh, the user chooses chips, result: the chips dispensed with change of 2 dh (see Fig. 6.24b).

Case # 3: 5 dh is inserted, the use chooses soda, result: soda is dispensed with no change (see Fig. 6.24c).

Case # 4: 5 dh is inserted, the user chooses the coffee, result: coffee is dispensed with change of 3dh because the coffee cost 2dh (see Fig. 6.24d).



◀Fig. 6.24 Validation of design of vending machine in the FPGA platform. **a** 2 dh inserted, the user chooses the coffee, result: coffee is dispensed with no change. **b** 5 dh inserted, the user chooses the chips, result: chips is dispensed with change of 2 dh. **c** 5 dh inserted, the user chooses soda, result: soda is dispensed with no change. **d** 5dh inserted, the user chooses coffee, result: coffee is dispensed with 2dh as change

6.6 Lab #5 Control of a 4-Phase Step Motor (Speed and Position)

Unlike ordinary motors, which run continuously when powered, stepper motors are devices that rotate a specific number of degrees for each step (i.e. moving by a fixed amount of degrees). A stepper motor is widely used in applications requiring precise positioning such as robotics, disk drives, printers, clocks and watches, factory automation, and machinery. These motors can rotate in either direction and their speed can be controlled. Our stepper motor mainly consists of two parts: permanent magnet rotor (hybrid technology) and wound stator (4 pole pairs). A cross-sectional view of the engine is shown in Fig. 6.25.

The coils of wound stator are called control windings. The rotation of the motors is controlled by switching ON/FF the current through control windings. Normally the

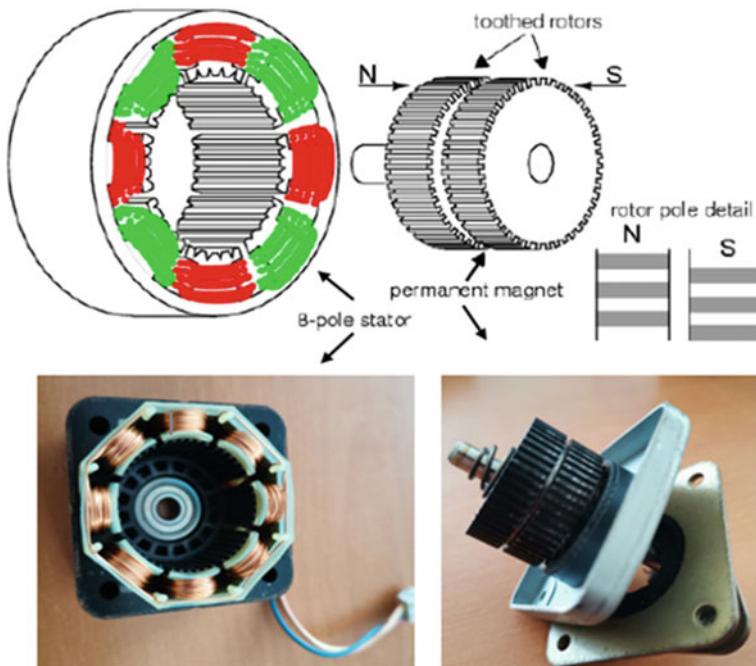


Fig. 6.25 Cross-sectional view of the stepper motor

control windings are excited through driver transistors. Stepper motors are classified into three types of stepper motors: variable reluctance motor (VR), permanent magnet motor (PM) and hybrid motors (HM). The purpose of this laboratory is to design in VHDL a stepper motor controller, allowing to control the stepper motor in position and in speed. The VHDL code which provides for the different operating modes of the motor, namely: full step mode, maximum torque or half step. Then the design of this controller is implemented and validated on the FPGA hardware platform.

6.6.1 Laboratory Design Specifications

The objective of this laboratory is to control the position and speed of a stepping motor using an FPGA board. The stepper motor-based controller hardware requires the following components: the stepper motor for converting digital information into mechanical motion, a phase sequence generator circuit to generate appropriate timed inputs to the stepper motor (this part will be designed here in the DE2-70 Altera FPGA), and a driver circuit to provide suitable voltage and current levels to the stepper motor as shown in Fig. 6.26.

The digital architecture of the controller is designed on an Altera Quartus II software platform using the Very High Speed Hardware Description Language (VHDL). At the top level, the controller takes as inputs the operating mode (Input C), direction of movement (Input dir), speed (Input A) and position (Input D) and faster position control (Input B) of the motor and generates four output signals (Q0, Q1, Q2 and Q3) used to switch the transistors which are used to drive the motor (see Table 6.5).

The specifications for the design of the stepper motor controller include the following points:

- Rotate the motor in both clockwise and counterclockwise directions.
- Drive the motor in different operating modes (full step and half step).
- Development of a power circuit to control the stepper motor
- Implementation of the program designed in the FPGA board.

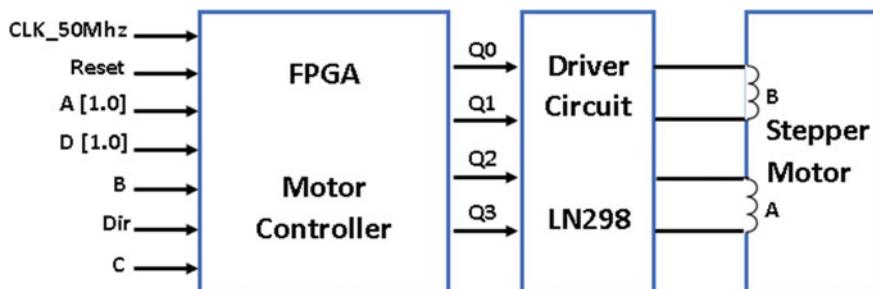


Fig. 6.26 Block diagram of stepper motor controller

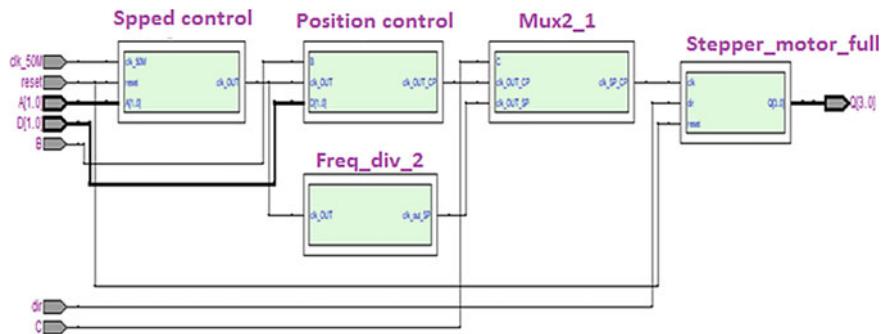
Table 6.5 Inputs and outputs for the Stepper motor controller

Signal	Direction	Description
RESET	Input	Resets the position of the motor to the initial reference position
Clk_50M	Input	Clock signal
Dir	Input	If Dir = 1, the motor turns clockwise otherwise it turns counterclockwise
A[1:0]	Input	This is the data address of the 4 to 1 multiplexer and of the 1 to 4 demultiplexer, its role is to select the desired clock signal
D[1:0]	Input	It is the data address of the multiplexer from 4 to 1 and of the demultiplexer from 1 to 4, its role is to select the desired position
B	Input	It is a push button, we send an impulse to force the motor to turn to a precise position (depending on the data address D)
C	Input	It allows you to choose the operating mode
Q[3:0]	Output	4-bit output of the device that goes to the MOSFET driver circuit

6.6.2 Stepper Motor Control: Speed and Position

The purpose of this part is to design in VHDL a stepper motor controller allowing to control the stepper motor in position and in speed. This is a VHDL program which provides for the different operating modes of the motor, namely: full step mode, maximum torque or half step. Then the design of this controller is implemented and validated on the FPGA hardware platform. Figure 6.27 shows the RTL implementation block diagram of the stepper motor controller. It has 5 blocks which are:

- Speed Control
- Position control
- Freq_div
- Mux2_1
- Stepper motor full.

**Fig. 6.27** RTL implementation block diagram of the stepper motor controller

Each of these blocks will be explained in future sections of this lab.

The following Listing 6.19 illustrates the VHDL code that allows us to control the speed and the position of the stepper motor.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Control_Stepper_Motor is
5  port(clk_50M,reset,dir,C,B:in std_logic;
6  A,D:in std_logic_vector(1 downto 0);
7  Q:out std_logic_vector(3 downto 0));
8  end Control_Stepper_Motor;
9
10 architecture Beh of Control_Stepper_Motor is
11 component Controle_position
12 Port(B,clk_OUT:in std_logic;D:in std_logic_vector(1 downto 0);clk_OUT_CP:out std_logic);
13 end component;
14 component Speed_Control
15 port(clk_50M,reset:in std_logic;A:in std_logic_vector(1 downto 0);clk_OUT:out std_logic);
16 end component;
17 component stepper_motor_full
18 port(clk,reset,dir:in std_logic;Q:out std_logic_vector(3 downto 0));
19 end component;
20 component Mux2_1_TO
21 port(clk_OUT_SP,clk_OUT_CP,C:in std_logic;clk_SP_CP:out std_logic);
22 end component;
23 component Div_par_2
24 port (clk_OUT: in std_logic;clk_out_SP: out std_logic);
25 end component;
26 signal signal_1:std_logic;
27 signal signal_2:std_logic;
28 signal signal_3:std_logic;
29 signal signal_4:std_logic;
30 begin
31 CCP: Controle_position port map (B,signal_1,D,signal_2);
32 CSC: Speed_Control port map(clk_50M,reset,A,signal_1);
33 CSM: stepper_motor_full port map(signal_4,reset,dir,Q);
34 CM: Mux2_1_TO port map (signal_3,signal_2,C,signal_4);
35 CD: Div_par_2 port map (signal_1,signal_3);
36 end Beh;
```

Listing 6.19 The VHDL code that controls the speed and position of the motor

6.6.2.1 Stepper Motor Speed Control

This part illustrates the working principle of the stepper motor speed controller system. It allows to control several speeds namely: revolution/400 s, revolution/40, revolution/4 s and revolution/0.8 s. As the L298 driver used by our motor is not suitable for the high frequency of 50 MHz generated by the DE2-70 Altera FPGA card (because of the switching frequency of the transistors), we slowed down the value of this frequency so that it was included in the operating interval of L298 (1 Hz-1 kHz). For each desired speed, the corresponding clock signal is generated as shown in the Table 6.6.

Figure 6.28 shows the RTL implementation block diagram of the speed controller system. It has 5 blocks which are:

- Demux1_4_SP
- Fre_Div
- Fre_Div_1
- Fre_Div_2

Table 6.6 Choice of clock signal according to the data address

Input of (Demux1_4_SP)	Data address (A)	Output of (Demux1_4_SP)
Clk_50MHz	00	Fre_Div
	01	Fre_Div_1
	10	Fre_Div_2
	11	Fre_Div_3

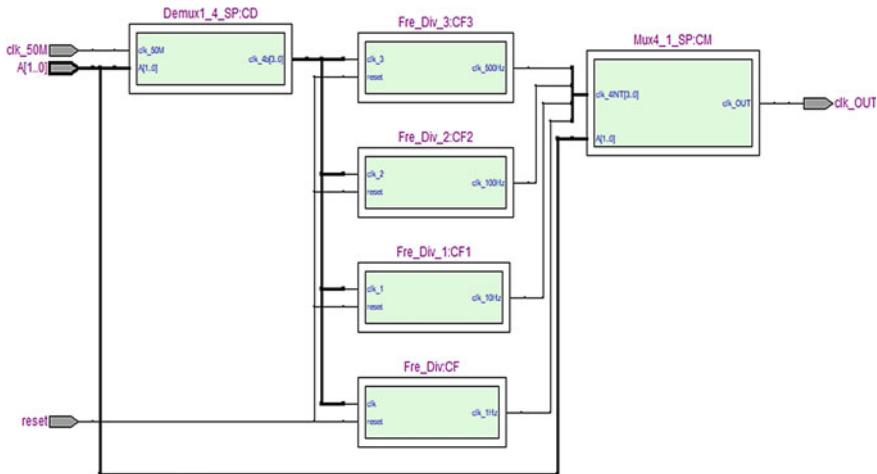


Fig. 6.28 RTL implementation block diagram of the speed controller system

- Fre_Div_3
- Mux4_1_SP.

As shown in Fig. 6.28, the speed controller system is based on the approach of frequency dividers. The frequency divider block (see Fig. 6.29) is needed to produce an idle clock signal in order to drive the motor at an appropriate speed. It accepts a 50 MHz clock signal input from the FPGA board and outputs a nHz signal, in our case $1 \text{ Hz} < n < 1 \text{ kHz}$.

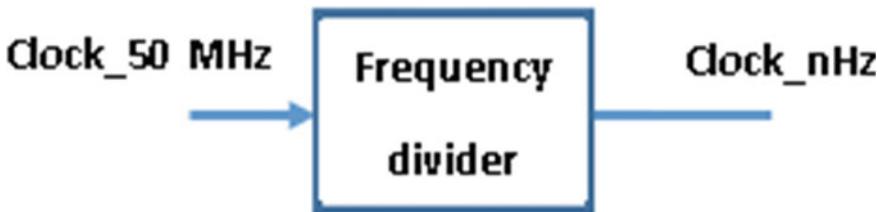


Fig. 6.29 A frequency divider module of nHz with $1 \text{ Hz} < n < 1 \text{ kHz}$

Table 6.7 The data of each frequency divider block

Block	Input	N	Speed	Output
Fre_Div	Clk_50M	25,000,000	rev/400 s	Clk_1Hz
Fre_Div_1	Clk_50M	2,500,000	rev/40 s	Clk_10Hz
Fre_Div_2	Clk_50M	250,000	rev/4 s	Clk_100Hz
Fre_Div_3	Clk_50M	50,000	rev/0.8 s	Clk_500Hz

Table 6.7 shows the data for each frequency divider block used to drive the stepper motor.

Listing 6.20 illustrates the VHDL code of the first frequency divider block “Fre_Div”:

```

1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_arith.all;
4   use ieee.std_logic_unsigned.all;
5
6   entity Fre_Div is
7     generic(N: integer :=25000000);
8   port (
9     reset,clk : in std_logic;
10    clk_1Hz : out std_logic);
11 end Fre_Div;
12
13 architecture methode of Fre_Div is
14 signal temp : std_logic;
15
16 begin
17 process(clk,reset,temp)
18 variable s:integer range 0 to N;
19 begin
20 if reset = '1' then
21   s := 0;
22   temp <= '0';
23 elsif (rising_edge(Clk)) then
24   if s = N then
25     s := 0;
26     temp <= not temp;
27   else
28     s := s + 1;
29   end if;
30 end if;
31 end process;
32 Clk_1Hz<= temp;
33 end methode;
```

Listing 6.20 The VHDL code of the “Fre_Div” block

The following Listing 6.21 represents the VHDL code for 1-to-4- demultiplexer (Demux1_4_SP block):

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Demux1_4_SP is
5  port (
6    clk_50M : in std_logic;
7    A: in std_logic_vector(1 downto 0);
8    clk_4b : out std_logic_vector(3 downto 0));
9  end Demux1_4_SP;
10
11 architecture one of Demux1_4_SP is
12 begin
13 process(clk_50M,A)
14 begin
15 case A is
16 when "00" => clk_4b(0) <= clk_50M;
17 when "01" => clk_4b(1) <= clk_50M;
18 when "10" => clk_4b(2) <= clk_50M;
19 when "11" => clk_4b(3) <= clk_50M;
20 end case;
21 end process;
22 end one;
--
```

Listing 6.21 The VHDL code of the “Demux1_4_SP” block

For the last block “Mux4_1_SP”, it is a 4-to-1 multiplexer, it allows to choose the desired clock signal according to the value of the data address “A”, as shown in the Table 6.8.

Table 6.8 Choice of clock signal according to the data address

Input	Data address (A)	Output
Clk_1Hz	00	Clk_OUT
Clk_10Hz	01	
Clk_100Hz	10	
Clk_500Hz	11	

The following Listing 6.22 represents the VHDL code of the multiplexer used.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Mux4_1_SP is
5  port (
6    clk_4INT :in std_logic_vector(3 downto 0);
7    A: in std_logic_vector(1 downto 0);
8    clk_OUT : out std_logic);
9  end Mux4_1_SP;
10
11 architecture one of Mux4_1_SP is
12 begin
13 process(clk_4INT,A)
14 begin
15 case A is
16 when "00" => clk_OUT <= clk_4INT(0);
17 when "01" => clk_OUT <= clk_4INT(1);
18 when "10" => clk_OUT <= clk_4INT(2);
19 when "11" => clk_OUT <= clk_4INT(3);
20 end case;
21 end process;
22 end one;
```

Listing 6.22 The VHDL code of the “Mux4_1_SP” block

The structural architecture is used to write the VHDL code of the motor speed controller. This approach is based on the instantiation of the components forming the speed controller system such as Demux1_4_SP, Fre_Div, Fre_Div_1, Fre_Div_2, Fre_Div_3 and Mux4_1_SP. Listing 6.23 shows the VHDL code for this controller.

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity Speed_Control is
5 port(clk_50M,reset:in std_logic;
6 A:in std_logic_vector(1 downto 0);
7 clk_OUT:out std_logic);
8 end Speed_Control;
9
10
11 architecture one of Speed_Control is
12 component Demux1_4_SP
13 port(clk_50M:in std_logic;A:in std_logic_vector(1 downto 0);clk_4b:out std_logic_vector(3 downto 0));
14 end component;
15 component Mux4_1_SP
16 port(clk_4INT:in std_logic_vector(3 downto 0);A:in std_logic_vector(1 downto 0);clk_OUT:out std_logic);
17 end component;
18 component Fre_Div
19 port(clk,reset:in std_logic;clk_lhz:out std_logic);
20 end component;
21 component Fre_Div_1
22 port(clk_1,reset:in std_logic;clk_10hz:out std_logic);
23 end component;
24 component Fre_Div_2
25 port(clk_2,reset:in std_logic;clk_100hz:out std_logic);
26 end component;
27 component Fre_Div_3
28 port(clk_3,reset:in std_logic;clk_500hz:out std_logic);
29 end component;
30 signal S1: std_logic_vector(3 downto 0);
31 signal S2: std_logic_vector(3 downto 0);
32 begin
33 CD: Demux1_4_SP port map (clk_50M,A,S1);
34 CF: Fre_Div port map (S1(0),reset,S2(0));
35 CF1: Fre_Div_1 port map (S1(1),reset,S2(1));
36 CF2: Fre_Div_2 port map (S1(2),reset,S2(2));
37 CF3: Fre_Div_3 port map (S1(3),reset,S2(3));
38 CM: Mux4_1_SP port map (S2,A,clk_OUT);
39 end one;

```

Listing 6.23 VHDL code of the speed controller system

In order to test the performance of the speed controller, we carried out two types of simulations according to data address command A. When A = 00, the speed



Fig. 6.30 Speed controller generates 1 Hz clock signal at its output

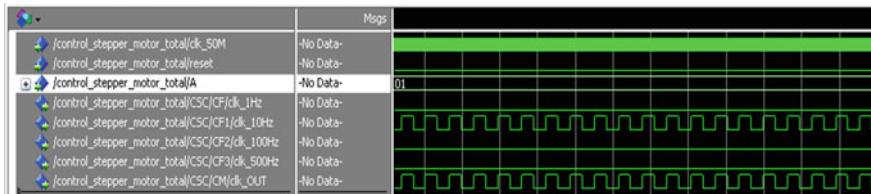


Fig. 6.31 Speed controller generates 10 Hz clock signal at its output

controller generates at its output a 1 Hz clock signal according as shown in the Fig. 6.30.

Similarly, when command A = 01, the speed controller delivers a 10 kHz clock signal, as shown in Fig. 6.31.

6.6.2.2 Position Control of a Stepper Motor

In this part, we present the position controller of our stepper motor. It allows to control four different positions: 45° , 90° , 180° and 360° , so.

- For a 1.8° turn, the motor takes one full step
- For a 45° turn, the motor takes 25 full steps.

Table 6.9 presents the positions of the motor according to the number of steps.

Figure 6.32 shows the RTL implementation block diagram of the position controller system. It has six blocks which are:

- Demux1_4_CP;
- Position_Control;
- Position_Control_1;
- Position_Control_2;
- Position_Control_3;
- Mux4_1_CP.

Table 6.9 Positions according to the number of steps

Number of steps	Positions
25	45°
50	90°
100	180°
200	360°

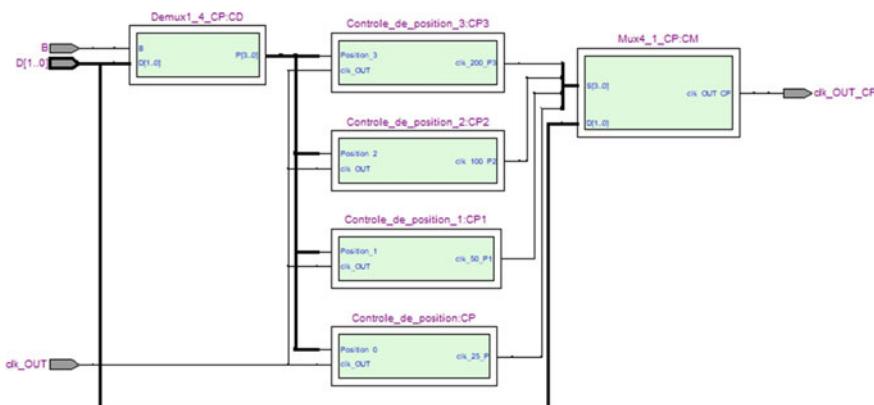


Fig. 6.32 RTL implementation block diagram of the position controller system

Listing 6.24 shows the VHDL code of the position controller block.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_arith.all;
4  use ieee.std_logic_unsigned.all;
5
6  entity Controle_de_position is
7    generic(P: integer:=50;Tr: integer :=1);
8  port (clk_OUT,Position_0 : in std_logic;
9        clk_25_P : out std_logic);
10 end Controle_de_position;
11
12 architecture step3 of Controle_de_position is
13 signal SI: std_logic;
14
15
16 begin
17 process(clk_OUT,Position_0,SI)
18 variable var: integer range 0 to Tr;
19 variable T:integer range 0 to P;
20 begin
21
22 if(rising_edge(clk_OUT)) then
23   if Position_0='0' then
24     if var=Tr then
25       T := 50;
26     end if;
27   end if;
28   if T=0 then
29     SI<='0';
30     var:=1;
31   else
32     T :=T- 1;
33     var:=0;
34     SI <= not(SI);
35   end if;
36 end if;
37 end process;
38 clk_25_P<=SI;
39 end step3;

```

Listing 6.24 VHDL code of the position controller block

The 1-to-4 demultiplexer block accepts at its inputs D, B (pushbutton) and delivers at its output, the position in n degree. When pushbutton “B” = 1, the motor rotates to a position of n degrees, with n chosen by the address data, as shown in Table 6.10.

Table 6.10 Choice of input signal according to the data address

Input of (Demux1_4_CP)	Data address (D)	Output of (Demux1_4_CP)
B	00	Contrôle_de_position ($n = 45^\circ$)
	01	Contrôle_de_position_1 ($n = 90^\circ$)
	10	Contrôle_de_position_2 ($n = 180^\circ$)
	11	Contrôle_de_position_3 ($n = 360^\circ$)

Listing 6.25 represents the VHDL code of the 1-to-4 demultiplexer block.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Demux1_4_CP is
5  port (
6    B: in std_logic;
7    D: in std_logic_vector(1 downto 0);
8    P: out std_logic_vector(3 downto 0));
9  end Demux1_4_CP;
10
11 architecture one of Demux1_4_CP is
12 begin
13 process(D)
14 begin
15 case D is
16 when "00" => P(0) <= B;
17 when "01" => P(1) <= B;
18 when "10" => P(2) <= B;
19 when "11" => P(3) <= B;
20 end case;
21 end process;
22 end one;
```

Listing 6.25 The VHDL code of the “Demux1_4_CP” block

The position control block is necessary in order to produce a clock signal which contains the exact number of rising edges corresponding to the step of the motor to be reached (n rising edges $\Leftrightarrow n$ no motor $\Leftrightarrow n \times 1, 8^\circ$).

Table 6.11 shows the data for each position control block used to control the stepper motor positions.

The “Mux4_1_CP” block is a 4-to-1 multiplexer; it allows us to choose the desired clock signal (which contains numbers of the specified rising edges) according to the value of the data address “D”. Table 6.12 represents the operations of the “Mux4_1_CP” block.

Listing 6.26 shows the VHDL code of the 4-to-1 multiplexer.

Table 6.11 The data of each position control block

Block	Input	P	Number of steps	Degree	Output
Controle_de_position	Clk_OUT	50	25	45°	Clk_25_P
Controle_de_position_1	Clk_OUT	100	50	90°	Clk_50_P1
Controle_de_position_2	Clk_OUT	200	100	180°	Clk_100_P2
Controle_de_position_3	Clk_OUT	400	200	360°	Clk_200_P3

Table 6.12 Choice of clock signal according to the data address

Input	Data address (D)	Output
Clk_25_P	00	Clk_OUT_CP
Clk_50_P1	01	
Clk_100_P2	10	
Clk_200_P3	11	

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Mux4_1_CP is
5  port (
6    S :in std_logic_vector(3 downto 0);
7    D :in std_logic_vector(1 downto 0);
8    clk_OUT_CP : out std_logic);
9  end Mux4_1_CP;
10
11 architecture one of Mux4_1_CP is
12 begin
13 process(D)
14 begin
15 case D is
16 when "00" => clk_OUT_CP<= S(0);
17 when "01" => clk_OUT_CP<= S(1);
18 when "10" => clk_OUT_CP<= S(2);
19 when "11" => clk_OUT_CP<= S(3);
20 end case;
21 end process;
22 end one;

```

Listing 6.26 VHDL code of the 4-to-1 multiplexer

The structural architecture is used to write the VHDL code of the motor speed controller. This approach is based on the instantiation of the components forming the speed controller system such as Demux1_4_CP, Position_Control, Position_Control_1, Position_control_2, position_control_3 and Mux4_1_CP. Listing 6.27 shows the VHDL code for the position control system.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Controle_position is
5    Port(B,clk_OUT:in std_logic;
6      D:in std_logic_vector(1 downto 0);
7      clk_OUT_Cp:out std_logic);
8  end Controle_position;
9
10 architecture one of Controle_position is
11   component Demux1_4_Cp
12     port (B: in std_logic;D: in std_logic_vector(1 downto 0);P: out std_logic_vector(3 downto 0));
13   end component;
14   component Mux4_1_Cp
15     port (S :in std_logic_vector(3 downto 0);D :in std_logic_vector(1 downto 0);clk_OUT_Cp : out std_logic);
16   end component;
17   component Controle_de_position_1
18     port (clk_OUT,Position_1 : in std_logic;clk_50_P1 : out std_logic);
19   end component;
20   component Controle_de_position
21     port (clk_OUT,Position_0 : in std_logic;clk_25_P : out std_logic);
22   end component;
23   component Controle_de_position_2
24     port (clk_OUT,Position_2 : in std_logic;clk_100_P2 : out std_logic);
25   end component;
26   component Controle_de_position_3
27     port (clk_OUT,Position_3 : in std_logic;clk_200_P3 : out std_logic);
28   end component;
29   signal S1:std_logic_vector(3 downto 0);
30   signal S2:std_logic_vector(3 downto 0);
31 begin
32   CD: Demux1_4_Cp port map (B,D,S1);
33   CP: Controle_de_position port map (clk_OUT,S1(0),S2(0));
34   CP1: Controle_de_position_1 port map (clk_OUT,S1(1),S2(1));
35   CP2: Controle_de_position_2 port map (clk_OUT,S1(2),S2(2));
36   CP3: Controle_de_position_3 port map (clk_OUT,S1(3),S2(3));
37   CM: Mux4_1_Cp port map (S2,D,clk_OUT_Cp);
38   end one;

```

Listing 6.27 VHDL code for the position control system

In order to test the performance of the position controller system, we carried out two types of simulations according to data address command D. When D = 00, the position controller generates at its output a clock signal corresponding to the position that the motor must turn 45° , according to Table 6.13.

From Fig. 6.33, it can be seen that when a clock signal is sent by pressing the push button, the position controller only allows the 25 rising edges of this signal to

Table 6.13 Pins assigned of stepper motor controller (inputs and Output)

	Inputs		Outputs
Clock	PIN AD15	Q(1)	PIN_C30
Rst	SW0	Q(2)	PIN_C29
Dir	SW4	Q(3)	PIN_E28
Mode	SW5	Q(4)	PIN_D29

**Fig. 6.33** Clock signal generation for 45°

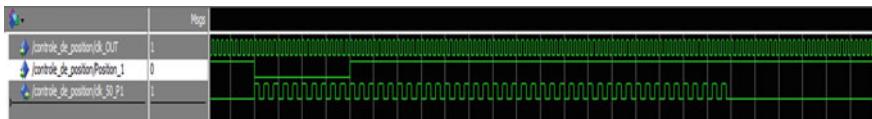


Fig. 6.34 Clock signal generation for 90°

pass at the output and blocks the rest. This allow the motor to rotate 45° and stop to wait for the next pushbutton command.

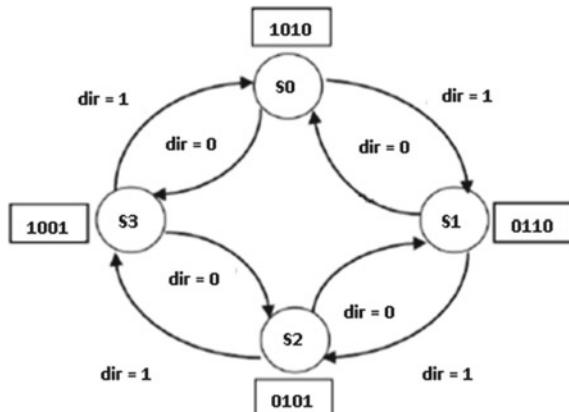
Figure 6.34 shows the simulation results of the position controller system for D = 01. In this situation, the controller generates an appropriate clock signal for 90-degree rotation performed by the motor.

6.6.2.3 Stepper Motor Full Control

A bipolar stepper motor has only two coils and four wires, as shown in Fig. 6.25. The rotation steps are controlled by the excitation on each of the coils that follows. The motor controller, designed in FPGA, must provide the appropriate sequence of four phases to rotate the stepper motor. For full stepping mode, the motor controller generates the following sequences 1010, 0110, 0101, 1001 to turn the motor. The VHDL code shown in Listing 6.28 controls the direction of the stepper motor operating in single-phase excitation. The FSM ensures that the correct sequence is followed for energizing the coil. The direction of rotation is changed by changing the coil feed sequence and the generated sequence is 1010, 0110, 0101 and 1001.

Figure 6.35 shows the state machine (FSM) for the stepper motor, where each state is drawn in a circle, and arcs between the states labeled with the input combinations that cause transitions from one state to another. With dir = 1, stepper motor will rotate in clockwise. With dir = 0, the direction will counterclockwise.

Fig. 6.35 FSM for stepper motor with direction control



```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity stepper_motor_full is
5  port(clk,reset,dir:in std_logic;
6  Q:out std_logic_vector(3 downto 0));
7  end stepper_motor_full;
8
9  architecture beh of stepper_motor_full is
10 type state_machine is (S0,S1,S2,S3);
11 signal state: state_machine;
12 begin
13 process(clk)
14 begin
15 if reset='1' then
16 state<=S0;
17 elsif clk'event and clk='1' then
18 if dir='1' then
19 case state is
20 when S0 => state <= S1;
21 when S1 => state <= S2;
22 when S2 => state <= S3;
23 when S3 => state <= S0;
24 end case;
25 else
26 case state is
27 when S0 => state <= S3;
28 when S3 => state <= S2;
29 when S2 => state <= S1;
30 when S1 => state <= S0;
31 end case;
32 end if;
33 end if;
34 end process;
35 with state select
36 Q<="1010" when S0,
37 "0110" when S1,
38 "0101" when S2,
39 "1001" when S3;
40 end beh;

```

Listing 6.28 VHDL code for the stepper motor operating in full step

Listing 6.28 reports the VHDL code for a 4-bit stepper motor with direction control. The inputs are clk, rst and dir and the output is a 4-bit Q vector. The state_machine is defined with four values: S0, S1, S2, S3. The internal signal is declared as TYPE: state_machine and will be assigned values in the CASE assignment group.

As can be seen in the process statement (line 13–34), when “reset” signal is 1 the entire finite state machine (FSM) is reset to state s0 and motor signal is initialized with “0101”. On the other hand, when “reset” signal is not activated, the rising edge of the input clock is taken into consideration to travel the FSM in next states and produce necessary output signals. When the rising edge of the clock is detected and the direction signal “dir” is high, the FSM moves to the following states and the motor rotates clockwise. Otherwise, the Dir signal is low, the motor rotates counterclockwise.

Listing 6.29 shows the total VHDL code which can control both the speed and position of the stepper motor.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Control_Stepper_Motor is
5  port(clk_50M,reset,dir,C,B:in std_logic;
6  A,Din:std_logic_vector(1 downto 0);
7  Q,out:std_logic_vector(3 downto 0));
8  end Control_Stepper_Motor;
9
10 architecture Beh of Control_Stepper_Motor is
11 component Controle_position
12  Port(B,clk_OUT:in std_logic;D:in std_logic_vector(1 downto 0);clk_OUT_CP:out std_logic);
13 end component;
14 component Speed_Control
15  port(clk_50M,reset:in std_logic;A:in std_logic_vector(1 downto 0);clk_OUT:out std_logic);
16 end component;
17 component stepper_motor_full
18  port(clk,reset,dir:in std_logic;Q:out std_logic_vector(3 downto 0));
19 end component;
20 component Mux2_1_TO
21  port(clk_OUT_SP,clk_OUT_CP,C:in std_logic;clk_SP_CP:out std_logic);
22 end component;
23 component Div_par_2
24  port (clk_OUT: in std_logic;clk_out_SP: out std_logic);
25 end component;
26 signal signal_1:std_logic;
27 signal signal_2:std_logic;
28 signal signal_3:std_logic;
29 signal signal_4:std_logic;
30 begin
31 CCP: Controle_position port map (B,signal_1,D,signal_2);
32 CSC: Speed_Control port map(clk_50M,reset,A,signal_1);
33 CSM: stepper_motor_full port map(signal_4,reset,dir,Q);
34 CM: Mux2_1_TO port map (signal_3,signal_2,C,signal_4);
35 CD: Div_par_2 port map (signal_1,signal_3);
36 end Beh;

```

Listing 6.29 The VHDL code to control the speed and position of the stepper motor

Figure 6.36 shows the simulation results for a stepper motor operating at full step (1.8°). When the reset (reset) is high, the stepper motor FSM remains in the reset state (S_0). When the reset goes low, then the state transition takes place in

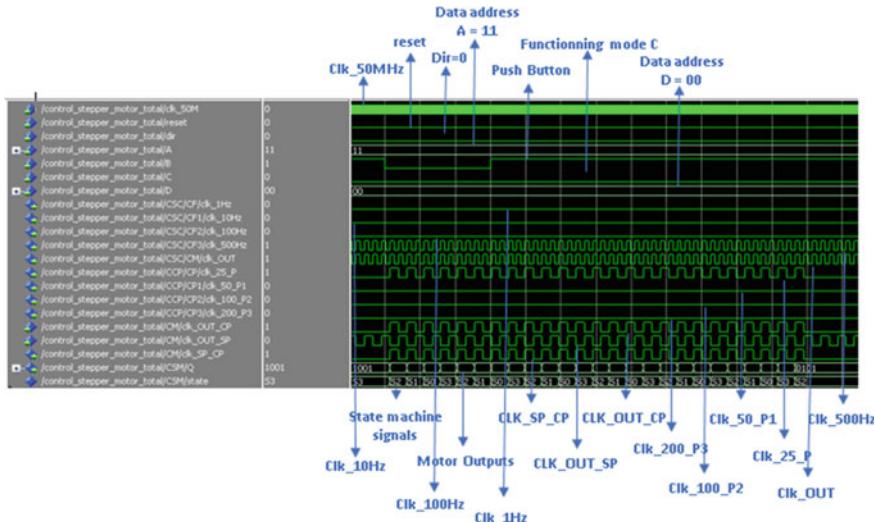


Fig. 6.36 Simulations results for the stepper motor operating in full step

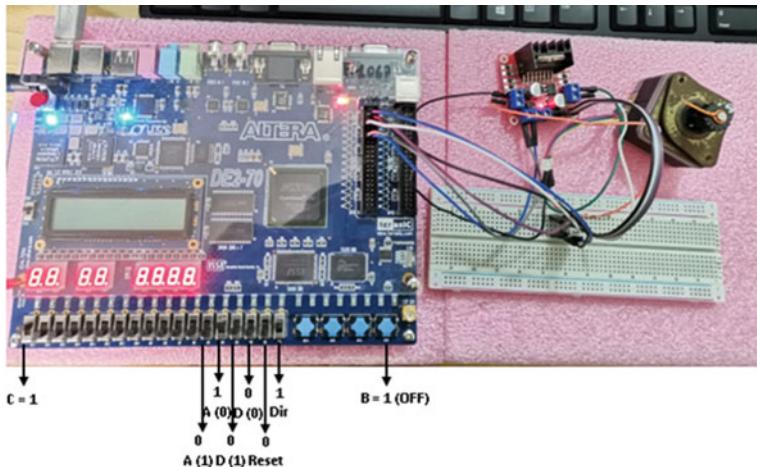


Fig. 6.37 Implementation of stepper motor controller on the FPGA platform: case 2: The speed control mode is selected ($C = 1$), the motor rotates clockwise ($Dir = 0$) at a revolution speed/40 s

one direction according to the direction control signal “dir”. When the dir signal is high, the stepper motor controller will produce the appropriate output signals to rotate the motor clockwise. Figure 6.36 also shows that the state machine performs its transition in the opposite direction when said signal is lowered.

6.6.2.4 Implementation of Stepper Motor Controller on the FPGA Platform

After the simulation check is completed, the motor controller is implemented into the DE2-70 FPGA board to perform a hardware test. Figure 6.37 illustrates the complete hardware configuration of the stepper motor controller. It shows a stepper motor connected to the FPGA board via an L293D driver circuit. This is an integrated circuit in the form of a double H-bridge. The FPGA board generates four signals Q1, Q2, Q3 and Q4 and sends them through a GPIO to the driver circuit L293 to run the motor. The motor frequency varies from 1 Hz to 1 kHz. The FPGA controller is driven only by the “Dir” and “Mode” pins. For the stepper motor side, four pins are connected to signals Q1, Q2, Q3 and Q4 through GPIO. Table 6.13 shows the pins assigned for stepper motor controller. Finally, depending on the mode selected (Speed if $C = 1$, Position if $C = 0$) and the direction of rotation chosen ($dir = 1$ or 0), the stepper has been driven up to a speed and a position desired demonstrating the correct operation of the controller designed in the FPGA platform (Fig. 6.38).

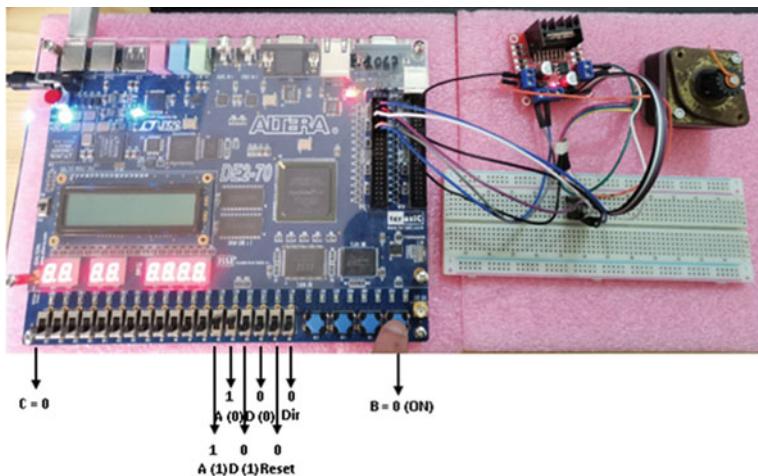


Fig. 6.38 Implementation of stepper motor controller on the FPGA platform: case 1: The position control mode is selected ($C = 0$), the motor rotates counter clockwise ($Dir = 1$) at a revolution speed/0.8 s (position of 45°)

6.7 Summary

In this chapter, a number of laboratory projects have been presented and discussed, this includes: a simple calculator, a digital clock, a traffic light control system, designing an automatic vending machine and controlling an engine, 4-phase step by step (Direction and speed). The operation of all projects has been verified by the simulation chronograms made using the Modelsim and Quartus tools. In addition, their designs have been also implemented and validated into the Altera FPGA platform.

These practical projects will constitute a good opportunity for those who wish to develop their skills in the design of digital systems.

Part IV

FPGA Applications

Chapter 7

FPGA Applications in Renewable Energy Systems: Photovoltaic, Wind-Turbine and Hybrid Systems



Abstract This part aims to present some examples of FPGA applications in photovoltaic and hybrid-photovoltaic systems. The chapter covers mainly four applications: (1) FPGA-based simulation of intelligent photovoltaic module, (2) FPGA-based implementation of irradiance equalization algorithm for reconfigurable photovoltaic (PV) arrays, (3) FPGA-based implementation of maximum power point tracking algorithms for PV modules, (4) and FPGA-XSG-based implementation control of grid-connected hybrid system (PV-WT). Examples presented are written under ISE 14.7, Quartus II and Xilinx System Generator DSP Matlab/Simulink environments. Four FPGA boards have been employed (Xilinx II pro, Spartan 3E, Cyclone II and Zinq-7000) for real time implementation and co-simulation. Basic skills and knowledge on Xilinx ISE design suite, Intel Quartus, ModelSim and Matlab/Simulink environments are required as well VHDL language.

7.1 Introduction

Solar photovoltaic (PV) and wind-turbine (WT) systems are among the most used renewable energy systems to produce electricity worldwide. With reference to the International Renewable Energy Agency (IRENA) [1] the global cumulative solar photovoltaic and wind energy capacity at the end of 2019 are 632.4 GW and 651 GW respectively.

In typical renewable energy systems implemented (RES) functionalities such as power conversion control and monitoring are on custom-built embedded boards. This is done using microcontrollers/digital signal processors (DSPs) that have fixed architectures, often limiting the kind of functionalities that a power converter designer wants to implement in their system. On the other hand, FPGAs are reconfigurable at hardware/logic gate level, giving some specific advantages to renewable energy designer [2, 3]. Nowadays, the FPGA platforms receive a considerable attention in several application fields due to their attractive features such us: parallelism, reprogramming capability, flexibility, and etc. [4]. These are some reasons why several scientists in engineering areas are very motivated by the using of FPGA devices.

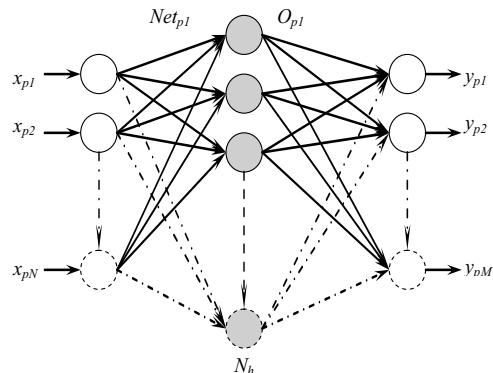
Various case studies will be presented in this chapter. We will cover practical aspects of FPGA-based renewable energy systems, particularly solar photovoltaic and hybrid photovoltaic-wind systems. This chapter is organized as follows: The next section describes a state of the art of recent application of FPGA in renewable energy systems, including photovoltaic, wind-turbine, grid-connected and off-grid PV systems. Section 7.2 presents a brief introduction to artificial neural networks (ANNs). The last Sect. 7.3 is devoted to some applications of FPGA mainly in photovoltaic systems and hybrid systems (PV-WT).

7.2 Neural Networks

ANNs are popular machine learning techniques (ML) that simulate the mechanism of learning in biological organisms [5]. ANNs provide successful models and metaphors to improve our understanding of the human brain. Figure 7.1 shows a typical structure of a feed-forward neural network (FFNN). It consists of one input layer, one output layer and few hidden layers. The number of the neurons in the input and the output layer are set before the training process, while the number of hidden layers is estimated during the learning process, as well as the number of the neurons within the hidden layers. There are many kinds of ANN (e.g. FFNN, radial basis function, recurrent NN and other).

Where x and y are the input and the output of the ANN, O , is the output of the neuron, N is the number of neurons in the hidden layer, Net is the network.

Fig. 7.1 Feed-forward neural network



7.3 Case Studies

7.3.1 Case Study 1: FPGA-Based Intelligent Photovoltaic Module Simulator

The idea behind the application of NNs and FPGA is to design of an intelligent PV module simulator. It could be useful for real-time predicting of PV technologies behavior. It is well known that a PV module can be simulated and modeled based on the basic equation of a solar cell, which is given by the following schematic (See Fig. 7.2). PV module of a number of solar cells connected in series can be written as follows:

$$I = I_{ph} - I_0 \left(e^{\left(\frac{V + IR_s}{N_s V_t A} \right)} - 1 \right) - \left(\frac{V + IR_s}{R_p} \right) \quad (7.1)$$

where I_{ph} is the light current, I_0 is the saturation current, A is the ideality factor, V_t is the thermal voltage, N_s is the number of cells in series, R_s series resistance and R_p is the shunt resistance.

To get the I-V curve, Eq. 7.1 needs to be solved by using for example an iterative method such as Newton–Raphson. However, predicting the behavior of a PV module under complex variable conditions or faults (See Fig. 7.3) using numerical methods is not so easy and cannot provide accurate results. Therefore, methods based on data-driven are the right solution for this application, without needing to solve complex mathematical equations. For example, ANNs have shown their capability in behavior prediction PV modules [6].

A dataset is needed to train the NN-based model. The main input that include any PV module technology are solar irradiance and air temperature [7]. Table 7.1 reports the main specifications of the used PV module.

So, a data-acquisition system is considered as an indispensable equipment for these methods. Figure 7.4 shows an example data-acquisition prototype used to collect PV current and voltages.

An example of the recorded data for one year is shown in Fig. 7.5.

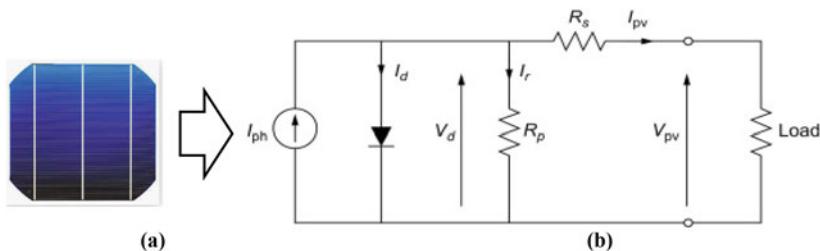


Fig. 7.2 a Solar cell, b one-diode electrical cell electrical circuit

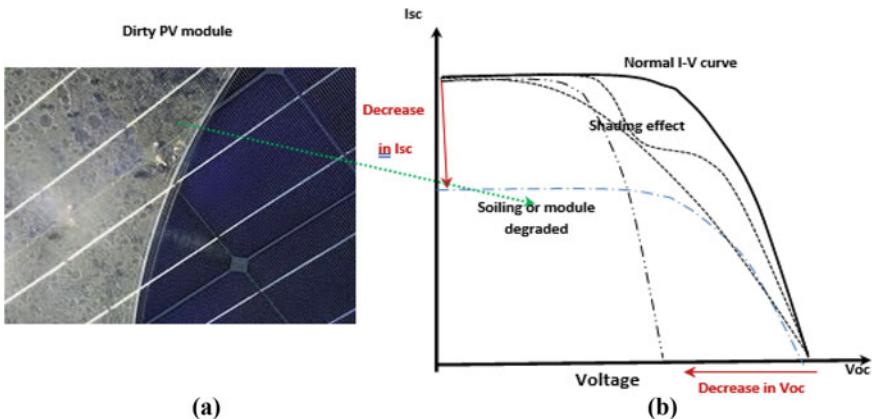


Fig. 7.3 **a** Dirty or soiling PV module, **b** I-V curves under normal and abnormal conditions

Table 7.1 PV module specifications at STC ($G = 1000 \text{ W/m}^2$ and $T = 25^\circ\text{C}$)

Open circuit voltage (V_{oc})	21.6 V
Short circuit current (I_{sc})	8.67 A
Voltage at maximum power point (V_{max})	17.3 V
Current at maximum power point (I_{max})	4.34 A
Produced maximum power (P_{max})	75 W



Fig. 7.4 Data-acquisition system for I and V measurement

A simple schematic of the application of an ANN for predicting the I-V curves is given Fig. 7.6. As can be seen we need as input some parameters such as solar irradiance and air temperature, the output are the predicted PV current and voltage.

Due to its simplicity implementation the well-known multi-layer perceptron (MLP), (see Fig. 7.7) is considered in the present application. The developed MLP-based PV module estimation consists of one input layer with two nodes or neurons

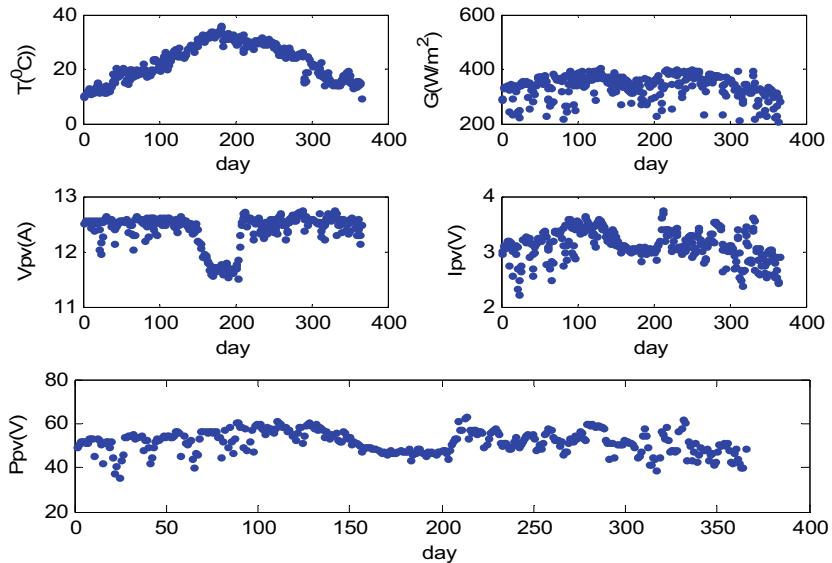


Fig. 7.5 Monitored data: solar irradiance, air temperature, PV module voltage, current and power

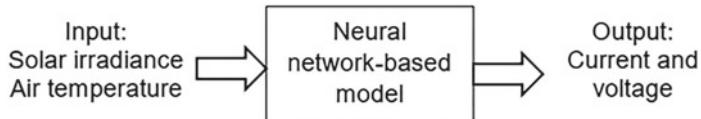
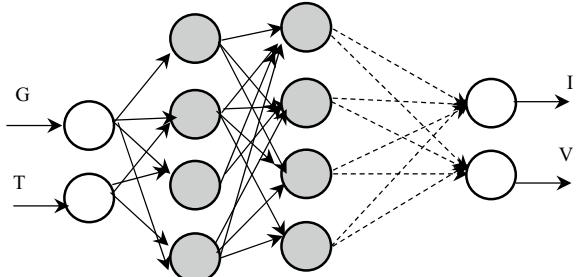


Fig. 7.6 Basic principle of an ANN for PV module output prediction

Fig. 7.7 Simplified MLP-based model for a PV module behavior prediction



(solar irradiation ‘G’ and air temperature ‘T’), two hidden layers within some nodes and one output layer with two nodes (produced current and voltage).

The above dataset is divided into two parts a) 70% training and 30% for testing and validation of the model. The available functions in NNs Toolboxes (e.g. *Trainlm*, *trainbr*, etc.) are used and tested. After several experiments by changing the number

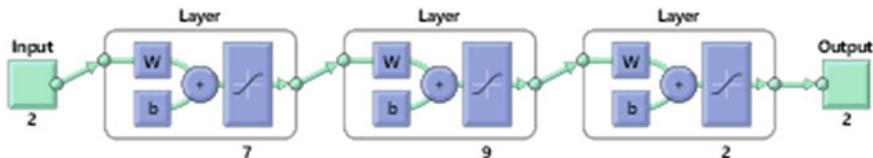


Fig. 7.8 The optimal MLP-PV module simulated under Matlab environment (*Trainlm*)

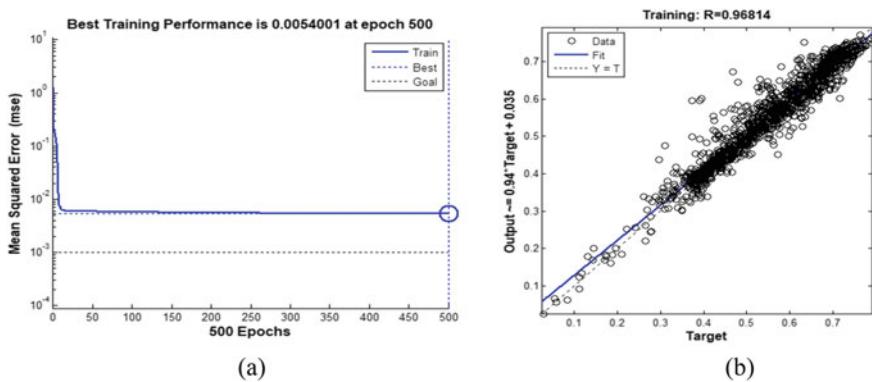


Fig. 7.9 **a** MSE and **b** correlation between neared and estimated dada

of hidden layers, neurons within the hidden layers and training algorithm. The good architecture is given for $2 \times 7 \times 9 \times 2$ as shown in Fig. 7.8.

Figure 7.9a shows the mean squared error (MSE) evolution during the training process. Figure 7.9 displays the correlation between measured and estimated, as can be observed good agreement is obtained (R is closer to 97%).

Once the optimal MLP configuration is selected (e.g., the number of hidden layers and the number neurons within the hidden layers), and the weights and bias are saved. The next step consists to implement this configuration into a reconfigurable FPGA. To do this, a VHDL code is developed. The first step consists to write the corresponding code of the neuron (basic element of the MLP network). So, the basic element (neuron) in the MLP can be presented by the following diagram of the neuron [7]. It consists mainly of three modules ROM, MAC and AF (see Fig. 7.10).

- **ROM** is used to store the weights (W_{ij}) and bias (b_i),
- **MAC** is a function that allow to calculate the following sum: $Y_k = \sum_{i=1}^N X_i W_{ij}$,
- **AF** is the activation function, for this example the *Tansig* function is selected which is given as: $Z_i = \frac{2}{1+\exp(-2Y_k)} - 1$

It should be pointed that MLP-based implementation of a PV module into a reconfigurable FPGA has some benefits: (1) design a miniature intelligent PV module, (2) real-time performance evaluation, and (3) involving less computational efforts.

The VHDL codes of the above modules are given in Appendix A. Figure 7.11a

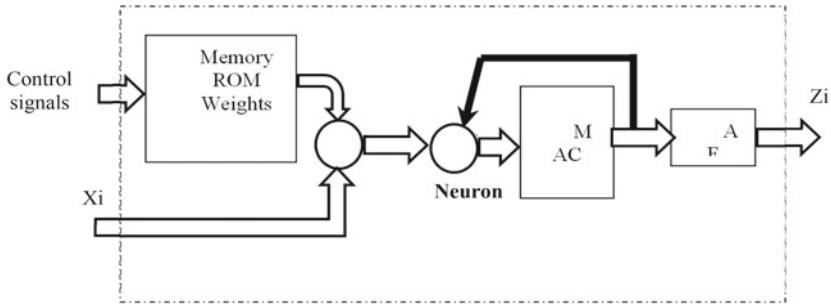


Fig. 7.10 Simplified schematic of the elementary neuron in the MLP network

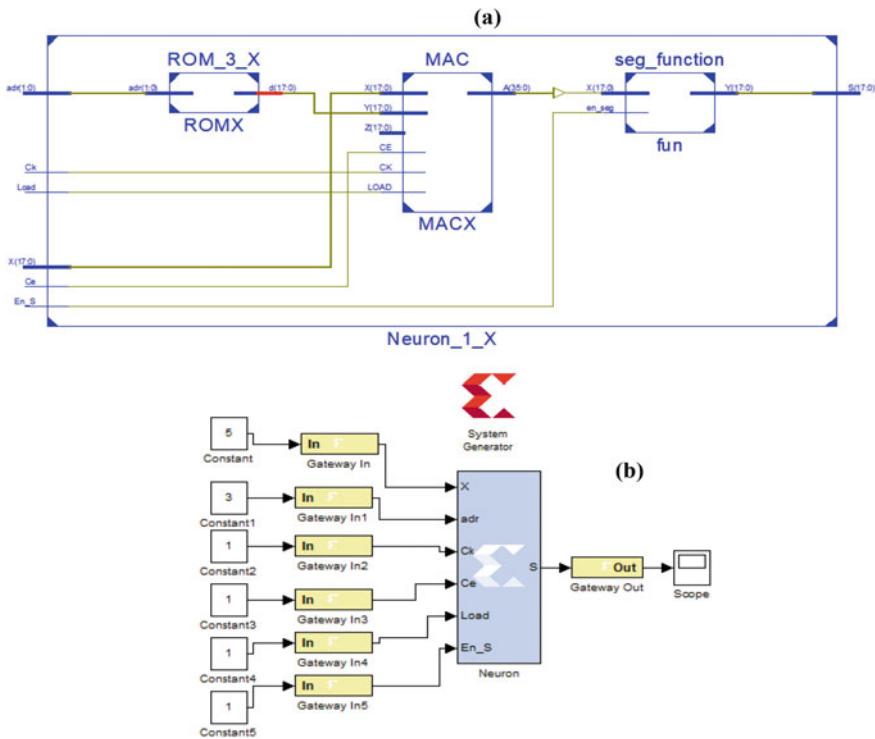


Fig. 7.11 **a** RTL schematic view of the neuron and **b** the corresponding XSG using block box function

shows the RTL view of the designed neuron under ISE Design Suite 14.7 and Fig. 7.11b displays the designed neuron based XSG using block function in Matlab/Simulink.

Figure 7.12 shows an example of a MLP based on the developed neuron in Fig. 7.8. It comprises two inputs, two hidden layers and one output.

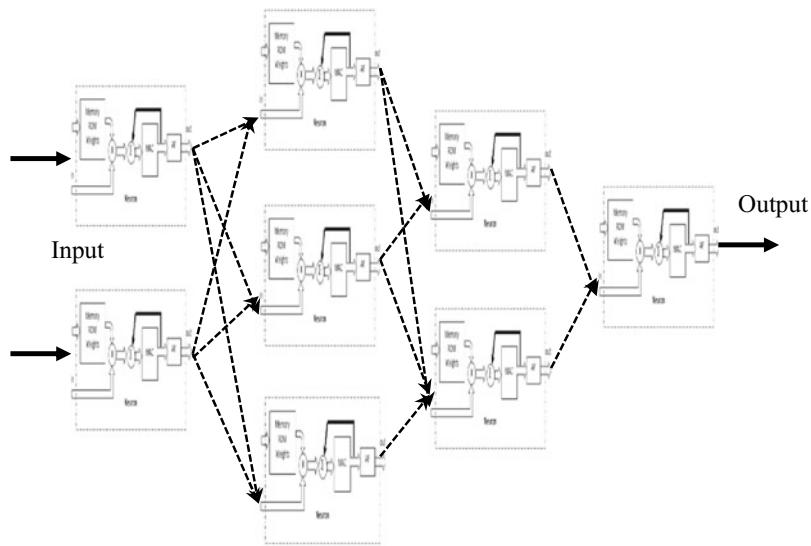


Fig. 7.12 MLP configuration $2 \times 3 \times 3 \times 1$ based on the developed basic element

Then, the next step consists to group the optimized MLP based on the basic element. In our cases the optimal configuration is $2 \times 7 \times 9 \times 2$, means 2 inputs, 7 neurons in the first hidden layer, 7 neurons in the second hidden layer and two neurons in the output layer. A controller (See Fig. 7.13) module is needed between all layers.

To simulate the designed model, we use ModelSim software. Figure 7.14 depicts the digital simulation of the designed MLP-PV module using the ModelSim simulator. As can be seen we have the input control signals: Ck, Ready and Reset. The input data are T and G , and the output data are: I_{pv} and V_{pv} .

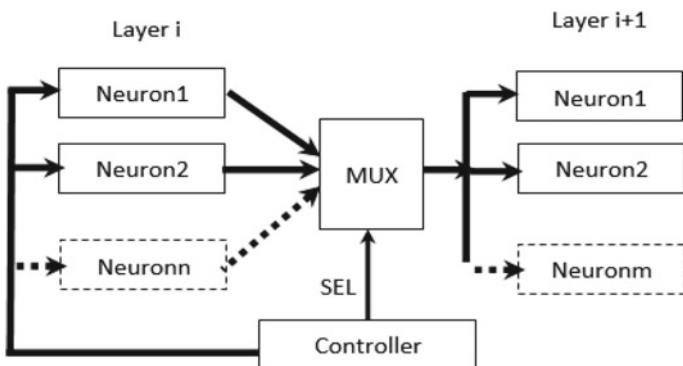


Fig. 7.13 Diagram of the neural controller

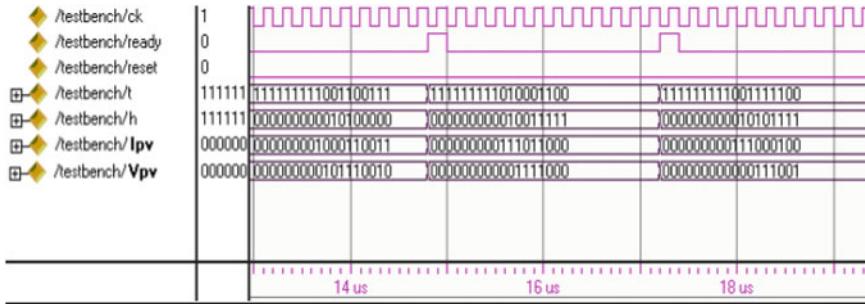


Fig. 7.14 Digital simulation of the designed MLP-PV under ModelSim simulator

Table 7.2 reports device utilization summary of the developed neuron-based model using Xilinx Virtex. The routing and placement of the neuron in the FPGA board is shown in Fig. 7.15a. The last step consists to download the code of the designed ANN-based PV model into the FPGA board (see Fig. 7.15b). The implemented model is now ready to be used.

In this case study, we have demonstrated the possibility to implement a static MLP network for PV module prediction. It should be pointed that the model was designed for a specific PV module and region, so the model is not dynamic (as it has a fixed structure, weights and bias cannot be updated throughout time). Generally, results are very promising and therefore to generalize this application, the following points should be considered for further improvement:

Table 7.2 FPGA device utilization summary

Logic utilization	Used	Available	Utilization
# of slice flip-flops	18	10,944	1%
# 4 inputs LUTs	261	10,944	2%
# of occupied slices	138	5,472	2%
# of slices containing only related logic	138	138	100%
# of slices containing only unrelated logic	0	138	0%
Total # 4 inputs LUTs	271	10,944	2%
# used as logic	261		
# used as route thru	10		
# of bonded IOBs	42	240	17%
IOB latches	18		
# of BUFGs/BUFGCTRLs	2	32	6%
# used as BUFGs	2		
# of DSP-48s	2	32	6%
Average Fanout Non clock-Netr	2.56		

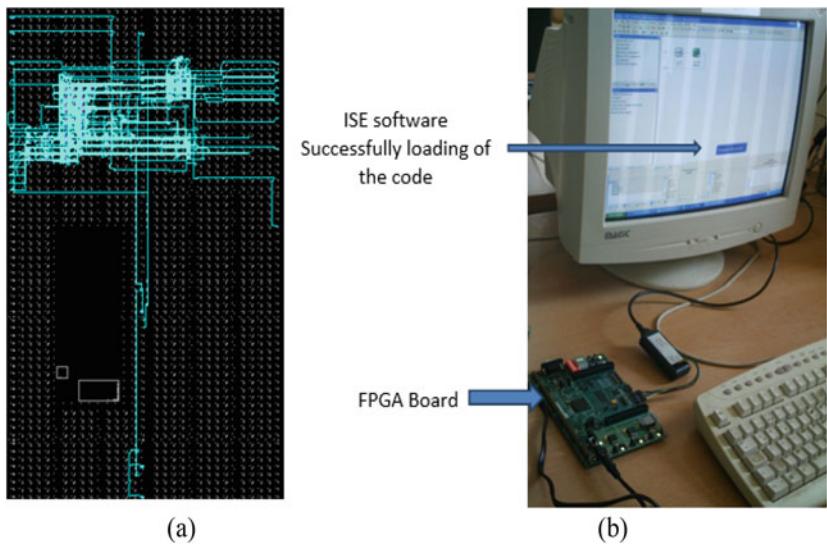


Fig. 7.15 **a** Routing and placement of the of the designed NN-based model using ISE, **b** loading the BitStream file of the developed MLP-PV module into a Xilinx Virtex II pro FPGA board

- Developing of dynamic NNs, so the parameters could be updated throughout time
- More PV modules of different technologies should be considered
- Using a large database at different regions.
- For fast development and rapid prototyping XSG is the most appropriate rather than VHDL coding (consuming time).
- PV modules degradation.

7.3.2 Case Study 2: *FPGA-Based Implementation of Irradiance Equalization Algorithm for Reconfigurable Photovoltaic*

The present case study aiming to demonstrate the hardware implementation of an equalization irradiance algorithm for reconfigurable PV architecture into a FPGA Altera DE2-70 board [8]. So, adaptive reconfiguration of PV array connections is one among the proposed solution in the literature, to reduce the effect of shading [9] (See Fig. 7.16).

The PV modules are continuously rearranged to allow to the photovoltaic system to operate as a constant source of energy, even if PV modules do not receive the same amount of irradiance. Irradiance equalization reconfiguration approach aims to form PVG rows with the same amount of irradiance in a TCT (Total-Cross-Tied) topology [10].



Fig. 7.16 Example of shaded PV modules in a PV array

The optimization algorithm searches all possible PVG configurations and for each configuration, the algorithm calculates the Equalization Index (EI) using the following expression: $EI = \max(G_i) - \min(G_i)$, G_i is the total solar irradiance of row i .

Example in Fig. 7.17 shows irradiance equalization example, before and after reconfiguration for a 3×3 PV array structure.

An example of optimal configuration searching phase for a 3×3 PV matrix is illustrated in Fig. 7.18.

The algorithm is designed using VHDL (See Appendix B). The University Program Vector WaveForm available in Intel Quartus software is used for the simulation purposes. The bloc diagram of the implemented algorithm into FPGA is shown in Fig. 7.19. As can be seen, it includes eight principal units: Irradiance data bloc: the ‘Slc’ signal allows us to choose between four sets of irradiance values. Data acquisition bloc, ensures a time interval between two consecutive sets of irradiance values. Block I, II, III, IV, V represent the step 2, 3, 4, 5, 6 of optimal configuration phase of proposed algorithms, respectively. Bloc VI: receives the final configuration and sends the appropriate driving signals (SP, SN) of the LEDs.

The simulation is performed for three irradiance matrices representing three different shading patterns ($Slc = 00; 01; 10$) and for matrix representing the case of uniform irradiance ($Slc = 11$). The simulated SN and SP of different ‘Slc’ values are shown in Fig. 7.20.

As can be seen, the designed controller reaches the optimal configuration and set the driving signal SP and SN in their suitable values. Furthermore, in the case of uniform irradiance, the controller puts SP and SN in their initial state, which confirms the accuracy of the developed code under VHDL. For processing of an acquired irradiance matrix by the controller, 32 clock-edge is required to find the proper values of SP and SN. For this reason, a frequency divider by 32 is used. Thus, rapid variation of irradiance during 32 clock-edge interval doesn't taken into account in order to not put the algorithm in an infinite loop, as the fourth case of slc in the Fig. 7.20. The summary of FPGA resources used by the reconfiguration controller is reported in Table 7.3. The results of the implementation on the FPGA DE2-70

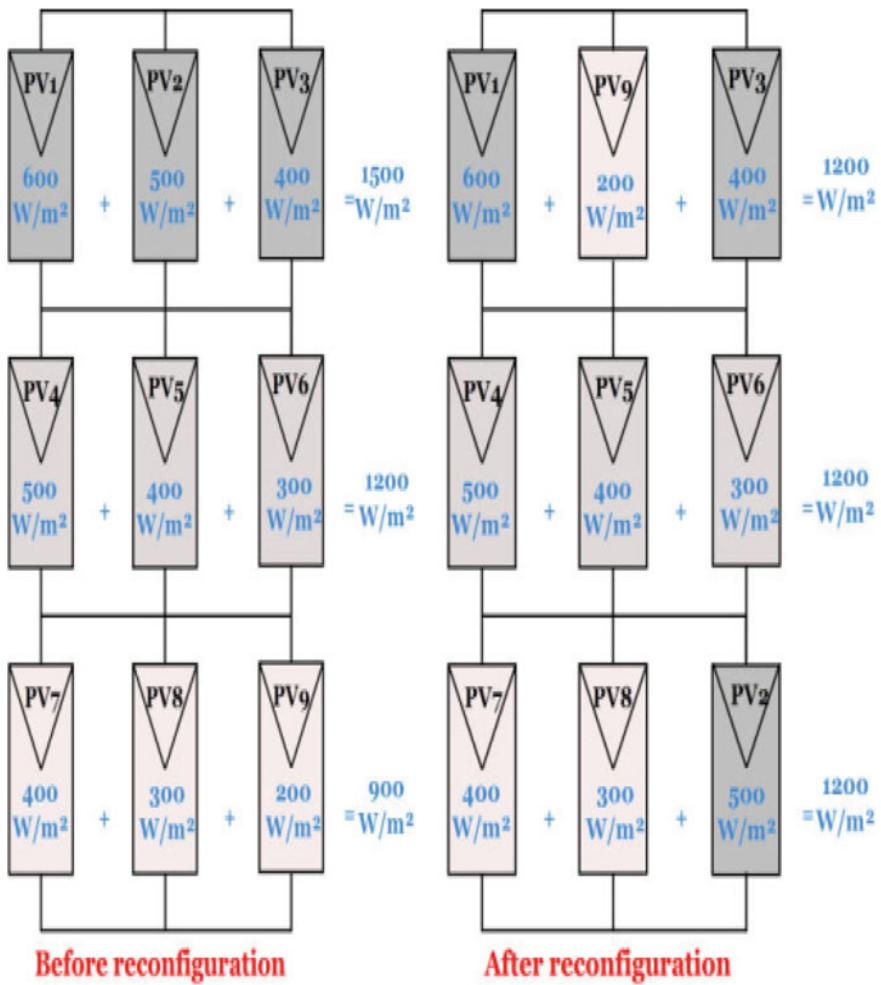


Fig. 7.17 Irradiance equalization example, before and after reconfiguration

card for the different cases of ‘Slc’ are illustrated in Fig. 7.21. As shown, for each irradiance matrix case, the appropriate LEDs are turned on.

In this application we have demonstrated the possible implementation of an irradiance equalization algorithm into FPGA Altera DE2-70 board for real time application. It has been shown that the designed algorithm reaches successfully the optimal configuration and the appropriate LEDs are turned on. The final step consists to validate experimentally the implemented algorithms.

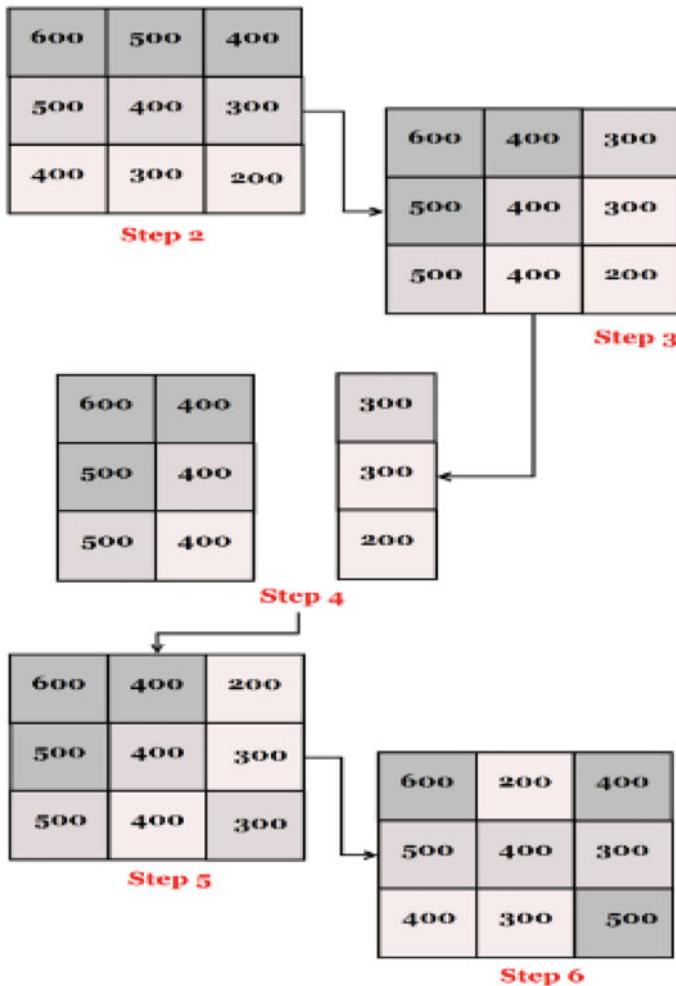


Fig. 7.18 Example of optimal configuration searching phase for a 3×3 PV matrix

7.3.3 Case Study 3: FPGA-Based Implementation of an MPPT Algorithm

Tracking the maximum power point (MPP) of a PV module/array is an essential task in a PV control system, since it maximizes the power output of the PV system, and therefore maximizes the PV module's efficiency. To enhance the conversion efficiency of the electric power generation a maximum power point tracking (MPPT) module (i.e., it consists of MPPT algorithm used to control a DC-DC converter) is usually integrated with the PV power installations so that the photovoltaic arrays will be able to deliver the maximum power available in real time under all possible

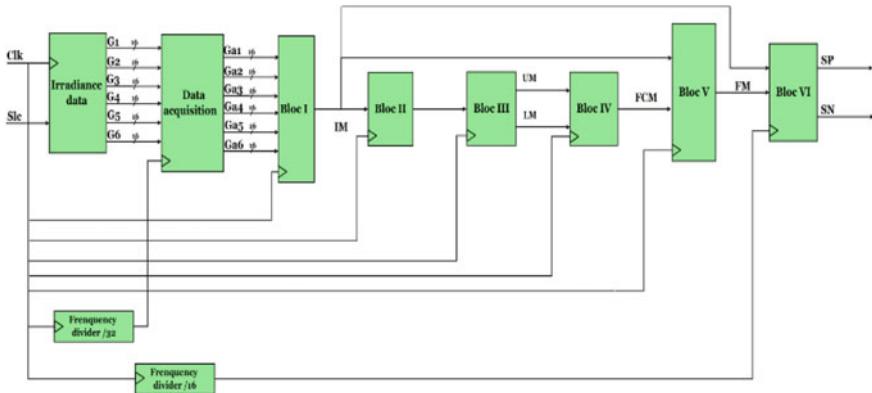


Fig. 7.19 Block diagram of the implemented algorithm into FPGA

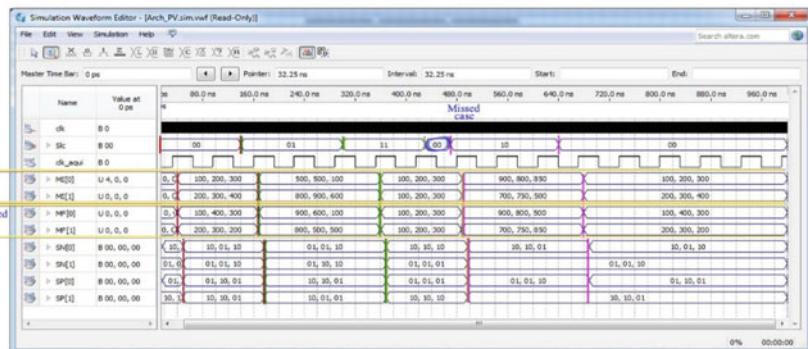


Fig. 7.20 Simulation results under university program VWF

Table 7.3 Switches and their associated FPGA

Resources	Used
Total combinational functions	1333
Total registers	442
I/O pins	220
Maximum frequency	59.56 MHz

system operating conditions [11]. Figure 7.22 shows the test facility used for this application. This experiment permits to produce the I-V curves under normal and abnormal conditions.

Figure 7.23 shows the power-voltage curves under uniform and non-uniform irradiance.

It can be seen that in the case of uniform irradiance only one global MPPT is observed, while in the case of non-uniform irradiance which is always associated

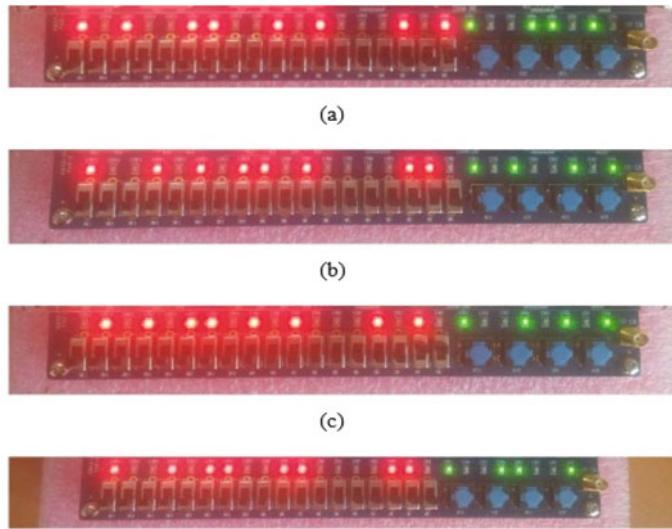


Fig. 7.21 The output LED for different values of S_{lc} **a** $S_{lc} = 01$, **b** $S_{lc} = 10$, **c** $S_{lc} = 11$, **d** $S_{lc} = 00$



Fig. 7.22 Test facility used for measuring I-V curves under uniform and non-uniform irradiance

with shading effect, we have multiples points and only one the global. Figure 7.24 shows a simplified diagram for tracking the MPP. It consists mainly of a PV module or generator, DC-DC converter and a load.

The next sub-sections aiming at providing two experiments about the FPGA implementation of MPPT using a simple algorithm named perturb and observe (P&O) and a relatively complex fuzzy-logic controller (FLC).

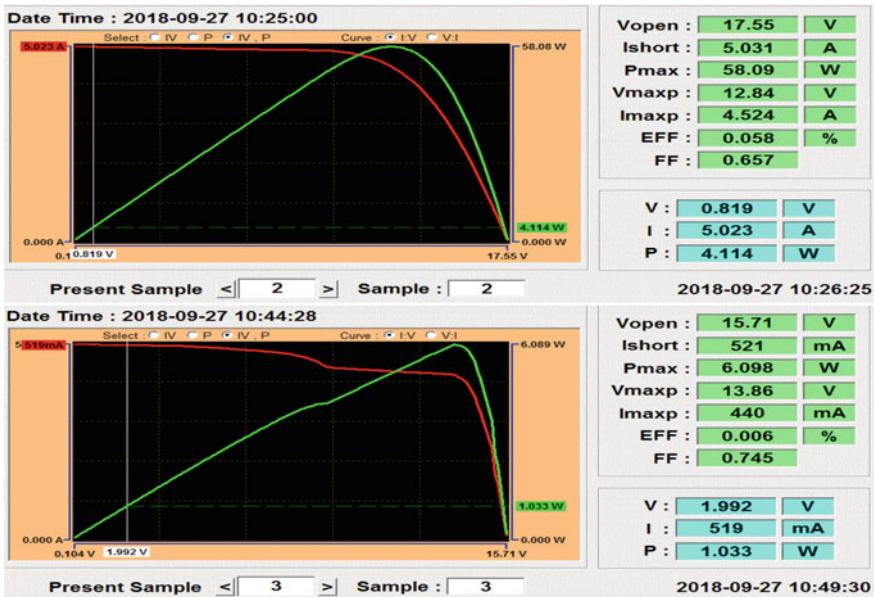


Fig. 7.23 P–V curve under uniform and non-uniform conditions

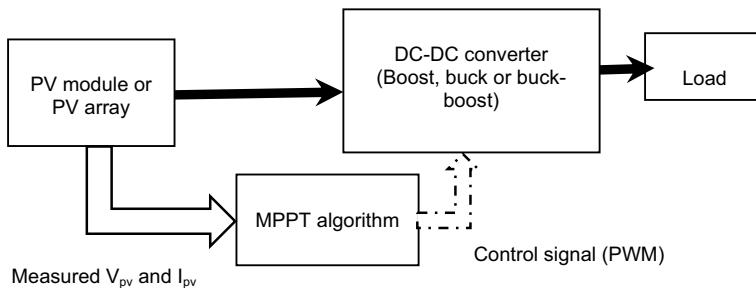


Fig. 7.24 Block diagram of MPPT in a PV system

P&O is commonly used and implemented due to its simplicity. The pseudo-code of the P&O algorithm is summarized as follows (see Listing 7.1):

```

1  Read  $V_{pv}(k)$ ,  $I_{pv}(k)$ ,  $V_{pv}(k-1)$  and  $I_{pv}(k-1)$ 
2  Initialize dD
3  calculate  $dP(k) = P_{pv}(k) - P_{pv}(k-1)$ ;  $dV_{pv}(k-1) = V_{pv}(k) - V_{pv}(k-1)$ 
4  If  $dP_{pv} > 0$  then
5    If  $dV_{pv} > 0$  then
6       $D(k) \leq D(k) + dD$ 
7    Else
8       $D(k) \leq D(k) - dD$ 
9  End if
10 Elsif  $dP_{pv} < 0$  then
11   If  $dV_{pv} > 0$  then
12      $D(k) \leq D(k) - dD$ 
13   Else
14      $D(k) \leq D(k) + dD$ 
15 End if

```

Listing 7.1 P&O MPPT algorithm

where dD is the variation step, D is duty cycle.

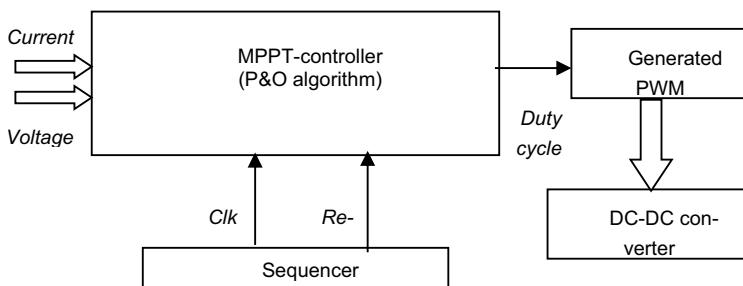
Figure 7.25 shows the different elements of the P&O MPPT algorithm. It consists of the MPPT-P&O algorithm, PWM:

- MPPT-P&O algorithm, for seeking the maximum power
- PWM, which is used to control DC-DC boost converter
- Sequencer module which is used to control different signal inside the MPPT.

The corresponding VHDL code of the P&O algorithm is given in the Appendix C. The RTL of the designed MPPT under ISE and the routing space are shown in Fig. 7.26.

Table 7.4 reports the resources used by the developed P&O algorithm.

As example, simulation results are displayed in Fig. 7.27 for a perturbation step of $dD = 0.5$, it can be clearly seen the developed algorithm converges quickly to

**Fig. 7.25** Different VHDL-modules for design the P&O MPPT algorithm

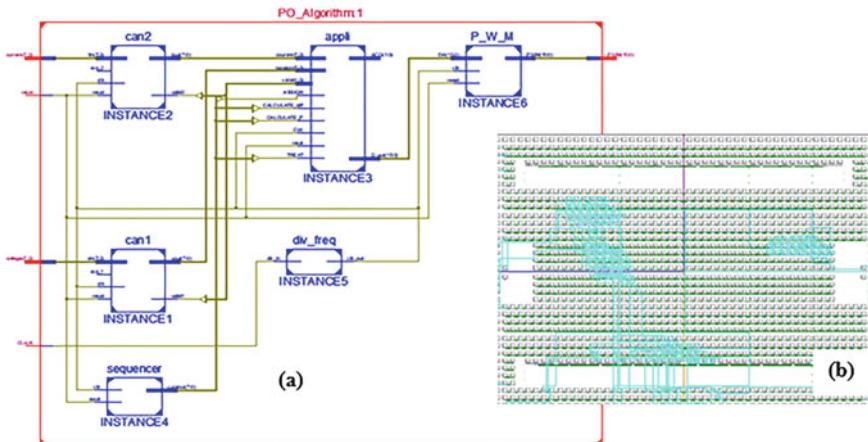


Fig. 7.26 **a** RTL schematic of the developed P&O algorithm, **b** used space by the designed P&O MPPT algorithm for a FPGA Spartan 3E

Table 7.4 FPGA device utilization summary

Logic utilization	Used	Available	Utilization (%)
# of slice flip-flops	172	9,312	1
# 4 inputs LUTs	283	9,312	3
# of slices	194	4,656	4
# of clocks	2	24	8
# of MULT18 × 18SIOs	1	20	5
# of bonded IOBs	34	66	51

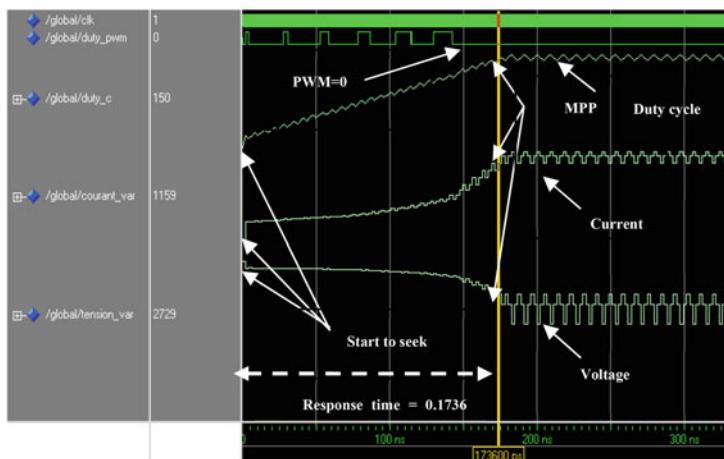


Fig. 7.27 Simulated results under ModelSim of the developed P&O algorithm

the maximal power point with a response time of 0.01736 ms. In addition, we can note that, once the MPP is reached the PWM signal takes the value zero and the system start to oscillate around the MPP between two extreme values P1 and P2. The oscillation around a maximum power point causes a power loss that depends on the step width of a single perturbation, these oscillations remain acceptable since they don't exceed 4.76%.

Figure 7.28a shows the co-simulation of the P&O MPPT under XSG and Fig. 7.28b the used FPGA board.

It should be noted that the ideal step width needs to be determined experimentally to follow the tradeoff of increased losses under stable or slowly changing conditions. A DC-DC boost converter is designed (See Fig. 7.29) in order to verify the implemented P&O in for time application.

To reduce the power loss caused by the oscillation, the duty step size needs to be adjusted dynamically according to different weather conditions. Longer sampling period can be used if weather condition is constant [12]. The possibility implementation of a simple MPPT is justified in this part. The efficiency of the implemented digital controller is about 96.13%. The total cost of the designed hardware is approximately (50 \$). It is obvious that there are compromises amongst the MPPT approaches

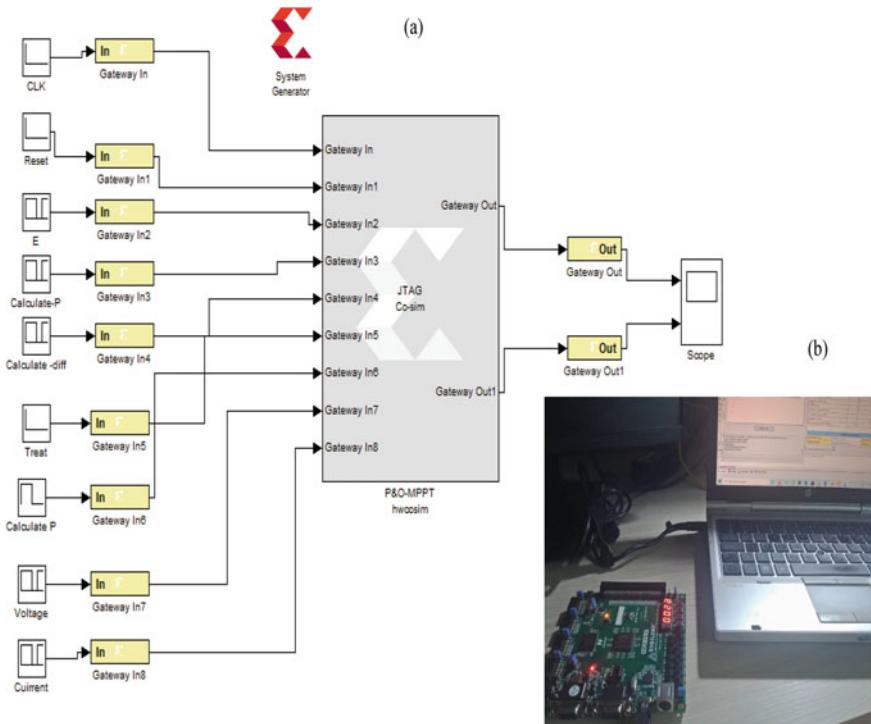


Fig. 7.28 **a** The co-simulation of P&O-MPPT model using XSG, **b** FPGA Spartan 3E Nexys 2 board

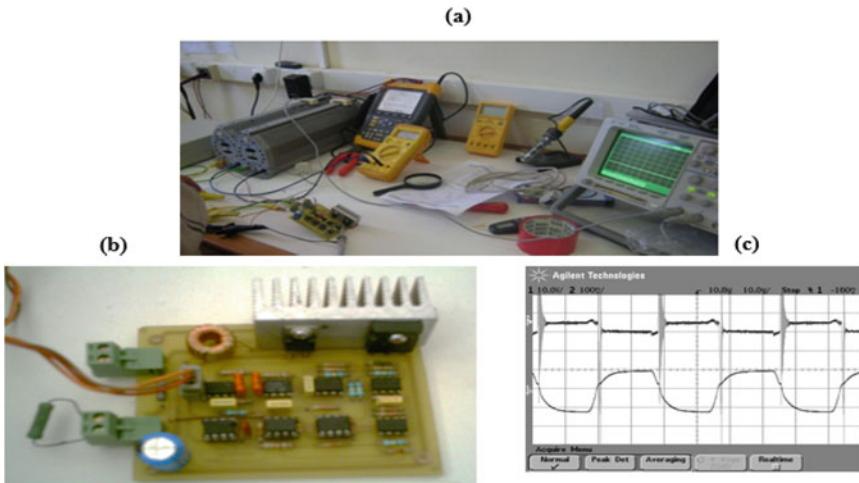


Fig. 7.29 **a** Test facility, **b** DC-DC boost converter with ADC, **c** example of the generated PWM

under dynamic atmospheric and partial shading conditions. Many improved evolutions were developed in the literature in order to improve the efficiency of the classical P&O by reducing the oscillation around the MPP.

7.3.4 Case Study 5: XSG-Based Implementation Control of Grid-Connected Hybrid System (PV-WT)

Grid-connected hybrid systems including photovoltaic and wind turbine (PV-WT), played very important role in electricity production. The RESs (PV and WT) exhibit nonlinear electric characteristics depending on the varying climatic conditions. Numerous MPPT methods [13] are proposed in the literature in order to enhance the energy generation efficiency of RESs. In this case study we present an overall control scheme of the PV-WT power system is established using the Xilinx System Generator (XSG) design tool.

The structure of the grid-connected hybrid system PV-WIN is shown in Fig. 7.30. It is composed of a PV array, a boost converter, a WTG that uses a Permanent Magnetic Synchronous Generator (PMSG), a rectifier; a three-phase DC-AC converter and RL filter [13].

The control blocks are MPPT + FCO, VFOC based Backstepping and Inc-MPPT. In this application we are motivated by the co-simulation of the IncCond-MPPT algorithm. More details about the rest controllers can be found in [12]. The IncCond MPPT is summarized in the following pseudo-code source (see Listing 7.2).

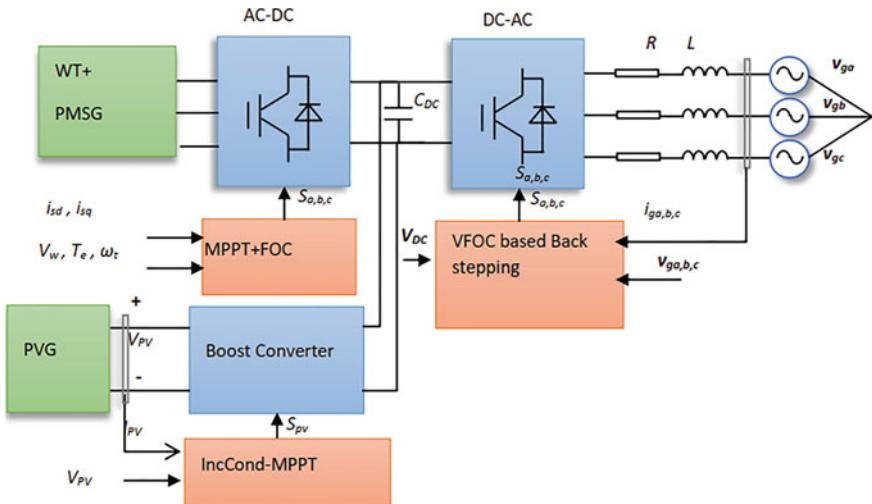


Fig. 7.30 The structure of the grid connected PV-WT system

1	Read $v(k)$; $I(k)$; $v(k-1)$ and $I(k-1)$; $e = (dI/dV) - (I/V)$
2	Initialize dD
3	Calculate $dI(k) = P(k) - P(k-1)$; $dV(k-1) = V(k) - V(k-1)$
4	If $dV > 0$ then
5	If $dI = 0$ then
6	$I(k) = I(k+1)$
7	$V(k) = V(k+1)$
7	Elsif $dI > 0$ then
8	$D(k) \leq D(k) + dD$
9	Else
10	$D(k) \leq D(k) + dD$
11	$I(k) = I(k+1)$
12	$V(k) = V(k+1)$
13	End if;
14	Elsif $e \geq 0$ then
15	$I(k) = I(k+1)$
16	$V(k) = V(k+1)$
17	Elsif $e > 0$ then
18	$D(k) \leq D(k) + dD$
19	Else
20	$D(k) \leq D(k) + dD$
21	$I(k) = I(k+1)$
22	$V(k) = V(k+1)$
23	End if
24	$I(k) = I(k+1); V(k) = V(k+1)$
25	End if
26	End if
27	Endif

Listing 7.2 IncCond MPPT algorithm

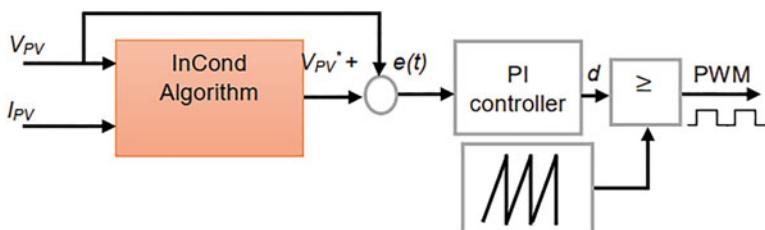


Fig. 7.31 The control scheme of the boost converter

Figure 7.31 illustrates the control scheme adopted for the boost converter. In order to reach the MPP of the PV source, the Perturb and Observe (P&O) controller adjusts the voltage reference (VPV^*) based on the measurement of PV current and voltage.

The XSG module of P&O algorithm, the PI voltage controller and the PWM modulator are shown in Fig. 7.32.

- The XSG co-simulation of the IncCond algorithm is shown in Fig. 7.32a
- The XSG bloc of the PI voltage controller is established by applying the backward Euler approximation method (see Fig. 7.32b).
- The PWM control signal of the boost converter is generated by comparing, at each iteration, the duty ratio value $d(k)$ with the value of a triangle signal. So, a 8 bits up-counter is used to generate an asymmetric triangle signal at the frequency of 2 kHz (see Fig. 7.32b). The counting sequence is limited between zero and 255 such that the sampling period is fixed to 2 μ s. the counting maximal value is determined according to the following formula [12]:

$$\text{Counting limit} = \text{required triangle period}/\text{sampling period} = 5E^{-4}/2E^{-6} = 250.$$

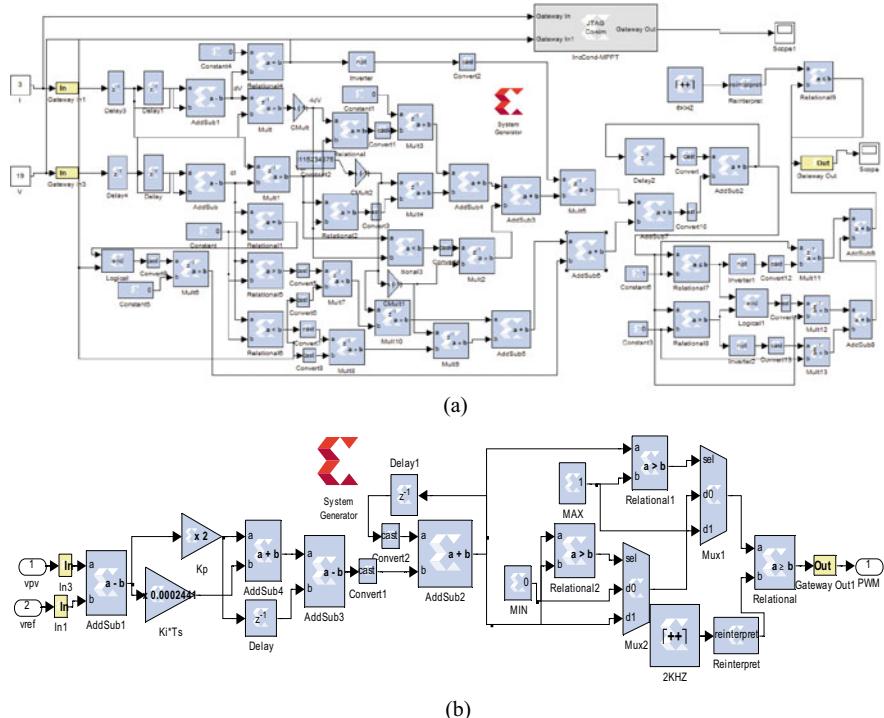


Fig. 7.32 The implementation based on the XSG of: **a** the InCond controller, **b** the PI controller with the PWM generator

The nearest integer to the value $250 = 2^8$; so 8 bits counter should be used. The Reinterpret bloc is used to obtain a normalized triangle signal at the output. Figure 7.33 shows the simulated results for the following test conditions ($G = 1000 \text{ W/m}^2$, $T = 25^\circ\text{C}$ and $W_s = 12 \text{ m/s}$). It can be seen that the XSG based control blocs work correctly and ensure a correct operation.

The MPPT controller InCond determine correctly the MPP of the PVG and WTG as exhibited in Fig. 7.33. Figure 7.34 shows the IncCond co-simulation block and the used FPGA board (ZYBO Zynq-7000).

Figure 7.34 shows the ZYBO-Zynq 7000 FPGA board used for the co-simulation of the XSG/Simulink IncCond algorithm. The XSG/Simulink simulation is accomplished in order to verify the correctness and the validity of the developed control circuit. The simulation results show satisfactory static and dynamic performance for

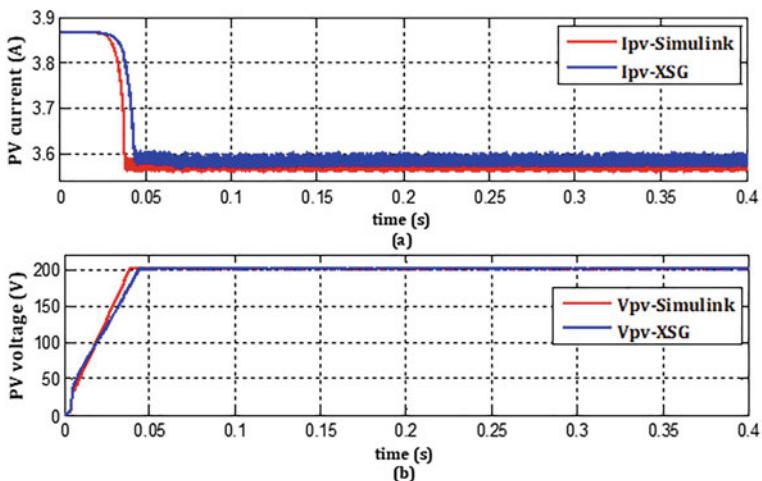


Fig. 7.33 Simulation results: Ipv and Vpv simulated by Matlab/Simulink and XSG

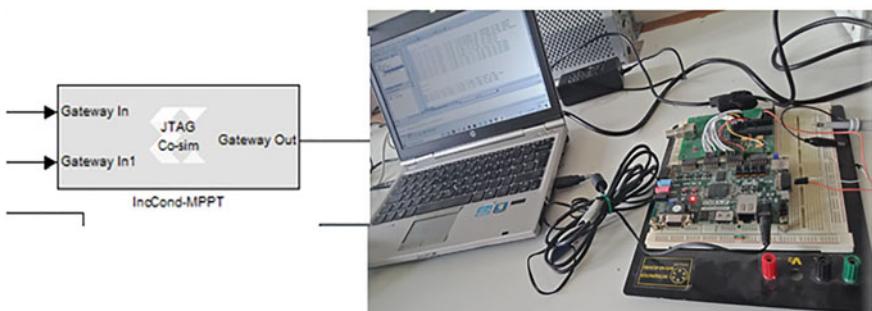


Fig. 7.34 FPGA board (ZYBO Zynq-7000) used for the co-simulation of the XSG/Simulink model

different operating conditions. At this stage the designed controller is ready to be implemented into a FPGA board for real time application.

7.4 Summary

In this chapter, we have presented and discussed in details some case studies of FPGA applications in renewable energy systems, including photovoltaic modules, photovoltaic arrays and hybrid PV systems (e.g. wind-photovoltaic). The covered applications are: (1) implementation of photovoltaic module simulator using ANNs, (2) design of an automatic reconfiguration method for photovoltaic arrays, (3) tracking the maximum power points using classical algorithms (e.g. P&O) and (4) Control of hybrid wind-photovoltaic systems (IncCond and Back stepping).

Two ways are presented to implement algorithms into FPGA boards: using hardware language (e.g. VHDL or Verilog), or using Xilinx System Generator based Matlab-Simulink. We recommend readers to use the second one, which is the most suitable for fast prototyping. In addition, using recent developed FPGA (boards) and tools (software) allow design teams and engineers to spend less time developing the structure of such application and more time building differentiating features into the end application. Opportunities of using such reconfigurable programmable (FPGA) devices to design photovoltaic and hybrid energy application has been shown.

References

1. <https://www.irena.org>
2. Seals RC, Whaphott GF (1997) Programmable logic: PLDs and FPGAs. Macmillan International Higher Education
3. <https://www.ee.co.za/article/can-fpgas-renewable-energy-system.html>
4. Pellerin D, Thibault S (2005) Practical FPGA programming in C. Prentice Hall Press
5. Livingstone DJ (ed) (2008) Artificial neural networks: methods and applications. Humana Press, Totowa, NJ, USA, pp 185–202
6. Alqahtani A, Alsaffar M, El-Sayed M, Alajmi B (2016) Data-driven photovoltaic system modeling based on nonlinear system identification. Int J Photoenergy
7. Mekki H, Mellit A, Kalogirou SA, Messai A, Furlan G (2010) FPGA-based implementation of a real time photovoltaic module simulator. Prog Photovoltaics Res Appl 18(2):115–127
8. Bouselham L, Hajji B, Mellit A, Rabhi A, Kassmi K (2019) Hardware implementation of new irradiance equalization algorithm for reconfigurable PV architecture on a FPGA platform. In 2019 international conference on wireless technologies, embedded and intelligent systems (WITS). IEEE, pp 1–8
9. Das SK, Verma D, Nema S, Nema RK (2017) Shading mitigation techniques: state-of-the-art in photovoltaic applications. Renew Sustain Energy Rev 78:369–390
10. Velasco G, Negroni JJ, Guinjoan F, Pique R (2005) Irradiance equalization method for output power optimization in plant oriented grid-connected PV generators. In: 2005 European conference on power electronics and applications. IEEE, 10 pp

11. Mellit A, Kalogirou SA (2014) MPPT-based artificial intelligence techniques for photovoltaic systems and its implementation into field programmable gate array chips: review of current status and future perspectives. *Energy* 70:1–21
12. Chettibi N, Mellit A (2019) Study on control of hybrid photovoltaic-wind power system using Xilinx system generator. *Solar photovoltaic power plants*. Springer, Singapore, pp 97–120
13. Selvamuthukumaran R, Gupta R (2014) Rapid prototyping of power electronics converters for photovoltaic system application using Xilinx system generator. *IET Power Electron* 7(9):2269–2278

Appendix A

VHDL Codes of MLP Modules

MAC Function

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
entity MAC is
    Port ( X,Y,Z : in std_logic_vector(17 downto 0);
            A : out std_logic_vector(35 downto 0);
            CK,CE,LOAD : in std_logic);
end MAC;
architecture Behavioral of MAC is
signal ACC:std_logic_vector(35 downto 0);
signal XY :std_logic_vector(35 downto 0);
signal Z0 :std_logic_vector(35 downto 0);
begin
    Z0<=Z(17)&Z(17)&Z(17)&Z(17)&Z(17)&Z(17)&Z(17)&Z(17)&Z&'0'&
    X"00";
    XY<=X*Y;
process(ck,ce,Z0,XY)
begin
    if falling_edge(CK)and (CE='1') then
        if (LOAD='1')then
            ACC<= Z0;
        else
            ACC<=ACC+XY;
        end if;
    else
        ACC<=ACC;
    end if;
end process;
A<=ACC;
end Behavioral;
```

Activation Function

<pre> library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.STD_LOGIC_ARITH.ALL; use IEEE.STD_LOGIC_SIGNED.ALL; entity seg_function is Port (X : in std_logic_vector(17 downto 0); Y : out std_logic_vector(17 downto 0); en_seg : in std_logic); end seg_function; --Segment Architecture Behavioral of seg_function is constant a0:integer:=508; constant a1:integer:=473; constant a2:integer:=414; constant a3:integer:=342; constant a4:integer:=270; constant a5:integer:=205; constant a6:integer:=152; constant a7:integer:=110; constant a8:integer:=78; constant a9:integer:=55; constant aa:integer:=39; constant ab:integer:=27; constant ac:integer:=19; constant ad:integer:=13; constant ae:integer:=9; constant af:integer:=5; constant b0:integer:=0; constant b1:integer:=7; constant b2:integer:=29; constant b3:integer:=70; constant b4:integer:=124; constant b5:integer:=185; constant b6:integer:=244; constant b7:integer:=299; constant b8:integer:=347; constant b9:integer:=386; constant ba:integer:=416; constant bb:integer:=441; </pre>	<pre> constant bd:integer:=473; constant be:integer:=483; constant bf:integer:=495; --Interval limits constant x0:integer:=0; constant x1:integer:=96; constant x2:integer:=192; constant x3:integer:=288; constant x4:integer:=384; constant x5:integer:=480; constant x6:integer:=576; constant x7:integer:=672; constant x8:integer:=768; constant x9:integer:=864; constant xa:integer:=960; constant xb:integer:=1056; constant xc:integer:=1152; constant xd:integer:=1248; constant xe:integer:=1344; constant xf:integer:=1440; constant x10:integer:=1792; signal Abs_X: std_logic_vector(17 downto 0); signal LATCH:std_logic_vector(17 downto 0); signal Y_S:std_logic_vector(17 downto 0); signal Abs_Y:std_logic_vector(35 downto 0); signal a,b:std_logic_vector(17 downto 0); signal bx :std_logic_vector(26 downto 0); </pre>
--	--

```

begin
abs_X<=abs(X);
-- when X(17)='1' else abs_X<=X;
cmp00:process(abs_X)
begin
if (abs_x>= X0)and (abs_x< X1) then
  a<=CONV_std_logic_vector(a0,18);b<=CONV_std_logic_vector(b0,18);
elsif (abs_x< X2) then
  a<=CONV_std_logic_vector(a1,18);b<=CONV_std_logic_vector(b1,18);
  elsif (abs_x< X3) then
    a<=CONV_std_logic_vector(a2,18);b<=CONV_std_logic_vector(b2,18);
    elsif (abs_x< X4) then
      a<=CONV_std_logic_vector(a3,18);b<=CONV_std_logic_vector(b3,18);
      elsif (abs_x< X5) then
        a<=CONV_std_logic_vector(a4,18);b<=CONV_std_logic_vector(b4,18);
        elsif (abs_x< X6) then
          a<=CONV_std_logic_vector(a5,18);b<=CONV_std_logic_vector(b5,18);
          elsif (abs_x< X7) then
            a<=CONV_std_logic_vector(a6,18);b<=CONV_std_logic_vector(b6,18);
            elsif (abs_x< X8) then
              a<=CONV_std_logic_vector(a7,18);b<=CONV_std_logic_vector(b7,18);
              elsif (abs_x< X9) then
                a<=CONV_std_logic_vector(a8,18);b<=CONV_std_logic_vector(b8,18);
                elsif (abs_x< XA) then
                  a<=CONV_std_logic_vector(a9,18);b<=CONV_std_logic_vector(b9,18);
                  elsif (abs_x< XB) then
                    a<=CONV_std_logic_vector(aa,18);b<=CONV_std_logic_vector(ba,18);
                    elsif (abs_x< XC) then
                      a<=CONV_std_logic_vector(ab,18);b<=CONV_std_logic_vector(bb,18);
                      elsif (abs_x< XD) then
                        a<=CONV_std_logic_vector(ac,18);b<=CONV_std_logic_vector(bc,18);
                        elsif (abs_x< XE) then
                          a<=CONV_std_logic_vector(ad,18);b<=CONV_std_logic_vector(bd,18);
                          elsif (abs_x< XF) then
                            a<=CONV_std_logic_vector(ae,18);b<=CONV_std_logic_vector(be,18);
                            elsif (abs_x< X10)then
                              a<=CONV_std_logic_vector(af,18);b<=CONV_std_logic_vector(bf,18);
                              else
                                a<=(others=>'0');
                                b<=CONV_std_logic_vector(512,18);
                                end if;
                                end process;

```

```

bx(8 downto 0)<=X"00"&"0";
bx(26 downto 9)<=b;
ABS_Y<=a*abs_X+bx;
Sym_out:process(x,ABS_Y)
begin
    if x(17)='1' then
        Y_S<="00"&X"0000"-abs_Y( 26 downto 9);
    else
        Y_S<=abs_Y( 26 downto 9);
    end if;
    end process;
LATCHED_0: LATCH<=Y_S when (en_Seg='1') else LATCH;
LATCHED_OUT: Y<=LATCH;
end Behavioral;

```

ROM Function

<pre> library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.STD_LOGIC_ARITH.ALL; use IEEE.STD_LOGIC_SIGNED.AL L; entity ROM_3_X is generic (constant x0:integer:=10; constant x1:integer:=20; constant x2:integer:=20; constant Size: Natural:=32); port(adr: in std_logic_vector (1 downto 0); d: out std_logic_vector(Size-1 downto 0)); end ROM_3_X; architecture Behavioral of ROM_3_X is begin process (adr) </pre>	<pre> when "00" => d<=CONV_std_logic_vector(x0,Size);wh en "01" => d<=CONV_std_logic_vector(x1,Size); when "10" => d<=CONV_std_logic_vector(x2,Size); when others => d<=(others=>'0') ; when "010" => d<=CONV_std_logic_vector(x2,Size); when "011" => d<=CONV_std_logic_vector(x3,Size); when "100" => d<=CONV_std_logic_vector(x4,Size); when "101" => d<=CONV_std_logic_vector(x5,Size); when "110" => d<=CONV_std_logic_vector(x6,Size); when others =>d<=(others=>'0') ; end case; end process; end Behavioral; </pre>
--	--

<pre> begin case adr is </pre>	<pre> d<=CONV_std_logic_vector(x6,Size); end case; end process; end Behavioral; library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.STD_LOGIC_ARITH.ALL; use IEEE.STD_LOGIC_SIGNED.ALL; entity ROM_12_X is generic (constant x0:integer:=10; constant x1:integer:=20; constant x2:integer:=10; constant x3:integer:=20; constant x4:integer:=10; constant x5:integer:=20; constant x6:integer:=10; constant x7:integer:=20; constant x8:integer:=10; constant x9:integer:=10; constant xA:integer:=10; constant xB:integer:=10; constant Size: Natural:=32); port(adr: in std_logic_vector(3 downto 0); d: out std_logic_vector(Size-1 downto 0)); end ROM_12_X; architecture Behavioral of ROM_12_X is begin process (adr) begin case adr is when X"0" => d<=CONV_std_logic_vector(x0,Size); when X"1" => d<=CONV_std_logic_vector(x1,Size); when X"2" => d<=CONV_std_logic_vector(x2,Size); when X"3" => d<=CONV_std_logic_vector(x3,Size); when X"4" => d<=CONV_std_logic_vector(x4,Size); </pre>
	<pre> library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.STD_LOGIC_ARITH.ALL; use IEEE.STD_LOGIC_SIGNED.ALL; entity ROM_7_X is generic (constant x0:integer:=10; constant x1:integer:=20; constant x2:integer:=10; constant x3:integer:=20; constant x4:integer:=10; constant x5:integer:=20; constant x6:integer:=10; constant x7:integer:=20; constant x8:integer:=10; constant x9:integer:=10; constant xA:integer:=10; constant xB:integer:=10; constant Size: Natural:=32); port(adr: in std_logic_vector(2 downto 0); d: out std_logic_vector(Size-1 downto 0)); end ROM_7_X; architecture Behavioral of ROM_7_X is begin process (adr) begin case adr is when "000" => d<=CONV_std_logic_vector(x0,Size); when "001" => d<=CONV_std_logic_vector(x1,Size); </pre>

when X"5" => d<=CONV_std_logic_vector(x5,Size); when X"6" =>	end case; end process; end Behavioral;
--	--

Appendix B

VHDL Codes of the Interconnection Reconfiguration Algorithm Between the PV Panels

Main Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE work.Tri_pkg.ALL;
entity Arch_PV is
port
( clk      : IN std_logic;
  Slc      : IN std_logic_vector (1 downto 0);
  clk_acqui : OUT std_logic;
  Gst,msf   : OUT Matrix;
  SP,SN     : OUT Matrix_swh);
end entity Arch_PV;
Architecture behav_Arch_PV of Arch_PV is
----- real values of G-----
component data
port
( Slc          : IN std_logic_vector (1 downto 0);
  G1,G2,G3,G4,G5,G6 : OUT std_logic_vector(15 DOWNTO 0));
end component;
```

```

----- Data acquisition -----
component Acquis_G
port
( clk           : in std_logic;
  G1,G2,G3,G4,G5,G6   : IN std_logic_vector(15 DOWNTO 0);
  Ga1,Ga2,Ga3,Ga4,Ga5,Ga6 : OUT std_logic_vector(15 DOWNTO 0));
end component;

----- Block_1 -----
component bloc_1
port (
  G1,G2,G3,G4,G5,G6   : IN std_logic_vector(15 DOWNTO 0);
  hor          : IN std_logic;
  Go           : OUT Matrix);
end component;

----- Block_2 -----
component bloc_2
PORT(
  Go          : IN Matrix;
  Sm          : OUT Sub_MATRIX;
  clk         : IN std_logic;
  Vec         : OUT Sub_Vec);
end component;

----- Block_3 -----
component bloc_3
PORT( clk       : IN std_logic;
      V        : IN Sub_Vec;
      S        : IN Sub_MATRIX;
      MF       : OUT MATRIX );
END component;

----- initial matrix of G -----
component MI_sans_Tri
PORT(
  G1       : IN std_logic_vector(15 DOWNTO 0);
  G2       : IN std_logic_vector(15 DOWNTO 0);
  G3       : IN std_logic_vector(15 DOWNTO 0);
  G4       : IN std_logic_vector(15 DOWNTO 0);
  G5       : IN std_logic_vector(15 DOWNTO 0);
  G6       : IN std_logic_vector(15 DOWNTO 0);
  clk      : IN std_logic;
  Gst     : OUT Matrix
);
END component;

```

```
----- Block_4-----
component bloc4_compare
PORT( clk           : IN  std_logic;
      MSF          : IN  Matrix;
      MI           : IN  Matrix;
      MF           : OUT Matrix
 );
END component;
----- Block_5-----
component bascule_matrix
port
(
  clk        : in std_logic;
  d          : IN matrix;
  q          : out matrix
);
end component;
----- Block_6-----
component Swh_activate
PORT(
  clk         : IN  std_logic;
  Slc         : IN  std_logic_vector (1 downto 0) ;
  MI,MF       : IN  Matrix;
  SP,SN       : OUT Matrix_swh);
end component;
----- frequency divider /16-----
component div_16
port(
  clk         : in STD_LOGIC;
  out_clk     : out STD_LOGIC
 );
end component;
----- frequency divider /32-----
component div_32
port(
  clk         : in STD_LOGIC;
  out_clk     : out STD_LOGIC
 );
end component;
```

```

----- signal declaration-----
signal      G1x,G2x,G3x,G4x,G5x,G6x,G1s,G2s,G3s,G4s,G5s,G6s;
std_logic_vector(15 DOWNTO 0);
signal Sm_x          : Sub_MATRIX;
signal Vec_x          : Sub_Vec;
signal clk_8,clk_16,clk_32      : std_logic;
signal Go_x,Gst_x,msf_x,Gstt,msff,mfx : Matrix;
begin
C1: div_32      port map (clk,clk_32);
C11:div_8       port map (clk,clk_8);
C6: Data        port map (slc,G1s,G2s,G3s,G4s,G5s,G6s);
C5:   Acquis_G           port      map
(clk_32,G1s,G2s,G3s,G4s,G5s,G6s,G1x,G2x,G3x,G4x,G5x,G6x);
C4: bloc_1       port map (G1x,G2x,G3x,G4x,G5x,G6x,clk,Go_x);
C0: bloc_2       port map (Go_x,Sm_x,clk,Vec_x);
C2: bloc_3       port map (clk,Vec_x,Sm_x,msf_x);
C13: bloc4_compare port map (clk,msf_x,Gst_x,MFx);
C7: MI_sans_Tri  port map (G1x,G2x,G3x,G4x,G5x,G6x,clk_8,Gst_x);
C10: div_16      port map (clk,clk_16);
C8: bascule_matrix port map (clk_16,Gst_x,Gstt);
C9: bascule_matrix port map (clk_16,MFx,msff);
C12: Swh_activate port map (clk_16,slc,Gstt,msff,SP,SN);
clk_acqui <= clk_32;
Sortie1: FOR m IN 0 TO 1 GENERATE
  Sortie2: FOR n IN 0 TO 2 GENERATE
    gst(m,n) <= gst_x(m,n);
    msf(m,n) <= mfx(m,n);
  END GENERATE;
END GENERATE;
end behav_Arch_PV;

```

Package Definition

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

PACKAGE Tri_pkg IS

    TYPE vector_of_std_logic_vector10 IS ARRAY (0 to 5) OF unsigned(15
DOWNTO 0);

    TYPE vector_of_unsigned10 IS ARRAY (0 to 5) OF unsigned(15 DOWNTO
0);

    TYPE MATRIX is array (0 to 1, 0 to 2) of unsigned(0 to 15);

    TYPE MATRIX_Fixe IS ARRAY (0 to 1,0 to 1) of unsigned(15 DOWNTO
0);

    TYPE Vec IS ARRAY (0 to 1) of unsigned(0 to 15);
    TYPE Sub_MATRIX is array (0 to 1, 0 to 1) of unsigned(0 to 15);
    TYPE Sub_Vec is array (0 to 1) of unsigned(0 to 15);
    TYPE vector_of_unsigned9 IS ARRAY (NATURAL RANGE <>) OF un-
signed(15 DOWNTO 0);

    TYPE Matrix_swh is array (0 to 1, 0 to 2) of unsigned(1 DOWNTO 0);
    TYPE Matrix_idx is array (0 to 1, 0 to 2) of unsigned(2 DOWNTO 0);

    TYPE vector_of_std_logic_vector9 IS ARRAY (NATURAL RANGE <>)
OF std_logic_vector(15 DOWNTO 0);
    TYPE vector_of_signed32 IS ARRAY (NATURAL RANGE <>) OF
signed(31 DOWNTO 0);
```

```
    TYPE vector_of_unsigned2 IS ARRAY (NATURAL RANGE <>) OF un-
signed(1 DOWNTO 0);
```

```
END Tri_pkg
```

Data Acquisition

<pre> library IEEE; use IEEE.STD_LOGIC_1164.ALL; entity Acquis_G is port (clk : in std_logic; G1,G2,G3,G4,G5,G6: IN std_logic_vector(15 DOWNTO 0); Ga1,Ga2,Ga3,Ga4,Ga5,Ga6 : OUT std_logic_vector(15 DOWNTO 0)); end entity Acquis_G; ----- composable: Bascule_D----- component bascule port (clk : in std_logic; d : IN std_logic_vector(15 DOWNTO 0); q : out std_logic_vector(15 DOWNTO 0)); end component; signal hor_x6: std_logic; signal G1a,G2a,G3a,G4a,G5a,G6a: std_logic_vector(15 DOWNTO 0); begin C1:bascule port map (clk,G1,Ga1); C2:bascule port map (clk,G2,Ga2); C3:bascule port map (clk,G3,Ga3); C4:bascule port map (clk,G4,Ga4); C5:bascule port map (clk,G5,Ga5); C6:bascule port map (clk,G6,Ga6); end architecture Behav_G ; </pre>	<pre> library IEEE; use IEEE.STD_LOGIC_1164.ALL; entity bascule is port (clk : in std_logic; d : IN std_logic_vector(15 DOWNTO 0); architecture Behavioral of bascule is begin process (clk,d) is begin if rising_edge(clk) then q <= d; end if; end process; end architecture Behavioral; </pre>
--	--

Frequency Dividers

<pre> library IEEE; use IEEE.STD_LOGIC_1164.all; entity div_32 is port(clk : in STD_LOGIC; out_clk : out STD_LOGIC); end div_32; architecture arch_div_32 of div_32 is begin divider : process (clk) is variable m : integer range 0 to 64 := 0; begin if (rising_edge (clk)) then m := m + 1; end if; if (m=64) then m := 0; end if; if (m<32) then out_clk <= '0'; else out_clk <= '1'; end if; end process divider; end arch_div_32; </pre>	<pre> library IEEE; use IEEE.STD_LOGIC_1164.all; entity div_16 is port(clk : in STD_LOGIC; out_clk : out STD_LOGIC); end div_16; architecture arch_div_16 of div_16 is begin divider : process (clk) is variable m : integer range 0 to 32 := 0; begin if (rising_edge (clk)) then m := m + 1; end if; if (m=32) then m := 0; end if; if (m<16) then out_clk <= '0'; else out_clk <= '1'; end if; end process divider; end arch_div_16; </pre>
---	--

Bloc I

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;
USE work.Tri_pkg.ALL;
----- main entity -----
entity bloc_1 is port (
G1,G2,G3,G4,G5,G6      :      IN
std_logic_vector (15 DOWNTO 0);
hor   : IN std_logic;
Go    : OUT Matrix);
end bloc_1;
architecture arch_bloc_1 of bloc_1 is
component Tri_matlab
port (
G1  :  IN  std_logic_vector(15
DOWNTO 0);
G2  :  IN  std_logic_vector(15
DOWNTO 0);
G3  :  IN  std_logic_vector(15
DOWNTO 0);
G4  :  IN  std_logic_vector(15
DOWNTO 0);

```

G5 : IN std_logic_vector(15
DOWNTO 0);
G6 : IN std_logic_vector(15
DOWNTO 0);
clk : IN std_logic;
Gg: OUT vector_of_std_logic_vec-
tor10);
end component ;
component VecToMatrix
port (
Gg : IN vector_of_std_logic_vec-
tor10;
clk : IN std_logic;
Gm : OUT Matrix);
end component ;
signal G_X: vector_of_std_logic_vec-
tor10;
signal hor_X:std_logic;
begin
C1: Tri_matlab port
map(G1,G2,G3,G4,G5,G6,hor,G_X);
C3: VecToMatrix port map
(G_X,hor,Go);
end arch_bloc_1;

Block II

```

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
USE work.Tri_pkg.ALL;
entity bloc_2 is
PORT ( Go   : IN  Matrix;
Sm  : OUT Sub_MATRIX;
clk  : IN  std_logic;
Vec  : OUT Sub_Vec);
end bloc_2;
Architecture Arch_bloc_2 of bloc_2 is
begin
process(Go,clk)
variable k: integer :=0;

```

begin
if rising_edge(clk) then
ligne: for i IN 0 TO 1 loop
k:=0;
colonne: for j IN 0 TO 1 loop
Sm(i,k) <= Go(i,j);
k:=k+1;
end loop colonne;
end loop ligne;
Vecteur: for i IN 0 TO 1 loop
Vec(i) <= Go(i,2);
end loop Vecteur;
end if;
end process;
end Arch_bloc_2;

Block III

<pre> LIBRARY IEEE; USE IEEE.std_logic_1164.ALL; USE IEEE.numeric_std.ALL; USE work.Tri_pkg.ALL; ENTITY bloc_3 IS PORT(clk : IN std_logic; V : IN Sub_Vec; S : IN Sub_MATRIX; MF : OUT MATRIX); END bloc_3; Architecture Arch_bloc_3 of bloc_3 is component Sort PORT(clk : IN std_logic; Vec : IN Sub_Vec; Vt : OUT Sub_Vec); END component; component assemblage </pre>	<pre> PORT(clk : IN std_logic; Vt : IN Sub_Vec; Sm : IN Sub_MATRIX; MSF : OUT MATRIX); END COMPONENT; component div_2 port(clk : in STD_LOGIC; out_clk : out STD_LOGIC); end component; signal Vt_X : Sub_Vec; signal clk_X : std_logic; begin C1: div_2 port map(clk,clk_X); C2: Sort port map(clk,V,Vt_X); C3: assemblage port map(clk_X,Vt_X,S,MF); end arch_bloc_3; </pre>
---	---

Block IV

<pre> LIBRARY IEEE; USE IEEE.std_logic_1164.ALL; USE IEEE.numeric_std.ALL; USE work.Tri_pkg.ALL; ENTITY bloc4_compare IS PORT(clk : IN std_logic; MSF : IN Matrix; MI : IN Matrix; MF : OUT Matrix); END bloc4_compare; ARCHITECTURE arch_bloc_4 OF bloc4_compare IS -- Signals SIGNAL MSF_unsigned : Matrix; SIGNAL MI_unsigned : Matrix; SIGNAL MF_tmp : Matrix; BEGIN outputgen1: FOR i IN 0 TO 1 GENERATE MSF_unsigned(i,j) <= unsigned(MSF(i,j)); END GENERATE; outputgen2: FOR j IN 0 TO 2 GENERATE outputgen3: FOR i IN 0 TO 1 GENERATE outputgen4: FOR j IN 0 TO 2 GENERATE MI_unsigned(i,j) <= unsigned(MI(i,j)); END GENERATE; END GENERATE; END GENERATE; </pre>	<pre> END GENERATE; Compare : PROCESS (clk,MSF_unsigned, MI_unsigned) VARIABLE MF1,MF2: Matrix; VARIABLE mf_0 : unsigned(15 DOWNT0 0):="0000000000000000"; BEGIN MF1 := MI_unsigned; MF2 := MSF_unsigned; if rising_edge(clk) then FOR i IN 0 TO 1 LOOP FOR j IN 0 TO 2 LOOP FOR k IN 0 TO 2 LOOP if MF1(i,j)= MF2(i,k)THEN mf_0:= MF2(i,k); MF2(i,k):= MF2(i,j); MF2(i,j):= mf_0; end if; end loop; end loop; end loop; MF_tmp <= MF2; end if; end process; outputgen5: FOR i IN 0 TO 1 GENERATE outputgen: FOR j IN 0 TO 2 GENERATE MF(i,j) <= unsigned(MF_tmp(i,j)); END GENERATE; END GENERATE; END arch_bloc_4 ; </pre>
--	--

Block V

<pre> LIBRARY IEEE; USE IEEE.std_logic_1164.ALL; USE IEEE.numeric_std.ALL; USE work.Tri_pkg.ALL; ENTITY bloc4_compare IS PORT(clk : IN std_logic; MSF: IN Matrix; MI : IN Matrix; MF : OUT Matrix); END bloc4_compare; outputgen1: FOR i IN 0 TO 1 GENERATE outputgen2: FOR j IN 0 TO 2 GENERATE MSF_unsigned(i,j) <= unsigned(MSF(i,j)); END GENERATE; END GENERATE; outputgen3: FOR i IN 0 TO 1 GENERATE outputgen4: FOR j IN 0 TO 2 GENERATE MI_unsigned(i,j) <= unsigned(MI(i,j)); END GENERATE; END GENERATE; MF1 := MI_unsigned; MF2 := MSF_unsigned; if rising_edge(clk) then FOR i IN 0 TO 1 LOOP FOR j IN 0 TO 2 LOOP FOR k IN 0 TO 2 LOOP if MF1(i,j)= MF2(i,k)THEN ARCHITECTURE arch_bloc_4 OF bloc4_compare IS </pre>	<pre> -- Signals SIGNAL MSF_unsigned : Matrix; SIGNAL MI_unsigned : Matrix; SIGNAL MF_tmp : Matrix; BEGIN Compare : PROCESS (clk,MSF_unsigned, MI_unsigned) VARIABLE MF1,MF2: Matrix; VARIABLE mf_0 : unsigned(15 DOWNTO 0):="0000000000000000"; BEGIN mf_0:= MF2(i,k) MF2(i,k):= MF2(i,j); MF2(i,j):= mf_0; end if; end loop; end loop; end loop; MF_tmp <= MF2; end if; end process; outputgen5: FOR i IN 0 TO 1 GENERATE outputgen: FOR j IN 0 TO 2 GENERATE MF(i,j)<= unsigned(MF_tmp(i,j)); END GENERATE; END GENERATE; END arch_bloc_4 ; </pre>
---	---

Block VI

<pre> library IEEE; USE IEEE.std_logic_1164.ALL; USE IEEE.numeric_std.ALL; USE work.Tri_pkg.ALL; ENTITY Swh_activate IS PORT(clk : IN std_logic; Slc : IN std_logic_vector (1 downto 0) ; MI,MF : IN Matrix; SP,SN : OUT Matrix_swh); end Swh_activate; Architecture Arch_Swh of Swh_activate is Signal Sp_tmp , Sn_tmp: Matrix_swh; signal index_tmp: Matrix_idx; begin swh_output : PROCESS (clk,in- dex_tmp,MI,MF,Slc) variable sp_var,sn_var: matrix_swh; index(0,0):="000"; index(0,1):="000"; index(0,2):="000"; index(1,0):="000"; index(1,1):="000"; index(1,2):="000"; Sp_var(0,0):="10"; Sp_var(0,1):="10"; Sp_var(0,2):="10"; Sp_var(1,0):="01"; Sp_var(1,1):="01"; Sp_var(1,2):="01"; Sn_var(0,0):="01"; Sn_var(0,1):="01"; Sn_var(0,2):="01"; Sn_var(1,0):="10"; Sn_var(1,1):="10"; Sn_var(1,2):="10"; Sp_tmp <= Sp_var; </pre>	<pre> if rising_edge (clk) then if slc /= "11" then loop1: FOR i IN 0 TO 1 LOOP loop2: FOR j IN 0 TO 2 LOOP if MF(i,j) /= MI (i,j) THEN index (i,j) := "100"; else index (i,j) := "000"; end if; end loop loop2; end loop loop1; loop3: FOR i IN 0 TO 1 LOOP loop4: FOR j IN 0 TO 2 LOOP if index (i,j) = "100" then Sn_var(1,1) := "10"; Sn_var(1,2) := "10"; end if; end process swh_output; Sortie1: FOR m IN 0 TO 1 GENERATE Sortie2: FOR n IN 0 TO 2 GENERATE if i= 1 then Sp_var (i,j) := "10"; Sn_var (i,j) := "01"; else Sp_var (i,j) := "01"; Sn_var (i,j) := "10"; end if; else if i= 1 then Sp_var (i,j) := "01"; Sn_var (i,j) := "10"; else Sp_var (i,j) := "10"; Sn_var (i,j) := "01"; end if; end if; end loop loop4; </pre>
--	--

<pre>Sn_tmp <= Sn_var; index_tmp <= index; end if; else Sp_var(0,0) := "10"; Sp_var(0,1) := "10"; Sp_var(0,2) := "10"; Sp_var(1,0) := "01"; Sp_var(1,1) := "01"; Sp_var(1,2) := "01";</pre>	<pre>end loop loop3; Sn_var(0,0) := "01"; Sn_var(0,1) := "01"; Sn_var(0,2) := "01"; Sn_var(1,0) := "10"; SP(m,n) <= Sp_tmp(m,n); SN(m,n) <= Sn_tmp(m,n); END GENERATE; END GENERATE; END ARCHITECTURE Arch_Swh;</pre>
---	---

Appendix C

P & O Algorithm

<pre> library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.STD_LOGIC_ARITH.ALL; use IEEE.STD_LOGIC_UNSIGNED.ALL; entity appli is Port (CLK : in std_logic; valid: in std_logic_vector(1 downto 0); reset: in std_logic; tension : in std_logic_vector(7 downto 0); courant :in std_logic_vector(7 downto 0); ACQ : out std_logic_vector(1 downto 0); ASSIGN : IN std_logic; CALCULATE_P : IN std_logic; CALCULATE_diff : IN std_logic; TREAT: in std_logic; D_out:out std_logic_vector(15 downto 0)); end appli; architecture appli_arch of appli is constant dD:integer:=5; signal I2:std_logic_vector(7 downto 0); I2 <=transport Courant after 10 ns ; V2 <= transport Tension after 10. end if; end process ; </pre>	<pre> signal V2:std_logic_vector(7 downto 0); signal P2:std_logic_vector(15 downto 0); signal P1:std_logic_vector(15 downto 0); signal d2:std_logic_vector(15 downto 0); signal d1:std_logic_vector(15 downto 0); signal diff_D:std_logic_vector(15 downto 0); signal diff_P:std_logic_vector(15 downto 0); begin Proc00 : process (Valid,reset,clk) begin if reset = '1' then I2 <= conv_std_logic_vector(0,8); V2 <=conv_std_logic_vector(0,8); elsif (clk'event and clk = '1' and Valid(0) ='1'and Valid(1) ='1') then I2 <=transport Courant after 10 ns ; V2 <= transport Tension after 10 end if; end process ; </pre>
--	---

References

1. Hassan H, Mohab A (2009) Low-power design of nanometer FPGAs: architecture and Eda. Morgan Kaufmann Publishers
2. Clive M (2011) FPGAs instant success. Elsevier Science
3. Simpson PA (2015) FPGA design best practices for team-based reuse
4. Sklyarov V et al (2014) Synthesis and optimization of FPGA-based systems. Lect Notes Electr Eng 294. https://doi.org/10.1007/978-3-319-04708-9_1
5. Blanchardon A (2016) Synthesis of fault tolerant FPGA circuit architectures. HAL Id: tel-01243976
6. Farooq U et al (2012) Tree-based heterogeneous FPGA architectures. Springer Science+Business Media New York. https://doi.org/10.1007/978-1-4614-3594-5_2
7. QPro Virtex-E 1.8V QML Data Sheet, DS098-1 (v1.1) July 29, 2004
8. Spartan-II FPGA Family Data Sheet, DS001 March 12, 2021
9. Simpson P (2014) System design with FPGA, best practices for collaborative development, Dunod
10. Bossuet L (2012) Reconfigurable green terminals—towards sustainable electronics, HAL Id: hal-00753223
11. Airiau R, Berge JM, Olive V, Rouillard J (1998) VHDL: language, modeling, synthesis. Technical and scientific collection of telecommunications
12. Kapre N, Dehon A (2008) Programming FPGA applications in VHDL in book: reconfigurable computing. <https://doi.org/10.1016/B978-0-12370522-8.50011-X>
13. LaMeres BJ (2017) Introduction to logic circuits & logic design with VHDL. <https://doi.org/10.1007/978-3-319-34195-8>
14. Wilson P (2016) Chapter 3—A VHDL primer: the essentials. VHDL in book: design recipes for FPGAs, 2nd edn. <https://doi.org/10.1016/B978-0-08-097129-2.00003-9>
15. LaMeres BJ (2019) Introduction to logic circuits & logic design with VHDL. Springer
16. Gazi O (2019) A tutorial introduction to VHDL programming. Springer
17. Ramachandran S (2010) Digital VLSI systems design. Springer
18. Bezerra EA, Lettnin DV (2014) Synthesizable VHDL design for FPGAs. Springer
19. Pedroni VA (2010) Circuit design and simulation with VHDL. The MIT Press Cambridge
20. Gazi O, Arli AÇ (2021) State machine using VHDL. Springer
21. Lee S (2005) Advanced digital logic design using VHDL, state machine, and synthesis for FPGAs. Thomson-Engineering
22. Bezerra EA, Lettnin DV (2014) Synthesizable VHDL design for FPGAs. Springer
23. Pedroni VA (2010) Circuit design and simulation with VHDL. The MIT Press Cambridge

24. <https://www.irena.org>
25. Seals RC, Whaphott GF (1997) Programmable logic: PLDs and FPGAs. Macmillan International Higher Education
26. <https://www.ee.co.za/article/can-fpgas-renewable-energy-system.html>
27. Pellerin D, Thibault S (2005) Practical FPGA programming in C. Prentice Hall Press
28. Livingstone DJ (ed) (2008) Artificial neural networks: methods and applications. Humana Press, Totowa, NJ, USA, pp 185–202
29. Alqahtani A, Alsaffar M, El-Sayed M, Alajmi B (2016) Data-driven photovoltaic system modeling based on nonlinear system identification. *Int J Photoenergy*
30. Mekki H, Mellit A, Kalogirou SA, Messai A, Furlan G (2010) FPGA-based implementation of a real time photovoltaic module simulator. *Prog Photovoltaics Res Appl* 18(2):115–127
31. Bouselham L, Hajji B, Mellit A, Rabhi A, Kassmi K (2019) Hardware implementation of new irradiance equalization algorithm for reconfigurable PV architecture on a FPGA platform. In: 2019 international conference on wireless technologies, embedded and intelligent systems (WITS). IEEE, pp 1–8
32. Das SK, Verma D, Nema S, Nema RK (2017) Shading mitigation techniques: state-of-the-art in photovoltaic applications. *Renew Sustain Energy Rev* 78:369–390
33. Velasco G, Negroni JJ, Guinjoan F, Pique R (2005) Irradiance equalization method for output power optimization in plant oriented grid-connected PV generators. In: 2005 European conference on power electronics and applications. IEEE, p 10
34. Mellit A, Kalogirou SA (2014) MPPT-based artificial intelligence techniques for photovoltaic systems and its implementation into field programmable gate array chips: review of current status and future perspectives. *Energy* 70:1–21
35. Chettibi N, Mellit A (2019) Study on control of hybrid photovoltaic-wind power system using Xilinx system generator. In: Solar Photovoltaic Power Plants. Springer, Singapore, pp 97–120
36. Selvamuthukumaran R, Gupta R (2014) Rapid prototyping of power electronics converters for photovoltaic system application using Xilinx system generator. *IET Power Electron* 7(9):2269–2278