Eduardo Augusto Bezerra
Djones Vinicius Lettnin

Synthesizable VHDL Design for FPGAs

Springer

Synthesizable VHDL Design for FPGAs

Eduardo Augusto Bezerra
Djones Vinicius Lettnin

Synthesizable VHDL Design for FPGAs

Springer

Eduardo Augusto Bezerra
Djones Vinicius Lettnin
Department of Electrical and Electronic Engineering
Universidade Federal de Santa Catarina
Florianópolis, Santa Catarina
Brazil

ISBN 978-3-319-02546-9 ISBN 978-3-319-02547-6 (eBook)
DOI 10.1007/978-3-319-02547-6
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013950359

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Contents

Chapter 1
Digital Systems, FPGAs and the Design Flow

This chapter introduces the target technology used in the laboratory sessions, and provides a step-by-step guide for the design and simulation of a digital circuit. At the end of this chapter, the reader should be able:

- to have a basic understanding of the FPGA technology (internal blocks, applications);
- to understand the basic logic gates operation;
- to design a logic circuit using a schematic editor tool;
- to simulate and test a digital circuit;
- to understand the basic flow of an Electronic Design Automation (EDA) tool.

1.1 Digital Systems

Digital systems are composed of two basic components: Datapath and Control Unit. As shown in Fig. 1.1, the control unit has as its main function the generation of control signals to the datapath unit (also known as "operational block"). The control signals "command" the desired operations in the datapath. Furthermore, the control unit may receive control inputs from the external environment; for instance, it can be a simple "start" or even an operation code ("opcode" in microprocessors). Finally, this unit can also generate one or more output control signals to communicate with other digital systems (e.g., "done, bus request, nack").

The datapath unit performs operations on data received, usually, from the external environment. The operations are performed in one or more steps, where each step takes a clock cycle. The datapath generates "status" signals (sometimes also called as "flags") that are used by the control block to define the sequence of operations to be performed. Examples of datapath blocks include:

- Interconnection network—wires, multiplexers, buses, and tri-state buffers;
- Functional units—adders, subtractors, shifters, multipliers, and Arithmetic and Logic Unit (ALU);

E. A. Bezerra and D. V. Lettnin, *Synthesizable VHDL Design for FPGAs*,
DOI: 10.1007/978-3-319-02547-6_1,

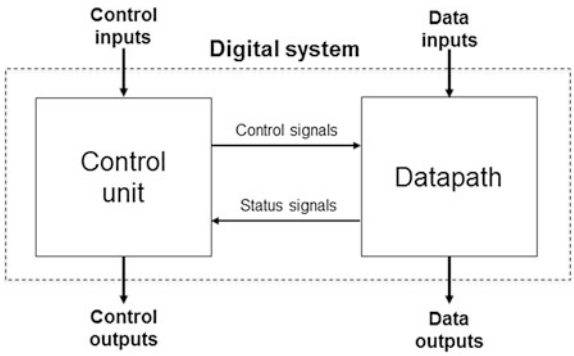


Fig. 1.1 Digital system (adapted from [ManoKime99])

- Memory elements—registers, and Random Access Memory (RAM).

The datapath and control units are based on two types of circuits: combinational and sequential circuits.

Combinational circuits have I_m inputs and O_n outputs, as shown in Fig. 1.2. In these circuits the output pins depend only on the values that are presented to the input pins and each output is defined by a different Boolean logic equation.

A combinational circuit can also be classified according to its application as:

- Interconnection circuit—e.g. multiplexers, demultiplexers, encoders, and decoders;
- Logical and arithmetic circuit—e.g. adders, subtractors, multipliers, shifters, comparators, and ALUs (circuits that combine more than one arithmetic or logical modules).

Sequential circuits have I_m inputs and O_n outputs. However, in these circuits the output pins depend not only on the values that are presented to the input signals (i.e. m signals), but they depend also on the current state stored in the memory module, as shown in Fig. 1.3. The number of both next state (i.e. N_j) and current state signals (i.e. S_j) depend on the encoding style of a Finite State Machine (FSM) states, such as, sequential binary, gray code or one hot.

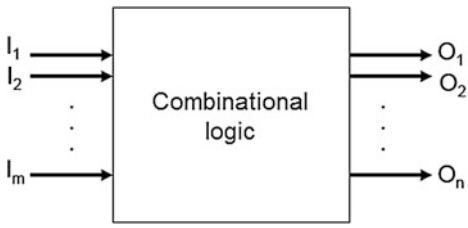


Fig. 1.2 Combinational circuit

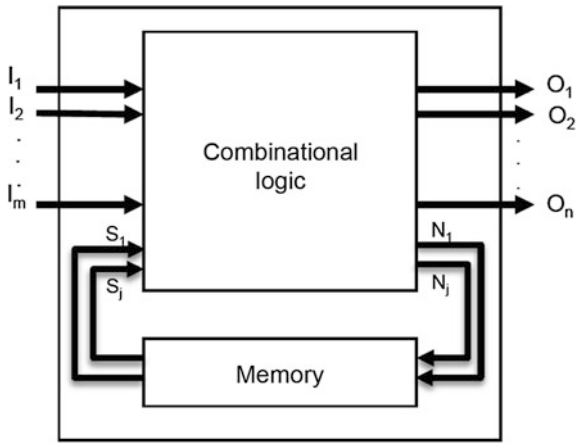


Fig. 1.3 Sequential circuit

1.2 Field Programmable Gate Array

The concept behind the Field Programmable Gate Array (FPGA) technology is better understood through a digital circuit design example.

Problem:
You are asked to design a circuit that implements the following logic function:

$$f(A, B) = A \text{ AND } B$$

In this circuit, inputs A, B, and output f are all 4 bits wide.

Solution 1: Using a 7408 TTL device (Application-specific Integrated Circuit—ASIC solution)
The operations to be performed are:

$$f(1) = A(1) \text{ AND } B(1)$$
$$f(2) = A(2) \text{ AND } B(2)$$
$$f(3) = A(3) \text{ AND } B(3)$$
$$f(4) = A(4) \text{ AND } B(4)$$

The 7408 TTL device, shown in Fig. 1.4, has four 1-bit AND gates. Therefore, a single 7408 chip is sufficient to implement the 4 bits function. In Fig. 1.4, the A inputs are connected to pins 1, 4, 9 and 12. The B inputs are connected to pins 2, 5, 10 and 13. The f output can be obtained from pins 3, 6, 8 and 11. The chip must be placed (soldered) on a printed circuit board (PCB) or protoboard, and all pin connections should be made, including the power lines (Vcc and GND).

This is a nice solution for the proposed problem, but it has some drawbacks. TTL is an obsolete technology, presenting high-energy consumption, and the chip itself takes considerable space on the PCB. Another drawback is in case of changes

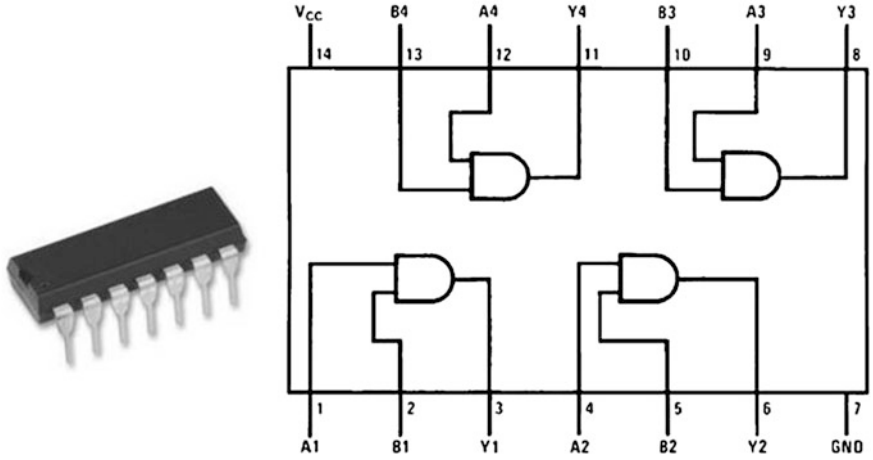


Fig. 1.4 7408 TTL chip

or upgrades in the design requirements. In these situations, a recall may be needed in order to replace the components (7408 chip) by the new ones. Depending on the required changes, a full PCB redesign may be needed.

Solution 2: Microprocessor (software solution)
A microprocessor (or microcontroller) based system is not the best solution for this sort of "simple problem", but it has been included here in order to show the differences between the hardware and software approaches.

In a similar way to the first solution, a microprocessor also needs to be placed on a PCB, but it has more flexibility for changes in the system functionality. As the solution is implemented at the "software level", any changes in the design requirements may be performed in the system with no need for hardware or PCB modifications. A possible solution for the problem is shown in the C language program as follows:

```
int main(){
    int A, B, F;
    F = A & B; //F = A AND B
}
```

However, this solution also has some drawbacks. A special C compiler may be required in order to have a 4-bits wide *int* type. Another difficulty would be to assign the inputs and outputs (A, B, F) to the microprocessor physical pins. Considering a standard C compiler, the developer would use *scanf* or *getc* input functions to obtain the *A* and *B* values, and *printf* to output the *F* result. Depending on the target systems, several adaptations would be necessary in order to associate the functions to the device's pins. In a microcontroller based C compiler, usually, special functions are provided to the programmer in order to access the device's pins.

Solution 3: FPGA
An FPGA device could be seen as an intermediary solution, between the hardware and the software approaches. In a similar way to the hardware (TTL chip) and the software (microprocessor chip) approaches, the FPGA chip also has to be placed on a PCB, and all its input and output pins should be connected. An important difference from the microprocessor approach is that the solution for the problem is not implemented in software. It is a hardware solution, similar to the 7408 chip, but with the same flexibility for changes as the microprocessor based approach. In case of bugs or design changes, a new circuit can be "placed" inside the FPGA, with no need for PCB modifications. A solution for the problem could be written in the VHDL language as follows:

```
library IEEE;
use ieee.std_logic_1164.all;
entity AND_VHDL is
   port (A, B : in std_logic_vector (3 downto 0);
   F : out std_logic_vector (3 downto 0)
   );
end entity;
architecture rtl of AND_VHDL is
begin
   F <= A AND B;
end rtl;
```

The microprocessor solution drawbacks are easily outwitted when using an FPGA. Different bit lengths for the inputs and outputs are defined by standard language constructions, as well as the assignment of physical pins.

However, an interesting advantage of Solution 3 (FPGA technology) is the flexibility for hardware changes, after the product has been completed. Assuming that a requirement mistake has been identified, and the circuit were supposed to perform an OR operation and not the implemented AND operation. In Solution 1, the 7408 chip would have to be replaced by the 7432 TTL device (four 1-bit OR gates), which means to remove the chip from the board, and to place the new one. In Solutions 2 and 3, there will be no need for chips replacement. In Solution 2 the software would have to be changed and reloaded in the microprocessor's memory. In Solution 3, the VHDL code would have to be changed, using OR instead of AND, and a new FPGA configuration should be generated. It is important to notice that in Solution 3 a "new hardware" is generated, for the FPGA, while in Solution 2 a new piece of "software" is loaded in the microprocessor.

1.3 FPGA Internal Organization

An FPGA is a device (a chip) whose internal logic can be changed by the developer (user) after the chip has been manufactured—it is "field programmable". From the developer point of view, its functionality is similar to a microprocessor, as it is

possible to change its operation according to the application requirements. In the ASIC approach, the chip has its functionality defined by the manufacturer (permanently), and it cannot be changed. For instance, the 7408 device introduced in "Solution 1" has four 1-bit AND gates, and this hardware arrangement is fixed and it is not possible to be modified. The microprocessor ("Solution 2"), also has a fixed hardware that cannot be changed (registers, ALU, …), but its functionality can be changed by the user at the "software level". The internal hardware of a microprocessor has been designed in order to execute instructions of a program, and different programs can be written according to the application requirements. An FPGA, on the other hand, has an internal logic that can be changed by the developer. The main similarity to a microprocessor is that the FPGA developer also uses a language to describe the application's functionality, and a tool to generate the new hardware to be configured inside the FPGA. However, as an FPGA is not a microprocessor, it has not been designed to execute a program written by the developer. The source code written by the developer is in fact a "hardware description", and it becomes the FPGA internal logic configuration.

In order to have its internal logic changed, the architecture of an FPGA chip employs "writable" technologies, such as SRAM or FLASH memories. FPGA devices have three main configurable components: the logic blocks; the routing elements; and the input/output (I/O) blocks. The three types of components are shown in Fig. 1.5, where the I/O blocks are located in the periphery of the chip, and the logic blocks are the larger squares, interconnected by the routing elements.

The "logic blocks" are configured in order to perform the logic operations required by the application, and so they are the "processing elements" of an FPGA. The logic blocks are interconnected through the configurable routing elements, which are used also for the connection to/from I/O blocks. The I/O blocks may be configured in order to function as input pins, output pins, or both. Figure 1.6 shows a possible hardware configuration for the Solution 3 described before. A single logic block is used to perform the AND operation. Eight I/O blocks are configured as

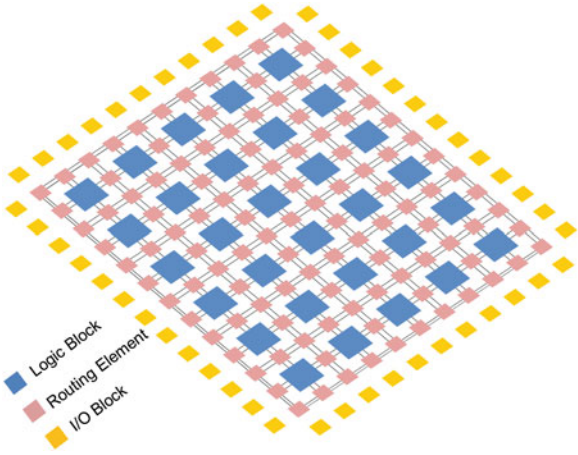


Fig. 1.5 A typical FPGA internal components

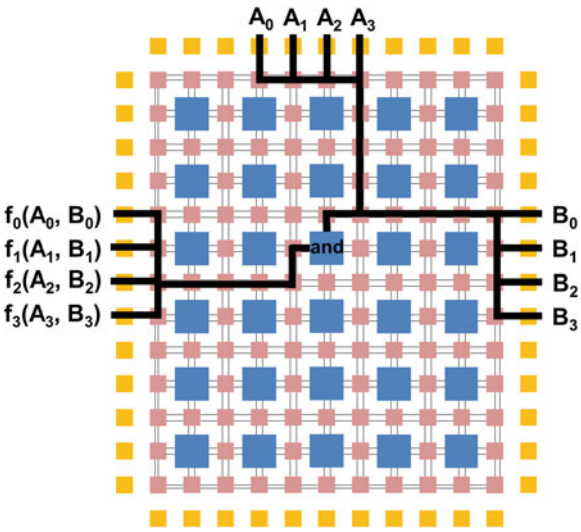


Fig. 1.6 Possible FPGA configuration for Solution 3

input, and a path (routing) is configured between them and the AND logic block. Four I/O blocks are configured as output, and they are connected to the output of the logic block.

1.4 Configurable Logic Block

As stated before, FPGAs are made of three major configurable resources: Logic Blocks; Routing Elements; and I/O Blocks. The internal organization of all three elements is very interesting and deserves a more detailed explanation, but this is not the main goal of this book. However, in order to better understand the developing tools functionality, it is important to have a basic idea of how logic functions are implemented in FPGAs. Most FPGAs are based on look-up tables (LUTs), which can be used as "universal function generators". An LUT is a configurable resource capable of implementing any n-input truth table. Figure 1.7 shows an abstract view of an LUT contents generation from a logic function.

In Fig. 1.7, the logic function in the upper left hand corner is written in an input format accepted by the selected Electronic Design Automation (EDA) tool (e.g. schematic diagram, VHDL, Verilog, …). The EDA tool generates the truth table for the logic function (I). During the programming process, the results column of all truth tables is written to the FPGA configuration memory (II). More precisely, in a 4-input LUT based FPGA, a truth table for a 4-input logic function (A, B, C, D) is written to a LUT of a logic block. The FPGA does not need to have any physical logic gates, as only the results of a truth table are stored in the LUTs (FPGA configuration memory). In order to perform the equation, the inputs (A, B, C, D) are used as the MUX selection (III), and the provided output is the final result (IV). This functionality

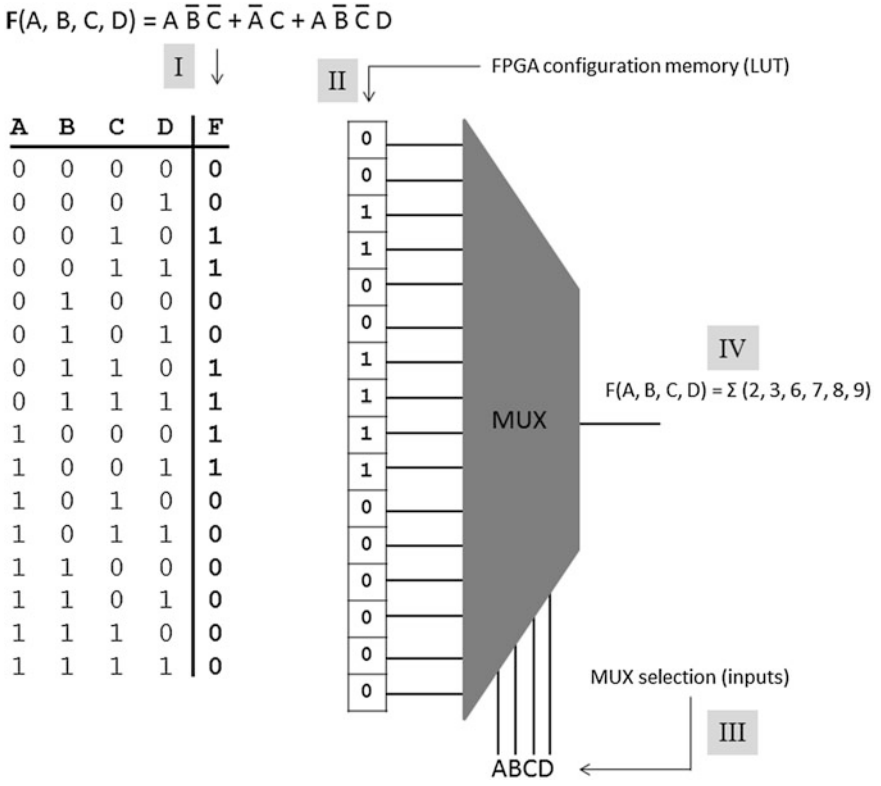


Fig. 1.7 LUT working model

explains how Solution 3 described before can be implemented in an FPGA. As no logic gates are used to implement a logic function, each time a new functionality is required in a design, the circuit developer just needs to change the hardware description, and the EDA tool generates a new truth table to be used in the LUT configuration. An LUT is the building block of the Logic Block shown in Fig. 1.5.

1.5 Electronic Design Automation and the FPGA Design Flow

In order to develop a digital circuit targeting an FPGA device, it is compulsory to use Electronic Design Automation tools. Figure 1.8 shows a typical FPGA design flow, where several EDA tools are employed.

The first step is the "Design Entry", where the developer uses a design-editing tool to provide their circuits description. Usually, EDA tools take a number of

description formats as, for instance, schematic diagrams, hardware description languages (HDLs), state machine diagrams, among others.

The next step in Fig. 1.8 is the circuit simulation. There are several simulation tools available, and the most used method in all of them is the waveform analysis. The circuit's logic and states can be fully simulated using tools that provide a detailed waveform generation and analysis. It is important to notice that this is a "functional simulation", as at this stage of the design flow there is no timing information yet. The functional simulation process tests only the design's logic operation, as it has no further information regarding the internal circuit in the target device (FPGA) or routing path delays. This step should be performed in order to verify the circuit functionality disregarding timing constraints and other performance parameters. In this step, in case of functional errors, the designer may change the circuit using the design entry tool, and perform new simulations until the design shows the expected behavior.

After have finished the functional simulation, the designer can perform the synthesis activity and a *netlist* representing the provided circuit is generated. This

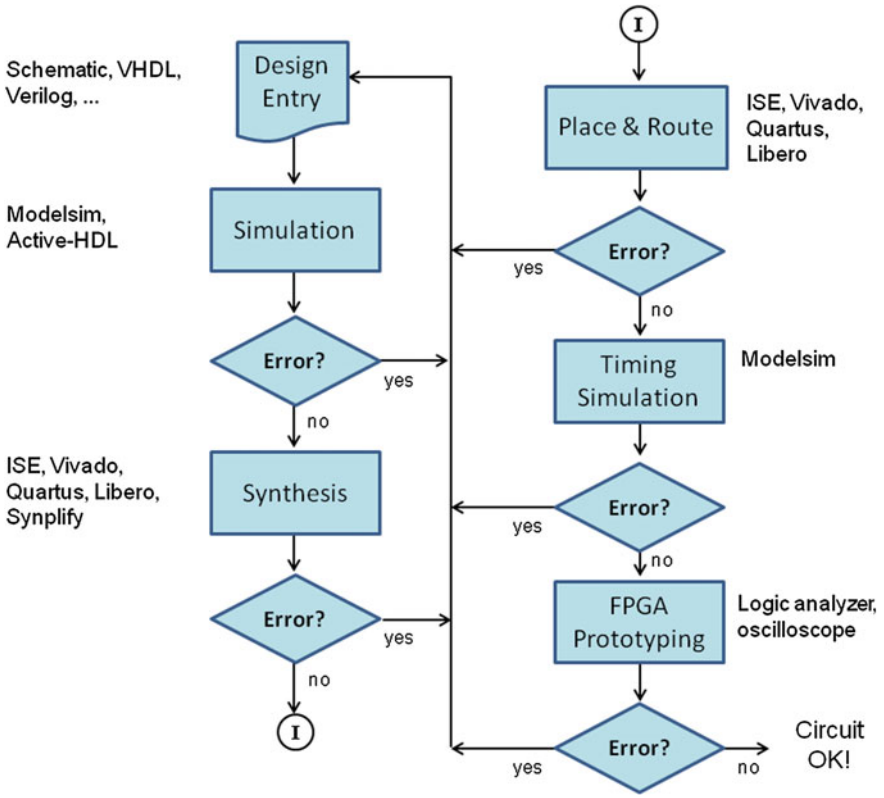


Fig. 1.8 Typical FPGA design flow

activity is split in two operations: the logic synthesis; and the physical synthesis. Traditionally, the logic synthesis role is to map the provided circuit to a set of gates (netlist) performing the same functionality as the original circuit description. This netlist has not yet any physical information such as power consumption or routing delays for the final physical implementation. The "design entry", the "functional simulation", and the "logic synthesis" are activities performed by a professional called the "Frontend Engineer".

The remaining activities shown in Fig. 1.8 are performed by the "Backend Engineer". In FPGA designs, it is not unusual to have the same engineer performing both, frontend and backend, activities. The first backend activity in an FPGA design flow is the "placement and routing". At the end of the process, in case of errors the designer may go back to the design entry step in order to fix the problems pointed out during the synthesis. The physical synthesis output is a netlist and the respective physical design data. The design data, used in the timing simulation step, include performance figures such as maximum clock rate, area usage, power estimation, and throughput. The netlist has all the physical design information needed for the FPGA configuration, including: I/O pins, routing paths, and operational blocks for LUTs configuration, among others. In the physical synthesis step activities as placement, routing and circuits optimizations are accomplished. In contemporary EDA tools, the logic and the physical synthesis are implemented in a cooperative way, employing incremental algorithms. This strategy results in better results, as the logic synthesis is performed considering feedback information obtained from the physical synthesis.

In the "timing simulation" step, the designer employs the data provided by the physical synthesis in order to verify whether or not the physical design will meet the defined performance constraints. If the timing simulation results in acceptable figures, so there is a higher confidence that the physical circuit will work according to the requirements. However, only in the final step, that is, the FPGA prototyping, it is possible to have a better assurance of the circuit correctness. In this step, usually, the circuit is tested in a development board.

1.6 FPGA Devices and Platforms

Xilinx and Altera are the major players in the FPGA market. Their main business is the manufacture of FPGA devices, but they also develop EDA tools. The FPGA internal organization complexity is a motivation for these companies to produce their own physical synthesis tool. The FPGA manufacturer has, undoubtedly, the best understanding of a device's internal organization and, consequently, is the most suitable developer for a low level EDA tool. Another motivation for the FPGAs companies to develop their own physical synthesis tools is related to the confidential information regarding the device's internal organization. Holding back this information is important for a company as it may result in technology advantages over competitors. Other important players in the FPGA market include:

Microsemi Corporation (formerly Actel); Lattice Semiconductor; and SiliconBlue Technologies.

These companies target very specific markets, where the most common applications are: networking (switches and routers); video and image processing; cryptography; space systems (satellites and deep space missions); avionics; defense; ASIC prototyping; audio equipment; medical solutions; motor control; high performance computing; and automotive systems. In order to attend the requirements of these applications, current FPGA devices have extra hardware resources as, for instance, DSP operators, RAM blocks, and even embedded microprocessors.

FPGA devices are used not only as basic components in the design of boards for a variety of applications, but also in the design of development and educational boards. The laboratory assignments described throughout this book have been elaborated targeting, mainly, the following educational boards:

- The Altera DE2 Development and Education board shown in Fig. 1.9 has an Altera Cyclone II 2C35 FPGA, and several peripherals. The board has been designed for teaching digital systems classes, in laboratory-based courses.
- The Mercurio IV board has been developed by Macnica-DHW Engineering and it has an Altera Cyclone IV FPGA. It also has a set of peripherals, and was developed aiming university courses.

Fig. 1.9 DE2 Development and Education board [Altera]

1.7 Writing Software for Microprocessors and VHDL Code for FPGAs

Applications for FPGAs and microprocessor based systems can be modeled and conceived at different levels of abstraction, and using different design entry formats. A compiler used to generate object code (machine code) for a microprocessor can have as input, for instance, C/C++ source code (computer program) and also other pieces of software and components available in libraries. A synthesis tool (hardware compiler) used to generate a netlist (bitstream, circuit) to an FPGA has as input a code written in an HDL (VHDL, Verilog, …), as well as components available in libraries. An important distinction here is that a piece of software written in the C language is at a level of abstraction many times higher than a hardware description written in VHDL. Even the assembly language can be considered as in a higher abstraction level than VHDL. The VHDL developer is in fact describing the behavior of a piece of hardware to be used to implement some desired functionality (application), while with the assembly language (or any other high level languages) the developer is describing the application's functionality that will be performed by an existing piece of hardware (microprocessor). The VHDL developer uses the language to design the hardware and the application to be performed, while a microprocessor programmer has to worry, basically, with the application functionality to be implemented.

Considering the low abstraction level of VHDL descriptions, an attractive approach used by software developers when targeting an FPGA implementation, is the use of software programming languages followed by the automatic conversion to VHDL. Problems appear when circuits with a certain level of optimization are expected. In this case programmer needs to use special keywords and special instructions to be used by the high level software compiler to produce a more optimized code. Depending on the design complexity, it may be harder to realize than developing the system straight in VHDL. Nested loops, the variety of data types, and some arithmetic operators are examples of restrictions for the translation. The main point is that a compiler for a high level language can translate the program to a low-level language (assembly language), which has similar instructions, making the translation process not too difficult. On the other hand, synthesis tools have to translate the high level programs to a much reduced set of low level "instructions". For instance, while a compiler can use a *je* (conditional jump in assembly) instruction as the target for the translation of a high level *if* constructor, a synthesis tool may have to use D flip-flops and multiplexers.

Over the years some "programming language" to "HDL" translators have been developed, but with no major impact in the market. Developers with a software background starting in the field of HDL design, usually try the way that appears to be the easiest one, which is the use of a language translator. As soon the design challenges start, and the developer must learn very specialized commands and constructions in order to produce a proper code suitable for the chosen translator, so there is a realization that learning a hardware description language may be a

better option. Learning a new programming paradigm is not easy, but the syntaxes of languages like Verilog or VHDL are not so different, for instance, from the C language. In addition, a C developer has to use only a sub-set of the language, because of the synthesis tools limitations. For example, the translation of dynamic structures (pointers) from C to VHDL is not useful, as this class of data type is not suitable for static hardware generation.

1.8 Laboratory Assignment

The laboratory objectives are:

- practicing the use of basic logic gates in the design of a digital circuit;
- to use a design entry tool based on schematic editing;
- to have a first contact with an EDA tool;
- to have a basic understanding of a complete FPGA design flow, starting from the schematic design entry step, performing the circuit simulation, and prototyping it in an FPGA development board.

1.8.1 Logic Gates

Figure 1.10 shows the basic logic gates: OR; AND; XOR; and NOT. The gate functionality is also described through different textual representations, and by its truth table.

A digital circuit can be designed as a collection of logic gates, connected in order to perform the required function. For instance, in Fig. 1.11 it is shown the design of a circuit that implements an add operation. This circuit is called a

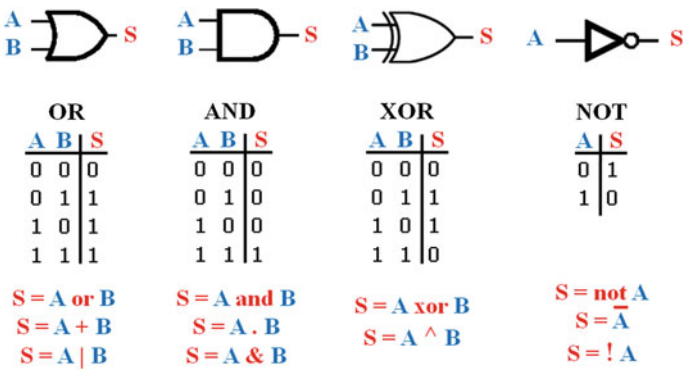


Fig. 1.10 Basic logic gates

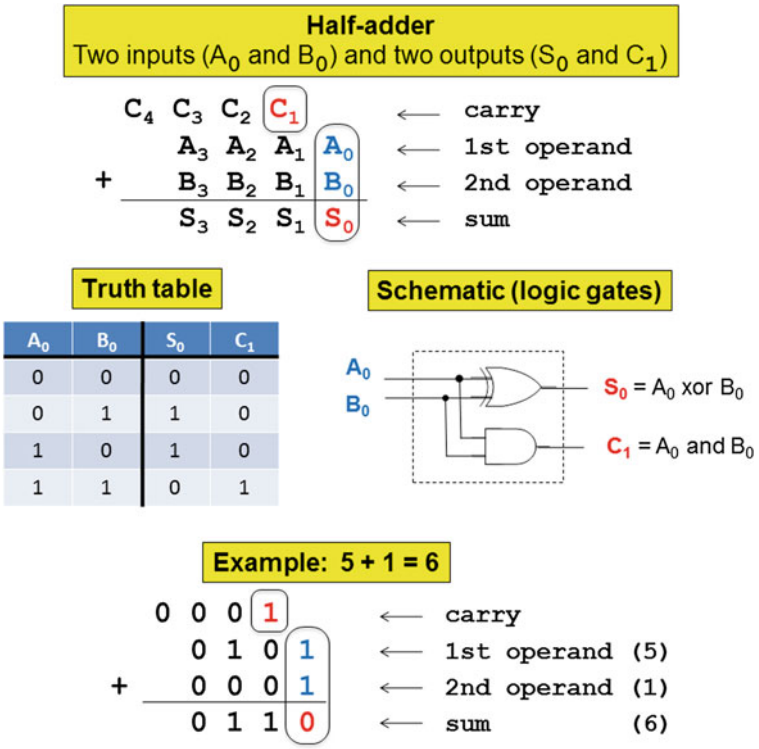


Fig. 1.11 Half-adder circuit implemented with an AND gate and an XOR gate

"half-adder" and it is used for the summation of the least significant bits (LSB) of two n-bits operands, providing as a result the sum and a carry. Figure 1.11 shows the sum of two 4-bit values, and the half-adder circuit used to add their LSBs.

1.8.2 Laboratory Session

The tasks to be completed in this laboratory session, using Altera's Quartus II EDA tool, are as follows:

- Create the half-adder circuit shown in Fig. 1.11, using the schematic design entry tool.
- Perform the synthesis (logic and physical).
- Perform the functional simulation.
- Fix simulation and synthesis errors.
- Prototype the half-adder circuit in the development board, in order to check its functionality.

For those who do not have the Quartus II software installed, there is a free version available on Altera's website at http://www.altera.com.

Quartus II 12.1 was the latest version available when this book was written, and this version has been used in all laboratory sessions.

In this first laboratory session, the idea is to follow the step-by-step instructions in order to get used to the EDA tool facilities and resources. This tutorial can be used as a guide to the next chapters as well.

Step 1: Creating a New Project

Start Quartus II, close all initial windows, and open the menu *File→New Project Wizard*, as shown in Fig. 1.12.

Next, choose the project's folder and name. In a VHDL design, it is important to provide the top-level entity's name. Do not use names or paths with spaces or special characters (e.g., à, ç, …) in any of the text boxes (Fig. 1.13).

In "page 2 of 5", just click on the "Next" button. In "page 3 of 5", choose the DE2's FPGA, which is the Cyclone II ("Family") EP2C35F672C6 ("Device"), as shown in Fig. 1.14. In Fig. 1.15, choose ModelSim-Altera as the simulation tool.

In "page 5 of 5" it is shown a summary of the new project. Just click "Finish".

Step 2: Design Entry (Schematic)

Chose *File→New* and *Block Diagram/Schematic File*, as shown in Fig. 1.16. This will open an empty editor, used to create the new circuit schematic.

The schematic to be created should implement the half-adder circuit shown in Fig. 1.11. The schematic editor tool has several pre-defined components stored in libraries. The logic gates are available in the Primitives/Logic library. To add a

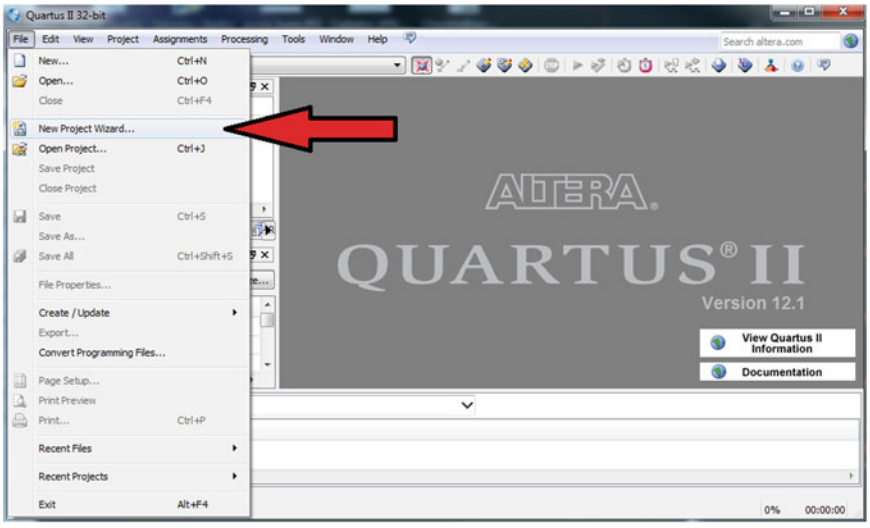


Fig. 1.12 Quartus II—New project wizard

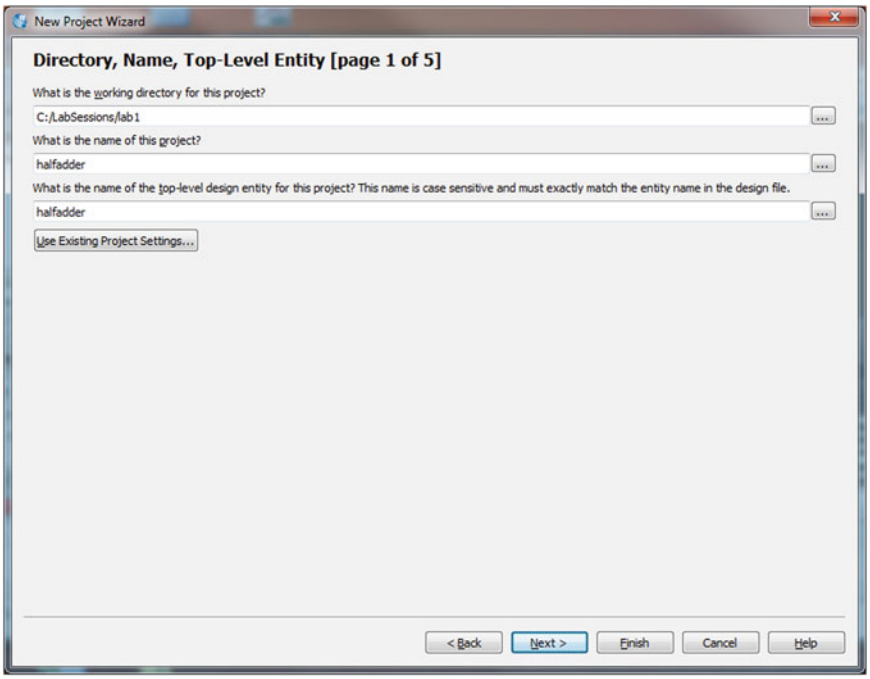


Fig. 1.13 Directory, name and top-level entity

logic gates to the schematic, click on the "Symbol Tool" icon highlighted in Fig. 1.17.

To connect the gates, use the "Orthogonal Node Tool" as indicated in Fig. 1.18. The input and output pins, can be found in library Primitives/Pin, as shown in Fig. 1.19. The pin names can be renamed, as shown in Fig. 1.20, completing the half-adder design.

When done, save the schematic design: *File→Save* (or just Ctrl-S). A window will open asking for the schematic's name, and for a place to save the design. The default name is *halfadder.bdf*, and the folder is *c:\LabSessions\lab1*.

Step 3: Synthesis

In the synthesis step, an analysis is performed in the schematic design and a circuit is generated targeting the selected FPGA device. To start the synthesis, choose *Processing→Start Compilation* (or just type Ctrl-L). The eleventh icon from right to left in the menu bar in Fig. 1.21 can also be used to start the synthesis process. At the end of the synthesis, in case of errors found, go back to the schematic design, and check for bad connections. In case of success, the next step is the simulation. Figure 1.22 shows a summary of the synthesis process, listing the four pins used, and two logic elements.

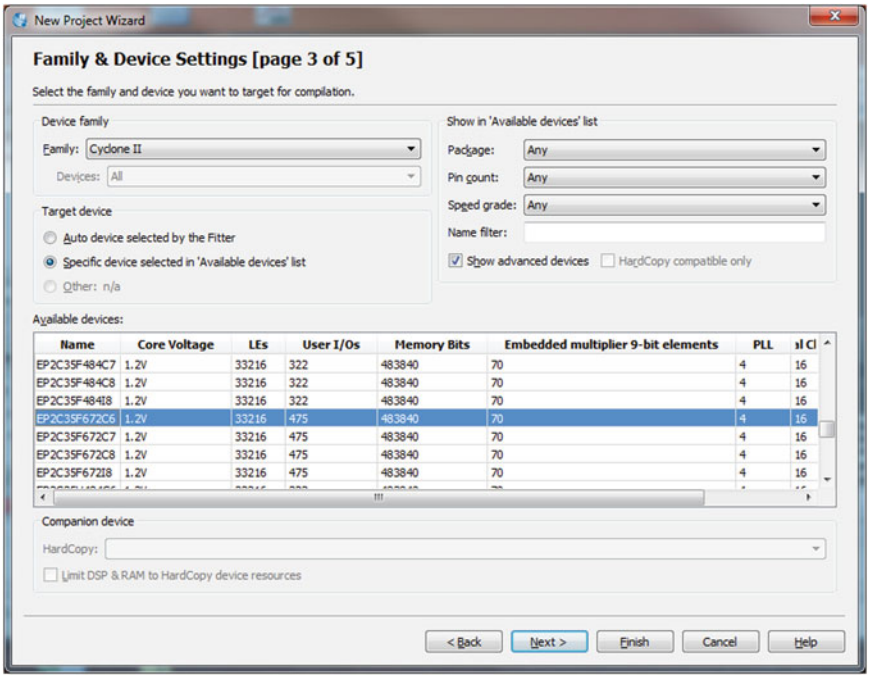


Fig. 1.14 Selecting Altera FPGA—Cyclone II EP2C35F672C6

Step 4: Simulation

Quartus II (v.10 and following) does not have its own simulator, and ModelSim, a third party software tool is used. The ModelSim-Altera tool is installed together with Quartus II, but before the first simulation it is recommended to verify that the tool is properly configured. In Quartus II, as shown in Fig. 1.22, open *Tools→Options*.

Click on EDA Tool Options, and type ModelSim path in the text box shown in Fig. 1.23.

To start the simulation tool, select *Tools→Run Simulation Tool → Gate Level Simulation* (or press the sixth icon from right to left shown in Fig. 1.21). This will run ModelSim software.

When ModelSim starts, just close the presentation window, and initiate the simulation by putting ModelSim in "simulation mode": select *Simulate→Start Simulation*. The "Start Simulation" window in Fig. 1.24 will show. This window has several tabs, but for this functional simulation just the first tab (Design) will be used. The Design tab lists all designs ready for simulation, including the half-adder circuit that has been synthesized.

The target design is located in the "work" library. Open this library clicking on the "+" sign. As a next step, click on "halfadder Entity" to select this module for simulation, as shown in Fig. 1.24. Click OK to start the simulation, and ModelSim

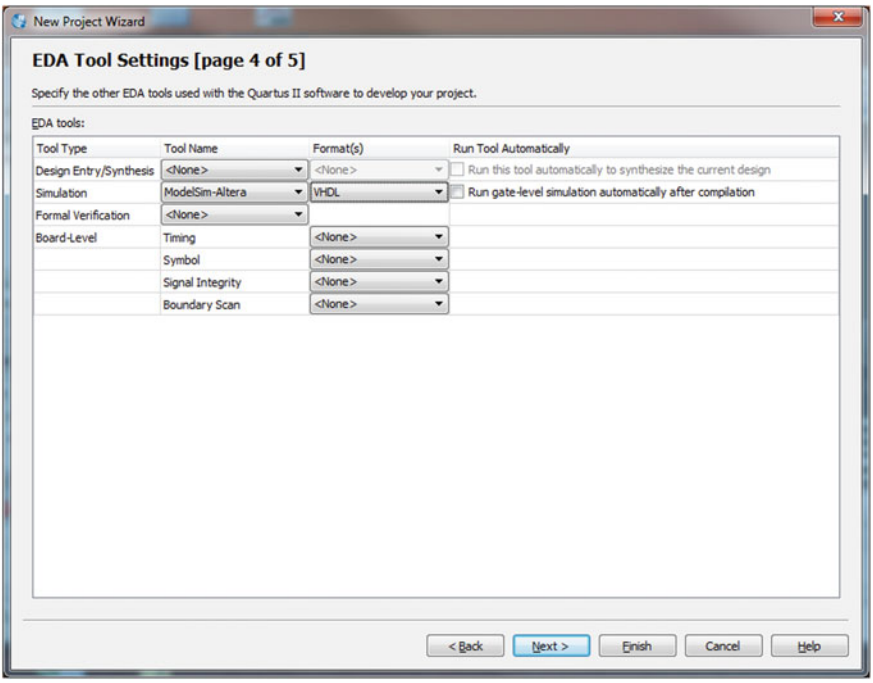


Fig. 1.15 Selecting the simulation tool—ModelSim-Altera

will open several new windows on the left side of Fig. 1.25. The "sim" window lists the hierarchical view of the circuit, and it is used in order to select the signals to be shown during the simulation. When the half-adder module is highlighted, the Objects window lists the signals that can be observed and changed in the simulation process. The signals of interest for the half-adder simulation include A0, B0, S0 and C1.

As shown in Fig. 1.26, to add signals to the simulation window, select the signals in the Objects window using the mouse pointer, and with a right click select *Add→To Wave→Selected Signals* from the pop-up menu. After have added signals A0, B0, S0 and C1, they will show in a waveform window.

In ModelSim the simulation process can be done manually or through scripts. The script language is relatively intuitive, but for the basic circuits introduced in most laboratory sessions it would be overwhelming. The manual option is described next.

The waveform window can be undocked using the button in the upper right hand corner, as shown in Fig. 1.27. To organize the signals (it is optional), place the mouse on the A0 signal, hold down the left button, and drag the signal to the desired position.

To set a signal to 0 or 1, right click on the signal and select the Force option from the pop-up menu. In Fig. 1.27, the B0 signal is "forced" to 1 (one). Next,

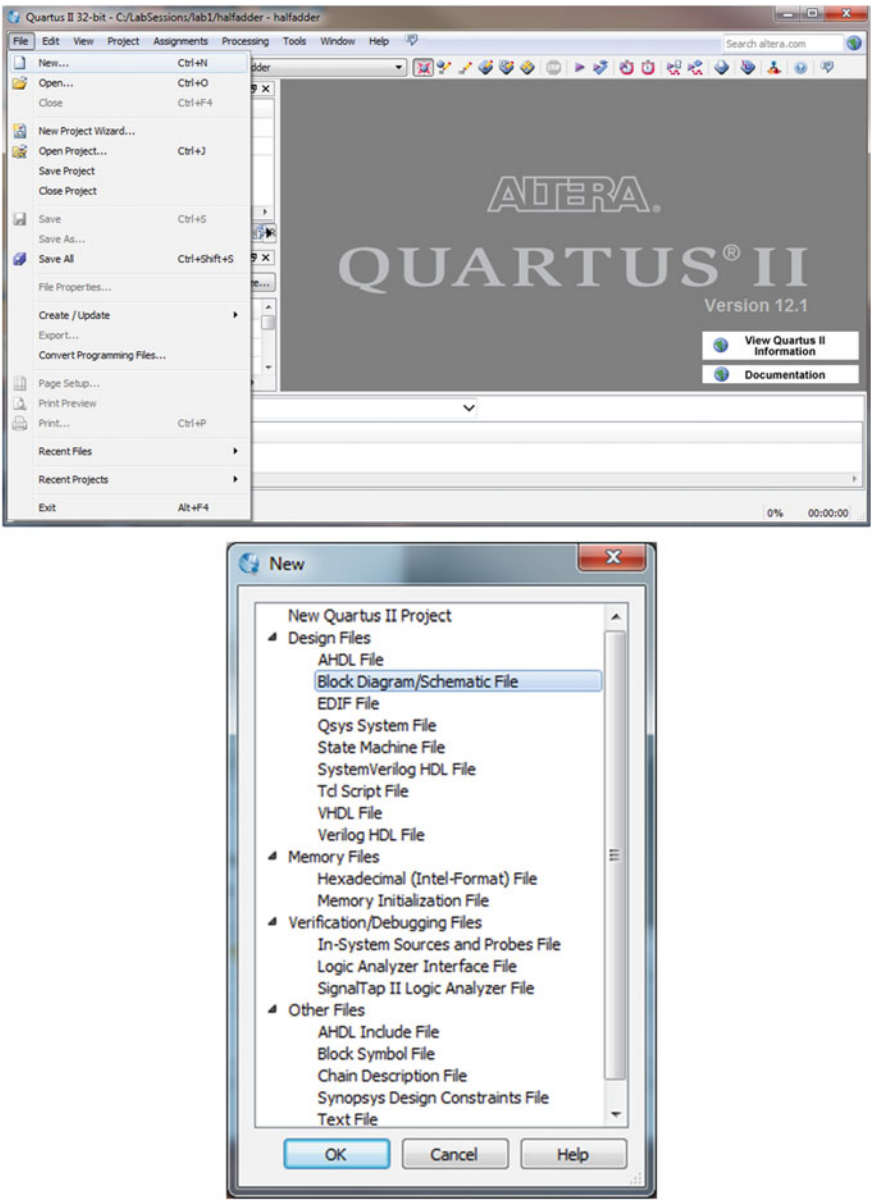


Fig. 1.16 New schematic file

repeat the operation for signal A0, but forcing it to 0 (zero). Set the simulation run length to 100 ps, and press the Run button. This will make the simulation run for 100 ps. In Fig. 1.27 there is an arrow pointing at the Run button. The remaining simulation buttons are:

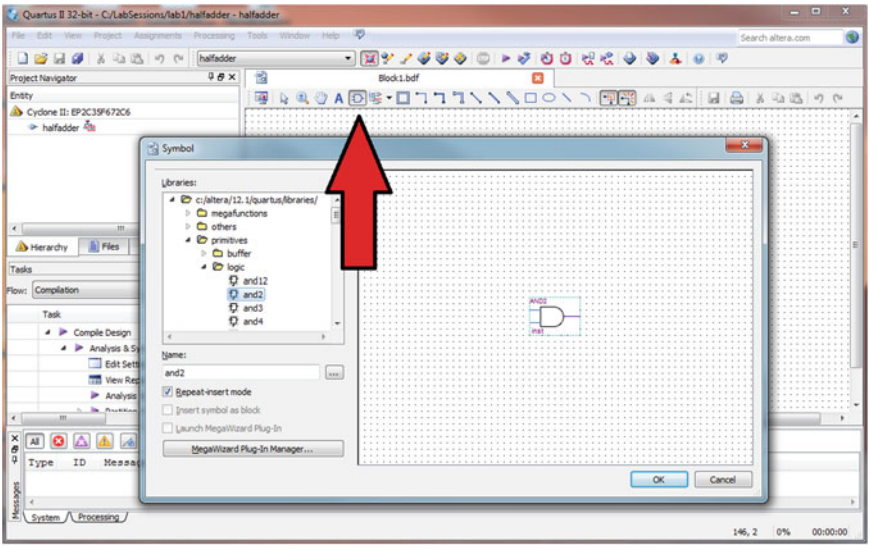


Fig. 1.17 Symbol tool icon

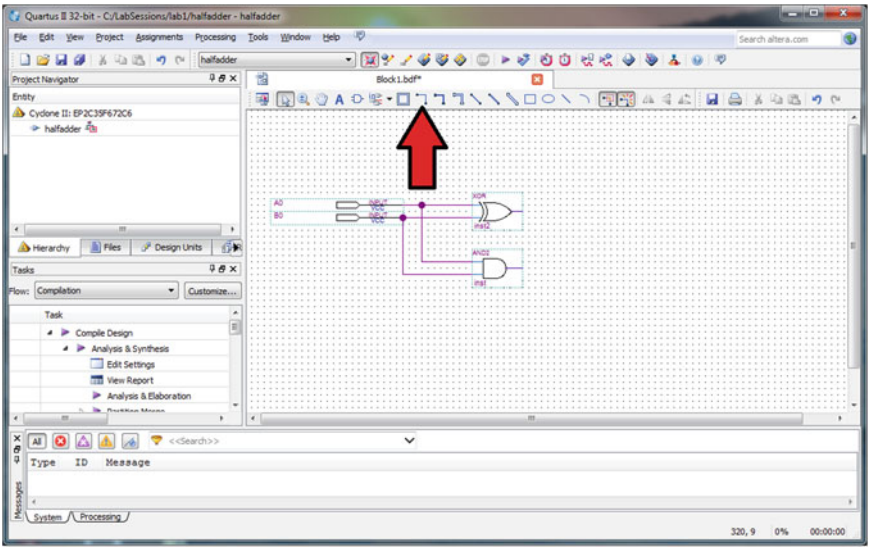


Fig. 1.18 Schematic entry

- Restart simulation button—it is located on the left of "Run length" (100 ps);
- Run length text box—defines for how long the simulation will run;
- Run button—runs the simulation for the duration specified in the Run length text box;

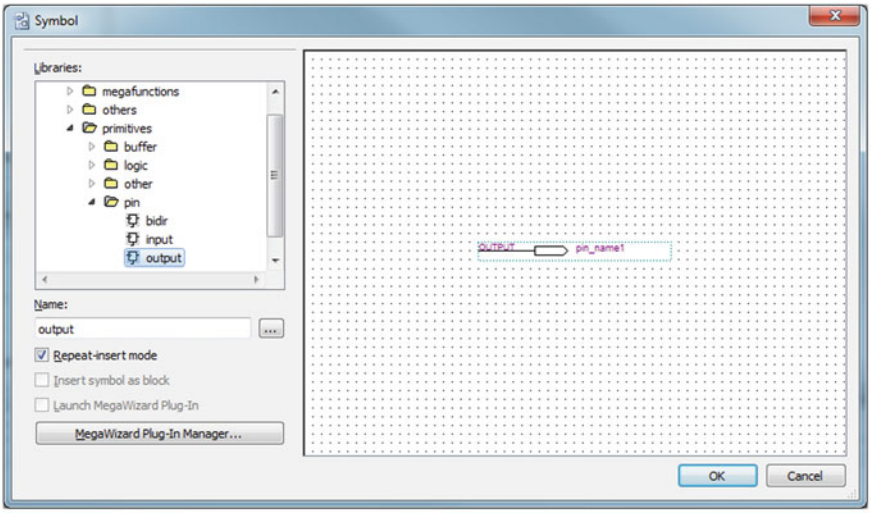


Fig. 1.19 Input and output pins

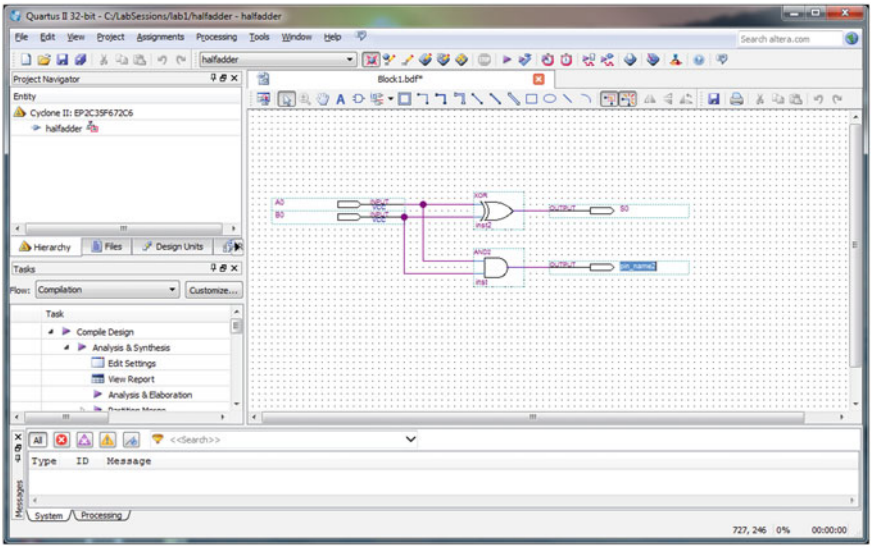


Fig. 1.20 Renaming the input and output pins

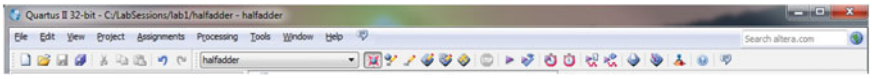


Fig. 1.21 Quartus II menu bar

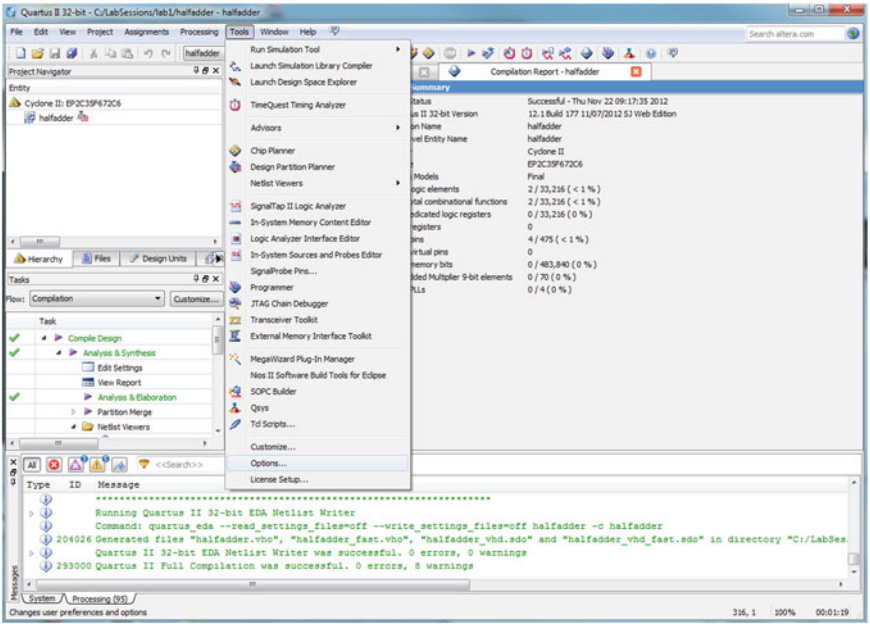


Fig. 1.22 ModelSim-Altera installation in Quartus II

- Continue run button—used to resume a simulation;
- Run—All button—used to start a simulation;
- Break button—used to interrupt a simulation.

In Fig. 1.28, the input signals A0 and B0 are changed to 0 and 1, respectively, and the Run button is pressed again, making the simulation run for another 100 ps period.

Figure 1.29 shows the four possible inputs for A0 and B0, according to the truth table in Fig. 1.11. The functional simulation shows that the half-adder design presents the expected behavior, but it does not considers time constraints which may result in unexpected delays and, consequently, in a faulty circuit. The timing simulation can be used to check this situation, but in this laboratory session it will not be executed.

The next step is the physical implementation of the circuit, using the FPGA board. Close (quit) all ModelSim windows, and go back to the Quartus II environment.

Step 5: FPGA Prototyping

Using Quartus II, choose menu *Assignments→Pin Planner*. As shown in Figs. 1.30 and 1.31, this tool is used to associate the input and output signals defined in the half-adder schematic design, to the actual FPGA pins. The list of FPGA pins and their connections to the DE2 resources (peripherals) can be found in the

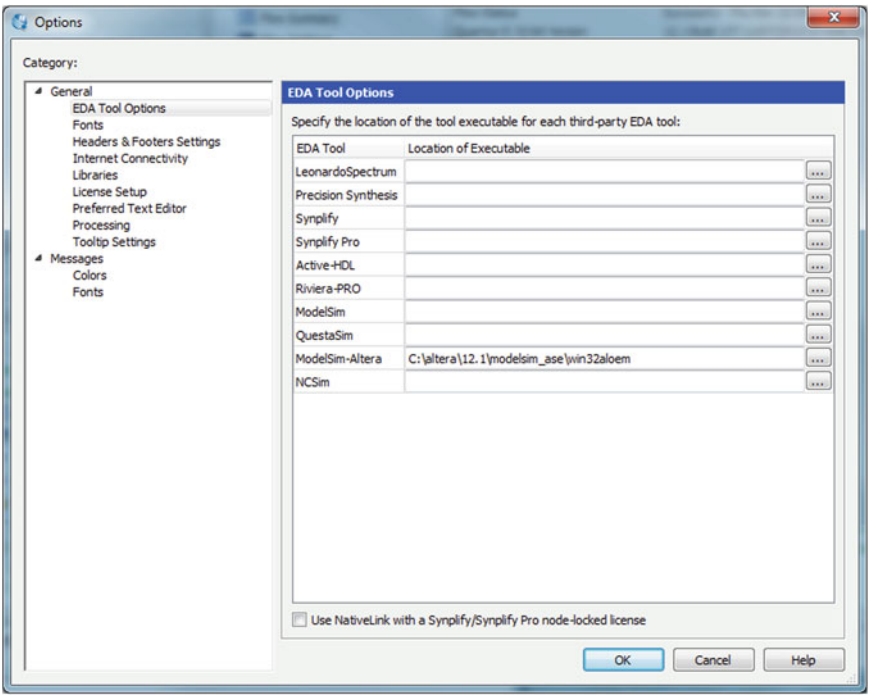


Fig. 1.23 ModelSim-Altera path configuration

board's user manual available on the DE2 folder: DE2\DE2_user_manual\ DE2_UserManual.pdf.

In DE2 user manual there is a list of all available pins, with the respective connection between the FPGA pins and the board's peripherals. For instance, SW[0] and SW[1] switches are connected to the N25 and N26 FPGA pins (see DE2_UserManual.pdf Table 4.1 in page 28). The half-adder output signals can be shown on LEDR[0] (sum) and LEDR[1] (carry), which are connected to FPGA pins AE23 and AF23 (see DE2_UserManual.pdf Table 4.3 in page 29).

Using the Pin Planer tool, perform these pin assignments in the Location column, as shown in Fig. 1.31, and close the Pin Planner.

Perform a new synthesis (button Compile in Quartus II menu), in order to assign the physical pins to the design. It will generate the final netlist that can be downloaded to the FPGA.

After the user has powered on and connected the DE2 board to the USB (use the "blaster" USB connector), press the red button to switch on the board, and run the programming tool: Menu *Tools→Programmer* (or fourth icon from right to left in Fig. 1.21). The window in Fig. 1.32 shows the Programmer tool interface. In order to perform a download of a configuration file to the FPGA, be sure to have the

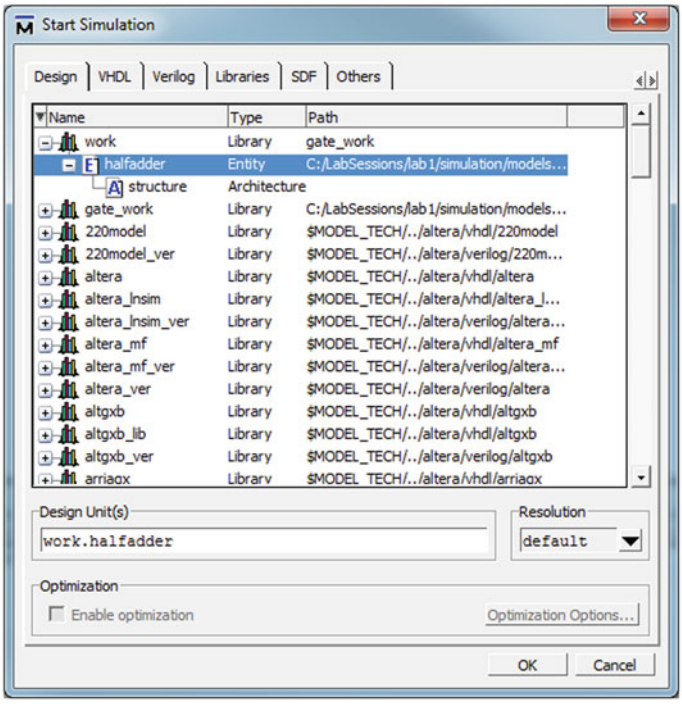


Fig. 1.24 Start simulation window

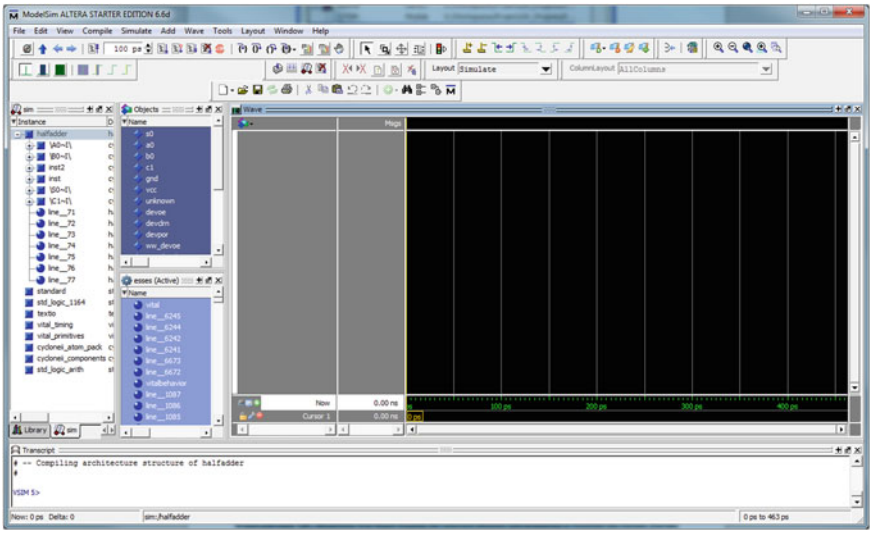


Fig. 1.25 Simulation environment

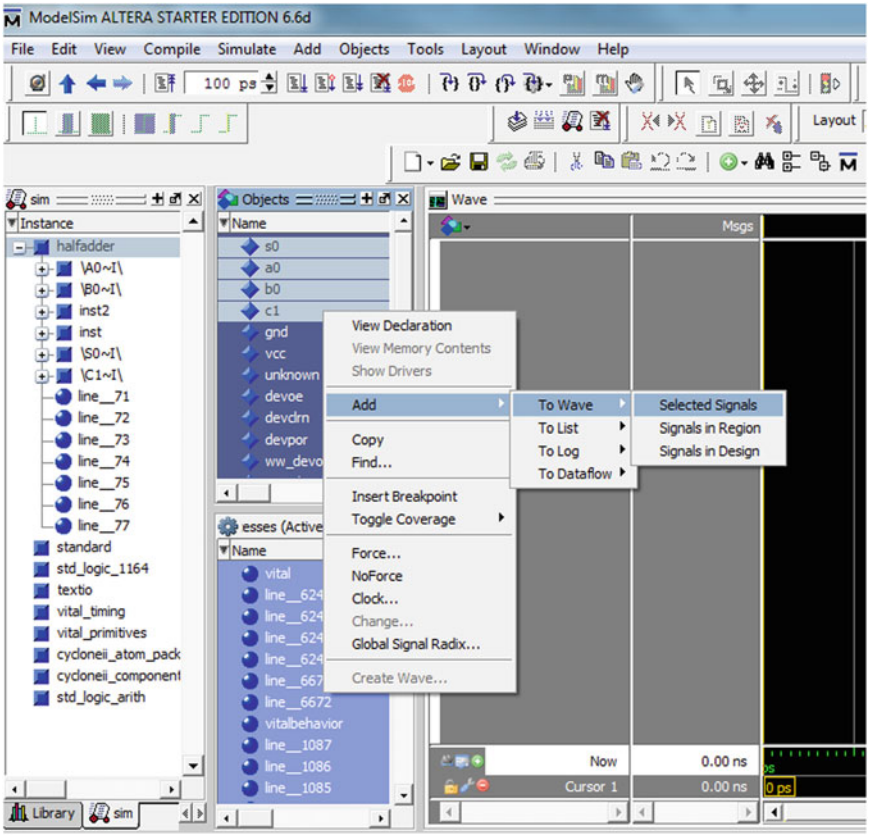


Fig. 1.26 Selecting signals for simulation

"USB-Blaster [USB-0]" message showing next to the "Hardware Setup" button, and that the configuration file "halfadder.sof" is listed in the File column.

The "Start" button is used to perform the download of the halfadder.sof to the FPGA configuration memory (Fig. 1.33).

Some problems that may happen regarding the FPGA Programmer tool are listed next:

- The USB port has not been found (DE2 board not detected). Solution: start "Devices and Printers" in Windows start menu and proceed with the new driver installation. The DE2 driver is located in the Quartus II folder: C:\altera\12.1\quartus\drivers. For Linux systems, follow the standard procedure for driver installation.
- The USB-Blaster [USB-0] is not shown. Solution: certify that the USB cable is connected to the "Blaster" input. Press the "Hardware Setup" button, double-click on "USB-Blaster", and close the Hardware Setup window.

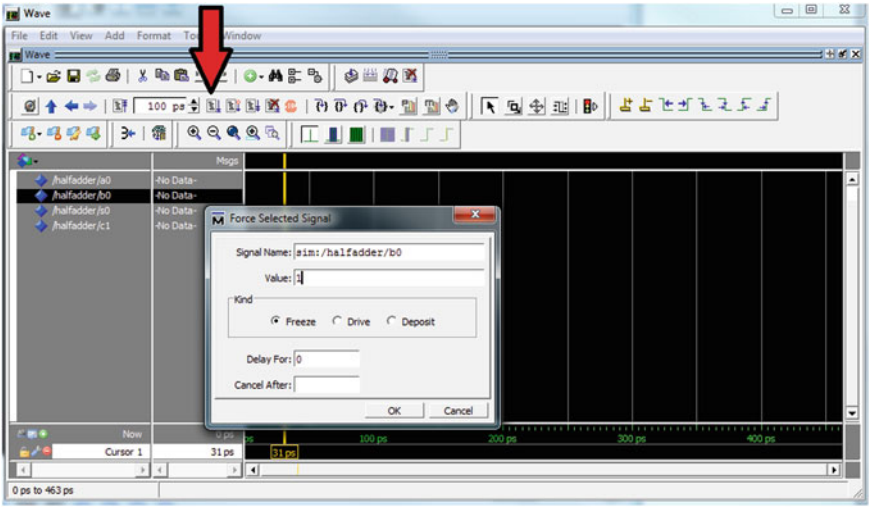


Fig. 1.27 Waveform window

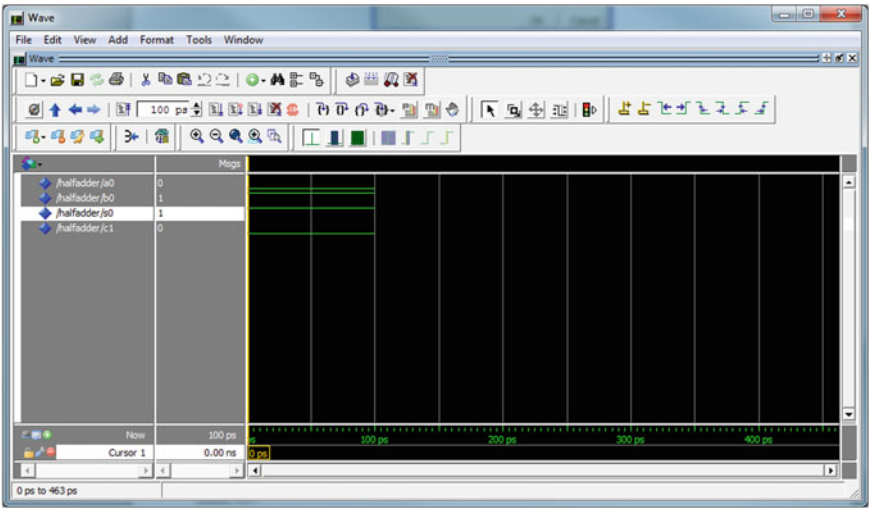


Fig. 1.28 A0 = 0 and B0 = 1, resulting in S0 = 1 and C1 = 0

- The halfadder.sof file is not shown. Solution: press the "Add File" button, go to c:\LabSession\lab1\output_files, and double-click on halfadder.sof

After have finished the download to the FPGA, in order to check the circuit functionality in the board, just switch inputs SW0 (A0) and SW1 (B0) "on" and

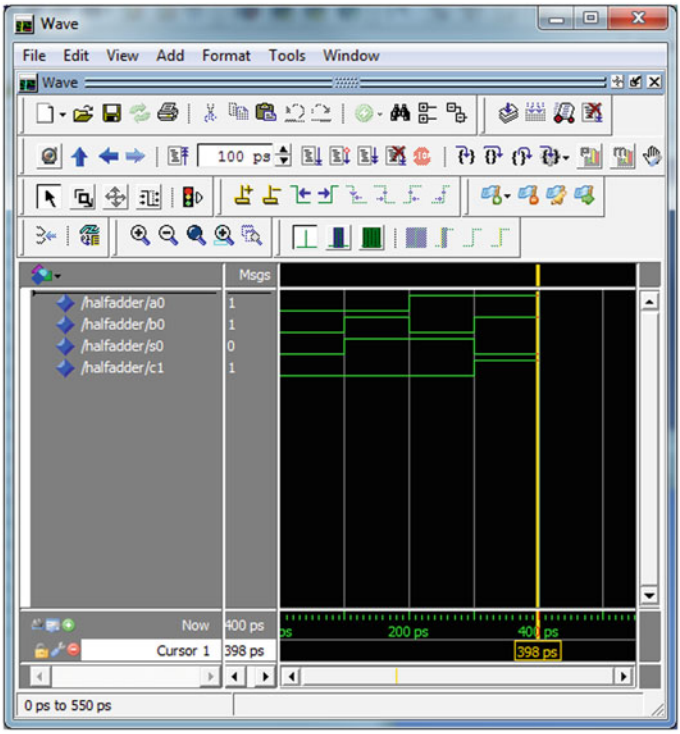


Fig. 1.29 All four combinations for A0 and B0

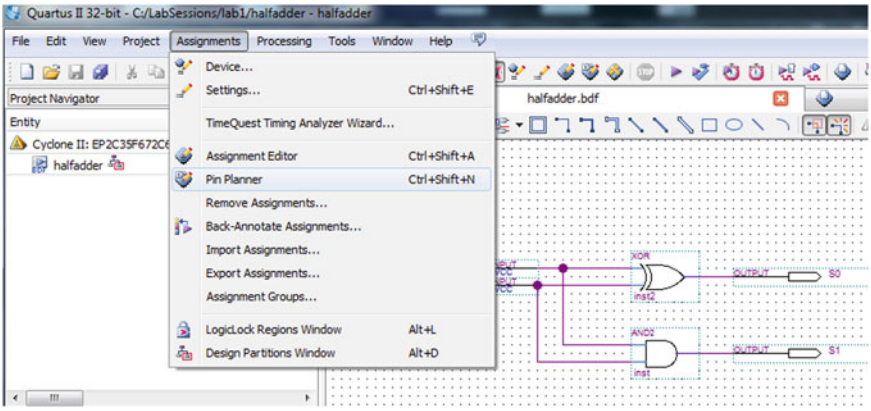


Fig. 1.30 FPGA pin assignments

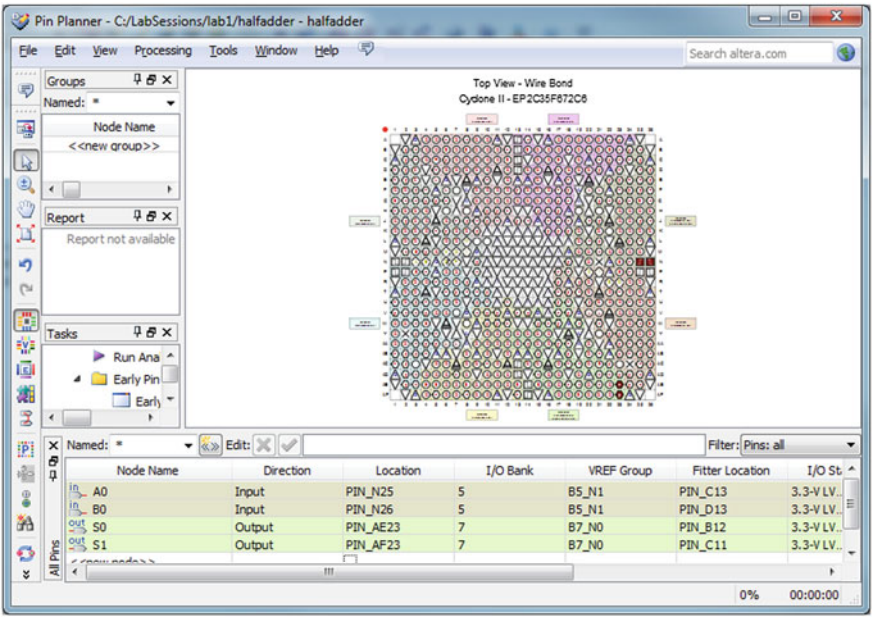


Fig. 1.31 Half-adder pin assignment

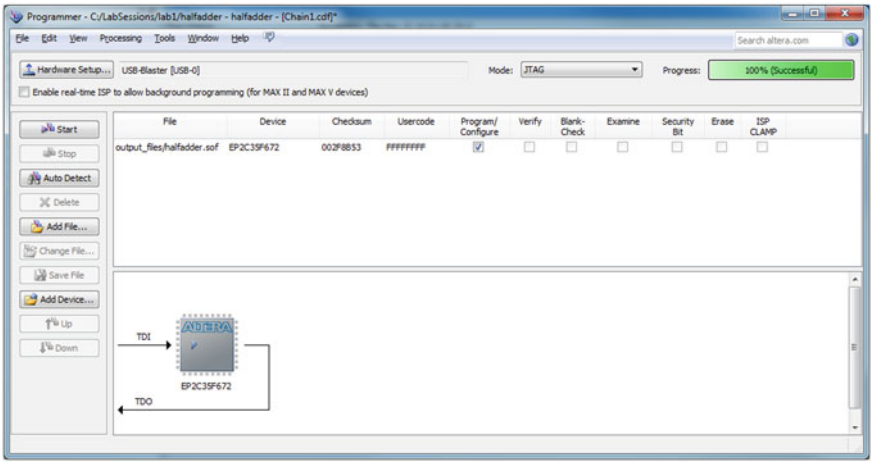


Fig. 1.32 Quartus II FPGA programming tool

"off", and make a comparison between the results showed in the red LEDs (Light-Emitting Diode) with the truth table shown in Fig. 1.11.

A summary of the activities developed in this laboratory session is as follows:

1. Project creation *File→New Project Wizard*

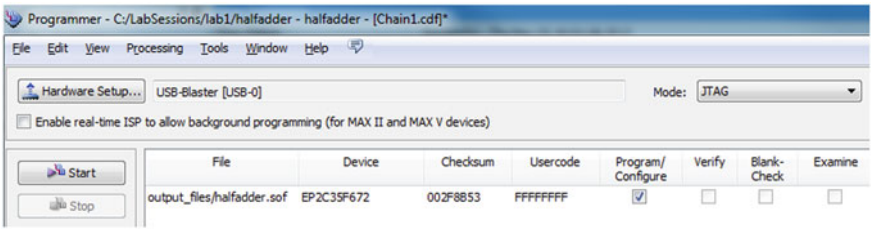


Fig. 1.33 FPGA programmer

2. In "project wizard", follow *exactly* the steps listed in this tutorial. Any oversight may result in errors in the hardware generation for the FPGA.
3. Design Entry (schematic) *File→New→Block Diagram*
4. Draw the half-adder schematic (Fig. 1.11).

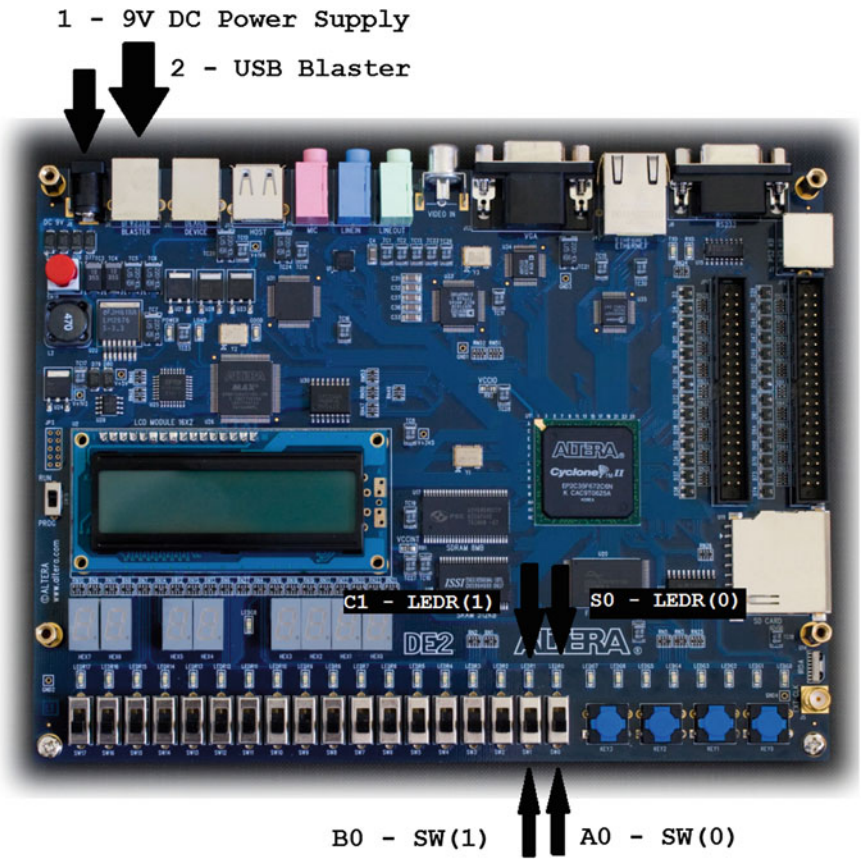


Fig. 1.34 DE2 board

5. Compile the circuit (synthesis).
6. Perform the functional simulation using ModelSim (no timing information).
7. Perform the pin assignment, which is the interface between FPGA's internal signals (half-adder input/output signals) and the board switches and LEDs—*Assignments—Assignment Editor (or Pin Planer)*.
8. Compile the circuit again, in order to generate the final circuit (with pin assignments) to be used in the FPGA. In the physical synthesis (place and route), the logic elements defined in the logic synthesis are placed using the physical resources found in the FPGA. In this process, all connections (routing) between logic elements are performed.
9. Temporal Analysis—Analysis of propagation delays of signals after physical synthesis resulting in a report consisting of the expected performance.
10. Programming—The synthesized circuit is downloaded to the FPGA—*Tools—Programmer. Hardware Setup—USB-Blaster. Start!*
11. Testing the prototyped circuit—Fig. 1.34 shows the power connector (1), the USB blaster port (2), the inputs SW(0) and SW(1), and the outputs LEDR(0) and LEDR(1). Switch the inputs on and off, and observe the outputs (sum and carry) in the LEDs, according to the truth table in Fig. 1.11.

Chapter 2
HDL Based Designs

In this chapter the FPGA design flow is revisited, but using a hardware description language for the design entry instead of the schematic diagram discussed in Chap. 1. At the end of the chapter, the reader should be able:

- to understand the basic structure of a VHDL description;
- to model a simple digital circuit in VHDL;
- to follow the VHDL design flow;
- to simulate and test a digital circuit designed in VHDL;
- to prototype a circuit described in VHDL, using an FPGA board.

2.1 Theoretical Background

A Hardware Description Language (HDL) is a modeling language used to describe the structure and behavior of a digital circuit. In addition, an HDL allows the test of the designed circuits through simulations. Hardware description languages express a temporal behavior and circuit structure in a text format. Additionally, the syntax and semantics of HDLs include notations to express temporal sequences and concurrences, as required by a hardware module.

A good example of hardware description language is VHDL, which is an acronym for VHSIC Hardware Description Language. VHSIC stands for Very High Speed Integrated Circuits and it was a government program of USA in the early 1980s. Later, the VHDL language became an IEEE (Institute of Electrical and Electronic Engineers) standard and nowadays there are several tools to simulate and synthesize (i.e., generate hardware) circuits described in VHDL. Other hardware description languages are SystemC, Handel-C, Verilog, and System Verilog.

In VHDL, the design of a digital circuit can be described in two abstraction levels, that is, structural or behavioral. Descriptions at the register transfer level (RTL) are widely used to develop digital systems. VHDL is not equivalent to a

E. A. Bezerra and D. V. Lettnin, *Synthesizable VHDL Design for FPGAs*,
DOI: 10.1007/978-3-319-02547-6_2,

software programming language and the synthesis tools are not equal to the compiler tools, since they do not generate executable code from a VHDL description. Additionally, these descriptions can be used to generate a piece of hardware, for example, a file for configuring an FPGA.

VHDL descriptions can be simulated, that is, run in a simulator. The testing process can be performed via *testbenches* where stimuli are generated for simulating VHDL descriptions. A testbench defines the external stimulus to be used as test cases to a circuit input. The testbenches can be written in VHDL or in many other languages (e.g. C, C++, and SystemC).

The VHDL models consist of two main parts, the *entity* and the *architecture*. As shown in Fig. 2.1, the entity part of a VHDL description has only the input and output pins (interface) to be used by the circuit. It has no information at all regarding the circuit's internal logic (circuit operation). In the architecture part, as shown in Fig. 2.2, there is the operational description of the circuit. Figure 2.3 shows the complete VHDL description for the half-adder circuit. It is important to notice that in this example, the *std_logic* type has been used and, for this reason, its library has to be included in the description. As a result, a VHDL description usually has the following sessions:

- **Library** definition—includes the required libraries and packages, for instance, the IEEE package (see lines 1–2 in Fig. 2.3);
- **Entity**—defines the "pins" of the digital circuit (signals), i.e. the interface between the implemented logic and the external world (see lines 4–11 in Fig. 2.3);
- **Architecture**—defines the functionality of a digital circuit, using the input and output "pins" that are listed at the entity part (see lines 13–17 in Fig. 2.3). Thus, an entity may have different implementations (i.e., architectures) for the same function.

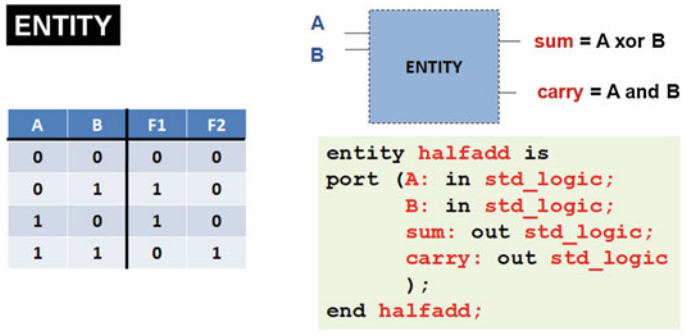
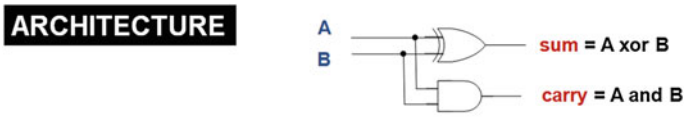


Fig. 2.1 "Entity" declaration for the half-adder circuit




```
architecture ha_stru of halfadd is
begin
  sum <= A xor B;
  carry <= A and B;
end ha_stru;
```

Fig. 2.2 "Architecture" declaration for the half-adder circuit

Fig. 2.3 Example of entity and architecture in VHDL. Sum and carry are assigned in parallel

```
1   library IEEE;
2   use IEEE.Std_Logic_1164.all;
3
4   entity halfadder is
5   port (
6       A    : in std_logic;
7       B    : in std_logic;
8       sum : out std_logic;
9       carry    : out std_logic
10      );
11  end halfadder;
12
13  architecture ha_stru of halfadder is
14  begin
15      sum <= A xor B;
16      carry <= A and B;
17  end ha_stru;
```

2.2 Laboratory Assignment

The laboratory objectives are:

- to allow the reader to have a first contact with the VHDL language;
- to provide a first contact with an EDA tool;
- to continue the training and understanding of a complete FPGA design flow, but this time starting from a VHDL description design entry.

2.2.1 Laboratory Session

The tasks to be completed in this laboratory session, using Altera's Quartus II EDA tool, are as follows:

- Create the half-adder circuit shown in Fig. 2.3 using the VHDL design entry editor;
- Perform the synthesis;
- Fix simulation and synthesis errors;
- Prototype and test the half-adder circuit in the FPGA board.

Step 1: Creating a New Project
This step is exactly the same as "Step 1" described in Chap. 1. It is important to remember to type "halfadder" as the project's name (page 1 of 5 in the New Project Wizard—see Fig. 1.13).
Step 2: Design Entry (VHDL)
Chose *File* → *New* and *VHDL File*, as shown in Fig. 2.4. This will open an empty text editor.

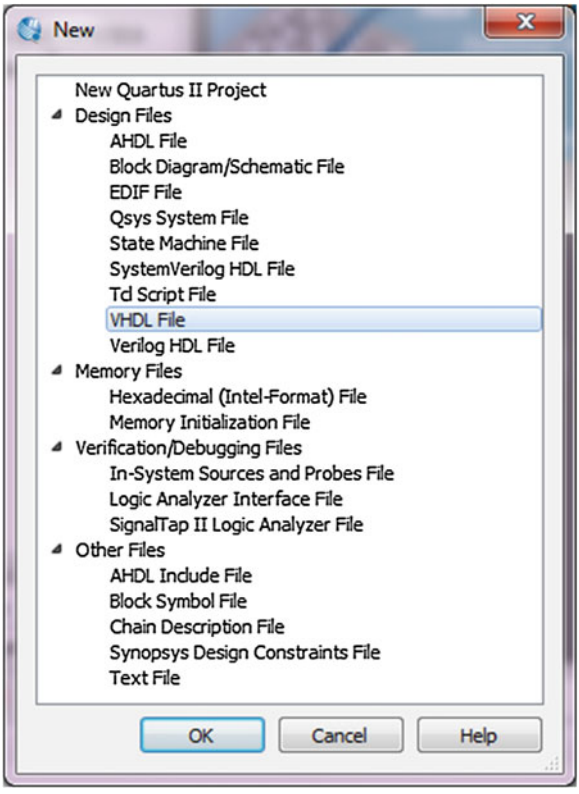


Fig. 2.4 New VHDL file

Type in the text editor the half-adder VHDL code shown in Fig. 2.3, and save it: *File* → *Save* (or just ctrl-S). The default name to save the file is halfadder.vhd, as it is the name provided as the project name in Step 1. The file will be saved, also in the default location, in the project folder defined in Step 1. Be sure that the check box "Add file to current project" is checked. Again, at this point it is important to be sure that the entity's name is also "halfadder", otherwise, the synthesis tool will not be able to find the entity to be synthesized.

Step 3: Synthesis

This step is the same as "Step 3" described in Chap. 1, that is, choose *Processing* → *Start Compilation* (or Ctrl-L). In case of errors, read carefully the "Processing" messages (bottom window), and fix them in the VHDL source code (in the text editor). There will be some warning messages, and at this point look just for the "critical" ones.

Step 4: Simulation

Before starting the simulation, see "Step 4" in Chap. 1 in order to configure simulator options. After that, to run ModelSim, in Quartus II menu choose *Tools* → *Run Simulation Tool* → *RTL Simulation* (or press the 7th icon from right to left shown in Fig. 1.21). Next, follow the instructions provided in Chap. 1, "Step 4":

- Simulate → Start Simulation;
- In the "Start Simulation" window, look for the "Design" tab (see Fig. 1.24);
- In the "work" Library, click on the "+" sign, and select the "halfadder" entity as shown in Fig. 1.24;
- Click OK to start the simulation;
- Select the *A*, *B*, *sum* and *carry* signals in the Objects window, and add them to the Wave window, as shown in Fig. 2.5;
- Set the *A* signal to 0, and the *B* signal to 0, using the "Force" option as explained in Chap. 1, "Step 4". Set the simulation run length to 100 ps, and press the *Run* button. Next, change *A* to 0 and *B* to 1, and run for another 100 ps. Change *A* to 1 and *B* to 0, and run for 100 ps. Finally, change *A* to 1 and *B* to 1, and run for 100 ps.
- The expected simulation results are shown in Fig. 2.6.

Step 5: FPGA Prototyping

The first task in order to build a physical design to prototype in an FPGA board is to specify which entity signal goes to which FPGA pin. For instance, in the half-adder design, if nothing is done regarding the *A*, *B*, *sum* and *carry* signals, the synthesis tool will randomly allocate pins in the FPGA device, and the design may not work in the target board. In Chap. 1 the half-adder entity signals were assigned to the physical (FPGA) pins using the Pin Planner tool (see Fig. 1.31). This is an adequate strategy for designs with few external interface pins, which is the case of the half-adder circuit. However, this is not the case in real world designs were dozens or even hundreds of signals are used in their top-level entities. For this reason, for the VHDL version of the half-adder design an automatic pin assignment strategy will be adopted.

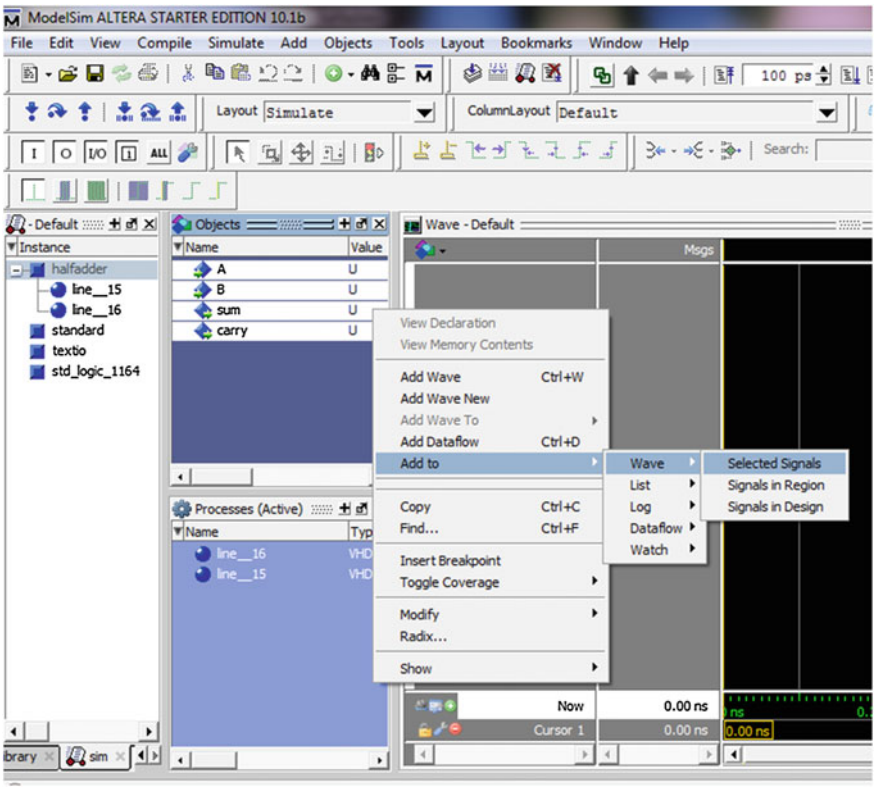


Fig. 2.5 Selecting signals from the half-adder design

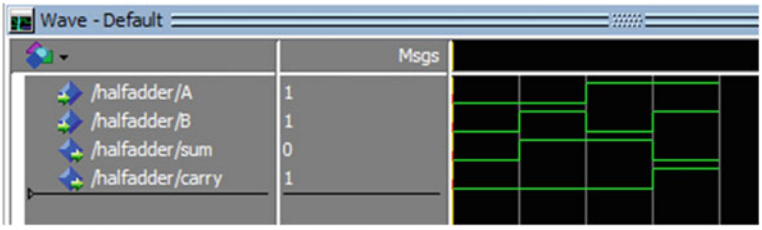


Fig. 2.6 Simulation results for the half-adder in VHDL

The listing of all pins available in the DE2 board is included in a text file. In this listing, each pin has a specific name, and the signals of the half-adder entity should be renamed according to the FPGA board nomenclature. Alternatively, the pin names in the file can be renamed to match the signal names in the design entity, but this is not a common practice.

Each FPGA vendor has its own format, but usually only the relation between the signal name and the target pin is specified. There is no indication regarding the pin direction (input, output).

Xilinx pin assignment files have the ".ucf" extension, and lines such as:

```
NET "sum" LOC = "P28";
NET "carry" LOC = "P110";
```

In this case, the sum and carry signals of the half-adder entity will be assigned to FPGA pins 28 and 110, respectively.

Altera has a different format for their pin assignments files, as shown next:

```
set_location_assignment PIN_N25 -to A
set_location_assignment PIN_N26 -to B
set_location_assignment PIN_AE23 -to sum
set_location_assignment PIN_AF23 -to carry
```

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

The full list of available pins on the board is included in the *DE2_pin_assignments.qsf* file, which can be found in the official website for this book. The Quartus II Setting File (".qsf") can be downloaded from the book's website and copied to the project's folder (e.g., C:\LabSessions\halfadder), the FPGA pins listed in this file must be imported into the design: menu *Assignments* → *Import Assignments*. Browse to find the folder, and press OK as shown in Fig. 2.7.

This file has 425 pins listed, and each pin is assigned to a pre-defined label. The N25, N26, AE23 and AF23 FPGA pins are assigned to the labels SW[0], SW[1], LEDR[0] and LEDR[1], as shown next:

```
set_location_assignment PIN_N25 -to SW[0]
set_location_assignment PIN_N26 -to SW[1]
…
set_location_assignment PIN_AE23 -to LEDR[0]
set_location_assignment PIN_AF23 -to LEDR[1]
…
```

In the DE2 board Printed Circuit Board (PCB) design, the FPGA pin N25 is connected to the switch SW_0 shown at the bottom of Fig. 1.34. The FPGA pin N26 is connected to switch SW_1, and so on.

In the *DE2_pin_assignments.qsf* file, the label SW represents a vector with 18 positions (0–17). LEDR is also a vector with 18 positions.

To perform the connection between the physical world (DE2 switches and LEDs) and internal FPGA signals, a VHDL entity should include the input and output pins available on the board. As shown in Fig. 2.8, on line 6 the label SW is used. SW is an 18 bits vector, and it has the same name (and size) as the SW

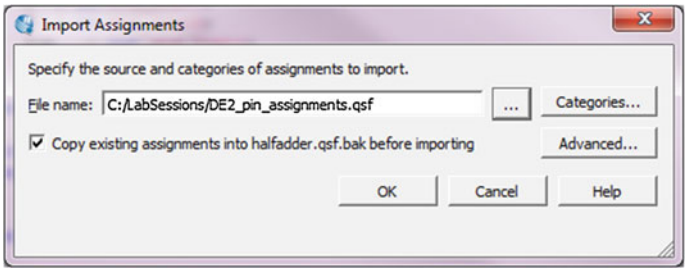


Fig. 2.7 Import assignments for the half-adder design

symbol connected to the FPGA switch pins listed in the *DE2_pin_assignments.qsf* file. On line 7, label LEDR is used in order to match the name used in the pin assignments file. The VHDL commands in lines 13–15 write to the red LEDs: switch values (line 13); a constant value (line 14), and the results of a logical operation (line 15).

In order to adapt the half-adder circuit to the DE2 board, the design's entity should be changed according to the *DE2_pin_assignments.qsf* file contents. The changes consist in renaming the *A*, *B*, *sum* and *carry* signals to *SW(0)*, *SW(1)*, *LEDR(0)*, and *LEDR(1)*, respectively. The changes are shown in Fig. 2.9.

After have concluded the modifications in the VHDL source code, perform a new synthesis (Compile button in Quartus II menu), and download the generated configuration file to the FPGA. The steps to perform the download are the same as "Step 5" in Chap. 1 (see Figs. 1.32 and 1.33).

To test the circuit, follow the instructions provided at the end of Chap. 1, using the switches and LEDs as shown in Fig. 1.34, where *A* is *SW(0)*, *B* is *SW(1)*, *sum* is *LEDR(0)*, and *carry* is *LEDR(1)*.

```
1  library IEEE;
2  use IEEE.Std_Logic_1164.all;
3
4  entity DE2_pins is
5  port (
6      SW  : in  std_logic_vector(17 downto 0);
7      LEDR    : out std_logic_vector(17 downto 0)
8      );
9  end DE2_pins;
10
11 architecture logic_circuit of DE2_pins is
12 begin
13     LEDR(7 downto 0) <= SW(15 downto 8);
14     LEDR(15 downto 8) <= "01010101";
15     LEDR(17) <= (SW(17) AND SW(0)) OR SW(16);
16 end logic_circuit;
```

Fig. 2.8 DE2 pins declared in an entity

```
1  library IEEE;
2  use IEEE.Std_Logic_1164.all;
3
4  entity halfadder is
5  port (                                          -- A -> SW(0)
6      SW  : in  std_logic_vector(17 downto 0);    -- B -> SW(1)
7      LEDR: out std_logic_vector(17 downto 0)     -- sum   -> LEDR(0)
8      );                                          -- carry -> LEDR(1)
9  end halfadder;
10
11 architecture ha_stru of halfadder is
12 begin
13   LEDR(0) <= SW(0) xor SW(1);      -- sum   <= A xor B
14   LEDR(1) <= SW(0) and SW(1);      -- carry <= A and B
15 end ha_stru;
```

Fig. 2.9 Half-adder design adapted to DE2 board pin names

2.2.2 Going Beyond

The DE2 board has an LCD which can be programmed and written manually, using the available switches. The VHDL code to access the LCD pins is shown in Fig. 2.10.

After have synthesized and downloaded the generated configuration file to the FPGA board, follow the next steps to write control and data to the LCD.

```
library ieee;
use ieee.std_logic_1164.all;
entity LCD is
port (
      LCD_DATA    : out std_logic_vector(7 downto 0);
      LCD_RW      : out std_logic;
      LCD_EN      : out std_logic;
      LCD_RS      : out std_logic;
      LCD_ON      : out std_logic;
      LCD_BLON    : out std_logic;
      SW          : in std_logic_vector(17 downto 0)
);
end LCD;
architecture LCD_WR of LCD is
begin
      LCD_ON   <= SW(17);
      LCD_BLON <= SW(16);
      LCD_DATA <= SW(7 downto 0);
      LCD_RS   <= SW(8);
      LCD_EN   <= SW(9);
      LCD_RW   <= SW(10);
end LCD_WR;
```

Fig. 2.10 VHDL code for LCD writing in DE2 board

Step 1: Write Command 1 (38H)—Switches the LCD ON, Cursor ON, and Blink ON

Switch (SW)	Value (switch position)	Effect
17	1	LCD_ON
16	1	LCD_BLON
7..0	0011 1000	Command
8	0	LCD_RS (0 = control)
9	0 → 1 → 0	LCD_EN
10	0	LCD_RW

Step 2: Write Command 2 (0FH)—Switches the LCD ON, Cursor ON, and Blink ON

Switch (SW)	Value (switch position)	Effect
17	1	LCD_ON
16	1	LCD_BLON
7..0	0000 1111	Command
8	0	LCD_RS (0 = control)
9	0 → 1 → 0	LCD_EN
10	0	LCD_RW

Step 3: Write Command 3 (06H)—Switches the LCD ON, Cursor ON, and Blink ON

Switch (SW)	Value (switch position)	Effect
17	1	LCD_ON
16	1	LCD_BLON
7..0	0000 0110	Command
8	0	LCD_RS (0 = control)
9	0 → 1 → 0	LCD_EN
10	0	LCD_RW

Step 4: Write Data to LCD

Switch (SW)	Value (switch position)	Effect
17	1	LCD_ON
16	1	LCD_BLON
7..0	0100 0001	'A' or 41H (ASCII)
8	1	LCD_RS (1 = data)
9	0 → 1 → 0	LCD_EN
10	0	LCD_RW

Step 5: Clear the LCD

Switch (SW)	Value (switch position)	Effect
17	1	LCD_ON
16	1	LCD_BLON
7..0	0000 0001	Clear command
8	0	LCD_RS (0 = control)
9	0 → 1 → 0	LCD_EN
10	0	LCD_RW

Chapter 3
Hierarchical Design

This chapter introduces the concept of hierarchical design in digital systems. A full VHDL project made of several components is developed as a case study, and its components are glued together using the VHDL reserved words *component* and *portmap*. These VHDL statements are used in all remaining projects throughout this book. At the end of the chapter, the reader should be able:

- to identify and comprehend the use of components in VHDL;
- to understand the concept of hierarchical design, and its implementation in VHDL;
- to add a component to the proposed case study;
- to simulate and test both, the original and the modified version of the case study;
- to prototype the case study using an FPGA board.

3.1 Hierarchical Design in VHDL

A digital system can be seen as a set of components interconnected in order to perform some required functionality. As applications grow in complexity, so do the number of components in a design. To deal with system complexity, VHDL offers mechanisms for the design organization. The language has appropriate structures allowing complex components to be built from less complex components. The hierarchical design methodology is not a novelty in traditional hardware and software systems. In a similar way, in VHDL a component oriented solution results not only in a more organized design, but also in more opportunities for components reusability. Components already employed in other systems may have been tested and verified, shortening the design cycle.

In order to introduce the VHDL structure and statements used in the hierarchical design methodology, consider the development of a digital system to implement the following equation:

$$F = (F1 \; and \; F2) \; or \; F3 \tag{3.1}$$

E. A. Bezerra and D. V. Lettnin, *Synthesizable VHDL Design for FPGAs*,
DOI: 10.1007/978-3-319-02547-6_3,

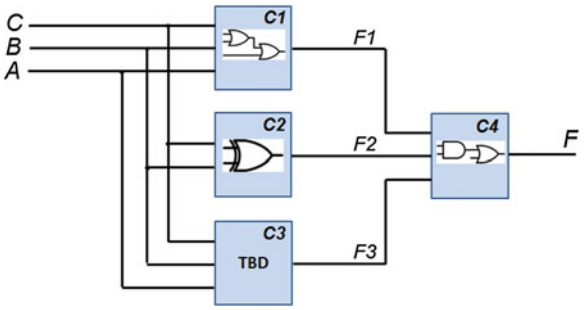


Fig. 3.1 Digital system design based on components

where $F1$, $F2$ and $F3$ are outputs in the following equations:

$$F1 = A\,or\,B\,or\,C \tag{3.2}$$

$$F2 = B\,xor\,C \tag{3.3}$$

$$F3 = \text{``}to\ be\ defined\text{''} \tag{3.4}$$

As shown in Fig. 3.1, the circuit to be designed in VHDL has four components: $C1$ implements Eq. (3.2); $C2$ implements Eq. (3.3); $C3$ implements Eq. (3.4) (TBD—*to be defined*); and $C4$ implements Eq.3.1. Component $C1$ has three inputs (A, B, C) and one output ($F1$). Component $C2$ has two inputs (B, C) and one output ($F2$). Component $C3$ has three inputs (A, B, C) and one output ($F3$). Component $C4$ has three inputs ($F1$, $F2$, $F3$) and one output (F).

In VHDL, each of these components is implemented through an *Entity/Architecture* construction. Component $C1$ is shown in Fig. 3.2, component $C2$ in Fig. 3.3, and component $C4$ in Fig. 3.4. Component $C3$ functionality will be defined next in Sect. 3.2.

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity C1 is
port (A: in std_logic;
      B: in std_logic;
      C: in std_logic;
      F: out std_logic
     );
end C1;

architecture c1_estr of C1 is
begin
  F <= A or B or C;
end c1_estr;
```

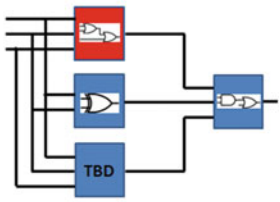


Fig. 3.2 VHDL implementation for component $C1$

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity C2 is
port (A: in std_logic;
      B: in std_logic;
      F: out std_logic
     );
end C2;

architecture c2_estr of C2 is
begin
  F <= A xor B;
end c2_estr;
```

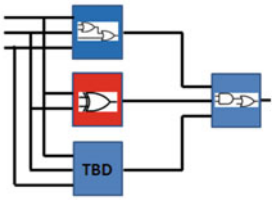


Fig. 3.3 VHDL implementation for component *C*2

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity C4 is
port (A: in std_logic;
      B: in std_logic;
      C: in std_logic;
      F: out std_logic
     );
end C4;

architecture c4_estr of C4 is
begin
  F <= (A and B) or C;
end c4_estr;
```

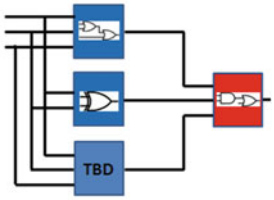


Fig. 3.4 VHDL implementation for component *C*4

Having all the required components, the next step in a hierarchical design is to glue the components together. As can be observed in Fig. 3.1, the four components are connected through wires. The three inputs to *C*4 are the wires named *F*1, *F*2 and *F*3, which are also the outputs for *C*1, *C*2 and *C*3. The inputs for *C*1, *C*2 and *C*3 are the signals *A*, *B* and *C*, and the circuit output is signal *F*. In VHDL, all these connections are defined using the *port map* statement. In order to use this statement, first it is necessary to list the components to be connected, using the *component* statement. Figure 3.5 has the VHDL code used to generate the circuit shown in Fig. 3.1. This VHDL code is usually known as the "top" component in a design hierarchy. The "top" component is the higher level one, and it has the purpose of instantiating and connecting lower hierarchy level components.

Fig. 3.5 VHDL code for inferring and connecting Fig. 3.1 components (top-level component)

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity top is
5    port ( A, B, C : in  std_logic;
6           F       : out std_logic
7    );
8  end top;
9
10 architecture top_map of top is
11    signal F1, F2, F3: std_logic;
12    component C1
13       port (   A : in  std_logic;
14                B : in  std_logic;
15                C : in  std_logic;
16                F : out std_logic
17       );
18    end component;
19    component C2
20       port (   A : in  std_logic;
21                B : in  std_logic;
22                F : out std_logic
23       );
24    end component;
25    --
26    -- ADD C3 HERE
27    --
28    component C4
29       port (   A : in  std_logic;
30                B : in  std_logic;
31                C : in  std_logic;
32                F : out std_logic
33       );
34    end component;
35 begin            -- begin architecture
36    L0: C1 port map (A, B, C, F1);
37    L1: C2 port map (B, C, F2);
38    --
39    -- ADD C3 CONNECTIONS HERE
40    --
41    L3: C4 port map (F1, F2, F3, F);
42 end top_map;     -- end architecture
```

In Fig. 3.5, components *C*1 to *C*4 are listed in the *architecture* section, just before the *begin* statement (lines 12–34). The VHDL syntax for defining the components is similar to the entity declaration. In fact, a good practice in order to avoid coding mistakes is to copy the desired component's entity section to the top-

level file (Fig. 3.5), replacing the word "entity" by "component", removing the "is" word, and adding an "end component" at the end of the declaration. Changes in a component header should be performed just in its entity, in order to avoid double changes mistakes. After have made the needed changes in an entity, just copy the whole entity section to the architecture section of the top-level file, and perform the modifications as stated before.

The component declarations in the architecture section of a top-level file inform the synthesis tool which components are to be used in the design. However, so far no component instantiation of connection has been made. In order to "place" the components in the design, and to connect them, the *port map* VHDL statement must be used. In Fig. 3.5, line 36, an instance (one copy) of component *C*1 is placed in the circuit. On line 37, an instance of *C*2 is place in the design, and on line 41 an instance of *C*4 is placed in the design. The connections are established according to the arguments in the *port maps* parameters list. On line 36, the first parameter (*A*) tells the synthesis tool to connect the first input signal of component *C*1 (*A* on line 13) to the *A* signal listed in the top-level entity (line 5). The parameter *B* on line 36 results in the connection between the second input signal of *C*1 (*B* on line 14) and the *B* signal listed in the top-level entity (line 5). Parameter *C* on line 36 is the connection from the *C* signal listed in the top-level entity (line 5) to the third input signal of *C*1 (*C* on line 15). Finally, the *F*1 parameter connects *C*1 output to one of *C*4 inputs (see line 41).

An important remark is that in the parameters list of *port map* statements, only entity and architecture signals are allowed. This means that all signals listed in the components declarations cannot be used. For instance, the *A*, *B*, and *C* parameters used on line 36 are the signals listed on line 5 of the entity, and not the *A*, *B* and *C* signals that are listed on lines 13, 14 and 15, respectively. The *F*1 parameter is defined as an internal signal, in the architecture section.

The synthesis tool makes the relation between the informed parameter in the *port map* and the respective component's signal, according to their order in the argument's list and in the component's declaration. For instance, on line 37, *C*2 inputs *A* and *B* are connected to the top-level inputs *B* and *C* (listed on line 5). *F*2 on line 37 is connected to *F*2 on line 41. Therefore, the signals are mapped in the same order as defined in the component declaration.

VHDL has another syntax for signals mapping, which is the "explicit mapping". For example, line 37 can be rewritten as follows:

```
L1: C2 port map (F=>F2, A=>B, B=>C);
```

In this alternative syntax, there is no need to keep the parameters order, as the => operator forces the signals mapping. Thus, *A*, *B* and *F* signals of *C*2 are explicitly connected to *B*, *C* and *F*2, where *B* and *C* are top-level entity signals, and *F*2 is an internal signal.

Another important remark is that each entity/architecture construction needs its own library declaration. Also, it is a good coding practice to keep each component in a separate file. The top-level entity/architecture (see Fig. 3.5) should also be

kept in another file, and the synthesis tool can find all components of a design just looking for the "component" statements in this file.

Figure 3.6 shows the circuit generated from the VHDL code in Fig. 3.5. As stated on line 36, *C*1 has *A*, *B* and *C* as inputs, and *F*1 as its output. On line 37, *B* and *C* are defined as the inputs for *C*2, and *F*2 is its output. Component *C*4 is instantiated on line 41, having *F*1, *F*2 and *F*3 as inputs, and *F* as output. All 4 components are in the box named "top" in Fig. 3.6, and this box represents the VHDL code in Fig. 3.5 (see the entity's name).

3.2 Laboratory Assignment

The laboratory objectives are:

- to practice the basic concepts of hierarchical design in VHDL;
- to build a basic design to be used as a skeleton in the remaining lab sessions in this book;
- to carry on the training and understanding of a complete FPGA design flow.

3.2.1 Laboratory Session

This lab has as its main task the design of the circuit shown in Fig. 3.6, including all the aforementioned components. The whole circuit is built from five VHDL files, each of them comprising an entity and an architecture section. The four components *C*1, *C*2, *C*3 and *C*4, are saved in files c1.vhd, c2.vhd, c3.vhd, and c4.vhd, respectively. Component *C*3 has to be implemented according to the specifications provided next. The top-level component (see Fig. 3.5), performs all the instantiations and connections and it is saved in file *top.vhd*.

The tasks to be completed in this laboratory session, using Altera's Quartus II EDA tool, are as follows:

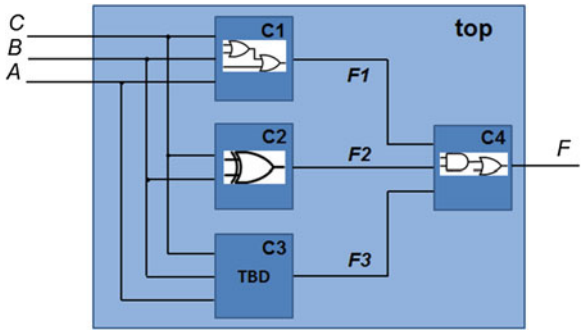


Fig. 3.6 Final circuit generated from VHDL code shown in Fig. 3.5

- Create a new project with all the components shown in Fig. 3.6—Steps 1, 2;
- Create the missing component (*C*3) using the VHDL design entry editor—Step 3;
- Modify the top-level component provided in Fig. 3.5, in order to add *C*3 to the circuit—Step 4;
- Modify the top-level component provided in Fig. 3.5, in order to adapt its inputs and outputs to the signal labels used in the DE2 (prototyping board) interface—Step 5;
- Perform the synthesis—Step 6;
- Perform the simulation and fix errors, if any—Step 7;
- Prototype and test the circuit in the FPGA board—Step 8.

Step 1—Creating a New Project

This step is essentially the same as "Step 1" described in Chap. 1. The major difference is in the project's name. In this new design, instead of "halfadder" use "top" as the project's name (page 1 of 5 in the New Project Wizard—see Fig. 1.13), as this is the name of the entity in Fig. 3.5 (top-level entity).

Step 2—Adding VHDL Files to the Project

Chose *File* → *New* and *VHDL File*, as shown in Fig. 3.7. This will open an empty text editor.

Type in the text editor the VHDL code for *C*1 as shown in Fig. 3.2, and save it: *File* → *Save* (or just Ctrl-S). The default name to save the file is top.vhd, as it is the name provided as the "project name" in Step 1. Before saving it, change the name to c1.vhd. The file will be saved, also in the default location, in the project folder defined in Step 1. Be sure that the check box "Add file to current project" is checked. Again, at this point it is important to be sure that the entity's name is also "*C*1" otherwise, the synthesis tool will not be able to find the entity to be synthesized.

Repeat the same procedure to create c2.vhd, c4.vhd, and top.vhd:

1. *File* → *New* → *VHDL File*;
2. Type de VHDL code for each component and the top-level—use Fig. 3.3 for c2.vhd, Fig. 3.4 for c4.vhd, and Fig. 3.5 for top.vhd;
3. Use *File* → *Save* for saving each file, always changing the name according to what is being saved;
4. Be sure to select the "Add file to current project" checking box;

Alternatively, if you already have c1.vhd, c2.vhd, c3.vhd, c4.vhd and top.vhd, just select *Project* → *Add/Remove Files in Project*, as shown in Fig. 3.8. This option will open a window allowing the import of VHDL files into the new project.

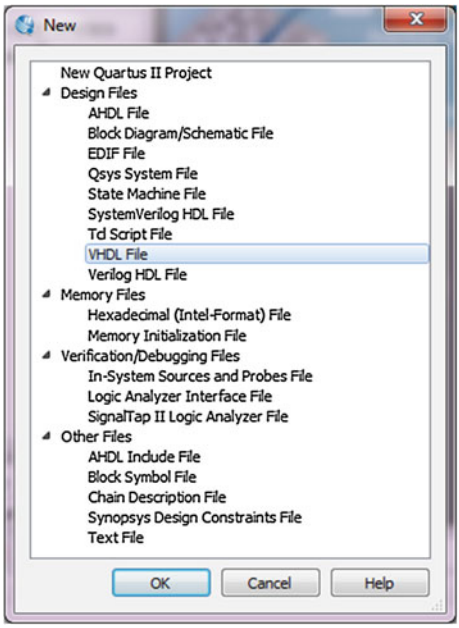


Fig. 3.7 New VHDL file

Step 3—Creating and Adding *C*3 to the Project

Using *C*1, *C*2 or *C*4 as a model, write in VHDL the circuit presented in Fig. 3.9. Follow "Step 2" instructions to create and add this component as c3.vhd to the project.

Step 4—Editing top.vhd to Add *C*3 to the Project

As shown in Fig. 3.10, select *Project Navigator* (1), *Files* (2), and double-click on top.vhd (3). This will open the top-level component in the VHDL text editor. Add *C*3 component interface (entity) to the designated location in Fig. 3.5 (see line 26), and make the necessary modifications in order to have the wright syntax for components definition in VHDL. Use components *C*1, *C*2 and *C*4 of Fig. 3.5 as examples.

Create an instance of *C*3 using the port map statement on line 39 of Fig. 3.5. Use the port maps of components *C*1, *C*2 and *C*4 as examples, but be careful to make the correct connections according to Figs. 3.6 and 3.9. Notice that in Fig. 3.9, *B* and *C* are the inputs for the OR gate, and *A* is the input for the NOT gate. *F*3 is the component output. All these connections should be performed by the port map statement.

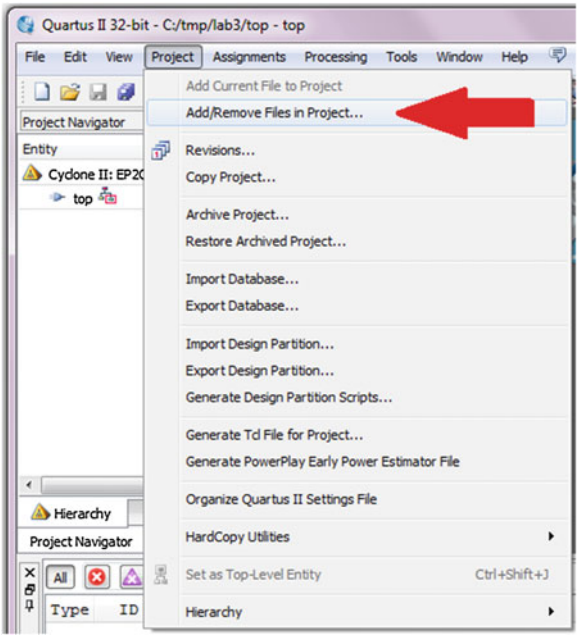


Fig. 3.8 Adding existing files to the project

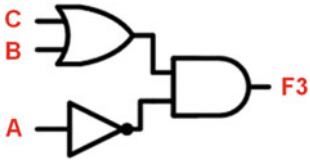


Fig. 3.9 *C*3 schematic diagram

Step 5—Adapting the Design to Work in the DE2 FPGA Board

At this point, the circuit is ready for synthesis (see Chap. 2, Step 3) and simulation (see Chap. 2, Step 4), but it will not work on DE2 FPGA board. As explained in Chap. 2, and introduced in Fig. 2.8, a circuit to be prototyped in DE2 board should employ specific signal names in its entity.

Figure 3.11 shows the available switches and LEDs on DE2 board. In the design shown in Fig. 3.12, *A*, *B* and *C* inputs will be provided by DE2 switches 0, 1 and 2, respectively. The *F* output will be presented in red LED 0.

In order to adapt inputs and outputs of top.vhd component to the DE2 board, replace its entity signals in Fig. 3.5, lines 5 and 6 by:

```
4:    entity top is
5:      port ( SW  : in  std_logic_vector(17 downto 0);
6:             LEDR: out std_logic_vector(17 downto 0)
7:      );
8:    end top;
```

SW is an 18 bits vector, where each bit represents a DE2 switch. In a similar way, LEDR is also an 18 bits vector but representing the 18 red LEDs. In this new entity there is no longer *A*, *B*, *C* and *F* signals. So, all occurrences of *A*, *B* and *C* in Fig. 3.5 should be replaced by SW(0), SW(1) and SW(2), respectively. All occurrences of *F* should be replaced by LEDR(0). The port map statements for *C*1

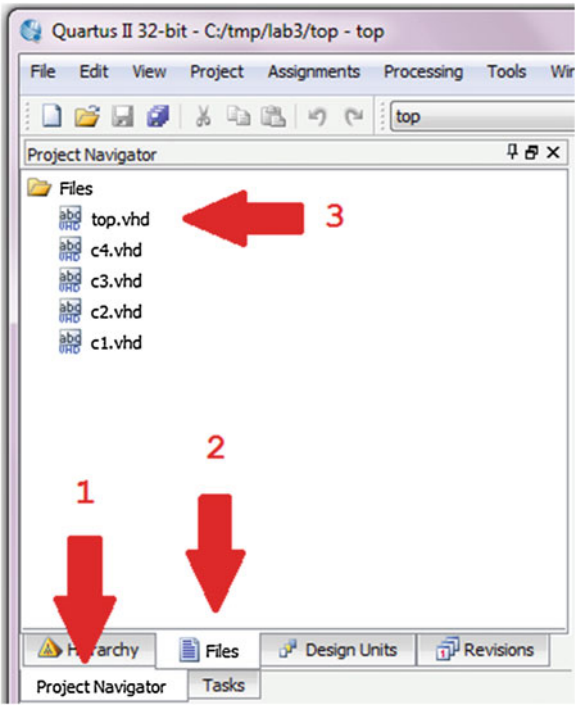


Fig. 3.10 Making changes to top.vhd

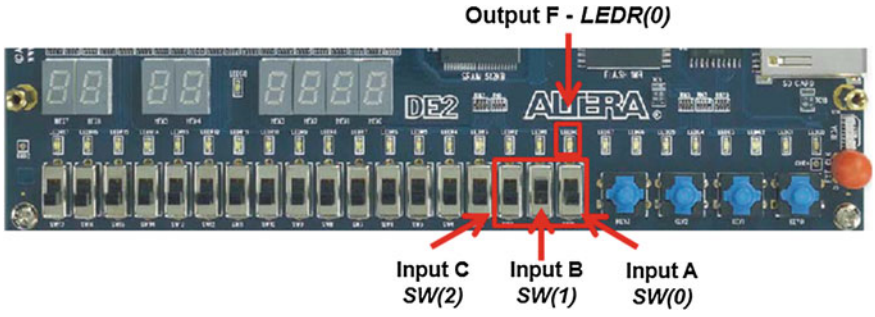


Fig. 3.11 DE2 input and output resources

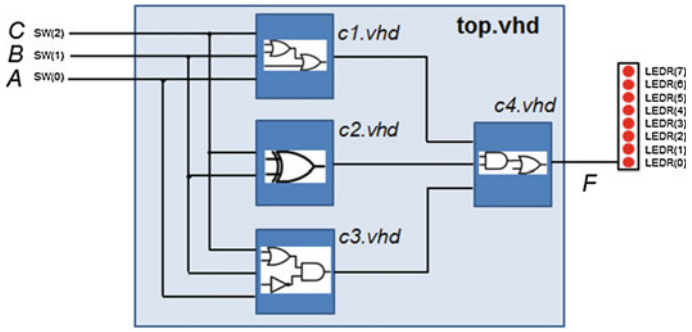


Fig. 3.12 Final circuit with DE2 board connections

and *C*4 for instance, should look like:

```
L0: C1 port map (SW(0), SW(1), SW(2), F1);
L3: C4 port map (F1, F2, F3, LEDR(0));
```

Only the top-level entity signals *A*, *B*, and *C*, and the internal signal *F* should be replaced by SW and LEDR, respectively. The remaining *A*, *B*, *C* and *F* signals belonging to the components should not be changed.

Step 6—Synthesis

To generate the configuration file (bitstream) to be downloaded to the FPGA board, the first step is to import the *DE2_pin_assignments.qsf* file, as discussed in Chap. 2, Step 5, and shown in Fig. 2.7. To perform the synthesis, use the Compile button in Quartus II menu. Fix any errors pointed out by the synthesis tool.

Step 7—Simulation

If the simulation tool has not been configured already, follow the instructions in Chap. 1, "Step 4". To start the simulator in Quartus II, select *Tools* → *Run Simulation Tool* → *RTL Simulation*. Wait for ModelSim to open and perform the following steps:

- Simulate → Start Simulation;
- In the "Start Simulation" window, look for the "Design" tab;
- In the "work" Library, click on the "+" sign, and select the "top" Entity;
- Click OK to start the simulation;
- Look for the Objects window and click on the "+" sign next to the SW label. This will open all 18 bits of vector SW, as shown in Fig. 3.13. Select SW(0), SW(1), and SW(2) in the Objects window, drag and drop them in the Wave

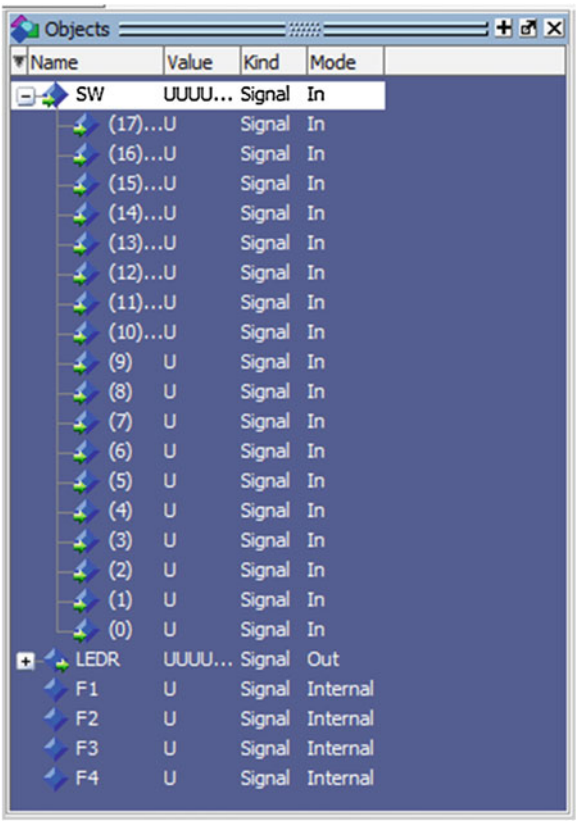


Fig. 3.13 Simulation signals selection

window. Do the same for LEDR(0), *F*1, *F*2, and *F*3. At the end, the Wave window should look like Fig. 3.14.

- Set SW(0), SW(1) and SW(2) to 0 using the "Force" option, as explained in Chap. 1, "Step 4". Set the simulation run length to 100 ps, and press the *Run* button. Next, change SW(0), SW(1) and SW(2) to the remaining seven possible combinations, always running for 100 ps after each input combination.
- The expected simulation results are shown in Fig. 3.15.

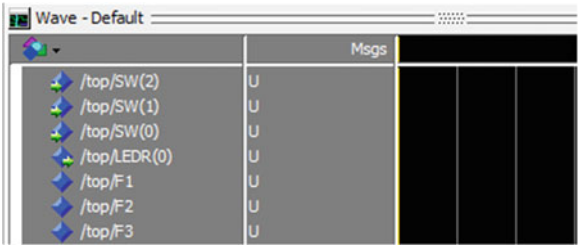


Fig. 3.14 Wave window

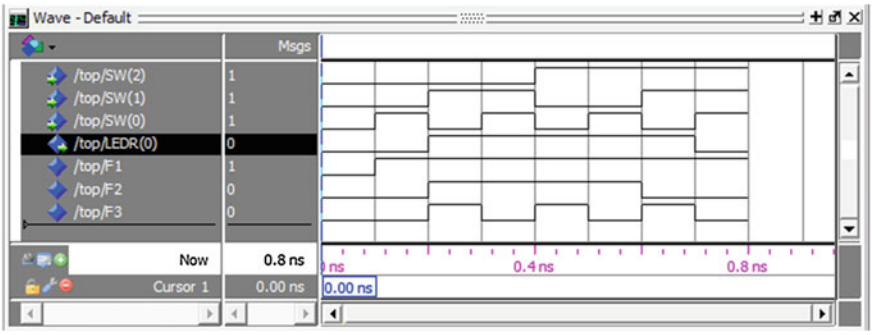


Fig. 3.15 Expected simulation results

Step 8—Prototyping the Circuit in the FPGA Board

Power on and connect the FPGA board to the host computer through the USB port (always using the "blaster" USB connector). Press the red button to switch on the board, and run the programming tool: Menu *Tools* → *Programmer*.

Follow the instructions provided in Chap. 1, Step 5, observing the "hardware setup" instructions in Figs. 1.32 and 1.33.

Test the circuit in the FPGA board, using switches SW(0), SW(1) and SW(2) as inputs, and observing the results in LEDR(0).

Fill in the truth table provided next in Table 3.1. The second column should be filled in with the results obtained from the evaluation of Eq. (3.1). The third column can be obtained straight from the simulation results shown in Fig. 3.15. The fourth column should be filled in with the FPGA execution results.

In case of mismatches in any line of the truth table, the circuit should be fully revised in order to find the error.

Table 3.1 Truth table for the case study

Input SW(2..0) C B A	Output $F = (F1$ *and* $F2)$ or $F3$ *where:* $F1 = A$ *or* B *or* C $F2 = B$ *xor* C $F3 = (B$ *or* $C)$ *and (not* $A)$	Output Obtained in Step 7 simulation LEDR(0)	Output Obtained in Step 8 FPGA LEDR(0)
0 0 0			
0 0 1			
0 1 0			
0 1 1			
1 0 0			
1 0 1			
1 1 0			
1 1 1			

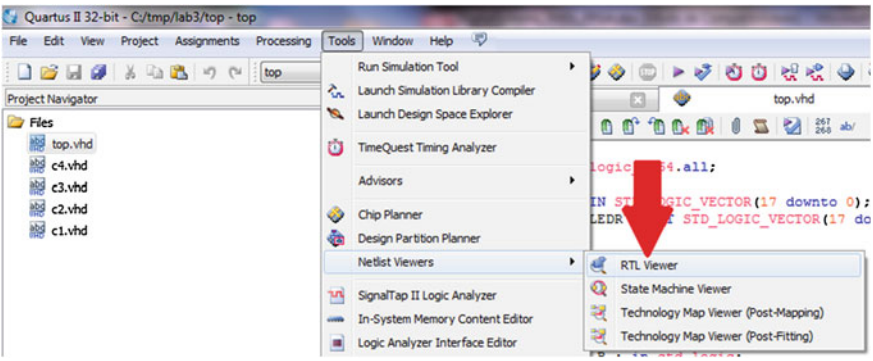


Fig. 3.16 Quartus II RTL viewer tool

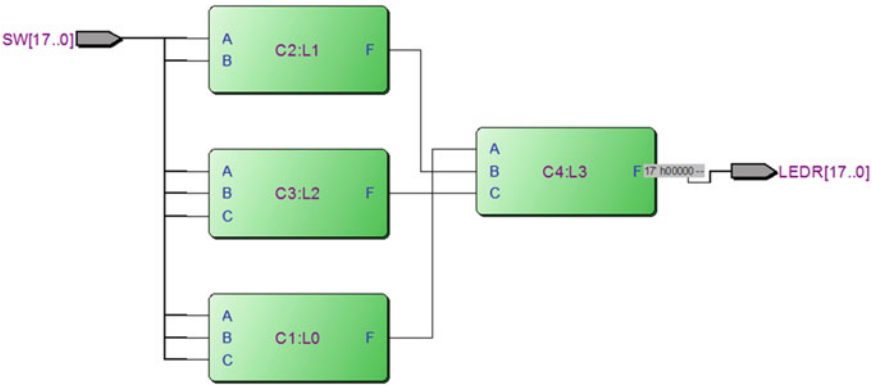


Fig. 3.17 Final circuit in RTL viewer

The Quartus II tool "RTL viewer" can be used to check if a circuit has been generated according to its requirements. Figure 3.16 shows the menu used to start the tool, and Fig. 3.17 shows the circuit's block diagram generated by Quartus II. In this block diagram, the developer can travel through the design hierarchy by double-clicking on the components.

Chapter 4
Multiplexer and Demultiplexer

In this chapter components for building decision maker circuits are introduced. The reader will be guided through the design of multiplexer components in VHDL using only logical gates (i.e., structural level), and also in a higher level of abstraction using *when/else* statements (i.e., behavioral level). As a case study the hierarchical design described in Chap. 3 will be re-used and modified in order to include a multiplexer. At the end of the chapter, the reader should be able:

- to understand the concept of multiplexer and demultiplexer;
- to implement a multiplexer in VHDL using only Boolean functions;
- to implement a multiplexer in VHDL using *when/else* statements;
- to design, simulate and prototype a proposed case study using an FPGA board.

4.1 Theoretical Background

A multiplexer (Mux) component has several inputs, a control signal, and one output. The output receives the value present in one of its inputs, according to the control signal. The block diagram of a general mux in Fig. 4.1a has n input signals (I) and m control signals (S). A 2^n inputs mux, needs n control signals.

The mux component in Fig. 4.1b has four 1 bit inputs A, B, C and D, set to the values 0, 1, 1, and 0, respectively. The selection is set to allow the third input to be connected to the output and, consequently, the F signal presents the value 1. Every time C has its input value changed, F will also be changed. To connect another input to the F output, it is necessary to change the selection signal.

A mux circuit, also known as "data selector", only allows one input to be available at its output at a time, according to the control signals ("Selection" in Fig. 4.1). As examples of applications, this component can be used for data selection, for data routing, in parallel to serial conversions, and also to implement truth tables.

E. A. Bezerra and D. V. Lettnin, *Synthesizable VHDL Design for FPGAs*,
DOI: 10.1007/978-3-319-02547-6_4,

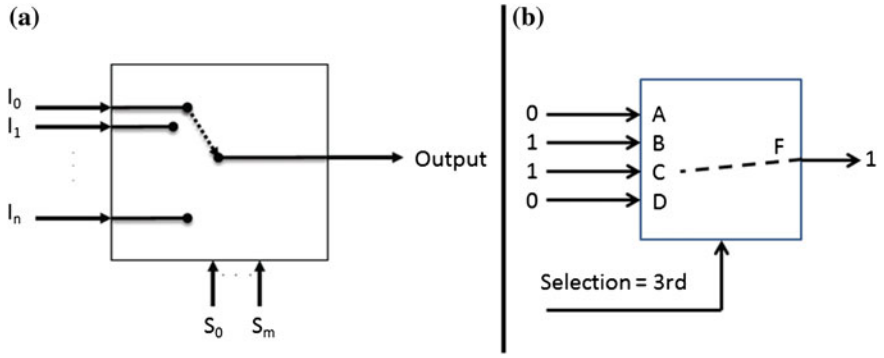


Fig. 4.1 Mux component block diagram. **a** n:1 Mux. **b** 4:1 Mux

A demultiplexer (Demux) circuit, also known as "data distributor", has the opposite function of the multiplexer, that is, it allows an input to be available at only one output at a time. This decision is also defined by control signals. The block diagram of a general demux shown in Fig. 4.2a has n output signals (O) and m control signals (S). A 2^n outputs demux needs n control signals. In Fig. 4.2b the value 0 is provided to input A, and the control signal "Selection" is set to the second output. As a result, the output F2 is connected to the input A, providing also the 0 value. If the selection is changed, for instance, to the first output, so F1 will also provide the 0 value. When the A input receives a new value, the corresponding output will also be updated.

The 2:1 multiplexer (or 2:1 selector) has the function to select one of its two inputs, making the selected entry appear on its output. In Fig. 4.3, if selector s is equal to 0, then m output receives the x input. If $s = 1$, then the m output receives the y input.

This multiplexer can be implemented in VHDL using only Boolean functions (i.e., structural VHDL) as well as using *when/else* statements (i.e., behavioral VHDL). In

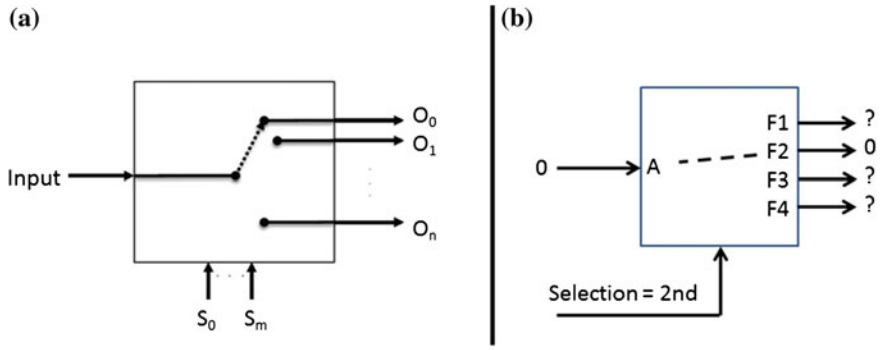


Fig. 4.2 Demux component block diagram. **a** 1:n Demux. **b** 1:4 Demux

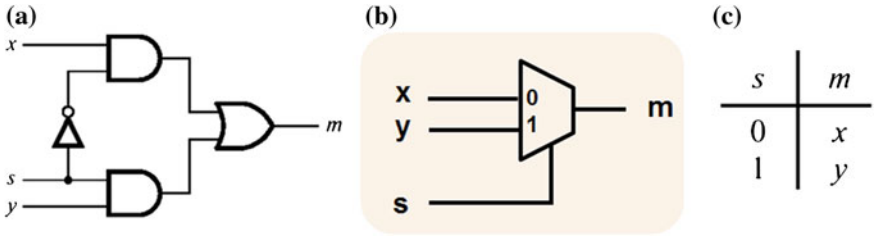


Fig. 4.3 2:1 Multiplexer. **a** Schematic diagram. **b** Block diagram. **c** Truth table

Fig. 4.4, in the structural VHDL version only logic gates are used to implement the functionality described in the truth table shown in Fig. 4.3c. In the algorithmic version, the behavior of the truth table is modeled using the *when/else* VHDL statements.

The 4:1 multiplexer selects one of its four inputs, making the selected entry appear on its output. In a four inputs mux, its selector must be 2 bits long. In Fig. 4.5, if selector *s* is equal to "00", then the *m* output receives the *w* input. If *s* = "01", then the *m* output receives the *x* input, and so on. In Fig. 4.5a, the 2 bits of the *s* selector are represented explicitly by S0 and S1. In Fig. 4.5b, c, the *s* symbol is used to represent the 2 bits needed to select one of the four inputs.

4.2 Laboratory Assignment

The laboratory objectives are:

- to implement a multiplexer in VHDL;
- to learn and to practice the use of *when/else* VHDL statements;
- to add a multiplexer to the circuit of the case study design.

4.2.1 Laboratory Session

In this laboratory session the reader will have the opportunity to design a multiplexer in VHDL in two levels of abstraction: structural; and behavioral. At the structural level, the multiplexer is written in VHDL using only logic gates

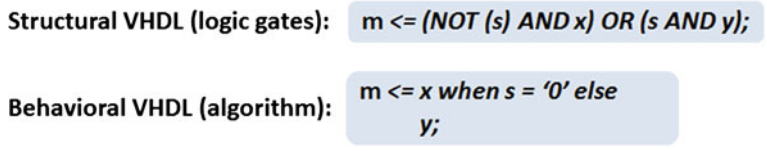


Fig. 4.4 2:1 Multiplexer in VHDL

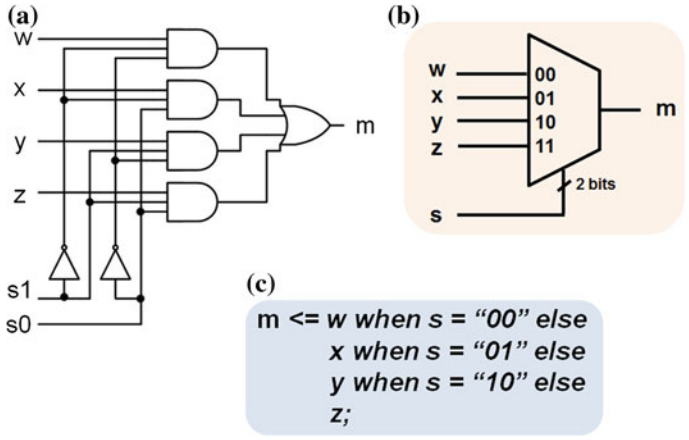


Fig. 4.5 4:1 Multiplexer. **a** Schematic diagram. **b** Block diagram. **c** Truth table

(see Chap. 1). At the behavioral level, the *when/else* statements in VHDL are used in a more algorithmic approach to the design of digital systems. Figure 4.6 shows the two versions to be implemented. In version I, the two bits of the selector input are explicated presented, as the figure shows the internal gate connections of the multiplexer. In version II, the multiplexer is described using the *when/else* VHDL statements, in a higher level of abstraction, with no information regarding the circuit's internal gate connections.

The case study discussed in previous chapters is reused, but with the C4 component shown in Fig. 3.6 replaced by a 4:1 multiplexer component. In the circuit shown in Figs. 4.7 and 4.8, the multiplexer is used to direct one of its inputs F1, F2 or F3 (w, x, y, z), to the F output (m). Thus, LEDR(0) will light on when the

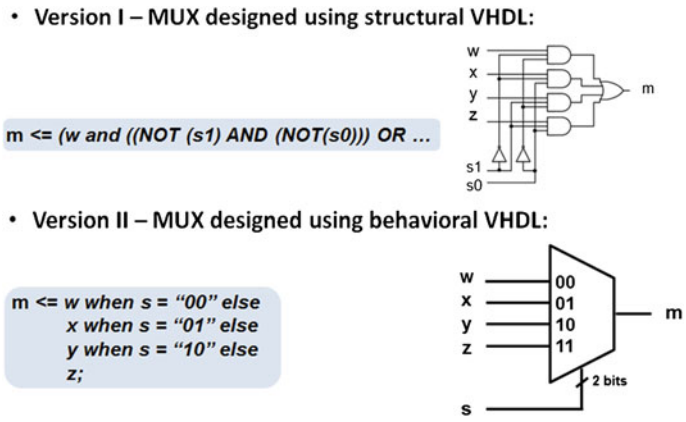


Fig. 4.6 Multiplexer described in two abstraction levels

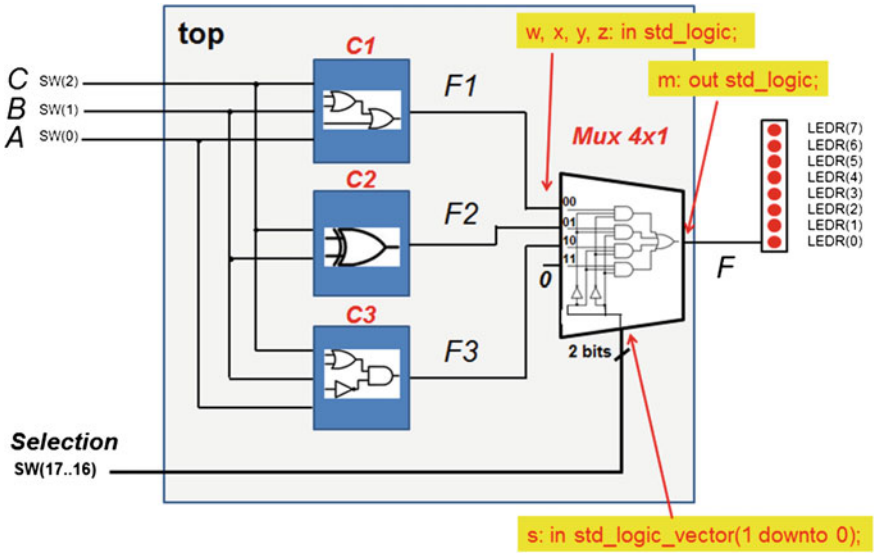


Fig. 4.7 Using a multiplexer instead of C4, version I—structural VHDL

2 bits selection input is set to a position whose component C1, C2 or C3 provides an '1' logic on its output.

For instance, when the selection is set to "00" (SW_{17} and SW_{16} are switched off), the F output receives F1. In this case, if one of the three inputs C1, C2 or C3 is

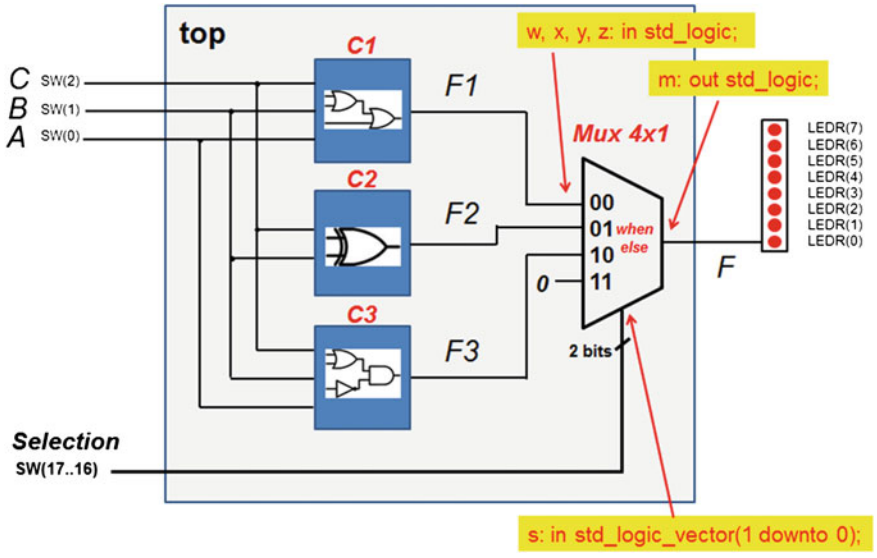


Fig. 4.8 Using a multiplexer instead of C4, version II—behavioral VHDL

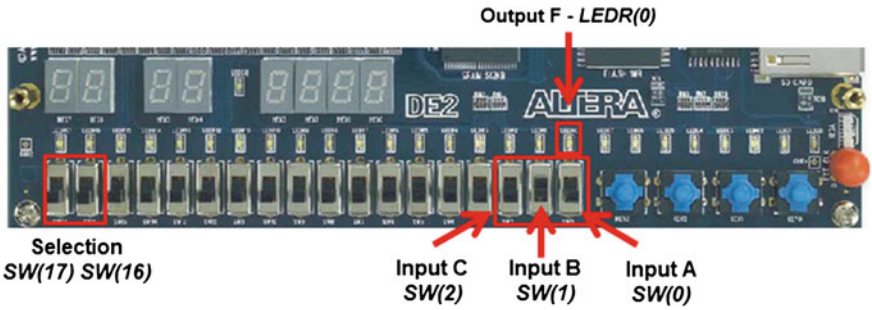


Fig. 4.9 User interface—MUX input selection

switched on, so C1 will evaluate to '1', and LEDR(0) will light on. In Fig. 4.9, the selection bits are represented by switches SW_{17} and SW_{16}, input C is SW_2, input B is SW_1, input A is SW_0, and output F is LEDR(0). Table 4.1 shows all possible combinations for the multiplexer selection bits, and the corresponding output in LEDR(0).

In this laboratory assignment, two versions for the multiplexer based design are to be implemented. Version I, i.e., multiplexer implemented in structural VHDL is shown in Fig. 4.7. In version II, shown in Fig. 4.8, the multiplexer is to be implemented in behavioral VHDL. Therefore, in this laboratory assignment, two different VHDL designs should be created. The tasks to be completed in this laboratory session, using Altera's Quartus II EDA tool are as follows:

- Create a project with all the components shown in Fig. 4.8 (last chapter components can be reused);
- Create a MUX component using the VHDL design entry editor—two MUX components should be developed, one of them using a gate level VHDL description, and another one using a behavioral (algorithmic) VHDL description;
- Modify the top-level component provided in Fig. 3.5, in order to replace C4 by the new MUX component;
- Perform the synthesis;
- Perform the simulation and fix errors, if any;
- Prototype and test the circuit in the FPGA board.

Table 4.1 Mux input selection and corresponding outputs

Input $SW_{17..16}$	Output $LEDR_0$
0 0	F1
0 1	F2
1 0	F3
1 1	0 (LED off)

4.2.2 Version I: Multiplexer in Structural VHDL

Step 1—Creating a New Project

This step is essentially the same as "Step 1" described in previous chapters. Just remember to use "top" as the project's name (see Fig. 1.13), as this is the name of the top-level entity. Remember also to create the folder for this new project in a place such as c:\LabSessions\mux4x1, where "mux4x1" is the name of the new folder.

Step 2—Adding VHDL Files to the Project and Creating the Mux Component

Using File Explorer (formerly known as Windows Explorer), or any other file manager program, locate the folder where the five VHDL files developed in last chapter are located (c1.vhd, c2.vhd, c3.vhd, c4.vhd, and top.vhd), and copy all files to the new folder c:\LabSessions\mux4x1. This procedure is shown in Fig. 4.10.

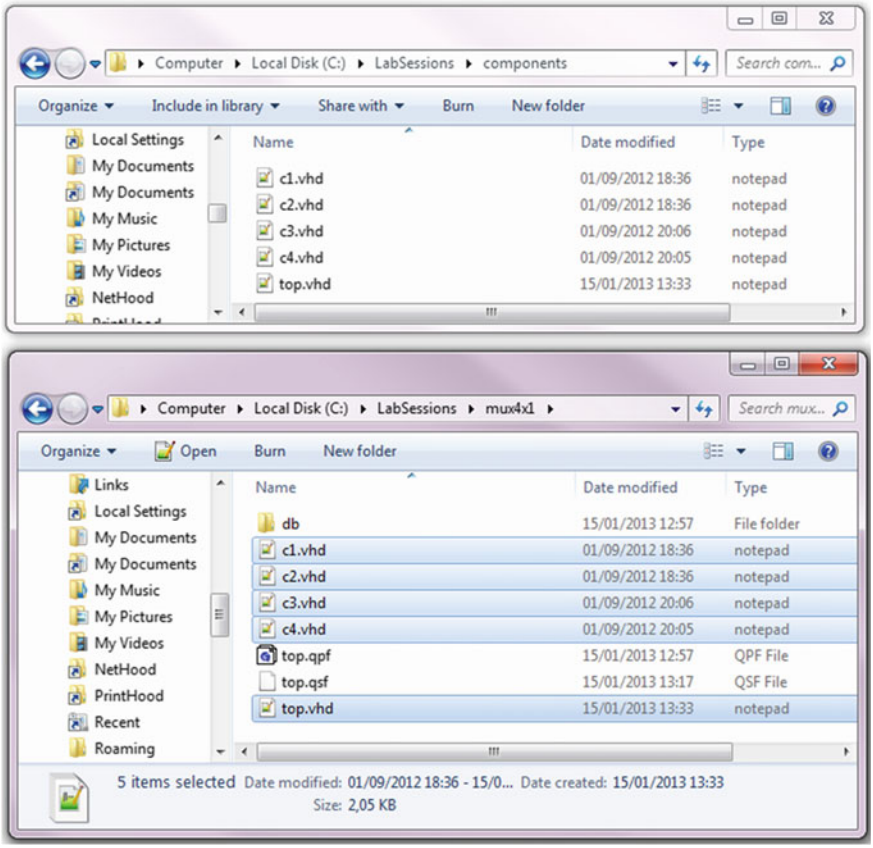


Fig. 4.10 Copying files using File Explorer

In Quartus II, choose *Project* → *Add/Remove Files in Project*, as shown in Fig. 3.8. A window will open allowing c1.vhd, c2.vhd, c3.vhd, c4.vhd and top.vhd, to be imported into the project. In this window, after adding the files press the *Apply* button.

In the Project Navigator window, select the Files tab, and double-click on c4.vhd in order to open it in the text editor. Make the following changes in c4.vhd, in order to turn it into a multiplexer component:

- Change the entity's name from "C4" to "Mux" (three changes are needed);
- Change the architecture's name to "mux_stru";
- Add the new entity's signals w, x, y, z, s, and m, as shown in Fig. 4.7;
- Replace the expression F <= (A and B) or C), by the multiplexer described in Fig. 4.6, version I;

Chose *File* → *Save as*, and rename c4.vhd to mux4x1.vhd. The file will be saved, also in the default location, in the project folder defined in Step 1. Be sure that the check box "Add file to current project" is checked. At this point it is important to be sure that the entity's name is "mux" otherwise the synthesis tool will not be able to find the entity to be synthesized.

A mux4x1.vhd component should appear in the Files tab of the Project Navigator window. As shown in Fig. 4.11, the old c4.vhd component can be removed from the project by right-clicking on it.

Step 3—Editing top.vhd to Add Mux to the Circuit

As shown in Fig. 3.10, select *Project Navigator* (1), *Files* (2), and double-click on top.vhd (3). This will open the top-level component in the VHDL text editor. In Fig. 3.5, remove C4 component (lines 28–33), and *copy and paste* the entity of the new Mux. Make the necessary modifications in order to have the wright syntax for component definition in VHDL (e.g., replace "entity" by "component", …).

Next, remove line 41 (C4), and create an instance of Mux using the port map statement. Use the port map declarations for components *C1*, *C2* and *C3* as examples, but be careful to make the correct connections according to Fig. 4.7. Notice that in Fig. 4.7, F1, F2 and F3 are connected to *w*, *x* and *y*, respectively. The *z* input is connected to a '0' constant. The *s* input (mux selection) is connected straight to DE2 signals SW_{17} and SW_{16}. The *m* output is connected to DE2 LEDR(0) signal. All these connections should be performed by the port map statement.

Step 4—Synthesis

To generate the configuration file (bitstream) to be downloaded to the FPGA board, the first step is to import the *DE2_pin_assignments.qsf* file, as discussed in Chap. 2, Step 5, and shown in Fig. 2.7. To perform the synthesis, use the Compile button in Quartus II menu. Fix any errors pointed out by the synthesis tool.

Use the Quartus II tool "RTL viewer" to check if the generated circuit is as expected. Figure 4.12 shows the circuit's block diagram generated by Quartus II.

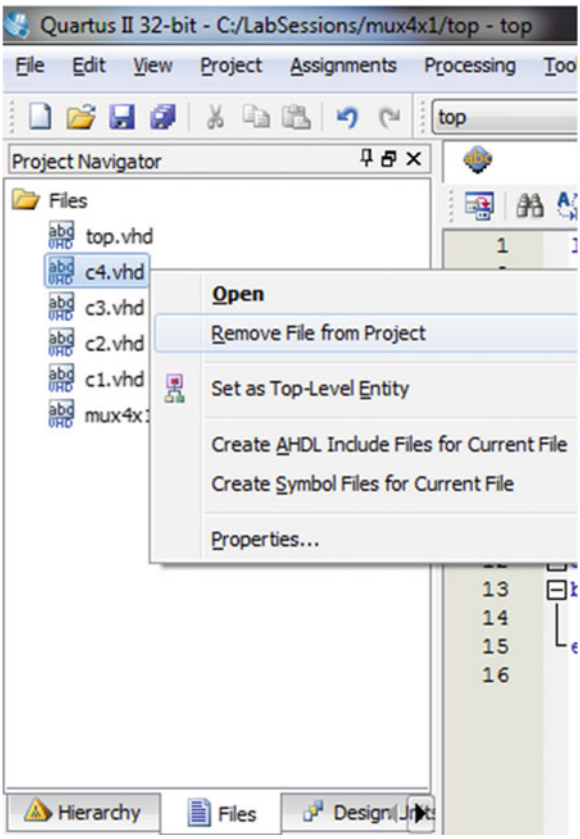


Fig. 4.11 Removing c4.vhd from the project

Step 5—Simulation

To start the simulator in Quartus II, select *Tools* → *Run Simulation Tool* → *RTL Simulation*. Wait for ModelSim to open and perform the following steps:

- Simulate → Start Simulation;
- In the "Start Simulation" window, look for the "Design" tab;
- In the "work" Library, click on the "+" sign, and select the "top" Entity;
- Click OK to start the simulation;
- Look for the Objects window and click on the "+" sign next to the SW label. This will open all 18 bits of vector SW, as shown in Fig. 3.13. Select SW(0), SW(1), SW(2), SW(16) and SW(17) in the Objects window, drag and drop them in the Wave window. Do the same for LEDR(0), F1, F2, F3 and F4.
- Set SW(0), SW(1), SW(2), SW(16) and SW(17) to 0 using the "Force" option, as explained in Chap. 1, "Step 4". Set the simulation run length to 100 ps, and press the *Run* button. Next, change SW(0), SW(1), SW(2), SW(16) and SW(17) to the remaining 31 possible combinations, always running for 100 ps after each input combination.

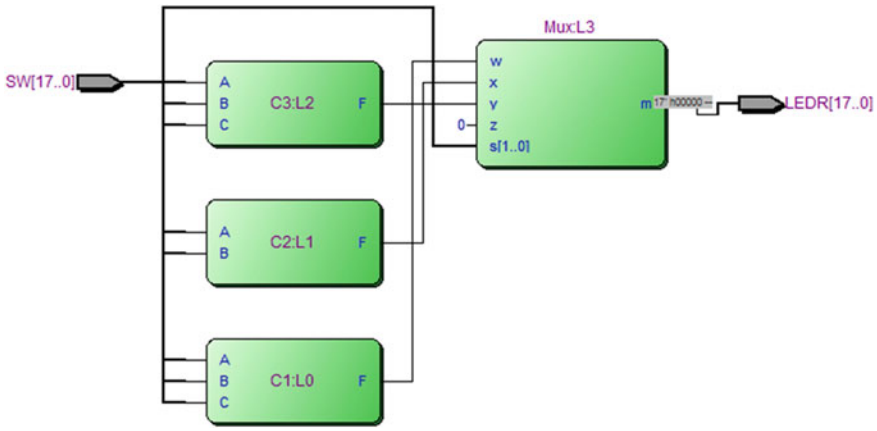


Fig. 4.12 RTL viewer block diagram

Figure 4.13 shows the simulation results when the selection bits SW_{17} and SW_{16} are set to "00", and the inputs A, B and C (SW_0, SW_1, SW_2) receive all possible 8 values ("000", "001", …, "111"). Every time SW_{17} and SW_{16} are set to "11", the z input is selected and, as shown in Fig. 4.7, for this circuit this mux input is fixed in zero. As shown in the simulation results, at 0.8 ns SW_{17} and SW_{16} are set to "11", and the $LEDR_0$ output results in "0".

Step 6—Prototyping the Circuit in the FPGA Board

Power on and connect the FPGA board to the host computer through the USB port (always using the "blaster" USB connector). Press the red button to switch on the board, and run the programming tool: Menu *Tools* → *Programmer*.

Follow the instructions provided in Chap. 1, Step 5, observing the "hardware setup" instructions in Figs. 1.32 and 1.33.

Test the circuit in the FPGA board, using switches SW_0, SW_1, SW_2 as inputs, switches SW_{16} and SW_{17} as selection bits, and observing the results in $LEDR_0$.

Fill in the truth table provided next in Table 4.2. The second column is filled in with the selection bits ("00", "01", "10", and "11"). The third column should be filled in with the results obtained from the manual evaluation of equations F1, F2

Fig. 4.13 Expected simulation results

and F3. In the third column, when the selection bits are "11", the output is always '0' disregarding the input values. The fourth column can be obtained straight from the simulation results, and it is partially filled in according to the waveforms in Fig. 4.13. The fifth column should be filled in with the FPGA execution results. In case of mismatches in any line of the truth table, the circuit should be fully revised in order to find the error.

4.2.3 Version II: Multiplexer in Behavioral VHDL

Follow, basically, the same steps as before in order to implement the behavioral VHDL version of the multiplexer based circuit. There are just a couple of remarks to be considered:

Table 4.2 Truth table for the MUX based circuit

Inputs		Outputs		
$SW_{2..0}$ C B A	$SW_{17..16}$ Selection	*F1* = *A* or *B* or *C* *F2* = *B* xor *C* *F3* = (*B* or *C*) and (not *A*)	Simulation Step 5 $LEDR_0$	FPGA Step 6 $LEDR_0$
0 0 0	00	F1 =	0	
0 0 1	00	F1 =	1	
0 1 0	00	F1 =	1	
0 1 1	00	F1 =	1	
1 0 0	00	F1 =	1	
1 0 1	00	F1 =	1	
1 1 0	00	F1 =	1	
1 1 1	00	F1 =	1	
0 0 0	01	F2 =		
0 0 1	01	F2 =		
0 1 0	01	F2 =		
0 1 1	01	F2 =		
1 0 0	01	F2 =		
1 0 1	01	F2 =		
1 1 0	01	F2 =		
1 1 1	01	F2 =		
0 0 0	10	F3 =		
0 0 1	10	F3 =		
0 1 0	10	F3 =		
0 1 1	10	F3 =		
1 0 0	10	F3 =		
1 0 1	10	F3 =		
1 1 0	10	F3 =		
1 1 1	10	F3 =		
X X X	11	0	0	

Step 1

A new folder may be created in order to hold the behavioral version of the project. A suggestion of a folder to be created is c:\LabSessions\mux4x1_beh.

Step 2

Copy the version I VHDL files (.vhd) to the new folder, and add them to the project as explained before.

Edit the mux4x1.vhd file, and make the necessary changes in order to have a behavioral implementation of the multiplexer. The VHDL code based on *when/else* statements is listed in Figs. 4.5c and 4.6.

Steps 3, 4, 5 and 6

There are no need for changes in the top.vhd file, as the new component has exactly the same interface (entity) and functionality as the structural version (version I).

Follow the same instructions for synthesis, simulation and FPGA implementation, as defined for version I.

Chapter 5
Code Converters

This chapter introduces a category of circuits known as "code converters". This type of circuit includes the encoders and the decoders. Encoders are circuits used in the conversion of information in one format to another. Decoders are complementary circuits, used to undo a conversion performed by an encoder. The concepts of code converter circuits are investigated in this chapter through their implementation in VHDL.

In previous chapters, a single LED has been employed in the circuits as their outputs. An LED was a feasible output interface, as so far the case study circuits in this book have been conceived in order to perform single bit functions. In this chapter a more real world case study is developed. It is a basic calculator that performs 8 bits operations, and presents the results in 7-segment displays. A decoder circuit is used to convert the binary result provided by the calculator, into a hexadecimal value to be shown in the 7-segment displays. In order to perform 8 bits operations, the single bit signals are arranged in VHDL as an array of signals, also known as a "bus". For a better understanding of the proposed case study, in this chapter there is also a discussion on "array of signals in VHDL" and "7-segment displays decoders". At the end of the chapter, the reader should be able:

- to understand the concepts of encoder and decoder;
- to implement a decoder in VHDL;
- to design, to simulate and to prototype the proposed case study using an FPGA board.

5.1 Arrays of Signals

As shown in Fig. 4.8, A, B, C and F are all 1 bit signals. In Fig. 5.1 the same signals are represented as arrays of 4 bits signals. This means that A, B, C and F can now hold values in the range from 0 to 15 in decimal or, in binary, from 0000 to 1111.

E. A. Bezerra and D. V. Lettnin, *Synthesizable VHDL Design for FPGAs*,
DOI: 10.1007/978-3-319-02547-6_5,

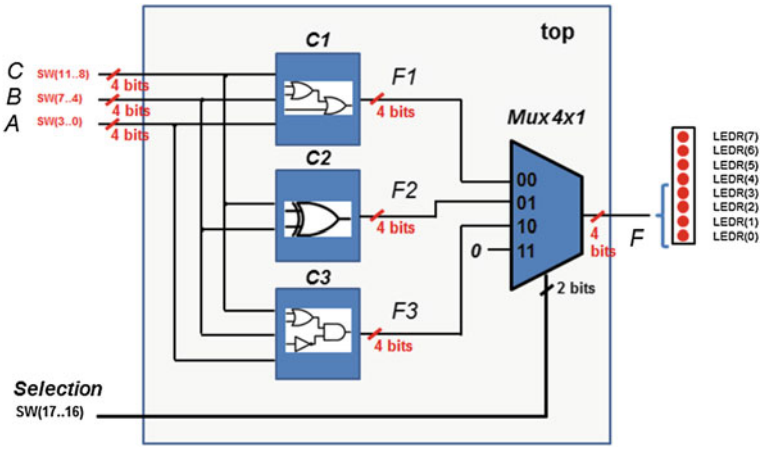


Fig. 5.1 Block diagram of a 4 bits circuit

Considering in Fig. 4.8 and in Fig. 5.1 that:

- C1 performs the function F1 = A or B or C
- C2 performs the function F2 = B xor C
- In Fig. 4.8 (1 bit operands), assume A = 0, B = 1, C = 0
- In Fig. 5.1 (4 bits operands), assume A = 0101, B = 1000, C = 0001.

Figure 5.2 shows the results obtained considering 1 bit and 4 bits wide operands. For 1 bit operands, F1 and F2 result in 1. For the 4 bits version, F1 results in 1101 (13 in decimal) and F2 results in 1001 (9 in decimal).

In VHDL the array of signals used in Fig. 5.1 can be defined through the *std_logic_vector* declaration:

- *std_logic* is used to declare a single signal, a single bit, a wire.
- *std_logic_vector* is used to declare an array of signals, a set of bits, a bus.

Considering a multiple bit implementation (array of signals), the components shown in Fig. 5.1 have been rewritten, and they are listed next. It is important to notice in Figs. 5.3, 5.4, 5.5 and 5.6, that only the entity has changed. The architecture, where the functions are performed, has not been changed.

In Fig. 5.7, the A, B and C inputs are now 4 bits signals represented, respectively, by $SW_{3..0}$, $SW_{7..4}$, and $SW_{11..8}$. The output is now also a 4 bits signals represented by $LEDR_{3..0}$.

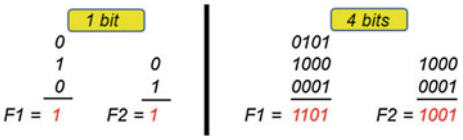


Fig. 5.2 C1 and C2 components using 1 bit and 4 bits operators

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity C1 is
port (A: in std_logic_vector(3 downto 0);
      B: in std_logic_vector(3 downto 0);
      C: in std_logic_vector(3 downto 0);
      F: out std_logic_vector(3 downto 0)
     );
end C1;

architecture c1_str of C1 is
begin
  F <= A or B or C;
end c1_str;
```

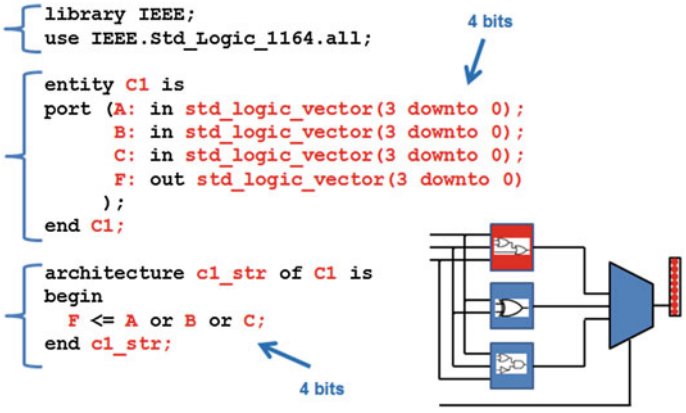


Fig. 5.3 VHDL implementation for component C1 (4 bits)

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity C2 is
port (A: in std_logic_vector(3 downto 0);
      B: in std_logic_vector(3 downto 0);
      F: out std_logic_vector(3 downto 0)
     );
end C2;

architecture c2_str of C2 is
begin
  F <= A xor B;
end c2_str;
```

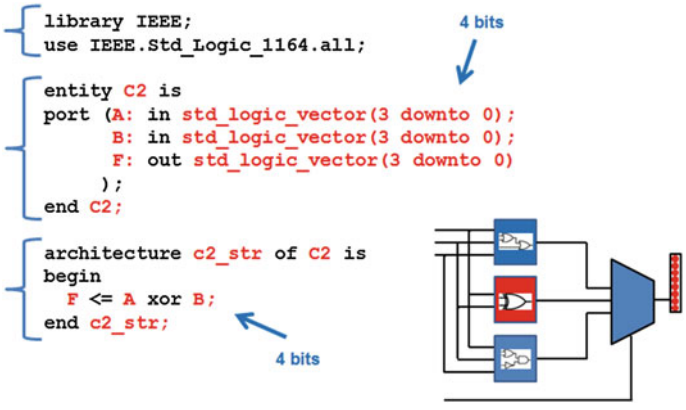


Fig. 5.4 VHDL implementation for component C2 (4 bits)

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity C3 is
port (A: in std_logic_vector(3 downto 0);
      B: in std_logic_vector(3 downto 0);
      C: in std_logic_vector(3 downto 0);
      F: out std_logic_vector(3 downto 0)
     );
end C3;

architecture c3_str of C3 is
begin
  -- proposed lab exercise
end c3_str;
```

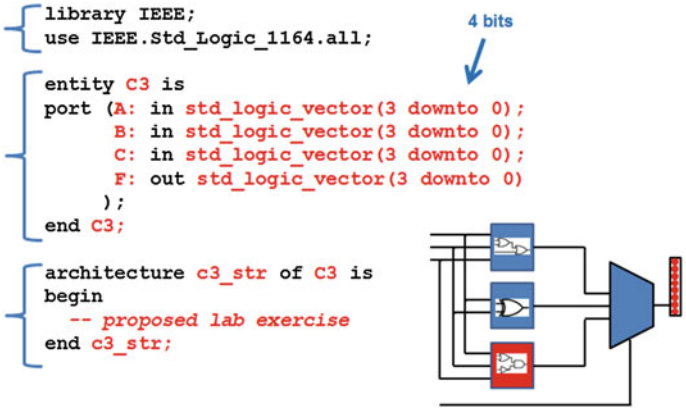


Fig. 5.5 VHDL implementation for component C3 (4 bits)

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity Mux is
port (w: in std_logic_vector(3 downto 0);
      x: in std_logic_vector(3 downto 0);
      y: in std_logic_vector(3 downto 0);
      z: in std_logic_vector(3 downto 0);
      s: in std_logic_vector(1 downto 0);
      m: out std_logic_vector(3 downto 0)
     );
end Mux;

architecture mux_bhv of Mux is
begin
   -- see previous lab.
end mux_bhv;
```

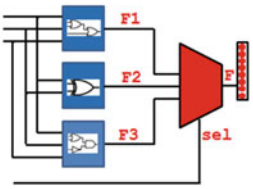
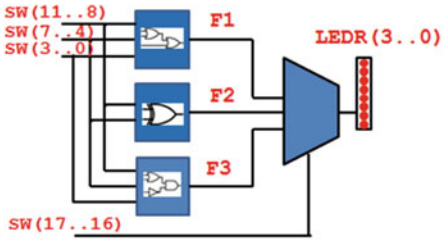


Fig. 5.6 VHDL implementation for component Mux (4 bits inputs/outputs)

```
library ieee;
use ieee.std_logic_1164.all;
entity top is
 port ( SW : IN STD_LOGIC_VECTOR(17 downto 0);
        LEDR : OUT STD_LOGIC_VECTOR(17 downto 0)
 );
end top;
architecture top_stru of top is
 signal F1, F2, F3: std_logic_vector(3 downto 0);
 component C1
   port (A : in std_logic_vector(3 downto 0);
         B : in std_logic_vector(3 downto 0);
         C : in std_logic_vector(3 downto 0);
         F : out std_logic_vector(3 downto 0));
 end component;
 component C2
   port (A : in std_logic_vector(3 downto 0);
         B : in std_logic_vector(3 downto 0);
         F : out std_logic_vector(3 downto 0));
 end component;
 component C3
   port (A : in std_logic_vector(3 downto 0);
         B : in std_logic_vector(3 downto 0);
         C : in std_logic_vector(3 downto 0);
         F : out std_logic_vector(3 downto 0)
   );
 end component;
 -- Mux component declaration goes here
```




```
begin

 L0: C1 port map (SW(3 downto 0),
       SW(7 downto 4), SW(11 downto 8), F1);

 L1: C2 port map (SW(3 downto 0),
       SW(7 downto 4), SW(11 downto 8), F2);

 L2: C3 port map (SW(3 downto 0),
       SW(7 downto 4), SW(11 downto 8), F3);

 L3: Mux port map (F1, F2, F3,
       SW(17 downto 16), LEDR(3 downto 0));

end top_stru;   -- END architecture
```

Fig. 5.7 VHDL implementation for the top circuit of Fig. 5.1

5.2 Seven Segment Displays

In digital circuits, the 7-segment display is a widely used component. This component has seven LEDs that can be turned on and off independently. As shown in Fig. 5.8, there are two types of 7-segment displays:

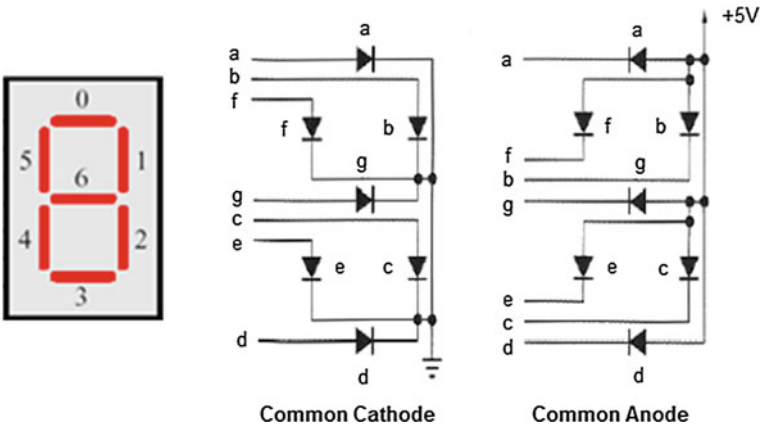


Fig. 5.8 Seven segment display

- **Common cathode**—the LEDs cathode terminals are connected together to the ground and each anode terminal is connected individually to a Vcc source.
- **Common anode**—the LEDs anode terminals are connected together to Vcc, and each cathode terminal is connected individually to ground.

In Fig. 5.1, the 4 bits result is shown in four LEDs (4 bits). A 7-segment display could have been used to show the result in another format as, for instance, in decimal or hexadecimal. In this case, a decoder circuit would be necessary in order to convert the binary output into the appropriate format and coding expected by the 7-segment display.

5.3 Encoders and Decoders

As stated before, encoders are used to convert a piece of information from one format to another, and decoders are used to undo the conversion. Figure 5.9 shows an example of encoder, used to convert the 4-bits input value I to a 2-bits coded

(a)

I_3	I_2	I_1	I_0	O_1	O_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

(b)

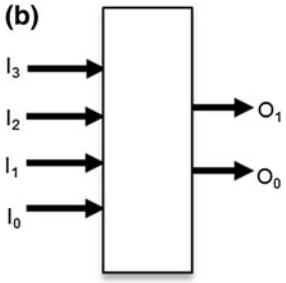


Fig. 5.9 Encoder. **a** Truth table. **b** Symbol

(a)

I_1	I_0	O_3	O_2	O_1	O_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

(b)

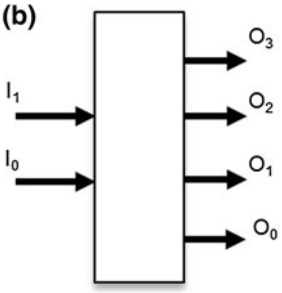


Fig. 5.10 Decoder. **a** Truth table. **b** Symbol

output O. In this example, the input shown in Fig. 5.9a presents just 1 bit set at a time, which is also known as the "one-hot" code. When comparing to the decoder, an encoder component can be used to reduce the amount of bits required to represent any piece of information.

The decoder shown in Fig. 5.10 was designed to convert back the code provided by the encoder shown in Fig. 5.9, that is, it receives a 2-bits code on its input, and provides a one-hot output. This decoder activates only one O_i output at a time, according to the combination of the I input values. In this sense, there is a direct relationship between the number of outputs (N_o) and the number of inputs (N_i), i.e., $N_o = 2 \times N_i$.

5.4 Designing a Seven Segment Decoder

As shown in Fig. 5.8, algorisms can be exhibited in a 7-segment display by turned on and off the different segments. For instance, in Fig. 5.8 when providing the value "0000000" in binary to the seven inputs (a, b,…, g), all LEDs are lighted on, and the '8' value will show on the display. In this case, in order to display the number '8', the code "1000" (eight in binary) should be decoded into "0000000". A 7-segment decoder is a very useful component, as it can be used in digital instruments that have a numerical output, for instance, multimeters, frequency-meters, etc.

The DE2 board has seven 7-segment displays, all of them common anode (LEDs are turned on with a logic zero in the inputs). To show a binary value in one of DE2's displays, a conversion from binary code to 7-segment code should be performed.

As an example of a decoder design, consider a circuit whose functionality is to display the letters 'U', 'F', 'S', and 'C' in a 7-segment display. The truth table in Fig. 5.11a is conceived considering 5 codes: "000" for letter 'U'; "001" for letter 'F'; "010" for letter 'S'; "011" for letter 'C'; and "111" to clear the display (all LEDs are off). The remaining 3 possible codes to represent in 3 bits ("100",

(a)

C2	C1	C0	7 bits *6543210*	Display
0	0	0	**1000001**	U
0	0	1	**0001110**	F
0	1	0	**0010010**	S
0	1	1	**1000110**	C
1	1	1	**1111111**	

(b)

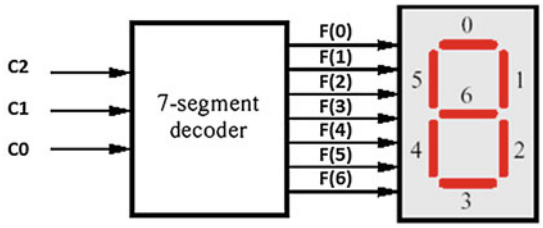


Fig. 5.11 A 7-segment decoder. **a** Truth table, **b** block diagram

"101", and "110") are not used in this circuit. The 7-segment decoder has to be designed in order to convert the 3-bit input codes into 7-bit output codes shown in the 4th column of Fig. 5.11a, which are used to turn on and off the LEDs of the display. For instance, to show the letter 'U', LEDs 0 and 6 (a and g in Fig. 5.8) should be turned off. So, in the 4th column of Fig. 5.11a, there is a '1' on this positions (common anode).

Considering the truth table in Fig. 5.11a, a component to implement this 7-segment decoder could be designed using different methods as, for instance:

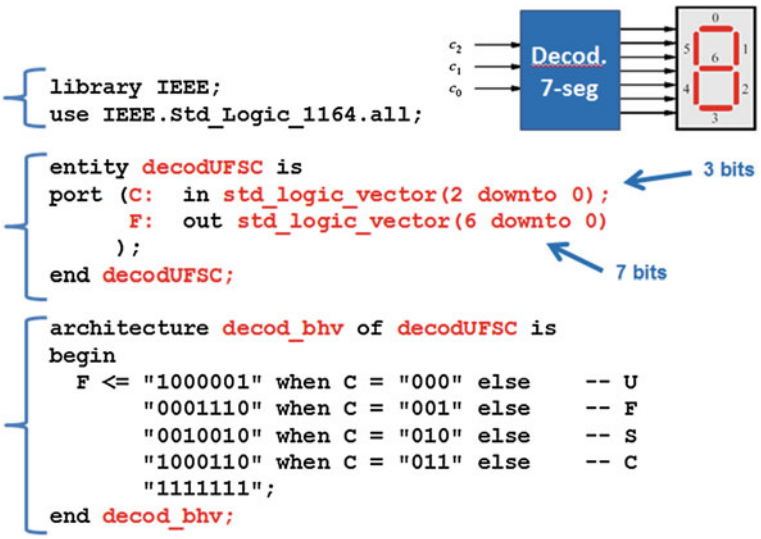
- Using the Sum of Products synthesis method

```
F(0)   <= C2′ C1′ C0′ + C2 C1 C0
F(1)   <= C2′ C1′ C0 + C2′ C1 C0′ + C2′ C1 C0 + C2 C1 C0
F(2)   <= …
```

- Using behavioral analysis

```
F <=   ″1000001″ when C2C1C0 = ″000″ else
       ″0001110″ when C2C1C0 = ″001″ else
       …
       else ″1111111″
```

The full VHDL implementation for the proposed decoder component, based on the behavioral analysis method is shown in Fig. 5.12.




```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity decodUFSC is
port (C:  in std_logic_vector(2 downto 0);
      F:  out std_logic_vector(6 downto 0)
     );
end decodUFSC;

architecture decod_bhv of decodUFSC is
begin
  F <= "1000001" when C = "000" else    -- U
       "0001110" when C = "001" else    -- F
       "0010010" when C = "010" else    -- S
       "1000110" when C = "011" else    -- C
       "1111111";
end decod_bhv;
```

Fig. 5.12 VHDL code for the UFSC 7-segment decoder

5.5 Case Study: A Simple but Fully Functional Calculator

A case study, adapted from previous chapters is presented next. The case study follows the same structure as shown in Fig. 4.8, but considering a multi-bit implementation as shown in Fig. 5.1. Furthermore, instead of components implementing arbitrary logic functions, in the proposed case study the components are conceived targeting the design of a simple but actual calculator.

It is an 8 bits calculator (operands are 8 bits long) that performs 4 operations, one arithmetic (F1, summation) and three logic functions (F2, F3 and F4), as follows:

- F1 = A + B
- F2 = A or B
- F3 = A xor B
- F4 = not A

Figure 5.13 shows the switches, displays and LEDs used by the calculator in the DE2 board.

The calculator's usage is straightforward:

1. Operand A is provided in switches $SW_{7..0}$
2. Operand B is provided in switches $SW_{15..0}$
3. One of the four operations is provided in switches $SW_{17..16}$
4. Result is shown in hexadecimal on displays HEX_1 and HEX_0 and in binary on $LEDR_{7..0}$ (red LEDs).

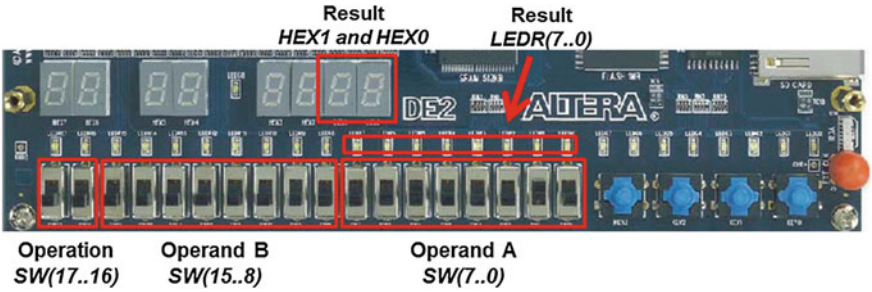


Fig. 5.13 Inputs and outputs for the calculator using DE2 board

In Table 5.1, all possible combinations for the selection bits SW_{17} and SW_{16} are listed. For instance, if the user wants to perform an exclusive or (xor) operation between inputs A (provided in $SW_{7..0}$) and B (provided in $SW_{15..8}$), SW_{17} should be switched on ('1') and SW_{16} should be switched off ('0'). The operation result is shown on the 7-segment displays and LEDs straightaway.

The block diagram in Fig. 5.14 shows the calculator input/output (I/O) signals, and the top-level component "top_calc", which is used to glue together the internal components. It shows also the two 7-segment displays and the eight LEDs used to output the calculator results. The internal components and all the connections are shown in Fig. 5.15.

The listing in Fig. 5.16 is a VHDL implementation for the top-level component shown in Fig. 5.14. In this implementation there are some important remarks:

- The "entity" section lists the signals used to identify the switches, 7-segments displays and LEDs, according to the *DE2_pin_assignments.qsf* file, as discussed in Chap. 2, Step 5;
- The internal signals, used to connect the internal components, are declared right below the "architecture" statement;
- Components C1 and C4 are listed next, but the VHDL code for components C2 and C3 are not included, as they are exactly the same as component C1;
- The Decod7seg component declaration is also not included, as this is the task to be achieved in this laboratory session;
- All the connections shown in Fig. 5.15 are performed by the VHDL code listed in the body of the architecture (right after the "begin" statement);

Table 5.1 Operations performed by the calculator

Input $SW_{17..16}$	Output $LEDR_{7..0}$ and 7-seg displays
0 0	A + B
0 1	A or B
1 0	A xor B
1 1	Not A

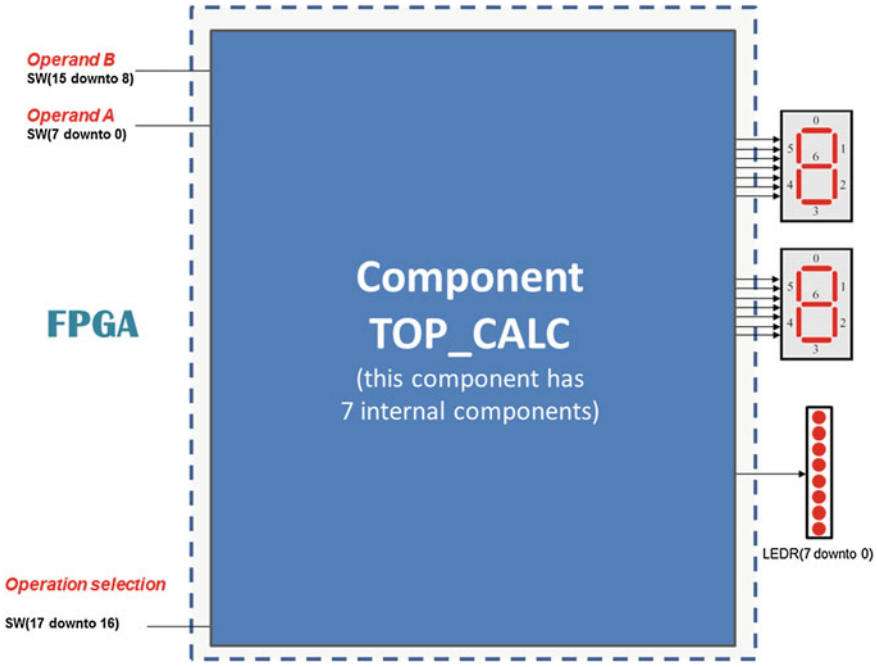


Fig. 5.14 Top-level view of the calculator's circuit

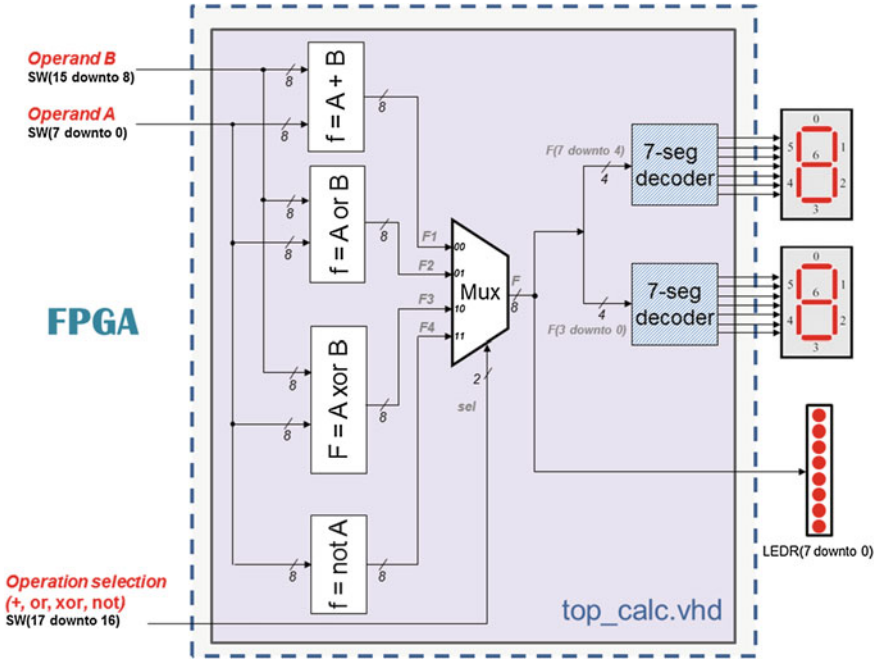


Fig. 5.15 Calculator internal components

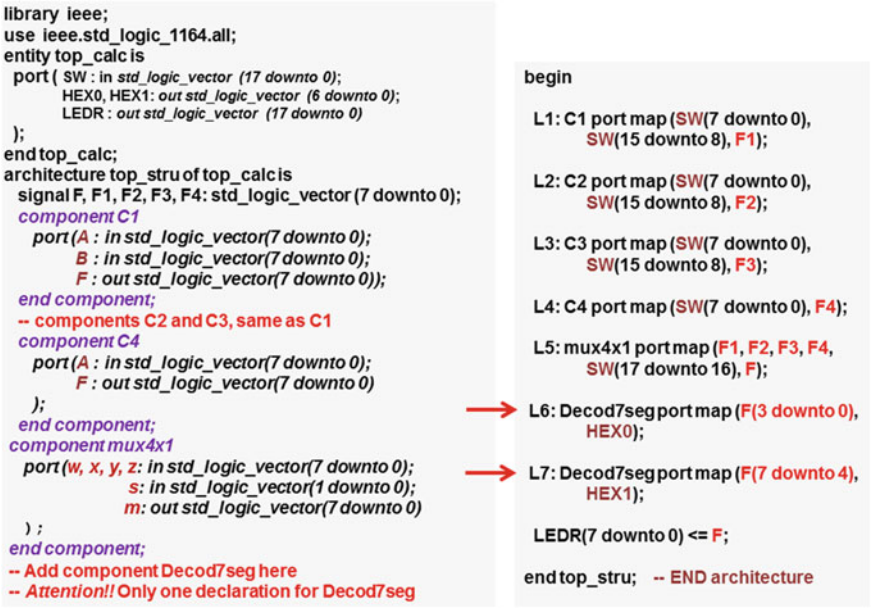


Fig. 5.16 VHDL implementation for the top circuit of Fig. 5.14

- Components C1, C2, C3 and C4 provide signals F1, F2, F3 and F4 as their outputs;
- The mux4 × 1 component has F1, F2, F3 and F4 as its inputs. $SW_{17..16}$ is the operation selection, and the internal signal F is its output;
- The Decod7seg component was declared just once in the architecture's declarations section, and in the architecture's body the two copies of the component are instantiated (see the two pointers to L6 and L7).

The VHDL implementations for the remaining components, but for the 7-segments decoder, are provided next.

5.6 Laboratory Assignment

The laboratory objectives are:

- to implement a 7-segments decoder in VHDL;
- to add the implemented decoder to the calculator.

Using the components provided in Sect. 5.5, write the VHDL implementation for the 7-segments decoder, and make the necessary modifications in the top_calc component as indicated in Fig. 5.16.

The whole design has in total 7 files (8 components), as follows:

- *c1.vhd*, provided in Fig. 5.17;
- *c2.vhd*, provided in Fig. 5.18;
- *c3.vhd*, provided in Fig. 5.19;
- *c4.vhd*, provided in Fig. 5.20;
- *mux4x1.vhd*, provided in Fig. 5.21;
- *decod7seg.vhd*, to be designed (see Sect. 5.4 and Fig. 5.12);
- *top_calc.vhd*, partially provided in Fig. 5.16.

Before starting the modifications in the top-level file (*top_calc.vhd*), make sure to have a good understanding of component declaration and instantiation procedures in VHDL. As pointed out in Fig. 5.16, just one *decod7seg* should be

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.std_logic_unsigned.all; -- to use the + operator

entity C1 is
port (A: in std_logic_vector(7 downto 0);
      B: in std_logic_vector(7 downto 0);
      F: out std_logic_vector(7 downto 0)
     );
end C1;

architecture circuit of C1 is
begin
  F <= A + B;
end circuit;
```

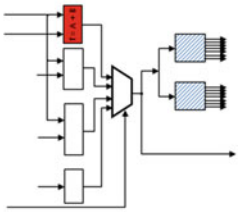


Fig. 5.17 VHDL implementation for the C1 component

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity C2 is
port (A: in std_logic_vector(7 downto 0);
      B: in std_logic_vector(7 downto 0);
      F: out std_logic_vector(7 downto 0)
     );
end C2;

architecture circuit of C2 is
begin
  F <= A or B;
end circuit;
```

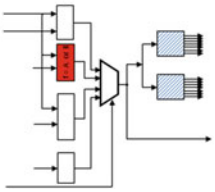


Fig. 5.18 VHDL implementation for the C2 component

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity C3 is
port (A: in std_logic_vector(7 downto 0);
      B: in std_logic_vector(7 downto 0);
      F: out std_logic_vector(7 downto 0)
     );
end C3;

architecture circuit of C3 is
begin
  F <= A xor B;
end circuit;
```

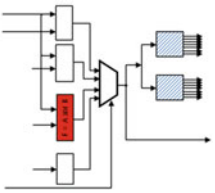


Fig. 5.19 VHDL implementation for the C3 component

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity C4 is
port (A: in std_logic_vector(7 downto 0);
      F: out std_logic_vector(7 downto 0)
     );
end C4;

architecture circuit of C4 is
begin
  F <= not A;
end circuit;
```

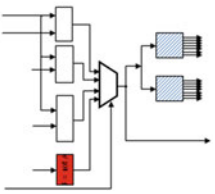


Fig. 5.20 VHDL implementation for the C4 component

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity mux4x1 is
port (w, x, y, z: in std_logic_vector(7 downto 0);
      s: in std_logic_vector(1 downto 0);
      m: out std_logic_vector(7 downto 0)
     );
end mux4x1;

architecture circuit of mux4x1 is
begin
 m <= w when s = "00" else
      x when s = "01" else
      y when s = "10" else
      z;
end circuit;
```

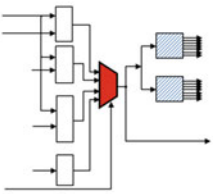


Fig. 5.21 VHDL implementation for the mux4x1 component

Fig. 5.22 Binary to 7-segments decoding table

Input	Output 6543210	Display
0000	1000000	
0001	1111001	
0010	0100100	
0011	0110000	
0100	0011001	
0101	0010010	
0110	0000010	
0111	1111000	
1000	0000000	
...	...	9, A, b, C, d
1110	0000110	
1111	0001110	

included in the components declaration section of the architecture, and two copies of this component should be instantiated using *port map* statements.

The VHDL implementation listed in Fig. 5.12 decodes only four codes, "000" ('U'), "001" ('F'), "010" ('S'), and "011" ('C'). The calculator needs to displays all possible hexadecimal values, which means that the decoder receives 16 different input data, as shown in Fig. 5.22. The expected decoded outputs are shown in Fig. 5.22, but the values in the range between 9H and DH. Thus, to design the decoder, the missing codes should be defined, and the VHDL implementation in Fig. 5.12 should be adapted according. The entity for the 7-segments decoder, as shown in Fig. 5.15, has a 4 bits input signal, and a 7 bits output signal.

The tasks to be completed in this laboratory session, using Altera's Quartus II EDA tool are as follows:

- Create a project with all the components shown in Fig. 5.15;
- Create a 7-segments decoder component using the VHDL design entry editor;

Table 5.2 Truth table for the calculator

Inputs		Outputs				
$SW_{15..0}$ B A	$SW_{17..16}$ selection	*F1* = *A* + *B* *F2* = *A or B* *F3* = *A xor B* *F4* = *not A*	Simulation HEX1 HEX0	Simulation $LEDR_{6..0}$	FPGA HEX1 HEX0	FPGA $LEDR_{6..0}$
00 00	00	F1 =				
02 03	00	F1 =				
05 0A	01	F2 =				
05 FF	01	F2 =				
05 00	10	F3 =				
05 0A	10	F3 =				
00 0A	11	F4 =				
FF F5	11	F4 =				

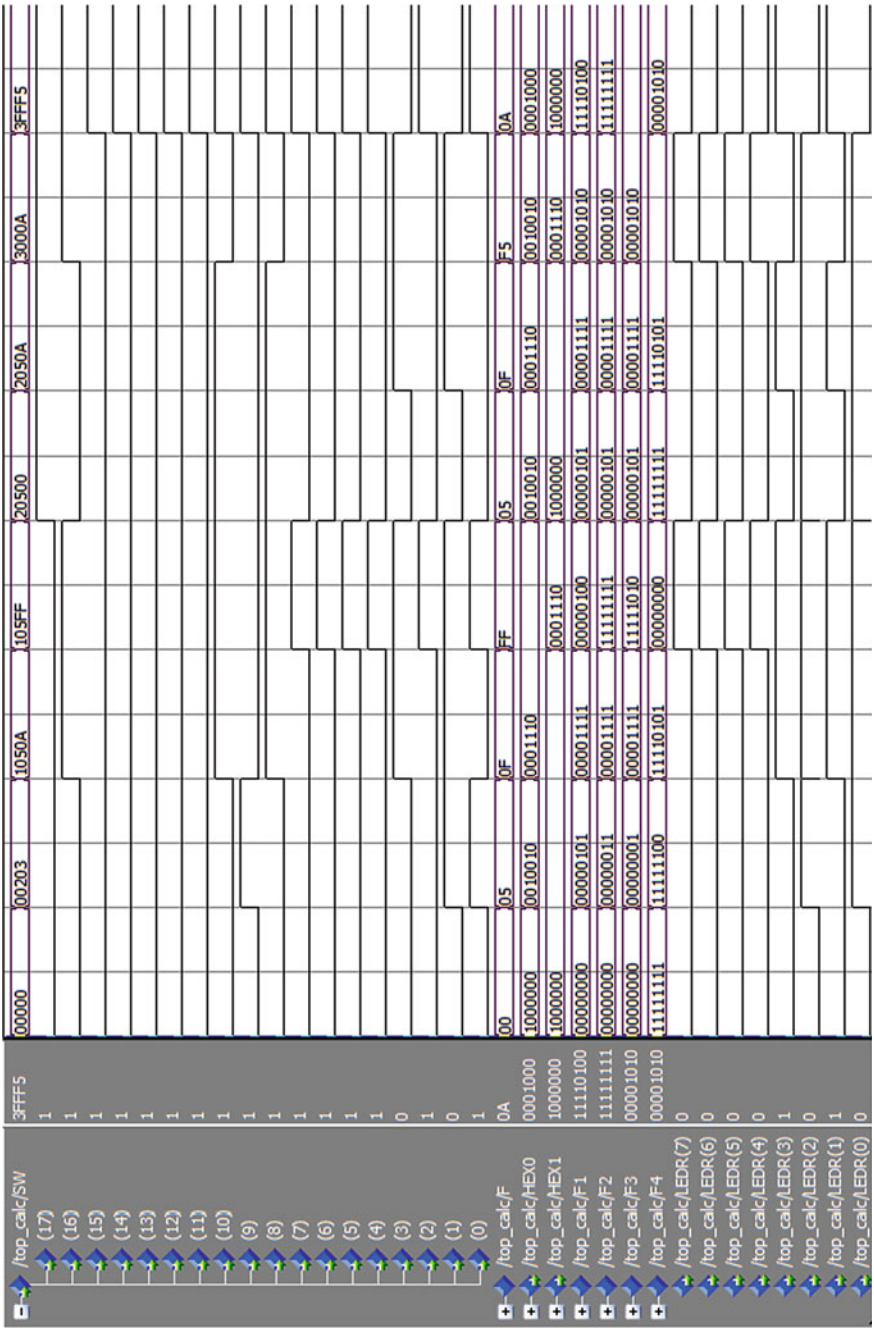


Fig. 5.23 Expected simulation results

- Modify the top-level component provided in Fig. 5.16, in order to add the new component declaration. The two instances are already created and identified by labels L6 and L7;
- Perform the synthesis;
- Perform the simulation and fix errors, if any;
- Prototype and test the circuit in the FPGA board.

The **project creation** process has been well discussed and explained in previous chapters. For the **synthesis** activity, it is essential to import the *DE2_pin_assignments.qsf* file, as discussed in Chap. 2, Step 5, and shown in Fig. 2.7.

The **simulation** activity is performed using ModelSim. To start the simulator in Quartus II, select *Tools* → *Run Simulation Tool* → *RTL Simulation*. Wait for ModelSim to open and perform the following steps:

- Simulate → Start Simulation;
- In the "Start Simulation" window, look for the "Design" tab;
- In the "work" Library, click on the "+" sign, and select the "top_calc" Entity;
- Click OK to start the simulation;
- In the Objects window drag and drop the SW vector (A, B, and the two selection bits) in the Wave window. Do the same for $LEDR_{7..0}$, $HEX0_{6..0}$, $HEX1_{6..0}$, F, F1, F2, F3 and F4.
- Using the "Force" option, as explained in Chap. 1 "Step 4", set $SW_{7..0}$ to 03H (A = 00000011), $SW_{15..8}$ to 02H (B = 00000010), and $SW_{17..16}$ to 00. Set the simulation run length to 100 ps, and press the *Run* button. Next, set $SW_{17..0}$ to the remaining values listed in Table 5.2, always running for 100 ps after each input combination.

In Fig. 5.23, the selection bits SW_{17} and SW_{16} are set to "00", "01", "10", and "11", in order to exercise all four operations. The A ($SW_{7..0}$) and B ($SW_{15..8}$) inputs are set to the sequence of values as listed in Table 3.1.

When the expected results are obtained in the simulation process, the next step is the **FPGA prototyping**. To test the calculator functionality in an actual piece of hardware, go back to Quartus II, in choose menu *Tools* → *Programmer*. Follow the instructions provided in Chap. 1, Step 5, observing the "hardware setup" instructions in Fig. 1.32 and in Fig. 1.33. Test the circuit in the FPGA board, using switches $SW_{15..0}$ to provide the A and B inputs, and switches $SW_{17..16}$ to choose the operation to be performed. The operation result is shown in binary in $LEDR_{7..0}$, and in hexadecimal on the 7-segment displays HEX0 and HEX1.

In Table 5.2, the first column has some input value examples. The second column is filled in with the selection bits ("00", "01", "10", and "11"). The third column should be filled in with the results obtained from the manual evaluation of equations F1, F2, F3 and F4. The fourth and fifth columns can be obtained straight from the simulation results, according to the waveforms in Fig. 5.23. The sixth and seventh columns should be filled in with the FPGA execution results.

Chapter 6
Sequential Circuits, Latches and Flip-Flops

The first part of this book introduces the concepts related to the design of combinational circuits in VHDL. In the second part of this book, starting from this chapter on, the design of sequential circuits in VHDL is discussed and explained. At the end of the chapter, the reader should be able:

- to understand the concepts and principles of sequential circuits;
- to understand the concepts and differences between latches and flip-flops;
- to understand the principles of registers design in VHDL;
- to design and test flip-flops, latches and registers in VHDL;
- to design and implement the proposed case study using an FPGA board.

6.1 Sequential Circuits in VHDL: The Process Statement

In the *combinational circuits* discussed in previous chapters, the results presented in their outputs depend only on their current input values. The outputs of a *sequential circuit*, on the other hand, depend also on previous values internally stored. For this reason, a sequential circuit must be designed making use of some sort of memory resources. 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. The design of latches and flip-flops in VHDL is explored later on in this chapter.

In VHDL, as introduced in previous chapters, all signal assignments are performed in parallel. This language feature is very interesting to describe the behavior of combinational circuits. The behavior of sequential circuits is described in VHDL through the *process* statement. In Fig. 6.1 there is an example of process usage in VHDL. Important characteristics regarding the functionality of this statement are listed next:

E. A. Bezerra and D. V. Lettnin, *Synthesizable VHDL Design for FPGAs*,
DOI: 10.1007/978-3-319-02547-6_6,

Fig. 6.1 Process statement in VHDL

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity ProcessExample is port (
4      C: in  std_logic;
5      D: in  std_logic;
6      Q: out std_logic
7  );
8  end ProcessExample;
9  architecture behv of ProcessExample is
10     signal A, B: std_logic;
11 begin
12     A <= D;
13     Q <= B;
14     P1: process (C, A)
15     begin
16         B <= '0';
17         if (C = '1') then
18             B <= A;
19         end if;
20     end process P1;
21 end behv;
```

- A *process* defines a sequence of commands to be performed by the circuit. This means that all the commands listed from line 16–19 in Fig. 6.1 are performed in the defined order, and one after the other.
- A *process* is cyclic, and it never ends. However, its list of commands (e.g. lines 16–19 in Fig. 6.1, and lines 17–21 in Fig. 6.2) is performed only when a change is noticed in one of its activation signals (e.g. C and D on line 14 in Fig. 6.1).
- The signals activation list of a process is known as *the sensitivity list*. A process is triggered by new "events" detected on its sensitivity list.
- After the last command of a process has been performed, the first command in the sequence is run again, but only after the next time a change is picked up in one or more signals of the sensitivity list (e.g. line 14 in Fig. 6.1).
- Some VHDL constructions can only be used inside a process as, for instance, the IF.. THEN.. ELSE shown on lines 17–19 in Fig. 6.1.
- Signals cannot be declared inside a process.
- In a process, in case of several assignments for the same signal, only the last assignment will be performed. In Fig. 6.2, B will be assigned only on line 18, and C will be assigned only on line 20.

An important VHDL process concept is that all signal attributions performed will only be valid when the execution reaches the *end process* statement. Two versions of the example in Fig. 6.2 have been implemented, and the simulation results are shown in Fig. 6.3. In Version 1, the sensitivity list of the process has only the A signal, and in Version 2 it has the A and C signals. The waveforms in Fig. 6.3 show the simulation results for Version 1 (top of the figure) and Version 2

```
1  library IEEE;
2  use IEEE.Std_Logic_1164.all;
3  entity TestProc is
4  port(
5      data_in  : in  std_logic;
6      data_out : out std_logic
7  );
8  end TestProc;
9  architecture circuit of TestProc is
10   signal A, B, C, D: std_logic;
11 begin
12   A <= data_in;
13   data_out <= D;
14   -- Version 1      -- Version 2
15   process (A)       -- process (A, C)
16   begin
17     B <= A;
18     B <= '0';
19     C <= A and '1';
20     C <= not A;
21     D <= C;
22   end process;
23 end circuit;
```

sequential statements

Fig. 6.2 Sequence of signal attributions in a process

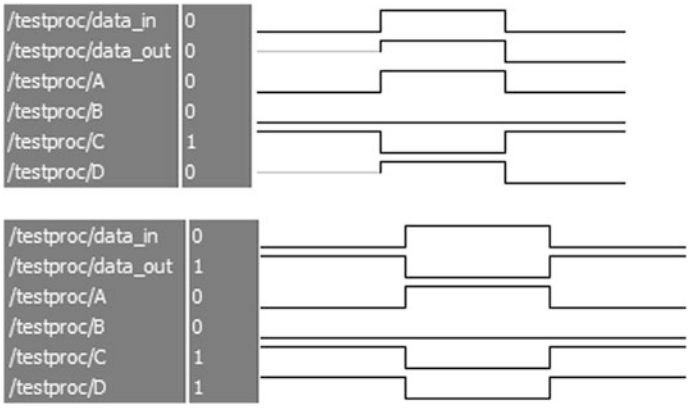


Fig. 6.3 Simulation results for versions 1 and 2 of the VHDL code listed in Fig. 6.2

(bottom of the figure). In both versions, the *data_in* input receives the values '0', '1' and '0', in this order, and when the simulation starts there is no history of previous values for any of the signals.

In Version 1, when A receives '0' from *data_in*, the process is triggered as it has signal A on its sensitivity list. Next, all the assignments are performed, sequentially. B receives '0' on line 18 in Fig. 6.2, overwriting the assignment performed on line 17. C receives '1' on line 20 (*Not A*, where A = *data_in* = '0'), overwriting the assignment performed on line 19. On line 21, D receives the value stored in C but, as shown in the top waveform in Fig. 6.3, the simulation has just started, and the simulator has no knowledge of the value of C. For this reason, the waveform shows neither '0' nor '1'. At this point, it is important to notice that the C value assigned on line 20 in Fig. 6.2 cannot be used for the D assignment on line 21, as in a process, all assignments are performed sequentially, and considering the values the signals hold when the process started.

Proceeding with the simulation of Version 1, next A receives '1' from *data_in*, and the process is triggered again as a result of this change in A ('0' to '1'). Once more, all the assignments are performed, sequentially. B receives '0' on line 18 in Fig. 6.2. C receives '0' on line 20 (*Not A*, where A = *data_in* = '1'). D receives the value stored in C on line 21, which is '1'. This is the value stored in C, when the process started, and not the value just assigned to C on line 20.

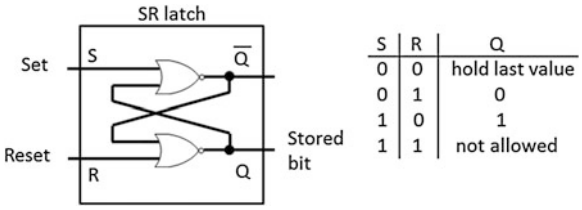
In Version 2, the only difference is the C signal added to the process sensitivity list. This tiny modification results in significant differences in the simulation results. Initially, A receives '0' from *data_in*, and the process is triggered. The assignments are performed sequentially starting by B receiving '0' on line 18 in Fig. 6.2, overwriting the assignment performed on line 17. Next, C receives '1' on line 20 (*Not A*, where A = *data_in* = '0'), overwriting the assignment performed on line 19. On line 21, D receives the value stored in C and, as there was a change in C, and as C is now on the process sensitivity list, then the process is triggered once more and D also receives '1', which is the new value of C. This simulation result is shown in the bottom waveform of Fig. 6.3.

In Fig. 6.1 the process has a label (P1 on line 14). This identification is optional, but it is very convenient when performing the simulation of a VHDL code with several processes. The sensitivity list is also optional in a process. Though, in case of processes with no sensitivity list, a *wait* statement must be included in the process body in order to hold the process from time to time.

The behavioral description of sequential circuits in VHDL is an important topic, and it will be further discussed in Chap. 9 together with a more detailed discussion of processes.

6.2 Describing a D Latch in VHDL

As mentioned before, the output of a sequential circuit depends not only on its inputs, but also on data previously stored. Consequently, sequential circuits require the implementation of storage resources. Figure 6.4 shows a basic storage element, the set-reset (SR) latch. This simple circuit manages to store data (a single bit)



S	R	Q
0	0	hold last value
0	1	0
1	0	1
1	1	not allowed

Fig. 6.4 A basic SR latch and its truth table (adapted from [Vahid11])

```vhdl
-- Version 1 - Using process
library ieee;
use ieee.std_logic_1164.all;
entity SR_Latch_Proc is port (
    S, R : in  std_logic;
    Q, Qn: out  std_logic
);
end SR_Latch_Proc;
architecture behv of SR_Latch_Proc is
   signal auxQ, auxQn: std_logic;
begin
   process (S, R, auxQ, auxQn)
   begin
      auxQ  <= auxQn nor R;
      auxQn <= auxQ nor S;
   end process;
   Q  <= auxQ;
   Qn <= auxQn;
end behv;
```

```vhdl
-- Version 2 - Combinational circuit
library ieee;
use ieee.std_logic_1164.all;
entity SR_Latch is port (
      S, R : in  std_logic;
      Q, Qn: out  std_logic
);
end SR_Latch;
architecture behv of SR_Latch is
   signal auxQ, auxQn: std_logic;
begin
   auxQ  <= auxQn nor R;
   auxQn <= auxQ nor S;
   Q  <= auxQ;
   Qn <= auxQn;
end behv;
```

Fig. 6.5 VHDL implementation for an SR latch

through the cross-feedback loop in the pair of cross-coupled NOR gates shown in Fig. 6.4. The stored bit is available on the Q output signal.

The truth table in Fig. 6.4 describes the SR latch functionality, which is straightforward: when the S input is '0' and the R input is '1', the circuit outputs '0' ("reset" the circuit, forcing its Q output to '0'); when the S input is '1' and the R input is '0', the circuit outputs '1' ("set" the circuit, forcing its Q output to '1'); when both S and R inputs are equal to '0', the circuit holds its current state (memory!). In all situations, one of the outputs is always the complement of the other. However, when both inputs S and R are '1', the circuit can become unstable and, consequently, this is not a valid input for this circuit.

Two versions for the VHDL implementation of an SR latch are shown in Fig. 6.5. The first version, on the left, uses a process, while the second version uses just combinational logics.

The SR latch is a good option to show how a basic storage circuit works. However, the "not allowed" input suggests that extra precautions should be taken during this circuit's usage. Basically, in most applications, the circuit around the SR latch should be designed in order to never allow a '1' value to be present in both S and R inputs simultaneously.

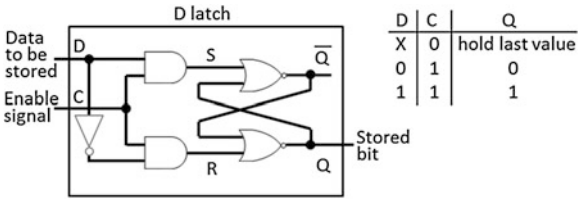


Fig. 6.6 A D latch and its truth table (adapted from [Vahid11])

A possible solution for this problem is the D latch, which is an evolution of the SR latch. A D latch is built adding two AND gates and an inverter in front of the SR latch, as shown in Fig. 6.6. With this arrangement, the "not allowed" input of the SR latch cannot happen. Now the circuit has just one data input, D. The other input, C, shown in Fig. 6.6 is used to control when a new data is stored in the memory (the D latch circuit). While the C input is '1', the input data D is stored. When C is '0', the AND gates are "closed", and changes on the D input cannot be stored in the circuit (the SR latch).

The D latch could be implemented in VHDL following the same strategy as the SR latch shown in Fig. 6.5. However, a better and more elegant way of describing a D latch is shown in Fig. 6.7. This VHDL implementation is a "behavioral" description of the D latch. The *if* statement on line 13, describes the exact behavior of the D latch truth table, i.e., when C is '1', the input data provided in D is stored. Otherwise, when C is '0', the circuit holds the value provided in D, when C was

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

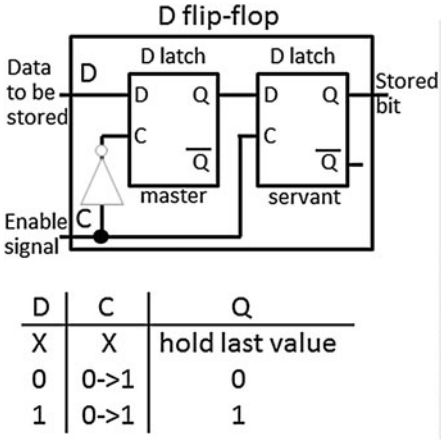
Fig. 6.7 VHDL implementation for a D latch

'1' for the last time. The waveform in Fig. 6.7 shows the D latch behavior according to C and D inputs.

The D latch is level activated, as the circuit output (Q) follows the value presented at its input (D), while the C input (which is usually connected to the clock) is active, i.e. C is kept at level '1'. This behavior is called "transparent". The advantage of the D latch is its low cost on circuitry. The 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 C = '1' for a long period of time, the data at the D input might be propagated through many latches. Otherwise, if C = '1' for a too short period of time, it may not enable a store. The terms "long period" and "short period" are dependent on the target technology.

6.3 Describing a D Flip-Flop in VHDL

The D flip-flop is conceived aiming applications where the aforementioned situation of controlling the exact moment when the input data is stored, is an important issue. In a D flip-flop, the input data is stored just when there is a transition from '0' to '1' (or from '1' to '0') on the C input. So, the D flip-flop is said to be controlled by "edge", while the D latch is controlled by "level". In a D latch, whenever the C input is '1' (active high logic level), the D value is stored in the circuit. In a D flip-flop, the D value is stored only when the C input changes from '0' to '1' (a rising edge event). In other versions of these circuits, the D latch stores the input data when C = '0' (active low logic level), and the D flip-flop stores the input data when C changes from '1' to '0' (a falling edge event). The D Flip-flop can be implemented according to different design styles. In a typical design, two D latches are connected in a master-servant topology, as shown in Fig. 6.8.



D	C	Q
X	X	hold last value
0	0->1	0
1	0->1	1

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

Fig. 6.8 D flip-flop. Block diagram, truth table, and VHDL implementation (adapted from [Vahid11])

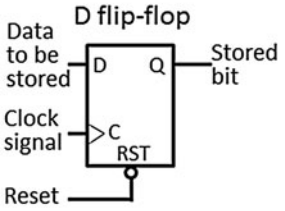
In Fig. 6.8, the output (Q) of the master latch is connected to the input (D) of the servant latch. The clock signal for the master latch is inverted in relation to the clock signal for the servant latch. In this sense, the master is loaded when C = '0', and the servant when C = '1'. When C changes from '0' to '1', the master is disabled and the servant is loaded with the value that was at D just before C has changed. Outside the active clock edge there is no transfer of the input value to the servant latch. As a tradeoff this architecture needs more logic gates to be implemented than the D latch. However, gate count is less of an issue nowadays.

The VHDL code listed in Fig. 6.8 is a behavioral implementation for the described D flip-flop. The hardware detection of a rising edge event on the C input is described in VHDL on line 13. A VHDL synthesis tool understands this sort of description and a physical D flip-flop is used in the final circuit, in case this D flip-flop resource is available in the target technology.

There are several variations for the D flip-flop implementation. In Fig. 6.9, a reset control signal has been added to the circuit. In this case, when the RST input is zero, the output will also be zero. Otherwise, when RST is '1', then the Q output will depend on the D and C inputs, as explained before. This D flip-flop with asynchronous reset VHDL implementation is shown on line 14. In a synchronous reset implementation, line 14 should be placed inside the *c'event* test.

Notice in Fig. 6.9, on line 16, the *elseif* VHDL statement usage. This combination of if (*RST* = '0') and *elseif* (*C'event and C* = '1'), tells the synthesis tool to generate D flip-flop with asynchronous reset. *Very important!! Any change in this syntax, may result in the synthesis tool generating another sort of circuit, which may not be the expected D flip-flop.*

Figure 6.9 shows the block diagram (symbol) of a D flip-flop with asynchronous reset. In this diagram, internal details as logic gates and latches are hidden,



D	C	RST	Q
X	X	0	0
X	X	1	hold value
0	0->1	1	0
1	0->1	1	1

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

Fig. 6.9 D flip-flop with asynchronous reset

and only the input and output signals are shown. The small triangle on the C input represents a "clock" signal in digital systems. For the sake of simplicity, this symbol and connotation has not been used in the previous flip-flop diagram shown in Fig. 6.8. However, usually in flip-flop based designs, the C input is always connected to a clock signal in order to synchronize the data read/write operations with the remaining components of the circuit. Basically, when a component needs to store a bit in a flip-flop, if the component operations are coordinated by the same clock signal as the flip-flop, then it is possible to assure that the correct data is written. In a typical synchronous digital system, operations take place in the rising (or falling) edge of the global clock signal. It is not unusual for current digital systems to present several clock signals, and the implementation of such systems is a challenge for the designers.

Figure 6.10 shows a D flip-flop with asynchronous reset, and an *Enable* input. The VHDL implementation describes the behavior of this flip-flop where, basically, whenever the reset signal is high (RST = '1'), and there is a rising clock edge, if the enable signal is high (EN = '1'), then the input data is stored (Q <= D).

The waveform in Fig. 6.11 shows the simulation results for the D flip-flop with enable and asynchronous reset. The CLK input is fed with a clock signal, and the simulation runs for 9 clock cycles. In the first 4 clock cycles, the reset signal (RST) is kept in active-low level ('0'), and no action takes place in the flip-flop. As the reset is asynchronous, the *elseif* statement cannot be reached while RST = '0' (see line 13 in Fig. 6.10), and the changes on input signals EN and D have no effect in the circuit state. Next, in the beginning of the 5th clock cycle, the reset goes high, the enable (EN) goes low, and the D input remains high. In this case, as the enable signal is low, again the input data (D) cannot be stored in the

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

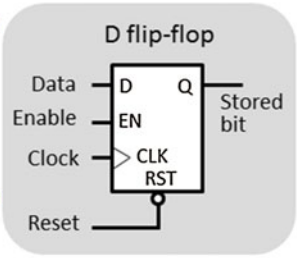


Fig. 6.10 D flip-flop with enable and asynchronous reset

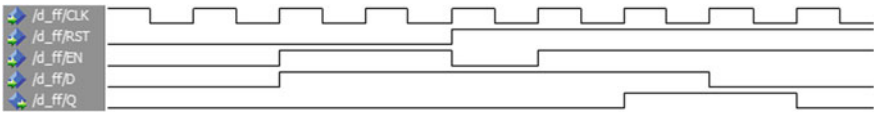


Fig. 6.11 Simulation waveform for the D flip-flop with enable and asynchronous reset

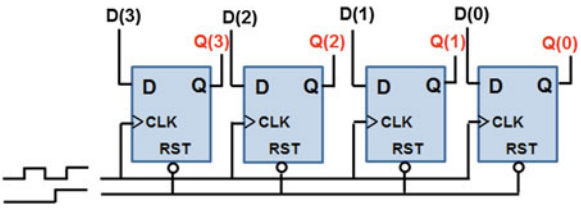


Fig. 6.12 Block diagram of a 4-bits register based on D flip-flops

flip-flop (see line 16 in Fig. 6.10). The enable signal goes high again in the beginning of the 6th clock cycle, simultaneously with the rising clock edge. In this case, there is no change in the flip-flop memory contents, as the enable signal should have been placed in an active-high level, before the rising clock edge event. Finally, in the rising edge of 7th clock cycle, the enable signal was already in active-high level, and the D input, which is '1', is stored in the flip-flop.

6.4 Implementing Registers with D Flip-Flops

A D flip-flop can be used to store a single bit of data. For circuits that perform operations with larger word sizes, registers should be used instead. A register can be built from an arrangement of D flip-flops, as can be seen in Fig. 6.12. Registers with 32 or 64 bits are usually found in commercial microprocessors.

In the VHDL implementation in Fig. 6.13, a 4-bits word present on the D input is stored in the register whenever a rising clock edge is provided in the CLK input. In order to implement a register with enable and asynchronous reset signals, just replace line 17 of the VHDL code listed in Fig. 6.13, by lines 16–18 of Fig. 6.10. It is also necessary, obviously, to add the enable signal declaration in the register entity (*EN: in std_logic;*). Notice that on line 14 of Fig. 6.10, the single bit *Q* assignment should be replaced by *Q* <="0000", as now Q is a 4 bits vector. In order to have a more generic code, the VHDL statement *Q* <= (*others* => '0'); could be used instead. In this case, all bits of Q will receive '0', no matters Q length.

Once more, it is important to notice that synthesis tools are not as smart as one expect them to be. This means that there is almost no flexibility to describe flip-flops and registers in VHDL. The designer who wants a D flip-flop or register properly synthesized, should follow the models shown in this book and in other available documents. Otherwise, the synthesis tool may not generate the hardware the designer wants.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity D_4FF is port (
4    CLK:  in std_logic;
5    RST:  in std_logic;
6    D   :  in std_logic_vector(3 downto 0);
7    Q   : out std_logic_vector(3 downto 0)
8  );
9  end D_4FF;
10 architecture behv of D_4FF is
11 begin
12   process(CLK, D)
13   begin
14     if RST = '0' then
15         Q <= "0000";  -- or (others => '0');
16     elsif (CLK'event and CLK = '1') then
17         Q <= D;
18     end if;
19   end process;
20 end behv;
```

Fig. 6.13 VHDL description of a 4-bits register based on D flip-flops

6.5 Laboratory Assignment

The laboratory objectives are:

- to design 4-bits and 8-bits registers with enable and asynchronous reset;
- to add the implemented registers to the calculator developed in the last chapter.

Using as example the registers and flip-flops provided in previous sections, write the VHDL implementations for the 4-bits and 8-bit registers shown in Fig. 6.14. Also, make the necessary modifications in the top_calc component as needed to add the registers to the design.

The whole design has a total of 9 files (8 components + top), as follows:

- *c1.vhd*, provided in Fig. 5.17;
- *c2.vhd*, provided in Fig. 5.18;
- *c3.vhd*, provided in Fig. 5.19;
- *c4.vhd*, provided in Fig. 5.20;
- *mux4x1.vhd*, provided in Fig. 5.21;
- *reg4bits.vhd*, should be adapted from Figs. 6.10 and 6.13;
- *reg8bits.vhd*, should be adapted from Figs. 6.10 and 6.13;
- *decod7seg.vhd*, implemented in the previous chapter, according to Sect. 5.4 and Fig. 5.12;

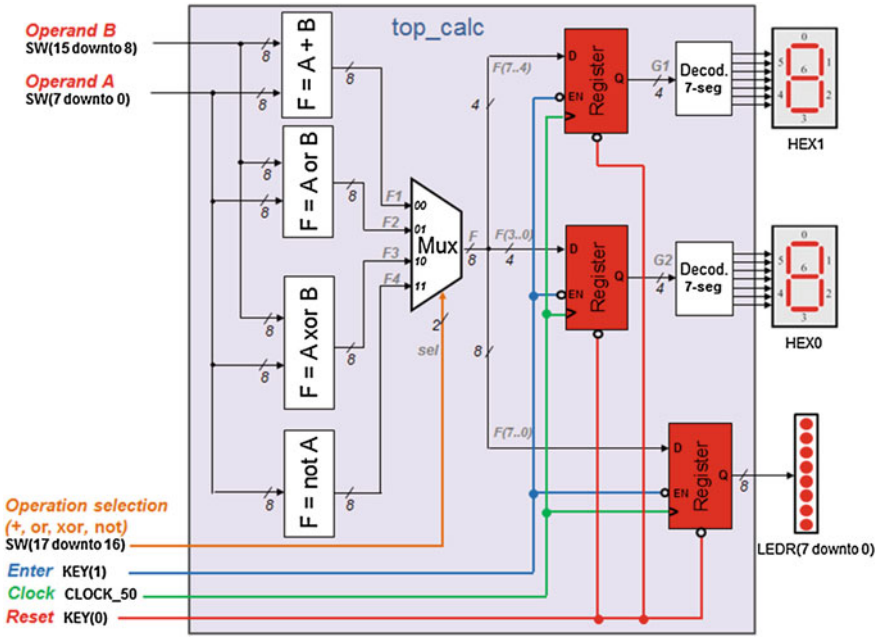


Fig. 6.14 Inserting registers in the calculator to store operation results

- *top_calc.vhd*, partially provided in Fig. 5.16. The previous chapter implementation should be adapted in order to add the new registers, and also the new input signals shown in Fig. 6.14.

The tasks to be completed in this laboratory session, using Altera's Quartus II are as follows:

- Create a project with all the components shown in Fig. 6.14;
- Using the VHDL design entry editor, create a 4-bits register and a 8-bits register, both with an enable signal, and asynchronous reset;
- Modify the top-level component developed in the previous chapter, in order to add the new components (the three registers shown in Fig. 6.14). Notice that new signals are needed to connect the register outputs, to the decoder inputs.
- Perform the synthesis;
- Perform the simulation and fix errors, if any;
- Prototype and test the circuit in the FPGA board.

The new registers used to store the calculator results are as follows:

- A 4-bits register to store the Less Significant Bits (LSB) of the operation result, to be presented in HEX0;
- A 4-bits register to store the Most Significant Bits (MSB) of the operation result, to be presented in HEX1;

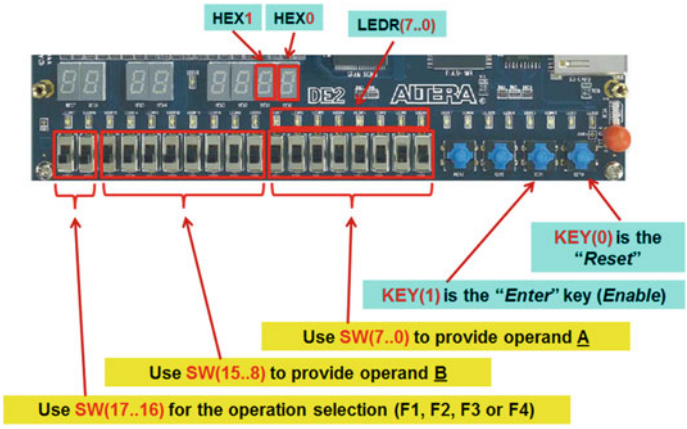


Fig. 6.15 Input and output user interface for the calculator

- An 8-bits register to store the operation result to be shown in binary on the red LEDs.

Figure 6.15 shows the user interface for the new calculator. The calculator's operation is straightforward:

- When KEY(0) is pressed (Reset), the flip-flops (registers) should be cleared. As a result, the LEDs are switched off, and the 7-segments displays (HEX0 and HEX1) should show a '0'.
- When KEY(1) is pressed (Enter), the registers are enabled for writing, allowing the operation result to be stored (memory!).

Additionally, make sure to connect the clock input to the CLOCK_50 signal provided by the DE2 board. This is a 50 MHz crystal available on the board. The top entity should be changed in order to include the new DE2 input signals:

- key: in std_logic_vector(1 downto 0); – – KEY(0) e KEY(1)
- clock_50: in std_logic; – – clock 50 MHz

Also, it is important to notice that the four push buttons available in DE2 board are active low. This means that when they are pressed, a zero ('0') is generated. In addition, there is a Schmitt Trigger based debounce circuit for the push buttons, in order to help to have just one zero pulse each time a button is pressed.

As stated before, the *project creation* process has been well discussed and explained in previous chapters. For the *synthesis* activity, it is essential to import the *DE2_pin_assignments.qsf* file, as discussed in Chap. 2, Step 5, and shown in Fig. 2.7.

The *simulation* activity is performed using ModelSim. To start the simulator in Quartus II, select *Tools→Run Simulation Tool→RTL Simulation*. Wait for ModelSim to open and perform the steps described in previous chapters.

When the expected results are obtained in the simulation process, the next step is the *FPGA prototyping*. As described in previous chapters, to download the generated bitstream to the FPGA, using Quartus II choose menu *Tools→Programmer*. Follow the instructions provided in Chap. 1, Step 5, observing the "hardware setup" instructions in Figs. 1.32 and 1.33. Test the circuit in the FPGA board, using the switches and keys shown in Fig. 6.15. The operation result is shown in binary in $LEDR_{7..0}$, and in hexadecimal on the 7-segment displays HEX0 and HEX1.

Chapter 7
Synthesis of Finite State Machines

A finite state machine (FSM) is a powerful tool used to build control circuits. Control circuits are employed in situations where a sequence of operations should be performed, according to a pre-defined execution order. The use of a FSM as a control circuit is better explored in the next chapters. This chapter introduces the basic concepts related to the design of FSMs. At the end of the chapter, the reader should be able:

- to understand the concept of FSMs;
- to understand the synthesis process of FSMs in VHDL;
- to understand the concept of counters implemented in VHDL using FSMs;
- to design and to implement in VHDL an FSM based counter.

7.1 Finite State Machines

As already discussed in Chap. 1, a computer system usually consists of a control module (i.e. control unit) and an operations execution module (i.e. datapath), as shown in Fig. 1.1. The control module is responsible for coordinating the sequence of activities to be performed by a given system. In digital systems, control signals are generated by sequential circuits. A sequential circuit passes through a series of states, and each state (every time), may provide a certain output. The outputs of each state are the signals used to control the sequence of activities executed by a target system.

In Fig. 7.1, a state diagram is used to model an FSM. The state diagram is represented by a directed graph, where the vertices (or nodes) are the FSM states, and the edges (or arcs) represent the transitions from one state to another. As shown in Fig. 7.1, an FSM has the following components:

- *States* are represented in the graph by nodes S_0, S_1, S_2 and S_3. The states are responsible, basically, for storing information regarding past changes in the

E. A. Bezerra and D. V. Lettnin, *Synthesizable VHDL Design for FPGAs*,
DOI: 10.1007/978-3-319-02547-6_7,

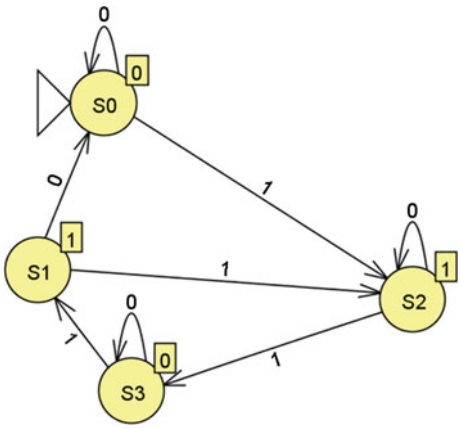


Fig. 7.1 Example of a state diagram for an FSM

inputs. In the example FSM, the triangle pointing to S_0 indicates that this is the initial state.

- *Transitions* are represented in the graph by the edges. A transition indicates a state change by means of a condition that enables the switch from a state to another.
- *Actions* are activities to be performed when the respective state is reached.

A hardware implementation of an FSM comprises a combinational circuit, and a register. As shown in Fig. 7.2, the FSM's current state is stored in the register. The combinational circuit uses the circuit's inputs and the stored "current state", in order to calculate the "next state". The combinational circuit calculates also the FSM's output.

An FSM can also be represented through a state transition table. In Table 7.1, it can be observed that when the FSM is in state S_0, a transition (change) to state S_2, takes place only when the circuit's input is '1' ("Inputs" in Fig. 7.2). While the circuit's input stays in '0', the FSM will remain in the same state S_0, as shown on the first row of Table 7.1. This action is also shown graphically in the loop with the '0' input in state S_0 in Fig. 7.1. The fourth column in Table 7.1 shows the output provided by the FSM for each state. In Fig. 7.1, the output for each state is represented by the numeric value in a square at the top right hand corner of the nodes. In Fig. 7.2, the output is represented by the signal "Outputs".

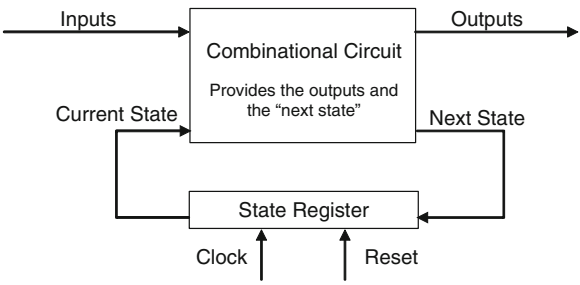


Fig. 7.2 Block diagram of an FSM hardware implementation

Table 7.1 State transition table for the FSM shown in Fig. 7.1

Current state	Input	Next state	Output
S_0	0	S_0	0
S_0	1	S_2	0
S_1	0	S_0	1
S_1	1	S_2	1
S_2	0	S_2	1
S_2	1	S_3	1
S_3	0	S_3	0
S_3	1	S_1	0

The state transition table (Table 7.1) is a direct representation of the state diagram of an FSM.

Considering that the circuit in Fig. 7.2 implements the FSM shown in Fig. 7.1, the register stores the FSM current state (Table 7.1 first column), and the combinational circuit calculates the "next state" (Table 7.1 third column) using the stored "current state" and the circuit's inputs (Table 7.1 second column). The combinational circuit calculates also the FSM's output (Table 7.1 fourth column).

A possible VHDL implementation for the FSM is shown in Fig. 7.3. In that implementation, a transition happens each time an event of a rising clock edge takes place. This event is represented in Fig. 7.1 by the graph's edges, and in Fig. 7.2 by the *clock* input.

7.2 VHDL Synthesis of Finite State Machines

An FSM is, essentially, a sequential circuit. Basically, in an FSM, from the combination of a given sequence of inputs and past states, the next states, and the respective outputs are generated sequentially.

As aforementioned in Chap. 6, a sequential circuit can be implemented in VHDL using the *process* statement. There are several ways to describe an FSM in VHDL, using 1, 2 or even 3 processes.

The FSM studied in Fig. 7.1 has been implemented in VHDL using the two processes strategy. The source code for this implementation is shown in Fig. 7.3. An important VHDL concept useful for FSMs description is the enumeration type. With the VHDL reserved word *type* the developer can create a new type. In Fig. 7.3, line 7, STATES is a new type that can hold the values S0, S1, S2 and S3. On line 8, CS (i.e. current state) and NS (i.e. next state) are two internal signals defined as STATES type. This means that these two signals can be assigned only to other signals or constants of the same type STATES. It is also important to highlight that S0, S1, S2 and S3 are not part of the VHDL language. These are just labels used in the set of possible values to be used in the new enumeration type. During the synthesis process, these labels will be translated into ones and zeros by

```
1  entity MOORE is
2      port(X, clock, reset: in std_logic;
3           Z              : out std_logic);
4  end;
5
6  architecture TwoProcesses of MOORE is
7      type STATES is (S0, S1, S2, S3);    VHDL Enumeration Type
8      signal CS, NS : STATES;
9  begin
10     process (clock, reset)              1st process: implements a register
11     begin                               that stores the current state (CS)
12         if reset= '1' then              as a function of the next state (NS)
13             CS  <= S0;
14         elsif clock'event and clock='1' then
15             CS <= NS;
16         end if;
17     end process;
18                                         2nd process: implements the circuit
19     process(CS, X)                      that generates the Z output,
20     begin                               according to the current state (CS)
21         case CS is                      and the X input
22             when S0 =>  Z <= '0';
23                         if X='0' then NS <= S0; else NS <= S2; end if;
24             when S1 =>  Z <= '1';
25                         if X='0' then NS <= S0; else NS <= S2; end if;
26             when S2 =>  Z <= '1';
27                         if X='0' then NS <= S2; else NS <= S3; end if;
28             when S3 =>  Z <= '0';
29                         if X='0' then NS <= S3; else NS <= S1; end if;
30         end case;
31     end process;
32 end TwoProcesses;
```

Fig. 7.3 VHDL implementation of an FSM using two processes

the EDA tool, in order to generate the final hardware implementation for the abstract VHDL description. The VHDL syntax for the enumeration type is as follows:

$$\text{type indentifier is (element1, element2, element3, ...);}$$

where, "type" and "is" are VHDL reserved words, and "identifier" is the name of the new type that is being created. Following the "is" statement, there is an ordered set of values to be used in the VHDL description as the values to be assigned to the signals of type "identifier".

In Fig. 7.3, line 13, an element belonging to the set of enumeration values is directly assigned to the CS signal. In Fig. 7.3, line 15, there is an example of assignment of a signal of type STATES to another signal of the same type.

The VHDL synthesis, using two processes, of the FSM described in Fig. 7.1 is straightforward. The process P1 defines the current state (CS) for the processes P2. Process P1 is sensitive to the clock and reset signals. If an asynchronous reset

(reset = '0') occurs, CS receives the S0 value, otherwise, if a rising clock edge occurs CS is updated with the next state (NS) signal, that is, it is responsible to perform the state transition process. The process P2 updates the (NS) signal and defines new values for the output Z signal according to the CS.

In both representations, the graphical (Fig. 7.1) and the textual (Fig. 7.3) one, it is possible to notice that the circuit's outputs are defined in each of the states. As listed in Table 7.1, for states S0 and S3, the output is '0', and for states S1 and S2, the output is '1'. This type of FSM is known as a Moore machine, as the output values depend only on the CS. Another type of FSM is known as a Mealy machine, where the outputs are defined according not only to the CS, but also to the current inputs.

Using the enumeration type in VHDL designs is important also in the simulation step. As shown in Fig. 7.4, the output values (Z) are well identified for each CS state. All the transitions between states are also well identified, as the names S0, S1, S2 and S3 are a good help for the simulation visualization. In case the enumeration type had not been used in VHDL code, the CS and NS signals would have shown "00", "01", "10" and "11" instead, making it a bit more difficult to visualize all the transitions. In FSMs with more states, the situation would be even more complicated.

Another approach to describe an FSM in VHDL employs three processes instead of two. As shown in Fig. 7.5a, the P1 process updates the (CS), and sends this information to P2 and P3. In Fig. 7.5b, P2 computes the next state, but does not perform the transition. The (NS) information is sent from P2 to P1, which is the responsible for updating the (CS). P3 is the process responsible for signals (outputs) assignments and updates.

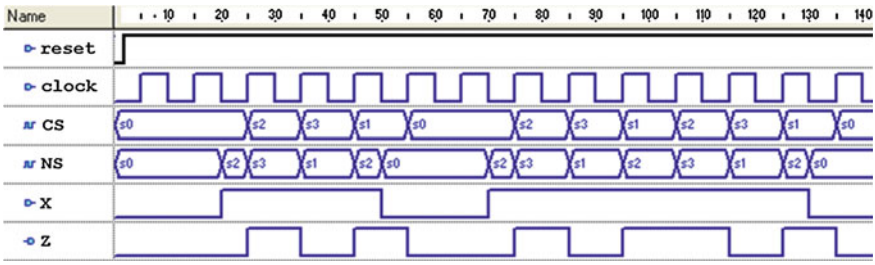


Fig. 7.4 Waveform diagram for the example FSM

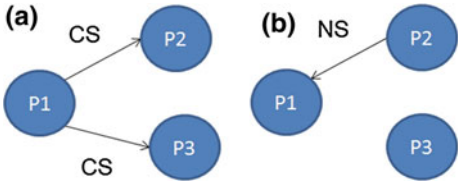


Fig. 7.5 Three processes FSM diagram. **a** P1 process updates the CS; **b** P2 process defines the NS

```
1  P1: process(clk)
2  begin
3     if falling_edge(clk) then
4        if rst = '0' then
5           CS <= S0;
6        else
7           CS <= NS;
8        end if;
9     end if;
10 end process;
11
12 P3: process(clk)
13 begin
14    if rising_edge(clk) then
15       case CS is
16          when S0 =>
17             Z <= '0'
18          when S1 =>
19             Z <= '0'
20          when S2 =>
21             Z <= '1';
22          when others =>
23       end case;
24    end if;
25 end process;
```

```
P2: process(CS, X)
begin
   case CS is
      when S0 =>
         NS <= S1;
      when S1 =>
         if X = '1' then
            NS <= S2;
         else
            NS <= S1;
         end if;
      when S2 =>
          NS <= S1;
      when others =>
   end case;
end process;
```

Fig. 7.6 A VHDL implementation for a three processes FSM

In Fig. 7.6 there is a VHDL implementation for a three processes case study. The P1 process is triggered by the clock falling edge (i.e. clk'event and clk = '0'), performing the state transitions (line 7, CS <= NS). P2 process is triggered by CS and the X input signal. According to the input (X), P2 defines the next state of the FSM, which will be assigned to CS by P1 in the next clock falling edge. The P3 process is triggered by the rising clock edge (i.e. clk'event and clk = '1'), and it is responsible for providing the FSM outputs, according to the (CS).

In the two processes FSM, one of the processes is used to model the state register, defining the next state, and the other process updates the next state and provides the FSM outputs.

In the three processes FSM, one process models the state register defining the next state, a second process updates the next state, and a third process provides the FSM output.

In a single process FSM implementation, all these activities are performed by the same process. Figure 7.7 shows a VHDL implementation for the Fig. 7.1 FSM, using only one process. In this case, when the reset signal is '1', the initial state is set to S0. When the reset signal is '0', and there is a rising clock edge event, then the Z output is provided according to the (CS), and the (NS) is defined according to the X input.

In the two and the three processes approaches, one of the processes provides combinatorial outputs, as there is no clock signal on its sensitivity list. This can be an interesting solution, in case the designer needs to have combinatorial outputs.

```
entity MOORE is
   port(X, clock, reset: in std_logic;
        Z              : out std_logic);
end;

architecture OneProcess of MOORE is
   type STATES is (S0, S1, S2, S3);
   signal CS: STATES;
begin
   process(clock, reset)
   begin
      if reset= '1' then
         CS  <= S0;
      elsif clock'event and clock='1' then
         case CS is
            when S0 => Z <= '0';
                       if X='0' then CS <= S0; else CS <= S2; end if;
            when S1 => Z <= '1';
                       if X='0' then CS <= S0; else CS <= S2; end if;
            when S2 => Z <= '1';
                       if X='0' then CS <= S2; else CS <= S3; end if;
            when S3 => Z <= '0';
                       if X='0' then CS <= S3; else CS <= S1; end if;
         end case;
      end if;
   end process;
end OneProcess;
```

Fig. 7.7 VHDL implementation of an FSM using one process

In the one process implementation, there is no combinatorial output. On the other hand, the one process approach, historically, is a better choice regarding faster simulation times, and also it is easier to debug as there is no combinatorial processes involved. However, designers usually find the two processes approach more readable, as it is an appropriate behavioral description of the FSM classical hardware architecture shown in Fig. 7.2.

7.3 FSM Case Study: Designing a Counter

A counter is a good example of a sequential circuit, and it has interesting features suitable for FSM modeling. A counter has an output that provides expected values for each of its states, according to the input clock pulses. The output values are provided in a sequence that can be ascending, descending, or out of order. In any case, when the last value of the counting is reached, the sequence can be restarted, according to the design requirements. There are several types of counters, and among the most used ones there are the up counters, and the down counters. In Fig. 7.8 there is the VHDL description of a ring counter, performing an up counting.

```
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           Z      : out std_logic_vector(7 downto 0)
8           );
9  end ring_counter;
10
11 architecture beh of ring_counter is
12    signal aux : std_logic_vector(7 downto 0);
13 begin
14    process(CLK, RST)
15    begin
16       if (RST = '0') then
17             aux <= "00000001";
18       elsif (CLK'event and CLK = '1') then
19          aux(7 downto 1) <= aux(6 downto 0);
20          aux(0) <= aux(7);
21       end if;
22    end process;
23    Z <= aux;
24 end beh;
```

Fig. 7.8 A ring counter described in VHDL

The ring counter is initialized on line 17, and at each rising clock edge, the counter value is rotated to the left. On line 20, the most significant bit is copied to the less significant bit position, closing the "ring".

The simulation results for this counter are shown in Fig. 7.9. After the reset (RST) signal goes up, the ring counter is load with the start value "00000001". The bit in the less significant position holds a '1', which is rotated passing through all the eight bit positions, and going back to the original position after eight clock pulses. The pattern provided on the Z output by this ring counter is also known as the "one hot" code. As shown in Fig. 7.9, at each clock pulse only one of the eight output bits is '1' ("one hot"). This is a circular up counter as the provided Z outputs, in decimal notation, are: 1, 2, 4, 8, 16, 32, 64, 128, 1, 2, 4, ….

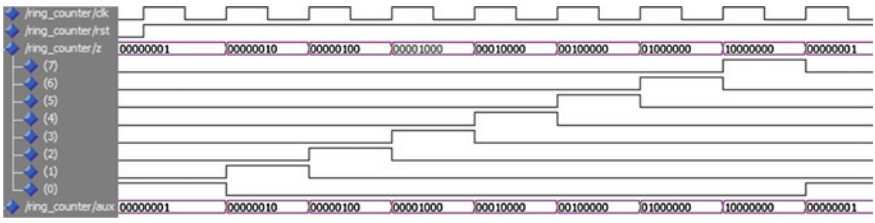


Fig. 7.9 Simulation results for the ring counter

Another example of counter is shown in Fig. 7.10. In this example, a counter is used to generate a delay. This sort of circuit is very useful, for instance, in applications where data need to be shown in a display. In Fig. 6.14, the calculation result stored in the registers is shown in the 7-segment displays each time the KEY(1) push-button (Enter) is pressed. A similar application may require the stored data to be shown on the displays periodically as, for example, every one second.

The FSM in Fig. 7.10 is used to generate a 1 s delay, considering an input clock of 27 MHz. The S1 state is included just to indicate the place where the 7-segment code could be inserted. Usually, after the data are sent to the display (in S1, S2, …, states), the FSM flow is diverged to the D1 state, which is the initial state of the delay routine. In this state, the delay counter is set to an initial value (zero in this example). Next, in the following rising clock edge, the counter (delay) is incremented (D2 state). In the next state (D3), there is a test inorder to check if the final value for the counting has been reached. If the test results in true, then 1 s has passed, and the FSM diverges back to the application (S1 state). Otherwise, if the test results in false, the FSM needs to keep counting on each rising clock edge, and the FSM goes to the D4 state. In this example, the final test value was calculated considering the 27 MHz input clock (i.e. 27 million pulses per second), and the FSM has to count 8,388,608 times in order to waste, approximately 1/3 s of time. As shown in Fig. 7.10, the FSM needs three clock pulses to perform a complete round, i.e. increment in D2, test in D3, and do nothing in D4. The D4 state has been included in this FSM with the only goal of spending one extra clock pulse. Therefore, 3 clock pulses times 1/3 s is equal to the required 1 s delay.

```
process(CLK, RST)       -- Considering a 27MHz clock, the process is triggered
begin                   -- 27 million times per second.
   if RST = '0' then
     CS  <= S0;
   elsif CLK'event and CLK = '1' then
     case CS is
       when S1 =>   ...              -- Do something (e.g. write to LCD, ...).
       when D1 =>                    -- This state starts the delay counting (1s).
         delay <= ( others => '0' );
         CS <= D2;
       when D2 =>                    -- State that generates the chosen delay
         delay <= delay + 1;         -- "delay" was set to zero in D1.
         CS <= D3;
       when D3 =>              -- Tests if the final counting has been reached
         if delay >= x"800000" then  -- 8,388,608 / 27,000,000 = 0.3 * 3 = 1 s.
           CS <= S1;  -- When the upper limit is reached, leaves the delay
         else         -- loop, and goes back to the application.
           CS <= D4;  -- Remains in the counting loop to generate the delay.
         end if;
       when D4 =>     -- Continues the delay counting.
         CS <= D2;
     end case;
   end if;
end process;
```

Fig. 7.10 A counter used for delay generation

7.4 Laboratory Assignment

In previous chapters, several components have been glued together in order to build a basic calculator. In the next chapters, an FSM will be used in order to control the calculator operations.

In this chapter, a counter is used as a case study. Counters represent an important type of digital circuits, and FSMs can be used to model their behavior and functionality.

The laboratory objectives are:

- to design an FSM based digital circuit;
- to design a counter in VHDL.

7.4.1 Laboratory Session

The tasks to be performed in this laboratory session are as follows:

- Design and implement in VHDL a one process FSM, for the generation of the ASCII characters 'A' to 'Z'.
- The characters should be presented on the green LEDs (binary format) and on two 7-segments displays (hexadecimal format), as shown in Fig. 7.11.

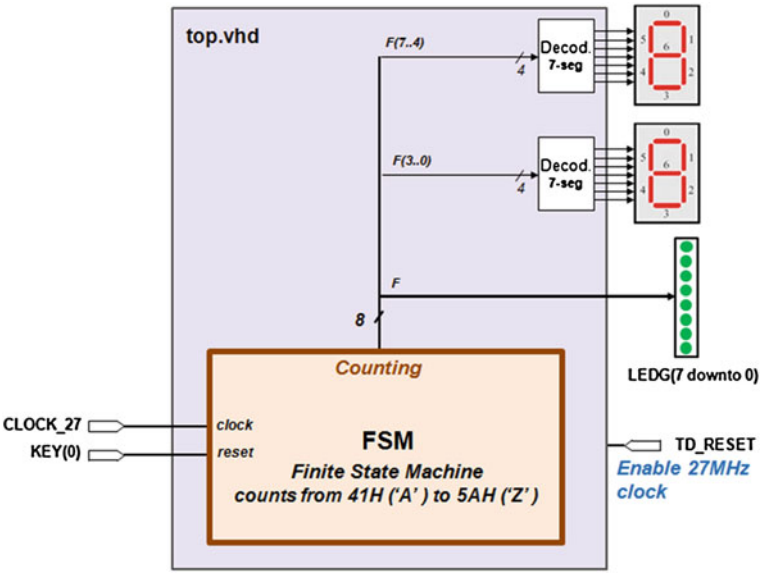


Fig. 7.11 'A' to 'Z' ASCII counter block diagram

- To write in the 7-segment displays, it is necessary to use two instances of the *decod7seg.vhd* component, described in Chap. 5 (see Fig. 5.22).
- The FSM has an asynchronous reset, KEY(0) button, used to initialize a counter with the first character in the sequente ('A' = 41H).
- On the rising clock edge (27 MHz), the counter should be incremented, generating the next ASCII table character ('B', 'C', 'D', …).
- The FSM should have a small number of states, but sufficient to increment the counter and check whether the final counting ('Z' = 5AH) has been reached.
- When the final counting is reached, the FSM should be reset to the initial state (beginning of the counting).
- In order to activate the DE2 27 MHz clock signal, the TD_RESET output has to be set to '1'.
- In Fig. 7.12 it is shown a suggestion of VHDL description for the top component, without the 7-segment decoders. In order to include the 7-segment decoders, create an F signal in the top file to connect the FSM component to the 7-segment decoders and the LEDG output.

Note Alternatively, instead of the 27 MHz clock, the rising clock edge can be generated by a push button (KEY). Warning! Beware of the debounce problem, since a single press in a mechanical switch may result in multiple presses in a digital circuit. A clock signal is a much more precise way to control the counting sequence.

In this laboratory session, all the usual activities should be performed: project creation; VHDL synthesis; VHDL simulation; and FPGA prototyping.

For the FPGA prototyping, it is important to include in the FSM some sort of the delay procedure as, for instance, the one shown in Fig. 7.10. Without a delay routine, the user will not be able to watch the counting presented on the LEDs and displays. However, for the simulation, the delay routine should be set to a smaller final counting value, otherwise the process would take too long. For instance, a 20 s simulation time in an i7 quad core processor (hyper threading, so 8 cores),

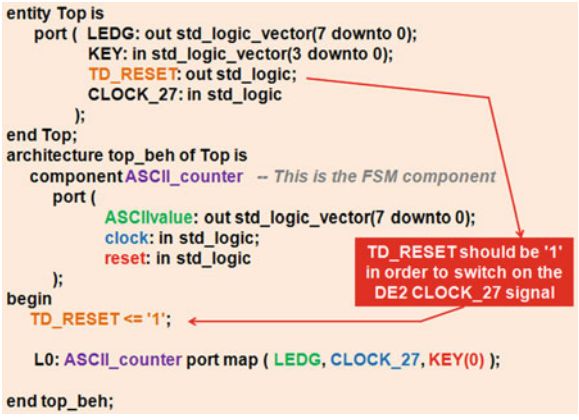


Fig. 7.12 Suggestion of top file for the ASCII counter design without the 7-segment decoders

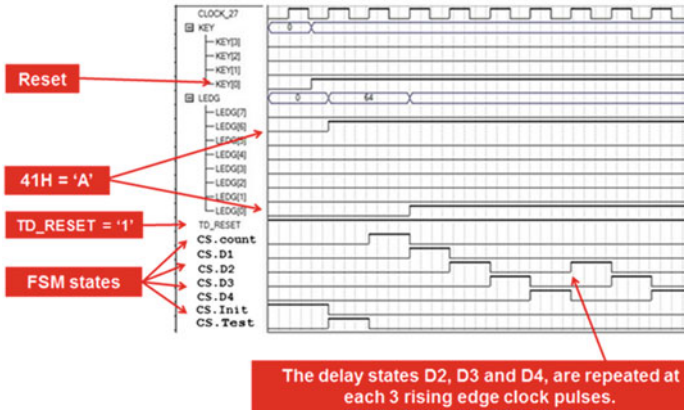


Fig. 7.13 Simulation results showing the delay states of the FSM

running at 2.93 GHz and with 8 GB of RAM, took 19 h and 28 min to be performed.

Figure 7.13 shows the ASCII counter simulation waveform. In the CS.Init state, after the KEY(0) button is released, the initial counting value is set to 41H, which is the 'A' character in the ASCII table (see LEDG). In the CS.Test state, the FSM checks if the upper counting limit 5AH ('Z') has been reached. Next, the delay states are performed, starting by the CS.D1 state, where the delay counting signal is initialized. The CS.D2, CS.D3, and CS.D4 states are performed 8,388,608, as discussed before in Fig. 7.10.

Chapter 8
Using Finite State Machines as Controllers

In the previous chapter, FSMs have been used in the design of counters. This is a good example of FSM usage, where each state of the FSM provides as output a counting value. The present chapter introduces the design of FSMs targeting one of their main applications, which is the control of a sequence of events. At the end of the chapter, the reader should be able:

- to understand the basic concepts of datapath and control unit;
- to design the FSM of a control unit;
- to implement a control module for the case study (calculator);
- to simulate and test the case study—calculator with control module;
- to prototype the case study using an FPGA board.

8.1 Designing an FSM Based Control Unit

As discussed in Chap. 1, and shown in Fig. 1.1, digital systems in general have a module responsible for performing the processing, and another module used to control the sequence of operations. Figure 8.1 shows three examples of typical digital systems where there is a control unit and a datapath. In each design there is a control unit sending a sequence of "commands" to the operational module.

In Fig. 8.1, FSMs can be used in the design of the control circuits. For instance, the Vending Machine controller has to perform all the necessary steps in order to deliver (or not) the selected product to the user. The controller's FSM should be designed considering that the operational part of the system has to perform the activities in a sequential order (i.e. payment selection; product selection; delivery product; …). The FSMs for the remaining controllers in Fig. 8.1 should be designed following similar principles, always considering a very specific sequence of events to be performed.

Strategies for describing FSMs in VHDL have been discussed in Chap. 7. However, before writing the VHDL implementation for an FSM, the target circuit

E. A. Bezerra and D. V. Lettnin, *Synthesizable VHDL Design for FPGAs*,
DOI: 10.1007/978-3-319-02547-6_8,

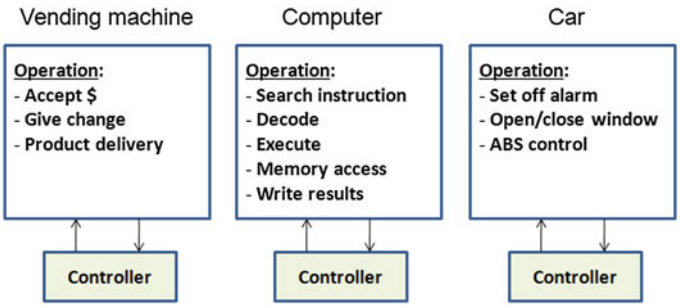


Fig. 8.1 Simulation results showing the delay states of the FSM

has to be properly designed. In Chap. 7, the FSM for the proposed counter case study has been written in VHDL in an empirical way, without following any design methodology.

In the present chapter, a controller for a new case study is implemented in VHDL, but this time a proper design methodology is followed.

The methodology adopted for the design of a controller in VHDL has the following steps:

STEP 1 Write a system description as complete as possible (requirements);

STEP 2 Prepare a graphical representation for the FSM, making as many versions as needed;

STEP 3 Write down a state transition table for the FSM, listing the inputs and outputs, including all the states (current and next);

STEP 4 Describe in VHDL the modeled FSM behavior, using the graph and the truth table as a guide, and following the examples provided in Chap. 7.

In the classical FSM controller design methodology used in digital systems in general (not in HDL based designs), the following additional steps are required:

STEP 5 Define the target circuit architecture which is, usually, represented by the block diagram in Fig. 1.3;

STEP 6 Define a unique identifier (coding strategy) for each state;

STEP 7 Make the necessary changes in the truth table, in order to have only binary numbers (replace the state symbolic identifiers by the binary codes);

STEP 8 From the truth table, obtain the Boolean equations, and design the circuit for the combinational component of Fig. 1.3.

It is important to notice that these additional steps are not used in a VHDL based design, as they are automatically performed by the synthesis tool. The synthesis tool not only extracts the required combinational logic from the VHDL FSM description, but also performs very efficient optimizations.

Next, a case study is presented and implemented in VHDL in order to demonstrate the FSM design methodology. The additional steps discussed are also performed for the case study, demonstrating the classical approach for the design of a FSM based digital system controller.

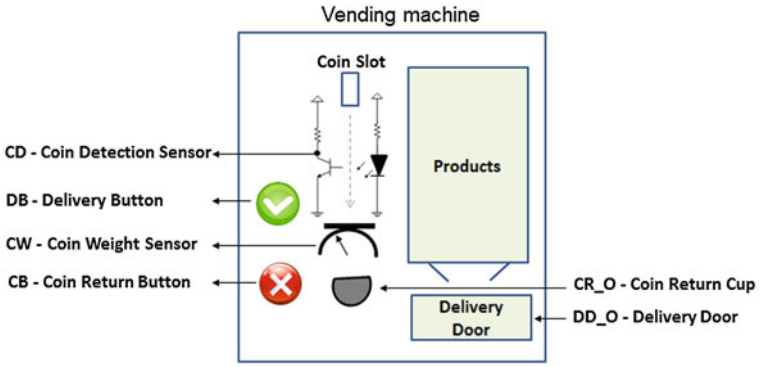


Fig. 8.2 Block diagram of the vending machine

8.2 Case Study: Designing a Vending Machine Controller

In this case study you are asked to implement a digital controller for the vending machine shown in Fig. 8.2.

STEP 1 Textual description.

The vending machine has the following features (textual description):

- The vending machine sells just one type of product, and each product costs $0.25;
- Only $0.25 coins are accepted;
- A valid $0.25 coin weights 5.670 g;
- Invalid coins are rejected automatically, and the control circuit does not need to do anything about that;
- There is a slot to insert $0.25 coins in the vending machine;
- There is a coin detection circuit that outputs '1' every time a $0.25 coin is inserted in the slot, or '0' otherwise;
- There is a digital scale that outputs '1' when a weight of 5.670 g (or more) is detected;
- There is a "delivery button" that outputs '1' every time it is pushed;
- There is a "coin return button" that outputs '1' every time it is pushed;
- There is a "coin return cup" to collect coins ejected by the "coin return button" (and also rejected coins);
- There is a door where the user can collect their product, after have pushed the "delivery button";
- When the vending machine runs out of products, it shuts itself, and the control circuit does not need to do anything about that.

The textual description together with the block diagram, comprises the Step 1 of the controller design methodology introduced in the previous section.

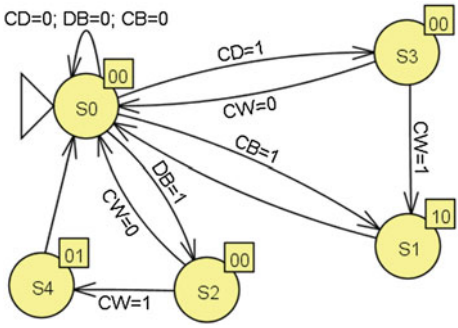


Fig. 8.3 FSM graph for the vending machine controller

STEP 2 FSM graphical representation.

In Step 2, a graphical representation for the FSM is built, and shown in Fig. 8.3. As shown in Fig. 8.2, the following list of abbreviations has been adopted for the design:

CD	Coin Detection Sensor (input)
DB	Delivery Button (input)
CW	Coin Weight Sensor (input)
CB	Coin Return Button (input)
CS	Current State (input)
NS	Next State (output)
CR_O	Coin Return Cup (output)
DD_O	Delivery Door (output)

In the graph diagram shown in Fig. 8.3, it is possible to observe that:

- S0 is the initial state.
- The FSM remains in S0 waiting for a coin be inserted in the slot (CD = 1), or the delivery button is pushed (DB = 1), or the coin return button is pushed (CB = 1).
- The output signals are represented by the two bits inside the small box on the right top of each state. The most significant bit is the "coin return" (CR_O) output, and the less significant bit is the "delivery door" (DD_O) output.
- The "coin return" (CR_O) and the "delivery door" (DD_O) outputs are equal to 0 in all states but in S1 and S4. In S1, CR_O = 1, which means that a $0.25 coin is returned. In S4, DD_O = 1, in order to deliver a product to the customer.
- When a coin is inserted in the slot, the FSM goes to S3 state. If there was a coin already in the vending machine, the next state is S1, where the coin is returned to the customer (CR_O = 1). Otherwise, the $0.25 coin is placed in the digital scale, and the FSM goes back to S0.
- When the delivery button is pushed, in the S2 state, the FSM goes to the S4 state to deliver a product only if the coin weight sensor indicates that there is a valid payment in the vending machine (CW = 1), otherwise, the FSM goes back to S0 state (CW = 0).

Fig. 8.4 State transition table for the vending machine

Inputs					Outputs		
CD	DB	CW	CB	CS	NS	CR_O	SD_O
0	0	X	0	S0	S0	0	0
0	0	X	1	S0	S1	0	0
0	1	X	0	S0	S2	0	0
1	0	X	0	S0	S3	0	0
X	X	X	X	S1	S0	1	0
X	X	0	X	S2	S0	0	0
X	X	1	X	S2	S4	0	0
X	X	0	X	S3	S0	0	0
X	X	1	X	S3	S1	0	0
X	X	X	X	S4	S0	0	1

STEP 3 State transition table.

In Step 3, a state transition table is built according to the FSM diagram. This sort of table is very similar to a truth table, and it has all the circuit's inputs and outputs. The main difference is the information regarding the "current state" and the "next state", even though they are also inputs and outputs for the circuit. Figure 8.4 shows the state transition table for the vending machine, where there are several "don't care" inputs. The table has been built considering that, for these inputs, the circuit has the same behavior when the respective input is high or low. For instance, on the first line of the table, it does not make any difference in the outputs if the CW input is '1' or '0'. This means that disregarding the CW input value, the outputs for this line will be NS = S0, CR_O = '0' and DD_O = '0'. The don't care inputs are very important in order to reduce the number of table lines.

STEP 4 VHDL description.

In this step, the FSM behavior is described in VHDL, as shown in Fig. 8.5. The two processes approach shown in Fig. 7.3 has been used in this implementation. The first process is used to model the state register, defining the next state, and the second process provides the controller outputs, and generates the next state, according to the input values.

IMPORTANT!!! Steps 5 through 8 are not used in VHDL based designs! They are implemented by synthesis tools, which are carefully designed in order to generate the best circuit according to the VHDL description provided in Step 4.

Steps 5 through 8 are included next just as an example of how this type of controller circuit can be manually implemented using logic gates and a register, with no synthesis tool hardware inference and automatic circuit optimizations.

As stated before, steps 1–4 are used by VHDL developers in the design of FSM based controllers, while steps 5–8 are used by synthesis tool developers. A synthesis tool developer is responsible for implementing all the algorithms used to obtain the most appropriate coding strategy for the FSM sates, the best optimization for the combinational circuit (Boolean equations), among other performance enhancement features.

```
entity VendingMachine is
    port(clock, reset  : in std_logic;
         CD, DB, CW, CB: in std_logic;
         CR_O, DD_O    : out std_logic);
end;

architecture behv of VendingMachine is
    type STATES is (S0, S1, S2, S3, S4);
    signal CS, NS : STATES;
begin
    process (clock, reset)
    begin
        if reset= '0' then
            CS <= S0;
        elsif rising_edge(clock) then
            CS <= NS;
        end if;
    end process;

    process(CS, CD, DB, CW, CB)
    begin
        case CS is
            when S0 =>  CR_O <= '0'; DD_O <= '0';     -- wait for an event
                        if CB='1' then NS <= S1; elsif DB='1' then NS <= S2;
                        elsif CD='1' then NS <= S3; else NS <= S0;
            when S1 =>  CR_O <= '1'; DD_O <= '0';     -- return a $0.25 coin
                        NS <= S0;
            when S2 =>  CR_O <= '0'; DD_O <= '0';     -- Is there a payment?
                        if CW='0' then NS <= S0; else NS <= S4;
            when S3 =>  CR_O <= '0'; DD_O <= '0';     -- Is it a valid coin?
                        if CW='0' then NS <= S0; else NS <= S1;
            when S4 =>  CR_O <= '0'; DD_O <= '1';     -- Product delivery
                        NS <= S0;
        end case;
    end process;
end behv;
```

Fig. 8.5 The vending machine controller described in VHDL

STEP 5 FSM controller architecture.

The block diagram of the proposed controller is shown in Fig. 8.6. All the input signals, including the "next state" are located on the left side of the diagram. The two output signals, and the "current state" information are located on the right side of the diagram. The register used to store the current state value is located in the bottom of the diagram. The combinational circuit is responsible for defining the outputs (DD_O and CR_O) and the next state, which will be stored as the FSM's current state (CS). The combinational circuit design will be provided in Step 8.

STEP 6 States coding.

During the synthesis process, all labels used to identify the FSM states are replaced by a binary code according to the available resources in the target hardware. For instance, FPGAs typically have a large amount of D flip-flops available and, in this case, the synthesis tools usually choose the "one-hot" coding strategy to identify the FSM states. In this strategy, for a group of bits that identify a state, only one bit is 1, and all the remaining are 0. The one-hot strategy uses

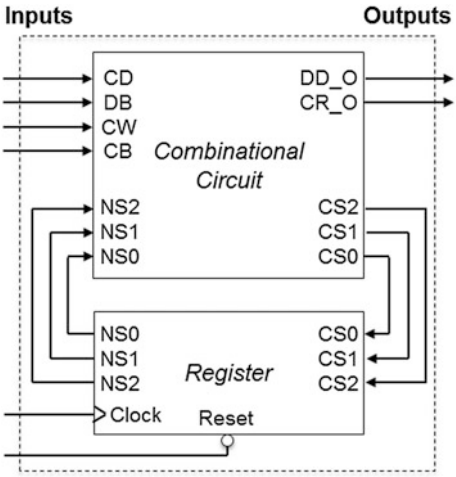


Fig. 8.6 Block diagram of the architecture for the vending machine controller

One-hot	Binary	Gray code
00000001	000	000
00000010	001	001
00000100	010	011
00001000	011	010
00010000	100	110
00100000	101	111
01000000	110	101
10000000	111	100

Fig. 8.7 Coding strategies for FSM states

more storage resources, but on the other hand it does not need any special decoder circuits. An FSM's state is identified in the group of bits that represent a state, according to the position of the bit '1'. For instance, considering an FSM with four states, S0, S1, S2, and S3, in the one-hot strategy, these states are coded, respectively, as 0001, 0010, 0100, and 1000. Other options for coding the states of an FSM include the binary and the gray code. Figure 8.7 shows the first eight codes for each one of these strategies.

STEP 7 Edit the state transition table, and replace the symbolic identifiers by their binary coding representation (Fig. 8.8).

STEP 8 Obtain the Boolean equations from the state transition table.

Using the sum of products method, the following Boolean equations are obtained, which can be further optimized using methods as Boolean equations simplification, Karnahgk maps (in cases where there are up to five variables), or the Quine–McCluskey algorithm (prime implicants method):

Fig. 8.8 State transition table, using binary codes instead of symbolic identifiers

Inputs					Outputs		
CD	DB	CW	CB	$CS_{2..0}$	$NS_{2..0}$	CR_O	SD_O
0	0	X	0	000	000	0	0
0	0	X	1	000	001	0	0
0	1	X	0	000	010	0	0
1	0	X	0	000	011	0	0
X	X	X	X	001	000	1	0
X	X	0	X	010	000	0	0
X	X	1	X	010	100	0	0
X	X	0	X	011	000	0	0
X	X	1	X	011	001	0	0
X	X	X	X	100	000	0	1

```
DD_O = CS2 CS1' CS0'
CR_O = CS2' CS1' CS0
NS0 = CS2' CS1' CS0' CD' DB' CB +
      CS2' CS1' CS0' CD DB' CB' + CS2' CS1 CS0 CW
NS1 = CS2' CS1' CS0' CD' DB CB' +
      CS2' CS1' CS0' CD DB' CB'
NS2 = CS2' CS1 CS0' CW
```

Figure 8.9 shows the block diagram of the vending machine connected to proposed FSM based controller.

In Fig. 8.9, the Boolean equations should be placed in the "Combinational Circuit" box, and the register box is implemented using three D flip-flops. The controller inputs CB, CW, DB and CD are provided by the vending machine hardware. The controller outputs DD_O and CR_O are used to command the vending machine product delivery, and coin return.

8.3 Laboratory Assignment

The laboratory objectives are:

- to understand the use of FSMs as a control module for the operations flow of a combinational circuit;
- to design a VHDL based FSM to be used as a controller for the calculator implemented in previous chapters.

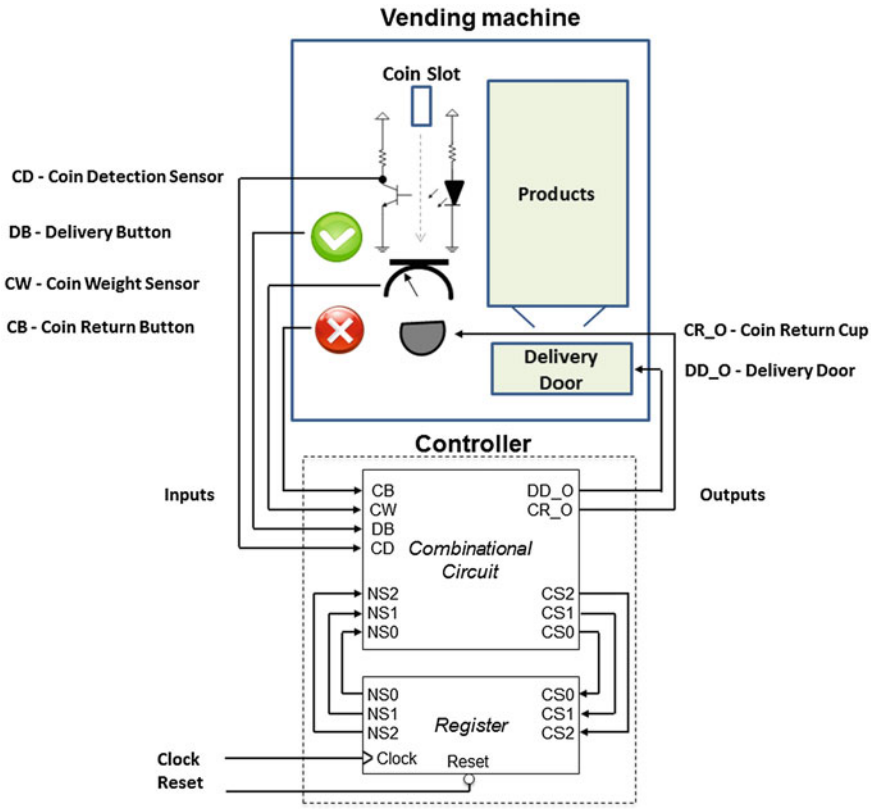


Fig. 8.9 Block diagram of the designed FSM controller, showing the input and output connections

8.3.1 Problem Definition: Calculator with Reduced Data Input Signals

The calculator designed in previous chapter has the following inputs:

- $SW_{7..0}$ switches for operand A; and
- $SW_{15..8}$ switches for operand B.

To decrease the amount of switches in the board layout, in this laboratory session the VHDL designer will have to use just one set of switches, $SW_{7..0}$, to input both operands A and B. In this new version, switches $SW_{17..16}$ are still used for the selection of the chosen operation. Figure 8.10 shows how the data input is performed in the DE2 board, considering the two calculator versions.

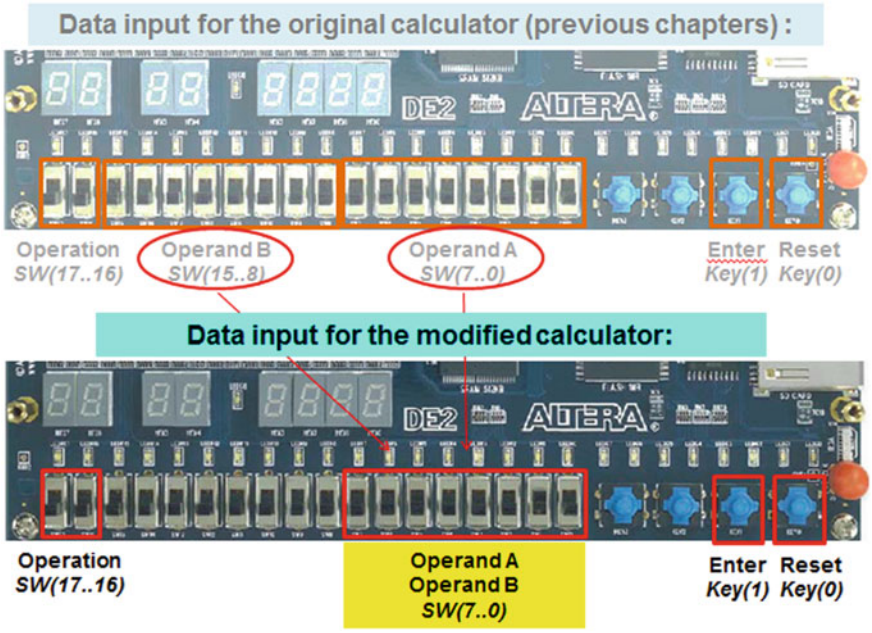


Fig. 8.10 Data input for the calculator: previous and new version

8.3.2 Laboratory Session

The tasks to be performed in this laboratory session are as follows:

- Add an input register to the calculator to hold the first operand;
- Design an FSM to control the sequence of operations in the new calculator;
- Add the FSM to the calculator design, simulate, and test the design in the FPGA board.

8.3.3 Adding a New Input Register to the Calculator Design

In Fig. 8.11, a new register is added to the calculator design. The register is located in the upper left hand corner, and it is used to store the first operand for operations that employ two operands.

Some considerations:

- There is no need to make changes in the calculator components used in previous chapters;
- In the "top_calc.vhd" file, the new register is added just by including a new *port map* VHDL statement in the architecture. Notice that there are already two

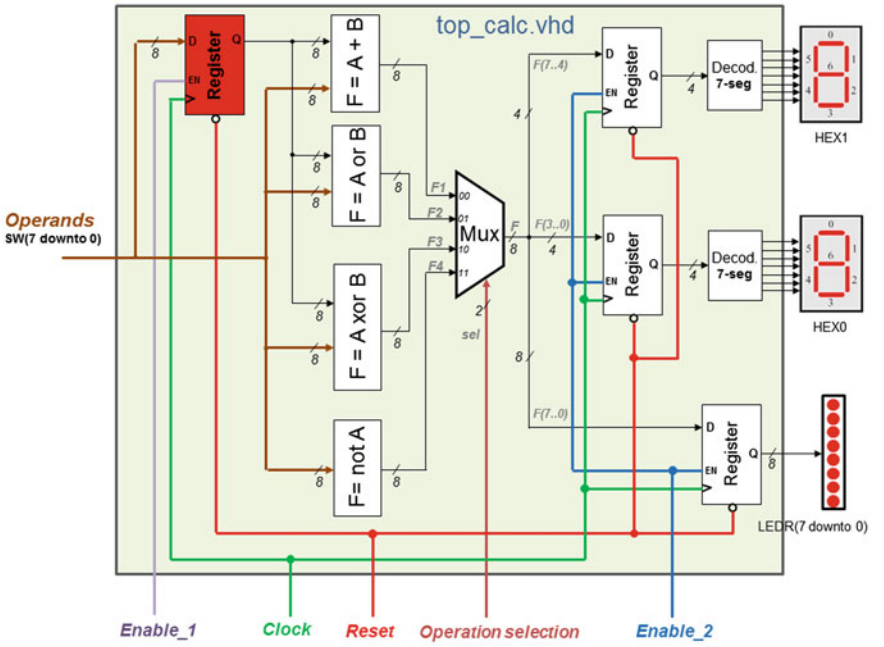


Fig. 8.11 Inserting an input register to hold the first operand

register components declared in the architecture, a four bits and an eight bits one;

- Some of the port map declarations should be changed in order to connect the new register to the remaining components, and also to the inputs;
- Another important remark is that all register enable signals, the multiplexor selector, among others, have been disconnected, as a controller FSM will be added to control the circuit operation sequence.

Next, the FSM controller is partially designed, according to the methodology described before. In Step 1, a textual description is provided for the calculator controller. In Step 2, a graphical representation for the FSM is built, according to the textual description discussed in Step 1. The proposed graphical representation is incomplete, as there are some states where all the possible outputs are not provided. Several graphical representations should be made, until one of the versions can be considered the "final" one. Step 3 is not used in this design, as the VHDL description can be extracted from the graphical view. In Step 4, a partial VHDL description for the FSM is provided. The reader should use the information in steps 3 and 4, in order to complete calculator's controller design.

8.3.4 Designing an FSM Based Controller for the Calculator

The design methodology described before is used in the calculator FSM design. In this case, the four steps are as follows:

STEP 1 Describing the calculator's sequence of operations and control flow. To operate the calculator the user should:

1. Select the chosen operation using switches $SW_{17..16}$;
2. Provide operand A using switches $SW_{7..0}$;
3. Press *Enter*—KEY_1 button is '0' when pressed;
4. For the *"not A"* operation, the result is shown on the 7-segments displays and LEDs;
5. For the remaining operations (*add*, *xor* and *or*), provide the second operand using switches $SW_{7..0}$, and press *Enter* again;
6. After the result is presented, this sequence is restarted from step 1.

Figure 8.12 shows the block diagram of the single input calculator, with the proposed FSM controller.

As can be observed from Fig. 8.12, the new top_calc.vhd file has 12 inferred components (using port map). These 12 components are inferred from nine components described in VHDL, and saved in the respective files as listed next:

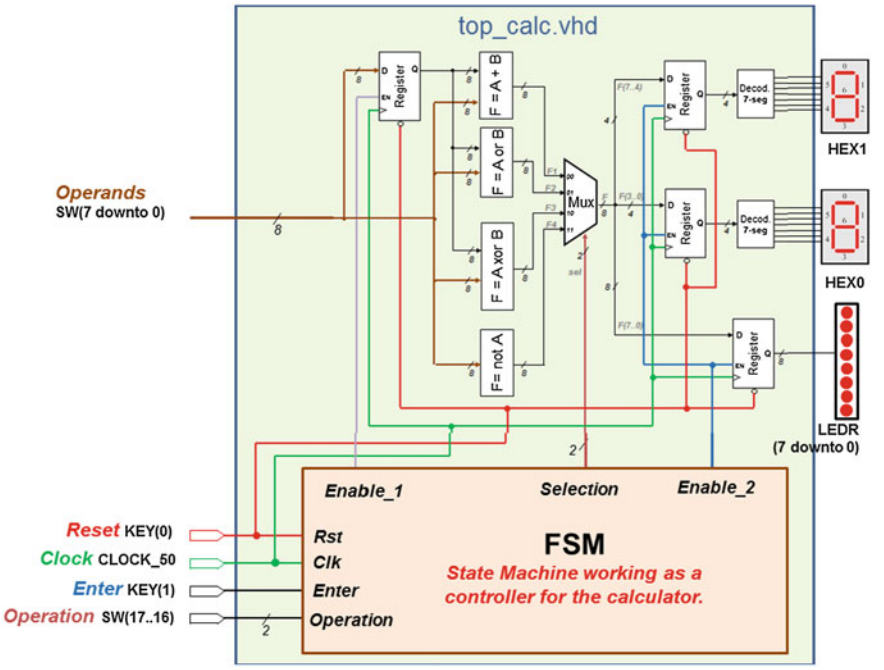


Fig. 8.12 Adding an FSM to control the calculator sequence of operations

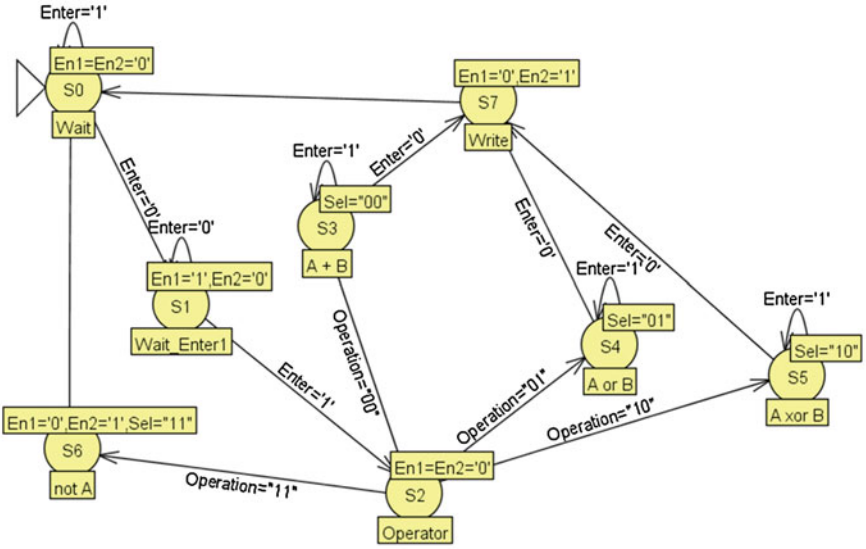


Fig. 8.13 FSM controller graphical view—incomplete version

Inputs				Outputs			
Reset	Enter	Operation	CS	NS	Enable_1	Selection	Enable_2
0	X	XX	S0	S0	0	00	0
1	1	XX	S0	S0	0	00	0
1	0	XX	S0	S1	0	00	0
1	0	XX	S1	S1	1	00	0
1	1	XX	S1	S2	1	00	0
1	X	00	S2	S3	0	00	0
1	X	01	S2	S4	0	00	0
1	X	10	S2	S5	0	00	0
1	X	11	S2	S6	0	00	0
1	0	XX	S3	S7	0	00	0
1	1	XX	S3	S3	0	00	0
1	0	XX	S4	S7	0	01	0
1	1	XX	S4	S4	0	01	0
1	0	XX	S5	S7	0	10	0
1	1	XX	S5	S5	0	10	0
1	X	XX	S6	S0	0	11	1
1	X	XX	S7	S0	0	00	1

Fig. 8.14 State transition table for the proposed FSM

```
1  library  ieee;
2  use  ieee.std_logic_1164.all;
3  entity FSMctrl is
4    port ( Clk, Rst, Enter : in std_logic;
5           Operation: in std_logic_vector(1 downto 0);
6           Selection: out std_logic_vector(1 downto 0);
7           Enable_1, Enable_2: out std_logic);
8  end FSMctrl;
9  architecture FSM_beh of FSMctrl is
10   type states is (S0, S1, S2, S3, S4, S5, S6);
11   signal CS, NS: states;
12   signal clock, reset: std_logic;
13 begin
14   clock <= Clk;    reset <= Rst;
15 process (clock, reset)
16 begin
17   if reset = '0' then
18     EA <= S0;
19   elsif clock'event and clock = '1' then
20     EA <= PE;
21   end if;
22 end process;
23 process (EA, Enter)
24 begin
25   case EA is
26     when S0 =>
27       if Enter = '1' then PE <= S0; else PE <= S1;
28       end if;
29       Enable_1 <= '0'; Enable_2 <= '0';
30     when S1 =>      -- ... to be done
31     when S2 =>      -- Operator
32       Enable_1 <= '0'; Enable_2 <= '0';
33       if Operation = "00" then
34         PE <= S3;  -- add
35       elsif Operation = "01" then
36         PE <= S4;  -- OR
37       elsif        -- ... to be done
38    end case;
39 end process;
40 end FSM_beh;
```

Fig. 8.15 Partial VHDL description for the proposed FSM

- 2 × 8 bits registers—file *reg8bits.vhd*;
- 2 × 4 bits registers—file *reg4bits.vhd*;
- 1 × adder component—file *c1.vhd*;
- 1 × OR component—file *c2.vhd*;
- 1 × XOR component—file *c3.vhd*;

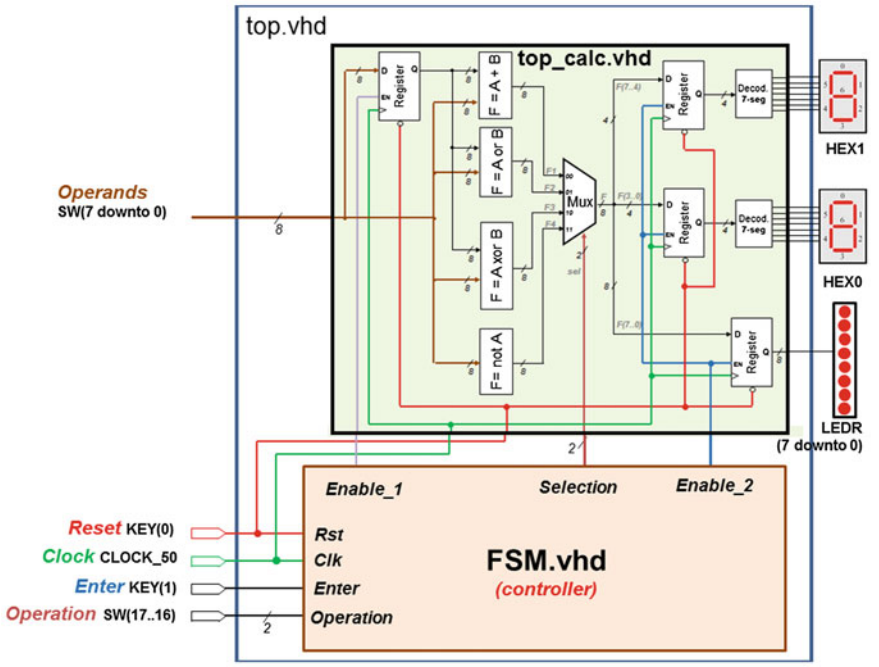


Fig. 8.16 Block diagram for the alternative implementation

- 1 × NOT component—file *c4.vhd*;
- 1 × 4 input multiplexer—file *mux4x1.vhd*;
- 2 × 7-segment decoders—file *decod7seg.vhd*;
- 1 × FSM controller—file *FSMctrl.vhd*.

STEP 2 FSM graphical representation.
The graphical view in Fig. 8.13 was built according to the description in step 1. The FSM has the following functionality:

- In the initial state "Wait", FSM outputs are disabled (*Enable_1* = '0', *Enable_2* = '0'), ensuring that there are no activities in the calculator, while the user does not provide the input data;
- When *Enter* is '0' (Key_1 is pressed), the FSM goes to the next state, and stays waiting until *Enter* goes back to '1' (push button is released);
- In the *Operator* state, according to the operation, there is a transition to the appropriate next state;
- For the "*Not A*" operation (only one operand), the result is stored in the output registers (*Enable_2* = *'1'*) and the FSM goes back to the initial state ("Wait");
- For the remaining operations, the second operator is read in an additional state—*Enable_2* = *'0'* in states *S3*, *S4*, and *S5*, followed by *Enable_2* = *'1'* in the *Write* state. In these operations, when Enter is pressed for the second time

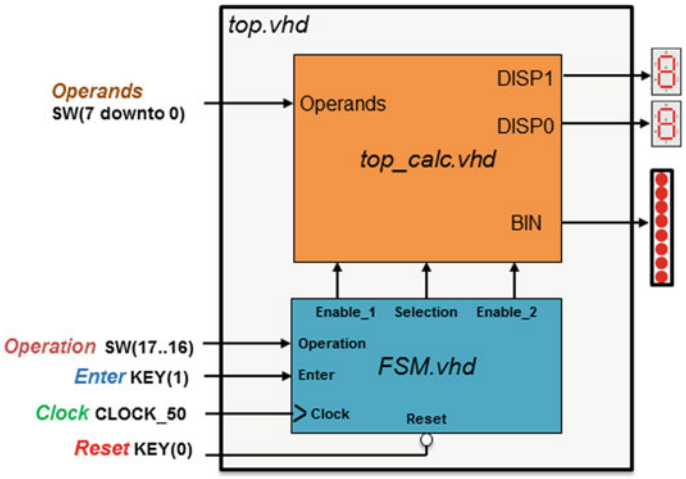


Fig. 8.17 Simplified view of the block diagram for the alternative implementation

(Key_1 = '0'), the result is written in the output registers, and the FSM goes back to the initial state.

STEP 3 FSM state transition table.

The state transition table can be obtained straight from the FSM graphical view, and it can be used to better visualize the inputs and outputs in each state (Fig. 8.14).

STEP 4 VHDL description for the modeled FSM behavior (Fig. 8.15).

Another option is to create a new top file, to connect together the FSM and the calculator components. Figure 8.16 shows this alternative implementation, where all calculator components are connected together in a top file called "top_-calc.vhd", and the FSM implementation is in the "FSM.vhd" file. Figure 8.17 shows a simplified view of this alternative implementation.

It is important to notice that in this alternative implementation it may be interesting to rename all signals listed in the "top_calc.vhd" entity (component interface), in order to avoid using DE2 labels SW, LEDR, HEX0, and HEX1. These labels are used in the entity of the new "top.vhd" file. In Fig. 8.17, the labels Operands, BIN, DISP0 and DISP1 were used instead.

Chapter 9
More on Processes and Registers

The behavioral description of sequential circuits in VHDL has been discussed in previous chapters. In Chaps. 7 and 8, the VHDL process statement has been used to describe FSMs. In Chap. 6, the process statement is introduced, and used to describe the behavior of latches, flip-flops and registers. This chapter discusses the use of processes in the implementation of both, combinational and sequential circuits. At the end of the chapter, the reader should be able:

- to understand the difference between implicit and explicit processes;
- to design combinational and sequential circuits using implicit and explicit processes;
- to implement shift registers in VHDL.

9.1 Implicit and Explicit Processes

Implicit and explicit processes have been used so far in VHDL examples all over this book. In implicit processes, the reserved word "process" is not used. Examples of implicit process include:

- Simple signal assignments, where any event in the source side of the assignment will trigger the process as, for instance, *SW*, *A*, *B*, and *C* next:
  - LEDR <= SW;
  - F <= (A and B) or C;
- Signal assignments using selection (*with/select*) and conditional (*when*) statements;
- Instantiated components (*component/portmap*).

Explicit processes are easily identified in a VHDL code by the usage of the reserved word "process". The two VHDL codes in Figs. 9.1, 9.2 when synthesized, will generate the same circuit. The code in the left hand side has three "implicit" processes performing their signal assignment operations in parallel. The

E. A. Bezerra and D. V. Lettnin, *Synthesizable VHDL Design for FPGAs*,
DOI: 10.1007/978-3-319-02547-6_9,

code in the right hand side performs the same functionality as the code on the left, but the signal assignment operations are placed inside three "explicit" processes. As shown in Fig. 9.3, both implicit and explicit processes implementations listed in Fig. 9.1 generate their Y output in parallel to the E and F assignments.

The VHDL code in Fig. 9.2 has the same three assignments as the codes in Fig. 9.1, but the circuit functionality is slightly different. As shown in Fig. 9.4, as the assignments appear in the same process, the Y output is not updated simultaneously to the E and F assignments. The Y output is updated next time the process is triggered and, consequently, it provides previous values of E and F. In Fig. 9.4, when the simulation starts, A = B = C = D = '0', and the Y output is undefined. Next, when A = B = '1', the Y output is '0', representing the previous circuit output when all input signals were '0'.

In Fig. 9.5, a multiplexer with priority is implemented using implicit (left hand side) and explicit (right hand side) processes. In VHDL, the IF statement can be used only in explicit processes. The synthesized multiplexer has a priority, as when *sel(0)* is '0', the *F* output is assigned the value in the *A* input, no matters the *sel(1)* value. The *C* input has the lowest assignment priority.

In Fig. 9.6, the selection mechanism is used in order to implement a no priority multiplexer. VHDL implicit processes use the *with/select/when* construction to synthesize no priority multiplexers. In explicit process, this mechanism is synthesized by using the *case/when* construction. In Fig. 9.6, as well as in Fig. 9.5, both the implicit and the explicit process implementation, generate the same piece of hardware.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity Implicit is
4  port (
5          A, B, C, D: in std_logic;
6          Y         : out std_logic
7  );
8  end Implicit;
9  architecture Arch of Implicit is
10    signal E, F: std_logic;
11 begin
12         E <= A and B;
13         F <= C and D;
14         Y <= E or F;
15 end Arch;
16
17
18
19
20
21
22
23
24
```

```
library ieee;
use ieee.std_logic_1164.all;
entity Explicit_v1 is
port (
        A, B, C, D : in std_logic;
        Y          : out std_logic
);
end Explicit_v1;
architecture Arch of Explicit_v1 is
    signal E, F: std_logic;
begin
    process (A, B)
    begin
        E <= A and B;
    end process;
    process (C, D)
    begin
        F <= C and D;
    end process;
    process (E, F)
    begin
        Y <= E or F;
    end process;
end Arch;
```

Fig. 9.1 Implicit and explicit processes in VHDL

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity Explicit_v2 is
4  port (
5          A, B, C, D : in std_logic;
6          Y    : out std_logic
7  );
8  end Explicit_v2;
9  architecture Arch of Explicit_v2 is
10    signal E, F: std_logic;
11 begin
12    process (A, B, C, D)
13    begin
14        E <= A and B;
15        F <= C and D;
16        Y <= E or F;
17    end process;
18 end Arch;
```

Fig. 9.2 Sequential signal assignments in an explicit process

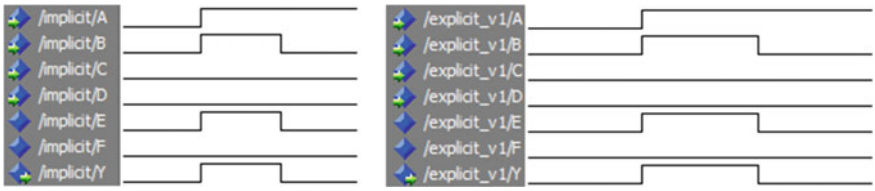


Fig. 9.3 Simulation waveforms for the VHDL codes listed in Fig. 9.1, showing their equivalence

Fig. 9.4 Simulation waveform for the VHDL code listed in Fig. 9.2

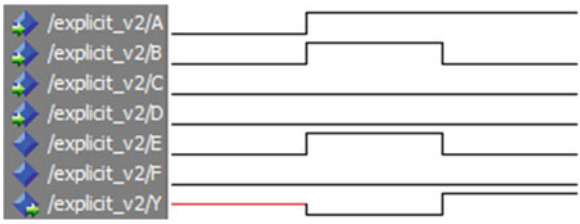
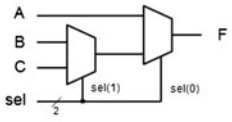


Fig. 9.5 Conditional assignment with priority using the "comparison" mechanism, implemented with implicit and explicit processes

```
architecture implicit of y is
begin

   F <= A when sel(0) = '0' else
        B when sel(1) = '0' else
        C;

end implicit;
```



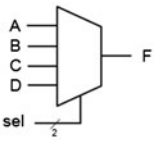

```
architecture explicit of y is
begin

  process (A, B, C, sel)
  begin
     if sel(0) = '0' then
       F <= A;
     elsif sel(1) = '0' then
       F <= B;
     else
       F <= C;
     end if;
   end process;

end explicit;
```

Fig. 9.6 Conditional assignment with no priority using the "selection" mechanism, implemented with implicit and explicit processes

```
architecture implicit of y is
begin
   with sel select
   F <= A when "00",
        B when "01",
        C when "10",
        D when others;
end implicit;
```




```
architecture explicit of y is
begin
 process (sel, A, B, C, D)
 begin
     case sel is
         when "00" =>
                 F <= A;
         when "01" =>
                 F <= B;
         when "10" =>
                 F <= C;
         when others =>
                 F <= D;
     end case;
  end process;
end explicit;
```

9.2 Designing a Shift Register

A shift register is a sequential circuit used to shift bits in a word. In a shift left register, the bits are moved to the left, and the most significant bit is lost, as shown in Fig. 9.7. In a shift right register, the bits are moved to the right and the less significant bit is lost. In a circuit that rotates the bits to the left, the most significant bit of a word is moved to the less significant bit position. In a rotate right circuit, the bits are moved to the right, and the less significant bit is moved to the most significant position.

The VHDL behavioral description of registers discussed in previous chapters can be easily modified in order to implement shift and rotate registers. It is important to notice that registers are implemented in VHDL using explicit processes.

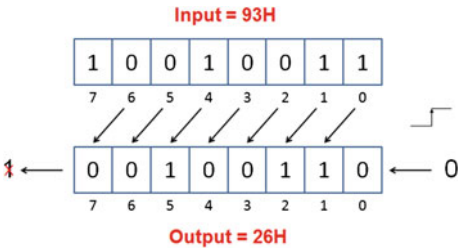


Fig. 9.7 Example of shift *left operation*

However, instead of reusing and adapting VHDL code from previous chapters, in this case study a template provided by Quartus II is used for the implementation of a shift register. To insert templates in a design, the target VHDL file must be opened in Quartus II text editor. Next, select the menu *Edit* → *Insert Template*… and *VHDL* → *Shift Registers* → *Basic Shift Registers*, as shown in Fig. 9.8.

In the proposed case study, an N bits input word is shifted left by one bit at each rising clock edge, and send to an output signal. This functionality is implemented by a shift register with enable and asynchronous reset. The original shift register template provided by Quartus II is changed, in order to implement the proposed functionality. The list of changes are as follows:

- A reset signal has been added.
- The "array" type was removed.
- The new input data is N bits long (instead of a single bit).
- After shifting, the less significant bit receives '0', and not the input bit.

The modified version of the shift register template is listed in Fig. 9.9, and it has the following features:

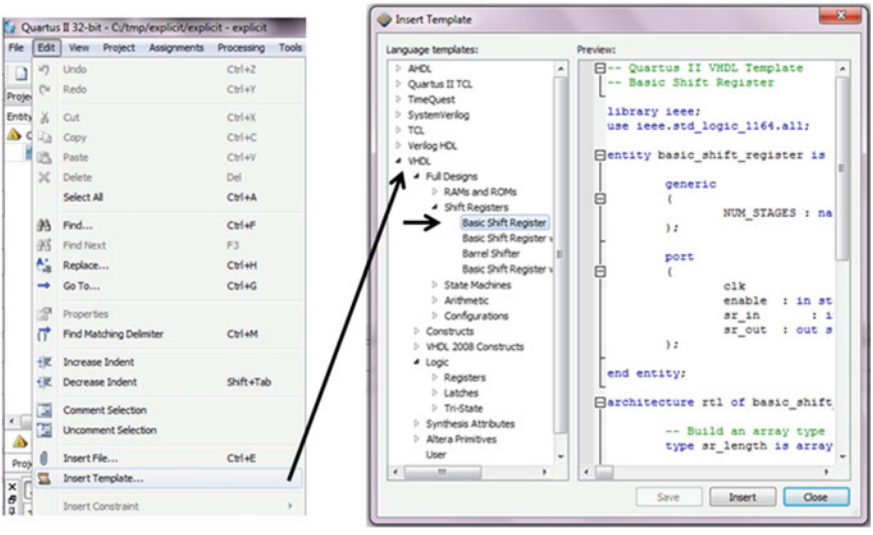


Fig. 9.8 Using quartus II to insert VHDL templates in a design

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity basic_shift_register is
4     generic(N : natural := 8);
5     port (clk    : in std_logic;
6           enable : in std_logic;
7           reset  : in std_logic;
8           sr_in  : in std_logic_vector((N - 1) downto 0);
9           sr_out : out std_logic_vector((N - 1) downto 0)
10    );
11 end entity;
12 architecture rtl of basic_shift_register is
13    signal sr: std_logic_vector ((N-1) downto 0);
14 begin
15    process (clk, reset)
16    begin
17       if (reset = '0') then
18          sr <= (others => '0');
19       elsif (rising_edge(clk)) then
20          if (enable = '1') then
21             -- Shift left 1 bit each clock
22             sr((N-1) downto 1) <= sr_in((N-2) downto 0);
23             sr(0) <= '0';
24          end if;
25       end if;
26    end process;
27    sr_out <= sr;
28 end rtl;
```

Fig. 9.9 Modified quartus II shift register template

- On line 4, the "generic" statement is used to make it easier to adapt this shift register implementation to different word sizes. For instance, in this example an 8 bits shift register is created, as N is replaced by 8 all over the code (see lines 8, 9, 13, and 22).
- The input (*sr_in*) and output (*sr_out*) data are 8 bits long (see lines 8 and 9).
- Each time the *reset* signal is not '0', *enable* is '1', and in the event of a rising clock edge, the 7 most significant bits of the internal signal *sr* receive the 7 less significant bits of the input data *sr_in* (see line 22). Also, the less significant bit of *sr* receives '0' (see line 23). The result is the input data *sr_in* shifted 1 bit to the left, as shown in Fig. 9.7.
- The circuit output is provided on line 27, when the internal signal *sr* is assigned to the output signal *sr_out*.

The synthesized shift register component is shown in Fig. 9.10.

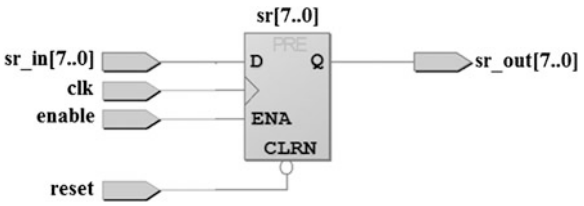


Fig. 9.10 Shift register synthesized by quartus II from VHDL listed in Fig. 9.9

9.3 Laboratory Assignment

The laboratory objectives are:

- to fix the concepts of shift register design in VHDL;
- to insert multiplication and division by 2 operations to the calculator case study.

Multiplication and division by 2 are simple operations to be performed in the base 2 numeral system. To multiply a binary number by 2, just shift the number 1 bit to the left and fill the less significant bit with '0'. The division by 2 is performed in a similar way, but shifting the number 1 bit to the right, and inserting a '0' in the most significant bit position. The operation represented in Fig. 9.7 is a

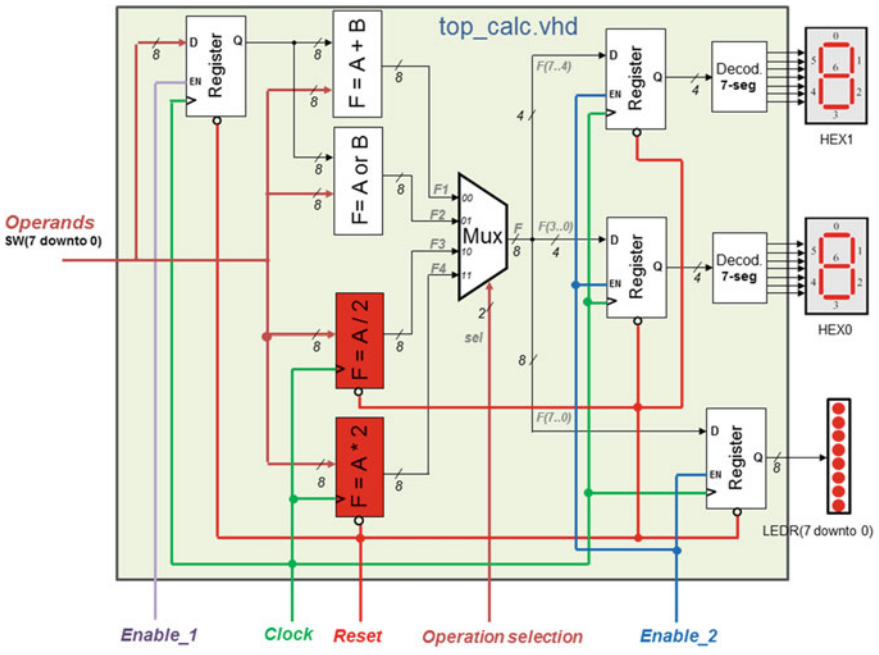


Fig. 9.11 Inserting the divide and multiply components to the calculator

multiply by 2, which resulted in an overflow as the target register is only 8 bits long, and the result is 9 bits long.

The component implemented in the shift register case study, and listed in Fig. 9.9, performs a multiply by 2 operation.

9.3.1 Laboratory Session

The tasks to be performed in this laboratory session are as follows:

- Using a shift register, design a component to perform division by 2 operations (*shift right*).
- Using a shift register, design a circuit to perform multiplication by 2 operations (*shift left*).
- Perform the simulation of both components, before inserting them in the calculator.
- Make the appropriate changes in the calculator implemented in previous chapters, in order to include the multiplication and division components:

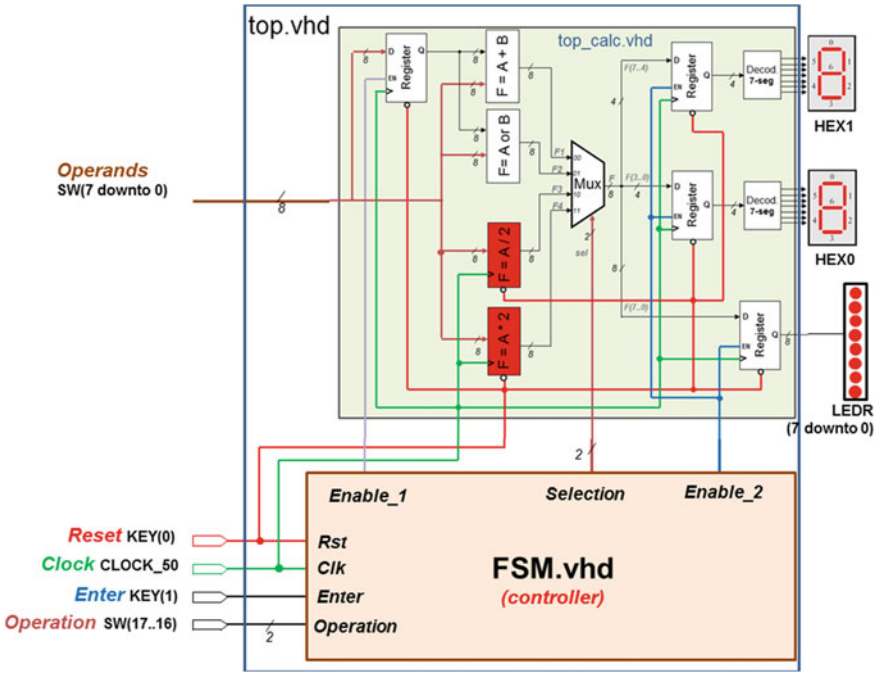


Fig. 9.12 The complete calculator design with the new components and the controller FSM

- The new multiplication and division components, as well as the not operation component, have also just one input. The FSM must be changed in order to deal with this situation.
- Replace the XOR component by the new divide by 2 component.
- Replace the NOT component by the new multiply by 2 component.

Figure 9.11 shows the block diagram of the new version for the calculator. The new components need a clock and a reset signals, but there is no need for an enable input. For instance, the component described in Fig. 9.9 shifts its input value 1 bit left. Thus, when the user provides a value to be multiplied by 2, and selects the multiplication operation, the mux output will be the input value shifted 1 bit left. At each rising clock edge, the multiplication component shifts its input 1 bit left and, while the input value remains the same, the output value will be also remain unchanged (but shifted 1 bit left in relation to the input value).

So, the shift register listed in Fig. 9.9 can be used to implement the multiply by 2 component, but without lines 20 and 24 as the enable input is not used in Fig. 9.11.

The division by 2 component can also be easily implemented from Fig. 9.9, but now the input value must be shifted to the right, and a '0' should be inserted in the most significant position bit.

Figure 9.12 shows the full block diagram of the calculator, including the controller FSM. As there is no need for enable signals in the new shift registers, the FSM can be used with minor changes. Basically, the FSM should be changed in order to manage two 1-input components, instead of just one 1-input component (NOT in the previous calculator version).

Chapter 10
Arithmetic Circuits

In previous chapters, the '+' VHDL operator is used in the adder design. In this chapter the adder component is conceived using logic gates, at the structural level. At the end of the chapter, the reader should be able:

- to understand the design process of adders using structural VHDL;
- to simulate the calculator case study using the new adder;
- to prototype the calculator case study in an FPGA board.

10.1 Half-Adder, Full-Adder, Ripple-Carry Adder

The adder is a fundamental circuit in digital systems. In an *n* bits adder, the main problem is the delay needed for the carry propagation. The carry propagation is a concern also in a multiplier circuit. The concepts behind the design of adders and multipliers are employed in the solution of several similar digital system problems. In Fig. 10.1, a four bits value ($A_{3..0}$) is added to another four bits value ($B_{3..0}$) resulting in a four bits summation ($S_{3..0}$). All bit position sum operations result also in a carry ($C_{3..0}$).

Two different components can be used in the design of this four bits adder. The half-adder component, shown in Fig. 10.2, has two inputs (A_0 and B_0) and two outputs (S_0 and C_1). The full-adder component, shown in Fig. 10.3, has three inputs (C_i, A_i and B_i) and two outputs (S_i and C_{i+1}). The half-adder component is used to perform the least significant bits (LSB) addition, and the full-adder component for the remaining bits.

In an FPGA based implementation, as there are no actual logic gates available in the device, an *n* bits adder can be implemented using only full-adders with no major resource usage penalties. The adder functionality will be implemented using the FPGA's look-up tables (LUTs), and the only concern is the amount of input signals, as there is no difference between a two input function performing a single AND operation (C_1 in the half-adder), or three ANDs and two ORs (C_{i+1} in the full-adder). Both functions will employ exactly the same amount of FPGA resources (LUTs).

E. A. Bezerra and D. V. Lettnin, *Synthesizable VHDL Design for FPGAs*,
DOI: 10.1007/978-3-319-02547-6_10,

$$
\begin{array}{rcccc}
 & C_4 & C_3 & C_2 & C_1 \\
 & & A_3 & A_2 & A_1 & A_0 \\
+ & & B_3 & B_2 & B_1 & B_0 \\
\hline
 & & S_3 & S_2 & S_1 & S_0
\end{array}
\begin{array}{l}
\longleftarrow \text{ carry} \\
\longleftarrow 1^{st} \text{ operand} \\
\longleftarrow 2^{nd} \text{ operand} \\
\longleftarrow \text{ sum}
\end{array}
$$

Fig. 10.1 A four bits sum operation

A_0	B_0	S_0	C_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

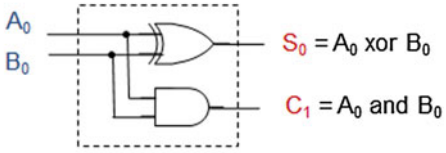


Fig. 10.2 Half-adder (HA) component. Truth table, schematic, and logic equations

C_i	A_i	B_i	S_i	C_{i+1}
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

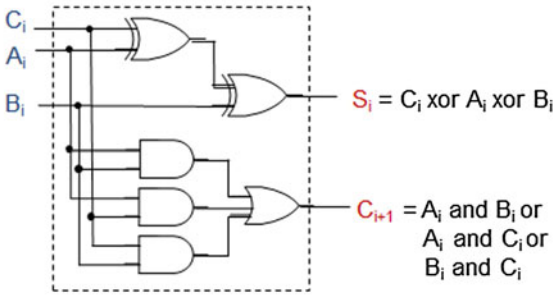


Fig. 10.3 Full-adder (FA) component. Truth table, schematic, and logic equations

There are several topologies available for implementing parallel adders. Figure 10.4 shows a four bits ripple-carry adder (RCA), which can be used to implement the operation described in Fig. 10.1. In this circuit, full-adders (FA) are used in all bit positions, including the LSB, where a half-adder could have been used. Having an FA in the LSB position is interesting also in order to have a component that can be used as a building block for larger adders or subtractors.

As shown in Fig. 10.1, each bit of the result (S_i) can only be obtained after the previous adder stage has finished its carry (C_{i-1}) calculation. Consequently, an RCA component provides a result ($S_{n..0}$) only after all carry bits ($C_{n..1}$) are stable.

When working with signed values, an RCA component can be adapted in order to implement an add/subtract circuit. In Fig. 10.5, four multiplexers and four

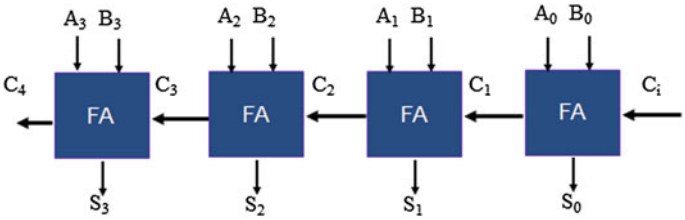


Fig. 10.4 A four bits ripple-carry adder (RCA)

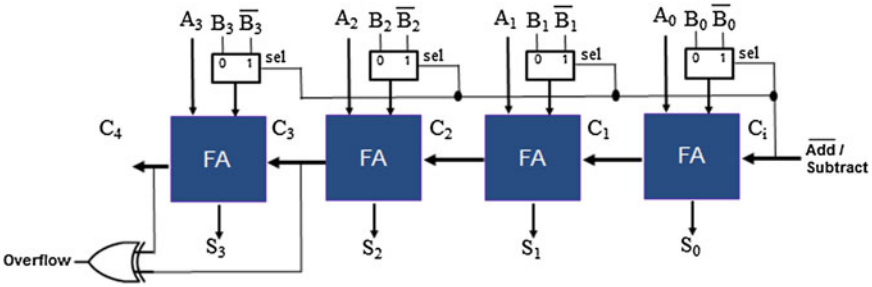


Fig. 10.5 Add/subtract circuit based on an RCA component

inverters are used in the B inputs of the RCA. When C_i is '1' the circuit works as a subtractor. In this case, the circuit performs an add operation between A and the two's complement of B:

$$S = A + B' + 1$$
$$S = A + (-B)$$
$$S = A - B$$

The RCA delay to provide a valid (stable) result depends on the input data. In order to have a better understanding of the carry propagation delay problem, a *time unit* is used as a performance figure. A *time unit* is the period required by a component to evaluate one level of logic gates. For instance, the half-adder component shown in Fig. 10.2 has just one level of logic gates, and all operations in this component are evaluated in one time unit. The full-adder shown in Fig. 10.3 has two levels of logic gates, and the operations are evaluated in two time units.

Considering A = 0101 and B = 1010 (with no carries), the circuit in Fig. 10.4 provides a valid result in two time units (2t). In the worst case scenario, 8t is the maximum delay acceptable for a four bits RCA. Larger adders would result in unacceptable delay figures. An 8 bits RCA should have a maximum delay of 16t, a 32 bits RCA the delay should be 64t, and so on.

Proposed solutions to decrease the delay usually perform the carry calculation in parallel, instead of the sequential approach employed in the RCA topology.

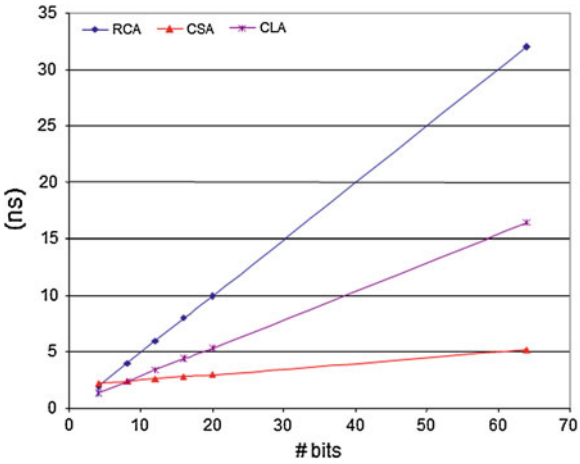


Fig. 10.6 Carry propagation delay for the three adders

The *Carry Lookahead Adder* (CLA) uses the functions *propagate* and *generate* in order to perform the parallel calculation of the carry. CLAs are faster than RCAs for four bits operands max. For longer length operands, the CLA carry generation logic complexity make it slower than the RCA. The *Carry Select Adder* (CSA) uses two FAs for each bit operation and, as a consequence, the carry generation is faster than the RCA and CLA, when considering larger operands.

Figure 10.6 shows a perform analysis between the adders. CSA and CLA present similar performance figures when for operands up to 10 bits long. For larger operands, the CSA circuit is the fastest adder. For instance, for 64 bits long operands, RCA takes more than 30 ns to provide a result, CLA takes more than 15 ns, and CSA provides the result in just 5 ns.

The drawback is that faster adders require more hardware resources to be implemented. In Fig. 10.7 it is possible to observe that the synthesized CLA takes almost twice the area than the RCA. On the other hand, CLA is about 10 % faster than RCA.

RCA is the slowest adder, but it has the less complex implementation. Two VHDL implementations of the RCA are discussed next, and compared to the

Fig. 10.7 Synthesis results summary for an RCA and a CLA implementation

Ripple-Carry Adder (RCA)

# bits	Area (logic gates)	Max delay (ns)
4	13	15.647
8	33	19.058

Carry Lookahead Adder (CLA)

# bits	Area (logic gates)	Max delay (ns)
4	23	14.268
8	62	17.484

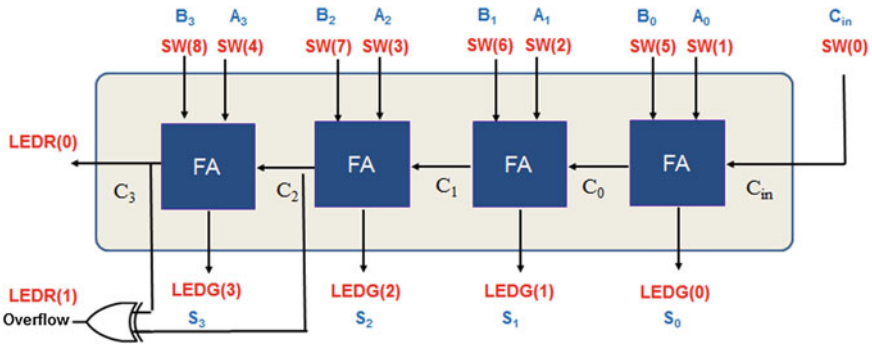
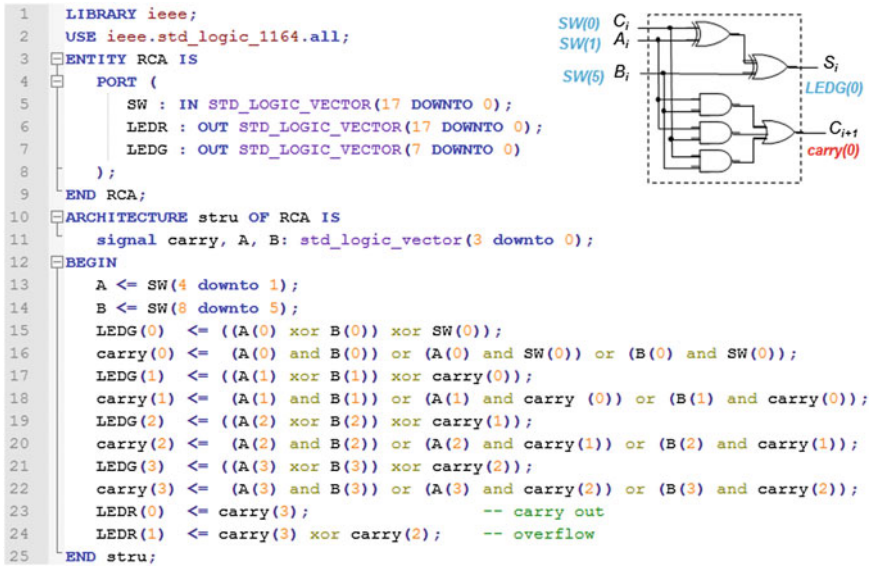


Fig. 10.8 Block diagram of the four bits RCA circuit implemented in VHDL

synthesis tool built-in adder alternative. Figure 10.8 shows the block diagram of the four bits RCA case study. This circuit considers two's complement signed numbers. The red LEDs 0 and 1 are used to indicate carry out and overflow, respectively. The green LEDs 0–3 provide sum results. The inputs are provided by the SW switches, including A and B operands, and also the carry in. Internal signals are used to connect all carries between FAs.

In the first implementation, the A and B internal signals are used just to make the code more readable. On lines 13 and 14 in Fig. 10.9, the FPGA board switches SW4..1 and SW8..5 are assigned to the internal signals A3..0 and B3..0, respectively.




```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  ENTITY RCA IS
4     PORT (
5        SW : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
6        LEDR : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
7        LEDG : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
8     );
9  END RCA;
10 ARCHITECTURE stru OF RCA IS
11    signal carry, A, B: std_logic_vector(3 downto 0);
12 BEGIN
13    A <= SW(4 downto 1);
14    B <= SW(8 downto 5);
15    LEDG(0)  <= ((A(0) xor B(0)) xor SW(0));
16    carry(0) <=  (A(0) and B(0)) or (A(0) and SW(0)) or (B(0) and SW(0));
17    LEDG(1)  <= ((A(1) xor B(1)) xor carry(0));
18    carry(1) <=  (A(1) and B(1)) or (A(1) and carry (0)) or (B(1) and carry(0));
19    LEDG(2)  <= ((A(2) xor B(2)) xor carry(1));
20    carry(2) <=  (A(2) and B(2)) or (A(2) and carry(1)) or (B(2) and carry(1));
21    LEDG(3)  <= ((A(3) xor B(3)) xor carry(2));
22    carry(3) <=  (A(3) and B(3)) or (A(3) and carry(2)) or (B(3) and carry(2));
23    LEDR(0)  <= carry(3);                -- carry out
24    LEDR(1)  <= carry(3) xor carry(2);   -- overflow
25 END stru;
```

Fig. 10.9 Solution I—RCA implementation using logic equations

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  ENTITY RCA IS
4     PORT (
5        SW : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
6        LEDR : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
7        LEDG : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
8     );
9  END RCA;
10 ARCHITECTURE stru OF RCA IS
11    signal carry, A, B: std_logic_vector(3 downto 0);
12    component FA
13       port (a, b, c: in std_logic;
14             sum, carry: out std_logic);
15    end component;
16 BEGIN
17    A <= SW(4 downto 1);
18    B <= SW(8 downto 5);
19    FA0: FA port map (A(0), B(0), SW(0),    LEDG(0), carry(0));
20    FA1: FA port map (A(1), B(1), carry(0), LEDG(1), carry(1));
21    FA2: FA port map (A(2), B(2), carry(1), LEDG(2), carry(2));
22    FA3: FA port map (A(3), B(3), carry(2), LEDG(3), carry(3));
23    LEDR(0)  <= carry(3);                   -- carry out
24    LEDR(1)  <= carry(3) xor carry(2);      -- overflow
25 END stru;
26 -- FA component
27 LIBRARY ieee;
28 USE ieee.std_logic_1164.all;
29 ENTITY FA IS
30    PORT (a, b, c: in std_logic;
31          sum, carry: out std_logic);
32 END FA;
33 ARCHITECTURE FA_stru OF FCA IS
34 BEGIN
35    sum   <= (a xor b) xor c;
36    carry <= b when ((a xor b) = '0') else c;
37 END FA_stru;
```

Fig. 10.10 Solution II—RCA implementation using component/port map

In the second implementation, the FA circuit is written as a VHDL component, making the RCA design more organized and easier to follow. The FA entity/architecture start on line 27 in Fig. 10.10. The four FA instances are created and interconnected on lines 19 through 22.

The VHDL coding style adopted in Solution II results in a more organized circuit also when using the Netlist Viewer option available in Quartus II tool. In Fig. 10.11, the four FA components used for the RCA implementation are explicitly shown in the Solution II netlist view. In order to see the FA logic gates

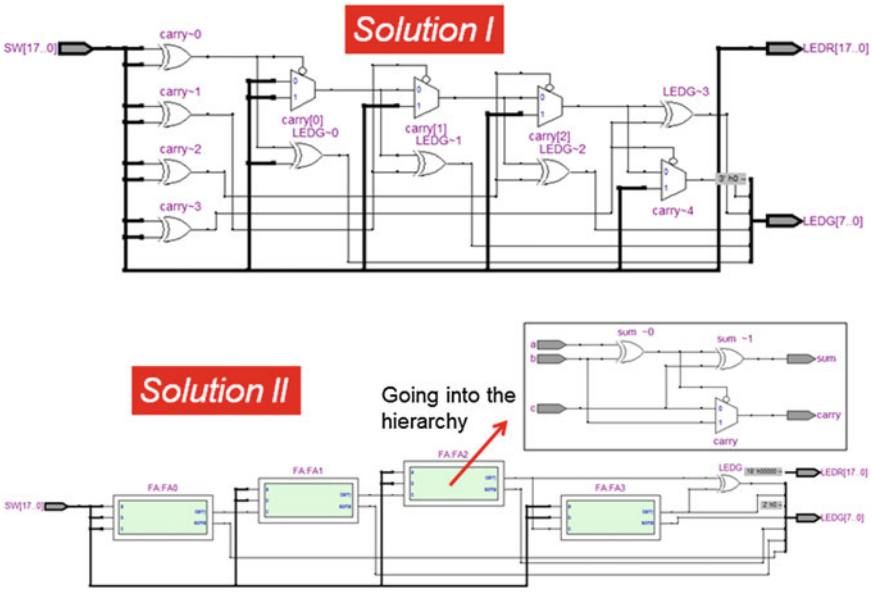


Fig. 10.11 Quartus II netlist view for the RCA circuit

implementation, the designer has to double-click in one of the FA components. On the other hand, in Solution I circuit view, the adder functionality understanding is not as straightforward as in Solution II.

In previous chapters, the adder used in the calculator design was implemented using the VHDL '+' operator. This is a much easier way to implement an adder, but the drawback is the loss of circuit's observability and controllability. In Fig. 10.12, the adder is written in a higher abstraction level, and the designer does not know which topology the synthesis tool will select for the implementation. For instance, there is no way to force the synthesis tool to implement the circuit using an RCA topology. In this example, with the lower observability degree, it is not possible to extract the overflow and carry out data.

In Fig. 10.13 there is a comparison between the synthesis results for the RCA implementation listed in Fig. 10.10 (left hand side), and the adder implementation using the '+' operator listed in Fig. 10.12 (right hand side). As expected, the '+' operator alternative, resulted in less FPGA resources usage (5 logic elements), as the synthesis tool will select the best algorithm available for the adder implementation.

However, in applications where a higher degree of controllability/observability is required, the RCA implementations described before as Solutions I and II are a better alternative for an adder design than the '+' VHDL operator.

The *carry out* and *overflow* signals observed in Solutions I and II are also known as *flags*. A flag is a strategy widely used in digital systems to indicate the status or a condition happened in the last arithmetic or logic operation. In the

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  use IEEE.std_logic_unsigned.all;
4  ENTITY RCA IS
5      PORT (
6          SW : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
7          LEDR : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
8          LEDG : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
9      );
10 END RCA;
11 ARCHITECTURE RCA_beh OF RCA IS
12     signal carry, A, B: std_logic_vector(3 downto 0);
13 BEGIN
14     A <= SW(4 downto 1);
15     B <= SW(8 downto 5);
16     LEDG(3 downto 0) <= A + B + ("000" & SW(0));
17     -- LEDR(0)  <= carry(3);                  -- carry out
18     -- LEDR(1)  <= carry(3) xor carry(2);     -- overflow
19 END stru;
```

Fig. 10.12 Adder implementation using the VHDL '+' operator

Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Met timing requirements	Yes
Total logic elements	10 / 33,216 (< 1 %)
Total combinational functions	10 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)

Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Met timing requirements	Yes
Total logic elements	5 / 33,216 (< 1 %)
Total combinational functions	5 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)

Fig. 10.13 Quartus II synthesis summary for Solution II and the '+' operator alternative

discussed examples, two flags have been implemented. One of them is used to indicate the event of a carry out in the operation, and the other one to indicate that an overflow has happen in the last operation. In this case, the user (or another circuit) can use this information in order to decide whether or not to use the result provided by the adder.

10.2 Laboratory Assignment

The laboratory objectives are:

- to understand the design of a structural RCA component in VHDL;
- to replace the adder used in the calculator by the new component.

10.2.1 Laboratory Session

The tasks to be performed in this laboratory session are as follows:

- Using the VHDL implementation of Solution II listed in Fig. 10.10, make the necessary modifications in order to have an 8-bits RCA component.
- The new component should have two 8-bits inputs, a 1-bit *carry in* input, an 8-bits sum output, and four 1-bit flags output.
- The flags should indicate the following events happened in the last operation performed by the adder: a *carry out*, an *overflow*, a *negative result*, and a *zero result*.
- Assume that the calculator makes operations in two's complement signed numbers.
- The *negative* flag can be obtained straight from the result (most significant bit).
- The *zero* flag should be on when all bits of the result are equal to zero.
- The four flags can be connected to the green LEDs, as some of the red LEDs are already in use by the calculator.

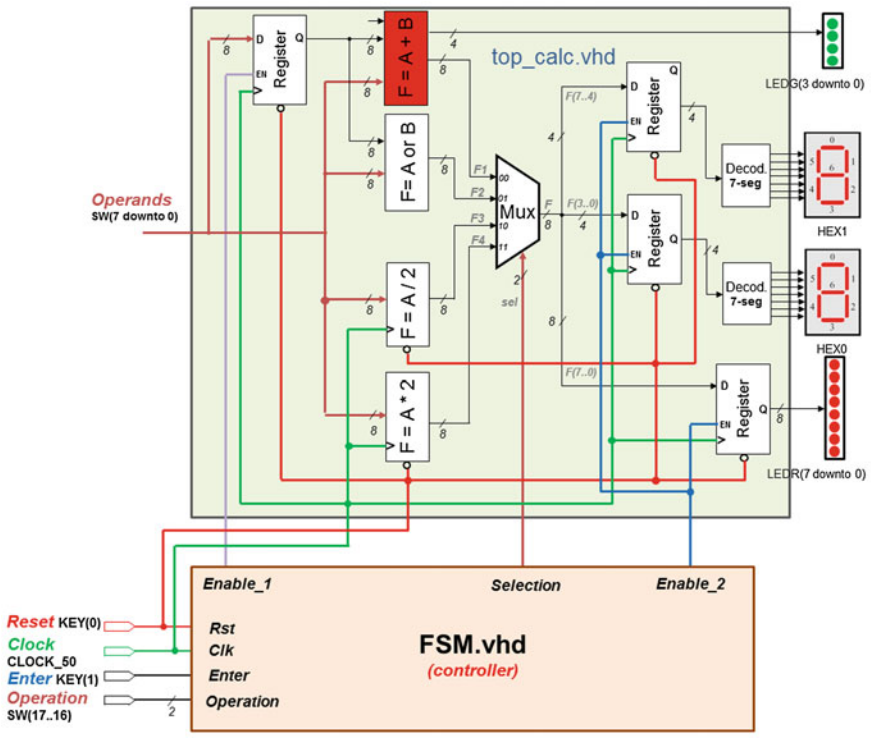


Fig. 10.14 The final calculator design, with the new RCA component

- It could be interesting to perform a timing simulation, instead of the functional simulation, in order to observe the delay needed by the RCA component to propagate the internal carry signals.
- Replace the C1 component used in previous chapters, by the new RCA component.
- The new design should be carefully designed, in order to avoid major changes in the remaining calculator components.
- Perform a final simulation in the design as a whole, this time with the new component added to the calculator.

Figure 10.14 shows the calculator design with the new RCA component, replacing the C1 component used in previous chapters. The new adder has an additional 4-bits output connected to the green LEDs (flags), and also a 1-bit input that is not connected (*carry in* signal).

Chapter 11
Writing Synthesizable VHDL Code for FPGAs

This chapter describes strategies for writing VHDL code for FPGA based designs. The strategies are presented in a way to show the stages of an application conception, considering a designer with some software development skills, but still a beginner in the hardware design language field. It starts presenting a high-level abstraction approach (software like description), but inadequate for the implementation of FPGA designs. The main mistakes made in all stages of the project are analyzed, and hardware design strategies are used in order to refine the design towards a working and efficient VHDL code.

11.1 Synthesis and Simulation

VHDL is a very complete language, but only a subset of it can be used to write synthesizable code. During the synthesis process, the VHDL source code is translated into a circuit for a target technology such as an FPGA or an ASIC, using the appropriate library components. Since VHDL offers similar resources found in typical software programming languages as, for instance, pointers and file management structures, consequently most of the features available in the language cannot be used in the synthesis processes for hardware generation. These "software-like" features have been thought targeting the development of simulation models. Considering all the available facilities, the possibility of developing hardware in a similar way to software appeals to designers with a software background. Usually, VHDL books describe the language as a whole, and it can be a bit confusing for beginners which may have the impression that systems can be implemented in VHDL in the same way as in languages like Java or C. In general, VHDL books discuss aspects of the language as, for instance, procedures, functions, data types, sequential and concurrent assignments, attributes, delta delays, but, in the examples presented of how to write a synthesizable VHDL code, only a small subset of the language is used.

The IEEE Std 1076™-2008—"IEEE Standard VHDL Language" defines the language as a whole, including simulation and synthesis semantics. The IEEE Std

E. A. Bezerra and D. V. Lettnin, *Synthesizable VHDL Design for FPGAs*,
DOI: 10.1007/978-3-319-02547-6_11,

1076.6™-2004—"IEEE Standard for VHDL register transfer level (RTL) Synthesis", describes only VHDL synthesis semantics. IEEE Std 1076.6 states that the synthesis tools should be compliant to the standard, but may have additional features. The standard covers the synthesis semantics to model, among others, level-sensitive and edge-sensitive logic. An important contribution of the standard is regarding code interoperability, as it defines in details all constructs that should be supported and also what should not be supported by the synthesis tools. The idea is to mitigate errors and mismatches between the simulation and synthesis results. This is a relatively new standard, and up to now each synthesis tool available still adopts its own VHDL synthesis semantics. The vendors are more interested in high quality synthesis results than VHDL code portability. Certainly, an acceptable level of compliance with the standard is found in all major synthesis tools, but it is still far from the ideal situation where a VHDL code written for a specific synthesis tool could be direct ported to another vendor's tool.

The situation is completely different when considering VHDL for simulation. The available tools seem to follow the IEEE 1076 standard, and usually there are no major problems when porting a VHDL simulation code from one vendor to another. VHDL simulation is a well-established subject, and its study is not one of the targets of this book. The strategies for the VHDL-based design discussed in this book target the VHDL synthesis topic.

11.2 VHDL Semantics for Synthesis

When asked to initialize a RAM memory (an array) with a sequence of values, a hardware designer learner with some sort of software background could easily come up with the VHDL code shown in Fig. 11.1.

This is an acceptable VHDL implementation to initialize a RAM memory, but it is suitable for a simulation model, and a synthesis tool would not generate a circuit with the expected functionality. This code was implemented without taking consideration of any synthesis criteria, with VHDL used like a programming language in a level of abstraction similar to the C language. The source code was written in this way because the designer had a general knowledge of the VHDL language, but was not familiar with VHDL for synthesis restrictions. Some of the problems in Fig. 11.1 cause synthesis errors, whilst others cause warnings that must be fixed in order to avoid discrepancies in the simulation before and after synthesis. The most dangerous situations are those where there are neither errors nor warnings, but in spite of this the synthesis tool creates hardware with behavior different from the expected one. The *for loop* construction, for instance, may result in a circuit having 256 adders storing a value sequentially. Another mistake made in this piece of VHDL code, is the use of two *wait* statements in the same process. This is not

```
1  entity RampUpMEM is
2     port (clock, reset: in std_logic;
3           stop: in std_logic;
4           addr: out std_logic_vector(7 downto 0);
5           ...
6           );
7  end RampUpMEM;
8
9  architecture bhv of RampUpMEM is
10    type RAM_TYP is array(255 downto 0) of std_logic_vector(7 downto 0)
11    signal my_mem: RAM_TYP;
12    signal stop1, stop2: std_logic;
13 begin
14    process
15       variable A: std_logic_vector(7 downto 0) := "00000000";
16    begin
17       for i in 0 to 255 loop
18          my_mem(i) <= A   ;
19          A := A + 1;
20          wait for 10 ns;
21       end loop;
22       wait on stop, stop1, stop2;
23       if rising_edge(clock) then
24    ...
```

Fig. 11.1 Partial VHDL code for a RAM initialization application

acceptable in a VHDL for synthesis coding. Next, this and other problems are discussed and the proper VHDL semantics for synthesis is presented.

Variables and Signals

Variables and signals are both valid in VHDL for synthesis and for simulation. Signals are more suitable for hardware behavior description, as they do not change until the end of a process cycle. Variables, on the other hand, are updated instantaneously, in the same moment a value is assigned to them, and this behavior is not observed in actual hardware which is subjected to signal propagation delays. Despite that, synthesis tools allow the use of variables, but very specific rules and conventions must be followed, otherwise the generated circuit is not guarantee to function as the expected. For instance, in order to generate any sort of hardware from a variable, it is compulsory to assign the final variable result to a signal. Variables are very useful in simulation activities, as there is no delay associated to values attribution, making the simulation process much faster when comparing to the use of signals.

Wait Statements

In VHDL for synthesis, a process can have only one *wait* statement, and just in processes with no sensitivity list. The *wait* should be the first or the last statement of a process, and no timing information can be used in the *wait*. In Fig. 11.1 the VHDL semantic used is fine for simulation, but it has several problems regarding the use of *wait*, and a synthesis process will certainly fail. The process has more

than one *wait* statement, and on line 20 there is even timing information (10 ns). The synthesis tool would not be able to create a circuit to generate such delay. Wait for time statements are used in testbenches, for delay generation. A synthesizable delay circuit could be designed employing timers, or counters embedded in state machines.

Memory Inference

In an FPGA design the array contents of an application can be stored in actual memory blocks, in LUTs, or in available flip-flops. Most FPGAs have a small amount of embedded and distributed memory blocks that can be freely used by the designer. FPGA vendors usually provide specific tools for memory generation. The designer can use these tools to achieve optimal memory usage figures, or they can write their own synthesizable VHDL code describing the memory behavior. The problem with the former is that very specific synthesis rules should be followed, otherwise the tool may not understand the VHDL code, and decide for allocating LUTs or flip-flops instead of available FPGA memory resources. Also, a VHDL description that generates proper memory in a synthesis tool, may not work in the same way in another tool. This sort of synthesizable VHDL code is not as portable as it should be and, unfortunately, for efficient memory generation the best alternative is still the IP creation using the vendor's tools for the target FPGA. In Fig. 11.1, the type declaration on line 10 and the signal definition on line 11 can be used by a synthesis tool to infer memory using the available memory blocks in FPGAs in general. However, the synthesis process will fail to infer memory, as the array access in the process body is not performed in an expected way.

The VHDL code in Fig. 11.2 is suitable for memory inference using Quartus II, and it worked as expected for the target Cyclone II FPGA resulting in 2,048 memory bits usage. The same VHDL code has been used to infer memory for a Xilinx FPGA, using Synplify from Synopsys, and it also worked as expected. However, when using the same VHDL code with Synplify to infer memory for the Microsemi's ProASIC3e FPGA, the inferred memory presented access problems. This is a nice way to describe a RAM memory in VHDL for synthesis, but some tools may use, for instance, flip-flops instead of the available RAM in the target device.

Additionally, Quartus II provides the following facilities for memory designs:

- There are VHDL examples of synthesizable RAM and ROM components available in the menu *Edit → Insert Template … → VHDL → Full Designs → RAMs and ROMs*.
- An IP core can be created using *Tools → MegaWizard Plug-In Manager → Create a new custom megafunction variation → Installed Plug-Ins → Memory Compiler*.

It is important to emphasize that the component generated using the second alternative can be used only in Quartus II based designs, and in this case there is no portability.

```
6  entity RAM is
7      port (en, clk : in std_logic;
8            data_in : in std_logic_vector(7 downto 0);
9            addr : in std_logic_vector(7 downto 0);
10           data_out : out std_logic_vector(7 downto 0)
11           );
12 end RAM;
13 architecture bhv of RAM is
14     type mem_type is array (255 downto 0) of std_logic_vector (7 downto 0);
15     signal mem : mem_type;
16 begin
17     process(clk, en, addr)
18     begin
19         if (rising_edge(clk)) then
20             if (en = '1') then
21                 mem(conv_integer(addr)) <= data_in;
22             end if;
23         end if;
24     end process;
25     data_out <= mem(conv_integer(addr));
26 end bhv;
```

Fig. 11.2 RAM description suggested in synthesis tool documentation for VHDL RAM inference

Loop Statements in VHDL for Synthesis

The *for* loop in Fig. 11.1 is used, during simulation, to fill in the *my_mem* array with a numeric sequence from 0 to 255. As discussed before the VHDL code in Fig. 11.1 is not synthesizable for several reasons, and from the *for* loop construction the synthesis tool would not be able to produce and initialize a memory component. In order to generate the described loop behavior, the synthesis tool would have to unroll the *for* loop, and create a circuit to initialize all the 256 array positions simultaneously, in just one clock pulse. This is the expected behavior for the *for* loop listed in the process in Fig. 11.1. In VHDL for synthesis, the behavior of loop constructions (i.e. for, while, loop) are implicitly performed by processes. As described before, a process is cyclic, and it runs forever. Considering this behavior, a given algorithm can be modeled in VHDL, using clocked processes with test statements (i.e. if, case) to describe the behavior of all repetitions. A VHDL code with only test statements is easier to be synthesized than a code written using explicit loop constructions. The drawback is that the VHDL implementation with no explicit loop statements becomes more distant from the original algorithm, and may be more complex to test/debug.

A synthesizable version of the RAM initialization code shown in Fig. 11.1, can be obtained using the RAM component listed in Fig. 11.2 and the FSM listed in Fig. 11.3.

Therefore, in order to implement a synthesizable version for the *for* loop (Fig. 11.1, lines 17–21), in Fig. 11.3 a three process FSM has been used. In the first process the next state (PE) is assigned. The second process is responsible for defining the next state, and also for performing the "end condition" test for the

```
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 top is
7   port (clk, reset: in std_logic;
8         outp: out std_logic_vector(7 downto 0)
9   );
10 end top;
11 architecture top_beh of top is
12  type states is (S0, S1, S2);
13  signal EA, PE: states;
14  signal address, dt_out,
15         dt_in: std_logic_vector(7 downto 0);
16 component RAM
17  port (en, clk: in std_logic;
18        data_in, addr: in std_logic_vector(7 downto 0);
19        data_out: out std_logic_vector(7 downto 0)
20        );
21 end component;
22 begin
23  R0: RAM port map ('1', clk, dt_in, address, dt_out);
24  process(clk)
25  begin
26    if clk'event and clk='0' then
27      if reset='0' then
28        EA <= S0;
29      else
30        EA <= PE;
31      end if;
32    end if;
33  end process;
```

```
34  process( EA )
35  begin
36    case EA is
37      when S0 =>
38        PE <= S1;
39      when S1 =>
40        PE <= S2;
41      when S2 =>
42        if address = 255 then
43          PE <= S0;
44        else
45          PE <= S1;
46        end if;
47      when others =>
48    end case;
49  end process;
50  process(clk)
51  begin
52    if clk'event and clk='1' then
53      case EA is
54        when S0 =>
55          address <= "00000000";
56        when S1 =>
57          address <= address + '1';
58        when S2 =>
59          dt_in <= address;
60        when others =>
61      end case;
62    end if;
63  end process;
64  outp <= dt_out;
65 end top_beh;
```

Fig. 11.3 RAM initialization with no explicit loop statements

"*for*" loop (see line 42). The third process defines the "*for*" loop start value (see line 55), the memory address increment (see line 57), and the output (see line 18 in Fig. 11.1, and line 59 in Fig. 11.3). From Fig. 11.1 observation, it is straightforward that the fill in memory algorithm has an O(n) cost, as it is supposed to write a value to a memory position in each clock cycle. However, the same conclusion is not so direct when observing the three processes FSM in Fig. 11.3. This is a drawback when writing VHDL for synthesis code, that is, code readability and debugging. The IEEE Std 1076.6^{TM}-2004—"IEEE Standard for VHDL register transfer level (RTL) Synthesis", is a compulsory reading for good synthesis results. For instance, the asynchronous reset strategy used in previous chapters, states that a process must have only one clock signal, and its clock edge must be in the last *elseif* condition. No sequential statements are allowed before or after the *if* statement.

A *for* loop can be used in a synthesizable VHDL code, but the designer has to follow very specific rules. In Fig. 11.4 a for loop is used in a VHDL function.

The synthesis tool unrolls the *for* loop described in the function, and generates a hardware with 256 adders. This means that in order to perform the equation on line 23, the circuit will need one adder for each loop repetition.

```
1  library ieee;
2  use ieee.numeric_std.all;
3  use ieee.std_logic_1164.all;
4  use ieee.std_logic_unsigned.all ;
5  use ieee.std_logic_arith.all ;
6
7  entity count_ones is
8  port (
9     data_in: in std_logic_vector (255 downto 0);
10    cnt_out: out std_logic_vector (7 downto 0)
11 );
12 end entity count_ones;
13
14 architecture bhv of count_ones is
15
16 function count_1 (signal x: std_logic_vector)
17       return std_logic_vector is
18       variable total_1: integer;
19 begin
20    total_1 := 0;
21    for i in 0 to 255 loop
22       if x(i) = '1' then
23          total_1 := total_1 + 1;
24       end if;
25    end loop;
26    return std_logic_vector(to_unsigned(total_1, 8));
27 end function count_1;
28
29 begin
30  cnt_out <= count_1 (data_in);
31 end architecture bhv;
```

Fig. 11.4 A synthesizable *for* loop statement

11.3 HDLGen: Automatic Generation of Synthesizable VHDL

A good design approach is to split the circuit in components, and simulate/synthesize each component separately. Following this approach, when a component shows an adequate behavior, then it can be included in the whole design. The ideal situation is the one where each component is a file containing only one entity/architecture pair, with the architecture split into combinational and sequential parts, and with the sequential part having no more than two processes (ideally one). In the case of inexperienced VHDL designers, it is suggested that they first execute the synthesis and then the simulation of the design, otherwise much time is spent implementing and simulating code that may not be synthesized.

A tool has been developed targeting designers more familiar with software algorithms than hardware implementations. As shown in Fig. 11.5, using HDLGen

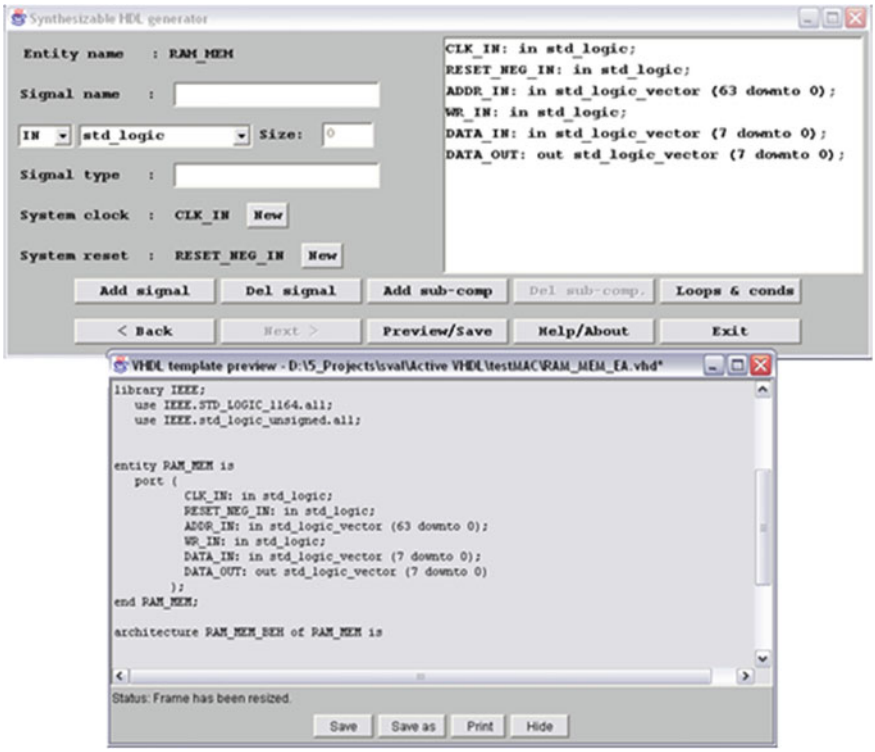


Fig. 11.5 HDLGen main window, and generated template file

the designer provides the description of the system's interface (signals), and the tool generates a template VHDL source code.

HDLGen was written in Java and it is based on strategies, methodologies and guidelines described in this book. HDLGen is free and can be downloaded off the author's website. This tool can be used by beginners in the development of synthesizable VHDL code, as a guide to avoid simple but, sometimes, expensive mistakes.

For instance, as discussed before, an experienced programmer when using VHDL for synthesis for the first time, may use a *for* statement for searching in an array which will not be understood by the synthesis tool. Using HDLGen, the programmer selects the "loop generation" option, provides the loop condition and then a state machine representing the loop constructor is generated in VHDL. More experienced developers may use HDLGen to link several components of a system, which facilitates the hierarchical view and project management. Fig. 11.5 shows HDLGen's main window (top) and the template file generated by the user

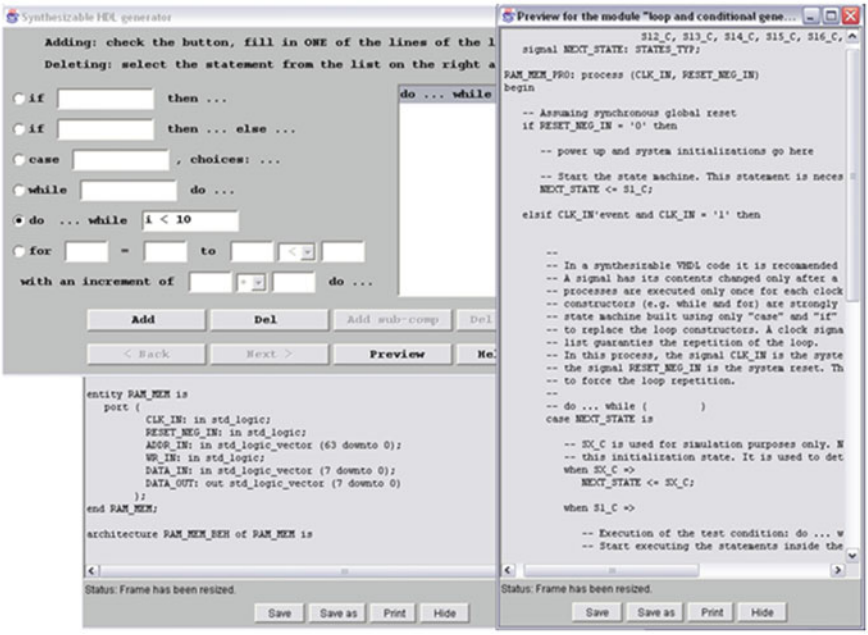


Fig. 11.6 HDLGen FSM generation for a loop statement

(bottom window). The "Loops & conds" button in the main window starts the HDLGen module responsible for the generation of synthesizable VHDL code for loops and conditional constructions. This module is shown in Fig. 11.6 (top left window), where the tool is used to generate an FSM in VHDL for the Do … While (I < 10) loop entered in the example. This loop unrolling feature can be used as an aid to some of the problems discussed before.

Bibliography

Vahid F (2011) Digital design with RTL design, VHDL, and verilog, 2nd edn. John Wiley and Sons Publishers, New York

IEEE Standard 1076-2008–IEEE Standard VHDL Language, IEEE Computer Society, 2008

IEEE Standard 1076.6-2004—IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, IEEE Computer Society, 2004

Ashenden PJ (2008) The designer's guide to VHDL, 3rd edn (systems on silicon). Morgan Kaufmann, Burlington

Katz R, Borrielo G (2005) Contemporary logic design, 2nd edn. Prentice Hall, New Jersey

Tocci RJ (2010) Digital systems: principles and applications, 11th edn. Prentice Hall, New Jersey

Kilts S (2007) Advanced FPGA design: architecture, implementation, and optimization. Wiley-IEEE Press, Hoboken

Pedroni V (2010) Circuit design and simulation with VHDL, 2nd edn. MIT Press, Cambridge

Mano M, Kime C (1999) Logic and computer design fundamentals, 2nd edn. Prentice Hall PTR, New Jersey

E. A. Bezerra and D. V. Lettnin, *Synthesizable VHDL Design for FPGAs*,
DOI: 10.1007/978-3-319-02547-6,