FPGA designs with VHDL

Meher Krishna Patel
Created on: October, 2017
Last updated: October, 2018

More documents are freely available at PythonDSP
# Table of contents

<table>
<thead>
<tr>
<th>1</th>
<th>First project</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Creating the project</td>
<td>1</td>
</tr>
<tr>
<td>1.3</td>
<td>Digital design using ‘block schematics’</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>Manual pin assignment and compilation</td>
<td>7</td>
</tr>
<tr>
<td>1.5</td>
<td>Load the design on FPGA</td>
<td>9</td>
</tr>
<tr>
<td>1.6</td>
<td>Digital design using ‘VHDL codes’</td>
<td>11</td>
</tr>
<tr>
<td>1.7</td>
<td>Pin assignments using ‘.csv’ file</td>
<td>11</td>
</tr>
<tr>
<td>1.8</td>
<td>Converting the VHDL design to symbol</td>
<td>13</td>
</tr>
<tr>
<td>1.9</td>
<td>Convert Block schematic to ‘VHDL code’ and ‘Symbol’</td>
<td>14</td>
</tr>
<tr>
<td>1.10</td>
<td>Conclusion</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>Overview</td>
<td>18</td>
</tr>
<tr>
<td>2.1</td>
<td>Introduction</td>
<td>18</td>
</tr>
<tr>
<td>2.2</td>
<td>Entity and Architecture</td>
<td>18</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Entity declaration</td>
<td>18</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Architecture body</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>Modeling styles</td>
<td>19</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Dataflow modeling</td>
<td>20</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Structural modeling</td>
<td>22</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Behavioral modeling</td>
<td>25</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Mixed modeling</td>
<td>26</td>
</tr>
<tr>
<td>2.4</td>
<td>Packages</td>
<td>27</td>
</tr>
<tr>
<td>2.5</td>
<td>Conclusion</td>
<td>28</td>
</tr>
<tr>
<td>3</td>
<td>Data types</td>
<td>29</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Lexical rules</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Library and packages</td>
<td>29</td>
</tr>
<tr>
<td>3.3.1</td>
<td>‘std_logic_1164’ package</td>
<td>29</td>
</tr>
<tr>
<td>3.3.2</td>
<td>‘numeric_std’ package</td>
<td>29</td>
</tr>
<tr>
<td>3.3.3</td>
<td>‘textio’ and ‘std_logic_textio’ packages</td>
<td>30</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Other standard packages</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>Entity and architecture</td>
<td>30</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Entity declaration</td>
<td>30</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Architecture body</td>
<td>30</td>
</tr>
<tr>
<td>3.5</td>
<td>Keyword ‘others’, ‘downto’ and ‘to’</td>
<td>31</td>
</tr>
<tr>
<td>3.6</td>
<td>Data objects</td>
<td>31</td>
</tr>
<tr>
<td>3.7</td>
<td>Data types</td>
<td>33</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Standard data type</td>
<td>33</td>
</tr>
<tr>
<td>3.7.2</td>
<td>User-defined scalar types</td>
<td>33</td>
</tr>
</tbody>
</table>
# 9 Finite state machines

## 9.1 Introduction

## 9.2 Comparison: Mealy and Moore designs

## 9.3 Example: Rising edge detector
- 9.3.1 State diagrams: Mealy and Moore design
- 9.3.2 Implementation
- 9.3.3 Outputs comparison
- 9.3.4 Visual verification

## 9.4 Glitches
- 9.4.1 Combinational design in asynchronous circuit
- 9.4.2 Unfixable Glitch
- 9.4.3 Combinational design in synchronous circuit

## 9.5 Moore architecture and VHDL templates
- 9.5.1 Regular machine
- 9.5.2 Timed machine
- 9.5.3 Recursive machine

## 9.6 Mealy architecture and VHDL templates
- 9.6.1 Regular machine
- 9.6.2 Timed machine
- 9.6.3 Recursive machine

## 9.7 Examples
- 9.7.1 Regular Machine: Glitch-free Mealy and Moore design
- 9.7.2 Timed machine: programmable square wave
- 9.7.3 Recursive Machine: Mod-m counter

## 9.8 When to use FSM design

## 9.9 Conclusion

# 10 Testbenches

## 10.1 Introduction

## 10.2 Testbench for combinational circuits
- 10.2.1 Half adder
- 10.2.2 Simple testbench
- 10.2.3 Testbench with process statement
- 10.2.4 Testbench with look-up table
- 10.2.5 Read data from file
- 10.2.6 Write data to file
- 10.2.7 Half adder testing using CSV file

## 10.3 Testbench for sequential circuits
- 10.3.1 Simulation with infinite duration
- 10.3.2 Simulation for finite duration and save data

## 10.4 Conclusion

# 11 Design examples

## 11.1 Introduction

## 11.2 Random number generator
- 11.2.1 Linear feedback shift register (LFSR)
- 11.2.2 Visual test

## 11.3 Shift register
## Table of contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.3.1</td>
<td>Bidirectional shift register</td>
<td>135</td>
</tr>
<tr>
<td>11.3.2</td>
<td>Visual test</td>
<td>136</td>
</tr>
<tr>
<td>11.3.3</td>
<td>Parallel to serial converter</td>
<td>137</td>
</tr>
<tr>
<td>11.3.4</td>
<td>Serial to parallel converter</td>
<td>139</td>
</tr>
<tr>
<td>11.3.5</td>
<td>Top level connection between serial and parallel converters</td>
<td>140</td>
</tr>
<tr>
<td>11.3.6</td>
<td>Visual test for serial and parallel converters</td>
<td>142</td>
</tr>
<tr>
<td>11.4</td>
<td>Random access memory (RAM)</td>
<td>143</td>
</tr>
<tr>
<td>11.4.1</td>
<td>Single port RAM</td>
<td>143</td>
</tr>
<tr>
<td>11.4.2</td>
<td>Visual test: single port RAM</td>
<td>145</td>
</tr>
<tr>
<td>11.4.3</td>
<td>Dual port RAM</td>
<td>146</td>
</tr>
<tr>
<td>11.4.4</td>
<td>Visual test: dual port RAM</td>
<td>147</td>
</tr>
<tr>
<td>11.5</td>
<td>Read only memory (ROM)</td>
<td>148</td>
</tr>
<tr>
<td>11.5.1</td>
<td>ROM implementation using RAM (block ROM)</td>
<td>148</td>
</tr>
<tr>
<td>11.5.2</td>
<td>ROM implementation logic cells (distributed ROM)</td>
<td>150</td>
</tr>
<tr>
<td>11.5.3</td>
<td>Distributed ROM vs Block ROM</td>
<td>150</td>
</tr>
<tr>
<td>11.5.4</td>
<td>Visual test</td>
<td>150</td>
</tr>
<tr>
<td>11.5.5</td>
<td>Defining ROM contents in file</td>
<td>152</td>
</tr>
<tr>
<td>11.6</td>
<td>LCD interface</td>
<td>155</td>
</tr>
<tr>
<td>11.6.1</td>
<td>Example 1</td>
<td>155</td>
</tr>
<tr>
<td>11.6.2</td>
<td>Example 2</td>
<td>161</td>
</tr>
<tr>
<td>11.6.3</td>
<td>Example 3: Error counting</td>
<td>166</td>
</tr>
<tr>
<td>11.7</td>
<td>VGA interface</td>
<td>170</td>
</tr>
<tr>
<td>11.7.1</td>
<td>Synchronization circuit</td>
<td>171</td>
</tr>
<tr>
<td>11.7.2</td>
<td>Visual test: change screen color with switches</td>
<td>173</td>
</tr>
<tr>
<td>11.7.3</td>
<td>Visual test: display different colors on screen</td>
<td>174</td>
</tr>
<tr>
<td>12</td>
<td>Simulate and implement SoPC design</td>
<td>177</td>
</tr>
<tr>
<td>12.1</td>
<td>Introduction</td>
<td>177</td>
</tr>
<tr>
<td>12.2</td>
<td>Creating Quartus project</td>
<td>177</td>
</tr>
<tr>
<td>12.3</td>
<td>Create custom peripherals</td>
<td>179</td>
</tr>
<tr>
<td>12.4</td>
<td>Create and Generate SoPC using Qsys</td>
<td>180</td>
</tr>
<tr>
<td>12.5</td>
<td>Create Nios system</td>
<td>183</td>
</tr>
<tr>
<td>12.6</td>
<td>Add and Modify BSP</td>
<td>185</td>
</tr>
<tr>
<td>12.6.1</td>
<td>Add BSP</td>
<td>185</td>
</tr>
<tr>
<td>12.6.2</td>
<td>Modify BSP (required for using onchip memory)</td>
<td>186</td>
</tr>
<tr>
<td>12.7</td>
<td>Create application using C/C++</td>
<td>186</td>
</tr>
<tr>
<td>12.8</td>
<td>Simulate the Nios application</td>
<td>188</td>
</tr>
<tr>
<td>12.9</td>
<td>Adding the top level VHDL design</td>
<td>190</td>
</tr>
<tr>
<td>12.10</td>
<td>Load the Quartus design (i.e., .sof/.pof file)</td>
<td>191</td>
</tr>
<tr>
<td>12.11</td>
<td>Load the Nios design (i.e., <code>.elf</code> file)</td>
<td>192</td>
</tr>
<tr>
<td>12.12</td>
<td>Saving NIOS-console's data to file</td>
<td>194</td>
</tr>
<tr>
<td>12.13</td>
<td>Conclusion</td>
<td>194</td>
</tr>
<tr>
<td>13</td>
<td>Reading data from peripherals</td>
<td>195</td>
</tr>
<tr>
<td>13.1</td>
<td>Introduction</td>
<td>195</td>
</tr>
<tr>
<td>13.2</td>
<td>Modify Qsys file</td>
<td>195</td>
</tr>
<tr>
<td>13.3</td>
<td>Modify top level design in Quartus</td>
<td>196</td>
</tr>
<tr>
<td>13.4</td>
<td>Modify Nios project</td>
<td>196</td>
</tr>
<tr>
<td>13.4.1</td>
<td>Adding Nios project to workspace</td>
<td>197</td>
</tr>
<tr>
<td>13.5</td>
<td>Add <code>C</code> file for reading switches</td>
<td>197</td>
</tr>
<tr>
<td>13.6</td>
<td>Simulation and Implementation</td>
<td>199</td>
</tr>
<tr>
<td>13.7</td>
<td>Conclusion</td>
<td>199</td>
</tr>
<tr>
<td>14</td>
<td>UART, SDRAM and Python</td>
<td>201</td>
</tr>
<tr>
<td>14.1</td>
<td>Introduction</td>
<td>201</td>
</tr>
<tr>
<td>14.2</td>
<td>UART interface</td>
<td>201</td>
</tr>
<tr>
<td>14.3</td>
<td>NIOS design</td>
<td>201</td>
</tr>
<tr>
<td>14.4</td>
<td>Communication through UART</td>
<td>204</td>
</tr>
<tr>
<td>14.5</td>
<td>SDRAM Interface</td>
<td>206</td>
</tr>
</tbody>
</table>
Chapter 1

First project

1.1 Introduction

In this tutorial, full adder is designed with the help of half adders. Here we will learn following methods to create/implement the digital designs using Altera-Quartus software,

- Digital design using ‘block schematics’,
- Digital design using ‘VHDL codes’,
- Manual pin assignment for implementation,
- Pin assignments using ‘.csv’ file,
- Loading the design on FPGA,
- Converting the ‘VHDL design’ to ‘Symbols’
- Converting the ‘Block schematic’ to ‘VHDL code’ and ‘Symbols’.

If you do not have the FPGA-board, then skip the last part i.e. ‘loading the design on FPGA’. Simulation of the designs using ‘Modelsim’ is discussed in Chapter 2.

Quartus II 11.1sp2 Web Edition and ModelSim-Altera Starter software are used for this tutorial, which are freely available and can be downloaded from the Altera website. All the codes can be downloaded from the website. First line of each listing in the tutorial, is the name of the VHDL file in the downloaded zip-folder.

1.2 Creating the project

- To create a new project, first open the Quartus and go to File→New Project Wizard, as shown in Fig. 1.1.
- ‘Introduction’ window will appear after this, click ‘next’ and fill the project details as shown in Fig. 1.2.
- After this, ‘Add files’ window will appear, click on ‘next’ here as we do not have any file to add to this project.
- Next, ‘Family and Device settings’ page will appear, select the proper device setting based on your FPGA board and click ‘Finish’ as shown in Fig. 1.3. If you don’t have FPGA board, then simply click ‘Finish’.
- After clicking on finish, the project will be created as shown in Fig. 1.4. Note that, the tutorials are tested on DE2-115, DE2 (cyclone-II family) or DE0-Nano boards, therefore project settings may be different for different chapters. You need to select the correct device while running the code on your system. This can be done by double-clicking on the device name, as shown in Fig. 1.4.
1.2. Creating the project

Fig. 1.1: Create new project

Fig. 1.2: Name and location of project
Chapter 1. First project

Fig. 1.3: Devices settings

Fig. 1.4: Updated Devices settings
1.3 Digital design using ‘block schematics’

Digitals design can be create using two methods i.e. using 'block-schematics' and with programming language e.g. VHDL or verilog etc. Both have their own advantages in the design-process, as we will observe in the later chapters of the tutorials.

In this section, we will create a half_adder using block-schematics method, as shown below,

- For this, click on File–>New–>Block diagram/ Schematics files, as shown in Fig. 1.5; and a blank file will be created.

![Fig. 1.5: Create new block schematics](image)

- Double click (anywhere) in the blank file, and a window will pop-up; select the ‘and’ gate from this window as shown in Fig. 1.6. Similarly, select the ‘xor’ gate.

- Next, right click on the ‘xor’ gate and then click on ‘Generate Pins for Symbol Ports’, as shown in Fig. 1.7.

- Now, connect the input ports of ‘xor’ gate with ‘and’ gate (using mouse); then Next, right click on the ‘and’ gate and then click on ‘Generate Pins for Symbol Ports’. Finally rename the input and output ports (i.e. x, y, sum and carry) as shown in Fig. 1.8.

- Finally, save the design with name ‘half_adder_sch.bdf’. It’s better to save the design in the separate folder, so that we can distinguish the user-defined and system-generated files, as shown in Fig. 1.9 where VHDL codes are saved inside the ‘VHDLCodes’ folders, which is inside the main project directory.

- Since the project name is ‘full_adder’, where as the half adder’s design name is ‘half_adder_sch.bdf’ (i.e. not same as the project name), therefore we need to set this design as top level entity for compiling the project. For this, go to project navigator and right click on the ‘half_adder_sch.bdf’ and set it as top level entity, as shown in Fig. 1.10.
Chapter 1. First project

Fig. 1.6: Select ‘and’ gate

Fig. 1.7: Add ports
1.3. Digital design using 'block schematics'

Fig. 1.8: Make connections

Fig. 1.9: Save project in separate directory i.e. VHDLCodes here
1.4 Manual pin assignment and compilation

Please enter correct pin location according to your FPGA board, as shown in this section. If you do not have the board, then skip this section and go to Section 1.6.

Once design is analyzed, then next step is to assign the correct pin location to input and output ports. This can be done manually or using .csv file. In this section, we will assign pin manually. Follow the below steps for pin assignments,

- First open the ‘Pin-planner’ by clicking Assignments→Pin Planner as shown in Fig. 1.12.
1.4. Manual pin assignment and compilation

Next, type the names of the input and output ports along with the pin-locations on the board, as shown in Fig. 1.13. Details of the Pin-locations are provided with the manual of the FPGA-boards e.g. in DE2-115 board, pin ‘PIN_AB28’ is connected with switch SW0. By assign this pin to ‘x’, we are connecting the port ‘x’ with switch SW0.

After assigning the pin, analyse the design again (see Fig. 1.11). After this, we can see the pin numbers in the ‘.bdf’ file, as shown in Fig. 1.14.

Finally, compile the design using ‘ctrl+L’ button (or by clicking processing->Start compilation, as shown in Fig. 1.15).
1.5 Load the design on FPGA

Follow the below steps to load the design on FPGA,

- Connect the FPGA to computer and turn it on.
- Full compilation process (Section 1.4), generates the .sof/.pof files, which can be loaded on the FPGA board. To load the design on FPGA board, go to Tools–>Programmer. And a programmer window will pop up.
- In the programmer window (see Fig. 1.17), look for two things i.e. position ‘1’ should display ‘USB-BLASTER’ and position ‘6’ should display the ‘.sof’ file. If any of this mission then follow below steps,
  - If USB-BLASTER is missing, then click on ‘Hardware setup (location 2 in Fig. 1.17)’ and then double click on USB-BLASTER in the pop-up window (location 3). This will display the USB-BLASTER at location 4. Finally close the pop-up window.
  - If ‘.sof’ file is not displayed at location 6, then click on ‘Add file...’ (location 7) and select the ‘.sof’ file from main project directory (or in output_files folder in main project directory).
- Finally click on the ‘start’ button in Fig. 1.17 and check the operation of ‘half adder’ using switches SW0 and SW1; output will be displayed on green LEDs i.e. LEDG0 and LEDG1.
1.5. Load the design on FPGA

Fig. 1.16: Direction of the ports

Fig. 1.17: Load the design on FPGA
1.6 Digital design using ‘VHDL codes’

In this section, half adder is implemented using VHDL codes. For this, click on File→New→VHDL files, as shown in Fig. 1.5; and a blank file will be created. Type the Listing ref{vhdl_half_adder_vhdl' in this file and save it as ‘half_adder_vhdl.vhd’.

Now, set this design as ‘top level entity’ (Fig. 1.10). We can analyze the design now, but we will do it after assigning the pins using .csv file in next section.

Listing 1.1: VHDL code for half adder

```vhdl
-- half_adder_vhdl.vhd
library ieee;
use ieee.std_logic_1164.all;

entity half_adder_vhdl is
    port(
        a, b : in std_logic;
        sum, carry : out std_logic
    );
end half_adder_vhdl;

architecture arch of half_adder_vhdl is
begin
    sum <= a xor b;
    carry <= a and b;
end arch;
```

1.7 Pin assignments using ‘.csv’ file

In this section, we will learn to assign the pins using .csv files. For this, Note that, we used input port as ‘a’ and ‘b’ in VHDL design (instead of ‘x’ and ‘y’ as in Fig. 1.8), so that we can observe the change in the pin assignments.

To assign the pins using csv file, follow the below steps,

- First type the content in Fig. 1.18 in a text-file and save it as ‘pin_assg_file.csv’.

Fig. 1.18: Content of pin_assg_file.csv

- Next, click on the Assignments→Import Assignments as shown in Fig. 1.19. And locate the file pin_assg_file.csv by clicking on the $cdots$ button, in the popped-up window, as shown in Fig. 1.20.

- Now, analyze the design (Fig. 1.11) and then open the pin planner (Fig. 1.12). We can see the new pin assignments as shown in Fig. 1.21 (If proper assignments do not happen then check, whether the VHDL design is set as top level or not and import assignments again and analyze the design).
1.7. Pin assignments using `.csv` file

Fig. 1.19: Import assignments

Fig. 1.20: Locate the csv file
Finally, compile and load and check the design as discussed in Section 1.5.

1.8 Converting the VHDL design to symbol

VHDL code can be converted into block schematic format, which is quite useful for connecting various modules together. In this section, half adder’s vhdl file is converted into schematic and then two half adder is connected to make a full adder. Note that, this connection can be made using VHDL code as well, which is discussed in Chapter 2.

Follow the below steps to create a full adder using this method,

- Right click on the ‘half_adder_vhdl.vhd’ and click on ‘Create symbol file for current file’ as shown in Fig. 1.22. It will create a symbol for half adder design.

Fig. 1.22: Convert VHDL code to symbol
• Now, create a new ‘block schematic file’ (Fig. 1.5).
• Next, double click on this file and add the half adder symbol as shown in Fig. 1.23.

![Add half adder symbol](image1.jpg)

**Fig. 1.23: Add half adder symbol**

• Again add one more ‘half adder symbol’ along with ‘or’ gate and connect these components as shown in Fig. 1.24.

![Full adder using half adders](image2.jpg)

**Fig. 1.24: Full adder using half adders**

• Since, one more port (i.e. c) is added to the design, therefore modify the ‘pin_assg_file.csv’ as shown in Fig. 1.25.

• Save the design as ‘full_adder_sch.bdf’.
• Import the assignment again; and compile the design (see pin assignments as well for 5 ports i.e. a, b, c, sum and carry). Finally load the design on FPGA.

### 1.9 Convert Block schematic to ‘VHDL code’ and ‘Symbol’

We can convert the ‘.bdf’ file to VHDL code as well. In this section, full adder design is converted to VHDL code. For this open the file ‘full_adder_sch.bdf’. Then go to File->Create/Update->Create HDL Design File... as shown in Fig. 1.26 and select the file type as ‘VHDL’ and press OK; the file will be saved in the VHDLCodes folder (see Fig. 1.27). The content of the generated ‘VHDL’ file are shown in Listing ref{vhdl_full_adder_sch'}. 

---

**PythonDSP**
Chapter 1. First project

Now, we can convert this VHDL code into symbol as shown in Section 1.8.

Note: Note that, if we can to convert the `.bdf` file into symbol, then we need to convert it into VHDL code first, and then we can convert the VHDL code into symbol file.

Listing 1.2: VHDL code for full adder

```
-- Copyright (C) 1991-2011 Altera Corporation
-- Your use of Altera Corporation's design tools, logic functions
-- and other software and tools, and its ANPP partner logic
-- functions, and any output files from any of the foregoing
-- (including device programming or simulation files), and any
-- associated documentation or information are expressly subject
-- to the terms and conditions of the Altera Program License
-- Subscription Agreement, Altera MegaCore Function License
-- Agreement, or other applicable license agreement, including,
-- without limitation, that your use is for the sole purpose of
-- programming logic devices manufactured by Altera and sold by
-- Altera or its authorized distributors. Please refer to the
-- applicable agreement for further details.

-- PROGRAM "Quartus II 32-bit"
-- VERSION "Version 11.1 Build 259 01/25/2012 Service Pack 2 SJ Web Edition"
-- CREATED "Tue May 30 09:12:57 2017"

LIBRARY ieee;
```

Fig. 1.25: Modify ‘pin_assg_file.csv’ file

Fig. 1.26: Convert schematic to VHDL
1.9. Convert Block schematic to ‘VHDL code’ and ‘Symbol’

Fig. 1.27: Select VHDL

(continued from previous page)

USE ieee.std_logic_1164.all;
LIBRARY work;

ENTITY full_adder_sch IS
  PORT
  (a : IN STD_LOGIC;
   b : IN STD_LOGIC;
   c : IN STD_LOGIC;
   sum : OUT STD_LOGIC;
   carry : OUT STD_LOGIC
  );
END full_adder_sch;

ARCHITECTURE bdf_type OF full_adder_sch IS

COMPONENT half_adder_vhdl
  PORT(a : IN STD_LOGIC;
       b : IN STD_LOGIC;
       sum : OUT STD_LOGIC;
       carry : OUT STD_LOGIC
  );
END COMPONENT;

SIGNAL SYNTHESIZED_WIRE_0 : STD_LOGIC;
SIGNAL SYNTHESIZED_WIRE_1 : STD_LOGIC;
SIGNAL SYNTHESIZED_WIRE_2 : STD_LOGIC;

BEGIN

  b2v_inst : half_adder_vhdl
  PORT MAP(a => a,
            b => b,
            sum => SYNTHESIZED_WIRE_0,
            carry => SYNTHESIZED_WIRE_1);

(continues on next page)
1.10 Conclusion

In this chapter, we learn to implement the design using schematic and coding methods. Also, we did the pin assignments manually as well as using csv file. Finally, we learn to convert the VHDL code into symbol file; and schematic design into VHDL code.

Note: Please see the Appendix B as well, where some more details about symbol connections are shown, along with the methods for using the codes provided in this tutorial.
Chapter 2

Overview

2.1 Introduction

VHDL is the hardware description language which is used to model the digital systems. VHDL is quite verbose, which makes it human readable. In this tutorial, following 3 elements of VHDL designs are discussed briefly, which are used for modeling the digital system.

- Entity and Architecture
- Modeling styles
  - Dataflow modeling
  - Structural modeling
  - Behavioral modeling
  - Mixed modeling
- Packages

The 2-bit comparators are implemented using various methods and corresponding designs are illustrated, to show the differences in these methods. All these topics are elaborated in later chapters. Note that, all the features of VHDL can not be synthesized i.e. these features can not be converted into designs. Non-synthesizable features are used to test the design by writing testbenches, which are discussed in Chapter 10. Rest of the chapters use only those features of VHDL which can be synthesized. All the codes in this tutorial are tested using Modelsim and implemented on FPGA board. Further, the implementation processes, i.e. pin-assignments and downloading the design on FPGA etc, are discussed in Chapter 1 and Chapter 8.

Note: Unlike any other electronics designs, if the VHDL design pass the simulation, then it guarantees that it will pass the physical implementation as well. Also, simulation is the only way to verify the large designs and lots of template are shown in Chapter 10. Some visual verification can also be performed for smaller designs by reducing the clock rate as discussed in Chapter 8.

2.2 Entity and Architecture

In this section, we discuss ‘entity declaration’ and ‘architecture body’ along with three different ways of modeling i.e. ‘data flow’, ‘structural’ and ‘behavioral’ modeling. In practice, these three styles are mixed together to model a digital circuit.

2.2.1 Entity declaration

Entity specifies the input-output ports of the design along with optional ‘generic constants’. The ‘generic constants’ are discussed in Section 3.11.2. Further, the architecture contains the VHDL codes which describe the functionality of the design, which is converted into hardware by the compiler. Lastly, library contains implementation the
commonly used designs. Some of the standard libraries are shown in Section 3.3. Also, we can create our own libraries using packages which are discussed in Section 2.4 and Chapter 6.

In Fig. 2.1, a simple ‘and’ gate is shown; which is generated by Listing 2.1. Listing 2.1 is included to understand the meaning of ‘entity declaration’ and ‘architecture body’. Also in VHDL, ‘–’ is used for comments; please read comments as well to understand the codes.

The entity declaration (lines 6-11) contains all the name of the input and outputs ports as shown in Listing 2.1. Here, the design has two input ports i.e. ‘x’ and ‘y’ and one output port i.e. ‘z’, which are defined inside the ‘port’ block in line 7. Name of the entity ‘andEx’ is defined in line 6. Lastly, we need to import libraries to the listing which contains various functions e.g. library ‘IEEE’ (line 3) contains the package ‘std_logic_1164’ (line 4), in which ‘std_logic’ is defined. ‘std_logic’ is used in line 8 and 9, to define the 1-bit input and output data-types. Lastly, entity block is closed with ‘end’ keyword in line 11. All these terms, i.e. IEEE library and packages along with data-types, are discussed in detail in Chapter 3.

Fig. 2.1: Circuit generated by Listing 2.1

Listing 2.1: ‘and’ gate example, design: Fig. 2.1

```vhdl
-- andEx.vhd
library ieee;
use ieee.std_logic_1164.all;

entity andEx is
  port(
    x, y : in std_logic;
    z: out std_logic
  );
end andEx;

architecture arch of andEx is
begin
  z <= x and y;
end arch;
```

### 2.2.2 Architecture body

Actual behavior of the design is defined in the ‘architecture body’. In Listing 2.1, ‘and’ gate is implemented with ‘x’ and ‘y’ as input, and ‘z’ as output. This behavior is defined in line 15. In line 13, the name of the architecture is defined as ‘arch’ and then name of the entity is given i.e. ‘andEx’. Complete logic is defined between ‘begin’ and ‘end’ statements i.e. line 14 and 16. Further, we can define intermediate signals of the design (i.e. apart from ports) between line 13-14 as shown in next sections. Next section contains more details about architecture body along with different modeling styles.

### 2.3 Modeling styles

In VHDL, the architecture can be defined in four ways as shown in this section. Two bit comparator is designed with different styles; which generates the output ‘1’ if the numbers are equal, otherwise output is set to ‘0’.
2.3. Dataflow modeling

In this modeling style, the relation between input and outputs are defined using signal assignments. In the other words, we do not define the structure of the design explicitly; we only define the relationships between the signals; and structure is implicitly created during synthesis process. Listing 2.1 is the example of dataflow design, where relationship between inputs and output are given in line 15.

Table 2.1: Truth table for 1 bit comparator, Listing 2.2

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>eq</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.2: Truth table for 2 bit comparator, Listing 2.3

<table>
<thead>
<tr>
<th>a[1] a[0]</th>
<th>b[1] b[0]</th>
<th>eq</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>1</td>
</tr>
<tr>
<td>0 0</td>
<td>0 1</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0</td>
<td>0 1</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 1</td>
<td>1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

In this section, two more examples of dataflow modeling are shown i.e. ‘1 bit’ and ‘2 bit’ comparators; which are used to demonstrate the differences between various modeling styles in the tutorial. Table 2.1 and Table 2.2 show the truth tables of ‘1 bit’ and ‘2 bit’ comparators. As the name suggests, the comparator compare the two values and sets the output ‘eq’ to 1, when both the input values are equal; otherwise ‘eq’ is set to zero. The corresponding boolean expressions are shown below.

For 1 bit comparator:
\[ eq = x' y' + xy \] (2.1)

For 2 bit comparator:

Above two expressions are implemented using VHDL in Listing 2.2 and Listing 2.3, which are explained below.

Explanations

Listing 2.2 implements the 1 bit comparator based on (2.1). Two intermediate signals are defined between ‘architecture declaration’ and ‘begin’ statement (known as declaration section) as shown in line 14. These two signals (‘s0’ and ‘s1’) are defined to store the values of x’y’ and ‘xy’ respectively. Values to these signals are assigned at line 16 and 17. Finally (2.1) performs ‘or’ operation on these two signals, which is done at line 19. When we compile this code using ‘Quartus software’, it implements the code into hardware design as shown in Fig. 2.2.
The compilation process to generate the design is shown in Appendix B. Also, we can check the input-output relationships of this design using Modelsim, which is also discussed briefly in Appendix B.

### Listing 2.2: Comparator 1 Bit

```vhdl
--comparator1Bit.vhd

library ieee;
use ieee.std_logic_1164.all;

entity comparator1Bit is
  port(
    x, y : in std_logic;
    eq : out std_logic
  );
end comparator1Bit;

architecture dataflow1Bit of comparator1Bit is
  signal s0, s1: std_logic;
begin
  s0 <= (not x) and (not y);
  s1 <= x and y;
  eq <= s0 or s1;
end dataflow1Bit;
```

**Note:** Note that, the statements in ‘dataflow modeling’ and ‘structural modeling’ (described in section Section 2.3.2) are the concurrent statements, i.e. these statements execute in parallel. In the other words, order of statements do not affect the behavior of the circuit; e.g. if we exchange line 16 and 19 in Listing 2.2, again we will get the Fig. 2.2 as implementation. This is discussed in detail in Section 4.3.

On the other hand, statements in ‘behavior modeling’ (described in section Section 2.3.3) executes sequentially and any changes in the order of statements will change the behavior of circuit.

### Explanation Fig. 2.2: 1 bit comparator

Fig. 2.2 is generated by Quartus software according to the VHDL code shown in Listing 2.2. Here, ‘s0’ is the ‘and’ gate with inverted inputs ‘x’ and ‘y’, which are generated according to line 16 in Listing 2.2. Similarly, ‘s1’ ‘and’ gate is generated according to line 17. Finally output of these two gates are applied to ‘or’ gate (named as ‘eq’) which is defined at line 19 of the Listing 2.2.

![Fig. 2.2: 1 bit comparator, Listing 2.2](image)

### Explanation Listing 2.3: 2 bit comparator

This listing implements the (2.2). Here, we are using two bit input, therefore ‘std_logic_vector’ is used at line 8. ‘1 downto 0’ sets the 1 as MSB (most significant bit) and 0 as LSB (least significant bit) i.e. the ‘a[1]’ and ‘b[1]’ are the MSB, whereas ‘a[0]’ and ‘b[0]’ are the LSB. Since we need to store four signals (lines 16-19), therefore ‘s’ is defined as 4-bit vector in line 14. Rest of the working is same as Listing 2.2. The implementation of this listing is shown in Fig. 2.3.
2.3. Modeling styles

---

Listing 2.3: Comparator 2 Bit

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity comparator2Bit is
  port(
    a, b : in std_logic_vector(1 downto 0);
    eq : out std_logic
  );
end comparator2Bit;

architecture dataflow2Bit of comparator2Bit is
  signal s: std_logic_vector(3 downto 0);
  begin
    s(0) <= (not a(1)) and (not a(0)) and (not b(1)) and (not b(0));
    s(1) <= (not a(1)) and a(0) and (not b(1)) and b(0);
    s(2) <= a(1) and (not a(0)) and b(1) and (not b(0));
    s(3) <= a(1) and a(0) and b(1) and b(0);
    eq <= s(0) or s(1) or s(2) or s(3);
  end dataflow2Bit;
```

Fig. 2.3: 2 bit comparator, Listing 2.3

2.3.2 Structural modeling

In previous section, we designed the 2 bit comparator based on (2.2). Further, we can design the 2 bit comparator using 1-bit comparator as well, with following steps,

- First compare each bit of 2-bit numbers using 1-bit comparator; i.e. compare ‘a[0]’ with ‘b[0]’ and ‘a[1]’ with ‘b[1]’ using 1-bit comparator (as shown in Table 2.2).
- If both the values are equal, then set the output ‘eq’ as 1, otherwise set it to zero.
This method is known as ‘structural’ modeling, where we use the pre-defined designs to create the new designs (instead of implementing the ‘boolean’ expression). This method is quite useful, because most of the large-systems are made up of various small design units. Also, it is easy to create, simulate and check the various small units instead of one large-system. Listing 2.4 and Listing 2.5 are the examples of structural designs, where 1-bit comparator is used to created a 2-bit comparator.

**Explanation Listing 2.4**

In this listing, line 6-11 defines the entity, which has two input ports of 2-bit size and one 1-bit output port. Then two signals are defined (line 14) to store the outputs of two 1-bit comparators, as discussed below.

‘eq_bit0’ and ‘eq_bit1’ in lines 16 and 18 are the names of the two 1-bit comparator used in this design. We can see these names in the resulted design, which is shown in Fig. Listing 2.4.

Next, ‘comparator1bit’ in lines 16 and 18 is the name of entity of 1-bit comparator (Listing 2.2). With this declaration, i.e. comparator1bit, we are calling the design of 1-bit comparator to current design.

Then, ‘port map’ statements in lines 17 and 19, are assigning the values to the input and output port of 1-bit comparator. For example, in line 17, input ports of 1-bit comparator, i.e. ‘x’ and ‘y’, are assigned the values of ‘a(0)’ and ‘b(0)’ from this design; and the output ‘y’ of 1-bit comparator is stored in the signal ‘s0’. Further, in line 21, if signals ‘s0’ and ‘s1’ are 1 then ‘eq’ is set to 1 using ‘and’ gate, otherwise it will be set to 0.

Lastly, ‘work’ in lines 16 and 18, is the compilation library; where all the compiled designs are stored. The statement ‘work.comparator1bit’ indicates to look for the ‘comparator1bit’ entity in ‘work’ library. Final design generated by Quartus software for Listing 2.4 is shown in Fig. 2.4.

```
--comparator2BitStruct.vhd
library ieee;
use ieee.std_logic_1164.all;

entity comparator2BitStruct is
  port(
    a, b : in std_logic_vector(1 downto 0);
    eq : out std_logic
  );
end comparator2BitStruct;

architecture structure of comparator2BitStruct is
  signal s0, s1: std_logic;
begin
  eq_bit0: entity work.comparator1bit
    port map (x=>a(0), y=>b(0), eq=>s0);
  eq_bit1: entity work.comparator1bit
    port map (x=>a(1), y=>b(1), eq=>s1);
  eq <= s0 and s1;
end structure;
```

**Explanation Fig. 2.4**

In this figure, a[1..0] and b[1..0] are the input bits whereas ‘eq’ is the output bit. Thick lines after a[1..0] and b[1..0] show that there are more than 1 bits e.g. in this case these lines have two bits. These thick lines are changed to thin lines before going to comparators; which indicates that only 1 bit is sent as input to comparator.

In ‘comparator1Bit: eq_bit0’, the ‘comparator1Bit’ is the name of the entity defined for 1-bit comparator (Listing 2.2); whereas the ‘eq_bit0’ is the name of this entity defined in line 16 of listing Listing 2.4. Lastly outputs of two 1-bit comparator are sent to ‘and’ gate according to line 21 in listing Listing 2.4.
2.3. Modeling styles

Hence, from this figure we can see that the 2-bit comparator can be designed by using two 1-bit comparators.

**Explanation Listing 2.5**

The working of the listing is same as Listing 2.5, with some small differences as discussed here. In Listing 2.4, work directory is used to find the 1-bit comparator design; whereas in Listing 2.5, the 1-bit comparator is explicitly declared as ‘component’ in 2-bit comparator design as shown in line 14-19. Further, in lines 22 and 24 of Listing 2.5, the name of the component i.e. ‘comparator1Bit’ is defined; instead of ‘work.comparator1Bit’ which is used in lines 16 and 18 of Listing 2.4. The final design generated by Quartus software is shown in Fig. 2.5 which is exactly same as Fig. 2.4.

Listing 2.5: Structure modeling using component declaration

```vhdl
--comparator2BitStructComponent.vhd
library ieee;
use ieee.std_logic_1164.all;

entity comparator2BitStructComponent is
  port(
    a, b : in std_logic_vector(1 downto 0);
    eq : out std_logic
  );
end comparator2BitStructComponent;

architecture structure of comparator2BitStructComponent is
  component comparator1Bit
    port(
      x, y : in std_logic;
      eq : out std_logic
    );
  end component;
  signal s0, s1: std_logic;
begin
  eq_bit0: comparator1Bit
    port map (x=>a(0), y=>b(0), eq=>s0);
  eq_bit1: comparator1Bit
    port map (x=>a(1), y=>b(1), eq=>s1);
  eq <= s0 and s1;
end structure;
```

- Note that, multiple architectures can be defined for one entity. For example, in this tutorial, various architectures are created for two bit comparator with different entity names; but these architectures can be saved in single file with one entity name. Then, ‘configuration’ method can be used to select a particular architecture, which may result in complex code.
- Throughout the tutorials, we use only single architecture for each entity, therefore ‘configuration’ is not
Chapter 2. Overview

Fig. 2.5: 2 bit comparator, Listing 2.5

discussed in this tutorial.

Note: Remember that, all the input ports must be connected in ‘port map’ whereas connections with output ports are optional e.g. in line 13, ‘eq=>s0’ is optional, if we do not need the output ‘eq’ in the current design, then we can skip this declaration. But ‘x’ and ‘y’ are the input ports, therefore these connection can not be skipped in port mapping.

2.3.3 Behavioral modeling

In behavioral modeling, the ‘process’ keyword is used and all the statements inside the process statement execute sequentially, and known as ‘sequential statements’. Various conditional and loop statements can be used inside the process block as shown in Listing 2.6. Further, process blocks are concurrent blocks, i.e. if an architecture body contains multiple process blocks (see Listing 2.7), then all the process blocks will execute in parallel.

Explanation Listing 2.6: Behavioral modeling

Entity is declared in line 6-11 which is same as previous codes. In architecture body, the ‘process’ block is declared in line 15, which begins and ends at line 16 and 22 respectively. Therefore all the statements between line 16 to 22 will execute sequentially and Quartus Software will generate the design based on the sequences of the statements. Any changes in sequences will result in different design.

The ‘process’ keyword takes two argument in line 15 (known as ‘sensitivity list’), which indicates that the process block will be executed if and only if there are some changes in ‘a’ and ‘b’. In line 17-21, the ‘if’ statement is declared which sets the value of ‘eq’ to 1 if both the bits are equal (line 17-18), otherwise ‘eq’ will be set to 0 (line 19-20). Fig. 2.6 shows the design generated by the Quartus Software for this listing. ‘=’ in line 17 is one of the condition operators, which are discussed in detail in Chapter 3. Unlike python, we can not interchange single (‘’) and double quotation mark (‘”’); single quotation is used for 1-bit (i.e. ‘1’), whereas double quotation is used for more than one bits (i.e. “101”) e.g. if we use double quotation in line 18, then it will generate error during compilation.

Listing 2.6: Behavioral modeling

```vhdl
--comparator2BitProcess.vhd

library ieee;
use ieee.std_logic_1164.all;

entity comparator2BitProcess is
    port(
        a, b : in std_logic_vector(1 downto 0);
        eq : out std_logic
    );
end comparator2BitProcess;
```

(continues on next page)
2.3. Modeling styles

(continued from previous page)

architecture procEx of comparator2BitProcess is
begin
  process(a,b)
  begin
    if (a(0)=b(0)) and (a(1)=b(1)) then
      eq <= '1'; -- "1" is wrong; as ' and " has different meaning
    else
      eq <= '0';
    end if;
  end process;
end procEx;

Fig. 2.6: 2 bit comparator, Listing 2.6

2.3.4 Mixed modeling

We can mixed all the modeling styles together as shown in Listing 2.7. Here two process blocks are used in line 16 and 25, which is the behavior modeling style. Then in line 34, dataflow style is used for assigning the value to output variable ‘eq’.

Explanation Listing 2.7: Mixed modeling

Entity is declared in line 6-11, which is same as previous listings. Two process blocks are used here. Process block at line 16 checks whether the LSB of two numbers are equal or not; if equal then signal ‘s0’ is set to 1 otherwise it is set to 0. Similarly, the process block at line 25, sets the value of ‘s1’ based on MSB values. Lastly, line 34 sets the output ‘eq’ to 1 if both ‘s0’ and ‘s1’ are 1, otherwise it is set to 0. The design generated for this listing is shown in Fig. 2.7.

Listing 2.7: Behavioral modeling with multiple ‘process’ statements

(continues on next page)
Chapter 2. Overview

(continued from previous page)

19 s0 <= '1';
else
 s0 <= '0';
end if;
end process;

process(a,b)
begin
 if (a(1)=b(1)) then
 s1 <= '1';
 else
 s1 <= '0';
 end if;
end process;

ep <= s0 and s1;
end procEx2;

---

2.4 Packages

If certain declarations are used frequently, e.g. components and functions etc., then these declaration can store in 'packages' as shown in Listing 2.8. After this, we can import these declaration in the design as shown in Listing 2.9, where the design in Listing 2.5 is rewritten using packages.

Explanation Listing 2.8: Package declaration

We define the component ‘compare1Bit’ in Listing 2.5 for structure modeling. Suppose this component declaration is used at various other designs as well, then it’s better to store it in the ‘package’ and call the package in the designs; instead of rewriting the component-declaration in all the designs.

In Listing 2.8, the package is defined with name ‘packageEx’ (line 6) and inside this package the component ‘compare1Bit’ is defined (line 7-12), which is exactly same as Listing 2.5.

Listing 2.8: Package declaration

Meher Krishna Patel
Explanations

**Listing 2.9**

Next, we need to call the package (defined in Listing 2.8) to the current design, which can be done as shown in line 6 of Listing 2.9. This line includes the packageEx in the current design. The only difference between Listing 2.9 and Listing 2.5 is the component declaration at line 14-19 in Listing 2.5, which is not done in Listing 2.9, as it is imported from package at line 6. The final design generated is shown in Fig. 2.8, which is exactly same as Fig. 2.5.

Listing 2.9: Using Packages

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use work.packageEx.all;

entity comparator2BitPackage is
  port(
    a, b : in std_logic_vector(1 downto 0);
    eq : out std_logic);
end comparator2BitPackage;

architecture structure of comparator2BitPackage is
  signal s0, s1: std_logic;
begin
  eq_bit0: comparator1Bit
    port map (x=>a(0), y=>b(0), eq=>s0);
  eq_bit1: comparator1Bit
    port map (x=>a(1), y=>b(1), eq=>s1);
  eq <= s0 and s1;
end structure;
```

Fig. 2.8: 2 bit comparator, Listing 2.9

### 2.5 Conclusion

In this tutorial, various features of VHDL designs are discussed briefly. We designed the two bit comparator with four modeling styles i.e. dataflow, structural, behavioral and mixed styles. Also, differences between the generated-designs with these four methods are shown. Lastly, packages are discussed to store the common declaration in the designs.
Chapter 3

Data types

3.1 Introduction

In the Chapter 2, we used the data-types i.e. ‘std_logic’ and ‘std_logic_vector’ to define ‘1-bit’ & ‘2-bit’ input and output ports and signals. Also, some operators e.g. ‘and’ and ‘or’ etc. were discussed. In this chapter, some more information is provided on these topics.

3.2 Lexical rules

VHDL is case insensitive language i.e. upper and lower case letters have same meanings. Further, 1-bit numbers are written in single quotation mark and numbers with more than 1-bit are written in double quotation mark, e.g. ‘0’ and “01” are the valid notations. Also, VHDL is free formatting language (i.e. spaces can be added freely), but we use the python like approach to write the codes, as it is clear and readable. Lastly in VHDL, ‘–’ is used for comments.

3.3 Library and packages

In the tutorials, we use only two packages i.e. ‘std_logic_1164’ and ‘numeric_std’ packages, which are approved by IEEE. Further, There are various other non-IEEE-standard packages available e.g. std_logic_arith etc., which allow quick and easy coding with VHDL. Since these are not standardized library therefore it may result in compatibility issues in future.

3.3.1 ‘std_logic_1164’ package

The ‘std_logic_1164’ package contains the various data types e.g. ‘std_logic’, ‘std_logic_vector’ and ‘integer’ etc. But we can not control the size of the integer (default 32 bit size) and format (i.e. sign and unsigned) using this package. Further, ‘Natural’ data type is available in this package, which allows only ‘0’ and positive integer values.

3.3.2 ‘numeric_std’ package

We can not perform various mathematical operations on the data type which are defined in ‘std_logic_1164’ package. To perform various mathematical operations and to gain more control over integer values, ‘numeric_std’ package can be used. This package allows ‘sign’ and ‘unsigned’ integer values along with the size control. For example, if integer has values from 0-7 only, then we can define ‘unsigned’ data type of width 3 as shown in Listing 3.7.
3.3.3 ‘textio’ and ‘std_logic_textio’ packages

These packages are used in testbenches to write and read the data to the file. These packages are discussed in Chapter 10.

3.3.4 Other standard packages

There are some more standard packages which are not used in this tutorial i.e. ‘numeric_bit’, ‘standard’,‘numeric_bit_unsigned’ and ‘numeric_std_unsigned’. Further, there are various standard packages are available for ‘fixed point’ and ‘floating point’ operations as well (not discussed in the tutorial) e.g.‘fixed_pkg’, ‘float_pkg’, ‘fixed_float_types’, ‘fixed_generic_pkg’ and ‘float_generic_pkg’.

3.4 Entity and architecture

Although Entity and Architecture declarations are discussed in Chapter 2; but following are the some additional information about these declarations,

3.4.1 Entity declaration

Entity can have three types of ports i.e. ‘in’, ‘out’ and ‘inout’ as shown in lines 3-6 in Listing 3.1. Note that last declaration does not contain ‘;’ at the end (see line 6). Also, types with different port names, can be defined in separate lines as shown in line 3-4. This is helpful for writing comments for different ports.

Listing 3.1: Entity Declaration

```vhdl
entity entityEx is
  -- entityEx is the name of entity
  port(
    a, b : in std_logic;  -- inputs for encoder
    c : in std_logic;    -- input for decoder
    d : inout std_logic; -- inout
    y, z : out std_logic  -- out (no ';' in the last declaration)
  );
end entityEx;  -- end entity declaration
```

3.4.2 Architecture body

Architecture declaration consists of name of the architecture and name of the entity as shown in line 1 of Listing 3.2. It contains two parts i.e. ‘declaration section’ and ‘body’. Declaration section is optional which is defined between architecture name and ‘begin’ statement as shown in line 2. In declaration section signals, variables and constants etc. can be defined (line 2), whereas design-logics are defined in body (line 4-7). We already see the use of signals and design-logics in Chapter 2; variables, constants are other data types are discussed in next section.

Listing 3.2: Architecture body

```vhdl
architecture arch_Name of entityEx is
  signal s0, s1: std_logic;  -- declaration section
begin  -- begin architecture
  s0 <= (not a) and (not b);
  s1 <= a and b;
  z <= s0 or s1;
end arch_Name;  -- end architecture
```
3.5 Keyword ‘others’, ‘downto’ and ‘to’

Keyword ‘others’ is very useful for making assignments. In the tutorial, we used others for assigning the initial values as zero e.g. Listing 5.4 and Listing 11.1 etc. but it can be used for assigning different values as well as shown in Listing 3.3. Also, we already used ‘downto’ keyword at several places e.g. ‘std_logic_vector(7 downto 0)’, which indicates that ‘7th’ bit is the MSB (preferred style). But, if we want to have the 0th bit as MSB, then ‘to’ keyword can be used as ‘std_vector(0 to 7)’. Various usage of keywords ‘others’, ‘downto’ and ‘to’ are shown in Listing 3.3.

Listing 3.3: Keywords ‘others’, ‘downto’ and ‘to’

1: -- assign zero to constant 'a' i.e a = 0000
2: constant a : std_logic_vector(3 downto 0) := (others => '0');
3: -- assign signal 'b' = "0011"
4: -- position 0 or 1 = 1, rest 0
5: signal b : std_logic_vector(3 downto 0) := (0|1=>'1', others => '0');
6: -- assign signal 'c' = "11110000"
7: -- position 7 downto 4 = 1, rest 0
8: signal c : std_logic_vector(7 downto 0) := (7 downto 4 => '1', others => '0');
9: -- or
10: signal c : std_logic_vector(7 downto 0) := (4 to 7 => '1', others => '0');
11: -- assign signal 'd' = "010000"
12: -- note that d starts from 1 and ends at 5, hence position 2 will be 1.
13: signal d : std_logic_vector(1 to 5) := (2=>'1', others => '0');

3.6 Data objects

In this section three types of data-objects are discussed.

- **Signal**: Signals can be seen as the intermediate connections for different ports. We already saw various signals in Chapter 2, which are used to design the 1-bit and 2-bit comparators e.g. in Listing 2.2, the signal ‘s0’ and ‘s1’ are used for 1-bit comparator.
- **Variable**: Variables are defined inside the process statements only and can be accessed within the ‘process’ (i.e. the process in which it is defined). The difference between ‘variable’ and ‘signal’ are shown in Listing 3.4. Variables can be very useful in sequential designs, as these are visible only inside the process.

**Explanation Listing 3.4**

Variables can defined declaration part of the process i.e. between ‘process’ and ‘begin’ keywords as shown in Line 25. Also, value to the variable are assigned using ‘:=’ (see Lines 25 and 28). In the listing, two processes are defined (Lines 15 and 23). Following points are important to note in these processes,

- Process at Line 15 updates the value of signal ‘a’ and then assign the value of ‘a’ to ‘b’. The updated value of ‘a’ is assigned to ‘b’ in the next clock cycle (not in the same clock cycle) as shown in Fig. 3.1.
- Process at Line 23 updates the value of ‘variable v’ and then assign the value to signal ‘c’. This time value will be assigned to signal immediately as shown in Fig. 3.1. **Note that, the variables are not shown in the simulation graph directly, we need to add them manually as shown in the figure. Since variable is declared at line 24, therefore line 24 is added to waveform as illustrated in the figure.**
- Further, variables are not visible outside the process i.e. these can be excessed within the process; hence Line 34 will generated error.
- Lastly, the variable can be shared between between processes by defining them in the ‘architecture declaration part’ using keyword ‘**shared variable**’. In this case, only one process can modify it and other process can use the variable.
3.6. Data objects

Note: Remember:

- If a ‘signal’ is updated inside the process and then assigned to other signal or ports etc., then ‘old value’ of the signal will be assigned. The updated value will appear in next clock cycle.
- Whereas, if a ‘variable’ is updated inside the process and then assigned to other signal or ports etc. then ‘updated value’ will be assigned.
- Further, we will use ‘variables’ only for the ‘testbenches’ (not for the synthesis).
- Lastly, avoid update and assignment of signals within the same process block. This can be done by defining two different signals for updating and assignments, which is discussed with several examples in Chapter 9 and Section 11. In these chapters, suffix ‘_reg’ and ‘_next’ are used for assignment and updating the signal respectively.

### Fig. 3.1: Signal vs Variable

### Listing 3.4: Signal vs Variable

```vhdl
-- signal_variable.vhd
library ieee;
use ieee.std_logic_1164.all;

entity signal_variable is
    port (clk : in std_logic;
          dout1 : out std_logic);
end entity;

architecture arch of signal_variable is
begin
    process(clk)
    begin
      if (clk'event and clk='1') then
        a <= '1';
        b <= a; -- b will be '1' in next clock cycle
      end if;
    end process;

    process(clk)
    begin
      variable v : std_logic := '0';
      if (clk'event and clk='1') then
        v := '1';
        c <= v; -- c will be '1' immediately
      end if;
    end process;

    dout1 <= c; -- signal can be used anywhere in the code
    -- -- error : variable can not be used outside it's process block
    -- dout1 <= v;
```

(continues on next page)
• **Constant**: Constants are the names which are provided to a specific data-type with some values as shown in Section 3.11.1. As the name suggests, the value of ‘constant’ can not be changed after defining it. Further, constants can be defined without defining it’s value in the packages only, such constants are known as ‘deferred constants’. But value of the constants must be declared in the package body.

### 3.7 Data types

In this section, commonly used data-types are discussed. Data types can be categorized in various ways e.g. standard/user-defined data type, scalar/composite and synthesizable/non-synthesizable etc. We categorize the data type in five ways i.e. **Standard data types**, **user-defined scalar data types**, **user-defined composite data types**, **file type** and **access type**. Access type are the pointers to the object of other type (similar to C pointer). Since access types are not used in the designs therefore it is not discussed in the tutorial. Note that we can implement all kinds of complex-designs without using access types.

#### 3.7.1 Standard data type

Some of the widely used standard data types are listed below, which are defined in two packages i.e. ‘std_logic_1164’ and ‘numeric_std’.

- std_logic
- std_logic_vector
- unsigned
- signed
- integer
- natural
- time
- string

We will use the above data types for synthesis purpose except ‘time’ and ‘string’ which will be used in testbenches. Also, these data types can be converted from one type to another (except time and string) as shown in Section 3.10.

Lastly, Table Table 3.1 shows the list of synthesizable and non-synthesizable data types along with the packages in which those data types are defined.

<table>
<thead>
<tr>
<th>Category</th>
<th>Data type</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Synthesizable</strong></td>
<td>bit, bit_vector</td>
<td>standard library (automatically included, no need to include explicitly)</td>
</tr>
<tr>
<td></td>
<td>boolean, boolean_vector</td>
<td>std_logic_1164</td>
</tr>
<tr>
<td></td>
<td>integer, natural, positive, integer_vector</td>
<td>std_logic_1164</td>
</tr>
<tr>
<td></td>
<td>character, string</td>
<td>std_logic_1164</td>
</tr>
<tr>
<td></td>
<td>std_logic, std_logic_vector</td>
<td>numeric_std</td>
</tr>
<tr>
<td></td>
<td>signed, unsigned</td>
<td>textio</td>
</tr>
<tr>
<td><strong>Non-synthesizable</strong></td>
<td>real, real_vector</td>
<td>standard library (automatically included, no need to include explicitly)</td>
</tr>
<tr>
<td></td>
<td>time, time_vector</td>
<td>standard library (automatically included, no need to include explicitly)</td>
</tr>
<tr>
<td></td>
<td>delay_length</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read, write, line, text</td>
<td></td>
</tr>
</tbody>
</table>

#### 3.7.2 User-defined scalar types

In the tutorials, we will use two user-defined scalar types i.e. **integer** and **enumerated**. Since other two scalar types (i.e. **physical** and **floating point**) are not synthesizable, therefore not discussed in the tutorial.
• **Integer Types**: Custom integer ranges can be defined using ‘range’ keywords as shown in Listing 3.5. Note that, all the mathematical operations can be performed on ‘custom integer type’ but the signals must be of same type e.g. in Listing 3.5 ‘a, b, c and v3’ have same range, but ‘v3’ has different type, therefore Line 37 will generate error. Please see all the comments in the listing for better understanding of the scaler type.

• **Enumerated Types**: Enumerated data types have a set of user defined values, as shown in Listing 3.5. Enumerated data types are very useful in sequential designs, where we use the finite state machines for implementation, which is discussed in Chapter 9.

**Explanation Listing 3.5**

This listing contains the example of ‘integer’ and ‘enumerated’ data types. Since ‘out’ port is not readable, therefore in line 11, ‘x’ is define as ‘inout’ port (instead of out), because we are reading the ‘x’ in line 48 i.e ‘if (x >= 0)’.

The signals ‘a’, ‘b’ and ‘c’ are of user-defined integer type whose range is [0,5]. Further, two more integer types are defined at Lines 19-20 and 23-24. Note that first we need to define the ‘type (Lines 19 and 23)’ and then create the signal of that type (Lines 20 and 24).

Enumerated data type is defined in line 27 with name ‘stateTypes’, which has two values i.e. ‘posState’ and ‘negState’. Further, stateTypes, posState and negState are the user-defined name (not the keywords). Then in line 28, the signal ‘currentState’ of type ‘stateTypes’ is declared. Hence, ‘currentState’ can have only two values i.e. posState and negState.

Process block at line 46 executes whenever there is any event on ‘x’. Line 48 checks whether x is greater than or equal to 0 and set the value to ‘posState, if condition is true; otherwise value is set to ‘negState’. Please read the comments for more details of the listing.

**Listing 3.5: Scalar data types**

```vhdl
-- scalarTypeEx.vhd
library ieee;
use ieee.std_logic_1164.all;

entity scalarTypeEx is
port(
    clk : in std_logic;
    a, b : in integer range 1 to 5; -- input values can be 1 to 5 only
    c : out integer range 1 to 5;
    x : inout integer;
    pA : out integer -- output for process block
);
end scalarTypeEx;

architecture arch of scalarTypeEx is
-- integer type : range 1 to 5
type voltage_range is range 1 to 5; -- user-defined integer data type
signal v1, v2, v3, v4 : voltage_range := 1; -- signal of user-defined integer data type

-- integer type : range 10 to 0
type time_range is range 10 downto 0; -- user-defined integer data type
signal t1, t2 : time_range := 0; -- signal of user-defined integer data type

-- enumerated type
type stateTypes is (posState, negState); -- enumerate data type
signal currentState : stateTypes; -- signal of enumerate data type

begin
-- ########## Integer Example ##########
v1 <= 3;
-- v2 <= 7; -- error : outside range
```

(continues on next page)
3.7.3 User-defined composite types

The composite data types are the collection of values. In VHDL, list with same data types is defined using ‘Array’ keyword; whereas list with different data types is defined using ‘Record’. VHDL examples of array and record are shown in Listing 3.6. Further, random access memory (RAM) is implemented in Section 11.4 using composite type.

Explanation Listing 3.6

In line 18, the array ‘newArray’ is defined which can store 2 values (i.e. 0 to 1) of ‘std_logic’ type. Line 19 creates a signal ‘arrValue’ of newArray type. Then in line 29 and 30, values are assigned to 0th and 1st position of ‘arrValue’ signal. Finally at line 32, the value is assigned to z using & operator. & is known as Concatenation operator, which is discussed in Section 3.9.

Similarly, the record with name ‘newRecord’ is defined in lines 21-25, with 4 items i.e. d1, d2, v1 and v2. In line 26, the signal ‘recordValue’ of newRecord type is defined. Then in lines 35-39, values are assigned to recordValue signal. Finally, ‘and’ operations are performed in lines 41 and 42, which are sent to output ports i.e. rY and rZ.

Simulation results for the listing is shown in Fig. 3.2.
3.7. Data types

(continued from previous page)

```vhdl
architecture arch of compositeTypeEx is

  type newArray is array (0 to 1) of std_logic; -- create Array
  signal arrValue : newArray; -- signal of Array type

  type newRecord is -- create Record
    record
      d1, d2 : std_logic;
      v1, v2: std_logic_vector(1 downto 0);
    end record;
  signal recordValue : newRecord; -- signal of Record type

begin

  -- array example
  arrValue(0) <= a; -- assign value to array
  arrValue(1) <= b;
  z <= arrValue(0) & arrValue(1);

  -- record example
  recordValue.d1 <= a; -- assign value to record
  recordValue.d2 <= b;
  recordValue.v1 <= c;
  recordValue.v2 <= d;
  rY <= recordValue.d1 and recordValue.d2;
  rZ <= recordValue.v1 and recordValue.v2;

end arch;
```

Fig. 3.2: Composite datatypes

3.7.4 File Type

File types are used to read and write contents to files. ‘File type’ is used with testbenches therefore it is discussed in Chapter 10.
3.8 Tristate buffer

The input/output (IO) ports can be left open using ‘Z’ value of ‘std_logic’, which is synthesize using ‘tristate buffer’. Suppose, we want to read the IO port ‘a’ only when ‘enable’ is ‘1’ then we can write the code as below:

\[ r <= a \text{ when enable='1' else 'Z';} \]

3.9 Operators

In this section, various operators are discussed which are shown in Table Table 3.2.

3.9.1 Arithmetic operators

Table Table 3.2 shows various arithmetic operators. Note that, arithmetic operations may not be performed to all data types, e.g. Mod operation can be applied to ‘integer’ data type and it’s subtype (i.e. natural and positive) only. Hence, if we want to use ‘mod’ operation with ‘std_logic_vector’ then we need to perform type-conversion first as shown in Listing 3.8.

3.9.2 Logical or Boolean operators

We already see some logical operators in previous examples e.g. ‘and’ and ‘or’ etc. VHDL provides 7 logical operators i.e. \text{and}, \text{or}, \text{not}, \text{nand}, \text{nor}, \text{xor} and \text{xnor}.

3.9.3 Relational operators

In previous examples, we used various relational operators to check the conditions i.e. ‘=’, ‘>=’. VHDL provides 6 relational operations i.e. ‘=’, ‘>’, ‘<’, ‘>=’ (i.e. greater or equal), ‘<=’ and ‘/=’ (i.e. not equal to).

3.9.4 Concatenation operators

Concatenation operator is used to combine the two or more values e.g. ['1' & '0' = “10”] and ['10' & '0' = “100”]. An example of this operator is shown in line 32 of Listing 3.6. Also, concatenation operators can be used for shifting operations which is discussed in Section 11.3.

3.9.5 Assignment operator

There are three operators which are used to assign values in VHDL i.e. ‘<’, ‘:=’ and ‘=>’. The first two operators are discussed in Section 3.6 whereas last operator is discussed in Section 3.5.
### Table 3.2: VHDL operators

<table>
<thead>
<tr>
<th>Type</th>
<th>Symbol</th>
<th>Description</th>
<th>Example (a = 4 bit, b = 5 bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>+</td>
<td>integer, natural, positive, signed, unsigned</td>
<td>c (exactly 5 bit) = a + b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(if a and b are signed/unsigned)</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td></td>
<td>c (exactly 5 bit) = a - b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(if a and b are signed/unsigned)</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td></td>
<td>c (exactly 9 bit i.e. 4+5) = a * b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(if a and b are signed/unsigned)</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td></td>
<td>c (exactly 5 bit) = a / b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(if a and b are signed/unsigned &amp; ignores the decimal values e.g. 5/-2 = -2)</td>
</tr>
<tr>
<td></td>
<td>abs</td>
<td>integer, natural, positive</td>
<td>abs(a) (absolute value of a)</td>
</tr>
<tr>
<td></td>
<td>rem</td>
<td></td>
<td>a rem b (returns remainder with sign of a)</td>
</tr>
<tr>
<td></td>
<td>mod</td>
<td></td>
<td>a mod b (returns remainder with sign of b)</td>
</tr>
<tr>
<td></td>
<td>**</td>
<td>natural, positive, constant positive integer</td>
<td>a ** b (e.g. (2^{3}=8))</td>
</tr>
<tr>
<td>Boolean</td>
<td>not</td>
<td></td>
<td>not(a)</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td></td>
<td>a or b</td>
</tr>
<tr>
<td></td>
<td>nor</td>
<td></td>
<td>a nor b</td>
</tr>
<tr>
<td></td>
<td>and</td>
<td></td>
<td>a and b</td>
</tr>
<tr>
<td></td>
<td>nand</td>
<td></td>
<td>a nand b</td>
</tr>
<tr>
<td></td>
<td>xor</td>
<td></td>
<td>a xor b</td>
</tr>
<tr>
<td></td>
<td>xnor</td>
<td></td>
<td>a xnor b</td>
</tr>
<tr>
<td>Relational</td>
<td>&gt;</td>
<td>greater than</td>
<td>a &gt; b</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>less than</td>
<td>a &lt; b</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>greater than or equal</td>
<td>a &gt;= b</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>less than or equal</td>
<td>a &lt;= b</td>
</tr>
<tr>
<td></td>
<td>==</td>
<td>equal</td>
<td>a == b</td>
</tr>
<tr>
<td></td>
<td>/=</td>
<td>not equal</td>
<td>a /= b</td>
</tr>
<tr>
<td>Concatenation</td>
<td>&amp;</td>
<td>can be used for shift operations</td>
<td>“0” &amp; “101” = “0101”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>“101” &amp; ‘0’ &amp; “11“= “10101”</td>
</tr>
<tr>
<td>Assign-</td>
<td>&lt;=</td>
<td>assign value to signal</td>
<td>“a &lt;= “1001“</td>
</tr>
<tr>
<td>ment</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>:=</td>
<td>assign value to variable</td>
<td>“b := “1001“</td>
</tr>
<tr>
<td></td>
<td></td>
<td>assign initial values to signal and variable</td>
<td>signal a : std_logic := ‘1’</td>
</tr>
<tr>
<td></td>
<td>=&gt;</td>
<td>assign value using ‘others’</td>
<td>a &lt;= (others =&gt; ‘0’)</td>
</tr>
</tbody>
</table>

### 3.9.6 Shift operators

VHDL provides some shift operators as well e.g. SLL (shift left logic) and SRL (shift right logic) etc. But these operations can be used with ‘bit_vector’ only, therefore these are not discussed in the tutorial. Further, we can use the concatenation operator for shifting operation as discussed in Section 3.9.4.

### 3.10 Type conversion

VHDL is strongly typed language; in the other words, if we declare the two numbers e.g. ‘101’ and ‘111’ using two different data types e.g. ‘std_logic_vector’ and ‘unsigned’, then VHDL considers these numbers as different data types and we can not perform ‘or’ and ‘xor’ etc. operations directly on these two numbers. We need to convert the type of the number, to perform such operations as shown in line 18 of Listing 3.7. Various conversion functions are shown in Table Table 3.3, which is known as ‘Type Casting’. Further, Listing 3.8 performs the type conversion...
for performing the ‘mod’ operation on ‘std_logic_vector’ data type. Note that, in Line 26 of Listing 3.8, the ‘natural data type (i.e. b)’ is first converted into ‘signed’ and then to ‘std_logic_vector’, as no direct conversion is possible.

<table>
<thead>
<tr>
<th>Original Data Type (a)</th>
<th>Convert to</th>
<th>Type Casting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed, std_logic_vector</td>
<td>unsigned</td>
<td>unsigned(a)</td>
<td></td>
</tr>
<tr>
<td>unsigned, std_logic_vector</td>
<td>signed</td>
<td>signed(a)</td>
<td></td>
</tr>
<tr>
<td>signed, unsigned, std_logic_vector</td>
<td>std_logic_vector</td>
<td>std_logic_vector(a)</td>
<td></td>
</tr>
<tr>
<td>signed, unsigned</td>
<td>integer</td>
<td>to_integer(a)</td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>signed</td>
<td>to_signed(a, size)</td>
<td>convert 10 into 5 bit unsigned i.e. 01010</td>
</tr>
<tr>
<td>natural</td>
<td>unsigned</td>
<td>to_unsigned(a, size)</td>
<td></td>
</tr>
</tbody>
</table>

Listing 3.7: Type conversion

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity typeConvertEx is
  port(
    a: in std_logic_vector(2 downto 0);
    b: in unsigned(2 downto 0);
    c: out std_logic_vector(2 downto 0)
  );
end typeConvertEx;

architecture arch of typeConvertEx is
begin
  -- c <= a and b; -- error: as a and b has different data type
  c <= a and std_logic_vector(b); -- type conversion
end arch;
```

Listing 3.8: MOD operation with type conversion

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity modEx is
  port (  
    CLK : in std_logic;
    -- input values are taken from switch
    SW  : in std_logic_vector(3 downto 0);
    -- output value (i.e. SW mod 10) is displayed on LEDG
    LEDG: out std_logic_vector(3 downto 0)
  );
end entity;
```

(continues on next page)
3.11 Constant and Generics

Constant and Generics can be used to create reusable codes, along with avoiding the ‘hard literals’ from the code as shown in following sections.

3.11.1 Constants

Constants are defined in ‘architecture declaration’ part and can not be modified in the architecture-body after declaration. In Listing 3.9, constant ‘N’ is defined in line 14 with value 3. Then this value is used in line 15 and 16. Suppose we want to change the constant value to 4. Now we need to change the value from 3 to 4 in line 14 only (instead of changing everywhere in the code e.g. line 15 and 16 in this example). In this way, we can remove the ‘hard literal’ from the codes.

```
architecture arch of modEx is
    signal a, b : natural := 10;
begin
    b <= to_integer(unsigned(SW)) mod a; -- type conversion to integer
    LEDG <= std_logic_vector(to_signed(b, 4));
end architecture;
```

Listing 3.9: Constants

```
-- constantEx.vhd
library ieee;
use ieee.std_logic_1164.all;

entity constantEx is
    port(
        a, b : in std_logic_vector(3 downto 0);
        z: out std_logic_vector(3 downto 0)
    );
end constantEx;

architecture arch of constantEx is
    constant N : integer := 3; -- define constant
    signal x : std_logic_vector(N downto 0); -- use constant
    signal y : std_logic_vector(2**N downto 0); -- use constant
begin
    -- use x and y here
    z <= a and b;
end arch;
```

3.11.2 Generics

Generics are defined in ‘entity’ and can not be modified in the architecture-body. VHDL code for generic is shown in Listing 3.10. Further, we can override the default value of generic during component instantiation in structural modeling style as shown in Listing 3.11.

Explanation Listing 3.10

In Lines 7-10, two generics are defined i.e. ‘N’ and ‘M’. Then ports ‘a’ and ‘b’ are defined using generic ‘N’. The process block (lines 20-27) compares ‘a’ and ‘b’ and set the value of ‘eq’ to 1 if these inputs are equal, otherwise set ‘eq’ to 0.
Chapter 3. Data types

Listing 3.10: Generics

```vhdl
-- genericEx.vhd

library ieee;
use ieee.std_logic_1164.all;

entity genericEx is
  generic (
    N: integer := 2; -- define more generics here
    M: std_logic := '1'
  );
  port(
    a, b: in std_logic_vector(N-1 downto 0);
    eq: out std_logic
  );
end genericEx;

architecture arch of genericEx is
begin
  process(a,b)
  begin
    if (a=b) then -- compare to numbers
      eq<='1'; -- set eq to 1, if equal
    else
      eq<='0'; -- otherwise 0
    end if;
  end process;
end arch;
```

Explanation Listing 3.11

In line 8, ‘x’ and ‘y’ are defined as 4-bit vector. Structural modeling is used in Line 15-17, where generic mapping and port mapping is done at line 16 and 17 respectively.

Note that, in line 16 N=>4 will override the default value of N i.e. N=2 in Listing 3.10. Also, generic ‘M’ is not mapped, therefore default value of M will be used, which is defined in Listing 3.10. In this way, we can remove ‘hard literals’ from the codes, which enhances the reusability of the designs.
3.12 Attributes

Attributes are the additional information about the signals, variables and types etc. In the tutorial three attributes are used frequently in the tutorial i.e. ‘event (see Listing 8.1)’, ‘image (see Listing 10.4)’ and ‘range (see Listing 10.4).

- Attribute ‘event’ is set to ‘1’ whenever there is any change in the signal e.g. ‘clk’event’ will be 1 when clock goes from ‘0’ to ‘1’ or vice-versa.
- Attribute ‘image’ is used for string representation of integers, natural and std_logic_vector etc., which is very useful in testbenches for printing the values in string format.
- Attribute ‘range’ can be used for iterating over all the test-data using for-loop in the testbenches.
- Lastly, there are several other predefined attributes available in VHDL e.g. ‘low’, ‘high’, ‘active’, ‘length’ and ‘reverse_range’ etc. Further, we can create custom attribute as well.

3.13 Conclusion

In this chapter, we saw various data objects, data types and operators. Further, the two libraries i.e. ‘std_logic_1164’ and ‘numeric_std’ are discussed. Generics and constants are shown which can be useful in creating the reusable designs. Lastly, since VHDL is strongly typed language therefore ‘type casting’ is used for performing operations on two different data types.
Chapter 4

Dataflow modeling

4.1 Introduction

In Chapter 2 and Chapter 3, we saw various elements of VHDL language along with several examples. More specifically, Chapter 2 presented various ways to design the ‘comparator circuits’ i.e. using dataflow modeling, structural modeling and packages etc.; and then Chapter 3 presented various elements of VHDL language which can be used to implement the digital designs. These two chapters are the foundation of the VHDL language, and the illustrated designs-examples were not of any particular usage (especially in Chapter 3). From this chapter we will concentrate on the proper digital design using various elements of VHDL which are described in previous chapters.

In this chapter, differences between ‘combinational designs’ and ‘sequential designs’ are shown. Also, various methods are discussed which are used for designing the ‘combinational circuit’ and ‘sequential circuits’. Main focus of this chapter is the ‘combinational designs’ using ‘Dataflow’ modeling style; in which functionality of the entity is described using ‘concurrent statements’ (without defining the structure of the design); whereas Chapter 5 will present the ‘behavioral modeling’ which can be used to create both ‘sequential’ and ‘combinational’ design.

4.2 Combinational circuit and sequential circuit

Digital design can be broadly categorize in two ways i.e. combinational designs and sequential designs. It is very important to understand the difference between these two designs and see the relation between these designs with various elements of VHDL.

- **Combinational designs**: Combinational designs are the designs in which output of the system depends on present value of the inputs only. Since, the outputs depends on current inputs only, therefore ‘no memory’ is required for these designs. Further, memories are nothing but the ‘flip flops’ in the digital designs, therefore there is no need of ‘flip flops’ in combination designs. In the other words, only ‘logic gates (i.e. and, not and xor etc.)’ are required to implement the combinational designs.

- **Sequential designs**: Sequential designs are the designs in which the output depends on current inputs and previous states of the system. Since output depends on previous states, therefore memories are required for these systems. Hence, in sequential designs the ‘flip flops’ are need along with the logic gates.

**Note:**
- Only ‘logic gates (i.e. and, not and xor etc.)’ are required to implement the combinational designs.
- Both ‘logic gates’ and ‘flip flops’ are required for implementing the sequential designs.
- Lastly, the ‘sequential design’ contains both ‘combinational logic’ and ‘sequential logic’, but the combinational logic can be implement using ‘sequential statements’ only as shown in Fig. 4.1; whereas the ‘combination logic’ can be implement using ‘concurrent or sequential statement’ in the combinational designs. These statements are discussed in Section 4.3.
4.3 Concurrent statements and sequential statements

In Section 2.3.1, we saw that the concurrent statements execute in parallel, i.e. the order of the statement does not matter. Whereas in Section 2.3.3 shows the example of ‘sequential statements’ where the statements execute one by one. Following are the relationship between ‘statements’ and ‘design-type’, which are illustrated in Table 4.1 as well.

**Note:**
- Remember, the ‘sequential statements’ and the ‘sequential designs’ are two different things. Do not mix these together.
- Combinational designs can be implemented using both ‘sequential statements’ and ‘concurrent statements’.
- Sequential designs can be implemented using ‘sequential statements’ only.
- Sequential statements can be defined inside ‘process’, ‘function’ and ‘procedure’ block only. Further, these blocks executes concurrently e.g. if we have more than one process block then these block will execute in parallel, but statements inside each block will execute sequentially.
- VHDL constructs for combinational designs are ‘when-else’, ‘with-select’ and ‘generate’, which are discussed in this chapter.
- VHDL constructs for sequential designs are ‘if’, ‘loop’, ‘case’ and ‘wait’, which are discussed in Chapter 5.

**Table 4.1: Relationship between ‘design-type’ and ‘VHDL statements’**

<table>
<thead>
<tr>
<th>Design</th>
<th>Statement</th>
<th>VHDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>Sequential statements</td>
<td>If</td>
</tr>
<tr>
<td>(Flip-flops and logic gates)</td>
<td></td>
<td>Case</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Wait</td>
</tr>
<tr>
<td>Combinational</td>
<td>Concurrent and Sequential</td>
<td>when-else</td>
</tr>
<tr>
<td>(only logic gates)</td>
<td></td>
<td>with-select</td>
</tr>
<tr>
<td></td>
<td></td>
<td>generate</td>
</tr>
</tbody>
</table>

---

Fig. 4.1: Block diagram of ‘combinational’ and ‘sequential’ designs
4.4 Delay in signal assignments

In Chapter 2, we saw that concurrent statements do not execute sequentially i.e. order of statements do not matter in concurrent statements. More precisely, these statements are ‘event triggered’ statements i.e. these statements execute whenever there is any event on the signal, as discussed in Listing 4.1 and Listing 4.2.

Remember that, lines 15 and 16 in Listing 4.1 and Listing 4.2 are the example of signal assignments. In this section, two types of delays in signal assignments, i.e. ‘delta delay ($\Delta t$)’ and ‘delay using ‘after’ statement’, are discussed.

4.4.1 Delta delay

If ‘no delay’ or delay of ‘0 ns’ is specified in the statement, then delta delay is assumed in the design. These two delays are shown in Listing 4.1 which is explain below.

Explanation Listing 4.1

In this listing, ‘x’ and ‘z’ are the input and output ports respectively, whereas ‘s’ is the signal. In line 15, delay is not defined explicitly; whereas in Line 16, 0 ns delay is defined. Hence, in both the cases ‘delta delay ($\Delta t$)’ is assumed which is explained in next paragraph.

As explained before, concurrent statements execute whenever there is any change in the signals; therefore line 16 will be executed, when there is any change in the input ‘x’. This statement will execute in $\Delta t$ time. Hence, the value of signals ‘s’ will be changed after $\Delta t$ time. Since the value of ‘s’ is changed, therefore in next $\Delta t$ time, line 15 will execute and the value of ‘z’ will be changed.

In the other words, the code will execute ‘two times’ in $2\Delta t$ time to complete the signal assignments; in first $\Delta t$ time, value of ‘s’ will change and in next $\Delta t$ time, the value of ‘s’ will be assigned to ‘z’. In general, code will execute until there is no further change in the signals. Also, delta delay is very small and can not be seen in simulation results as illustrated in Fig. 4.3. Further, in the figure, all the values i.e. ‘x’, ‘s’ and ‘z’ are changed at the same time, which indicates that delta delay is very small and can not be observed.

Note:
- Multiple signal assignments are not allowed in VHDL, as shown in line 17 of Listing 4.1. Here, signal assignment for ‘z’ is done twice i.e. line 15 and 17. If we uncomment the line 17, then model will be invalid and error will be generated.
- The ‘after’ statements in the VHDL codes are discarded by the synthesizer. But ‘after’ keyword is very useful for writing testbenches.
- A buffer is added to the system for ‘inout port’ as shown in Fig. 4.2.

Fig. 4.2: Buffer created for ‘inout port (x)’

Listing 4.1: Delta delay

```vhdl
--deltaDelayEx.vhd
library ieee;
use ieee.std_logic_1164.all;
```

(continues on next page)
4.4. Delay in signal assignments

(entity deltaDelayEx is
  port( x: inout std_logic;
       z: out std_logic
  );
end deltaDelayEx;

architecture dataflow of deltaDelayEx is
signal s : std_logic;
begun
  z <= s;
  s <= x after 0 ns;
  -- z <= z and s; -- error: multiple signal assignment not allowed
end dataflow;

4.4.2 Delay with ‘after’ statement

We can assign the desired delay in the signal assignment statement as shown in lines 15 and 16 of Listing 4.2. Here signal ‘s’ is assigned after the delayed of 1 ns (line 15), then the value of ‘s’ is assign to ‘z’ after 1 ns (line 16). Hence the output ‘z’ will be delay by 2 ns with respect to ‘x’ as shown in Fig. 4.4 using vertical red lines (which show the propagation of value ‘1’). Also, horizontal red lines for ‘s’ and ‘z’ show that these signals are uninitialized for that time period (because values are assigned after 1 and 2 ns respectively).
Chapter 4. Dataflow modeling

Fig. 4.4: Delay

Listing 4.2: Specify delay using ‘after’

```vhdl
--afterDelay.vhd

library ieee;
use ieee.std_logic_1164.all;

entity afterDelay is
  port( x : in std_logic;
       z: out std_logic
  );
end afterDelay;

architecture dataflow of afterDelay is
  signal s: std_logic;
begin
  z <= s after 1 ns;
  s <= x after 1 ns;
end dataflow;
```

4.5 Concurrent signal assignments

Similar to ‘if’ statement in behavioral modeling, the ‘dataflow’ modeling provides two signal assignments i.e. ‘Conditional signal assignment’ and ‘Selected signal assignment’. In this chapter, the multiplexer is designed using these two assignments.

Multiplexer is a combinational circuit which selects one of the many inputs with selection-lines and direct it to output. Table 4.2 illustrates the truth table for $4 \times 1$ multiplexer. Here ‘i0 - i3’ the input lines, whereas ‘s0’ and ‘s1’ are the selection line. Base on the values of ‘s0’ and ‘s1’, the input is sent to output line, e.g. if s0 and s1 are 0 then i0 will be sent to the output of the multiplexer.

<table>
<thead>
<tr>
<th>s0</th>
<th>s1</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>i0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>i1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>i2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>i3</td>
</tr>
</tbody>
</table>

4.5.1 Conditional signal assignment

Syntax for conditional signal assignment is shown in lines 15-18 of Listing 4.3, where ‘when’ and ‘else’ keywords are used for assigning the value to output port ‘y’. Conditional signal assignments are implemented using ‘2$times$1’ multiplexer as shown in Fig. 4.5. Here, three ‘2$times$1’ multiplexer (i.e. y:$math:\sim'0$, y:$math:\sim'2$ and y:$math:\sim'4$) are used to design the ‘$math:\sim'4$times 1’ multiplexer.
Listing 4.3: Multiplexer using Conditional signal assignment

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity multiplexerEx is
  port( s: in std_logic_vector(1 downto 0);
        i0, i1, i2, i3: in std_logic;
        y: out std_logic
      );
end multiplexerEx;

architecture dataflow of multiplexerEx is
begin
  y <= i0 when s = "00" else
       i1 when s = "01" else
       i2 when s = "10" else
       i3 when s = "11";
end dataflow;
```

Fig. 4.5: Multiplexer using Conditional signal assignment

Fig. 4.6 shows the waveform of Listing 4.3. Here i0, i1, i2 and i3 are set to 1, 0, 1 and 0 respectively. Then simulator is run 4 times for all the combination of ‘s’ i.e. 00, 01, 10 and 11 respectively; and output ‘y’ is assigned the value based on the ‘s’ value e.g if s is ‘00’ then y is assigned with the value of ‘i0’ i.e. 1 etc.

Fig. 4.6: Multiplexer waveform for Listing 4.3 and Listing 4.4

4.5.2 Selected signal assignment

Syntax for selected signal assignment is shown in lines 15-23 of Listing 4.4, where ‘with-select’ and ‘when’ keywords are used for assigning the value to output port ‘y’. Unlike conditional signal statement, in selected signal assignment implements the $4 \times 1$ multiplexer with $4 \times 1$ multiplexer itself as shown in Fig. 4.7, rather than with multiple $2 \times 1$ multiplexer. Further, the waveforms for Listing 4.4 is shown in Fig. 4.6.
Note that ‘unaffected when others’ is used in line 23. ‘others’ is used to avoid the error which is caused by the logics which can not be synthesized, as mentioned in comments in lines 20-22. Further, ‘unaffected’ is the keyword which does not allow any change in the signal.

Fig. 4.7: Multiplexer using Selected signal assignment

Listing 4.4: Multiplexer using Selected signal assignment

```vhdl
--multiplexerVhdl.vhd

library ieee;
use ieee.std_logic_1164.all;

entity multiplexerVhdl is
  port( s: in std_logic_vector(1 downto 0);
        i0, i1, i2, i3: in std_logic;
        y: out std_logic
  );
end multiplexerVhdl;

architecture dataflow of multiplexerVhdl is
begin
  with s select
    y <= i0 when "00",
     i1 when "01",
     i2 when "10",
     i3 when "11",
     -- use 'when others' to avoid modelsim error
     -- i.e. 'cover only 4 out of 81 cases', which occurs due to
     -- the logics which can not be synthesized e.g. Forcing unknown 'X' etc.
     unaffected when others;
end dataflow;
```

4.6 Generate statement and problem with Loops

‘Generate statement’ is used for creating loops in concurrent assignments. These loops are very different from software loops. Suppose ‘for i = 1 to N’ is a loop, then, in software ‘i’ will be assigned one value at a time i.e. first i=1, then next cycle i=2 and so on. Whereas in VHDL, N logic will be implement for this loop, which will execute in parallel. Also, in software, ‘N’ cycles are required to complete the loop, whereas in VHDL the loop will execute in one cycle.

**Note:** As loops implement the design-units multiple times, therefore design may become large and sometimes can
not be synthesized as well. If we do not want to execute everything in one cycle (which is almost always the case),
then loops can be replaced by ‘case’ statements and ‘conditional’ statements as shown in Section 5.7. Further, due
to these reasons, we do not use loops in the design, and hence these are not discussed in the tutorial.

4.7 Conclusion

In this chapter, we saw various features of Dataflow modeling style. We discussed the delays in VHDL designs.
Also, 4 × 1 multiplexer is implemented using conditional and selected signal assignments. Further, the differences
in the designs generated by these two assignments are shown using figures.
Chapter 5

Behavioral modeling

5.1 Introduction

In Chapter 2, 2-bit comparator is designed using behavior modeling. In that chapter, ‘if’ keyword was used in the ‘process’ statement block. This chapter presents some more such keywords.

5.2 Process block

All the statements inside the process block execute sequentially. Further, if the architecture contains more than one process block, then all the process blocks execute in parallel, i.e. process blocks are the concurrent blocks. Also, if a signal is assigned values multiple times, then only last assignments will be considered as shown in Listing 5.1. In the listing, value to port ‘z’ is assigned at Lines 19 and 21. In this case, last assignment will be considered i.e. ‘and’ gate will implemented by Line 21, as shown in Fig. 5.1

Listing 5.1: Multiple assignments to same signal

```vhdl
-- deltaDelayEx2.vhd
library ieee;
use ieee.std_logic_1164.all;

entity deltaDelayEx2 is
  port( x : inout std_logic;
       z : out std_logic);
end entity;

architecture arch of deltaDelayEx2 is
  signal s : std_logic;
begin
  process(x)
  begin
    -- this line has no effect as value to z is assign below again
    z <= s; -- ignore by the compiler
    s <= x after 2 ns;
    z <= x and s; -- only last assignment will be considered
  end process;
end arch;
```

Note:
5.3 If-else statement

In this section, 4 × 1 multiplexed is designed using If-else statement. We already see the working of ‘if’ statement in Chapter 2. In lines of 17-27 of Listing 5.2, ‘elsif’ and ‘else’ are added to ‘if’ statement. Note that, If-else block can contain multiple ‘elsif’ statements between one ‘if’ and one ‘else’ statement. Further, ‘null’ is added in line 26 of Listing 5.2, whose function is same as ‘unaffected’ in concurrent signal assignment as shown in Listing 4.4. Fig. 5.3 shows the waveform generated by Modelsim for Listing 5.2.

Note:
- The ‘multiplexer design’ in Fig. 5.2 (generated by if-else in Listing 5.2) is exactly same as the design in Fig. 4.5 (generated by when-else in Listing 4.3).
- Further, in Section 4.3, it is said that the ‘combinational logic can be implemented using both the concurrent statements and the sequential statements. Note that, the multiplexer is the ‘combinational design’ as the output depends only on current input values. And it implemented using ‘concurrent statements’ and ‘sequential statements’ in Listing 4.3 and Listing 5.2 respectively.

```
--ifEx.vhd
library ieee;
use ieee.std_logic_1164.all;

entity ifEx is
    port( s: in std_logic_vector(1 downto 0);
         i0, i1, i2, i3: in std_logic;
         y: out std_logic
    );
end ifEx;

architecture behave of ifEx is
begin
    process(s)
```
Chapter 5. Behavioral modeling

(continued from previous page)

begin
  if s = "00" then
    y <= i0;
  elsif s = "01" then
    y <= i1;
  elsif s = "10" then
    y <= i2;
  elsif s = "11" then
    y <= i3;
  else
    null;
  end if;
end process;
end behave;

5.4 Case statement

Case statement is shown in lines 17-28 of Listing 5.3. ‘s’ is defined in case statement at line 17; whose value is checked using ‘when’ keyword at lines 18 and 20 etc. The value of ‘y’ depends on the value of ‘s’ e.g. if ‘s’ is “01”, then line 20 will be true, hence value of ‘i1’ will be assigned to ‘y’.

Note: Note that the ‘multiplexer design’ in Fig. 5.4 (generated by case in Listing 5.3) is exactly same as the design in Fig. 4.7 (generated by with-select in Listing 4.4).
5.5. Wait statement

‘Wait statement’ is used to hold the system for certain time duration. It can be used in three different ways as shown below,

- **wait until**: It is *synthesizable* statement and holds the system until the defined condition is met e.g. ‘wait until clk = ‘1’’ will hold the system until clk is ‘1’.

---

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity caseEx is
    port( s: in std_logic_vector(1 downto 0);
         i0, i1, i2, i3: in std_logic;
         y: out std_logic );
end caseEx;

architecture behave of caseEx is
begin
    process(s)
    begin
        case s is
        when "00" =>
            y <= i0;
        when "01" =>
            y <= i1;
        when "10" =>
            y <= i2;
        when "11" =>
            y <= i3;
        when others =>
            null;
        end case;
    end process;
end behave;
```

---

Fig. 5.4: Multiplexer using case statement, Listing 5.3
Chapter 5. Behavioral modeling

- **wait on**: It is *synthesizable* statement and holds the system until the defined signal is changed e.g. ‘wait on clk’ will hold the system until clk changes its value i.e. ‘0’ to ‘1’ or vice-versa.
- **wait for**: It is *not synthesizable* and holds the system for the defined timed e.g. ‘wait for 20ns’ will hold the system for 20 ns. This is used with testbenches as shown in Chapter 10.

5.6 Problem with Loops

VHDL provides two loop statements i.e. ‘for’ loop and ‘while’ loop’. These loops are very different from software loops. Suppose ‘for i = 1 to N’ is a loop, then, in software ‘i’ will be assigned one value at a time i.e. first i=1, then next cycle i=2 and so on. Whereas in VHDL, N logic will be implement for this loop, which will execute in parallel. Also, in software, ‘N’ cycles are required to complete the loop, whereas in VHDL the loop will execute in one cycle.

**Note:** As loops implement the design-units multiple times, therefore design may become large and sometimes cannot be synthesized as well. If we do not want to execute everything in one cycle (which is almost always the case), then loops can be replaced by ‘case’ statements and ‘conditional’ statements as shown in Section 5.7. Further, due to these reasons, we do not use loops for the design. Lastly, the loops can be extremely useful in testbenches, when we want to iterate through all the test-data which is shown in Listing 10.4.

5.7 Loop using ‘if’ statement

In **Listing 5.4**, a loop is created using ‘if’ statement, which counts the number upto input ‘x’.

**Explanation Listing 5.4**

In the listing, two ‘process’ blocks are used i.e. at lines 20 and 31. The process at line 20 checks whether the signal ‘count’ value is ‘less or equal’ to input x (line 22), and sets the currentState to ‘continueState’; otherwise if count is greater than the input x, then currentState is set to ‘stopState’.

Then next process statement (line 31), increase the ‘count’ by 1, if currentState is ‘continueState’; otherwise count is set to 0 for stopState. Finally count is displayed at the output through line 39. In this way, we can implement the loops using process statements.

Fig. 5.5 shows the loop generated by the listing with generic value N=1. Further, Fig. 5.6 shows the count-waveforms generated by the listing with generic value N = 3.

Sensitivity list of the process block should be implemented carefully. For example, if we add ‘count’ in the sensitivity list at line 31 of Listing Listing 5.4, then the process block will execute infinite times. This will occur because the process block execute whenever there is any event in the signals in the sensitivity list; therefore any change in ‘count’ will execute the block, and then this block will change the ‘count’ value through line 34. Since ‘count’ value is changed, therefore process block will execute again, and the loop will never exit.

**Listing 5.4: Loop using ‘if’ statement**

```vhdl
--ifLoop.vhd (-- This code is for simulation purpose only)
-- ideally positive or negative clock edge must be used; which will be discussed later.
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ifLoop is
  generic ( N: integer := 3);
  port( clk : in std_logic; -- clock: increase count on each clk
       x: in unsigned(N downto 0); -- count upto x
       z: out unsigned(N downto 0) -- display count )
end entity ifLoop;

architecture ifLoop_arch of ifLoop is
  signal count : unsigned(N downto 0) := (others => '0');
  signal currentState : state := 'continueState';

begin
  -- process 1: check count <= x
  process (clock)
  begin
    if count <= x then
      currentState := 'continueState';
    else
      currentState := 'stopState';
    end if;
  end process;

  -- process 2: increase count
  process (clock)
  begin
    if currentState = 'continueState' then
      count := count + 1;
    else
      count := 0;
    end if;
  end process;

end architecture ifLoop_arch;
```

(continues on next page)
5.7. Loop using ‘if’ statement

(continued from previous page)

```vhdl
architecture behave of ifLoop is

signal count : unsigned(N downto 0) := (others =>'0'); -- count signal
type stateType is (continueState, stopState); -- states to continue or stop the count
signal currentState : stateType;
begin
    process(clk, currentState, count)
    begin
        if (count <= x) then -- check whether count is completed till x
            currentState <= continueState; -- if not continue the count
        else
            currentState <= stopState; -- else stop further count
        end if;
    end process;
end behave;
```

Fig. 5.5: Loop using ‘if’ statement, Listing 5.4 with N = 1

Fig. 5.6: Loop using ‘if’ statement, Listing 5.4 with N = 3

Note: Note that, the design in Listing 5.4 is not good because of following reasons,

- First, the clock is used in the sensitive list, therefore it may arise race-around condition in the system. To avoid it, the system should be sensitive to only edge of the clock (i.e. positive and negative) , which is discussed in Listing 8.1 using ‘event’ keyword.
- Secondly, the process statement in Line 31, does not required the ‘clk’ in the sensitive list. In Fig. 4.1, we saw
Chapter 5. Behavioral modeling

that the sequential designs contain both ‘combinational logic’ and ‘sequential logic’; also the figure shows that the ‘clock’ is required only for ‘sequential logics’. But in the current design, we used clock in both ‘combinational logic’ and ‘sequential logic’. Detailed discussion about FSM designs are in Chapter 9, where such issues are raised for careful designs.

5.8 Unintentional memories in combinational designs

In previous sections, we saw various statements which can be used within the process-block. Also, in Section 4.2, we discussed that the combinational designs do not have memories. Note that, processes are very easy to implement, but careful design is required, otherwise unintended memories (latches) or internal states (via feedback paths) may be created in combination designs, which is not desirable. This section shows one example of ‘unintentional latches’ along with the precautions to avoid such errors.

Listing 5.5 assign the values to ‘large’ and ‘small’ based on the values of ‘a’ and ‘b’. The design generated by the listing is shown in Fig. 5.7. Note that two latches are created in the design. Latches are introduced in the circuit, because we did not define the complete set of outputs in the conditions, e.g. we did no specify the value of ‘small’ when ‘a > b’ at Line 17. Therefore, a latch is created to store the previous value of ‘small’ for this condition.

Listing 5.5: Unintentional latch

```vhdl
-- latchEx.vhd
library ieee;
use ieee.std_logic_1164.all;

entity latchEx is
  port(
    a, b : in std_logic;
    large, small : out std_logic
  );
end entity;

architecture arch of latchEx is
begin
  process(a, b)
  begin
    if (a > b) then
      large <= a;
    elsif (a < b) then
      small <= a;
    end if;
  end process;
end arch;
```

Fig. 5.7: Latch generated by Listing 5.5

Note: The latches can be removed by following two simple rules which are applied in Listing 5.6 and corresponding
design is shown in Fig. 5.8.

- Include all the input signals related to combinational designs in the sensitivity list of process statement.
- Always define the ‘else’ block in ‘if statement’ and ‘others’ block in ‘case’ statement.
- Assign all outputs inside every block of statements e.g define all outputs inside every ‘elsif’ block of ‘if-else’ statement and inside every ‘when’ block of ‘case’ statement etc.

Listing 5.6: Remove unintentional latch

```vhdl
-- latchRemoveEx.vhd
library ieee;
use ieee.std_logic_1164.all;

entity latchRemoveEx is
    port(
        a, b : in std_logic;
        large, small : out std_logic
    );
end entity;

architecture arch of latchRemoveEx is
begin
    process(a, b)
    begin
        -- define all outputs in each if-elsif statement
        -- also include the 'else' statement
        if (a > b) then
            large <= a;
            small <= b;
        elsif (a < b) then
            large <= b;
            small <= a;
        else
            large <= '0';
            small <= '0';
        end if;
    end process;
end arch;
```

Fig. 5.8: Latch removed by Listing 5.6

Another method remove the unintended memories is the use of ‘default values’ as shown in Listing 5.7. Here we can defined, all the outputs at the beginning of the process statement (Lines 18-19). Then, we can overwrite the values in different statements (i.e. using multiple assignments) e.g. value of ‘large’ is modified in Line 21. The design generated by the listing is shown in Fig. 5.9.

Note: Note that, ‘multiple assignments’ are error-prone as discussed in Listing 5.1. It’s use should be limited to
Chapter 5. Behavioral modeling

‘default-value-assignments’ only. Do not use it for any other purposes.

5.9 Code-size vs design-size

Note that, in VHDL/Verilog designs the code-size has nothing to do with the design size. Remember, the well defined design (code may be very lengthy) can be easily understood by the synthesizer and will be implemented using less number of components.

Let’s understand this with an example. In Listing 5.8 and Listing 5.9, an square wave generator is implemented whose on/off duration is programmable as shown in Fig. 5.10. In Listing 5.8, the whole code is written in one process block, therefore the code is smaller as compare to Listing 5.9 where FSM approach is used to design the same circuit. The designs generated by these methods are shown in Fig. 5.11 and Fig. 5.12 respectively. In these figures, we can observe that, the code is larger for the FSM approach (Listing 5.9) but the resultant design is smaller than the first approach (Listing 5.8).
5.9. Code-size vs design-size

Fig. 5.10: Generated square wave by Listing 5.8 and Listing 5.8: On-time = 2, Off-time = 4

Fig. 5.11: Design generated by Listing 5.8

Fig. 5.12: Design generated by Listing 5.9
Note: Remember, code-size and design-size are independent of each other. The well defined designs are implemented with less number of components.

Listing 5.8: Code is small but design is large (see Fig. 5.11)

```vhdl
-- square_wave.vhd

-- High : on_duration * time_scale
-- Low : off_duration * time_scale

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity square_wave is
  port(
    clk, reset : in std_logic;
    on_duration, off_duration : in unsigned(2 downto 0);
    out_wave : out std_logic
  );
end entity;

architecture arch of square_wave is
  signal count_on, count_off : unsigned(2 downto 0);
  constant time_scale : unsigned(7 downto 0) := unsigned(to_signed(2, 8));
begin
  process(clk, reset)
  begin
    if reset = '1' then
      out_wave <= '0';
      count_on <= (others => '0');
      count_off <= (others => '0');
    elsif rising_edge(clk) then
      if count_on < on_duration * time_scale then
        out_wave <= '1';
        count_on <= count_on + 1;
      elsif count_off < off_duration * time_scale - 1 then
        out_wave <= '0';
        count_off <= count_off + 1;
      else
        count_on <= (others => '0');
        count_off <= (others => '0');
      end if;
    end if;
  end process;
end arch;
```

Listing 5.9: code is lengthy but design is smaller (see Fig. 5.12)

```vhdl
-- square_wave2.vhd

-- High : on_duration * time_scale
-- Low : off_duration * time_scale

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

(continues on next page)
```
5.10 Conclusion

In this chapter, various statements of behavioral modeling styles are discussed. Also, we saw the relationship between the designs generated by behavior modeling and dataflow modeling. Further, problem with loops are discussed and finally loop is implemented using ‘if’ statement.
Chapter 6

Procedures, functions and packages

6.1 Introduction

In this chapter, procedure and packages are discussed. Procedures are used to define common operations within many designs. Further, packages are used to define common declarations i.e. port names and constants values etc. in one file, instead of declaring in each file as shown in Section 2.

6.2 Procedure

A procedure contains a list of input and outputs arguments, and defined in declaration part of the architecture as shown in lines 19-25 of Listing 6.1.

Explanation Listing 6.1

In line 19, ‘sum2Num’ is the name of the procedure, which has two input signal (a and b) and two output signals (sum and diff). Then, line 27 maps the input and output port of the entity to this procedure. Note that signal ‘p’ is mapped to ‘sum’ signal, therefore line 28 is used to assign the value of ‘p’ to output port ‘d’. Fig. 6.1 is the design generated by the listing, whereas Fig. 6.2 is the simulation waveforms.

Fig. 6.1: Design generated by Listing 6.1
6.3. Function

'Functions' are similar to 'procedures' but can have input-ports only and return only one value. Note that, a 'return' statement is required in the functions as shown in Lines 23 and 21 of Listing 6.2 and Listing 6.3 respectively. Listing 6.2 implements the adder using function; whereas Listing 6.3 changes the 'binary number' into 'seven-segment display format' which is discussed in Section 8.5. Further the example of function inside the package is shown in Section 11.6.

Listing 6.2: Adder using function

```vhdl
-- funcEx.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

--sum2num procedure: add and subtract two numbers
procedure sum2num(signal a: in unsigned(N-1 downto 0);
                    signal b: in unsigned(N-1 downto 0);
                    signal sum, diff : out unsigned (N-1 downto 0)) is
begin
    sum <= a + b;
    diff <= a - b;
end sum2num; -- procedure ends
begin
    sum2num(a=>x,b=>y,diff=>p,sum=>s); -- procedure call
    d <= p; -- assing signal p to d
end arch;
```
entity funcEx is
  generic (N: integer := 2);
  port (
    x, y : in unsigned(N-1 downto 0);
    s : out unsigned (N-1 downto 0)
  );
end funcEx;

architecture arch of funcEx is
--sum2Num function: add and subtract two numbers
function sum2num(
  -- list all input here
  signal a: in unsigned(N-1 downto 0);
  signal b: in unsigned(N-1 downto 0)
)
  -- only one value can be return
  return unsigned is variable sum : unsigned(N-1 downto 0);
begin
  sum := a + b;
  return sum;
end sum2num; -- function ends
begin
  s <= sum2num(a=x, b=y); -- function call
end arch;

Listing 6.3: Sevensegment display using function

-- funcEx2.vhd
library ieee;
use ieee.std_logic_1164.all;

entity funcEx2 is
port(
  SW : in std_logic_vector(3 downto 0);
  LEDG : out std_logic_vector(3 downto 0);
  HEX0 : out std_logic_vector(6 downto 0)
);
end entity;

architecture arch of funcEx2 is
-- begin function
function binary_to_ssd (
  -- list all input here
  signal switch : std_logic_vector(3 downto 0)
)
  -- only one value can be return
  return std_logic_vector is variable sevenSegment : std_logic_vector(6 downto 0);
begin
  case switch is
    -- active low i.e. 0:display & 1:no display
    when "0000" => sevenSegment := "1000000"; -- 0,
    when "0001" => sevenSegment := "1111001"; -- 1
    when "0010" => sevenSegment := "0100100"; -- 2
    when "0011" => sevenSegment := "0110000"; -- 3
    when "0100" => sevenSegment := "0011001"; -- 4
    when "0101" => sevenSegment := "0010010"; -- 5
    when "0110" => sevenSegment := "0000010"; -- 6
    when "0111" => sevenSegment := "1111000"; -- 7
    when "1000" => sevenSegment := "0000000"; -- 8
    when "1001" => sevenSegment := "0010000"; -- 9
  end case;
end arch;
6.4 Packages

Note: Differences between the function and the procedure blocks,

- Procedures can have both input and output ports, whereas the functions can have only input ports.
- Functions can return only one value using ‘return’ keyword; whereas procedures do not have ‘return’ keyword but can return multiple values using ‘output’ port.

6.4 Packages

Note that the functions and the procedures can be defined in declaration parts of the entities and architectures; but the best place for defining these are in the packages. The packages are already discussed in Section 2. In this section, package body is discussed using Listing 6.4 and Listing 6.5.

Explanation Listing 6.4

In the listing constants (line 9), signals (line 17), data-types (line 18) and procedure (line 12) are defined inside the package ‘myPackage’. Further, procedure is declared in the package (line 12-19), and then defined in the package body (line 24-29). All these declarations are used by Listing 6.5.

Listing 6.4: Package
Chapter 6. Procedures, functions and packages

-- procedure for adding two numbers i.e. num1 + num2

procedure sum2num(signal a: in signed(3 downto 0);
     signal b: in signed(3 downto 0);
     signal sum : out signed (3 downto 0)) is
begin
    sum <= a + b;
end sum2num;
end myPackage;

Explanation Listing 6.5

In this listing, line 6 adds all the declaration of ‘myPackage’ to current design. Note that, ‘work’ is the
default directory where all the compiled file are stored.

Next, line 30 adds two number, in which ‘S’ is defined in the package.

Line 33 is using the procedure ‘sum2Num’ which is declared in the package. The working of this part
is same as Listing 6.1.

Line 26 defines the signal ‘currentState’ of type ‘stateType’ which is declared in package. Similarly,
signal ‘f’ used in line 38 which is also defined in the package. Rest of the working of line 38-55 is same
as Listing 5.4.

Listing 6.5: Adding Package to current design

--packageCall.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.myPackage.all;

entity packageCall is
    port (       
        clk : std_logic;
        x : in unsigned(2 downto 0);
        count : out unsigned(2 downto 0);
        num1, num2 : in signed(3 downto 0);
        sumConstant : out signed (3 downto 0);
        sumProcedure: inout signed (3 downto 0)
    );
end packageCall;

architecture arch of packageCall is
begin
    -- stateType is defined in package
    signal currentState : stateType;
    begin
        -- add constant number S = 1
        -- S is defined in package
        sumConstant <= num1 + S;

        sum2num procedure is defined in package
        sum2num(num1, num2, sumProcedure);

    -- if loop begin
    -- f is defined in package

(continues on next page)
6.5 Conclusion

In this chapter, we discuss the procedure and package. We define some of the previous designs in the package, and then use the package to create new design.

```vhdl
process(clk, currentstate, f)
begin
  if (f <= x) then
    currentState <= continueState;
  else
    currentState <= stopState;
  end if;
end process;

process(clk, currentstate)
begin
  if (currentState = continueState) then
    f <= f + 1;
  else
    f<=(others => '0');
  end if;
end process;

count <= f;
-- if loop end
end arch;
```
Chapter 7

Verilog designs in VHDL

7.1 Introduction

Since, both VHDL and Verilog are widely used in FPGA designs, therefore it be beneficial to combine both the designs together; rather than transforming the Verilog code to VHDL and vice versa. This chapter presents the use of Verilog design in the VHDL codes.

7.2 Verilog designs in VHDL

Design of 1 bit comparator in Listing 7.1 (which is written using Verilog) is same as the design of Listing 2.2. Design generated by Listing 7.1 is shown in Fig. 7.1. To use this Verilog design in VHDL, we need to declare the Verilog design as component, which is discussed in Listing 2.5. Further, we can not use the ‘work.comparator1BitVerilog’ method (discussed in Listing 2.4) to include the Verilog codes in the VHDL designs.

Listing 7.1: The 1 bit comparator in Verilog

```verilog
// comparator1BitVerilog.v

module comparator1BitVerilog(
  input wire x, y,
  output wire eq
);

wire s0, s1;

assign s0 = ~x & ~y; // (not x) and (not y)
assign s1 = x & y; // x and y
assign eq = s0 | s1; // s0 or s1

endmodule
```

Fig. 7.1: The 1 bit comparator using Verilog
7.3 Simulation of mixed designs

Explanation Listing 7.2

Verilog design is declared as component in lines 17-22. Then this component is instantiated in line 26 and 28 to design the 2 bit comparator. The final design generated for the two bit comparator is shown Fig. 7.2 In this way, we can use the Verilog designs in VHDL codes.

Listing 7.2: Verilog design in VHDL

```vhdl
--comparator2BitWithVerilog.vhd
library ieee;
use ieee.std_logic_1164.all;

entity comparator2BitWithVerilog is
    port(
        a, b : in std_logic_vector(1 downto 0);
        eq : out std_logic
    );
end comparator2BitWithVerilog;

architecture structure of comparator2BitWithVerilog is
    signal s0, s1: std_logic;
    -- define Verilog component
    component comparator1BitVerilog is
        port(
            x, y : in std_logic;
            eq : out std_logic
        );
    end component;
    begin
        -- use Verilog component i.e. comparator1BitVerilog
        eq_bit0: comparator1BitVerilog
            port map (x=>a(0), y=>b(0), eq=>s0);
        eq_bit1: comparator1BitVerilog
            port map (x=>a(1), y=>b(1), eq=>s1);
        eq <= s0 and s1;
    end structure;
```

Fig. 7.2: The 2 bit comparator using Verilog and VHDL

7.3 Simulation of mixed designs

Note that, Altera-Modelsim-Starter version does not allow simulation of mixed designs i.e. VHDL design mixed with Verilog design, can not be simulated in free version. You need to buy the full edition of Altera-modelsim for this.
Further, Active-HDL student version can be downloaded for free and can be used for mixed-modeling simulation. Lastly, in Active-HDL, all the waveforms can be imported as ‘.vcd’ format (if required); which can be used by Modelsim software as well. To use the .vcd file in Modelsim, first convert it into ‘.wlf’ file. For this, first go to Modelsim’s transcript window and then go to the desired directory (which contains the .vcd file) e.g. cd D:/VcdFiles; and then type conversion command i.e. ‘vcd2wlf VCD_file_name.vcd WLF_file_name.wlf’. This will convert the .vcd file to .wlf file, which can be used in Modelsim to display waveforms.

7.4 Conclusion

In this chapter, we used the Verilog designs in VHDL. Further, Verilog design must be declared as component to use it with VHDL.
Chapter 8

Visual verifications of designs

8.1 Introduction

In previous chapters, we saw various elements of VHDL language with some design examples, which were verified using simulations. In this chapter, various smaller units are designed and then combined together to make the large systems. Further, visual verification is performed in this chapter i.e. the designs are verified using LED displays. Finally, in Section 8.6, output of mod-m is displayed on various LEDs using smaller designs i.e. counters, clock-ticks and seven segment displays.

8.2 Flip flops

Flip flops are the sequential circuits which are used to store 1-bit. In this section, D flip flop is designed with various functionalities in it.

8.2.1 D flip flop

In Listing 8.1, the basic D flip flop is designed with reset button. Output of the flip flop is set to zero if reset value is ‘1’, otherwise output has the same value as input. Further, change in the output value occurs during ‘positive edge’ of the clock. Design generated by Listing 8.1 is shown in Fig. 8.1.

Explanation Listing 8.1

In the listing, ‘d’ and ‘q’ are the input and output respectively of the D flip flop. In line 19, output of the D flip flop is set to zero when reset is ‘1’. In line 20, ‘clk’event’ checks whether there is any change (or event) in the clock and then ‘clk=1’ checks if the changed value is ‘1’; in this way ‘rising edge’ of the clock can be checked in VHDL. Next in line 21, input value is sent to the output during the rising edge of the clock. Lastly, ‘null’ is used for ‘else’ block as we do not want to perform any further operation.

Listing 8.1: Basic D flip flop

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity basicDFF is
  port (    
    clk, reset: in std_logic;    
    d : in std_logic;    
    q : out std_logic
  );
end basicDFF;
```
Chapter 8. Visual verifications of designs

(continues from previous page)

```
end basicDFF;

architecture arch of basicDFF is
begin
  process(clk, reset)
  begin
    if (reset = '1') then
      q <= '0';
    elsif (clk'event and clk = '1') then -- check for rising edge of clock
      q <= d;
    else -- note that, else block is not required here
      null; -- do nothing
    end if;
  end process;
end arch;
```

Fig. 8.1: Basic D flip flop, Listing 8.1

8.2.2 D flip flop with Enable port

Note that, in Fig. 8.1, the enable button i.e. ‘ENA’ is still not connected. Enable button can be used for allowing the change in the output at desired time instant; e.g. if we want to change the output of the D flip flop on every 10th clock, then we can set the enable to ‘1’ for every 10th clock and ‘0’ for rest of the clocks. We call this as ‘tick’ at every 10 clock cycle; and we will see various examples of ‘ticks’ in this chapter. In this way, we can control the output of the D flip flop. To add the enable functionality in the flip flop, ‘en = 1’ is added in line 20 of Listing 8.2. Rest of the code is same as Listing 8.1. The design generated by the listing is shown in Fig. 8.2.

Listing 8.2: D flip flop with enable

```
-- D_FF.vhd
library ieee;
use ieee.std_logic_1164.all;

entity D_FF is
  port (
    clk, reset, en: in std_logic; -- en: enable
    d : in std_logic; -- input to D flip flop
    q : out std_logic -- output of D flip flop
  );
end D_FF;

architecture arch of D_FF is
begin
  process(clk, reset)
  begin
    if (reset = '1') then
      q <= '0';
    elsif (clk'event and clk = '1') then -- check for rising edge of clock
      q <= d;
    else -- note that, else block is not required here
      null; -- do nothing
    end if;
  end process;
end arch;
```
8.3 Counters

In this section, two types of counters are designed i.e. binary counter and mod-m counter.

8.3.1 Binary counter

In Listing 8.3, N-bit binary counter is designed which counts the number from 0 to $2^N - 1$. After reaching to maximum count i.e. $2^N - 1$, it again starts the count from 0.

Explanation Listing 8.3

In the listing, two output ports are defined i.e. ‘count’ and ‘complete_tick’, where ‘complete_tick’ is used to generate tick when the ‘count’ reached to its maximum value. In line 20, ‘MAX_COUNT’ is used to define the maximum count for ‘N’ bit counter; where ‘N’ is defined as generic in line 8.

Signal ‘count_reg’ is defined in line 21 and assigned to output at line 39. Since ‘count_reg’ is defined as ‘unsigned’, therefore its type is changed to ‘std_logic_vector’ in line 39 as output port ‘count’ is of type ‘std_logic_vector’.

Value of ‘count_reg’ is increased by one and assigned to ‘count_next’ in line 34. Then in next clock cycle, this increased value of ‘count_reg’ (which is store in ‘count_next’) is assigned to ‘count_reg’ again through line 28. Design generated by the listing is shown in Fig. 8.3.

Listing 8.3: N-bit binary counter

```vhdl
-- binaryCounter.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

(continues on next page)
Chapter 8. Visual verifications of designs

(entity binaryCounter is
    generic (N : integer := 3 -- N-bit binary counter);
    port(clk, reset : in std_logic;
         complete_tick : out std_logic;
         count : out std_logic_vector(N-1 downto 0));
end binaryCounter;

architecture arch of binaryCounter is
    constant MAX_COUNT : integer := 2**N - 1; -- maximum count for N bit
    signal count_reg, count_next : unsigned(N-1 downto 0);
    begin
        process(clk, reset)
        begin
            if reset = '1' then
                count_reg <= (others=>'0'); -- set count to 0 if reset
            elsif clk'event and clk='1' then
                count_reg <= count_next; -- assign next value of count
            else -- note that else block is not required
                null;
            end if;
        end process;
        count_next <= count_reg+1; -- increase the count
        complete_tick <= '1' when count_reg = MAX_COUNT else '0';
        count <= std_logic_vector(count_reg); -- assign value to output port
    end arch;

Fig. 8.3: N-bit binary counter, Listing 8.3

Explanation Fig. 8.3

In the figure, component ‘Equal0’ is generated according to line 37. 3’h7 shows that counter is designed for 3 bit which has the maximum value 7. Output of ‘Equal0’ i.e. complete_tick is set to 1 whenever output is equal to maximum value i.e. 7.

‘count_reg[2:0]’ shows that three D flip flops are used to create the 3 bit register. Also, we used only ‘clk’ and ‘reset’ ports in the design therefore enable port i.e. ‘ENA’ is unconnected. ‘Add0’ is included in the design to increase the value of count according to line 34. Finally, this increased value is assigned to output port in next clock cycle according to line 28. The simulation waveforms for this design is shown in Fig. 8.4. In the waveforms, we can see that a clock pulse is generated at the end of the count (see ‘complete_tick’) i.e. ‘111’ in the current example.
8.3.2 Mod-m counter

Mod-m counter counts the values from 0 to (m-1), which is designed in Listing 8.4.

**Explanation** Listing 8.4

The listing is same as Listing 8.3 with some minor changes which are explained here. In line 9, maximum count i.e. M is defined for ‘Mod-m counter, then in line 10, number for bits ‘N’ is defined which is required to count upto (M-1).

In line 37, count is set to zero, when maximum count is reached otherwise it is increased by 1. Line 40 generates a tick for each count completion. The design generated by the listing is shown in Fig. 8.5.

---

```
-- modMCounter.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity modMCounter is
  generic (
    M : integer := 5; -- count from 0 to M-1
    N : integer := 3 -- N bits required to count upto M i.e. 2**N >= M
  );

  port(
    clk, reset : in std_logic;
    complete_tick : out std_logic;
    count : out std_logic_vector(N-1 downto 0)
  );
end modMCounter;

architecture arch of modMCounter is

  signal count_reg, count_next : unsigned(N-1 downto 0);

begin

  process(clk, reset)
  begin
    if reset = '1' then
      count_reg <= (others=>'0');
    elsif clk'event and clk='1' then
      count_reg <= count_next;
    else -- note that else block is not required
      count_reg <= count_reg;
    end if;
  end process;

  -- set count_next to 0 when maximum count is reached i.e. (M-1)
  -- otherwise increase the count
```

(continues on next page)
count_next <= (others=>'0') when count_reg=(M-1) else (count_reg+1);
-- Generate 'tick' on each maximum count
complete_tick <= '1' when count_reg = (M-1) else '0';
count <= std_logic_vector(count_reg); -- assign value to output port
end arch;

Explanation Fig. 8.5

This figure is same as Fig. 8.3 with few changes to stop the count at 'M'. Firstly, in ‘Equal0’ component '3'h4' (i.e. ‘M-1’) is used instead of 3'h7, as ‘M’ is set to 5 in line 9; and the output of ‘Equal0’ is set to 1 whenever the count reaches to 4. Then output of ‘Equal0’ is sent to the multiplexer ‘MUX21’; which selects the count 3'h0 whenever the output of ‘Equal0’ is one, otherwise incremented value (given by ‘Add0’) is sent to the D flip flops. The simulation waveforms for this design is shown in Fig. 8.6. In the waveforms, we can see that a clock pulse is generated at the end of the count i.e. ‘100’ (see ‘complete_tick’) in the current example.

8.4 Clock ticks

In Listing 8.5, Mod-m counter is used to generate the clock ticks of different frequencies, which can be used for operating the devices which work on different clock frequencies. Further, we will use the listing for visual verification of the designs using ‘LEDs’ and ‘seven segment displays’.

Explanation Listing 8.5

The listing uses the ‘Mod-m’ counter (as shown lines 22-24) to generate the ticks with different time period. In line 12, M = 5 is set, to generate the ticks after every 5 clock cycles as shown in Fig. 8.8. In the figure, two ‘red cursors’ are used to display the 5 clocks cycles, and during 5th cycle the output port i.e. ‘clkPulse’ is set to 1. Further, Fig. 8.7 shows the structure of the design, whose internal design is defined by Mod-m counter in Listing 8.4. Lastly, different values of ‘m’ and corresponding ‘N’ are shown in lines 8-10, to generated the clock ticks of different time period.
8.5 Seven segment display

In this section, VHDL code for displaying the count on seven segment display device is presented, which converts the hexadecimal number format (i.e. 0 to F) into 7-segment display format. Further, a test circuit is designed to check the output of the design on the FPGA board.

8.5.1 Implementation

Listing 8.6 is designed for active-low seven segment display device, i.e. LEDs of the device will glow if input is '0'. Then, at the end of the design, output is generated for both active low and active high seven segment display devices.
Chapter 8. Visual verifications of designs

Fig. 8.9: Seven segment display

Explanation Listing 8.6

In the listing, hexadecimal number i.e. ‘hexNumber’ is converted into seven segment format i.e. ‘sevenSegment’. Lines 17 to 33 perform this conversion e.g. if hexadecimal number is 0 (i.e. “0000” in binary format), then it is converted into “1000000” in line 18. Since, seven segment display device in the Altera DE2 board is active low, therefore ‘1’ is used in the beginning (i.e. ‘7th’ position is set to ‘1’ in Fig. 8.9), so that 7th LED will not glow and ‘0’ will be displayed on the seven segment display.

Since the design is for active low system, therefore in line 35, the signal ‘sevenSegment’ is assigned directly to the output port ‘sevenSegmentActiveLow’; whereas it is inverted for the active high output port i.e. ‘sevenSegmentActiveHigh’ in line 36. In this way, we can use this design for any kind of devices. In the next section, test circuit is shown for this design.

Listing 8.6: Hexadecimal to seven segment display conversion

```vhdl
-- hexToSevenSegment.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity hexToSevenSegment is
  port(
    hexNumber : in std_logic_vector(3 downto 0);
    sevenSegmentActiveLow, sevenSegmentActiveHigh : out std_logic_vector(6 downto 0)
  );
end hexToSevenSegment;

architecture arch of hexToSevenSegment is
  signal sevenSegment : std_logic_vector(6 downto 0);
begin
  with hexNumber select
  begin
    sevenSegment <= "1000000" when "0000", -- 0, active low i.e. 0:display & 1:no display
    "1111001" when "0001", -- 1
    "0100100" when "0010", -- 2
    "0110000" when "0011", -- 3
    "0011001" when "0100", -- 4
    "0010010" when "0101", -- 5
    "0000010" when "0110", -- 6
    "1111000" when "0111", -- 7
    "0000000" when "1000", -- 8
    "0010000" when "1001", -- 9
    "0001000" when "1010", -- a
    "0000111" when "1011", -- b
    "1000110" when "1100", -- c
    "0100001" when "1101", -- d
    "0000110" when "1110", -- e
  end with;
end arch;
```
8.5. Seven segment display

(continued from previous page)

```
"0001110" when others; -- f
sevenSegmentActiveLow <= sevenSegment; -- Seven segment with Active Low
sevenSegmentActiveHigh <= not sevenSegment; -- Seven segment with Active High
end arch;
```

8.5.2 Test design for 7 segment display

Till now, we checked the outputs of the circuits using simulation. In this section, output of the code is displayed on the Seven segment devices which is available on the DE2 FPGA board. Further, we can use the design with other boards as well; for this purpose, the only change required is the pin-assignments, which are board specific.

VHDL code for testing the design is presented in Listing 8.6. Note that, in this listing, we use the names ‘SW’ and ‘HEX0’ etc., which are defined in ‘DE2_PinAssg_PythonDSP.csv’ file. Partial view of this file is shown in Fig. 8.10, which is provided in the zip folder along with the codes and can be downloaded from the website. This file is used for ‘pin assignment’ in Altera DE2 board. DE2 board provides the 18 switches i.e. SW0 to SW17. Pin number for SW0 is ‘PIN_N25’, therefore in the ‘.csv file’, we used the name SW[0] for (SW0) and assign the pin number ‘PIN_N25’ in location column. Note that, we can not change the header names i.e. ‘To’ and ‘Location’ for using the ‘.csv file’. Further, we can change the names to any other desired names e.g. SW17 is named as ‘reset’, which will be used for resetting the system in later designs.

For pin assignments with ‘.csv file’, go to Assignments–>Import Assignments and then select the file ‘DE2_PinAssg_PythonDSP.csv’. Next, to see the pin assignments, go to Assignments–>Pin Planner, which will display the pin assignments as shown in Fig. 8.11. Further, we can manually change the pin location by clicking on the ‘Location’ column in the figure.

Note: Names ‘SW’ and ‘HEX0’ etc. are used along with ‘std_logic_vector’ in the testing circuits, so that pin numbers are automatically assigned to these ports according to ‘.csv file’. Otherwise we need to manually assign the pin numbers to the ports.

Explanation Listing 8.7

In Line 10 of the listing, 4 bit input port ‘SW’ is defined. Therefore, pin number will be assigned to these switches according to name ‘SW[0]’ to ‘SW[3]’ etc. in the ‘.csv file’. In line 11, two output ports are defined i.e. HEX0 and HEX1. Next, in line 18 and 22, HEX0 and HEX1 are mapped to active low and active high outputs of the Listing 8.6 respectively. Note that, it is optional to define all the output ports in the port mapping, e.g. output port ‘sevenSegmentActiveHigh’ is not declared in line 19; whereas all the input ports must be declared in port mapping.

Now this design can be loaded on the FPGA board. Then, change the switch patterns to see the outputs on the seven segment display devices. Since, HEX0 and HEX1 are set for active low and active high respectively, therefore HEX1 will display the LEDs which are not glowing on the HEX0 e.g. when HEX0 displays the number ‘8’, then HEX1 will not glow any LED as all the LEDs of HEX0 are in the ‘on’ condition.

Listing 8.7: Test design for seven segment display

```
--hexToSevenSegment_testCircuit
-- testing circuit for hexToSevenSegment.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity hexToSevenSegment_testCircuit is
    port(
```
### Chapter 8. Visual verifications of designs

#### Fig. 8.10: Partial display of ‘Pin assignments file’ i.e. DE2_PinAssg_PythonDSP.cs

<table>
<thead>
<tr>
<th>To</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW[0]</td>
<td>PIN_N25</td>
</tr>
<tr>
<td>SW[1]</td>
<td>PIN_N26</td>
</tr>
<tr>
<td>SW[3]</td>
<td>PIN_AE14</td>
</tr>
<tr>
<td>SW[4]</td>
<td>PIN_AF14</td>
</tr>
<tr>
<td>reset</td>
<td>PIN_V2</td>
</tr>
<tr>
<td>HEX0[6]</td>
<td>PIN_AF10</td>
</tr>
<tr>
<td>HEX0[5]</td>
<td>PIN_AB12</td>
</tr>
<tr>
<td>HEX0[4]</td>
<td>PIN_AC12</td>
</tr>
<tr>
<td>HEX0[3]</td>
<td>PIN_AD11</td>
</tr>
<tr>
<td>HEX0[2]</td>
<td>PIN_AE11</td>
</tr>
<tr>
<td>HEX0[1]</td>
<td>PIN_V14</td>
</tr>
<tr>
<td>HEX0[0]</td>
<td>PIN_V13</td>
</tr>
<tr>
<td>HEX1[6]</td>
<td>PIN_V20</td>
</tr>
</tbody>
</table>

#### Fig. 8.11: Partial display of Pin Assignments

<table>
<thead>
<tr>
<th>Node Name</th>
<th>Direction</th>
<th>Location</th>
<th>I/O Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW[3]</td>
<td>Inout</td>
<td>PIN_AE14</td>
<td>3.3-V LV...default</td>
</tr>
<tr>
<td>SW[2]</td>
<td>Inout</td>
<td>PIN_P25</td>
<td>3.3-V LV...default</td>
</tr>
<tr>
<td>SW[1]</td>
<td>Inout</td>
<td>PIN_N26</td>
<td>3.3-V LV...default</td>
</tr>
<tr>
<td>SW[0]</td>
<td>Inout</td>
<td>PIN_N25</td>
<td>3.3-V LV...default</td>
</tr>
<tr>
<td>SW[4]</td>
<td>Unknown</td>
<td>PIN_AE14</td>
<td>3.3-V LV...default</td>
</tr>
<tr>
<td>SW[5]</td>
<td>Unknown</td>
<td>PIN_AD13</td>
<td>3.3-V LV...default</td>
</tr>
<tr>
<td>SW[6]</td>
<td>Unknown</td>
<td>PIN_AC13</td>
<td>3.3-V LV...default</td>
</tr>
<tr>
<td>SW[7]</td>
<td>Unknown</td>
<td>PIN_C13</td>
<td>5.5-V LV...default</td>
</tr>
<tr>
<td>SW[8]</td>
<td>Unknown</td>
<td>PIN_B13</td>
<td>3.3-V LV...default</td>
</tr>
<tr>
<td>SW[9]</td>
<td>Unknown</td>
<td>PIN_A13</td>
<td>3.3-V LV...default</td>
</tr>
</tbody>
</table>
8.6 Visual verification of Mod-m counter

In previous section, we displayed the outputs on 7 segment display devices, but clocks are not used in the system. In this section, we will verify the designs with clocks, by visualizing the outputs on LEDs and seven segment displays. Since, 50 MHz clock is too fast to visualize the change in the output with eyes, therefore Listing 8.8 uses the 1 Hz clock frequency for mod-m counter, so that we can see the changes in the outputs.

Explanation Listing 8.8

Since, DE2 provides clock with 50 MHz frequency, therefore it should count upto $5 \times 10^7$ to elapse 1 sec time i.e. $\frac{50\text{MHz}}{5 \times 10^7} = \frac{50 \times 10^6 \text{Hz}}{5 \times 10^7} = 1\text{Hz} = 1\text{sec}$. Therefore $M=50000000$ is used in line 29.

In the listing, three component are instantiated i.e. ‘clockGenerator’, ‘modMCounter’ and ‘hexToSevenSegment’ for the visual verification of mod-m counter. Further, counts are verified using both LEDs and seven segment display. In line 25, ‘clockGenerator’ is instantiated which generates the clock of 1 second i.e. ‘clk_Pulse1s’. Then this 1 second clock is used by second instantiation i.e. mod-m counter as shown in line 33. This can be seen in Fig. 8.12 where output of ‘clockGenerator’ is connect with input clock of mod-m counter. Lastly, all these signals are sent to output port i.e. 1 second clock is displayed by LEDR[0] (line 30), whereas ‘completed-count-tick’ is displayed by LEDR[1] (line 36). Also, counts are displayed by green LEDs i.e. LEDG (line 38). Further, seven segment display is also instantiated at line 41, to display the count on seven segment display as well.

Listing 8.8: Mod-m counter verification with 1 second clock
Chapter 8. Visual verifications of designs

(continued from previous page)

15  LEDR : out std_logic_vector (1 downto 0);
16  LEDG : out std_logic_vector(N-1 downto 0)
17  );
18  end modMCounter_VisualTest;
19
architecture arch of modMCounter_VisualTest is
20
21  signal clk_Pulse1s: std_logic;
22  signal count : std_logic_vector(N-1 downto 0);
23  begin
24    -- clock 1 s
25    clock_1s: entity work.clockTick
26      generic map (M=>50000000, N=>26)
27      port map (clk=>CLOCK_50, reset=>reset,
28        clkPulse=>clk_Pulse1s);
29    LEDR(0) <= clk_Pulse1s; -- display clock pulse of 1 s
30
31    -- modMCounter with 1 sec clock pulse
32    modMCounter1s: entity work.modMCounter
33      generic map (M=>M, N=>N)
34      port map (clk=>clk_Pulse1s, reset=>reset,
35        complete_tick=>LEDR(1), count=>count);
36    LEDG <= count; -- display count on green LEDs
37
38    -- display count on seven segment
39    hexToSevenSegment0 : entity work.hexToSevenSegment
40      port map (hexNumber=>count, sevenSegmentActiveLow=>HEX0);
41  end;
42
Fig. 8.12: Mod-m counter verification with 1 second clock

8.7 Conclusion

In this chapter, we designed the circuit which generates various ‘ticks’ of different frequencies. Then the ticks are used for visual verifications of the designs using LEDs and seven segment displays. Further, structural modeling approach is used for the visual verifications of the systems.
Chapter 9

Finite state machines

9.1 Introduction

In previous chapters, we saw various examples of the combinational circuits and sequential circuits. In combinational circuits, the output depends on the current values of inputs only; whereas in sequential circuits, the output depends on the current values of the inputs along with the previously stored information. In the other words, storage elements, e.g. flip flogs or registers, are required for sequential circuits. The information stored in these elements can be seen as the states of the system. If a system transits between finite number of such internal states, then finite state machines (FSM) can be used to design the system. The FSM designed can be classified as ‘Moore machine’ and ‘Mealy machine’ which are discussed in this chapter.

This chapter is organized as follows. First, Moore and Mealy designs are discussed in Section 9.2. Then an example of these designs are shown in Section 9.3. After this the sequential circuit designs using FSM are discussed in details.

9.2 Comparison: Mealy and Moore designs

FSM design is known as Moore design if the output of the system depends only on the states (see Fig. 9.2); whereas it is known as Mealy design if the output depends on the states and external inputs (see Fig. 9.1). Further, a system may contain both types of designs simultaneously.

Note: Following are the differences in Mealy and Moore design,

- In Moore machine, the outputs depend on states only, therefore it is ‘synchronous machine’ and the output is available after 1 clock cycle as shown in Fig. 9.3. Whereas, in Mealy machine output depends on states along with external inputs; and the output is available as soon as the input is changed therefore it is ‘asynchronous machine’ (See Fig. 9.15 for more details).
- Mealy machine requires fewer number of states as compared to Moore machine as shown in Listing 9.1.
- Moore machine should be preferred for the designs, where glitches (see Section 9.4) are not the problem in the systems.
- Mealy machines are good for synchronous systems which requires ‘delay-free and glitch-free’ system (See example in Section 9.7.1), but careful design is required for asynchronous systems. Therefore, Mealy machine can be complex as compare to Moore machine.

9.3 Example: Rising edge detector

Rising edge detector generates a tick for the duration of one clock cycle, whenever input signal changes from 0 to 1. In this section, state diagrams of rising edge detector for Mealy and Moore designs are shown. Then rising edge
detector is implemented using VHDL code. Also, outputs of these two designs are compared.

### 9.3.1 State diagrams: Mealy and Moore design

Fig. 9.1 and Fig. 9.2 are the state diagrams for Mealy and Moore designs respectively. In Fig. 9.1, the output of the system is set to 1, whenever the system is in the state ‘zero’ and value of the input signal ‘level’ is 1; i.e. output depends on both the state and the input. Whereas in Fig. 9.2, the output is set to 1 whenever the system is in the state ‘edge’ i.e. output depends only on the state of the system.

![State Diagrams](image)

**Fig. 9.1: Mealy Design**

### 9.3.2 Implementation

Both Mealy and Moore designs are implemented in Listing 9.1. The listing can be seen as two parts i.e. Mealy design (Lines 36-54) and Moore design (Lines 56-80). Please read the comments for complete understanding of the code. The simulation waveforms i.e. Fig. 9.3 are discussed in next section.

**Listing 9.1: Edge detector: Mealy and Moore designs**

```vhdl
1 -- edgeDetector.vhd
2 -- Moore and Mealy Implementation
3
4 library ieee;
5 use ieee.std_logic_1164.all;
6
7 entity edgeDetector is
8 port(
9   clk, reset : in std_logic;
10  level : in std_logic;
11  Mealy_tick, Moore_tick: out std_logic
12 );
13 end edgeDetector;
14
15 architecture arch of edgeDetector is
16   type stateMealy_type is (zero, one); -- 2 states are required for Mealy
17   signal stateMealy_reg, stateMealy_next : stateMealy_type;
18
(continues on next page)
```
9.3. Example: Rising edge detector

Fig. 9.2: Moore Design

```vhdl
type stateMoore_type is (zero, edge, one); -- 3 states are required for Moore
signal stateMoore_reg, stateMoore_next : stateMoore_type;

begin

process(clk, reset)
begin
if (reset = '1') then -- go to state zero if reset
    stateMoore_reg <= zero;
    stateMealy_reg <= zero;
elsif (clk'event and clk = '1') then -- otherwise update the states
    stateMoore_reg <= stateMoore_next;
    stateMealy_reg <= stateMealy_next;
else
    null;
end if;
end process;

-- Mealy Design
process(stateMealy_reg, level)
begin
-- store current state as next
stateMealy_next <= stateMealy_reg; -- required: when no case statement is satisfied
Mealy_tick <= '0'; -- set tick to zero (so that 'tick = 1' is available for 1 cycle only)
case stateMealy_reg is
when zero => -- set 'tick = 1' if state = zero and level = '1'
    if level = '1' then -- if level is 1, then go to state one,
        stateMealy_next <= one; -- otherwise remain in same state.
        Mealy_tick <= '1';
    end if;
when one =>
    if level = '0' then -- if level is 0, then go to zero state,
        stateMealy_next <= zero; -- otherwise remain in one state.
end process;
```

(continues on next page)
Chapter 9. Finite state machines

(continued from previous page)

```
end if;
end case;
end process;

-- Moore Design
process(stateMoore_reg, level)
begin
  -- store current state as next
  stateMoore_next <= stateMoore_reg; -- required: when no case statement is satisfied

  Moore_tick <= '0'; -- set tick to zero (so that ‘tick = 1’ is available for 1 cycle only)
  case stateMoore_reg is
    when zero => -- if state is zero,
      if level = '1' then -- and level is 1
        stateMoore_next <= edge; -- then go to state edge.
      end if;
    when edge =>
      Moore_tick <= '1'; -- set the tick to 1.
      if level = '1' then -- if level is 1,
        stateMoore_next <= one; -- go to state one,
      else
        stateMoore_next <= zero; -- else go to state zero.
      end if;
    when one =>
      if level = '0' then -- if level is 0,
        stateMoore_next <= zero; -- then go to state zero.
      end if;
  end case;
end process;
end arch;
```

9.3.3 Outputs comparison

In Fig. 9.3, it can be seen that output-tick of Mealy detector is generated as soon as the ‘level’ goes to 1, whereas Moore design generate the tick after 1 clock cycle. These two ticks are shown with the help of the two red cursors in the figure. Since, output of Mealy design is immediately available therefore it is preferred for synchronous designs.

![Fig. 9.3: Simulation waveforms of rising edge detector in Listing 9.1](image)

9.3.4 Visual verification

Listing 9.2 can be used to verify the results on the FPGA board. Here, clock with 1 Hz frequency is used in line 19, which is defined in Listing 8.5. After loading the design on FPGA board, we can observe on LEDs that the output of Moore design displayed after Mealy design, with a delay of 1 second.
9.4 Glitches

Glitches are the short duration pulses which are generated in the combinational circuits. These are generated when more than two inputs change their values simultaneously. Glitches can be categorized as ‘static glitches’ and ‘dynamic glitches’. Static glitches are further divided into two groups i.e. ‘static-0’ and ‘static-1’. ‘Static-0’ glitch is the glitch which occurs in logic ‘0’ signal i.e. one short pulse ‘high-pulse (logic-1)’ appears in logic-0 signal (and the signal settles down). Dynamic glitch is the glitch in which multiple short pulses appear before the signal settles down.

Note: Most of the times, the glitches are not the problem in the design. Glitches create problem when it occur in the outputs, which are used as clock for the other circuits. In this case, glitches will trigger the next circuits, which will result in incorrect outputs. In such cases, it is very important to remove these glitches. In this section, the glitches are shown for three cases. Since, clocks are used in synchronous designs, therefore Section 9.4.3 is of our main interest.

9.4.1 Combinational design in asynchronous circuit

Table 9.1 shows the truth-table for $2 \times 1$ multiplexer and corresponding Karnaugh map is shown in Fig. 9.4. Note that, the glitches occurs in the circuit, when we exclude the ‘red part’ of the solution from the Fig. 9.4, which results in minimum-gate solution, but at the same time the solution is disjoint. To remove the glitch, we can add the prime-implicant in red-part as well. This solution is good, if there are few such gates are required; however if the number of inputs are very high, whose values are changing simultaneously then this solution is not practical, as we need to add large number of gates.
Table 9.1: Truth table of $2 \times 1$ Multiplexer

<table>
<thead>
<tr>
<th>sel</th>
<th>in0</th>
<th>in1</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$$z = (\text{sel and in1}) \text{ or (not}\text{(sel) and in0})$$

$$z = (\text{sel and in1}) \text{ or (not}\text{(sel) and in0}) \text{ or (in0 and in1)}$$

Fig. 9.4: Reason for glitches and solution

Fig. 9.5: Glitches in design in Listing 9.3

Listing 9.3: Glitches in multiplexer

```vhdl
-- glitchEx.vhd
-- 2x1 Multiplexer using logic-gates
library ieee;
use ieee.std_logic_1164.all;

entity glitchEx is
  port(
    in0, in1, sel : in std_logic;
    z : out std_logic
  );
end entity;
```

(continues on next page)
9.4.2 Unfixable Glitch

Listing 9.4 is another example of glitches in the design as shown in Fig. 9.6. Here, glitches are continuous i.e. these are occurring at every change in signal ‘din’. Such glitches are removed by using D-flip-flop as shown in Section 9.4.3. Since the output of Manchester code depends on both edges of clock (i.e. half of the output changes on ‘+ve’ edge and other half changes at ‘-ve’ edge), therefore such glitches are unfixable; as in VHDL both edges can not be connected to one D flip flop. However, the simulation can be performed with both edges connected to D flip flop as shown in Listing 9.5; where glitches are removed using D flip flop.

![Fig. 9.6: Glitches in Listing 9.4](image-url)

Listing 9.4: Glitches in Manchester coding

```vhdl
architecture arch of manchester_code is
  port(
    clk, din : in std_logic;
    dout : out std_logic
  );
end entity;
architecture arch of manchester_code is
begin
  dout <= clk xor din;
end arch;
```

Listing 9.5: Non-synthesizable solution for removing glitches

```vhdl
library ieee;
use ieee.std_logic_1164.all;
architecture arch of manchester_code is
begin
  dout <= clk xor din;
end arch;
```
9.4.3 Combinational design in synchronous circuit

Combination designs in sequential circuits were discussed in Fig. 4.1. The output of these combination designs can depend on states only, or on the states along with external inputs. The former is known as Moore design and latter is known as Mealy design as discussed in Section 9.2. Since, the sequential designs are sensitive to edge of the clock, therefore the glitches can occur only at the edge of the clock. Hence, the glitches at the edge can be removed by sending the output signal through the D flip flop, as shown in Fig. 9.7. Various VHDL templates for sequential designs are shown in Section 9.5 and Section 9.6.

9.5 Moore architecture and VHDL templates

Fig. 9.7 shows the different block for the sequential design. In this figure, we have three blocks i.e. ‘sequential logic’, ‘combinational logic’ and ‘glitch removal block’. In this section, we will define three process-statements to implemented these blocks (see Listing 9.7). Further, ‘combinational logic block’ contains two different logics i.e. ‘next-state’ and ‘output’. Therefore, this block can be implemented using two different block, which will result in four process-statements (see Listing 9.6).

Moore and Mealy machines can be divided into three categories i.e. ‘regular’, ‘timed’ and ‘recursive’. The differences in these categories are shown in Fig. 9.8 - Fig. 9.10 for Moore machine. In this section, we will see different VHDL templates for these categories. Note that, the design may be the combinations of these three categories,
9.5. Moore architecture and VHDL templates

and we need to select the correct template according to the need. Further, the examples of these templates are shown in Section 9.7.

9.5.1 Regular machine

Please see the Fig. 9.8 and note the following points about regular Moore machine,

- Output depends only on the states, therefore no ‘if statement’ is required in the process-statement. For example, in Lines 76-78 of Listing 9.6, the outputs (Lines 77-78) are defined inside the ‘when’ statement which contains only state i.e. ‘s0’ (Line 76).
- Next-state depends on current-state and and current external inputs. For example, the ‘state_next’ at Line 46 of Listing 9.6 is defined inside ‘if statement’ (Line 45) which depends on current input. Further, this ‘if statement’ is defined inside the state ‘s0’ (Line 44). Hence the next state depends on current state and current external input.

Note: In regular Moore machine,

- Outputs depend on current external inputs.
- Next states depend on current states and current external inputs.
Fig. 9.10: Recursive Moore machine: output ‘z’ depends on output i.e. feedback required

Listing 9.6: VHDL template for regular Moore FSM: separate ‘next_state’ and ‘output’ logic

```
-- moore_regular_template.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity moore_regular_template is
  generic (param1 : std_logic_vector(...); param2 : unsigned(...) := <value>);
  port (clk, reset : in std_logic;
        input1, input2, ... : in std_logic_vector(...);
        output1, output2, ... : out signed(...));
end entity;

architecture arch of moore_regular_template is
  type stateType is (s0, s1, s2, s3, ...);
  signal state_reg, state_next : stateType;
begin
  -- state register : state_reg
  -- This process contains sequential part and all the D-FF are
  -- included in this process. Hence, only 'clk' and 'reset' are
  -- required for this process.
  process(clk, reset)
  begin
    if reset = '1' then
      state_reg <= s1;
    elsif (clk) then
      state_reg <= state_next;
    end if;
  end process;

  -- next state logic : state_next
  -- This is combinational of the sequential design,
  -- which contains the logic for next-state
  -- include all signals and input in sensitive-list except state_next
  process(input1, input2, state_reg)
  begin
    state_next <= state_reg; -- default state_next
    case state_reg is
      when s0 =>
        if <condition> then -- if (input1 = '01') then
          state_next <= s1;
        elsif <condition> then -- add all the required conditionstions
          state_next <= ...;
    end case;
  end process;
end arch;
```
else -- remain in current state
    state_next <= s0;
end if;
when s1 =>
    if <condition> then -- if (input1 = '10') then
        state_next <= s2;
    elsif <condition> then -- add all the required condition statements
        state_next <= ...;
    else -- remain in current state
        state_next <= s1;
    end if;
when s2 =>
    ... end case;
end process;

-- combination output logic
-- This part contains the output of the design
-- no if-else statement is used in this part
-- include all signals and input in sensitive-list except state_next
begin
    -- default outputs
    output1 <= <value>;
    output2 <= <value>;
    ...
    case state_reg is
    when s0 =>
        output1 <= <value>;
        output2 <= <value>;
        ...
    when s1 =>
        output1 <= <value>;
        output2 <= <value>;
        ...
    when s2 =>
        ...
    end case;
end process;

-- optional D-FF to remove glitches
begin
    if reset = '1' then
        new_output1 <= ...;
        new_output2 <= ...;
    elsif rising_edge(clk) then
        new_output1 <= output1;
        new_output2 <= output2;
    end if;
end process;
end architecture;

Listing 9.7 is same as Listing 9.6, but the output-logic and next-state logic are combined in one process block.

Listing 9.7: VHDL template for regular Moore FSM: combined 'next_state' and 'output' logic

```
-- moore_regular_template2.vhd
library ieee;
```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity moore_regular_template2 is
  generic (  
    param1 : std_logic_vector(...) := <value>;
    param2 : unsigned(...) := <value>
  );
  port (  
    clk, reset : in std_logic;
    input1, input2, ... : in std_logic_vector(...);
    output1, output2, ... : out signed(...)
  );
end entity;

architecture arch of moore_regular_template2 is
  type stateType is (s0, s1, s2, s3, ...);
  signal state_reg, state_next : stateType;
begin  
  -- state register : state_reg  
  -- This process contains sequential part and all the D-FF are  
  -- included in this process. Hence, only 'clk' and 'reset' are  
  -- required for this process.  
  process(clk, reset)  
  begin  
    if reset = '1' then  
      state_reg <= s1;
      elsif (clk) then
        state_reg <= state_next;
      end if;  
  end process;  
  -- next state logic and outputs  
  -- This is combinational of the sequential design,  
  -- which contains the logic for next-state and outputs  
  -- include all signals and input in sensitive-list except state_next  
  process(input1, input2, ..., state_reg)  
  begin  
    state_next <= state_reg; -- default state_next  
    output1 <= <value>;
    output2 <= <value>;
    ...  
    case state_reg is  
      when s0 =>  
        output1 <= <value>;
        output2 <= <value>;
        ...  
      if <condition> then -- if (input1 = '01') then
        state_next <= s1;
      elsif <condition> then -- add all the required conditionstions
        state_next <= ...;
      else -- remain in current state  
        state_next <= s0;
      end if;
      when s1 =>  
        output1 <= <value>;
        output2 <= <value>;
        ...  
      if <condition> then -- if (input1 = '10') then
        state_next <= s2;
    end case;  
end process;
9.5. Moore architecture and VHDL templates

(continued from previous page)

```vhdl
elsif <condition> then  -- add all the required conditions
    state_next <= ...;
else  -- remain in current state
    state_next <= s1;
end if;
when s2 =>
    ...
end case;
end process;
```

```vhdl
-- optional D-FF to remove glitches
process(clk, reset)
begin
    if reset = '1' then
        new_output1 <= ...;
        new_output2 <= ...;
    elsif rising_edge(clk) then
        new_output1 <= output1;
        new_output2 <= output2;
    end if;
end process;
end architecture;
```

9.5.2 Timed machine

If the state of the design changes after certain duration (see Fig. 9.9), then we need to add the timer in the VHDL
design which are created in Listing 9.6 and Listing 9.6. For this, we need to add one more process-block which
performs following actions,

- **Zero the timer**: The value of the timer is set to zero, whenever the state of the system changes.
- **Stop the timer**: Value of the timer is incremented till the predefined ‘maximum value’ is reached and then
  it should be stopped incrementing. Further, it’s value should not be set to zero until state is changed.

**Note:** In timed Moore machine,
- Outputs depend on current external inputs.
- Next states depend on time along with current states and current external inputs.

Template for timed Moore machine is shown in Listing 9.8, which is exactly same as Listing 9.7 except with
following changes,

- Timer related constants are added at Line 23-29.
- A process block is added to stop and zero the timer (Lines 44-56).
- Finally, timer related conditions are included for next-state logic e.g. Lines 67 and 69 etc.

Listing 9.8: VHDL template timed Moore FSM : separate
‘next_state’ and ‘output’ logic

```vhdl
-- moore_timed_template.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity moore_timed_template is
    generic (  
        param1 : std_logic_vector(...) := <value>;  
        param2 : unsigned(...) := <value>
    )
end entity;
```
architecture arch of moore_timed_template is
  type stateType is (s0, s1, s2, s3, ...);
  signal state_reg, state_next : stateType;

  -- timer
  constant T1 : natural := <value>;
  constant T2 : natural := <value>;
  constant T3 : natural := <value>;
  ...
  signal t : natural;
begin
  -- state register : state_reg
  -- This process contains sequential part and all the D-FF are
  -- included in this process. Hence, only 'clk' and 'reset' are
  -- required for this process.
  process(clk, reset)
  begin
    if reset = '1' then
      state_reg <= s1;
    elsif (clk) then
      state_reg <= state_next;
    end if;
  end process;

  -- timer
  process(clk, reset)
  begin
    if reset = '1' then
      t <= 0;
    elsif rising_edge(clk) then
      if state_reg /= state_next then -- state is changing
        t <= 0;
      else
        t <= t + 1;
      end if;
    end if;
  end process;

  -- next state logic : state_next
  -- This is combinational of the sequential design,
  -- which contains the logic for next-state
  -- include all signals and input in sensitive-list except state_next
  process(input1, input2, state_reg)
  begin
    state_next <= state_reg; -- default state_next
    case state_reg is
      when s0 =>
        if <condition> and t >= T1-1 then -- if (input1 = '01') then
          state_next <= s1;
        elsif <condition> and t >= T2-1 then -- add all the required conditionstion
          state_next <= ...;
        else -- remain in current state
          state_next <= s0;
        end case;
  end process;
end arch;
end if;
when s1 =>
    if <condition> and t >= T3-1 then -- if (input1 = '10') then
        state_next <= s2;
    elsif <condition> and t >= T2-1 then -- add all the required conditions
        state_next <= ...;
    else -- remain in current state
        state_next <= s1;
    end if;
when s2 => ... end case;
end process;

-- combination output logic
-- This part contains the output of the design
-- no if-else statement is used in this part
-- include all signals and input in sensitive-list except state_next
process(input1, input2, ..., state_reg)
begin
    -- default outputs
    output1 <= <value>;
    output2 <= <value>
    ...
    case state_reg is
    when s0 =>
        output1 <= <value>;
        output2 <= <value>
    ...
    when s1 =>
        output1 <= <value>;
        output2 <= <value>
    ...
    when s2 => ... end case;
end process;

-- optional D-FF to remove glitches
process(clk, reset)
begin
    if reset = '1' then
        new_output1 <= ... ;
        new_output2 <= ... ;
    elsif rising_edge(clk) then
        new_output1 <= output1;
        new_output2 <= output2;
    end if;
end process;
end architecture;

9.5.3 Recursive machine

In recursive machine, the outputs are fed back as input to the system (see Fig. 9.10). Hence, we need additional process-statement which can store the outputs which are fed back to combinational block of sequential design, as shown in Listing 9.9. The listing is same as Listing 9.8 except certain signals and process block are defined to feedback the output to combination logic; i.e. Lines 31-33 contain the signals (feedback registers) which are required to feedback the outputs. Here, ‘_next’ and ‘_reg’ are used in these lines, where ‘next’ value is fed back as ‘reg’ in the next clock cycle inside the process statement which is defined in Lines 64-75. Lastly, ‘feedback registers’ are also used to calculate the next-state inside the ‘if statement’ e.g. Lines 89 and 94. Also, the value of
feedback registers are updated inside these ‘if statements’ e.g. Lines 96 and 100.

**Note:** In recursive Moore machine,

- Outputs depend on current external inputs. Also, values in the feedback registers are used as outputs.
- Next states depend current states, current external input, current internal inputs (i.e. previous outputs feedback as inputs to system) and time (optional).

---

Listing 9.9: VHDL template recursive Moore FSM : separate ‘next_state’ and ‘output’ logic

```vhdl
-- moore_recursive_template.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity moore_recursive_template is
  generic (
    param1 : std_logic_vector(...) := <value>;
    param2 : unsigned(...) := <value>
  );
  port (
    clk, reset : in std_logic;
    input1, input2, ... : in std_logic_vector(...);
    -- inout is used here as we are reading output1 and output2
    new_output1, new_output2, output1, output2, ... : inout signed(...)
  );
end entity;

architecture arch of moore_recursive_template is
  type stateType is (s0, s1, s2, s3, ...);
  signal state_reg, state_next : stateType;

  -- timer (optional)
  constant T1 : natural := <value>;
  constant T2 : natural := <value>;
  constant T3 : natural := <value>;
  ...
  constant tmax : natural := <value>; -- tmax >= max(T1, T2, ...)-1
  signal t : natural range 0 to tmax;

  -- recursive : feedback register
  signal r1_reg, r1_next : std_logic_vector(...) := <value>;
  signal r2_reg, r2_next : signed(...) := <value>;
  ...
begin
  -- state register : state_reg
  -- This process contains sequential part and all the D-FF are
  -- included in this process. Hence, only 'clk' and 'reset' are
  -- required for this process.
  process(clk, reset)
  begin
    if reset = '1' then
      state_reg <= s1;
    elsif (clk) then
      state_reg <= state_next;
    end if;
    end process;

  -- timer (optional)
```

(continues on next page)
9.5. Moore architecture and VHDL templates

(continued from previous page)

```vhdl
process(clk, reset)
begin
  if reset = '1' then
    t <= 0;
  elsif rising_edge(clk) then
    if state_reg /= state_next then -- state is changing
      t <= 0;
    elsif t /= tmax then
      t <= t + 1;
    end if;
  end if;
end process;

-- feedback registers: to feedback the outputs
process(clk, reset)
begin
  if (reset = '1') then
    r1_reg <= <initial_value>;
    r2_reg <= <initial_value>;
  elsif rising_edge(clk) then
    r1_reg <= r1_next;
    r2_reg <= r2_next;
  end if;
end process;

-- next state logic : state_next
-- This is combinational of the sequential design,
-- which contains the logic for next-state
-- include all signals and input in sensitive-list except state_next
process(input1, input2, state_reg)
begin
  state_next <= state_reg; -- default state_next
  r1_next <= r1_reg; -- default next-states
  r2_next <= r2_reg;
  case state_reg is
  when s0 =>
    if <condition> and r1_reg = <value> and t >= T1-1 then -- if (input1 = '01') then
      state_next <= s1;
      r1_next <= <value>;
      r2_next <= <value>;
    elsif <condition> and r2_reg = <value> and t >= T2-1 then -- add all the required, conditions
      state_next <= <value>;
      r1_next <= <value>;
    else -- remain in current state
      state_next <= s0;
      r2_next <= <value>;
    end if;
  when s1 =>
    ... end case;
end process;

-- combination output logic
-- This part contains the output of the design
```

(continues on next page)
9.6 Mealy architecture and VHDL templates

Template for Mealy architecture is similar to Moore architecture. The minor changes are required as outputs depend on current input as well, as discussed in this section.

9.6.1 Regular machine

In Mealy machines, the output is the function of current input and states, therefore the output will also defined inside the if-statements (Lines 50-51 etc.). Rest of the code is same as Listing 9.7.

Listing 9.10: VHDL template for regular Mealy FSM : combined 'next_state' and 'output' logic

```vhdl
-- mealy_regular_template.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mealy_regular_template is
  generic (
    param1 : std_logic_vector(...) := <value>
  );
begin
  process(input1, input2, ..., state_reg)
  begin
    -- default outputs
    output1 <= <value>;
    output2 <= <value>;
    ...
    case state_reg is
      when s0 =>
        output1 <= <value>;
        output2 <= <value>;
        ...
      when s1 =>
        output1 <= <value>;
        output2 <= <value>;
        ...
      when s2 =>
        ...
    end case;
  end process;

  -- optional D-FF to remove glitches
  process(clk, reset)
  begin
    if reset = '1' then
      new_output1 <= ...;
      new_output2 <= ...;
    elsif rising_edge(clk) then
      new_output1 <= output1;
      new_output2 <= output2;
    end if;
  end process;
end architecture;
```
param2 : unsigned(...) := <value>
);
port (
  clk, reset : in std_logic;
  input1, input2, ... : in std_logic_vector(...);
  output1, output2, ... : out signed(...) 
);
end entity;

architecture arch of mealy_regular_template is
  type stateType is (s0, s1, s2, s3, ...);
  signal state_reg, state_next : stateType;
begin
  -- state register : state_reg
  -- This process contains sequential part and all the D-FF are
  -- included in this process. Hence, only 'clk' and 'reset' are
  -- required for this process.
  process(clk, reset)
  begin
    if reset = '1' then
      state_reg <= s1;
    elsif (clk) then
      state_reg <= state_next;
    end if;
  end process;

  -- next state logic and outputs
  -- This is combinational of the sequential design,
  -- which contains the logic for next-state and outputs
  -- include all signals and input in sensitive-list except state_next
  process(input1, input2, ..., state_reg)
  begin
    state_next <= state_reg; -- default state_next
    output1 <= <value>;
    output2 <= <value>;
    ...
    case state_reg is
    when s0 =>
      if <condition> then -- if (input1 = '01') then
        output1 <= <value>;
        output2 <= <value>;
      ...
      state_next <= s1;
    elsif <condition> then -- add all the required conditionstions
      output1 <= <value>;
      output2 <= <value>;
      ...
      state_next <= ...;
    else -- remain in current state
      output1 <= <value>;
      output2 <= <value>;
      ...
      state_next <= s0;
    end if;
    when s1 =>
      ...
    end case;
  end process;

  -- optional D-FF to remove glitches
  (continues on next page)
9.6.2 Timed machine

Listing 9.11 contains timer related changes in Listing 9.10. See description of Listing 9.8 for more details.

Listing 9.11: VHDL template for timed Mealy FSM : combined ‘next_state’ and ‘output’ logic
9.6. Mealy architecture and VHDL templates

(continued from previous page)

```vhdl
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
end process;

-- timer
process(clk, reset)
begin
  if reset = '1' then
    t <= 0;
  elsif rising_edge(clk) then
    if state_reg /= state_next then -- state is changing
      t <= 0;
    else
      t <= t + 1;
    end if;
  end if;
end process;

-- next state logic and outputs
-- This is combinational of the sequential design,
-- which contains the logic for next-state and outputs
-- include all signals and input in sensitive-list except state_next
process(input1, input2, ..., state_reg)
begin
  state_next <= state_reg; -- default state_next
  -- default outputs
  output1 <= <value>;
  output2 <= <value>;
  ...
  case state_reg is
  when s0 =>
    if <condition> and t >= T1-1 then -- if (input1 = '01') then
      output1 <= <value>;
      output2 <= <value>;
      ...
      state_next <= s1;
    elsif <condition> and t >= T2-1 then -- add all the required conditions
      output1 <= <value>;
      output2 <= <value>;
      ...
      state_next <= ...;
    else -- remain in current state
      output1 <= <value>;
      output2 <= <value>;
      ...
      state_next <= s0;
    end if;
  when s1 =>
    ...
  end case;
end process;

-- optional D-FF to remove glitches
process(clk, reset)
begin
  if reset = '1' then
    new_output1 <= ... ;
    new_output2 <= ... ;
  elsif rising_edge(clk) then
    new_output1 <= output1;
    new_output2 <= output2;
  end if;
end process;
```

(continues on next page)
9.6.3 Recursive machine


Listing 9.12: VHDL template for recursive Mealy FSM: combined 'next_state' and 'output' logic

```vhdl
-- mealy_recursive_template.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mealy_recursive_template is
  generic (
    param1 : std_logic_vector(...) := <value>
    param2 : unsigned(...) := <value>
  );
  port (clk, reset : in std_logic;
        input1, input2, ... : in std_logic_vector(...);
        -- inout is used here as we are reading output1 and output2
        new_output1, new_output2, output1, output2, ... : inout signed(...)
  );
end entity;

architecture arch of mealy_recursive_template is
  type stateType is (s0, s1, s2, s3, ...);
  signal state_reg, state_next : stateType;

  -- timer (optional)
  constant T1 : natural := <value>
  constant T2 : natural := <value>
  constant T3 : natural := <value>
  ...
  constant tmax : natural := <value> -- tmax >= max(T1, T2, ...)-1
  signal t : natural range 0 to tmax;

  -- recursive : feedback register
  signal r1_reg, r1_next : std_logic_vector(...) := <value>
  signal r2_reg, r2_next : signed(...) := <value>
  ...

begin
  -- state register : state_reg
  -- This process contains sequential part and all the D-FF are
  -- included in this process. Hence, only 'clk' and 'reset' are
  -- required for this process.
  process(clk, reset)
  begin
    if reset = '1' then
      state_reg <= s1;
    elsif (clk) then
      state_reg <= state_next;
    end if;
  end process;

  -- timer (optional)
```

(continues on next page)
9.6. Mealy architecture and VHDL templates

```vhdl
process(clk, reset)
begin
  if reset = '1' then
    t <= 0;
  elsif rising_edge(clk) then
    if state_reg /= state_next then -- state is changing
      t <= 0;
    elsif t /= tmax then
      t <= t + 1;
    end if;
  end if;
end process;

-- feedback registers: to feedback the outputs
process(clk, reset)
begin
  if (reset = '1') then
    r1_reg <= <initial_value>
    r2_reg <= <initial_value>;
  elsif rising_edge(clk) then
    r1_reg <= r1_next;
    r2_reg <= r2_next;
  end if;
end process;

-- next state logic and outputs
-- This is combinational part of the sequential design,
-- which contains the logic for next-state and outputs
-- include all signals and input in sensitive-list except state_next
process(input1, input2, ..., state_reg)
begin
  state_next <= state_reg; -- default state_next
  output1 <= <value>;
  output2 <= <value>;
  ...
  -- default next-states
  r1_next <= r1_reg;
  r2_next <= r2_reg;
  ...
  case state_reg is
    when s0 =>
      if <condition> and r1_reg = <value> and t >= T1-1 then -- if (input1 = '01') then
        output1 <= <value>;
        output2 <= <value>;
        ...
        r1_next <= <value>;
        r2_next <= <value>;
        ...
        state_next <= s1;
      elsif <condition> and r2_reg = <value> and t >= T2-1 then -- add all the required conditions
        output1 <= <value>;
        output2 <= <value>;
        ...
        r1_next <= <value>;
        ...
        state_next <= ...;
      else -- remain in current state
      end case;
end process;
```

(continues on next page)
output1 <= <value>;
output2 <= <value>;
... 
r2_next <= <value>);
... 
state_next <= s0;
end if;
when s1 =>
  ...
end case;
end process;

-- optional D-FF to remove glitches
process(clk, reset)
begin
  if reset = '1' then
    new_output1 <= ... ;
    new_output2 <= ... ;
  elsif rising_edge(clk) then
    new_output1 <= output1;
    new_output2 <= output2;
  end if;
end process;
end architecture;

9.7 Examples

9.7.1 Regular Machine : Glitch-free Mealy and Moore design

In this section, a non-overlapping sequence detector is implemented to show the differences between Mealy and Moore machines. Listing 9.13 implements the ‘sequence detector’ which detects the sequence ‘110’; and corresponding state-diagrams are shown in Fig. 9.11. The RTL view generated by the listing is shown in Fig. 9.14, where two D-FF are added to remove the glitches from Moore and Mealy model. Also, in the figure, if we click on the state machines, then we can see the implemented state-diagrams e.g. if we click on ‘state_reg_mealy’ then the state-diagram in Fig. 9.13 will be displayed, which is exactly same as Fig. 9.12.

Further, the testbench for the listing is shown in Listing 9.14, whose results are illustrated in Fig. 9.15. Please note the following points in Fig. 9.15,

- Mealy machines are asynchronous as the output changes as soon as the input changes. It does not wait for the next cycle.
- If the output of the Mealy machine is delayed, then glitch will be removed and the output will be same as the Moore output. Note that, there is no glitch in this system. This example shows how to implement the D-FF to remove glitch in the system (if exists).
- Glitch-free Moore output is delayed by one clock cycle.
- If glitch is not a problem, then we should use Moore machine, because it is synchronous in nature. But, if glitch is problem and we do not want to delay the output then Mealy machines should be used.

Listing 9.13: Glitch removal using D-FF

```vhdl
-- sequence_detector.vhd
-- non-overlap detection : 110
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity sequence_detector is
```
9.7. Examples

Fig. 9.11: Non-overlap sequence detector: 110 (Moore design)

Fig. 9.12: Non-overlap sequence detector: 110 (Mealy design)

Fig. 9.13: State diagram generated by Quartus for Mealy machine in Listing 9.13
Fig. 9.14: RTL view generated by Listing 9.13

Fig. 9.15: Mealy and Moore machine output for Listing 9.14
9.7. Examples

(continued from previous page)

```vhdl
port(
    clk, reset : in std_logic;
    x : in std_logic;
    z_mealy_glitch, z_moore_glitch : out std_logic;
    z_mealy_glitch_free, z_moore_glitch_free : out std_logic
); end entity;

architecture arch of sequence_detector is

-- Moore

type stateType_moore is (zero_moore, one_moore, two_moore, three_moore);
signal state_reg_moore, state_next_moore : stateType_moore;

-- Mealy

type stateType_mealy is (zero_mealy, one_mealy, two_mealy);
signal state_reg_mealy, state_next_mealy : stateType_mealy;

signal z_moore, z_mealy : std_logic;

begin
    process(clk, reset)
    begin
        if reset = '1' then
            state_reg_moore <= zero_moore;
            state_reg_mealy <= zero_mealy;
        elsif rising_edge(clk) then
            state_reg_moore <= state_next_moore;
            state_reg_mealy <= state_next_mealy;
        end if;
    end process;

    -- Moore
    process(state_reg_moore, x)
    begin
        z_moore <= '0';
        state_next_moore <= state_reg_moore; -- default 'next state' is 'current state'
        case state_reg_moore is
            when zero_moore =>
                if x = '1' then
                    state_next_moore <= one_moore;
                end if;
            when one_moore =>
                if x = '1' then
                    state_next_moore <= two_moore;
                else
                    state_next_moore <= zero_moore;
                end if;
            when two_moore =>
                if x = '0' then
                    state_next_moore <= three_moore;
                end if;
            when three_moore =>
                z_moore <= '1';
                if x = '0' then
                    state_next_moore <= zero_moore;
                else
                    state_next_moore <= one_moore;
                end if;
        end case;
    end process;

    -- Mealy

    (continues on next page)
```

(continues on next page)
Chapter 9. Finite state machines

(continued from previous page)

69  process(state_reg_mealy, x)
70  begin
71      z_mealy <= '0';
72      state_next_mealy <= state_reg_mealy;  -- default 'next state' is 'current state'
73      case state_reg_mealy is
74          when zero_mealy =>
75              if x = '1' then
76                  state_next_mealy <= one_mealy;
77              end if;
78          when one_mealy =>
79              if x = '1' then
80                  state_next_mealy <= two_mealy;
81              else
82                  state_next_mealy <= zero_mealy;
83              end if;
84          when two_mealy =>
85              state_next_mealy <= zero_mealy;
86              if x = '0' then
87                  z_mealy <= '1';
88              else
89                  state_next_mealy <= two_mealy;
90              end if;
91      end case;
92  end process;
93
94  -- D-FF to remove glitches
95  process(clk, reset)
96  begin
97      if reset = '1' then
98          z_mealy_glitch_free <= '0';
99          z_moore_glitch_free <= '0';
100      elsif rising_edge(clk) then
101          z_mealy_glitch_free <= z_mealy;
102          z_moore_glitch_free <= z_moore;
103      end if;
104  end process;
105  z_mealy_glitch <= z_mealy;
106  z_moore_glitch <= z_moore;
107  end architecture;


-- sequence_detector_tb.vhd

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity sequence_detector_tb is
5      end sequence_detector_tb;
6
7  architecture arch of sequence_detector_tb is
8      constant T : time := 20 ps;
9      signal clk, reset : std_logic;  -- input
10      signal x : std_logic;
11      signal z_moore_glitch, z_moore_glitch_free : std_logic;
12      signal z_mealy_glitch, z_mealy_glitch_free : std_logic;
13  begin
14      sequence_detector_unit : entity work.sequence_detector
15      (continues on next page)
9.7. Examples

(continued from previous page)

```vhdl
port map (clk=>clk, reset=>reset, x=>x,
    z_moore_glitch=>z_moore_glitch, z_moore_glitch_free => z_moore_glitch_free,
    z_mealy_glitch=>z_mealy_glitch, z_mealy_glitch_free => z_mealy_glitch_free
);

-- continuous clock
process
begin
    clk <= '0';
    wait for T/2;
    clk <= '1';
    wait for T/2;
end process;

-- reset = 1 for first clock cycle and then 0
reset <= '1', '0' after T/2;

x <= '0', '1' after T, '1' after 2*T, '0' after 3*T;
end;
```

9.7.2 Timed machine: programmable square wave

Listing 9.15 generates the square wave using Moore machine, whose ‘on’ and ‘off’ time is programmable (see Lines 10 and 11) according to state-diagram in Fig. 9.16. The simulation waveform of the listing are shown in Fig. 9.17.

![State diagram for programmable square-wave generator](image1.png)

![Moore Simulation waveform of Listing 9.15](image2.png)

Listing 9.15: Square wave generator

```vhdl
-- square_wave_ex.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity square_wave_ex is
    generic(
        N : natural := 4; -- Number of bits to represent the time
    on_time : signed(3 downto 0) := "0101";
    off_time : signed(3 downto 0) := "0011"
    );
    port(
        clk: in std_logic;
        reset: in std_logic;
        x: in std_logic;
        z_moore_glitch: out std_logic;
        z_moore_glitch_free: out std_logic;
        z_mealy_glitch: out std_logic;
        z_mealy_glitch_free: out std_logic
    );
end entity square_wave_ex;
```

(continues on next page)
9.7.3 Recursive Machine : Mod-m counter

Listing 9.16 implements the Mod-m counter using Moore machine, whose state-diagram is shown in Fig. 9.18. Machine is recursive because the output signal ‘count_moore_reg’ (Line 52) is used as input to the system (Line 35). The simulation waveform of the listing are shown in Fig. 9.19

Note: It is not good to implement every design using FSM e.g. Listing 9.16 can be easily implement without FSM as shown in Listing 8.4. Please see the Section 9.8 for understading the correct usage of FSM design.
9.7. Examples

Fig. 9.18: State diagram for Mod-m counter

Fig. 9.19: Simulation waveform of Listing 9.16

Listing 9.16: Mod-m Counter

```vhdl
-- counterEx.vhd
-- count upto M

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counterEx is
  generic(
    M : natural := 6;
    N : natural := 4 -- N bits are required for M
  );
  port(
    clk, reset : in std_logic;
    out_moore : out std_logic_vector(N-1 downto 0)
  );
end entity;

architecture arch of counterEx is
  type stateType_moore is (start_moore, count_moore);
  signal state_moore_reg, state_moore_next : stateType_moore;
  signal count_moore_reg, count_moore_next : unsigned(N-1 downto 0);
  begin
  process(clk, reset)
  begin
    if reset = '1' then
      state_moore_reg <= start_moore;
      count_moore_reg <= (others => '0');
    elsif rising_edge(clk) then
      state_moore_reg <= state_moore_next;
      count_moore_reg <= count_moore_next;
    end if;
  end process;

  process(count_moore_reg, state_moore_reg)
  begin
    case state_moore_reg is
      when start_moore =>
        count_moore_next <= (others => '0');
      when others =>
        count_moore_next <= (others => '0');
    end case;
  end process;
end arch;
```

(continues on next page)
9.8 When to use FSM design

We saw in previous sections that, once we have the state diagram for the FSM design, then the VHDL design is a straightforward process. But, it is important to understand the correct conditions for using the FSM, otherwise the circuit will become complicated unnecessarily.

- We should not use the FSM diagram, if there is only ‘one loop’ with ‘zero or one control input’. ‘Counter’ is a good example for this. A 10-bit counter has 10 states with no control input (i.e. it is free running counter). This can be easily implemented without using FSM as shown in Listing 8.3. If we implement it with FSM, then we need 10 states; and the code and corresponding design will become very large.
- If required, then FSM can be use for ‘one loop’ with ‘two or more control inputs’.
- FSM design should be used in the cases where there are very large number of loops (especially connected loops) along with two or more controlling inputs.

9.9 Conclusion

In this chapter, Mealy and Moore designs are discussed. Also, ‘edge detector’ is implemented using Mealy and Moore designs. This example shows that Mealy design requires fewer states than Moore design. Further, Mealy design generates the output tick as soon as the rising edge is detected; whereas Moore design generates the output tick after a delay of one clock cycle. Therefore, Mealy designs are preferred for synchronous designs.
Chapter 10

Testbenches

10.1 Introduction

In previous chapters, we generated the simulation waveforms using modelsim, by providing the input signal values manually; if the number of input signals are very large and/or we have to perform simulation several times, then this process can be quite complex, time consuming and irritating. Suppose input is of 10 bit, and we want to test all the possible values of input i.e. $2^{10} - 1$, then it is impossible to do it manually. In such cases, testbenches are very useful; also, tested design more reliable and prefer by the other clients as well. Further, with the help of testbenches, we can generate results in the form of csv (comma separated file), which can be used by other softwares for further analysis e.g. Python, Excel and Matlab etc.

Since testbenches are used for simulation purpose only (not for synthesis), therefore full range of VHDL constructs can be used e.g. keywords ‘assert’, ‘report’ and ‘for loops’ etc. can be used for writing testbenches.

Modelsim-project is created in this chapter for simulations, which allows the relative path to the files with respect to project directory as shown in Section 10.2.5. Simulation can be run without creating the project, but we need to provide the full path of the files as shown in Lines 30-34 of Listing 10.5.

Lastly, mixed modeling is not supported by Altera-Modelsim-starter version, i.e. Verilog designs with VHDL and vice-versa can not be compiled in this version of Modelsim.

10.2 Testbench for combinational circuits

In this section, various testbenches for combinational circuits are shown, whereas testbenches for sequential circuits are discussed in next section. For simplicity of the codes and better understanding, a simple half adder circuit is tested using various simulation methods.

10.2.1 Half adder

Listing 10.1 shows the VHDL code for the half adder which is tested using different ways,

Listing 10.1: Half adder

```
-- half_adder.vhd

library ieee;
use ieee.std_logic_1164.all;

entity half_adder is
  port (a, b : in std_logic;
```

(continues on next page)
10.2.2 Simple testbench

Note that, testbenches are written in separate VHDL files as shown in Listing 10.2. Simplest way to write a testbench, is to invoke the ‘design for testing’ in the testbench and provide all the input values in the file, as explained below.

Explanation Listing 10.2

In this listing, a testbench with name ‘half_adder_simple_tb’ is defined at Lines 7-8. Note that, entity of testbench is always empty i.e. no ports are defined in the entity (see Lines 7-8). Then 4 signals are defined i.e. a, b, sum and carry (Lines 11-12) inside the architecture body; these signals are then connected to actual half adder design using structural modeling (see Line 15). Lastly, different values are assigned to input signals e.g. ‘a’ and ‘b’ at lines 16 and 17 respectively.

In Line 22, value of ‘a’ is 0 initially (at 0 ns), then it changes to ‘1’ at 20 ns and again changes to ‘0’ at 40 ns (do not confuse with after 40 ns, as after 40 ns is with respect to 0 ns, not with respect to 20 ns). Similarly, the values of ‘a’ becomes ‘0’ and ‘1’ at 40 and 60 ns respectively. In the same way value of ‘b’ is initially ‘0’ and change to ‘1’ at 40 ns at Line 23. In this way 4 possible combination are generated for two bits (‘ab’) i.e. 00, 01, 10 and 11 as shown in Fig. 10.1; also corresponding outputs, i.e. sum and carry, are shown in the figure.

To generate the waveform, first compile the ‘half_adder.vhd and then ‘half_adder_simple_tb.vhd’ (or compile both the file simultaneously.). Then simulate the half_adder_simple_tb.vhd file. Finally, click on ‘run all’ button (which will run the simulation to maximum time i.e. 60 ns here at Line 22) and then click then ‘zoom full’ button (to fit the waveform on the screen), as shown in Fig. 10.1.

Problem: Although, the testbench is very simple, but input patterns are not readable. By using the process statement in the testbench, we can make input patterns more readable along with inclusion of various other features e.g. report generation etc., as shown in next section.

Listing 10.2: Simple testbench for half adder
10.2. Testbench for combinational circuits

(continued from previous page)

```
18
-- 00 at 0 ns
19
-- 01 at 20 ns, as b is 0 at 20 ns and a is changed to 1 at 20 ns
20
-- 10 at 40 ns
21
-- 11 at 60 ns
22
a <= '0', '1' after 20 ns, '0' after 40 ns, '1' after 60 ns;
23
b <= '0', '1' after 40 ns;
24
end tb;
```

Fig. 10.1: Simulation results for Listing 10.2

10.2.3 Testbench with process statement

In Listing 10.3, process statement is used in the testbench; which includes the input values along with the corresponding output values. If the specified outputs are not matched with the output generated by half-adder, then errors will be generated. Note that, process statement is written without the sensitivity list.

Explanation Listing 10.3

The listing is same as previous listing till Line 15, and then process statement is used to define the input patterns, which can be seen at lines 20-21 (00), 27-28 (01), 33-34 (10) and 39-40 (11). Further, expected outputs are shown below these lines e.g. line 23 shows that the sum is 0 and carry is 0 for input 00; and if the generated output is different from these values, e.g. error is generated by line 50 for input pattern 01 as shown in Fig. 10.3; as sum generated by half_adder for line 46-47 is 1, whereas expected sum is defined as 0 for this combination at line 49. Note that, the adder output is correct, whereas the expected value is entered incorrectly; and error is displayed on ‘transcript window’ of modelsim.

Also, ‘period’ is defined as 20 ns at Line 18; and then used after each input values e.g line 22, which indicates that input will be displayed for 20 ns before going to next input values (see in Fig. 10.3).

Listing 10.3: Testbench with process statement

```
-- half_adder_process_tb.vhd

library ieee;
use ieee.std_logic_1164.all;

entity half_adder_process_tb is
end half_adder_process_tb;

architecture tb of half_adder_process_tb is
  signal a, b : std_logic;
  signal sum, carry : std_logic;
begin
  -- connecting testbench signals with half_adder.vhd
  UUT : entity work.half_adder port map (a => a, b => b, sum => sum, carry => carry);
```

(continues on next page)
### 10.2.4 Testbench with look-up table

The inputs patterns can be defined in the form of look-table as well as shown in Listing 10.4, instead of define separately at different location as done in Listing 10.3 e.g. at lines 20 and 27 etc.

**Explanation Listing 10.4**

```vhdl
-- Fail test
a <= '1';
b <= '1';
wait for period;
assert ((sum = '0') and (carry = '1'))
report "test failed for input combination 01 (fail test)" severity error;

wait; -- indefinitely suspend process
end process;
end tb;
```

![Simulation results for Listing 10.3](image)
10.2. Testbench for combinational circuits

Fig. 10.3: Error generated by Listing 10.3

Basic concept of this Listing is similar to Listing 10.3 but written in different style. Testbench with lookup table can be written using three steps as shown below,

- **Define record**: First we need to define a record which contains all the possible columns in the look-up table. Here, there are four possible columns i.e. `a`, `b`, `sum`, and `carry`, which are defined in record at Lines 15-18.

- **Create lookup table**: Next, we need to define the lookup table values, which is done at Lines 20-28. Here positional method is used for assigning values to columns (see line 22-27); further, name-association method can also be used as shown in the comment at Line 23.

- **Assign values to signals**: Then the values of the lookup table need to be assigned to half_adder entity (one by one). For this ‘for loop’ is used at line 35, which assigns the values of ‘test-vector’s `a` and `b` to signal `a` and `b` (see comment at Line 36 for better understanding). Similarly, expected values of `sum` and `carry` are generated at Lines 41-44. Lastly, report is generated for wrong outputs at Lines 46-50.

The simulations results and reported-error are shown in Fig. 10.4 and Fig. 10.5 respectively.

Listing 10.4: Testbench with look-up table

```vhdl
-- half_adder_lookup_tb.vhd

library ieee;
use ieee.std_logic_1164.all;

data half_adder_lookup_tb is
end half_adder_lookup_tb;

architecture tb of half_adder_lookup_tb is

signal a, b : std_logic; -- input
signal sum, carry : std_logic; -- output

-- declare record type
type test_vector is record
  a, b : std_logic;
  sum, carry : std_logic;
end record;

type test_vector_array is array (natural range <>) of test_vector;
constant test_vectors : test_vector_array := (
  -- positional method is used below
  ('0', '0', '0', '0'), -- or (a => '0', b => '0', sum => '0', carry => '0')
  ('0', '1', '1', '0'),
  ('1', '0', '1', '0'),
  ('1', '1', '0', '1'),
  ('0', '1', '0', '1') -- fail test
);

begin
  UUT : entity work.half_adder port map (a => a, b => b, sum => sum, carry => carry);
  tbl : process
```
begin
  for i in test_vectors’range loop
    a <= test_vectors(i).a; -- signal a = i-th-row-value of test_vector’s a
    b <= test_vectors(i).b;

    wait for 20 ns;

    assert (  
      (sum = test_vectors(i).sum) and 
      (carry = test_vectors(i).carry)  
    )  

    -- image is used for string-representation of integer etc.
    report "test_vector " & integer’image(i) & " failed " & 
      " for input a = " & std_logic’image(a) & 
      " and b = " & std_logic’image(b) 
    severity error;
  end loop;
  wait;
end process;
end tb;

Fig. 10.4: Simulation results for Listing 10.4

```
$VSM 40> run -all  
# ** Error: test_vector 4 failed for input a = '0' and b = '1'
# Time: 100 ns Iteration: 0 Instance: /half_adder_lookup_tb
```

Fig. 10.5: Error generated by Listing 10.4

### 10.2.5 Read data from file

In this section, data from file ‘read_file_ex.txt’ is read and displayed in simulation results. Date stored in the file is shown in Fig. 10.6.

```
0 0 11
0 1 01
1 0 11
```

Fig. 10.6: Data in file ‘read_file_ex.txt’

**Explanation Listing 10.5**

To read the file, first we need to define a buffer of type ‘text’, which can store the values of the file in it, as shown in Line 17; file is open in read-mode and values are stored in this buffer at Line 32.

Next, we need to define the variable to read the value from the buffer. Since there are 4 types of values (i.e. a, b, c and spaces) in file ‘read_file_ex.txt’, therefore we need to define 4 variables to store them,
as shown in Line 24-26. Since, variable \( c \) is of 2 bit, therefore Line 25 is 2-bit vector; further, for spaces, variable of character type is defined at Line 26.

Then, values are read and store in the variables at Lines 36-42. Lastly, these values are assigned to appropriate signals at Lines 45-47. Finally, file is closed at Line 52. The simulation results of the listing are show in Fig. 10.7.

Listing 10.5: Read data from file

```vhdl
-- read_file_ex.vhd

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use ieee.std_logic_textio.all; -- require for writing/reading std_logic etc.

entity read_file_ex is
end read_file_ex;

architecture tb of read_file_ex is

signal a, b : std_logic;
signal c : std_logic_vector(1 downto 0);

-- buffer for storing the text from input read-file
file input_buf : text; -- text is keyword

begin

begin

-- if modelsim-project is created, then provide the relative path of
-- input-file (i.e. read_file_ex.txt) with respect to main project folder
file_open(input_buf, "VHDLCodes/input_output_files/read_file_ex.txt", read_mode);
-- else provide the complete path for the input file as show below
-- file_open(input_buf, "E:/VHDLCodes/input_output_files/read_file_ex.txt", read_mode);

while not endfile(input_buf) loop
    readline(input_buf, read_col_from_input_buf);
    read(read_col_from_input_buf, val_col1);
    read(read_col_from_input_buf, val_SPACE);
    read(read_col_from_input_buf, val_col2);
    read(read_col_from_input_buf, val_SPACE);
    read(read_col_from_input_buf, val_col3);
    read(read_col_from_input_buf, val_SPACE);

    -- Pass the read values to signals
    a <= val_col1;
b <= val_col2;
c <= val_col3;

    wait for 20 ns; -- to display results for 20 ns
end loop;

file_close(input_buf);
wait;
end process;

(continues on next page)
```
Chapter 10. Testbenches

10.2.6 Write data to file

In this part, different types of values are defined in Listing 10.6 and then stored in the file. Here, only ‘write_mode’ is used for writing the data to file (not the ‘append_mode’).

Explanation Listing 10.6

To write the data to the file, first we need to define a buffer, which will load the file on the simulation environment for writing the data during simulation, as shown in Line 15 (buffer-defined) and Line 27 (load the file to buffer).

Next, we need to define a variable, which will store the values to write into the buffer, as shown in Line 19. In the listing, this variable stores three types of value i.e. strings (Lines 31 and 34 etc.), signal ‘a’ (Line 35) and variable ‘b’ (Line 37).

Note that, two keyword are used for writing the data into the file i.e. ‘write’ and ‘writeline’. ‘write’ keyword store the values in the ‘write_col_to_output_buf’ and ‘writeline’ writes the values in the file. Remember that, all the ‘write’ statements before the ‘writeline’ will be written in same line e.g. Lines 34-37 will be written in same line as shown in Fig. 10.8. Lastly, the simulation result for the listing is shown in Fig. 10.9.

Listing 10.6: Write data to file

```
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use ieee.std_logic_textio.all;  -- require for writing std_logic etc.
entity write_file_ex is
description write_file_ex is
end write_file_ex;
architecture tb of write_file_ex is
signal a : std_logic;
variable write_col_to_output_buf : line;  -- line is keyword
variable b : integer := 40;
begin
  a <= '1';  -- assign value to a
  wait for 20 ns;
  -- if modelsim-project is created, then provide the relative path of
  -- input-file (i.e. read_file_ex.txt) with respect to main project folder
```

(continues on next page)
10.2. Testbench for combinational circuits

(file_open(output_buf, "VHDLCodes/input_output_files/write_file_ex.txt", write_mode);
-- else provide the complete path for the input file as show below
--file_open(output_buf, "E:/VHDLCodes/input_output_files/write_file_ex.txt", write_mode);

write(write_col_to_output_buf, string("Printing values"));
write(output_buf, write_col_to_output_buf); -- write in line 1

write(write_col_to_output_buf, string("a = "));
write(write_col_to_output_buf, a);
write(write_col_to_output_buf, string(" , b = "));
write(write_col_to_output_buf, b);
write(output_buf, write_col_to_output_buf); -- write in new line 2

write(write_col_to_output_buf, string("Thank you"));
write(output_buf, write_col_to_output_buf); -- write in new line 3

file_close(output_buf);
wait; -- indefinitely suspend process
end process;
end tb ; -- tb

Fig. 10.8: Data in file ‘write_file_ex.txt’

(a = 1, b = 40)

Thank you

Fig. 10.9: Simulation results of Listing 10.6

10.2.7 Half adder testing using CSV file

In this section, both read and write operations are performed in Listing 10.7. Further, csv file is used for read and write operations. Content of input and output files are shown in Fig. 10.11 and Fig. 10.12 respectively.

Please read Listing 10.5 and Listing 10.6 to understand this part, as only these two listings are merged together here.

Addition features added to listing are shown below,

- Lines 63-64 are added to skip the header row, i.e. any row which does not start with boolean-number (see line 42).
- Also, error will be reported for value ‘b’ if it is not the boolean. Similarly, this functionality can be added to other values as well.
- Lastly, errors are reported in CSV file at Lines 96-109. This can be seen in Fig. 10.12. It’s always easier to find the location of error using csv file as compare to simulation waveforms (try to find the errors using Fig. 10.12 and compare it with Fig. 10.12).
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use ieee.std_logic_textio.all; -- require for writing/reading std_logic etc.

entity read_write_file_ex is
end read_write_file_ex;

architecture tb of read_write_file_ex is
    signal a, b : std_logic;
    signal sum_actual, carry_actual : std_logic;
    signal sum, carry : std_logic; -- calculated sum and carry by half_adder

    -- buffer for storing the text from input and for output files
    file input_buf : text; -- text is keyword
    file output_buf : text; -- text is keyword

begin
    UUT : entity work.half_adder port map (a => a, b => b, sum => sum, carry => carry);

    tbl : process
        variable read_col_from_input_buf : line; -- read lines one by one from input_buf
        variable write_col_to_output_buf : line; -- write lines one by one to output_buf
        variable buf_data_from_file : line; -- buffer for storing the data from input read-file
        variable val_a, val_b : std_logic;
        variable val_sum, val_carry : std_logic;
        variable val_comma : character; -- for commas between data in file
        variable good_num : boolean;
        begin

        -- ####################################################################
        -- Reading data
        -- if modelsim-project is created, then provide the relative path of
        -- input-file (i.e. read_file_ex.txt) with respect to main project folder
        file_open(input_buf, "VHDLCodes/input_output_files/half_adder_input.csv", read_mode);
        -- else provide the complete path for the input file as show below
        -- file_open(input_buf, "E:/VHDLCodes/input_output_files/read_file_ex.txt", read_mode);

        -- writing data
        file_open(output_buf, "VHDLCodes/input_output_files/half_adder_output.csv", write_mode);
        write(write_col_to_output_buf, "#a,b,sum_actual,sum,carry_actual,carry,sum_test_results,carry_test_results");
        writeline(output_buf, write_col_to_output_buf);

    end process
end architecture;
while not endfile(input_buf) loop
   readline(input_buf, read_col_from_input_buf);
   read(read_col_from_input_buf, val_a, good_num);
   next when not good_num; -- i.e. skip the header lines

   read(read_col_from_input_buf, val_comma); -- read in the space character
   read(read_col_from_input_buf, val_b, good_num);
   assert good_num report "bad value assigned to val_b";

   read(read_col_from_input_buf, val_comma); -- read in the space character
   read(read_col_from_input_buf, val_sum);
   read(read_col_from_input_buf, val_comma); -- read in the space character
   read(read_col_from_input_buf, val_carry);

   -- Pass the variable to a signal to allow the ripple-carry to use it
   a <= val_a;
   b <= val_b;
   sum_actual <= val_sum;
   carry_actual <= val_carry;

   wait for 20 ns; -- to display results for 20 ns

   write(write_col_to_output_buf, a);
   write(write_col_to_output_buf, string(",",));
   write(write_col_to_output_buf, b);
   write(write_col_to_output_buf, string(",",));
   write(write_col_to_output_buf, sum_actual);
   write(write_col_to_output_buf, string(",",));
   write(write_col_to_output_buf, sum);
   write(write_col_to_output_buf, string(",",));
   write(write_col_to_output_buf, carry_actual);
   write(write_col_to_output_buf, string(",",));
   write(write_col_to_output_buf, carry);
   write(write_col_to_output_buf, string(",",));

   -- display Error or OK if results are wrong
   if (sum_actual /= sum) then
      write(write_col_to_output_buf, string("Error,"));
   else
      write(write_col_to_output_buf, string("", ")); -- write nothing
   end if;

   -- display Error or OK based on comparison
   if (carry_actual /= carry) then
      write(write_col_to_output_buf, string("Error,"));
   else
      write(write_col_to_output_buf, string("OK,"));
   end if;

   --write(write_col_to_output_buf, a, b, sum_actual, sum, carry_actual, carry);
   writeln(output_buf, write_col_to_output_buf);
end loop;

file_close(input_buf);
file_close(output_buf);
wait;
end process;
end tb; -- tb
Chapter 10. Testbenches

Fig. 10.10: Simulation results of Listing 10.7

<table>
<thead>
<tr>
<th>#a</th>
<th>b</th>
<th>sum</th>
<th>carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 10.11: Content of input file ‘half_adder_input.csv’

<table>
<thead>
<tr>
<th>#a</th>
<th>b</th>
<th>sum_actual</th>
<th>sum</th>
<th>carry_actual</th>
<th>carry</th>
<th>sum_test_results</th>
<th>carry_test_results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Error</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Error</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 10.12: Content of input file ‘half_adder_output.csv’
10.3 Testbench for sequential circuits

In Section 10.2.3, we saw the use of process statement for writing the testbench for combination circuits. But, in the case of sequential circuits, we need clock and reset signals; hence two additional blocks are required. Since, clock is generated for complete simulation process, therefore it is defined inside the separate process statement. Whereas, reset signal is required only at the beginning of the operations, hence it is not defined inside the process statement. Rest of the procedures/methods for writing the testbenches for sequential circuits are same as the testbenches of the combinational circuits.

10.3.1 Simulation with infinite duration

In this section, we have created a testbench which will not stop automatically i.e. if we press the ‘run all’ button then the simulation will run forever, therefore we need to press the ‘run’ button as shown in Fig. 10.13.

Explanation Listing 10.8

Listing 10.8 is the testbench for mod-M counter, which is discussed in Section 8.3.2. Here ‘clk’ signal is generated in the separate process block i.e. Lines 27-33; in this way, clock signal will be available throughout the simulation process. Further, reset signal is set to ‘1’ in the beginning and then set to ‘0’ in next clock cycle (Line 37). If there are further, inputs signals, then those signals can be defined in separate process statement, as discussed in combination circuits’ testbenches.

The simulation results are shown in Fig. 10.13, where counter values goes from 0 to 9 as M is set to 10 (i.e. A in hexadecimal). Further, use ‘run’ button for simulating the sequential circuits (instead of run-all), as shown in the figure.

Listing 10.8: Testbench with infinite duration for modM-Counter.vhd

```
-- modMCounter_tb.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity modMCounter_tb is
end modMCounter_tb;

architecture arch of modMCounter_tb is
  constant M : integer := 10;
  constant N : integer := 4;
  constant T : time := 20 ns;

  signal clk, reset : std_logic; -- input
  signal complete_tick : std_logic; -- output
  signal count : std_logic_vector(N-1 downto 0); -- output

begin
  modMCounter_unit : entity work.modMCounter
    generic map (M => M, N => N)
    port map (clk=>clk, reset=>reset, complete_tick=>complete_tick,
      count=>count);

  -- continuous clock
  process
    begin
      clk <= '0';
      wait for T/2;
      clk <= '1';
  end process;
```

(continues on next page)
Chapter 10. Testbenches

(continued from previous page)

```
wait for T/2;
end process;

-- reset = 1 for first clock cycle and then 0
reset <= '1', '0' after T/2;
end arch;
```

Fig. 10.13: Simulation results of Listing 10.8

## 10.3.2 Simulation for finite duration and save data

To run the simulation for the finite duration, we need to provide the ‘number of clocks’ for which we want to run
the simulation, as shown in Line 23 of Listing 10.9. Then at Lines 47-52 are added to close the file after desired
number of clocks i.e. ‘num_of_clocks’. Also, the data is saved into the file, which is discussed in Section 10.2.6.
Now, if we press the run all button, then the simulator will stop after ‘num_of_clocks’ cycles. Note that, if
the data is in ‘signed or unsigned’ format, then it can not be saved into the file. We need to change
the data into other format e.g. ‘integer’, ‘natural’ or ‘std_logic_vector’ etc. before saving it into
the file, as shown in Line 73. The simulation waveforms and saved results are shown in Fig. 10.14 and Fig.
10.15 respectively.

```
-- modMCounter_tb2.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all; -- require for std_logic etc.
entity modMCounter_tb2 is
end modMCounter_tb2;
```

```
end arch;
```

(continues on next page)

---

Meher Krishna Patel

---

129
10.3. Testbench for sequential circuits

Fig. 10.15: Partial view of saved data by Listing 10.9

<table>
<thead>
<tr>
<th>clock_tick</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

(continued from previous page)

architecture arch of modMCounter_tb2 is

constant M : integer := 3; -- count upto 2 (i.e. 0 to 2)
constant N : integer := 4;
constant T : time := 20 ns;

signal clk, reset : std_logic; -- input
signal complete_tick : std_logic; -- output
signal count : std_logic_vector(N-1 downto 0); -- output

-- total samples to store in file
constant num_of_clocks : integer := 30;

signal i : integer := 0; -- loop variable
file output_buf : text; -- text is keyword

begin

modMCounter_unit : entity work.modMCounter
generic map (M => M, N => N)
port map (clk=>clk, reset=>reset, complete_tick=>complete_tick,
          count=>count);

-- reset = 1 for first clock cycle and then 0
reset <= '1', '0' after T/2;

-- continuous clock
process
begin
  clk <= '0';
  wait for T/2;
  clk <= '1';
  wait for T/2;

-- store 30 samples in file
  if (i = num_of_clocks) then

(continues on next page)
file_close(output_buf);
wait;
else
  i <= i + 1;
end if;
end process;

-- save data in file : path is relative to Modelsim-project directory
file_open(output_buf, "input_output_files/counter_data.csv", write_mode);
process(clk)
  variable write_col_to_output_buf : line; -- isne is keyword
begin
  if(clk'event and clk='1' and reset /= '1') then -- avoid reset data
    -- comment below 'if statement' to avoid header in saved file
    if (i = 0) then
      write(write_col_to_output_buf, string("clock_tick,count"));
      writeline(output_buf, write_col_to_output_buf);
    end if;
    write(write_col_to_output_buf, complete_tick);
    write(write_col_to_output_buf, string(";"));
    -- Note that unsigned/signed values can not be saved in file,
    -- therefore change into integer or std_logic_vector etc.
    -- following line saves the count in integer format
    write(write_col_to_output_buf, to_integer(unsigned(count)));
    writeline(output_buf, write_col_to_output_buf);
  end if;
end process;
end arch;

10.4 Conclusion

In this chapter, we learn to write testbenches with different styles for combinational circuits. We saw the methods by which inputs can be read from the file and the outputs can be written in the file. Simulation results and expected results are compared and saved in the csv file and displayed as simulation waveforms; which demonstrated that locating the errors in csv files is easier than the simulation waveforms. Further, we saw the simulation of sequential circuits as well, which is slightly different from combination circuits; but all the methods of combinational circuit simulations can be applied to sequential circuits as well.
Chapter 11

Design examples

11.1 Introduction

In previous chapters, some simple designs were introduced e.g. mod-m counter and flip-flops etc. to introduce the VHDL programming. In this chapter, various examples are added, which can be used to implement or emulate a system on the FPGA board.

All the design files are provided inside the ‘VHDLCodes’ folder inside the main project directory; which can be used to implement the design using some other software as well. Each section shows the list of VHDL-files required to implement the design in that section. Lastly, all designs are tested using Modelsim and on Altera-DE2 FPGA board. Set a desired design as ‘top-level entity’ to implement or simulate it.

11.2 Random number generator

In this section, random number generator is implemented using linear feedback shift register. VHDL files required for this example are listed below,

- rand_num_generator.vhd
- rand_num_generator_visualTest.vhd
- clockTick.vhd
- modMCounter.vhd

Note that, ‘clockTick.vhd’ and ‘modMCounter.vhd’ are discussed in Chapter 8.

11.2.1 Linear feedback shift register (LFSR)

Long LFSR can be used as ‘pseudo-random number generator’. These random numbers are generated based on initial values to LFSR. The sequences of random number can be predicted if the initial value is known. However, if LFSR is quite long (i.e. large number of initial values are possible), then the generated numbers can be considered as random numbers for practical purposes.

LFSR polynomial are written as \(x^3 + x^2 + 1\), which indicates that the feedback is provided through output of ‘xor’ gate whose inputs are connected to positions 3, 2 and 0 of LFSR. Some of the polynomials are listed in Table Table 11.1.
Table 11.1: Feedback polynomials

<table>
<thead>
<tr>
<th>Number of bits</th>
<th>Feedback polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>(x^3 + x^2 + 1)</td>
</tr>
<tr>
<td>4</td>
<td>(x^4 + x^3 + 1)</td>
</tr>
<tr>
<td>5</td>
<td>(x^5 + x^3 + 1)</td>
</tr>
<tr>
<td>6</td>
<td>(x^6 + x^3 + 1)</td>
</tr>
<tr>
<td>7</td>
<td>(x^7 + x^6 + 1)</td>
</tr>
<tr>
<td>9</td>
<td>(x^9 + x^5 + 1)</td>
</tr>
<tr>
<td>10</td>
<td>(x^{10} + x^7 + 1)</td>
</tr>
<tr>
<td>11</td>
<td>(x^{11} + x^9 + 1)</td>
</tr>
<tr>
<td>15</td>
<td>(x^{15} + x^{14} + 1)</td>
</tr>
<tr>
<td>17</td>
<td>(x^{17} + x^{14} + 1)</td>
</tr>
<tr>
<td>18</td>
<td>(x^{18} + x^{11} + 1)</td>
</tr>
</tbody>
</table>

Random numbers are generated using LFSR in Listing 11.1. The code implements the design for 3 bit LFSR, which can be modified for LFSR with higher number of bits as shown below.

**Explanation Listing 11.1**

The listing is currently set according to 3 bit LFSR i.e. \(N = 3\) in Line 16. \(q\) is the output of LFSR, which is random in nature. Lines 29-32 sets the initial value for LFSR to 1 during reset operations.

Note that, LFSR can not have ‘0’ as initial values. Feedback polynomial is implemented at Line 41. Line 52 shifts the last \(N\) bits (i.e. \(N\) to 1) to the right by 1 bit and the \(\$N\$\{th\}$ bit is feed with ‘feedback_value’ and stored in ‘r_next’ signal. In next clock cycle, value of r_next is assigned to r_reg through Line 34. Lastly, the value r_reg is available to output port from Line 53.

Simulation results are shown in Fig. 11.1. Here, we can see that total 7 different numbers are generated by LFSR, which can be seen between two cursors in the figure. Further, q values are represented in ‘hexadecimal format’ which are same as r_reg values in ‘binary format’.

Note that, in Fig. 11.1, the generated sequence contains ‘8, C, 6, B, 5, 2 and 1’; and if we initialize the system with any of these values, outputs will contain same set of numbers again. But, if we initialize the system with ‘3’ (which is not the set), then the generate sequences will be entirely different.

Fig. 11.1: Random number generation with \(N = 3\)

To modify the feedback polynomial, first insert the correct number of bits (i.e. \(N\)) in Line 16. Next, modify the feedback_value at line 41, according to new value of ‘\(N\)’.

Note that maximum-length for a polynomial is defined as \(2^N-1\), but not all the polynomials generate maximum length; e.g. \(N = 5\) generates total 28 sequences (not 31) before repetition as shown in Fig. 11.2.

**Note:** Distributions using LFSR:

- Even number polynomial should be used for generating the ‘Uniformly distributed numbers’, as it does not miss any number from the sequences.
- Then, the uniformly distributed numbers can be used to generate the ‘Gaussian distribution’ using ‘center limit theorem’.
11.2. Random number generator

Fig. 11.2: Total sequences are 28 (not 31) for \( N = 5 \)

Listing 11.1: Random number generation with LFSR

```vhdl
-- rand_num_generator.vhd
-- created by: Meher Krishna Patel
-- date: 22-Dec-16
-- Feedback polynomial: \( x^3 + x^2 + 1 \)
-- maximum length: \( 2^3 - 1 = 7 \)
-- if generic value is changed,
-- then choose the correct Feedback polynomial i.e. change 'feedback_value' pattern
library ieee;
use ieee.std_logic_1164.all;

entity rand_num_generator is
  generic (N : integer := 3);
  port(
    clk, reset : in std_logic;
    q : out std_logic_vector(N downto 0) -- output of LFSR i.e. random number
  );
end rand_num_generator;

architecture arch of rand_num_generator is
  signal r_reg, r_next : std_logic_vector(N downto 0);
  signal feedback_value : std_logic; -- based on feedback polynomial
begin
  process(clk, reset)
  begin
    if(reset='1') then
      -- set initial value to '1'.
      r_reg(0) <= '1'; -- 0th bit = 1
      r_reg(N downto 1) <= (others=>'0'); -- other bits are 0
    elsif (clk'event and clk='1') then
      r_reg <= r_next; -- otherwise save the next state
    end if;
  end process;

  -- N = 3
  -- Feedback polynomial: \( x^3 + x^2 + 1 \)
  -- total sequences (maximum): \( 2^3 - 1 = 7 \)
  feedback_value <= r_reg(3) xor r_reg(2) xor r_reg(0);

  -- N = 4
  -- feedback_value <= r_reg(4) xor r_reg(3) xor r_reg(0);

  -- N = 5, maximum length = 28 (not 31)
  -- feedback_value <= r_reg(5) xor r_reg(3) xor r_reg(0);

  -- N = 9
(continues on next page)
```
11.2.2 Visual test

Listing 11.2 can be used to test the Listing 11.1 on the FPGA board. Here, 1 second clock pulse is used to visualize the output patterns. Please read Chapter 8 for better understanding of the listing. Note that, \( N = 3 \) is set in Line 13 according to Listing 11.1.

For displaying outputs on FPGA board, set reset to 1 and then to 0. Then LEDs will blink to display the generated bit patterns by LFSR; which are shown in Fig. 11.1.

```
library ieee;
use ieee.std_logic_1164.all;

entity rand_num_generator_visualTest is
  generic (N :integer := 3);
  port(
    CLOCK_50, reset : in std_logic;
    LEDR : out std_logic_vector (N downto 0)
  );
end rand_num_generator_visualTest;

architecture arch of rand_num_generator_visualTest is
  signal clk_Pulse1s : std_logic;
begin
  -- clock 1 s
  clock_1s: entity work.clockTick
    generic map (M=>50000000, N=>26)
    port map (clk=>CLOCK_50, reset=>reset,
      clkPulse=>clk_Pulse1s);
  -- rand_num_generator testing with 1 sec clock pulse
  rand_num_generator_1s: entity work.rand_num_generator
    port map (clk=>clk_Pulse1s, reset=>reset,
      q =>LEDR);
end arch;
```

11.3 Shift register

Shift register are the registers which are used to shift the stored bit in one or both directions. In this section, shift register is implemented which can be used for shifting data in both direction. Further it can be used as parallel to serial converter or serial to parallel converter. VHDL files required for this example are listed below,
11.3. Shift register

- shift_register.vhd
- shift_register_visualTest.vhd
- clockTick.vhd
- modMCounter.vhd

Note that, 'clockTick.vhd' and 'modMCounter.vhd' are discussed in Chapter 8.

11.3.1 Bidirectional shift register

Listing 11.3 implements the bidirectional shift register which is explained below,

Explanation Listing 11.3

In the listing, the ‘ctrl’ port is used for ‘shifting’, ‘loading’ and ‘reading’ data operation. Lines 32-39 clear the shift register during reset operation, otherwise go to the next state.

Lines 41-53 performs various operations based on ‘ctrl’ values. Note that, to perform right shift (Line 47), data is continuously provided from last port i.e. (data(N-1)); whereas for left shift (Line 49) data is provided from first port i.e. (data(0)).

Next, ctrl="00" is provided for reading the data. It can be used for serial to parallel conversion i.e. when all the bits are shifted and register is full; then set ctrl to "00" and read the data, after that set ctrl to "01" or "10" for getting next set of bits.

Similarly, for parallel to serial converter, first load the data using ctrl="11"; and then perform the shift operation until all the bits are read and then again load the data. Note that, in this case, last bit propagates (i.e. data(N-1) for right shift or data(0) for left shift) during shifting; which is actually designed for serial to parallel converter. But this will affect the working of parallel to serial converter, as we will set ctrl to "11", when all the data is shifted, therefore all the register which were filled by values from last port, will be overwritten by the new parallel data.

Lastly, data is available on the output port ‘q_reg’ from Line 55. For, parallel to serial converter, use only one pin of ‘q_reg’ i.e. q_reg(0) for right shift or q(N-1) for left shift; whereas for serial to parallel conversion, complete ‘q_reg’ should be read.

Fig. 11.3 shows the shifting operation performed by the listing. Here first data (i.e. 00110000) is loaded with ctrl="11". Then shifted to right after first cursor and later to the left i.e. after second cursor.

![Fig. 11.3: Right and left shifting operations](image)

Listing 11.3: Bidirectional shift register

```vhdl
-- shift_register.vhd

-- created by : Meher Krishna Patel
-- date : 22-Dec-16

-- Functionality:
-- load data and shift it data to left and right
-- parallel to serial conversion (i.e. first load, then shift)
-- serial to parallel conversion (i.e. first shift, then read)
```

(continues on next page)
11.3.2 Visual test

Listing 11.4 can be used to test the Listing 11.3 on the FPGA board. Here, 1 second clock pulse is used to visualize the output patterns. Here, outputs (i.e. q_reg) are displayed on LEDR; whereas ‘shifting-control (i.e. ctrl)’ and data-load (i.e. data) operations are performed using SW[16:15] and SW[7:0] respectively. Here, we can see the shifting of LEDR pattern towards right or left based on SW[16:15] combination. Please read Chapter 8 for better understanding of the listing.
11.3 Shift register

Listing 11.4: Visual test : bidirectional shift register

```vhdl
-- shift_register_visualTest.vhd

-- created by : Meher Krishna Patel
-- date : 22-Dec-16

-- SW[16:15] : used for control

library ieee;
use ieee.std_logic_1164.all;

entity shift_register_visualTest is
    generic (N :integer :=8);
    port(
        CLOCK_50, reset : in std_logic;
        SW : in std_logic_vector(16 downto 0);
        LEDR : out std_logic_vector (N-1 downto 0)
    );
end shift_register_visualTest;

architecture arch of shift_register_visualTest is
    signal clk_Pulse1s : std_logic;
    begin
        -- clock 1 s
        clock_1s: entity work.clockTick
        generic map (M=>50000000, N=>26)
        port map (clk=>CLOCK_50, reset=>reset,
                 clkPulse=>clk_Pulse1s);

        -- shift_register testing with 1 sec clock pulse
        shift_register_1s: entity work.shift_register
        generic map (N=>N)
        port map (clk=>clk_Pulse1s, reset=>reset,
                 data => SW(N-1 downto 0), ctrl => (SW(16 downto 15)),
                 q_reg=>LEDR);
end arch;
```

11.3.3 Parallel to serial converter

If data is loaded first (i.e. ctrl = “11”), and later shift operation is performed (i.e., ctrl = “01” or “10”); then Listing 11.3 will work as ‘parallel to serial converter’. In Listing 11.5 shows the parallel to serial converter which is tested in Listing 11.8.

Listing 11.5: Parallel to serial converter

```vhdl
-- parallel_to_serial.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity parallel_to_serial is
    generic (N : integer := 8);
    port (
        clk, reset : in std_logic;
        data_in : in std_logic_vector(N-1 downto 0); -- parallel data
        data_out : out std_logic; -- serial data
        empty_tick : out std_logic -- 1 = empty, to control other devices
    );
end entity parallel_to_serial;
```
architecture arch of parallel_to_serial is
  type operation_type is (idle, convert, done);
  signal state_reg, state_next : operation_type;
  signal data_reg, data_next : std_logic_vector(N-1 downto 0) := (others => '0');
  signal count_reg, count_next : unsigned(N-1 downto 0);
begin
  -- current register values
  process(clk, reset)
  begin
    if reset = '1' then
      state_reg <= idle;
      data_reg <= (others => '0');
      count_reg <= (others => '0');
    elsif (clk'event and clk = '1') then
      state_reg <= state_next;
      data_reg <= data_next;
      count_reg <= count_next;
    end if;
  end process;

  -- next value in the register
  -- note that, it is poor style of coding as output data calculation
  -- and next values in register are calculated simultaneously. These
  -- should be done in different process statements.
  process(state_reg, data_in, count_reg, data_reg)
  begin
    empty_tick <= '0';
    state_next <= state_reg;
    data_next <= data_reg;
    count_next <= count_reg;
    case state_reg is
      when idle =>
        state_next <= convert;
        data_next <= data_in; -- load the parallel data data
        empty_tick <= '1';
        count_next <= count_reg + 1;
      when convert =>
        count_next <= count_reg + 1;
        if count_reg = N then -- note that N is used here (not N-1, see reason below)
          -- serial_to_parallel.vhd needs one clock cycle to transfer the converted
          -- data (i.e. parallel data) to next device, therefore it can not update the
          -- current value immediately after the conversion. Therefore, count_reg = N,
          -- is used in above line (instead of N-1), so that one extra bit will be added
          -- in the end and will be discarded by serial_to_parallel.vhd as it will not
          -- read the last value.
          -- Also, data_next is not defined here, as last bit is not, therefore, value of
          -- data_next at count 'N' will be same as at 'N-1'.
          state_next <= done;
        else
          -- shift the data to right and append zero in the beginning
          data_next <= '0' & data_reg(N-1 downto 1);
        end if;
      when done =>
        count_next <= (others => '0');
        state_next <= idle;
    end case;
  end process;
75 -- send bit to output port
76  data_out <= data_reg(0);
77 end arch;

11.3.4 Serial to parallel converter

If shifting is performed first (i.e., ctrl = "01" or "10"), and later data is read (i.e., ctrl = "00"), then Listing 11.3 will work as ‘serial to parallel converter’. In Listing 11.6 shows the serial to parallel converter which is tested in Listing 11.8.

Listing 11.6: Serial to parallel converter

```vhdl
-- serial_to_parallel.vhd
library ieee;
use ieee.std_logic_1164.all;

entity serial_to_parallel is
  generic (        
    N : natural := 4
  );
  port (           
    clk, reset : in std_logic;
    in_tick : in std_logic; -- input tick to control the conversion from other device
    din : in std_logic; -- use it as tick, to load data immidiately e.g. from FIFO
    load : out std_logic; -- use it as tick, if data is obtained after one-clock cycle e.g. from generators
    dout : out std_logic_vector(N-1 downto 0)   
  );
end entity serial_to_parallel;

architecture arch of serial_to_parallel is
  type state is (idle, store0, store1);
  signal state_reg, state_next : state;
  signal y_reg, y_next : std_logic_vector(N-1 downto 0);
  signal i_reg, i_next:natural range 0 to N;
begin
  -- update registers
  process(clk, reset)
  begin
    if(reset = '1') then
      i_reg <= 0;
      y_reg <= (others=>'0');
      elsif rising_edge(clk) then
        i_reg <= i_next;
        y_reg <= y_next;
      end if;
  end process;
  -- update states
  process(clk, reset)
  begin
    if reset='1' then
      state_reg <= idle;
    elsif rising_edge(clk) then
```
state_reg <= state_next;
end if;
end process;

-- output data calculation
process(clk, din, y_next, i_next, i_reg, y_reg, state_reg, state_next)
begin
  y_next <= y_reg;
  case state_reg is
  when idle =>
    i_next <= 0;
    load <= '1';
    done <= '0';
    -- this loop is used to load the data immediately in the registers
    if in_tick='1' and din='0' then -- i.e. if first value is zero
      state_next <= store0; -- then go to store0
    elsif in_tick='1' and din='1' then -- i.e. if first value is one
      state_next <= store1; -- then go to store1
    else
      state_next <= idle;
    end if;
    -- input data is 0.
  when store0 =>
    i_next <= i_reg + 1;
    y_next(i_reg) <= '0';
    load <= '0';
    done <= '0';
    if i_reg = N-1 then
      state_next <= idle;
      done <= '1';
    elsif din='1' then
      state_next <= store1;
    else
      state_next <= store0;
    end if;
    -- input data is 1.
  when store1 =>
    i_next <= i_reg + 1;
    y_next(i_reg) <= '1';
    load <= '0';
    done <= '0';
    if i_reg = N-1 then
      state_next <= idle;
      done <= '1';
    elsif din='0' then
      state_next <= store0;
    else
      state_next <= store1;
    end if;
  end case;
end process;

-- send data to output
dout <= y_next when i_reg = N-1;
end architecture arch;
11.3.5 Top level connection between serial and parallel converters

In Listing 11.7, the parallel-counter-data is converted into serial data using Listing 11.5. Then received serial data is converted back to parallel data by Listing 11.6. The simulation results are shown in Fig. 11.4.

Listing 11.7: Connection between serial and parallel converters

```vhdl
-- parallel_and_serial_top_v1.vhd
library ieee;
use ieee.std_logic_1164.all;
library work;

entity parallel_and_serial_top_v1 is
    generic (M : integer := 11; -- count upto M
              N : integer := 4); -- N bits required to count to M
    port
    (reset : in std_logic;
     clk : in std_logic;
     data_out : out std_logic_vector(3 downto 0);
     data_in : out std_logic_vector(3 downto 0));
end parallel_and_serial_top_v1;

architecture arch of parallel_and_serial_top_v1 is

    signal count : std_logic_vector(3 downto 0);
    signal dout : std_logic;
    signal e_tick : std_logic;
    signal not_e_tick : std_logic;

begin

-- register is not empty i.e. read data on this tick
not_e_tick <= not(e_tick);
-- generated count on data_out, whereas received count on data_out
data_in <= count;

-- parallel to serial conversion
unit_p_to_s : entity work.parallel_to_serial
    generic map(N => N)
    port map(clk => clk,
             reset => reset,
             data_in => count,
             data_out => dout,
             empty_tick => e_tick);

-- serial to parallel conversion
unit_s_to_p : entity work.serial_to_parallel
    generic map(N => N)
    port map(clk => clk,
             reset => reset,
             in_tick => not_e_tick,
             din => dout,
             dout => data_out);

-- modMCounter to generate data (i.e. count) for transmission
unit_counter : entity work.modMCounter
    generic map(M => M,
                N => N)

end arch;
```

(continues on next page)
11.3.6 Visual test for serial and parallel converters

In Listing 11.8, the reduced clock-rate is applied to Listing 11.7, so that output can be seen on LEDs. Here, generated counter outputs are displayed on LEDG, whereas the received outputs (i.e., after conversion) are displayed on LEDR.

Listing 11.8: Visual test for serial and parallel converters

```vhdl
-- parallel_and_serial_top_visual_v1.vhd
library ieee;
use ieee.std_logic_1164.all;
library work;

entity parallel_and_serial_top_visual_v1 is
  port (
    reset : in std_logic;
    CLOCK_50 : in std_logic;  
    -- generated count displayed on LEDG
    LEDG : out std_logic_vector(3 downto 0);
    -- received count displayed on LEDR
    LEDR : out std_logic_vector(3 downto 0)
  );
end parallel_and_serial_top_visual_v1;

architecture arch of parallel_and_serial_top_visual_v1 is
  signal clk : std_logic;
begin
  -- parallel and serial conversion test
  unit_p_and_s : entity work.parallel_and_serial_top_v1
  generic map(M => 7,
              N => 4)
  port map(clk => clk,
            reset => reset,
            data_in => LEDR,
            data_out => LEDG);
end arch;
```

Fig. 11.4: Simulation results for Listing 11.7
11.4 Random access memory (RAM)

RAM is memory cells which are used to store or retrieve the data. Further, FPGA chips have separate RAM modules which can be used to design memories of different sizes and types, as shown in this section. VHDL files required for this example are listed below,

- single_port_RAM.vhd
- single_port_RAM_visualTest.vhd
- dual_port_RAM.vhd
- dual_port_RAM_visualTest.vhd

11.4.1 Single port RAM

Single port RAM has one input port (i.e. address line) which is used for both storing and retrieving the data, as shown in Fig. 11.5. Here ‘addr[1:0]’ port is used for both ‘read’ and ‘write’ operations. Listing 11.9 is used to generate this design.

![Fig. 11.5: RTL view : Single port RAM (Listing 11.9)](image)

**Explanation Listing 11.9**

In the listing port ‘addr’ (Line 31) is 2 bit wide as ‘addr_width’ is set to 2 (Line 24). Therefore, total ‘4 elements’ (i.e. $2^2$) can be stored in the RAM. Further, ‘din’ port (Line 32) is 3 bit wide as ‘data_width’ is set to 3 (Line 25); which means the data should be 3 bit wide. **In summary, current RAM-designs can store ‘4 elements’ in it and each elements should be 3-bit wide.**

Write enable (we) port should be high and low for storing and retrieving the data respectively. ‘din’ port is used to write the data in the memory; whereas ‘dout’ port is used for reading the data from the memory. In Lines 43-48, the write operation is performed on rising edge of the clock; whereas read operation is performed at Line 53.

Note that, ‘ram_type’ (Line 38) is created using integer array (as ‘addr_width’ is integer at Line 2); whereas ‘addr’ port is of ‘std_logic_vector’ type (Line 31). Therefore, type conversion is required during write and read operations at Lines 46 and 53 respectively. Further, the type ‘std_logic_vector’ can not be converted directly to ‘integer’, therefore it is converted to ‘unsigned’ type before converting to ‘integer’.

Lastly, Fig. 11.6 shows the simulation results for the design. Here, ‘we’ is set to 1 after first cursor and the data is written at three different addresses (not 4). Next, ‘we’ is set to 0 after second cursor and
read operations are performed for all addresses. Since, no values is stored for address ‘10’, therefore
dout is displayed as ‘UUU’ for this address as shown after third cursor.

Fig. 11.6: Simulation results : Single port RAM (Listing 11.9)

Listing 11.9: Single port RAM

```vhdl
-- single_port_RAM.vhd

-- created by : Meher Krishna Patel
-- date : 26-Dec-16

-- Functionality:
-- store and retrieve data from single port RAM

-- ports:
-- we : write enable
-- addr : input port for getting address
-- din : input data to be stored in RAM
-- dout : output data read from RAM
-- addr_width : total number of elements to store (put exact number)
-- data_width : number of bits in each elements
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity single_port_RAM is
  generic (
    addr_width : integer := 2;
    data_width : integer := 3
  );

  port(
    clk : in std_logic;
    we : in std_logic;
    addr : in std_logic_vector(addr_width-1 downto 0);
    din : in std_logic_vector(data_width-1 downto 0);
    dout : out std_logic_vector(data_width-1 downto 0)
  );
end single_port_RAM;

architecture arch of single_port_RAM is
  type ram_type is array (2**addr_width-1 downto 0) of std_logic_vector (data_width-1 downto 0);
signal ram_single_port : ram_type;
begin
  process(clk)
  ...
11.4. Random access memory (RAM)

(continued from previous page)

begin
  if (clk'event and clk='1') then
    if (we='1') then
      -- write data to address 'addr'
      -- convert 'addr' type to integer from std_logic_vector
      ram_single_port(to_integer(unsigned(addr))) <= din;
    end if;
  end if;
end process;

-- read data from address 'addr'
-- convert 'addr' type to integer from std_logic_vector
dout<=ram_single_port(to_integer(unsigned(addr)));
end arch;

11.4.2 Visual test : single port RAM

Listing 11.10 can be use to test the Listing 11.9 on FPGA board. Different combination of switches can be used to store and retrieve the data from RAM. These data will be displayed on LEDs during read operations.

Listing 11.10: Visual test : single port RAM

```
-- single_port_RAM_visualTest.vhd

-- created by : Meher Krishna Patel
-- date : 26-Dec-16

-- Functionality:
-- store and retrieve data from single port RAM

-- ports:
-- Write Enable (we) : SW[16]
-- Address (addr) : SW[15-14]
-- din : SW[2:0]
-- dout : LEDR

use ieee.numeric_std.all;library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity single_port_RAM_visualTest is
  generic(
    ADDR_WIDTH : integer := 2;
    DATA_WIDTH : integer := 3
  );
  port(
    CLOCK_50: in std_logic;
    SW : in std_logic_vector(16 downto 0);
    LEDR : out std_logic_vector(DATA_WIDTH-1 downto 0)
  );
end single_port_RAM_visualTest;

architecture arch of single_port_RAM_visualTest is
begin
  single_port_RAM_test: entity work.single_port_RAM
    port map (clk=>CLOCK_50, we=>SW(16),
      addr => SW(15 downto 14),
      din => SW(2 downto 0),
      dout =>LEDR);
end arch;
```
11.4.3 Dual port RAM

In single port RAM, the same ‘addr’ port is used for read and write operations; whereas in dual port RAM dedicated address lines are provided for read and write operations i.e. ‘addr_rd’ and ‘addr_wr’ respectively, as shown in Fig. 11.7. Also, the listing can be further modified to allow read and write operation through both the ports.

Fig. 11.7: Dual port RAM

Listing 11.11 is used to implement Fig. 11.7, which is same as Listing 11.9 with three changes. First, two address ports are used at Line 32 (i.e. ‘addr_rd’ and ‘addr_wr’) instead of one. Next, ‘addr_wr’ is used to write the data at Line 47; whereas ‘addr_rd’ data is used to retrieve the data at Line 54. Hence, read and write operation can be performed simultaneously using these two address lines.

Fig. 11.8 shows the simulation results for dual port RAM. Here, on the first cursor, ‘011’ is written at address ‘01’. On next cursor, this value is read along with writing operation as location ‘10’. Lastly, last two cursor shows that if read and write operation is performed simultaneously on one address e.g. ‘01’, then new data will be available at ‘dout’ port after one clock cycle.

Fig. 11.8: Simulation results : Dual port RAM (Listing 11.11)

Listing 11.11: Dual port RAM

```
-- dual_port_RAM.vhd
-- created by : Meher Krishna Patel
-- date : 26-Dec-16

-- Functionality:
-- store and retrieve data from single port RAM

-- ports:
-- we : write enable
-- addr_wr : address for writing
-- addr_rd : address for reading
-- din : input data to be stored in RAM
-- data : output data read from RAM
-- addr_width : total number of elements to store (put exact number)
-- addr_bits : bits requires to store elements specified by addr_width
-- data_width : number of bits in each elements

library ieee;
```
11.4. Random access memory (RAM)

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity dual_port_RAM is
  generic (    
    addr_width : integer := 2;
    data_width : integer := 3  
  );

  port(    
    clk: in std_logic;
    we : in std_logic;
    addr_wr, addr_rd : in std_logic_vector(addr_width-1 downto 0);
    din : in std_logic_vector(data_width-1 downto 0);
    dout : out std_logic_vector(data_width-1 downto 0)  
  );
end dual_port_RAM;

architecture arch of dual_port_RAM is
  type ram_type is array (2**addr_width-1 downto 0) of std_logic_vector (data_width-1 downto 0);
  signal ram_dual_port : ram_type;
begin
  process(clk)
  begin
    if (clk'event and clk='1') then
      if (we='1') then -- write data to address 'addr_wr'
        -- convert 'addr_wr' type to integer from std_logic_vector
        ram_dual_port(to_integer(unsigned(addr_wr))) <= din;
      end if;
    end if;
  end process;

  -- get address for reading data from 'addr_rd'
  dout<=ram_dual_port(to_integer(unsigned(addr_rd)));
end arch;

11.4.4 Visual test : dual port RAM

Listing 11.12 can be use to test the Listing 11.11 on FPGA board. Different combination of switches can be used to store and retrieve the data from RAM. These data will be displayed on LEDs during read operations.

Listing 11.12: Visual test : dual port RAM

-- dual_port_RAM_visualTest.vhd
-- created by : Meher Krishna Patel
-- date : 26-Dec-16
-- Functionality:
-- store and retrieve data from dual port RAM
-- ports:
-- Write Enable (we) : SW[16]
-- Address (addr_wr) : SW[15-14]
-- Address (addr_rd) : SW[13-12]
-- din : SW[2:0]
-- dout : LEDR
Chapter 11. Design examples

(continued from previous page)

```vhdl
use ieee.numeric_std.all;library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity dual_port_RAM_visualTest is
  generic (
    ADDR_WIDTH : integer := 2;
    DATA_WIDTH : integer := 3
  );

  port(
    CLOCK_50: in std_logic;
    SW : in std_logic_vector(16 downto 0);
    LEDR : out std_logic_vector(DATA_WIDTH-1 downto 0)
  );
end dual_port_RAM_visualTest;

architecture arch of dual_port_RAM_visualTest is
begin
  dual_port_RAM_test: entity work.dual_port_RAM
  port map (clk=>CLOCK_50, we=>SW(16),
            addr_wr => SW(15 downto 14),
            addr_rd => SW(13 downto 12),
            din => SW(2 downto 0),
            dout =>LEDR);
end arch;
```

language = Vhdl, label = {} |{.vhd}

11.5 Read only memory (ROM)

ROMs are the devices which are used to store information permanently. In this section, ROM is implemented on FPGA to store the display-pattern for seven-segment device, which is explained in Section 8.5. VHDL files required for this example are listed below,

- ROM_sevenSegment.vhd
- ROM_sevenSegment_visualTest.vhd

11.5.1 ROM implementation using RAM (block ROM)

Listing 11.13 implements the ROM using a block of RAM, which stores the seven-segment display pattern in it. Fig. 11.9 shows the RTL view of the listing, where ROM is implemented using synchronous RAM (denoted by ‘SYNC RAM’ in the figure).

Note that, the ‘write enable (WE)’ port is not provided in the implementation, therefore ‘WE’ is set to ‘0’ permanently (see Fig. 11.9); hence, new data can not be written in the RAM. And data stored during initialization of RAM will be stored permanently. Therefore, the RAM can be considered as ROM. This implementation of ROM is also referred as ‘block ROM’.

![Fig. 11.9: RTL view: block ROM (Listing 11.13)](image_url)
11.5. Read only memory (ROM)

Data from the ROM is accessed through address line; therefore two ports are provided in the design i.e. `addr` and `data` (Lines 27-28). ‘addr_width’ (Line 22) is the number of addresses, which corresponds to total number elements to be stored. Further, ‘addr_bits’ (Line 23) is also defined, which is the minimum number of bits required to represent the total number of addresses i.e. ‘addr_width’. Lastly, ‘data_width’ is declared at Line 24, which defines the number of bits in each element.

Line 34 defines the ROM type (named as `rom_type`) i.e. ROM has 16 elements (i.e. `addr_width`) and each element contains 7 bits (i.e. `data_width`). Next in Line 36, a ROM is created (named as `sevenSegment_ROM`) of type ‘rom_type’; and patterns for seven-segment display are inserted in Lines 37-52 (see Section 8.5 for details). These values are stored sequentially in ROM i.e. patterns 0, A and F are stored at location 0, 10 and 15 respectively.

Note that, ROM is created using ‘integer’ data type (see Line 22 and 34); whereas address line ‘addr’ has type ‘std_logic_vector’. Therefore, to read the memory location, the ‘addr’ value must be changed to ‘integer’ which is done at Line 55. Since ‘std_logic_vector’ can not be converted directly to integer, therefore it is changed to ‘unsigned’ first, and then converted to ‘integer’.

In summary, input ‘addr’ is the ‘binary’ address, which is converted into ‘integer’ format. Then ROM element is accessed based on this ‘integer’ address, which is finally provided at the output port i.e. ‘data’. Fig. 11.10 shows the retrieved ‘data’ values from the ROM for different ‘addr’ values e.g. last waveform presents that ‘addr’ is ‘1111’ and the corresponding ‘data’ value is ‘0111000’.

---

**Listing 11.13**

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ROM_sevenSegment is
  generic(
    addr_width : integer := 16; -- store 16 elements
    addr_bits  : integer := 4; -- required bits to store 16 elements
    data_width : integer := 7 -- each element has 7-bits
  );
```

---

**Fig. 11.10: Retrieving data from ROM**

---
Chapter 11. Design examples

(continued from previous page)

```vhdl
port(
    addr : in std_logic_vector(addr_bits-1 downto 0);
    data : out std_logic_vector(data_width-1 downto 0)
);
end ROM_sevenSegment;
architecture arch of ROM_sevenSegment is

    type rom_type is array (0 to addr_width-1) of std_logic_vector(data_width-1 downto 0);

    signal sevenSegment_ROM : rom_type := (
        "1000000", -- 0, active low i.e. 0:display & 1:no display
        "1111001", -- 1
        "0100100", -- 2
        "0110000", -- 3
        "0011001", -- 4
        "0010010", -- 5
        "0000010", -- 6
        "1111000", -- 7
        "0000000", -- 8
        "0010000", -- 9
        "0010000", -- a
        "0000011", -- b
        "1000110", -- c
        "0100001", -- d
        "0000110", -- e
        "0001110" -- f
    );

    begin
        data <= sevenSegment_ROM(to_integer(unsigned(addr)));
    end arch;
```

11.5.2 ROM implementation logic cells (distributed ROM)

Note that, in Line 36 of Listing 11.13, the ‘signal’ keywords is used for ‘sevenSegment_ROM’, which stores the constant values in Lines 37-52. If the ‘signal’ keyword is replaced by ‘constant’, then the design will be implemented using logical cells i.e. with ‘Mux’ as presented in Fig. 11.11 (instead of block RAM). This design is referred as ‘distributed ROM’.

11.5.3 Distributed ROM vs Block ROM

Note that, Fig. 11.11 illustrate that large number of logical cells are required for distributed ROM implementation; hence it may result in lack of resources for implementing other logical designs.

Further, block RAM is the memory module which is embedded in the FPGA device. Therefore, ROM implementation using RAM does not use the regular logic cells; hence block ROM is the preferred way to implement the ROM as discussed in Section 11.5.1.

11.5.4 Visual test

Listing 11.14 is provided the input ‘addr’ through switches ‘SW’ (Line 20) and then output is from Listing 11.13 is received to signal ‘data’ (Lines 31-32); which is finally displayed on seven segment display devices (Line 34) and LEDs (Line 35). Here, we can see the switch value on the seven-segment display devices along with the ROM content on LEDs; e.g. if SW value is ‘011’ then ‘3’ will be displayed on seven-segment display and ‘0000110’ will be displayed on LEDs.
11.5. Read only memory (ROM)

Fig. 11.11: Partial RTL view: distributed ROM (modified Listing 11.13)
Chapter 11. Design examples

Listing 11.14: Display ROM data on seven segment display and LEDs

```vhdl
-- ROM_sevenSegment_visualTest.vhd
-- created by : Meher Krishna Patel
-- date : 25-Dec-16

-- Functionality:
-- retrieve data from ROM and display on seven-segment device and LEDs

-- ports:
-- SW : address in binary format
-- HEX0 : display data on seven segment device
-- LEDR : display data on LEDs

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ROM_sevenSegment_visualTest is
port(
     SW : in std_logic_vector(3 downto 0);
     HEX0 : out std_logic_vector(6 downto 0);
     LEDR : out std_logic_vector(6 downto 0)
);
end ROM_sevenSegment_visualTest;

architecture arch of ROM_sevenSegment_visualTest is
-- signal to store received data, so that it can be displayed on
-- two devices i.e. seven segment display and LEDs
signal data : std_logic_vector (6 downto 0);
begin
    seven_segment_ROM: entity work.ROM_sevenSegment
    port map (addr=>SW, data=>data);
    HEX0 <= data; -- display on seven segment devices
    LEDR <= data; -- display on LEDs
end arch;
```

11.5.5 Defining ROM contents in file

We can define the content of the ROM in the separate file and then read this file using VHDL code. Please note following items about this style,

- The code will become device specific because Altera devices support the `.mif` files whereas Xilinx devices support the `.CGF` files, which have different formats for storing the ROM contents.
- Design can not be simulated directly using Modelsim.

Since, we are using ‘Quartus software’ in this tutorial, therefore `.mif` files are discussed in this section. ROM data is defined in ‘seven_seg_data.mif’ file as shown in Listing 11.15. In `.mif` file, the comments are written between two ‘% %’ signs (both single line e.g. Line 1 and multiline e.g. Lines 8-12). Further, we need to define certain parameters i.e. data and address types (see comments for details). Lastly, at Line 18, we set the values at all the addresses as ‘0’ and then values are assigned at each address. This can be useful, when we want to store data at fewer locations.

Next, Listing 11.16 is same as the Listing 11.13 except Lines 35-41 where ROM data is read from the `.mif file`, instead of defining in the same file. Please read the comments for understand these lines.

Finally, Listing 11.17 is exactly same as Listing 11.14 except Listing 11.16 is instantiated at Line 31.
11.5. Read only memory (ROM)

Listing 11.15: ROM data stored in `.mif` file

```plaintext
% seven_seg_data.mif %
% ROM data for seven segment display %

% data width and total data %
width=7;  % number of bits in each data %
depth=16;  % total number of data (i.e. total address) %

% format of data and address stored in this file
uns : unsigned, dec : decimal, hex : hexadecimal
bin : binary, oct : octal

address_radix=uns;  % address is unsigned-type %
data_radix=bin;  % data is binary-type %

% ROM data %
content begin
[0..15] : 0000000;  % optional : assign 0 to all address %
  0 : 1000000;  % format => signed : binary %
    1 : 11110001;
    2 : 0100100;
    3 : 0110000;
    4 : 0011001;
    5 : 0010010;
    6 : 0000010;
    7 : 1111000;
    8 : 0000000;
    9 : 0010000;
   10 : 0001000;
   11 : 0000011;
   12 : 1000110;
   13 : 0100001;
   14 : 0000110;
   15 : 0001110;

end;
```

Listing 11.16: VHDL code to read `.mif` file

```vhdl
--ROM_sevenSegment_mif.vhd
--created by : Meher Krishna Patel
--date : 25-Dec-16

-- Functionality:
--seven-segment display format for Hexadecimal values (i.e. 0-F) are stored in ROM

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ROM_sevenSegment_mif is
(continues on next page)
```
Chapter 11. Design examples

(continued from previous page)

21
generic(
22    addr_width : integer := 16; -- store 16 elements
23    addr_bits : integer := 4;  -- required bits to store 16 elements
24    data_width : integer := 7 -- each element has 7-bits
25);  
26
generic(    
27    addr : in std_logic_vector(addr_bits-1 downto 0);
28    data : out std_logic_vector(data_width-1 downto 0) )
29); 
30  end ROM_sevenSegment_mif; 
31
32  architecture arch of ROM_sevenSegment_mif is 
33
34  type rom_type is array (0 to addr_width-1) of std_logic_vector(data_width-1 downto 0); 
35  signal sevenSegment_ROM : rom_type; 
36
37  -- note that 'ram_init_file' is not the user-defined-name (it is attribute name) 
38  attribute ram_init_file : string; 
39  -- "seven_seg_data.mif" is the relative address with respect to project directory 
40  -- suppose "*.mif" file is saved in folder "ROM", then use "ROM/seven_seg_data.mif" 
41  attribute ram_init_file of sevenSegment_ROM : signal is "seven_seg_data.mif"; 
42  begin 
43  data <= sevenSegment_ROM(to_integer(unsigned(addr))); 
44  end arch; 

Listing 11.17: Top level design for Listing 11.16

1
-- ROM.sevenSegment_mif_visualTest.vhd

2
-- created by : Meher Krishna Patel 
3
-- date : 25-Dec-16 
4

5
-- Functionality: 
6  -- retrieve data from ROM and display on seven-segment device and LEDs

7  -- ports: 
8  -- SW : address in binary format
9  -- HEX0 : display data on seven segment device
10  -- LEDR : display data on LEDs

11
library ieee; 
12  use ieee.std_logic_1164.all;  
13  use ieee.numeric_std.all; 
14
15  entity ROM_sevenSegment_mif_visualTest is 
16  port( 
17    SW : in std_logic_vector(3 downto 0); 
18    HEX0 : out std_logic_vector(6 downto 0); 
19    LEDR : out std_logic_vector(6 downto 0) 
20  );
21
22  end ROM_sevenSegment_mif_visualTest; 
23
24
25  architecture arch of ROM_sevenSegment_mif_visualTest is 
26  -- signal to store received data, so that it can be displayed on 
27  -- two devices i.e. seven segment display and LEDs 
28  signal data : std_logic_vector (6 downto 0); 
29  begin 
30    seven_segment_ROM: entity work.ROM_sevenSegment_mif 
31    port map (addr=>SW, data=>data); 
32
33  HEX0 <= data; -- display on seven segment devices

(continues on next page)
11.6 LCD interface

In this section, a package is defined in Listing 11.18 which converts the binary data into seven-segment-display-format and LCD-data-format.

11.6.1 Example 1

In Listing 11.19, this package is used to display the value provided by the switches on the seven-segment device and on the LCD. Further, the counting-pattern is also displayed on the LCD screen. The ASCII-values are shown in Fig. 11.12 which are used for displaying values on LCD screen. Note that, a clock divider is implemented in the design at Lines 85-100. Please see the comments for more details.

```
0 00000000 0 00 NUL  32 0100000 040 20 SP  64 0100000 100 40 @  98 0110000 140 60 ` 
1 00000001 01 01 SOH  33 0100001 041 21 !  65 0100001 101 41 A  97 0110001 141 61 a               
2 00000010 02 02 STX  34 0100010 042 22 "  66 0100010 102 42 B  98 0110010 142 62 b               
3 00000011 03 03 ETX  35 0100011 043 23 #  67 0100011 103 43 C  99 0110011 143 63 c               
4 00000100 04 04 ETX  36 0100100 044 24 $  68 0100100 104 44 D 100 0110100 144 64 d               
5 00000101 05 05 ENQ  37 0100101 045 25 %  69 0100101 105 45 E 101 0110101 145 65 e               
6 00000110 06 06 ACK  38 0100110 046 26 -  70 0100110 106 46 F 102 0111010 146 66 f               
7 00000111 07 07 BEL  39 0100111 047 27 '  71 0100111 107 47 G 103 0111011 147 67 g               
8 00001000 10 08 BS  40 0101000 050 28 (  72 0101000 110 48 H 104 0111000 150 68 h               
9 00001001 11 09 HT  41 0101001 051 29 )  73 0101001 111 49 I 105 0111001 151 69 i               
10 00001010 12 10 LF  42 0101010 052 2A  `  74 0101010 112 4A J 106 0111010 152 6A j               
11 00001011 13 11 VT  43 0101011 053 2B +  75 0101011 113 4B K 107 0111011 153 6B k               
12 00001100 14 12 FF  44 0101100 054 2C ,  76 0101100 114 4C L 108 0111100 154 6C l               
13 00001101 15 13 CR  45 0101101 055 2D -  77 0101101 115 4D M 109 0111101 155 6D m               
14 00001110 16 14 SO  46 0101110 056 2E .  78 0101110 116 4E N 110 0111110 156 6E n               
15 00001111 17 15 SI  47 0101111 057 2F /  79 0101111 117 4F O 111 0111111 157 6F o               
16 00010000 20 10 DLE  48 0110000 060 30 0  80 0110000 120 50 P 112 0110100 160 70 p               
17 00010001 21 11 DC1  49 0110001 061 31 1  81 0110001 121 51 Q 113 0110101 161 71 q               
18 00010010 22 12 DC2  50 0110010 062 32 2  82 0110010 122 52 R 114 0110110 162 72 r               
19 00010011 23 13 DC3  51 0110011 063 33 3  83 0110011 123 53 S 115 0110111 163 73 s               
20 00010100 24 14 DCA  52 0110100 064 34 4  84 0110100 124 54 T 116 0111000 164 74 t               
21 00010101 25 15 NAK  53 0110101 065 35 5  85 0110101 125 55 U 117 0111001 165 75 u               
22 00010110 26 16 SYN  54 0110110 066 36 6  86 0110110 126 56 V 118 0111010 166 76 v               
23 00010111 27 17 ETB  55 0110111 067 37 7  87 0110111 127 57 W 119 0111100 167 77 w               
24 00011000 30 18 CAN  56 0111000 070 38 8  88 0111000 130 58 X 120 0111100 170 78 x               
25 00011001 31 19 EM  57 0111001 071 39 9  89 0111001 131 59 Y 121 0111101 171 79 y               
26 00011010 32 1A SUB  58 0111010 072 3A :  90 0111010 132 5A Z 122 0111101 172 7a z               
27 00011011 33 1B ESC  59 0111011 073 3B  91 0111011 133 5B  \ 123 0111111 173 7b \               
28 00011100 34 1C FS  60 0111100 074 3C {  92 0111100 134 5C  \ 124 0111110 174 7c \               
29 00011101 35 1D GS  61 0111101 075 3D }  93 0111101 135 5D  } 125 0111111 175 7d }               
30 00011110 36 1E RS  62 0111110 076 3E ^  94 0111110 136 5E  ^ 126 0111111 176 7e ^               
31 00011111 37 1F US  63 0111111 077 3F _  95 0111111 137 5F _ 127 0111111 177 7f _               
```

Fig. 11.12: Chart : ASCII values

Listing 11.18: Package for converting binary-format into seven-segment and ASCII format

```
-- LCD_SSD_display_pkg.vhd
-- LCD and Seven-segment-display

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package LCD_SSD_display_pkg is

-- display on LEDs

end arch;
```

(continues on next page)
Chapter 11. Design examples

(continued from previous page)

```vhdl
-- binary to seven-segment format
function binary_to_ssd (  
    signal switch : unsigned  
)  
return unsigned;

-- binary to LCD format
function binary_to_lcd (  
    signal switch : unsigned  
)  
return unsigned;
end LCD_SSD_display_pkg;

package body LCD_SSD_display_pkg is
  -- begin function "binary_to_ssd"
  function binary_to_ssd (  
    -- list all input here  
    signal switch : unsigned(3 downto 0)  
  )  
  -- only one value can be return
  return unsigned is variable sevenSegment : unsigned(6 downto 0);
begin
  case switch is
    when "0000" => sevenSegment := "1000000"; -- 0,
    when "0001" => sevenSegment := "1111001"; -- 1
    when "0010" => sevenSegment := "0100100"; -- 2
    when "0011" => sevenSegment := "0110000"; -- 3
    when "0100" => sevenSegment := "0011001"; -- 4
    when "0101" => sevenSegment := "0010010"; -- 5
    when "0110" => sevenSegment := "0000010"; -- 6
    when "0111" => sevenSegment := "1111000"; -- 7
    when "1000" => sevenSegment := "0000000"; -- 8
    when "1001" => sevenSegment := "0010000"; -- 9
    when "1010" => sevenSegment := "0001000"; -- a
    when "1011" => sevenSegment := "0000110"; -- b
    when "1100" => sevenSegment := "1000110"; -- c
    when "1101" => sevenSegment := "0100010"; -- d
    when "1110" => sevenSegment := "0000110"; -- e
    when others => sevenSegment := "0001110"; -- f
  end case;
  return sevenSegment;
end binary_to_ssd; -- end function "binary_to_ssd"

-- begin function "binary_to_lcd"
function binary_to_lcd (  
    -- list all input here  
    signal switch : unsigned(3 downto 0)  
)  
-- only one value can be return
return unsigned is variable lcdDisplay : unsigned(7 downto 0);
begin
  case switch is
    when "0000" => lcdDisplay := "00110000"; -- 0, active low i.e. 0:display & 1:no display
    when "0001" => lcdDisplay := "00110001"; -- 1
    when "0010" => lcdDisplay := "00110010"; -- 2
    when "0011" => lcdDisplay := "00110011"; -- 3
    when "0100" => lcdDisplay := "01000100"; -- 4
    when "0101" => lcdDisplay := "01000101"; -- 5
    when "0110" => lcdDisplay := "01001100"; -- 6
    when "0111" => lcdDisplay := "01001101"; -- 7
    when "1000" => lcdDisplay := "00000110"; -- 8
    when "1001" => lcdDisplay := "00000111"; -- 9
    when "1010" => lcdDisplay := "00001000"; -- a
    when "1011" => lcdDisplay := "00001001"; -- b
    when "1100" => lcdDisplay := "00001100"; -- c
    when "1101" => lcdDisplay := "00001101"; -- d
    when "1110" => lcdDisplay := "00010000"; -- e
    when "1111" => lcdDisplay := "00010001"; -- f
  end case;
end binary_to_lcd; -- end function "binary_to_lcd"
```
(continues on next page)
when "0110" => lcdDisplay := "00110110"; -- 6
when "0111" => lcdDisplay := "00110111"; -- 7
when "1000" => lcdDisplay := "00111000"; -- 8
when "1001" => lcdDisplay := "00111001"; -- 9
when "1010" => lcdDisplay := "01000001"; -- a
when "1011" => lcdDisplay := "01000010"; -- b
when "1100" => lcdDisplay := "01000011"; -- c
when "1101" => lcdDisplay := "01000100"; -- d
when "1110" => lcdDisplay := "01000101"; -- e
when others => lcdDisplay := "01000110"; -- f
end case;
return lcdDisplay;
end binary_to_lcd; -- end function "binary_to_lcd"
end LCD_SSD_display_pkg ;

Listing 11.19: LCD and seven-segment display

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.LCD_SSD_display_pkg.all;
entity LCD_SSD_display is
generic (clk_dvd : natural := 500000;
max_count : natural := 16);
port(
CLOCK_50, reset : in std_logic;
SW : in unsigned(3 downto 0);
LEDG : out unsigned(3 downto 0);
HEX0 : out unsigned(6 downto 0);
LCD_RS, LCD_RW : out std_logic;
LCD_EN : out std_logic;
LCD_ON, LCD_BLON : out std_logic; -- LCD ON, Backlight ON
LCD_DATA : out unsigned(7 downto 0)
);
end entity;
architecture arch of LCD_SSD_display is
-- FSM for lcd display
type stateType is (init1, init2, init3, init4, clearDisplay, displayControl, Line1, Line2, writeData_
-->row1, writeData_row2, returnHome);
signal state_reg, state_next : stateType;
signal input_lcd : unsigned(7 downto 0);
Chapter 11. Design examples

(continued from previous page)

```vhdl
signal count : natural := 0;
signal char_count_reg, char_count_next : natural := 0;
signal inc_count_reg, inc_count_next : integer := 0;

signal lcd_clock : std_logic;

signal data1, data2 : unsigned(3 downto 0);

type character_string is array ( 0 to 31 ) of unsigned( 7 downto 0 );
signal LCD_display_string : character_string;

begin
  -- Data to be displayed:
  LCD_display_string <= ( 44
    -- use spaces (X"20") for blank positions, then update these blank position
    -- with desired values e.g. in row 1 we updated the blank position with switch value
    -- and in row 2, blank space is replaced by counting
    -- Line 1
    -- MEHER S
    X"4D",X"45",X"48",X"45",X"52",X"20",X"20",X"53",  45
    X"57",X"20",X"3D",X"20",X"20",X"20",X"20",X"20",  46
    -- W =
    X"43",X"6F",X"75",X"6E",X"74",X"20",X"3D",X"20",  47
    -- count =
    X"20",X"20",X"20",X"20",X"20",X"20",X"20",X"20"  48
    -- 8 spaces
  );

  -- LED display
  LEDG <= SW;

  -- seven-segment display
  HEX0 <= binary_to_ssd(SW);

  -- lcd display
  LCD_ON <= '1';
  LCD_BLON <= '1';
  LCD_EN <= lcd_clock;
  input_lcd <= binary_to_lcd(SW);

  -- clock divider which is used as "enable" signal for LCD
  -- this is required as for higher clock-rates, we can not see
  -- the data on the LCD.
  process(CLOCK_50, reset)
  begin
    if reset = '1' then
      count <= 0;
      lcd_clock <= '0';
    elsif (rising_edge(CLOCK_50)) then
      count <= count + 1;
      if count = clk_dvd then
        lcd_clock <= not lcd_clock;
        count <= 0;
      end if;
    end if;
  end process;

  process(lcd_clock, reset)
  begin
    (continues on next page)
  end process;
```

(continues on next page)
if reset = '1' then
    state_reg <= init1;
    inc_count_reg <= 0;
    char_count_reg <= 0;
elsif (rising_edge(lcd_clock)) then
    state_reg <= state_next;
    char_count_reg <= char_count_next;
    inc_count_reg <= inc_count_next;
end if;
end process;

process(state_reg, char_count_reg, inc_count_reg, data1, data2, LCD_display_string, input_lcd)
begin
    char_count_next <= char_count_reg;
    inc_count_next <= inc_count_reg;
    case state_reg is
    -- extra initialization states are required for proper reset operation
    when init1 =>
        LCD_RS <= '0';
        LCD_RW <= '0';
        LCD_DATA <= "00111000";
        -- change init2 to clearDisplay if extra-init is not required
        state_next <= init2;
    when init2 =>
        LCD_RS <= '0';
        LCD_RW <= '0';
        LCD_DATA <= "00111000";
        state_next <= init3;
    when init3 =>
        LCD_RS <= '0';
        LCD_RW <= '0';
        LCD_DATA <= "00111000";
        state_next <= init4;
    when init4 =>
        LCD_RS <= '0';
        LCD_RW <= '0';
        LCD_DATA <= "00111000";
        state_next <= clearDisplay;
    -- clear display and turn off cursor
    when clearDisplay =>
        LCD_RS <= '0';
        LCD_RW <= '0';
        LCD_DATA <= "00000001";
        state_next <= displayControl;
    -- turn on Display and turn off cursor
    when displayControl =>
        LCD_RS <= '0';
        LCD_RW <= '0';
        LCD_DATA <= "00001100";
        state_next <= Line1;
    -- Line 1
    -- write mode with auto-increment address and move cursor to the right
    when Line1 =>

(continues on next page)
Chapter 11. Design examples

166
167 LCD_RS <= '0';
168 LCD_RW <= '0';
169 LCD_DATA <= "00001110";
170 state_next <= writeData_row1;

171 -- write data
172 when writeData_row1 =>
173 LCD_RS <= '1';
174 LCD_RW <= '0';
175 state_next <= writeData_row1;
176 char_count_next <= char_count_reg + 1;
177
178 if char_count_reg = 12 then -- replace space at position 13 with number
179 LCD_DATA <= input_lcd;
180 elsif char_count_reg < 15 then
181 LCD_DATA <= LCD_display_string(char_count_reg);
182 else -- char_count_reg = 15 then
183 LCD_DATA <= LCD_display_string(char_count_reg);
184 state_next <= Line2;
185 end if;

186 -- Line 2
187 when Line2 =>
188 LCD_RS <= '0';
189 LCD_RW <= '0';
190 LCD_DATA <= "11000000";
191 state_next <= writeData_row2;

192 -- write data at line 2
193 when writeData_row2 =>
194 LCD_RS <= '1';
195 LCD_RW <= '0';
196 state_next <= writeData_row2;
197 char_count_next <= char_count_reg + 1;
198
199 if char_count_reg = 24 then -- replace space at location 26 with number
200 -- decimal place of inc_count_reg
201 data1 <= unsigned(to_signed((inc_count_reg / 10) mod 10, 4));
202 LCD_DATA <= binary_to_lcd(data1);
203 elsif char_count_reg = 25 then
204 -- unit place of inc_count_reg
205 data2 <= unsigned(to_signed(inc_count_reg mod 10, 4));
206 LCD_DATA <= binary_to_lcd(data2);
207 if inc_count_reg = max_count then -- reset if count reach to maximum
208 inc_count_next <= 0;
209 else
210 inc_count_next <= inc_count_reg + 1;
211 end if;
212 elsif char_count_reg < 31 then
213 LCD_DATA <= LCD_display_string(char_count_reg);
214 else -- char_count_reg = 31 then
215 LCD_DATA <= LCD_display_string(char_count_reg);
216 char_count_next <= 0;
217 state_next <= returnHome;
218 end if;

219 -- Return write address to first character position on line 1
220 when returnHome =>
221 LCD_RS <= '0';
222 LCD_RW <= '0';
223 LCD_DATA <= "10000000";

(continues on next page)
11.6. LCD interface

(continued from previous page)

```vhdl
state_next <= writeData_row1;
end case;
end process;
end architecture;
```

11.6.2 Example 2

In Listing 11.19, we used the the ‘mod’ and ‘division’ operation to display the integer values on the LCD. Note that, the division operation requires lots of hardware and for large integer values it can not be synthesized as well. Therefore it is better to display the number in ‘Hexadecimal’ format as shown in Listing 11.20. Note that, the binary format is not suitable as well, as it will need large number of characters to display the values on LCD.

Further, in Listing 11.20 the message format is defined, but the actual values are obtained from the Listing 11.21. In this way, we can separate the ‘calculation’ and ‘displaying’ designs, which is more manageable than previous case.

Listing 11.20: Message displayed on the LCD
end entity;

architecture arch of LCD_BER_display is
  -- FSM for lcd display
  type stateType is (init1, init2, init3, init4, clearDisplay, displayControl, Line1, Line2, writeData_row1, writeData_row2, returnHome);
  signal state_reg, state_next : stateType;
  signal input_lcd : unsigned(7 downto 0);
  signal count : natural := 0;
  signal char_count_reg, char_count_next : natural := 0;
  -- read the new input values after displaying the current values
  signal error_val_reg, error_val_next : unsigned(11 downto 0);
  signal total_bits_reg, total_bits_next : unsigned(31 downto 0);
  -- clock for LCD
  signal lcd_clock : std_logic;
  -- 16X2 LCD : ASCII-format
  type character_string is array ( 0 to 31 ) of unsigned( 7 downto 0 );
  signal LCD_display_string : character_string;
begin
  -- Data to be displayed :
  LCD_display_string <= ("X"45", "X"72", "X"72", "X"6F", "X"72", "X"73", "X"3D", "X"20", -- Errors=
                         "X"30", "X"78", "X"20", "X"20", "X"20", "X"20", "X"20", "X"20", -- 0x 6-spaces
                         "X"42", "X"69", "X"74", "X"73", "X"3D", "X"20", "X"30", "X"78", -- Bits= 0x
                         "X"20", "X"20", "X"20", "X"20", "X"20", "X"20", "X"20", "X"20", -- 8 spaces
                       );

  -- lcd display settings
  LCD_ON <= '1';
  LCD_BLON <= '1';
  LCD_EN <= lcd_clock;
  -- clock divider which is used as "enable" signal for LCD
  -- this is required as for higher clock-rates, we cannot see
  -- the data on the LCD.
  process(CLK_50, reset)
  begin
    if reset = '1' then
      count <= 0;
      lcd_clock <= '0';
    elsif (rising_edge(CLK_50)) then
      if count = clk_dvd then
        if count = clk_dvd then
          lcd_clock <= not lcd_clock;
          count <= 0;
        end if;
      end if;
    end if;
  end process;
11.6. LCD interface

(continued from previous page)

```vhdl
-- current states for LCD data
process(lcd_clock, reset)
begin
  if reset = '1' then
    state_reg <= init1;
    char_count_reg <= 0;
    total_bits_reg <= (others => '0');
    error_val_reg <= (others => '0');
  elsif (rising_edge(lcd_clock)) then
    state_reg <= state_next;
    char_count_reg <= char_count_next;
    total_bits_reg <= total_bits_next;
    error_val_reg <= error_val_next;
  end if;
end process;

-- write data on LCD and calculate next-states of LCD
process(state_reg, char_count_reg, error_val_reg, LCD_display_string, error_val, total_bits, total_bits_reg, input_lcd)
begin
  char_count_next <= char_count_reg;
  total_bits_next <= total_bits_reg;
  error_val_next <= error_val_reg;
  case state_reg is
    when init1 =>
      LCD_RS <= '0';
      LCD_RW <= '0';
      LCD_DATA <= "00111000";
      -- change init2 to clearDisplay if extra-init is not required
      state_next <= init2;
    when init2 =>
      LCD_RS <= '0';
      LCD_RW <= '0';
      LCD_DATA <= "00111000";
      state_next <= init3;
    when init3 =>
      LCD_RS <= '0';
      LCD_RW <= '0';
      LCD_DATA <= "00111000";
      state_next <= init4;
    when init4 =>
      LCD_RS <= '0';
      LCD_RW <= '0';
      LCD_DATA <= "00111000";
      state_next <= clearDisplay;
    when clearDisplay =>
      LCD_RS <= '0';
      LCD_RW <= '0';
      LCD_DATA <= "00000001";
      state_next <= displayControl;
    when others =>
      state_next <= state_reg;
  end case;
end process;
```

(continues on next page)
when displayControl =>
  LCD_RS <= '0';
  LCD_RW <= '0';
  LCD_DATA <= "00001100'';
  state_next <= Line1;

  -- Line 1
  -- write mode with auto-increment address and move cursor to the right
when Line1 =>
  LCD_RS <= '0';
  LCD_RW <= '0';
  LCD_DATA <= "00000110'';
  state_next <= writeData_row1;

  -- write data
when writeData_row1 =>
  LCD_RS <= '1';
  LCD_RW <= '0';
  state_next <= writeData_row1;
  char_count_next <= char_count_reg + 1;

  if char_count_reg = 10 then -- replace space at position 13 with number
    LCD_DATA <= binary_to_lcd(error_val_reg(11 downto 8));
  elsif char_count_reg = 11 then
    LCD_DATA <= binary_to_lcd(error_val_reg(7 downto 4));
  elsif char_count_reg = 12 then
    LCD_DATA <= binary_to_lcd(error_val_reg(3 downto 0));
  elsif char_count_reg < 15 then
    LCD_DATA <= LCD_display_string(char_count_reg);
  else -- char_count_reg = 15 then
    LCD_DATA <= LCD_display_string(char_count_reg);
    state_next <= Line2;
  end if;

  -- Line 2
when Line2 =>
  LCD_RS <= '0';
  LCD_RW <= '0';
  LCD_DATA <= "11000000'';
  state_next <= writeData_row2;

  -- write data at line 2
when writeData_row2 =>
  LCD_RS <= '1';
  LCD_RW <= '0';
  state_next <= writeData_row2;
  char_count_next <= char_count_reg + 1;

  if char_count_reg = 24 then -- replace space at position 13 with number
    LCD_DATA <= binary_to_lcd(total_bits_reg(31 downto 28));
  elsif char_count_reg = 25 then
    LCD_DATA <= binary_to_lcd(total_bits_reg(27 downto 24));
  elsif char_count_reg = 26 then
    LCD_DATA <= binary_to_lcd(total_bits_reg(23 downto 20));
  elsif char_count_reg = 27 then
    LCD_DATA <= binary_to_lcd(total_bits_reg(19 downto 16));
  elsif char_count_reg = 28 then
    LCD_DATA <= binary_to_lcd(total_bits_reg(15 downto 12));
  elsif char_count_reg = 29 then
    LCD_DATA <= binary_to_lcd(total_bits_reg(11 downto 8));
  end if;

11.6. LCD interface

(continued from previous page)

```vhdl
elsif char_count_reg = 30 then
  LCD_DATA <= binary_to_lcd(total_bits_reg(7 downto 4));
elsif char_count_reg = 31 then  -- reached to end, hence go to "returnHome"
  LCD_DATA <= binary_to_lcd(total_bits_reg(3 downto 0));
  state_next <= returnHome;
  char_count_next <= 0;
else
  LCD_DATA <= LCD_display_string(char_count_reg);
end if;

-- Return write address to first character position on line 1
when returnHome =>
  LCD_RS <= '0';
  LCD_RW <= '0';
  LCD_DATA <= "10000000";
  state_next <= writeData_row1;
  error_val_next <= error_val;
  total_bits_next <= total_bits;
end case;
end process;
end architecture;
```

Listing 11.21: Top level entity which provides the values to Listing 11.20

```vhdl
-- LCD_BER_display_top.vhd
-- test the LCD display
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.LCD_SSD_display_pkg.all;

entity LCD_BER_display_top is
  generic (  
    -- dont go below 5000 as LCD will refresh very fast and we can see the values on LCD
    clk_dvd : natural := 50000;
    max_count : natural := 16  
  );
  port(
    CLOCK_50, reset : in std_logic;
    SW : in unsigned(11 downto 0);
    LCD_RS, LCD_RW : out std_logic;
    LCD_EN : out std_logic;
    LCD_ON, LCD_BLON : out std_logic;  -- LCD ON, Backlight ON
    LCD_DATA : out unsigned(7 downto 0)
  );
end entity;

architecture arch of LCD_BER_display_top is
begin
  unit_LCD_BER : entity work.LCD_BER_display
  port map (  
    CLOCK_50 => CLOCK_50,
    reset => reset,
    -- convert 6 bits to 12 and 32 bits format
    error_val => to_unsigned(to_integer(SW(5 downto 0)),12),
    total_bits =>to_unsigned(to_integer(SW(11 downto 6)),32),
```

(continues on next page)
36
37
38
39
40
41
42
43
44
45
46
end architecture;

11.6.3 Example 3: Error counting

In Listing 11.21, we if we change the position of the switches, then corresponding values are shown in the LCD display. In this section, we will use two switches where one switch will act as transmitted signal and second switch will act as detected signal. If the two values are different, then error-count and transmitted-bit-count will increase; but if both the values are same then only bit-count will increase (and error-count will remain constant as there is not error).

For this, first we need to design a error-count circuit as shown in Listing 11.23. Next, we integrated Listing 11.23 and Listing 11.19 in one listing i.e. Listing 11.24; this is done so that we need to instantiate only one design to count and display the errors. In the other words, the main design should send the transmitted and detected bit patterns to Listing 11.24 and the rest of the job, i.e. error calculation and error display, will be performed by the Listing 11.24, as discussed in next paragraph.

Finally a top level design is created in Listing 11.22, which sends the transmitted and detected bit patterns (using SW) to the Listing 11.24. If both switches are at the same position then only transmitted-bit-count will increase, otherwise both error-count and transmitted-bit-count will increase. And the counting will stop when the maximum number of errors will be reached.

Note:

- Different clock-rates are used for LCD-display and error-calculation, as both the design needs different frequency of operations, as shown in top level design for the error counter (see Lines 58 and 61 of Listing 11.22).
- The reset_n is used in the design which is (KEY(3)) on the DE2 board. This is also important because the SW switches of the DE2 board are not implemented with debouncing circuits, therefore if the error calculation are performed on high clock rate (in actual system, not in the current design), then 200 errors will be detected on the reset operation itself.
- Further, error calculations is skipped for first few clocks as shown in Lines 67-76 of Listing 11.23. During these cycles, ‘0xFFF...’ are displayed on the screen. Please, read the comments for more details.
- Lastly is better to use ‘KEY (i.e. reset_n)’ for reset operations at the top level design only; because by default ‘KEY’ is set to high position, therefore ‘not’ operation is required as shown in Line 28 of Listing 11.22. If we use ‘reset_n’ then it will create lots of confusion, as not operation is used with it for resetting the system.

Listing 11.22: Top level design which sends the transmitted and detected bit pattern

1
2
3
4
5

-- error_lcd_connection_test_v1.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

(continues on next page)
entity error_lcd_connection_test_v1 is
  generic (
    -- total number of errors
    num_of_errors : unsigned(11 downto 0) := to_unsigned(100, 12)
  );
  port(
    CLOCK_50, reset_n : in std_logic;
    SW : in std_logic_vector(1 downto 0); -- 1 tx bit, 0 detected bit
    LCD_RS, LCD_RW : out std_logic;
    LCD_EN : out std_logic;
    LCD_ON, LCD_BLON : out std_logic; -- LCD ON, Backlight ON
    LCD_DATA : out unsigned(7 downto 0)
  );
end entity;

architecture arch of error_lcd_connection_test_v1 is
begin
  reset <= not reset_n;
  unit_error_counter : entity work.error_lcd_connection_v1
    generic map (num_of_errors => num_of_errors)
    port map (
      clk_error => clk_error,
      clk_LCD => clk_LCD,
      reset => reset,
      tx_bit => SW(1),
      detected_bit => SW(0),

      -- LCD control
      LCD_RS => LCD_RS,
      LCD_RW => LCD_RW,
      LCD_EN => LCD_EN,
      LCD_ON => LCD_ON,
      LCD_BLON => LCD_BLON,
      LCD_DATA => LCD_DATA
    );

  -- 1 ms clock
  unit_clockTick : entity work.clockTick
    generic map (
      M => 5000000,
      N => 26
    )
    port map (
      clk => CLOCK_50,
      reset => reset,
      clkPulse => clk_error -- clk_error
    );
  clk_LCD <= CLOCK_50; -- clk_LCD
end arch;

Listing 11.23: Error counter
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity error_counter_v1 is
  generic (
    -- total number of errors
    num_of_errors : unsigned(11 downto 0) := to_unsigned(50, 12);
    -- 6000 for SF = 50; 500 for SF = 100
    skip_clock : unsigned(15 downto 0) := to_unsigned(500, 16)
  );
  port(
    clk, reset : in std_logic;
    tx_bit, detected_bit : in std_logic;
    total_errors : out unsigned(11 downto 0);
    total_bits : out unsigned(31 downto 0)
  );
end entity;

architecture arch of error_counter_v1 is
  signal count_errors,count_errors_next : unsigned(11 downto 0) := (others =>'0');
  signal count_bits,count_bits_next : unsigned(31 downto 0) := (others =>'0');

  -- skip first few clocks, so that reset signal settle down,
  -- it is required for proper BER display on the LCD, otherwise
  -- error will be calculated while reset is settling down
  -- This happens because, we are manually resetting the system.
  -- if require, increase/decrease the value of "count_skip_errors"
  signal count_skip_errors : unsigned(31 downto 0) := to_unsigned(20, 32);
  signal count_skip_reg, count_skip_next : unsigned(32 downto 0);

  type stateType is (setup, start);
  -- done);
  signal state_reg, state_next : stateType;
begin

  process(clk, reset)
  begin
    if reset = '1' then
      count_bits <= (others =>'0');
      count_errors <= (others =>'0');
      count_skip_reg <= (others =>'0');
      state_reg <= setup;
    elsif falling_edge(clk) then
      count_bits <= count_bits_next;
      count_errors <= count_errors_next;
      count_skip_reg <= count_skip_next;
      state_reg <= state_next;
    end if;
  end process;

  process(tx_bit, detected_bit, count_errors, count_bits, state_reg, count_skip_reg, count_skip_errors)
  begin
    count_bits_next <= count_bits;
    count_errors_next <= count_errors;
  (continues on next page)
count_skip_next <= count_skip_reg;
state_next <= state_reg;
case state_reg is
when setup => -- do not count error for first few cycles
  if (count_skip_reg /= count_skip_errors) then
    count_bits_next <= (others=>'1'); -- display 0xFFF
    count_errors_next <= (others=>'1'); -- display 0xFFFFFFFF
    count_skip_next <= count_skip_reg + 1;
  elsif count_skip_reg = count_skip_errors then
    count_bits_next <= (others=>'0');
    count_errors_next <= (others=>'0');
    state_next <= start;
  end if;
when start => -- start counting-errors
  if (count_errors /= num_of_errors) then
    if tx_bit /= detected_bit then
      count_bits_next <= count_bits + 1; -- increment values
      count_errors_next <= count_errors + 1;
    elsif tx_bit = detected_bit then
      count_bits_next <= count_bits + 1;
      count_errors_next <= count_errors;
    end if;
  elsif count_errors = num_of_errors then -- if maximum error reached
    count_bits_next <= count_bits; -- then stop incrementing values
    count_errors_next <= count_errors;
  end if;
end case;
end process;

-- assign values to output ports
total_errors <= count_errors;
total_bits <= count_bits;
end arch;

Listing 11.24: Connect error-counting and LCD-display in one design

-- error_lcd_connection_v1.vhd
-- version : 1
-- Meher Krishna Patel
-- Date : 21-Sep-2017

-- connects the error_counter_v1.vhd and LCD_BER_display.vhd to display
-- BER on the LCD

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity error_lcd_connection_v1 is
  generic (---
    num_of_errors : unsigned(11 downto 0) := to_unsigned(200, 12)
  );
  port(
    clk_error, clk_LCD, reset : in std_logic;
  )
(continues on next page)
11.7 VGA interface

In this section, VGA interface is created for FPGA devices. VHDL files required for this example are listed below,

- sync_VGA.vhd
- sync_VGA_visualTest.vhd
- sync_VGA_visualTest2.vhd
- clockTick.vhd
- modMCounter.vhd

Note that, ‘clockTick.vhd’ and ‘modMCounter.vhd’ are discussed in Chapter 8.
11.7 VGA interface

11.7.1 Synchronization circuit

To display the data generated from FPGA on the VGA screen, we need to send synchronization signal along with some other signals e.g. horizontal synchronization signal, vertical synchronization signals and 25 MHz VGA clock etc. Listing 11.25 generates a synchronization circuit to display the data on the VGA screen.

Listing 11.25: VGA synchronization circuit

```vhdl
-- sync_VGA_visualTest.vhd
-- created by : Meher Krishna Patel
-- date : 24-Dec-16
-- Functionality:
-- synchronize the VGA system
-- ports:
-- vga_clk : 25 MHz clock for VGA operation (generated by sync_VGA.vhd file)
-- video_on : send video_on = '0' while synchronization otherwise '1' to display data
-- hsync and vsync : synchronization signals required for VGA operation
-- pixel_x and pixel_y : 10 bit pixel location
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity sync_VGA is
  port(
    clk, reset: in std_logic;
    hsync, vsync: out std_logic;
    video_on, vga_clk: out std_logic;
    pixel_x, pixel_y: out std_logic_vector (9 downto 0)
  );
end sync_VGA;

architecture arch of sync_VGA is
-- VGA 640-by-480
constant HD: integer:=640; -- horizontal display area
constant VD: integer:=480; -- vertical display area

-- Horizontal and Vertical retraces
constant HR: integer:=100; -- horizontal retrace
constant VR: integer:=10; -- vertical retrace

-- 25 MHz VGA clock
signal vga_tick: std_logic;

-- pixel location
signal h_pixel, h_pixel_next: unsigned(9 downto 0);
signal v_pixel, v_pixel_next: unsigned(9 downto 0);

-- store location of screen-ends for retracing operation
signal h_end, v_end: std_logic;
begin
  -- 25 MHz clock for VGA operations
  clock_25MHz: entity work.clockTick
  generic map (M=>2, N=>2)
  port map (clk=>clk, reset=>reset,
            clkPulse=>vga_tick);

  -- reset pixel location
end arch;
```

(continues on next page)
process (clk, reset)
begin
  if reset='1' then
    v_pixel <= (others=>'0');
    h_pixel <= (others=>'0');
  elsif (clk'event and clk='1') then
    v_pixel <= v_pixel_next;
    h_pixel <= h_pixel_next;
  end if;
end process;

-- video on/off
process(clk, h_pixel, v_pixel)
begin
  if (h_pixel < HD and v_pixel < VD) then
    video_on <= '1';
  else
    video_on <= '0';
  end if;
end process;

-- end points for retrace
h_end <= '1' when h_pixel=(HD+HR-1) else
  '0';
v_end <= '1' when v_pixel=(VD+VR-1) else
  '0';

-- set h_pixel_next to zero when end of horizontal-screen reached
-- otherwise increment on vga_tick
process (vga_tick, h_pixel, h_end)
begin
  if vga_tick='1' then
    if h_end='1' then
      h_pixel_next <= (others=>'0');
    else
      h_pixel_next <= h_pixel + 1;
    end if;
  else
    h_pixel_next <= h_pixel;
  end if;
end process;

-- set v_pixel_next to zero when end of horizontal and vertical screen reached
-- otherwise increment on vga_tick
process (vga_tick, v_pixel, h_end, v_end)
begin
  if vga_tick='1' and h_end='1' then
    if (v_end='1') then
      v_pixel_next <= (others=>'0');
    else
      v_pixel_next <= v_pixel + 1;
    end if;
  else
    v_pixel_next <= v_pixel;
  end if;
end process;

-- horizontal and vertical sync signals
hsync <= '1' when (h_pixel>=(HD))
  and (h_pixel<=(HD+HR-1)) else

11.7. VGA interface

(continued from previous page)

```vhdl
'0';
vsync <=
'1' when (v_pixel<=(VD))
and (v_pixel<=(VD+VR-1)) else
'0';

-- convert unsigned-pixel-locations to std_logic_vector format
pixel_x <= std_logic_vector(h_pixel);
pixel_y <= std_logic_vector(v_pixel);

-- send clock to output port
vga_clk <= vga_tick;
end arch;
```

11.7.2 Visual test : change screen color with switches

Listing 11.26 can be used to test Listing 11.25. The color of the screen will be change according to combination of switches.

Listing 11.26: Change screen color with switches

```vhdl
-- sync_VGA_visualTest.vhd

-- created by : Meher Krishna Patel
-- date : 24-Dec-16

-- Functionality:
-- change the color of screen based on switch combination i.e. 0 to 7

-- ports:
-- VGA_CLK : 25 MHz clock for VGA operation (generated by sync_VGA.vhd file)
-- VGA_BLANK : required for VGA operations and set to 1
-- SW : combination will change the color of screen
-- VGA_HS and VGA_VS : synchronization signals required for VGA operation
-- VGA_R, VGA_G and VGA_B : 10 bit RGB signals for displaying colors on screen

library ieee;
use ieee.std_logic_1164.all;

entity sync_VGA_visualTest is
    port(
        CLOCK_50, reset: in std_logic;
        VGA_CLK, VGA_BLANK : out std_logic;
        SW: in std_logic_vector(2 downto 0);
        VGA_HS, VGA_VS: out std_logic;
        VGA_R, VGA_G, VGA_B: out std_logic_vector(9 downto 0)
    );
end sync_VGA_visualTest;

architecture arch of sync_VGA_visualTest is
    signal rgb_reg: std_logic_vector(2 downto 0);
    signal video_on: std_logic;
begin
    -- set VGA_BLANK to 1
    VGA_BLANK <= '1';

    -- instantiate sync_VGA for synchronization
    sync_VGA_unit: entity work.sync_VGA
        port map(clk=>CLOCK_50, reset=>reset, hsync=>VGA_HS,
```

(continues on next page)
11.7.3 Visual test: display different colors on screen

Listing 11.27 displays four different colors on the screen as shown in Fig. 11.13.

Fig. 11.13: Screen with four different colors (Listing 11.27)

Listing 11.27: Display different colors on screen

```vhdl
-- sync_VGA_visualTest2.vhd
-- created by: Meher Krishna Patel
-- date: 27-Dec-16

-- Functionality:
-- display four squares of different colors on the screen

-- ports:
-- VGA_CLK: 25 MHz clock for VGA operation (generated by sync_VGA.vhd file)
-- VGA_BLANK: required for VGA operations and set to 1
-- VGA_HS and VGA_VS: synchronization signals required for VGA operation
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity sync_VGA_visualTest2 is
  port (  
    CLOCK_50, reset: in std_logic;
    VGA_CLK, VGA_BLANK : out std_logic;
    VGA_HS, VGA_VS: out std_logic;
    VGA_R, VGA_G, VGA_B: out std_logic_vector(9 downto 0)
  );
end sync_VGA_visualTest2;

architecture arch of sync_VGA_visualTest2 is
  signal rgb_reg: std_logic_vector(2 downto 0);
  signal video_on: std_logic;
  signal pixel_x, pixel_y : std_logic_vector (9 downto 0);
  signal pix_x, pix_y : integer;
  begin
    -- set VGA_BLANK to 1
    VGA_BLANK <= '1';
    -- instantiate sync_VGA for synchronization
    sync_VGA_unit: entity work.sync_VGA
      port map(clk=>CLOCK_50, reset=>reset, hsync=>VGA_HS,
        vsync=>VGA_VS, video_on=>video_on,
        vga_clk=>VGA_CLK, pixel_x=>pixel_x, pixel_y=>pixel_y);
    pix_x <= to_integer(unsigned(pixel_x));
    pix_y <= to_integer(unsigned(pixel_y));
    process(CLOCK_50)
    begin
      if (video_on = '1') then
        -- divide VGA screen i.e. 640-by-480 in four equal parts
        -- and display different colors in those parts
        -- Red color
        if (pix_x < 320 and pix_y < 240) then -- 640/2 = 320 and 480/2 = 240
          VGA_R <= (others=>'1'); -- send '1' to all 10 bits of VGA_R
          VGA_G <= (others=>'0');
          VGA_B <= (others=>'0');
        -- Green color
        elsif (pix_x >= 320 and pix_y < 240) then
          VGA_R <= (others=>'0');
          VGA_G <= (others=>'1');
          VGA_B <= (others=>'0');
        -- Blue color
        elsif (pix_x < 320 and pix_y >= 240) then
          VGA_R <= (others=>'0');
          VGA_G <= (others=>'0');
          VGA_B <= (others=>'1');
        -- Yellow color
        else
          VGA_R <= (others=>'1');
          VGA_G <= (others=>'1');
          VGA_B <= (others=>'0');
    end process;
end arch;
end if;
else
  VGA_R <= (others=>'0');
  VGA_G <= (others=>'0');
  VGA_B <= (others=>'0');
end if;
end process;
end arch;
Chapter 12

Simulate and implement SoPC design

12.1 Introduction

In this tutorial, we will learn to design embedded system on FPGA board using Nios-II processor, which is often known as ‘System on Programmable chip (SoPC)’. Before beginning the SoPC design, let’s understand the difference between ‘computer system’, ‘embedded system’ and ‘SoPC system’. Computer systems, e.g. personal computers (PCs), support various end user applications. Selection of these computer systems for few specific applications, e.g. timers settings, displays and buzzers in microwave-ovens or washing machines etc., is not a cost effective choice. The Embedded systems are cost effective solution for such cases, where only few functionalities are required in the system. Since, embedded systems are designed for specific tasks, therefore the designs can be optimized based on the applications. Further, SoPC systems are the programmable embedded systems i.e. we can design the embedded systems using Verilog/VHDL codes and implement them on the FPGA board.

Nios II is a 32-bit embedded-processor architecture designed for Altera-FPGA board. In Fig. 12.1, various steps are shown to design the SoPC system on the FPGA board using Nios-II processor. In this chapter, these steps are discussed and a system is designed which displays the message on the computer and blinks one LED using FPGA board. Further, the outputs of this system is verified using Modelsim. It is recommended to read first two chapters before practicing the codes, as various issues related to the use of Nios software are discussed in these chapters.

Note: Note that, the NIOS-projects can not be run directly on the system, after downloading it from the website; therefore only ‘necessary files are provided on the website. Please see Appendix B to compile and synthesize the provided codes.

Note: Also, if you change the location of the project in your system (after creating it) or change the FPGA-board, then you need to follow the instructions in the Appendix B again.

12.2 Creating Quartus project

First, we need to create the project at desired location, with following steps,

- Path to project directory should not contain any spaces in the names of folder as shown in Fig. 12.2, as path will not be detected by the Nios-II software in later part of the tutorial.

- Choose the correct FPGA device as shown in Fig. 12.3. If you do not have any, then select correct device family and use it for rest of the tutorial. Note that, ‘simulation’ is the most important process of the design, and FPGA board is not required for creating and simulating the embedded design. Hence, if you don’t have FPGA board, you can still use the tutorial. Implementation of the design on the FPGA board requires the loading the ‘.sof’ and ‘.elf’ files on the board, which is not a big deal.
Fig. 12.1: Flow chart for creating the embedded system

Fig. 12.2: Create project
12.3 Create custom peripherals

We can create peripheral devices e.g., adder, divider or clock generator using VHDL. Then these devices can be used in creating the SoPC using Nios-II software as discussed in Section 12.4. For simplicity of the tutorial, only predefined-peripherals are used in the designs, which are available in Nios-II software. Creating and using the custom-peripherals will be discussed in later chapters.
Chapter 12. Simulate and implement SoPC design

12.4 Create and Generate SoPC using Qsys

SoPC can be created using two tools in Quartus software i.e. ‘SOPC builder’ and ‘Qsys’ tools. Since Qsys is the recommended tool by the Altera, therefore we will use this tool for generating the SoPC system. To open the Qsys tool, go to Tools—>Qsys in Quartus software. Next, follow the below steps to create the SoPC system,

- Rename ‘clk_0’ to ‘clk’ by right clicking the name (optional step). Note that, renaming steps are optional; but assigning appropriate names to the components is good practice as we need to identify these components in later tutorial.
- In component library, search for Nios processor as shown in Fig. 12.5 and select the Nios II/e (i.e. economy version) for this tutorial. Note that various errors are displayed in the figure, which will be removed in later part of this section. Next, rename the processor to ‘nios_blinkingLED’ and connect the clock and reset port to ‘clk’ device, by clicking on the circles as shown in Fig. 12.6.

![Fig. 12.5: Add Nios processor](image1)

![Fig. 12.6: Rename Nios processor and connect the ports](image2)

- Next add the ‘onchip memory’ to the system with 20480 (i.e. 20k) bytes as shown in Fig. 12.7 and rename
it to RAM. For different FPGA boards, we may need to reduce the memory size, if the ‘memory-overflow’ error is displayed during ‘generation’ process in Fig. 12.14.

After that, search for ‘JTAG UART’ and add it with default settings. It is used for the communication of the FPGA device to Nios software. More specifically, it is required display the outputs of the ‘printf’ commands on the Nios software.

Next, add a PIO device for blinking the LED. Search for PIO device and add it as ‘1 bit’ PIO device (see Fig. 12.8), because we will use only one blinking LED. Since, the PIO device, i.e. ‘LED’, is available as external port, therefore name of the ‘external_connection’ must be defined as shown in Fig. 12.9.

Then, add all the connections as shown in the Fig. 12.9. Also, add the JTAG slave to the data master of Nios processors i.e. IRQ in Fig. 12.10

Next, double click on the to the Nios processor i.e. ‘nios_blinkingLED’ in the Fig. 12.9; and set RAM as reset and exception vectors as shown in Fig. 12.11.
Chapter 12. Simulate and implement SoPC design

Fig. 12.9: Settings for PIO and other device

Fig. 12.10: Connect IRQ
Then, go to Systems–>Assign Base Address. This will assign the unique addresses for all the devices. With
the help of these address, the devices can be accessed by C/C++ codes in Nios-II software. If everything is
correct till this point, we will get the ‘0 Errors, 0 Warnings’ message as shown in Fig. 12.12.

Finally, to allow the simulation of the designs, go to ‘generation’ tab and set the tab as shown in the Fig.
12.13. Note that, VHDL can not be used as simulation language, as this version of Quartus does not support it.
Since, generated verilog files are used for the simulation settings settings only, therefore
we need not to look at these files. Hence it does not matter, whether these files are created using VHDL or
Verilog.

Save the system with desired name e.g. ‘nios_blinkingLED.qsys’ and click on generate button. If system is
generated successfully, we will get the message as shown in Fig. 12.14

After this process, a ‘nios_blinkingLED.sopcinfo’ file will be generated, which is used by Nios-II software
to create the embedded design. This file contains all the information about the components along with
their base addresses etc. Further, two more folders will be created inside the ‘nios_blinkingLED’ folder
i.e. ‘synthesis’ and ‘testbench’. These folders are generated as we select the ‘synthesis’ and ‘simulation’
options in Fig. 12.13, and contains various information about synthesis (e.g. ‘nios_blinkingLED.qip’ file) and
simulation (e.g. ‘nios_blinkingLED_tb.qsys’ file). The ‘nios_blinkingLED.qip’ file will be used for the
creating the top module for the design i.e. LEDs will be connected to FPGA board using this file; whereas
‘nios_blinkingLED_tb.qsys’ contains the information about simulation waveforms for the testbenches.
Lastly, ‘nios_blinkingLED_tb.spd’ is generated which is used by Nios software to start the simulation
process.

In this way, SoPC system can be generated. Finally, close the Qsys tool. In next section, Nios software will
be used to create the blinking-LED system using ‘nios_blinkingLED.sopcinfo’ file.

12.5 Create Nios system

In previous section, we have created the SoPC system and corresponding .sopcinfo file was generated. In this
section, we will use the .sopcinfo file to create a system with blinking LED functionality.

To open the Nios software from Quartus software, click on Tools–>Nios II software Build Tools for Eclipse. If
you are using it first time, then it will ask for work space location; you can select any location for this.
Chapter 12. Simulate and implement SoPC design

Fig. 12.12: 0 errors and 0 warnings

Fig. 12.13: Simulation settings
12.6 Add and Modify BSP

To use the .sopcinfo file, we need to create a ‘board support package (BSP)’. Further, BSP should be modify for simulation purpose, as discussed in this section.

12.6.1 Add BSP

To add BSP, go to File→New→Nios II Board Support Package. Fill the project name field and select the .sopcinfo file as shown in Fig. 12.15. Rest of the field will be filled automatically.

After clicking on finish, BSP_blinkingLED folder will be created which contains various files e.g. system.h, io.h and drivers etc. These files are generated based on the information in the .sopcfile e.g. in Fig. 12.16, which shows the partial view of system.h file, contains the information about LED along with the base address (note that, this LED was added along with base address in Section 12.4). Note that, address location is defined as ‘LED_BASE’ in the system.h file; and we will use this name in the tutorial instead of using the actual base address. In this way,
we need not to modify the ‘C/C++’ codes, when the base address of the LED is changed during Qsys modification process.

```
#define ALT_MODULE_CLASS led altera_avalon_pic
#define LED_BASE 0x11000
#define LED_BIT_CLEARING_EDGE_REGISTER 0
#define LED_BIT_MODIFYING_OUTPUT_REGISTER 0
```

Fig. 12.16: system.h file

### 12.6.2 Modify BSP (required for using onchip memory)

**Note:** Some modifications are required in the BSP file for using onchip memory (due to it’s smaller size). Also, due to smaller size of onchip memory, C++ codes cannot be used for NIOS design. Further, these settings are not required for the cases where external RAM is used for memory e.g. SDRAM (discussed in Chapter 14); and after adding external RAM, we can use C++ code for NIOS design as well.

To modify the BSP, right click on ‘BSP_blinkingLED’ and then click on Nios II–>BSP Editor. Then select the ‘enable_reduce_device_drivers’ and ‘enable_small_c_library’ as shown in Fig. 12.17; then click on the ‘generate’ button to apply the modifications.

To modify the BSP, right click on ‘BSP_blinkingLED’ and then click on Nios II–>BSP Editor. Then select the ‘enable_reduce_device_drivers’ and ‘enable_small_c_library’ as shown in Fig. 12.17; then click on the ‘generate’ button to apply the modifications.

![Fig. 12.17: Modify BSP](image)

### 12.7 Create application using C/C++

In previous section, various information are generated by BSP based to .sopcinfo file. In this section, we will write the ‘C’ application to blink the LED. Further, we can write the code using C++ as well.

To create the application, go to File–>New–>Nios II Application. Fill the application name i.e. ‘Application_blinkingLED’ and select the BSP location as shown in Fig. 12.18.

To create the C application, right click on the ‘Application_blinkingLED’ and go to New–>Source File. Write the name of source file as ‘main.c’ and select the ‘Default C source template’.

Next, write the code in ‘main.c’ file as shown in Listing 12.1. After writing the code, right click on the ‘Application_blinkingLED’ and click on ‘build project’.

**Explanation Listing 12.1**

The ‘io.h’ file at Line 2, contains various functions for operations on input/output ports e.g. IOWR(base, offset, data) at Line 15 is define in it. Next, IOWR uses three parameter i.e. base, offset and data which are set to ‘LED_BASE’, ‘0’ and ‘led_pattern’ respectively at Line 15. ‘LED_BASE’ contains the based address, which is defined in ‘system.h’ file at Line 4 (see Fig. 12.16 as well). Lastly, ‘alt_u8’ is the custom data-type (i.e. unsigned 8-bit integer), which is used at line 7 and defined in ‘alt_types.h’ at Line 3. It is required because, for predefined C data types e.g. int and long etc. the
size of the data type is defined by the compiler; but using ‘alt_types.h’ we can define the exact size for the data.

Since ‘alt_u8’ is 8 bit wide, therefore led_pattern is initialized as ‘0$times$01’ (i.e. 0000-0001) at Line 7. Then this value is inverted at Line 14 (for blinking LED). Lastly, this pattern is written on the LED address at Line 15. Second parameter, i.e. offset is set to 0, because we want to write the pattern on LED_BASE (not to other location with respect of LED_BASE). Also, dummy loop is used at Line 19 is commented, which will be required for visual verification of the blinking LED in Section 12.11.

Listing 12.1: Blinking LED with C application

```c
#include "alt_types.h" // contains address of LED_BASE

int main(){
    alt_u8 led_pattern = 0x01; // on the LED

    printf("Blinking LED\n");
    while(1){
        led_pattern = ~led_pattern; // not the led_pattern
        IOWR(LED_BASE, 0, led_pattern); // write the led_pattern on LED

        // uncomment 'for loop' in below line for visual verification of blinking LED
        // dummy for loop to add delay in blinking, so that we can see the blinking.
        //for (i=0; i<itr; i++){}
    }
}
```
12.8 Simulate the Nios application

In previous section, we built the Nios application. To simulate this system, right click on ‘Application_blinkingLED’ and go to Run as->NOIS II Modelsim. Then select the location of Modelsim software i.e. ‘win32aloem’ or ‘win64aloem’ folder as shown in Fig. 12.19; this can be found inside the folders where Altera software is installed.

![Fig. 12.19: Search for Modelsim location](image1)

Next, click on the advance button in Fig. 12.19 and a new window will pop-up. Fill the project location as shown in Fig. 12.20 and close the pop-up window. This location contains the ‘.spd’ file, which is required for simulation.

![Fig. 12.20: Search for Project Location](image2)

After above step, ‘Qsys testbench simulation package descriptor file name’ column will be filled with ‘.spd’ file automatically as shown in Fig. 12.21. If not, do it manually by searching the file, which is available in the project folder. If everything is correct, then apply and run buttons will be activated. Click on apply and then run button to start the simulation.

Modelsim window will pop-up after clicking the run button, which will display all the external ports as shown in Fig. 12.22. Right click on the external signals and click on ‘add to wave’. Next go to transcript section at the bottom of the modelsim and type ‘run 3 ms’. The simulator will be run for 3 ms and we will get outputs as shown in Fig. 12.23. In the figure, there are 3 signals i.e. clock, reset and LED, where last waveform i.e. ‘nios_blinkingled_inst_led_external_connection_export’ is the output waveform, which will be displayed on the LEDs.
12.8. Simulate the Nios application

Fig. 12.21: Run simulation

Fig. 12.22: Modelsim window

Fig. 12.23: Blinking LED simulation waveform
12.9 Adding the top level VHDL design

In last section, we designed the Nios system for blinking LED and tested it in modelsim. Now, next task is to implement this design on FPGA board. For this we, need to add a top level design, which connects all the input and output ports on the FPGA. Here, we have only three ports i.e. clock, reset and LEDs.

First add the design generated by the Qsys system to the project. For this, we need to add the `.qip` file into the project. This file can be found in ‘synthesis folder as shown in Fig. 12.24.

![Fig. 12.24: Adding .qip file (available in synthesis folder)](image)

Next, we need to add a new VHDL file for port mapping. For this, create a new VHDL file with name ‘blinkingLED_VisualTest.vhd’ as shown in Listing 12.2. Then set it as ‘top level entity’ by right clicking it. Note that, for port-mapping, the component is declared in Lines 14-20 because Qsys generates the files in ‘Verilog’ format. Since, we are using Verilog files in VHDL design therefore ‘component’ declaration is required for port mapping (for more details, see the tutorial on Verilog designs in VHDL). Further, this declaration and port map code can be copied from Qsys software as well, which is available on ‘HDL example’ tab as shown in Fig. 12.25.

![Fig. 12.25: Copy port mapping code from Qsys](image)

Top level design can be implemented using ‘.bdf’ file as well. For this go to File–>new–>Block diagram/Schematic file; and double click on the file. Next add the QSys design from the popped up window, as shown in Fig. 12.26. Then right click on the design and select ‘Generate Pins for Symbol Ports’. Next, assign the pin-numbers by giving proper name to input and output ports, as shown in Fig. 12.27.

![Fig. 12.26: Adding Qsys design to the project](image)
Now, select any one of these designs (i.e. .vhd or .bdf file) as top level design.

Fig. 12.26: Create top level design using Block schematic method

Fig. 12.27: Pin assignments for block schematic method

12.10 Load the Quartus design (i.e. .sof/.pof file)

Before loading the design on FPGA board, import the pin assignments from Assignments->Import assignments and select the ‘DE2_PinAssg_PythonDSP.csv’ file, which is available on the website along with the codes. Lastly, compile the design and load the generated ‘.sof’ or ‘.pof’ file on the FPGA chip.

Listing 12.2: Top level design

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity blinkingLED_VisualTest is
  port(
    CLOCK_50, reset : in std_logic;
    LEDG : out std_logic_vector(1 downto 0)
  );
end blinkingLED_VisualTest;

architecture arch of blinkingLED_VisualTest is
  component nios_blinkingLED is
    port (  
      clk_clk : in std_logic; -- clk
      reset_reset_n : in std_logic; -- reset_n  
      led_external_connection_export : out std_logic -- export
    );
end component;
```

(continues on next page)
end component nios_blinkingLED;
begin
  u0 : component nios_blinkingLED
  port map (   clk_clk => CLOCK_50, -- clk.clk
               reset_reset_n => reset, -- reset.reset_n
               led_external_connection_export => LEDG(0) -- led_external_connection.export
             );
end arch;

12.11 Load the Nios design (i.e. ‘.elf’ file)

In previous section, VHDL design is loaded on the FPGA board. This loading creates all the necessary connections
on the FPGA board. Next we need to load the Nios design, i.e. ‘.elf file’, on the board. Since 50 MHz clock is too
fast for visualizing the blinking of the LED, therefore a dummy loop is added in the ‘main.c’ file (see Lines 10 and
19) as shown in Listing 12.3. This dummy loop generates delay, hence LED does not change its state immediately.

Listing 12.3: Blinking LED with C application

```c
//main.c
#include "io.h" // required for IOWR
#include "alt_types.h" // required for alt_u8
#include "system.h" // contains address of LED_BASE

int main(){
  alt_u8 led_pattern = 0x01; // on the LED
  printf("Blinking LED\n");
  int i, itr=250000;

  //uncomment below line for visual verification of blinking LED
  int i, itr=250000;

  printf("Blinking LED\n");
  while(1){
    led_pattern = ~led_pattern; // not the led_pattern
    IOWR(LED_BASE, 0, led_pattern); // write the led_pattern on LED
    //uncomment 'for loop' in below line for visual verification of blinking LED
    // dummy for loop to add delay in blinking, so that we can see the blinking.
    for (i=0; i<itr; i ++){}
  }
}
```

After these changes, right click on the ‘Application_blinkingLED’ folder and click on Run as–>Nios II Hardware.
The may load correctly and we can see the outputs. Otherwise a window will appear as shown in Fig. 12.28. If
neither window appears nor design loaded properly, then go to Run–>Run Configuration–>Nios II Harware and
select a hardware or create a harware by double-clicking on it.

Next, go to ‘Target connection’ tab and click on ‘refresh connection’ and select the two ignore-boxes of ‘System
ID checks’ as shown in Fig. 12.29. Then click on the run button. Keep the reset button high, while loading
the design, otherwise design will not be loaded on the FPGA chip.

Once Nios design is loaded successfully, then ‘Blinking LED’ will be displayed on the ‘Nios II console’. Now if
we change the reset to ‘0’ and then ‘1’ again; then the message will be displayed again as shown in Fig. 12.30. Further, bilking LED can be seen after this process.
12.11. Load the Nios design (i.e. `.elf` file)

Fig. 12.28: Run configuration

Fig. 12.29: Run configuration settings

Fig. 12.30: Final outputs
12.12 Saving NIOS-console’s data to file

In Section 12.11, the message ‘Blinking LED’ is displayed on the NIOS console. Most of the time, we need to store the data in a file for further processing e.g. in Chapter 14, sine waveforms are generated using NIOS, and the resulted data is saved in a file and then plotted using Python; note that this plotting operation can not be performed when data is displayed on the NIOS console.

Please see the video: NIOS II - Save data to the file using JTAG-UART if you have problem with this part of tutorial.

To save the data on a file, first build the project as discussed in the Section 12.7. Next, right click on the ‘Application_BlinkingLED’ and go to ‘NIOS II–>NIOS II Command Shell’. Then, in the command prompt, execute the following two commands,

- `nios2-download -g Application_BlinkingLED.elf` : This will load the NIOS design on the FPGA board.
- `nios2-terminal.exe -q --quiet > blinkOut.txt` : This will save the messages in the file ‘blinkOut.txt’, which will be saved in the ‘Application_BlinkingLED’ folder. We can now, reset the system and ‘Blinking LED’ message will be displayed twice as shown in Fig. 12.30.
- ‘-q --quiet’ options is used to suppress the NIOS-generated message in the file. For more options, we can use ‘-help’ option e.g. `nios2-terminal -help`.
- Lastly, ‘>’ sign erases the contents of existing file before writing in it. To append the current outputs at the end of the existing file, use ‘>>’ option, instead of ‘>’.
- We can run these two commands in one step using ‘&&’ operators i.e.

```
nios2-download -g Application_BlinkingLED.elf && nios2-terminal.exe -q --quiet > ../../python/data/blinkOut.txt.
```

In this case, ‘../../python/data/blinkOut.txt’ command will save the ‘blinkOut.txt’ file by going two folders-back i.e. ‘../..’, then it will save the file in ‘data’ folder which is inside the ‘python’ folder. In this way, we can save the results at any location.

12.13 Conclusion

In this tutorial, we saw various steps to create a embedded Nios processor design. We design the system for blinking the LED and displaying the message on the ‘Nios II console’ window. Also, we learn the simulation methods for Nios-II designs. In next chapter, we will learn about adding components to the ‘Qsys’ file. Also, we will see the process of updating the BSP according to new ‘.sopcinfo’ file.
Chapter 13

Reading data from peripherals

13.1 Introduction

In Chapter 12, the values of the variables in the code i.e. ‘led_pattern’ were sent to PIO (peripheral input/output) device i.e. LED. In this chapter, we will learn to read the values from the PIO. More specifically, the values will be read from the external switches. These values will decide the blinking rate of the LED. Further, we will not create a new ‘.qsys’ file, but modify the previous ‘.qsys’ file to generate the new system. Also, we will see the procedures to modify the existing Nios project i.e. BSP and Application files (instead of creating the new project).

13.2 Modify Qsys file

First create or open a Quartus project as discussed in Chapter 12. Then open the ‘Qsys’ file, which we have created in the previous chapter and modify the file as below,

- **Modify LED port**: Double click on the ‘led’ and change the ‘Width’ column to ‘2’ from ‘1’; because in this chapter, we will use two LEDs which will blink alternatively.
- **Add Switch Port**: Next, add a new PIO device and set it to ‘output’ port with width ‘8’ as shown in Fig. 13.1. Next, rename it to ‘switch’ and modify it’s clock, reset and external connections as shown in Fig. 13.2.

![Fig. 13.1: Add switch of width 8](image)

- Next, assign base address by clicking on System–>Assign base addresses.
- Finally, generate the system, by clicking on the ‘Generate button’ with correct simulation settings, as shown in Fig. 12.14.
Chapter 13. Reading data from peripherals

13.3 Modify top level design in Quartus

Since, LED port is modified and the ‘switch’ port is added to the system, therefore top level module should be modified to assign proper connections to the FPGA chip. Top level design for this purpose is shown in Listing 13.1.

Listing 13.1: Modify top level design in Quartus

```vhdl
-- blinkingLED_VisualTest.vhd

library ieee;
use ieee.std_logic_1164.all;

entity blinkingLED_VisualTest is
  port(
    CLOCK_50, reset : in std_logic;
    SW : in std_logic_vector(7 downto 0);
    LEDG : out std_logic_vector(1 downto 0)
  );
end blinkingLED_VisualTest;

architecture arch of blinkingLED_VisualTest is
begin
  u0 : component nios_blinkingLED
    port map (   
      clk_clk : in std_logic; -- clk
      reset_reset_n : in std_logic; -- reset_n
      -- switch;
      switch_external_connection_export : in std_logic_vector(7 downto 0) := (others => 'X');
      led_external_connection_export : out std_logic_vector(1 downto 0) -- LED
    );
end component nios_blinkingLED;

13.4 Modify Nios project

Since ‘Qsys’ system is modified, therefore corresponding ‘.sopcinfo’ file is modified also. Now, we need to update the system according to the new ‘.sopcinfo’ file.
13.4.1 Adding Nios project to workspace

If ‘workspace is changed’ or ‘BSP/Application files are removed’ from the current workspace, then we need to add these files again in the project, as discussed below.

**Note:** If the location of the project is changed, then we need to created the NIOS project again, as shown in Appendix B.

- First go to Files–>Import–>'Import Nios II software...' as shown in Fig. 13.3; and select the ‘application’ from the ‘software’ folder (see Fig. 13.4) inside the main directory. Finally, give it the correct name i.e. ‘Application_blinkingLED’ as shown in Fig. 13.5.

![Fig. 13.3: Import Application and BSP files](image)

- Similarly, add the BSP folder in the current workspace.
- Now, open the ‘system.h’ file in the BSP. Here, we can see that LED data width is still ‘1’ as shown in Fig. 13.6. Also, we can not find the ‘switch’ entries in the system.h file.
- To update the BSP, right click on the BSP folder and go to Nios–>Generate BSP. It will update the BSP files.
- Sometimes, proper addresses are not assigned during generation in ‘Qsys’ system as shown in Fig. 13.7. Here, SWITCH_BASE is set to ‘0$times$0’. To remove this problem, assign base addresses again and regenerate the system as discussed in Section 13.2.

13.5 Add ‘C’ file for reading switches

Next, modify the ‘main.c’ file as shown in Listing 13.2. Finally, build the system by pressing ‘ctrl+B’.

**Explanation Listing 13.2**

‘IORD(base, offset)’ command is used the read the values form I/O ports as shown at Line 19 of the listing. Line 19 reads the values of the switches; which is multiply by variable ‘itr’ (see Line 22), to set the delay value base on switch patterns. Also, ‘iter ‘value is set to ‘1000’ at Line 11, so that multiplication can produce sufficient delay to observe the blinking LEDs.
Chapter 13. Reading data from peripherals

Fig. 13.4: Application and BSP files are in software folder

Fig. 13.5: Add application Name i.e. Application_blinkingLED

```cpp
#define ALT_MODULE_CLASS_led altera_aavalon_pio
#define LED_BASE 0x11000
#define LED_BIT_CLEARING_EDGE_REGISTER 0
#define LED_BIT_MODIFYING_OUTPUT_REGISTER 0
#define LED_CAPTURE 0
#define LED_DATA_WIDTH 1
```

Fig. 13.6: LED data width is not updated to ‘2’
13.6 Simulation and Implementation

If build is successful, then we can simulate and implement the system as discussed in Chapter 12. Fig. 13.8 and Fig. 13.9 illustrate the simulation results for switch patterns ‘0000-0000’ and ‘0000-0001’ respectively. Note that, blinking patterns are shown by ‘01’ and ‘10’, which indicates that LEDs will blink alternatively. Also, blinking-time-periods for patterns ‘0000-0000’ and ‘0000-0001’ are ‘4420 ns’ and ‘1057091 ns’ respectively (see square boxes in the figures), which show that blinking periods depend on the switch patterns.

To implement the design, compile the top level design in Quartus and load the ‘.sof’ file on the FPGA chip. Then load the ‘.elf’ file from Nios software to FPGA chip. After this, we can see the alternatively blinking LEDs on the board.

13.7 Conclusion

In this chapter, values from the PIO device i.e. switch patterns are read using IORD command. These switch patterns are used to decide the blinking rate of the LEDs. Further, we discussed the procedure to modify the
existing Qsys, Quartus and Nios to extend the functionalities of the system.
Chapter 14

UART, SDRAM and Python

14.1 Introduction

In this chapter, UART communication is discussed for NIOS design. Values of $\sin(x)$ is generated using NIOS and the data is received by computer using UART cable. Since, onchip memory is smaller for storing these values, therefore external memory i.e. SDRAM is used. Further, the received data is stored in a file using ‘Tera Term’ software; finally live-plotting of data is performed using Python.

In this chapter, we will learn following topics,

- UART interface,
- Receiving the data on computer using UART communication,
- SDRAM interface,
- Saving data generated by NIOS design to a file using ‘Tera Term’,
- Updating an existing QSys design and corresponding VHDL and NIOS design,
- Live-plotting of data using Python.

14.2 UART interface

First, create a empty project with name ‘UartComm’ (see Section 1.2). Next, open the QSys from Tools->Qsys. Add ‘Nios Processor’, ‘On-chip RAM (with 20k total-memory-size), ‘JTAG UART’ and ‘UART (RS-232 Serial Port)’ (all with default settings). Note that, Baud rate for UART is set to ‘115200’ (see Fig. 14.1), which will be used while getting the data on computer. Lastly, connect these items as shown in Fig. 14.2; save it as ‘Uart_Qsys.qsys’ and finally generate the QSys system and close the Qsys. Please see Section 12.4, if you have problem in generating the QSys system.

Now, add the file ‘Uart_Qsys.qip’ to the VHDL project. Next, create a new ‘Block diagram (.bdf) file and import the Qsys design to it and assign correct pin numbers to it, as shown in Fig. 14.3. Save it as ‘Uart_top.bdf’ and set it as ‘top level entity’). Lastly, import the pin assignment file and compile the design. Finally, load the design on FPGA board.

14.3 NIOS design

In Chapter 12, we created the ‘BSP’ and ‘application’ file separately for NIOS design. In this chapter, we will use the template provided with NIOS to create the design. For this, open the NIOS software and go to ‘Files–>New–>NIOS II Application and BSP from Template’. Next, Select the ‘UART_Qsys.sopcinfo’ file and ‘Hello World’ template and provide the desired name to project e.g. UART_comm_app, as shown in Fig. , and click ‘next’. In this window, enter the desired name for BSP file in the ‘Project name’ column e.g. ‘UART_comm_bsp’; and click on Finish.
Fig. 14.1: UART settings

Fig. 14.2: Qsys connections
14.3. NIOS design

Fig. 14.3: Top level entity ‘Uart_top.bdf’

Fig. 14.4: Create NIOS project from template
14.4 Communication through UART

To receive the data on the computer, we need some software like Putty or Tera Term. In this tutorial, we are using Tera Term software, which can be downloaded freely. Also, we need to change the UART communication settings; so that, we can get messages through UART interface (instead of JTAG-UART) as shown next.

Right click on ‘UART_comm_bsp’ and go to ‘NIOS II–>BSP editor’; and select UART_115200 for various communication as shown in Fig. 14.5; and finally click on generate and then click on exit. Now, all the ‘printf’ statements will be sent to the computer via UART port (instead of JTAG-uart). We can change it to JTAG-UART again, by changing UART_115200 to JTAG-UART again. Note that, when we modify the BSP using BSP-editor, then we need to generate the system again.

Now, open the Tera Term and select the ‘Serial’ as shown in Fig. 14.6. Then go to ‘Setup–>Serial Port . . .’ and select the correct baud rate i.e. 115200 and click OK, as shown in Fig. 14.7.

Finally, right click on ‘UART_comm_app’ in NIOS and go to ‘Run As–>3 NIOS 2 Hardware’. Now, we can see the output on the Tera Term terminal, as shown in Fig. 14.8.
14.4. Communication through UART

Fig. 14.7: Select correct baud rate

Fig. 14.8: ‘Hello from NIOS II!’ on Tera Term
14.5 SDRAM Interface

Our next aim is to generate the Sine waves using NIOS and then plot the waveforms using python. If we write the C-code in current design, then our system will report the memory issue as onchip memory is too small; therefore we need to use external memory. In this section, first, we will update the Qsys design with SDRAM interface, then we will update the Quartus design and finally add the C-code to generate the Sine waves.

14.5.1 Modify QSys

First, Open the UART_Qsys.qsys file in QSys software. Now, add SDRAM controller with default settings, as shown in Fig. 14.9. Next, connect all the ports of SDRMA as shown in Fig. 14.10. Then, double click the ‘nios2_qsys_0’ and select ‘SDRAM’ as reset and exception vector memory, as shown in Fig. 14.11.

Next, we will add ‘Switches’ to control the amplitude of the sine waves. For this add the PIO device of ‘8 bit with...
type input’, and rename it as ‘switch’, as shown in Fig. 14.12. Finally, go to System–>Assign base addresses, and generate the system.

![Fig. 14.12: Add switches for controlling the amplitude of sine waves](image)

### 14.5.2 Modify Top level Quartus design

Now, open the ‘Uart_top.bdf’ file in Quartus. Right click on the ‘Uart_Qsys’ block and select ‘Update symbol or block’; then select the option ‘Selected symbol(s) or block(s)’ and press OK. It will display all the ports for ‘SDRAM’ and switches. Next, we need to assign the correct ‘pin names’ to these ports, as shown in Fig. 14.13.

![Fig. 14.13: Assigning Pins to SDRAM and Switches](image)

Note that, there should be ‘-3 ns clock delay’ for SDRAM as compare to FPGA clock, therefore we need to add the clock with ‘-3 ns delay’. For this, double click on the Uart_top.bdf (anywhere in the file), and select ‘MegaWizard Plug-In Manager’. Then select ‘Create a new custom megafunction variation’ in the popped-up window and click next. Now, select ALTPLL from IO in Installed Plug-Ins option, as shown in Fig. 14.14, and click next. Then, follow the figures from Fig. 14.15 to Fig. 14.20 to add the ALTPLL to current design i.e. ‘Uart_top.bdf’. Finally, connect the ports of this design as shown in Fig. 14.21. Note that, in these connections, output of ALPLL design is connected to ‘DRAM_CLK’, which is clock-port for DRAM. Lastly, compile and load the design on FPGA board.

### 14.5.3 Updating NIOS design

Since, we have updated the QSys design, therefore the corresponding .sopcinfo file is also updated. Further, BSP files depends on the .sopcinfo file, therefore we need to update the BSP as well. For this, right click on ‘Uart_comm_bsp’ and go to ‘NIOS II–>BSP Editor; and update the BSP as shown in Fig. 14.22 and click on
Chapter 14. UART, SDRAM and Python

Fig. 14.14: ALTPLL generation

Fig. 14.15: ALTPLL creation, step 1
14.5. SDRAM Interface

Fig. 14.16: ALTPLL creation, step 2

Fig. 14.17: ALTPLL creation, step 3
Chapter 14. UART, SDRAM and Python

Fig. 14.18: ALTPLL creation, step 4

Fig. 14.19: ALTPLL creation, step 5

Fig. 14.20: ALTPLL creation, step 6
14.5. SDRAM Interface

Fig. 14.21: Connect ALTPPLL design with existing design

‘generate’ and then click ‘exit’. Note that, ‘enable’ options are unchecked now, because we are using External
memory, which is quite bigger than onchip-memory, so we do not need ‘small’ size options.

Fig. 14.22: Update BSP for new Qsys design

Now, update the ‘hello_world.c’ file as shown in Listing 14.1.

Listing 14.1: Sin and Cos wave generation

```
//hello_world.c
#include "io.h"
#include "alt_types.h"
#include "system.h"
#include "math.h"

int main()
{
    float i=0, sin_value, cos_value;
    alt_u8 amplitude;

    while(1){
        amplitude = IORD(SWITCH_BASE, 0);

        sin_value = (int)amplitude * (float)sin(i);
        cos_value = (int)amplitude * (float)cos(i);

        printf("%.2f	%.2f\n", sin_value, cos_value);
        i = i+0.01;
    }
```

(continues on next page)
In Tera Term, we can save the received values in text file as well. Next, go Files->Log and select the filename at desired location to save the data e.g. ‘sineData.txt’.

Finally, right click on ‘UART_comm_app’ in NIOS and go to ‘Run As–>3 NIOS 2 Hardware’. Now, we can see the decimal values on the screen. If all the switches are at ‘0’ position, then values will be ‘0.000’ as amplitude is zero. Further, we can use any combination of 8 Switches to increase the amplitude of the sine and cosine waves. Also, result will be stored in the ‘sineData.txt’ file. Content of this file is shown in Fig. 14.23

```
2.407716, -1.789666
2.389699, -1.813653
2.371444, -1.837459
2.352951, -1.861081
2.334223, -1.884517
2.315261, -1.907765
2.296068, -1.930821
```

Fig. 14.23: Content of ‘sineData.txt’ file

### 14.6 Live plotting the data

In the previous section, we store the sine and cosine wave data on the ‘sineData.txt’ using UART communication. Now, our last task is to plot this data continuously, so that it look line animation. For this save the Listing 14.2, in the location where ‘sineData.txt’ is saved. Now, open the command prompt and go to the location of python file. Finally, type ‘python main.py’ and press enter. This will start plotting the waveform continuously based on the data received and stored on the ‘sineData.txt’ file. The corresponding plots are shown in Fig. 14.24.

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig = plt.figure()
ax1 = fig.add_subplot(2,1,1)
ax2 = fig.add_subplot(2,1,2)

def animate(i):
    readData = open("sineData.txt","r").read()
data = readData.split(\'\n\'
)
sin_array = []
cos_array = []
for d in data:
    if len(d)>1:
        sin, cos = d.split(\',\')
sin_array.append(sin)
cos_array.append(cos)
ax1.clear()
ax1.plot(sin_array)
ax2.clear()
ax2.plot(cos_array)

def main():
```

(continues on next page)
In this chapter, first we display the ‘Hello’ message using UART and Tera Term. Then, SDRAM is included in the design and correspondingly all the other designs are updated i.e. Quartus and NIOS. Then, the data is stored in the text file and finally it is plotted with the help of Python programming language.

Fig. 14.24: Plot of ‘sineData.txt’ file
Who says God has created this world? We have created it by our own imagination. God is supreme, independent. When we say he has created this illusion, we lower him and his infinity. He is beyond all this. Only when we find him in ourselves, and even in our day to day life, do all doubts vanish.

—Meher Baba

Appendix A

Script execution in Quartus and Modelsim

To use the codes of the tutorial, Quartus and Modelsim softwares are discussed here. Please see the video: Create and simulate projects using Quartus and Modelsim, if you have problem in using Quartus or Modelsim software.

A.1 Quartus

In this section, ‘RTL view generation’ and ‘loading the design on FPGA board’ are discussed.

A.1.1 Generating the RTL view

The execute the codes, open the ‘overview.qpf’ using Quartus software. Then go to the files and right-click on the vhdl file to which you want to execute and click on ‘Set as Top-Level Entity’ as shown in Fig. 1.1. Then press ‘ctrl+L’ to start the compilation.

To generate the designs, go to Tools–>Netlist Viewer–>RTL Viewer; and it will display the design.

A.1.2 Loading design on FPGA board

Quartus software generates two types of files after compilation i.e. ‘.sof’ and ‘.pof’ file. These files are used to load the designs on the FPGA board. Note that ‘.sof’ file are erased once we turn off the FPGA device; whereas ‘.pof’ files are permanently loaded (unless removed or overwrite manually). For loading the design on the FPGA board, we need to make following two changes which are board specific,

- First, we need to select the board by clicking on Assignments–>Device, and then select the correct board from the list.
- Next, connect the input/output ports of the design to FPGA board by clicking on Assignments–>Pin Planner. It will show all the input and output ports of the design and we need to fill ‘location’ column for these ports.
- To load the design on FPGA board, go to Tools–>Programmer.
- Then select JTAG mode to load the ‘.sof’ file; or ‘Active Serial Programming’ mode for loading the ‘.pof’ file. Then click on ‘add file’ and select the ‘.sof/.pof’ file and click on ‘start’. In this way, the design will be loaded on FPGA board.

A.2 Modelsim

We can also verify the results using modelsim. Follow the below steps for generating the waveforms,

- First, open the modelsim and click on ‘compile’ button and select all (or desired) files; then press ‘Compile’ and ‘Done’ buttons. as shown in Fig. 1.2.
A.2. Modelsim

Fig. 1.1: Quartus

Fig. 1.2: Modelsim: Compile and Simulate
Above step will show the compile files inside the ‘work library’ on the library panel; then right click the desired file (e.g. comparator2Bit.vhd) and press ‘simulate’, as shown on the left hand side of the Fig. 1.2. This will open a new window as shown in Fig. 1.3.

Fig. 1.3: Modelsim: Waveforms

- Right click the name of the entity (or desired signals for displaying) and click on ‘Add wave’, as shown in Fig. 1.3. This will show all the signals on the ‘wave default’ panel.
- Now go to transcript window, and write following command there as shown in the bottom part of the Fig. 1.3. Note that these commands are applicable for 2-bit comparators only; for 1-bit comparator assign values of 1 bit i.e. ‘force a 1’ etc.

```
force a 00
force b 01
run
```

Above lines with assign the value 00 and 01 to inputs ‘a’ and ‘b’ respectively. ‘run’ command will run the code and since ‘a’ and ‘b’ are not equal in this case, therefore ‘eq’ will be set to zero and the waveforms will be displayed on ‘wave-default’ window, as shown in Fig. 1.3. Next, run following commands,

```
force a 01
run
```

Now ‘a’ and ‘b’ are equal therefore ‘eq’ will be set to 1 for this case. In this way we can verify the designs using Modelsim.
Appendix B

How to implement NIOS-designs

Please implement the designs of Chapter 1 (i.e. VHDL design) and Chapter 12 (i.e. NIOS design), before following this section.

Unlike VHDL designs, NIOS designs cannot be run directly on a system just by downloading it. Therefore, only required design-files are provided (instead of complete project) for NIOS systems. We need to follow the steps provided in this section, to implement the NIOS design on the FPGA.

Note that VHDL codes and C/C++ codes of the tutorials are available on the website; these codes are provided inside the folders with name VHDLCodes (or vhdl) and CppCodes (or c, or software/ApplicationName_app) respectively. Along with these codes, .qsys files are also provided, which are used to generated the .sope and .sopcinfo files. Lastly, pin assignments files for various Altera-boards are also provided in the zip folders.

Please follow the below step, to compile the NIOS design on new system. Further, if you change the location of the project in the computer (after compiling it successfully), then NIOS design must be implemented again by following the instruction in Section B.4

Note: Codes files used in this section, are available in the folder ‘Appendix-How_to_implement_Nios_design’, which can be ‘downloaded from the website.

B.1 Create project

First create a new Quartus project (with any name) as shown in Section 1.2; and copy all the downloaded files (i.e. VHDLCodes, CppCodes, Pin-assignments and .qsys files) inside the main project directory.

B.2 Add all files from VHDLCodes folder

- Next, add all the files inside the folder ‘VHDLCodes’, to project as shown in Fig. 2.1. Do not forget to select ‘All files’ option while adding the files as shown in Fig. 2.1.
- In Chapter 1, we created ‘VHDL codes’ from the ‘Block schematic design’. These two designs are same, therefore while compilation the multiple-design error will be reported. Therefore we need to remove the duplicate designs as shown in Fig. 2.2. Note that, there are two duplicate designs i.e. one for half_adder and other is for full_adder as shown in the figure.
- In this project, ‘full_adder_nios_test.bdf’ is the top-level design, which is shown in Fig. 2.3. Note that, here ‘name method’ is used to connect the ‘addr_input[2..0] with port ‘a’, ‘b’ and ‘c’. The method for giving name to a wire is shown in figure (see on the bottom-left side).
- Now, select ‘full_adder_nios_test.bdf’ as the top level entity, as shown in Fig. 2.4.
Appendix B. How to implement NIOS-designs

Fig. 2.1: Add all files from VHDLCodes folder

Fig. 2.2: Add all files from VHDLCodes folder
B.2. Add all files from VHDLCodes folder

Fig. 2.3: Select this design i.e. ‘full_adder_nios_test.bdf’ as top level entity

Fig. 2.4: Select top level entity
Appendix B. How to implement NIOS-designs

- Modify the pin-assignment file and import it to the project. Also, make sure that correct FPGA-device is selected for the project. If problem in pin-assignments or device selection, then see Chapter 1 again.

B.3 Generate and Add QSys system

Open the Qsys from Tools→Qsys; and then open the downloaded '.qsys' file and follow the below steps,

- First, refresh the system, by clicking on Files→Refresh System.
- Next, select the correct device type as shown in Fig. 2.5.

![Fig. 2.5: Change device family](image)

- Now, assign base addresses and interrupt numbers by clicking on System→Assign base addresses' and 'Assign interrupt numbers'.
- If there are some errors after following the above steps, then delete and add the Nios-processor again; and make the necessary connection again i.e. clock and reset etc. Sometimes we may need to create the whole Qsys-design again, if error is not removed by following the above steps.
- Finally, generate the system as shown in Fig. 2.6 (or refer to Fig. 12.13 for generating system, if simulation is also used for NIOS design). Finally, close the Qsys after getting the message ‘Generate Completed’.
- Finally, add the Qsys design to main project. For this, we need to add the '.qip' file generated by Qsys, which is available inside the synthesis folder. To add this file, follow the step in Fig. 2.1. You need to select the ‘All files’ option again to see the ‘.qip file’ as shown in Fig. 2.7.
- Now, compile and load the design on FPGA system.

B.4 Nios system

Next, we need to create the NIOS system. For this, follow the below steps,

- Follow the steps in Section 12.5 and ref{sec_add_modify_bsp' to create the NIOS-BSP file. Note that, you need to select the '.sopcinfo' file, which is inside the current main-project-directory.
- Next, we need to create the application file. To create the application, go to File→New→Nios II Application. Fill the application name e.g. ‘Application_fullAdder’ and select the BSP location as shown in Fig. 12.18.
- Note that, if ‘c code’ is provided inside the ‘software folder (not in the ‘CppCodes’ or ‘c’ folders)’ e.g. ‘software/fullAdder_app’, then copy and paste folder-name as the application name i.e. ‘fullAdder_app’ to create the application file. Note that, we usually add ‘_app’ at the end of application name and ‘_bsp’ at the end of BSP name. In this case, ‘c code’ will automatically added to the project. Next, right click on the ‘c file’ and select ‘add to NIOS II build’; and skip the next step of adding ‘c file’, as it is already added in...
B.4. Nios system

Fig. 2.6: Generate QSys system

Fig. 2.7: Change device family
Appendix B. How to implement NIOS-designs

this case. Please see the video: Appendix - How to implement NIOS design, if you have problem in this part of tutorial.

• Next, we need to import the ‘c’ code from folder ‘CppCodes (or c)’. For this, right click on the application and click on Import–>General–>File System–>From Directory; browse the directory, where we have saved the CppCodes and select the files as shown in Fig. 2.8. Finally, simulate the system as described in Section 12.8.

![Fig. 2.8: Adding C files](image)

• Finally, simulate or load the design on FPGA. Please refer to Section 12.8 for simulation; and to Section 12.11 for loading the NIOS design on FPGA. **Do not forget to keep reset button high, while loading the NIOS II design.**

• The current example will display the outputs on NIOS terminal, as shown in Fig. 2.9. Also, sum and carry values will be displayed on the LEDs.

![Fig. 2.9: Nios Output of current design](image)