INTRODUCTION

The Practical Workbook for “VLSI Design” focuses on hands on examples of ASIC Design using a Hardware Description Language (HDL). Verilog and VHDL are two most famous HDLs used these days. The workbook specially focuses on Verilog but partially on VHDL as well. The brief introduction of both VHDL and Verilog HDL along with their comparison by examples is given in Appendices.

Each lab session begins with a brief theory and procedure of implementing a particular logic. Many details that are expected to be covered in the theory classes are not discussed here.

First two lab sessions are centered on the design of combinational circuits using Verilog HDL, behavioral and gate level modeling techniques. Third lab focuses on sequential circuit designing and the next lab is designed to provide a practical knowledge to the students about implementing Finite State machines which play very important role in VLSI design of digital circuits. Fifth lab enables the students to learn the design of synchronous shifting operation and data storage. Sixth lab is again a real-world design problem based on pattern matching technique in data communications. The next lab focuses on the design of a basic memory cell and a bidirectional port in Verilog HDL. In this lab, HDL synthesis is also covered.

Lab eight and nine cover the topic of data path design techniques with the help of two design examples. In Lab ten, students learn to design combinational shifter logic circuit and in the eleventh lab, a type of feedback shift register is to be designed to demonstrate the method of generating pseudo random codes.

Lab twelve is left for the teachers to randomly assign any real-world mini project in the field of VLSI design and digital logic.

Appendices cover the basics of digital design using Hardware Description Languages including both VHDL and Verilog and their comparison by given examples.
## CONTENTS

<table>
<thead>
<tr>
<th>Lab Session No.</th>
<th>Object</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Design and simulate <strong>2-to-4-line decoder</strong> using Verilog HDL</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Design and simulate <strong>3-to-8-line decoder</strong> using 2-to-4-line decoder</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Design and simulate <strong>8-bit Counter</strong> in Verilog HDL</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>Implementation of <strong>Finite State Machine</strong> in Verilog HDL</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>Design an <strong>8-bit PIPO-SISO shift register</strong> in Verilog HDL using behavioral description.</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>Design a <strong>64-bit CORRELATOR</strong> which produces 7-bit correlation score of two input words of up to 64-bits.</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>Design an <strong>8-bit RWM cell</strong> with bidirectional data bus and synthesize the design using the synthesis tool.</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>Design an <strong>8-bit ALU</strong> which performs eight basic arithmetic and logic operations and maintains a Flag register</td>
<td>31</td>
</tr>
<tr>
<td>9</td>
<td>Design a <strong>4-bit Bit-Serial adder</strong> in Verilog HDL, simulate it and synthesize using the synthesis tool</td>
<td>35</td>
</tr>
<tr>
<td>10</td>
<td>Design an <strong>8-bit Funnel Shifter</strong> that can perform all shift operations</td>
<td>40</td>
</tr>
<tr>
<td>11</td>
<td>Design a <strong>3-bit pseudo random pattern generator</strong> using linear feedback shift register (LFSR)</td>
<td>44</td>
</tr>
<tr>
<td>12</td>
<td>A <strong>Mini Project</strong> on any Current Practical Research oriented topic</td>
<td>48</td>
</tr>
</tbody>
</table>

Appendix A: Learning **basics of Verilog HDL**

Appendix B: Learning **basics of VHDL**

Appendix C: **Verilog and VHDL – Comparison** by Examples
Lab Session 01

OBJECT

- Design 2-to-4-line decoder using both Verilog HDL and VHDL and simulate it using any simulation technique.
- Follow the behavioral modeling technique to design the module

THEORY

The basic function of a decoder is to detect the specified combination of bits on its inputs and to indicate the presence of that code by a specified output level on a particular pin. In its general form, a decoder has n input lines and $2^n$ output lines.

The truth table and a block diagram (showing IO ports) of 2-to-4-line decoder are given below

<table>
<thead>
<tr>
<th>SNo</th>
<th>En</th>
<th>In1</th>
<th>In0</th>
<th>Out3</th>
<th>Out2</th>
<th>Out1</th>
<th>Out0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(Table 1.1)

![Block Diagram of 2-to-4-line Decoder](image)
<table>
<thead>
<tr>
<th>Verilog HDL Code</th>
<th>VHDL Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-to-4-line decoder module (using Case statement)</td>
<td>2-to-4-line decoder module (using Case statement)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Verilog test bench</td>
<td>VHDL test bench</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
EXERCISE

Apply the assigned bit values to the designed 2-to-4-line decoder for either Verilog or VHDL to complete the table.

<table>
<thead>
<tr>
<th>SNo</th>
<th>En</th>
<th>In1</th>
<th>In0</th>
<th>Out3</th>
<th>Out2</th>
<th>Out1</th>
<th>Out0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Instructor should assign different bit values in the shaded cells to individual groups)
Lab Session 02

OBJECT

- Design structural model of 3-to-8-line decoder using both Verilog HDL and VHDL and simulate it using any simulation technique
- Use two instances of 2-to-4-line decoder module, designed in Laboratory session 01, in cascade configuration to form 3-to-8-line decoder

THEORY

The basic function of a decoder is to detect the specified combination of bits on its inputs and to indicate the presence of that code by a specified output level on a particular pin. In its general form, a decoder has n input lines and $2^n$ output lines.

The block diagram (showing IO ports) and truth table of 3-to-8-line decoder are given below:

<table>
<thead>
<tr>
<th>SNo</th>
<th>In2</th>
<th>In1</th>
<th>In0</th>
<th>En_1</th>
<th>En_0</th>
<th>Out7</th>
<th>Out6</th>
<th>Out5</th>
<th>Out4</th>
<th>Out3</th>
<th>Out2</th>
<th>Out1</th>
<th>Out0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(Table 2.1)

(Figure 2.1)
<table>
<thead>
<tr>
<th>Verilog HDL Code</th>
<th>VHDL Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-to-8-line decoder module</td>
<td>3-to-8-line decoder module</td>
</tr>
</tbody>
</table>
Test bench for Verilog only
EXERCISE

1. Apply the assigned bit values to the designed 3-to-8-line decoder to complete the table.

<table>
<thead>
<tr>
<th>SNo</th>
<th>In2</th>
<th>In1</th>
<th>In0</th>
<th>Out7</th>
<th>Out6</th>
<th>Out5</th>
<th>Out4</th>
<th>Out3</th>
<th>Out2</th>
<th>Out1</th>
<th>Out0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   (Instructor should assign different bit values in the shaded cells to individual groups)

2. Attach printout of the waveform of different Input/Output signals in your design
Lab Session 03

OBJECT

Design an 8-bit synchronous counter using both Verilog HDL and VHDL and simulate it using any simulation technique.

THEORY

A counter is a sequential circuit that is used to count the number of signal transitions at the clock input.

A counter may also have provision to be initialized to a given value at the input so that subsequent transitions are added to this value to give total count at its output. A counter may also have a reset input signal that is used to reset the count to zero. The block diagram of such a counter to be called as up-counter is given below:

(Figure 3.1)
### Verilog HDL Code

**8-bit Synchronous Counter**

<table>
<thead>
<tr>
<th>Verilog HDL Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

### VHDL Code

**8-bit Synchronous Counter**

<table>
<thead>
<tr>
<th>VHDL Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Test bench for VHDL only
EXERCISE
1. Apply the assigned values to the designed 8-bit counter to complete the table.
   - Set Clock period = 20 time units in test bench.
   - “Time” column in the table will define the time units at which the students are required to observe the outputs.

<table>
<thead>
<tr>
<th>SNo</th>
<th>Time</th>
<th>Reset</th>
<th>InCount</th>
<th>Init/Countbar</th>
<th>CountOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>120</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>250</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>300</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>400</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>470</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>550</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Instructor should assign different values in the shaded cells to individual groups)

2. Modify the figure 3.1 by adding one or more single-bit input signal “UpDown”. This new module now should count in increment fashion if UpDown = 1 and in decrement fashion if UpDown = 0. Create the Verilog module for this scenario.

3. By setting Clock Period = 10 time units in the test bench for newly created module, complete the table given below:

<table>
<thead>
<tr>
<th>SNo</th>
<th>Time</th>
<th>Reset</th>
<th>InCount</th>
<th>Init/Countbar</th>
<th>UpDown</th>
<th>CountOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Instructor should assign different values in the shaded cells to individual groups)
Lab Session 04

OBJECT

Implement the given Finite State Machine in Verilog HDL at gate level using D-Flip flop as the basic building block. Synthesize and simulate it.

THEORY

A machine with finite number of states is to be called as Finite State Machine (FSM). A synchronous FSM is the one for which states and output transitions are synchronized with a clock (positive or negative edge). Any FSM can be described as a Moore machine or a Mealy machine.

A Moore machine is one, which associates outputs with states rather than transition. And a Mealy machine is one, which associates outputs with both the present state and input.

A Moore machine can easily be converted to a behaviorally equivalent Mealy machine just by addition or removal of one or more states and vice versa for Mealy machines.

State tables for both Moore and mealy machines are given below:

State Table (Moore Machine)

<table>
<thead>
<tr>
<th>Present state</th>
<th>0</th>
<th>1</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1</td>
<td>q4</td>
<td>q3</td>
<td>0</td>
</tr>
<tr>
<td>q2</td>
<td>q3</td>
<td>q4</td>
<td>1</td>
</tr>
<tr>
<td>q3</td>
<td>q2</td>
<td>q1</td>
<td>0</td>
</tr>
<tr>
<td>q4</td>
<td>q1</td>
<td>q4</td>
<td>1</td>
</tr>
</tbody>
</table>

Next states when Input (0/1) is applied

Output when Input (0/1) is applied to a particular state

State Table (Mealy Machine)

<table>
<thead>
<tr>
<th>Present state</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1</td>
<td>Q3</td>
<td>1</td>
</tr>
<tr>
<td>q2</td>
<td>Q2</td>
<td>0</td>
</tr>
<tr>
<td>q3</td>
<td>Q4</td>
<td>1</td>
</tr>
<tr>
<td>q4</td>
<td>q4</td>
<td>0</td>
</tr>
</tbody>
</table>

Where;
I : set of Inputs
Z: set of Outputs
Q: set of States
N(qi, Ij): Next state function
Z(qi, Ij): next Output function
Below is given a synchronous 5-state Moore type FSM that as described above is synchronized with a clock. Your task is to implement this FSM using three always block style (i.e. registered outputs).

(Figure 4.1)

Assume,
- *Clk* as the clock input for synchronization. At the positive edge (posedge) of clk, the present state may change to next state or to initial state.
- *nrst* as the asynchronous reset signal. At the negative edge (negedge) of nrst, the FSM goes to reset mode i.e. initial state.
- Binary state assignments are to be used
- *i1, i2, i3, i4* are the inputs of this FSM
- *n_o1, o2, o3, o4, err* are the outputs of this FSM
PROCEDURE

- Start creating a new project “fsmDesign” in verilog HDL. Create a new verilog module “fsmMooreI”
- Write down all the input/output ports in the module list and declare their respective type as input or output ports.
- Using parameter or `define statement, assign 3-bit unique binary combination to all state variables.
- Declare the reg variables as necessary.
- Write three always block code. Two sequential always blocks for both present state and output logic and one combinational always block for next state logic.
Verilog HDL Code
Test bench
EXERCISE

1. Apply the assigned values to the designed FSM to complete the table.
   - Value for clk signal should autochange after every 10 time units.

<table>
<thead>
<tr>
<th>SNo</th>
<th>Time</th>
<th>nrst</th>
<th>i1</th>
<th>i2</th>
<th>i3</th>
<th>i4</th>
<th>P State</th>
<th>N State</th>
<th>o1</th>
<th>o2</th>
<th>o3</th>
<th>o4</th>
<th>err</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   (Instructor should assign different values in the shaded cells to individual groups)

2. Create another verilog module “fsmMoore2” in the same project “fsmDesign”. For now use the output encoded style for the same fsm design. Write your verilog code on separate sheet of paper and submit to your teacher.
Lab Session 05

OBJECT

Design an 8-bit PIPO-SISO shift register in Verilog HDL using behavioral description. Show the RTL block diagram of your design as well.

- Use DFF as the basic sub-block for RTL block diagram. Make all necessary assumptions.

THEORY

The functionality of the module is described as follows:

a. 8-bit word, placed at the 8-bit ParallelInput port of the module, would be stored into the register if a LOW is asserted on ShiftLoadbar pin and would be made available at 8-bit ParallelOutput port.

b. If a HIGH is placed at ShiftLoadbar pin, at every rising edge of ShiftClock the register value will be left shifted once. The ParallelOutput would be updated accordingly.

c. During a shift left operation, MSB would be discarded and the logic level at SerialInput pin would be shifted in at the LSB. SerialOutput pin value would be updated accordingly.

d. If Enable pin is logic ‘1’, normal operation would be performed as described in points a to d. otherwise, the module would be in idle state and would hold the previous value regardless of the status of ShiftLoadbar and ShiftClock.

e. ShiftLoadbar and Enable pins are asynchronous inputs.

(Figure 5.1)
Verilog HDL Code

8-bit Shift Register Module

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---
Test bench
EXERCISE

Apply the assigned values to the designed shift register to complete the table.
- Value for clk signal should autochange after every 10 time units.

<table>
<thead>
<tr>
<th>SNo</th>
<th>Time</th>
<th>Enable</th>
<th>Shift Loadbar</th>
<th>Parallel Input</th>
<th>Serial Input</th>
<th>Parallel Output</th>
<th>Serial Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Instructor should assign different values in the shaded cells to individual groups)
Lab Session 06

OBJECT

Design a 64-bit CORRELATOR which produces 7-bit correlation score of two input words of up to 64-bits.

- Draw the RTL block diagram showing the interconnection of different sub-modules.
- Code the design in using Verilog HDL.

THEORY

The functionality of the module is described as follows:

f. 64-bit CORRELATOR module produces 7-bit correlation score of two input words of up to 64-bits denoted by A and B.
g. The A and B inputs are serially shifted into two independently clocked 64-bit shift registers.
h. The shift register “A” is clocked on rising edge of clock ‘A’ and shift register ‘B’ is clocked on rising on rising edge of clock ‘B’.
i. Contents of register ‘B’ are latched in Latch ‘C’ using a high on ‘LCL’ (Load C Latch).
j. Register ‘A’ and register ‘C’ are compared Bit by bit with each other and total number of successful comparisons are compared with a threshold value stored in a 7-bit Threshold Register.
k. A mask register ‘M’ is a 64-bit register which is loaded serially using ‘M’ input at the rising edge of clock ‘M’ and is used to enable of disable the correlation operation at each bit position.
l. If successful comparisons are more than or equal to the threshold value, successful correlation flag CFL is asserted.

(Figure 6.1)
Verilog HDL Code

64-bit Correlator Module
Test bench
EXERCISE

Apply the assigned values to the designed correlator to complete the table. Apply the following Clock to all the Shift Clocks of your correlator module.

- Set Clock period = 40 time units
- Set duty cycle = 50%

<table>
<thead>
<tr>
<th>SNo.</th>
<th>Test Pattern</th>
<th>A</th>
<th>B</th>
<th>M</th>
<th>ThIn</th>
<th>ThOut</th>
<th>CFL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Instructor should assign different bit values in the shaded cells to individual groups)
Lab Session 07

OBJECT

Design an 8-bit RWM cell with bidirectional data bus and synthesize the design using the synthesis tool.

THEORY

The functionality of the module is described as follows:

m. RWM cell provides 8-bit data storage.

n. Data is written by placing 8-bit word on the DB_8 (8-bit data bus) and enabling active low WR (write) signal.

o. Data is read through the DB_8 by enabling active low RD (read) signal.

p. RWM cell operation is enabled using the CS (chip select) input which is active low.

q. The 8-bit data bus must be designed as a bidirectional port and must be kept in high impedance state when the cell is disabled.

(Figure 7.1)
Verilog HDL Code

8-bit RWM Module

``````
Test bench
EXERCISE

Apply the assigned values to the designed RWM module to complete the table.

<table>
<thead>
<tr>
<th>SNo</th>
<th>Time</th>
<th>CS</th>
<th>RD</th>
<th>WR</th>
<th>DB_8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Instructor should assign different values in the shaded cells to individual groups)
Lab Session 08

OBJECT

Design an 8-bit ALU which performs following arithmetic and logic operations.

- ADD (ADD)
- SUBTRACT (SUB)
- LOGICAL SHIFT LEFT (LSL)
- LOGICAL SHIFT RIGHT (LSR)
- AND (AND)
- OR (OR)
- NOT (CMP)
- XOR (XR)

- ALU must also maintain a FLAG register showing the status of the result after arithmetic and logic operations. The FLAG register must store the following flags.

  - Sign (S)
  - Carry (C)
  - Zero (Z)
  - Overflow (OV)

- Draw the functional block diagram showing all the I/O ports and RTL block diagram showing all the sub-modules of ALU.

THEORY

The functionality of the module is described as follows:

r. The ALU takes two 8-bit operands stored in register A and B as inputs and performs 2’s complement arithmetic operations (ADD/SUB).
s. Bitwise logical operations (AND/OR/XR) are also performed on both the operands (A and B).
t. (NOT/ LSL/LSR) operations are performed on one operand stored in register A.
u. The result of any operation is stored in register X.
v. The ALU also contains a 4-bit Flag register and is updated after completion of each operation.
w. The functionality of each flag is shown as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Bit Position</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign (S)</td>
<td>0</td>
<td>Set if result is negative otherwise reset</td>
</tr>
<tr>
<td>Carry (C)</td>
<td>1</td>
<td>Set if carry is generated otherwise reset</td>
</tr>
<tr>
<td>Zero (Z)</td>
<td>2</td>
<td>Set if the result is 00h otherwise reset</td>
</tr>
<tr>
<td>Overflow (OV)</td>
<td>3</td>
<td>Set if an overflow occurs otherwise reset</td>
</tr>
</tbody>
</table>

(Table 8.1)
Verilog HDL Code

ALU module
Test bench
EXERCISE

Apply the assigned values to the ALU module and complete the table below.

<table>
<thead>
<tr>
<th>SNo</th>
<th>Operation</th>
<th>A</th>
<th>B</th>
<th>X</th>
<th>V</th>
<th>Z</th>
<th>C</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Instructor should assign different values in the shaded cells to individual groups)
Lab Session 09

OBJECT

Design a 4-bit Bit-Serial adder in Verilog HDL, simulate it and synthesize using the synthesis tool

THEORY

A bit-serial adder consists of one full adder, two source operand shift registers, one destination operand shift register and a carry latch. The two source operands are serially applied to the inputs of a full adder and the result is shifted in the destination operand shift register LSB first.

The functionality of the module is described as follows:

x. Two 4-bit Parallel-In-Serial-Out (PISO) shift registers A and B are used to store the two 4-bit source operands and apply bit by bit to the inputs of the full adder LSB first.
y. The bit by bit sum output, LSB first, from the full adder is shifted into a 4-bit Serial-In-Parallel-Out (SIPO) shift register X to store the Destination operand.
z. A Carry latch is connected between the carry output and carry input of the full adder to propagate the carry to the next summation operation.

aa. At start of addition operation, Carry latch can be set or reset depending on the C\textsubscript{in}
bb. At the end of addition operation, Carry latch holds the C\textsubscript{out} signal.
cc. A shift clock SClk is used to derive the shift registers and the Carry latch. One 4-bit addition operation completes in 4 clock cycles of SClk.

(Figure 9.1)
RTL Block Diagram

(Figure 9.2)
Verilog HDL Code

Bit Serial Adder module
Test bench
EXERCISE

Apply the following values to the Adder module and complete the table below.

- Set SClk = 1/40 cycles per unit

<table>
<thead>
<tr>
<th>SNo</th>
<th>A</th>
<th>B</th>
<th>X</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Instructor should assign different values in the shaded cells to individual groups)
Lab Session 10

OBJECT

Design an 8-bit Funnel Shifter that can perform all shift operations at any number of bits (< 8) just in a single operation. Your design should have asynchronous reset input that resets values of different registers to zero.

- Draw the block diagram of the design yourself.
- Write the Verilog code and test bench program to simulate it.

THEORY

A Funnel shifter is a combinational hardware device that can shift or rotate a data word by any number of bits in a single operation. As shown in figure below, it concatenates two N-bit inputs and selects an N-bit subfield Y as an output. Selection of Y field out of 16 bits starts and ends as:

- Starts from right at the bit position = offset (look at the table for offset values)
- Ends at bit position = offset + N - 1 (i.e. offset + 7 for 8-bit subfield)

![Figure 10.1](image)

The table below lists what inputs should be applied to perform a left or right shift of an N-bit word A by k bits.

<table>
<thead>
<tr>
<th>Shift Type</th>
<th>B</th>
<th>C</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Right</td>
<td>0……….0</td>
<td>A&lt;sub&gt;N-1&lt;/sub&gt;…..A&lt;sub&gt;0&lt;/sub&gt;</td>
<td>k</td>
</tr>
<tr>
<td>Logical Left</td>
<td>A&lt;sub&gt;N-1&lt;/sub&gt;…..A&lt;sub&gt;0&lt;/sub&gt;</td>
<td>0……….0</td>
<td>N-k</td>
</tr>
<tr>
<td>Arithmetic Right</td>
<td>A&lt;sub&gt;N-1&lt;/sub&gt;…..A&lt;sub&gt;N-1&lt;/sub&gt; (sign extension)</td>
<td>A&lt;sub&gt;N-1&lt;/sub&gt;…..A&lt;sub&gt;0&lt;/sub&gt;</td>
<td>k</td>
</tr>
<tr>
<td>Arithmetic Left</td>
<td>A&lt;sub&gt;N-1&lt;/sub&gt;…..A&lt;sub&gt;0&lt;/sub&gt;</td>
<td>0……….0</td>
<td>N-k</td>
</tr>
<tr>
<td>Rotate Right</td>
<td>A&lt;sub&gt;N-1&lt;/sub&gt;…..A&lt;sub&gt;0&lt;/sub&gt;</td>
<td>A&lt;sub&gt;N-1&lt;/sub&gt;…..A&lt;sub&gt;0&lt;/sub&gt;</td>
<td>k</td>
</tr>
<tr>
<td>Rotate Left</td>
<td>A&lt;sub&gt;N-1&lt;/sub&gt;…..A&lt;sub&gt;0&lt;/sub&gt;</td>
<td>A&lt;sub&gt;N-1&lt;/sub&gt;…..A&lt;sub&gt;0&lt;/sub&gt;</td>
<td>N-k</td>
</tr>
</tbody>
</table>

(Table 10.1)
Verilog HDL Code

8-bit Funnel Shifter Module
Test bench
EXERCISE

1. Apply the assigned test patterns to the designed 8-bit Funnel Shifter to complete the table.

<table>
<thead>
<tr>
<th>SNo</th>
<th>Shift Type</th>
<th>A</th>
<th>k</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ASL</td>
<td>8’d233</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>LSL</td>
<td>8’d220</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>RR</td>
<td>8’d110</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>RL</td>
<td>8’h53</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>LSR</td>
<td>8’hF6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Instructor should assign different values in the shaded cells to individual groups)

2. Attach printout of the waveform of different Input/Output signals in your design.
Lab Session 11

OBJECT

Design a 3-bit pseudo random pattern generator using linear feedback shift register (LFSR)

THEORY

A pseudo random pattern generator is constructed from a linear feedback shift register (LFSR), which is constructed, in turn, from a number of 1-bit registers connected in a serial fashion. The outputs of certain shift bits are XORed and fed back to the input of LFSR to calculate the next pseudo random number.

The functionality of the LFSR is described as follows:

dd. A 3-bit LFSR module produces a pseudo random repeatable sequence when it is initialized by a 3-bit seed value and then derived by a shift clock SClk.

ee. LFSR is initialized by loading 3-bit seed value from input SIn.

ff. The current value, stored in LFSR is readable from the 3-bit output port OutPort.

gg. The control input, ShiftLDbar, when derived High, allows the LFSR to sequence pseudo randomly at the rising edge of SClk.

hh. ShiftLDbar is used to load the seed value into LFSR, when derived Low.

ii. The LFSR performs its proper operation when its control input Enable is derived High. Otherwise, it holds the previous value.

jj. All the control inputs are asynchronous.

(Figure 11.1)
Verilog HDL Code

LFSR module
Test bench
EXERCISE

3. By setting Clock period at your own choice apply the assigned seed values to the designed 3-bit LFSR to complete the following table. Make sure that you run the simulation until you get seed value repeated at the register outputs. LFSR also needs at least three active clock pulses for input of the seed value.

<table>
<thead>
<tr>
<th>Seed values &gt;</th>
<th>Q_0Q_1Q_2</th>
<th>Q_0Q_1Q_2</th>
<th>Q_0Q_1Q_2</th>
<th>Q_0Q_1Q_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Instructor should assign different seed values in the shaded cells to individual groups)

4. Attach printout of the waveform of different Input/Output signals in your design for at least one of the assigned seed values.
Lab Session 12

OBJECT
A Mini Project on any Current Practical Research oriented topic

The Mini Project (not necessarily be same for all the students) will be assigned by the Teacher on group bases to various students in accordance with the current developments and practical research oriented topic in the field of VLSI Design. For which students are required to submit hardcopy of the report of their mini project. However students will write in workbook, the summary of not exceeding two pages of their mini project along with conclusion.

MINI PROJECT SUMMARY

___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
Appendix-A

Learning Basics of Verilog HDL

HDL

Traditionally, digital design was done using schematic entry. This has been replaced today by top-down design methodologies using Hardware Description Languages (HDLs). In electronics, HDL is a form of computer language for formal description of electronic circuits. It is a high-level programming language that includes specialized constructs to describe or model the behavior or structure of a hardware system. HDL offers several advantages over traditional design methodologies, such as efficient mechanism for the design, simulation and synthesis of large electronic circuits using millions of transistors, flexibility in changes to design and easy verification of design at early stages of development.

Two popular HDLs in wide use are VHDL and Verilog. While their syntax and semantics are quite different, they are used for similar purposes. The syntax of VHDL is closer to that of programming language ADA, while the syntax of Verilog is closer to that of the C programming Language.

Both the HDLs i.e. Verilog and VHDL are used in our labs. However more focus is on Verilog.

BASICS OF VERILOG HDL

The basic syntax of Verilog HDL is much like C programming language. Verilog is a case sensitive. Any HDL based designs start with description of design in any general language or pseudocodes. Then design is entered in specific HDL using plain text editor e.g. Notepad, Wordpad etc. or any special purpose language editors e.g. Verilog or VHDL editor following the HDL syntax and semantics.

At design entry level Verilog supports four different levels of abstractions which are discussed with examples next.

1) The switch level which includes MOS transistors modelled as switches.
2) The gate level or Structural Level
3) The Data-Flow level.
4) The Behavioral or procedural level.

The language also defines constructs that can be used to control the input and output of simulation.

VERILOG Lexical Conventions

Lexical conventions include comments, white space characters, delimiters, numbers, identifiers, keywords, etc. Because Verilog HDL is a case sensitive language, so all keywords are in lowercase.

Verilog source text files consists of the following lexical conventions:

White Space

White spaces separate words and can contain spaces, tabs, new-lines and form feeds. Thus a statement can extend over multiple lines without special continuation characters.
Comments
Comments can be specified in two ways:

- Begin the comment with double slashes ‘//’. All text between these characters and the end of the line will be ignored by the Verilog compiler.
- Enclose comments between the characters ‘/*’ and ‘*/’. Using this method allows you to continue comments on more than one line. This is good for “Commenting out” many lines of code, or for very brief in-line comments.

Numbers
Number storage is defined as a number of bits, but values can be specified in binary, octal, decimal or hexadecimal.
Examples are 3’b001, a 3-bit number, 5’d30, (=5’b11110), and 16’h5ED4, (=16’d24276)
Numbers may be sized or unsized. The compiler by default assumes 32-bit number if no any size is mentioned.

Identifiers
Identifiers are user-defined words for variables, function names, module names, block names and instance names. Identifiers begin with a letter or underscore following general naming conventions and can include any number of letters, digits and underscores.

Operators
There are a number of operators used in Verilog as listed below. These operators, depending on their nature of use can be categorized as unary, binary or ternary operators.

- Unary (Operating on One variable) Example: a = ~ b;
- Binary (Operating on two variables) Example: a = b & & c;
- Ternary (Operating on Three variables) Example: a = b ? c : d;

<table>
<thead>
<tr>
<th>Operators</th>
<th>Description</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>NOT</td>
<td>Highest</td>
</tr>
<tr>
<td>* , / , %</td>
<td>MUL, DIV, MODULO</td>
<td></td>
</tr>
<tr>
<td>+ , -</td>
<td>PLUS, MINUS</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt; , &gt;&gt;</td>
<td>Logical Left/ShiftRight</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;&lt;&lt; , &gt;&gt;&gt;&gt;</td>
<td>Arithmetic Left/ Shift Right</td>
<td></td>
</tr>
<tr>
<td>&lt; , &lt;= , &gt; , &gt;=</td>
<td>Relative Comparison</td>
<td></td>
</tr>
<tr>
<td>== , !=</td>
<td>Equality Comparison</td>
<td></td>
</tr>
<tr>
<td>&amp; , &amp; &amp;</td>
<td>AND, NAND</td>
<td></td>
</tr>
<tr>
<td>^ , ~ ^</td>
<td>XOR, XNOR</td>
<td></td>
</tr>
<tr>
<td>[ ,</td>
<td>, ]</td>
<td>OR, NOR</td>
</tr>
<tr>
<td>?:</td>
<td>Conditional Operator</td>
<td>Lowest</td>
</tr>
</tbody>
</table>

Verilog Keywords
These are words that have special meaning in Verilog. Some examples are assign, case, while, wire, reg, and, or, nand, and module. They should not be used as identifiers or user defined constructs etc. Verilog keywords also includes Compiler Directives and System Tasks and Functions.

Value Sets in Verilog
Verilog consists of only four basic values:
0 (logic zero, or false condition)
1 (logic one, or true condition)
x (unknown logic value)
z (high impedance state)
x and z have limited use for synthesis.

Data Types

Verilog HDL supports almost all varieties of data types as the general programming languages do. This includes Integer, real, time, arrays, strings, constants etc. However any data type must be associated with any one of the two types of Verilog signals i.e. Wires (Nets) or Registers. Wires or registers generally can be decalred as a single bit scalar or multiple bits vector. Any of bits from vector can be selected with vector part selection method.

Wire
A wire (or net) represents a physical wire in a circuit and is used to connect gates or modules. The value of a wire can be read, but not assigned to, in a function or block. A wire does not store its value but must be driven by a continuous assignment statement or by connecting it to the output of a gate or module.

Syntax
wire [msb:lsb] wire_variable_list;

Example
wire c;       //declare a wire c as scalar i.e. single bit size
wire x,y,z;  //declare wires x, y and z
wire [7:0] p, q;  //declare 8-bit wires p and q with defined vector

Reg
A reg (register) is a data object that holds its value from one procedural assignment to the next. They are used only in functions and procedural blocks. A reg is a Verilog variable type and does not necessarily imply a physical register.

Syntax
reg [msb:lsb] reg_variable_list;

Example
reg a;       // declare a register a as scalar
reg [7:0] r_vector;  // declare 8-bit register r_vector with defined vector

VERILOG MODULE

A module is the principal design entity in Verilog. The first line of a module declaration specifies the name and port list (arguments). The next few lines specifies the input/output type (input, output or inout) and width of each port. The default port width is 1 bit. Then the port variables must be declared wire or reg. The default is wire. Typically inputs are wire since their data is latched outside the module. Outputs are type reg if their signals were stored inside an always or initial block.
Syntax
module module_name (port_list);
input [msb:lsb] input_port_list;
output [msb:lsb] output_port_list;
inout [msb:lsb] inout_port_list;
... statements ...
endmodule

Example
module add_sub(add, in1, in2, sum_12);
input add; //wire by default
input [7:0] in1, in2; //wire by default
output [7:0] sum_12;
reg sum;
... statements ...
endmodule

Assignment in Verilog
There are primarily three different ways to assign any value to any variable. These include Continuous, blocking and non-blocking assignments.
The continuous assignment is used to assign a value onto a wire in a module. It is the normal assignment outside of always or initial blocks. Continuous assignment is done with an assign statement or by assigning a value to a wire during its declaration. The order of assign statements does not matter. A change in any of the right-hand-side inputs will immediately reflect on the left-hand-side output.

Syntax
wire wire_variable = value;
assign wire_variable = expression;

Both blocking and non-blocking assignment statements are used to assign value to a variable of type reg. They are allowed to be written inside any process Always or initial. Blocking assignments use a compound operator “=” to assign value. While as non-blocking uses a compound operator “<=” to assign value. Blocking assignemnts are expected to block the execution of forthcoming statements within process i.e. forces sequential execution, however non-blocking allows concurrent execution.

Example
initial begin
...
xyz = 8’hAB; //Blocking assignment
...
end
initial begin
...
abc <= 8’hF6; //Non-Blocking assignment
...
end
LEVELS OF ABSTRACTION
In Verilog, hardware can be described at four different levels of abstraction: switch-level, data-flow level, gate-level and behavioral level, with switch-level being lowest and behavioral being the highest level. The idea and ease of working on different levels can simply be understood by keeping in mind how you work on general programming languages like C++, Assembly language and machine (binary) language. We will briefly discuss these four levels with possible examples next.

Switch-Level Modeling
Sometimes designers may choose to model the circuit using transistors. This is the lowest level of abstraction, called switch-level or MOS-transistor level modeling. Since the complexity of circuits has increased immensely over the past few years, designers may rarely choose to use this level of design. It is like as to work on binary language (low level) that means very complex and time consuming. Nevertheless, we will study it for completion. Verilog allows definition of MOS switches using keywords ‘nmos’ and ‘pmos’

Example
nmos n1 (out, data, control)    //instantiate nmos switch
pmos p1 (out, data, control);  //instantiate pmos switch

When designing at switch-level, power (vdd, logic1) and ground (vss ,logic 0) also need to be specified, using keywords ‘supply1’ and ‘supply0’ respectively.

Example
module TwoTo1Mux(a, b, s, Out);
input a, b, s;
output Out;
wire w1, w2;
supply1 vcc;
supply0 gnd;
pmos (w1, vcc, s);
pmos (w2, w1, s);
nmos (w2, gnd, s);
nmos (w2, gnd, s);
cmos (Out, a, w2, s);
cmos (Out, b, c, w2);
endmodule

Gate-Level Modeling
At gate level, circuit is described in terms of gates (OR, AND, NAND, etc). This is also a low level of abstraction. It may be understood as analogous to working on assembly language. In Verilog, gates are predefined and can be instantiated without specifying module definition. Any circuit can be designed using basic logic gates called primitives in Verilog.
Let us consider the above example of 2-to-1 Mux now using switch-level modeling.

Example:
module TwoTo1Mux(a, b, s, Out);  //Gate level Modeling
input a, b, s;
output Out;
wire W1,W2;
not  M1(W1, s);
and  M2(W2,a,W1);
and  M3(W3,s,b);
or   M4(Out,W2,W3);
endmodule

Dataflow Modeling
Gate-level modeling approach works well for small circuits when number of gates is limited and connections between gates can be easily specified. However, as the circuits become complex, the number of gates increases and it becomes difficult to instantiate several gates. Dataflow level of design allows a circuit to be designed in terms of dataflow between registers. Designers specify a circuit at dataflow level and automated tools are used to create a gate-level description from the dataflow design. A circuit is simply described using a single Boolean equation by using continuous assignment statement.

Let us again consider our multiplexer example now using Dataflow modeling.

Example
module TwoTo1Mux(a, b, s, Out);     //Data Flow level
input a, b, s;
output Out;
assign Out = s ? b : a;            //Like a simple Boolean implementation
endmodule

Behavioral Modeling
Behavioral level is also named as procedural level. Verilog provides designers the ability to describe and design the functionality in an algorithmic manner, i.e. the designer describes the behavior of the circuit. Just keep in mind working on any high level language like C, C++, VB etc.

Behavioral or procedural statements in verilog are used to model a design at a higher level of abstraction than the other levels. All behavioral code is contained within a process i.e. an initial block or always block which are described next. These behavioral blocks contain statements that control simulation time, data flow statements (like if-else and case) and blocking and non-blocking assignments. Only reg variables and integers can be assigned values within process.

Consider below the Two-to-One Mux example now using behavioral modeling.

Example:
module TwoTo1Mux(a, b, s, Out);     //Behavioral level Modeling
input a, b, s;
output reg Out;

always @ (posedge clk)
begin
  if (s)  Out <= b;
  else    Out <= a;
end
endmodule
Processes
There are two process statements used in Verilog: *Always* and *Initial*. These statements are the two most basic statements in behavioral modeling. All other behavioral statements can appear only inside these structured procedure statements. Each *Always* and initial block begins a concurrent activity and that they can not be nested. There can be many initial blocks however they can be executed once only at simulation time step 0. Again there can be several always blocks which start at time step 0 but executed as many times as their sensitivity list changes. Sensitivity list is the list of events and level triggered variables that follow the keyword ‘Always’.

*Example*:
```verilog
always@(a, b, c, d) //Execute the block always when there is change on a,b,c or d
begin
    ...
end
```
```verilog
always@((posedge Clk) //Execute the block always when there is rising edge of Clock
begin
    ...
end
```
```verilog
always@((negedge Clk) //Execute the block always when there is falling edge of Clock
begin
    ...
end
```

Conditional Structures
Verilog supports two of the most important decision based structures i.e. if-else-if structure and case-endcase structure. They both also support their nested structures. Example use of if-else and case structures is demonstrated below. Note that in if-else structure like general programming languages, block of begin-end is required when there are more statements within any structure. This rule may also be used with case structure. Also note that there is no case keyword used to define case arms in Verilog.

```verilog
if (A == 4)
begin
    B = 2;
    C = 3;
end
else if (A == 2)
begin
    B = 3;
    C = 2;
end
else if (A == 3)
    B = 1;  //No begin is used
else
    C = 1;
//example of if-else-if
//structure
```
```verilog
case (myvar)
1 : dothis = 1;
2 : dothat = 3;
3 : dosomemore = 4;
default: noneofabove = 1;
endcase
case (s)
2'b 00: out = a;
2'b 01: out = b;
2'b 10: out = c;
2'b 11: out = d;
endcase
```
```verilog
//example of case structure
```
TESTING AND VERIFICATION

Any logic once designed in Verilog can be prepared for going through certain formal simulation and testing steps. Special module or unit in any HDL which is written for simulation of any design is called stimulus or testbench. Stimulus module instantiates the targeted design or unit under test and then applies test values (stimuli) on its inputs to monitor output behavior. Before learning how to write stimulus block in Verilog, let us first understand the concept of simulation, instantiation and delays.

Simulation
Simulation simply means an imitation of genuine. In HDLs simulation means to apply input on some predefined module or design and monitor the response before the design is physically available. Simulation in any HDLs is performed at various levels that categorize simulation as Functional or behavioral simulation, Post-synthesis simulation, timing simulation, etc.

Instantiation
A module provides a template from which you can create actual objects. When a module is invoked, Verilog creates a unique object from the template. Each object has its own name, variables, parameters, and I/O interface. The process of creating objects from a module template is called instantiation, and the objects are called instances.

There are two ways to instantiate the module one by ordered method where the exact order of ports in instantiated module is followed. The second is by named association method where port mapping is done by name. Which may be helpful in case if there are large number of I/O ports and again if a subset of ports is being required to map. As an example we see below that we have two modules i.e. Half and orGate being instantiated using both approaches separately.

Example:

```verilog
module FullAdd_Order(A,B,C_IN,SUM,C_OUT);
    //Declare connection variables
    input A,B;
    input C_IN;
    output SUM;
    output C_OUT;
    wire WS, WC0, WC1;
    ...
    Half HalfAdd0(WS, WC0, A, B);
    Half HalfAdd1(SUM, WC1, C_IN, WS);
    orGate (C_OUT, WC0, WC1);
    ...
endmodule
```

```verilog
module Half(sum, c_out, a, b);
    output sum;
    output c_out;
    input a, b;
    ...
    <module internals>
    ...
endmodule
```

```verilog
module orGate(z, x, y);
    output z;
    input x, y;
    ...
    <module internals>
    ...
endmodule
```
Delays
Delay is a pause for any event to occur. In Verilog, without any explicit delays, all assignments take place in the same simulation time step but not all operations in hardware circuits are always intended to be performed in the same time step. There may be need of using timed delays to stop particular functions for a definite time. Verilog supports various delay structures in design. Event based (e.g. posedge, negedge), level triggered (e.g. at the change of values of any wire) may also be considered as delays but the most common methods of implementing explicit delays are of two types discussed below.

Regular Delays (Delay to the Statement)
Regular delay is used at the left side of the statements. Statement will never be entered until the specified delay time is not expired. Regular delay, either used with blocking assignments or non-blocking assignments, not only delays the current statement but also delays execution of all the statements that follows i.e. regular delay is accumulated for all coming statements.

Intra-Assignment Delays (Delay to Assignment)
Delay used at right side of the assignment operator, delays only the assigning of data from right to left hand side. This delay is by default applicable to the statement itself. Given below is the example for use of both regular and intra-assignment delays with non-blocking assignments.

Example:
```
module non_blocking_regular;
  reg [7:0] a, b, c, d, e;
  initial begin
    a <= 2; //Time step 0
    b <= 5; //Time step 0
    #1 c <= a; //Time step 1
    #1 d <= b; //Time step 2
    #2 b <= 4; //Time step 4
    #2 a <= 3; //Time step 6
    e <=6; //Time step 6
    #2 e <= a; //Time step 8
  //Total simulation time = 8
  //Delay accumulates
endmodule
```

Example:
```
module non_blocking_intra;
  reg [7:0] a, b, c, d, e;
  initial begin
    a <= 2; //Time step 0
    b <= 5; //Time step 0
    c <= #1 a; //Time step 1
    d <= #1 b; //Time step 1
    b <= #2 4; //Time step 2
    a <= #2 3; //Time step 2
    e <=6; //Time step 0
    e <= #2 a; //Time step 2
  //Total simulation time = 2
  //Delay applies to assignment
  //only
endmodule
```
Creating Testbench

We create testbench in verilog for simulation of any predesigned module which may be called ‘design module’, ‘UUT (Unit Under Test)’, ‘DUT (design/ Device Under Test’, ‘DUV (Device Under Verification)’ or ‘MUT (Module Under Test’. Testbench are used to apply stimuli on the UUT and verify its response. This can be observed in the figure given below:

Unlike Verilog Design Module, the testbench module does not possess its own interfaces i.e. IO ports rather it declares wires and reg signals for assigning values to UUT. An example of testbench module in Verilog instantiating a predesigned module twoToFour decoder as UUT is given below.

```verilog
module Test();
//Module Doesnot inculde Ports
// Inputs
reg En;
reg [1:0] Inp;
// Outputs
wire [3:0] Outp;
// Instantiate the UUT
//Using named association
TwoTo4Decoder uut (
  .En(En),
  .Inp(Inp),
  .Outp(Outp)
);
// Initialize Inputs
initial begin
  En = 0;
  Inp = 0;
  #5 En=1;
  #5 Inp=3;
  #5 En=0;
  Inp=1;
  #5 En=1;
  Inp=2;
end
endmodule
//TwoTo4Decoder Testbench
```

```verilog
module TwoTo4Decoder(Inp,En,Outp);
  input [1:0] Inp;
  input En;
  output [3:0] Outp;
  reg [3:0]Out;
  always @(En or Inp)
    begin
      if(~En)
        Out<=4'b1111;
      else
        begin
          case (Inp)
            2'b00: Out<=4'b1110;
            2'b01: Out<=4'b1101;
            2'b10: Out<=4'b1011;
            2'b11: Out<=4'b0111;
            endcase
        end
      assign Outp=Out;
    end
  endmodule
//TwoTo4decoder Design Module
```

```verilog
//TwoTo04Decoder Testbench
```

```
```

Appendix-B

Learning Basics of VHDL

INTRODUCTION

VHDL is a language for describing digital electronic systems. It arose out of the United States Government’s Very High Speed Integrated Circuits (VHSIC) program, initiated in 1980. In the course of this program, it became clear that there was a need for a standard language for describing the structure and function of integrated circuits (ICs). Hence the VHSIC Hardware Description Language (VHDL) was developed, and subsequently adopted as a standard by the Institute of Electrical and Electronic Engineers (IEEE) in the US. VHDL is case insensitive.

VHDL LEXICAL CONVENTIONS

Comments
Comments in VHDL start with two adjacent hyphens (‘--’) and extend to the end of the line. They have no part in the meaning of a VHDL description. C style comments are not used.

Identifiers
Identifiers in VHDL are used as reserved words and as programmer defined names. They must conform to the rule:

identifier ::= letter { [ underline ] letter_or_digit }

Note that case of letters is not considered significant, so the identifiers cat and Cat are the same. Underline characters in identifiers are significant, so This_Name and ThisName are different identifiers.

Numbers
Literal numbers may be expressed either in decimal or in a base between two and sixteen. If the literal includes a point, it represents a real number, otherwise it represents an integer. Decimal literals are defined by:

decimal_literal ::= integer [ . integer ] [ exponent ]
integer ::= digit { [ underline ] digit }
exponent ::= E [ + ] integer | E - integer

Examples
0 1 123_456_789 987E6 -- integer literals
0.0 0.5 2.718_28 12.4E-9 -- real literals

Based literal
Based literal numbers are defined by:

based_literal ::= base # based_integer [ . based_integer ] # [ exponent ]
base ::= integer
based_integer ::= extended_digit { [ underline ] extended_digit }
extended_digit ::= digit | letter

The base and the exponent are expressed in decimal. The exponent indicates the power of the base by which the literal is multiplied. The letters A to F (upper or lower case) are used as extended digits to represent 10 to 15.

Examples
2#1100_0100# 16#C4# 4#301#E1 -- the integer 196
2#1.111_1111_1111E+11 16#F.FF#E2 -- the real number 4095.0
Characters
Literal characters are formed by enclosing an ASCII character in single-quote marks.

Example
'A' '*' '';

Strings
Literal strings of characters are formed by enclosing the characters in double-quote marks. To include a double-quote mark itself in a string, a pair of double-quote marks must be put together. A string can be used as a value for an object which is an array of characters.

Examples
"A string"
"" -- empty string
"A string in a string: ""A string""." -- contains quote marks

Bit Strings
VHDL provides a convenient way of specifying literal values for arrays of type bit ('0's and '1's). The syntax is:

\[
\text{bit\_string\_literal ::= base\_specifier " bit\_value "}
\]
\[
\text{base\_specifier ::= B | O | X}
\]
\[
\text{bit\_value ::= extended\_digit { [ underline ] extended\_digit }}
\]

Base specifier B stands for binary, O for octal and X for hexadecimal.

Examples
B"1010110" -- length is 7
O"126" -- length is 9, equivalent to B"001_010_110"
X"56" -- length is 8, equivalent to B"0101_0110"

Data Objects
A data object is created by an object declaration and has a value and type associated with it. An object can be a Constant, Variable, Signal or a File. Following is a brief discussion of each class of objects excluding file type of object in this workbook.

Constant
A constant can have a single value of a given type and cannot be changed during the simulation. A constant is declared as follows:

\[
\text{constant list\_of\_name\_of\_constant: type [: initial value] ;}
\]

where the initial value is optional. Constants can be declared at the start of an architecture and can then be used anywhere within the architecture. Constants declared within a process can only be used inside that specific process.

Examples
constant RISE_FALL_TME: time := 2 ns;
constant DELAY1: time := 4 ns;
constant RISE_TIME, FALL_TIME: time:= 1 ns;
constant DATA_BUS: integer:= 16;

Variable
A variable can have a single value, as with a constant, but a variable can be updated using a variable assignment statement. The variable is updated without any delay as soon as the statement is executed. Variables must be declared inside a process (and are local to the process). The variable declaration is as follows:

\[
\text{variable list\_of\_variable\_names: type [: initial value] ;}
\]

Examples
variable VAR1: boolean :=FALSE;
variable SUM: integer range 0 to 256 :=16;
variable STS_BIT: bit\_vector (7 downto 0);
The variable SUM, in the example above, is an integer that has a range from 0 to 256 with initial value of 16 at the start of the simulation. The last example defines a bit vector of 8 elements: STS_BIT(7), STS_BIT(6),… STS_BIT(0). A variable can be updated using a variable assignment statement such as:

```
Variable_name := expression;
```

As soon as the expression is executed, the variable is updated without any delay.

**Signal**

Signals are declared outside the process using the following statement:

```
signal list_of_signal_names: type [ := initial value] ;
```

**Examples**

```
signal SUM, CARRY: std_logic;
signal CLOCK: bit;
signal TRIGGER: integer :=0;
signal DATA_BUS: bit_vector (0 to 7);
signal VALUE: integer range 0 to 100;
```

Signals are updated when their signal assignment statement is executed, after a certain delay, as illustrated below:

```
SUM <= (A xor B) after 2 ns;
```

If no delay is specified, the signal will be updated after a delta delay. One can also specify multiple waveforms using multiple events as illustrated below:

```
signal waveform : std_logic;
    waveform <= '0', '1' after 5ns, '0' after 10ns, '1' after 20 ns;
```

Signals are used in VHDL in four different modes: **in, out, inout and as buffer**

**Data Types**

Each data object has a type associated with it. The type defines the set of values that the object can have and the set of operations that are allowed on it. The notion of type is key to VHDL since it is a strongly typed language that requires each object to be of a certain type. In general one is not allowed to assign a value of one type to an object of another data type (e.g. assigning an integer to a bit type is not allowed). Few of the most commonly used data types are given below in table B.1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Range of values</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>‘0’, ‘1’</td>
<td>signal A: bit :=1;</td>
</tr>
<tr>
<td>bit_vector</td>
<td>an array with each element of type bit</td>
<td>signal INBUS: bit_vector(7 downto 0);</td>
</tr>
<tr>
<td>boolean</td>
<td>FALSE, TRUE</td>
<td>variable TEST: Boolean :=FALSE’</td>
</tr>
<tr>
<td>character</td>
<td>any legal VHDL character (see package standard);</td>
<td>variable VAL: character :=’$’;</td>
</tr>
<tr>
<td></td>
<td>printable characters must be placed between single</td>
<td></td>
</tr>
<tr>
<td></td>
<td>quotes (e.g. ‘#’)</td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>range is implementation dependent but includes at</td>
<td>constant CONST1: integer :=129;</td>
</tr>
<tr>
<td></td>
<td>least –(2^{31} – 1) to +(2^{31} – 1)</td>
<td></td>
</tr>
<tr>
<td>std_logic</td>
<td>takes on one of nine different values,</td>
<td>c_out : out std_logic;</td>
</tr>
<tr>
<td></td>
<td>including 0,1, x, and z</td>
<td></td>
</tr>
<tr>
<td>std_logic_vector</td>
<td>takes on a vector of standard logic</td>
<td>a : in std_logic_vector (31 downto 0);</td>
</tr>
<tr>
<td></td>
<td>values</td>
<td></td>
</tr>
</tbody>
</table>

**Table B.1: Data Types**
Operators
In the table B.2 are listed the types of unary and binary operators that VHDL supports. Most of them are generally seen in some traditional programming languages.

<table>
<thead>
<tr>
<th>Logical Operators</th>
<th>Relational Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator</td>
<td>Description</td>
</tr>
<tr>
<td>and</td>
<td>And</td>
</tr>
<tr>
<td>or</td>
<td>Or</td>
</tr>
<tr>
<td>nand</td>
<td>Not And</td>
</tr>
<tr>
<td>nor</td>
<td>Not Or</td>
</tr>
<tr>
<td>xor</td>
<td>Exclusive OR</td>
</tr>
<tr>
<td>xnor</td>
<td>Exclusive NOR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Adding Operators</th>
<th>Multiplying Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator</td>
<td>Description</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>&amp;</td>
<td>Concatenation</td>
</tr>
<tr>
<td>rem</td>
<td>Remainder</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unary Sign Operators</th>
<th>Shift Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator</td>
<td>Description</td>
</tr>
<tr>
<td>+</td>
<td>sll</td>
</tr>
<tr>
<td>-</td>
<td>srl</td>
</tr>
<tr>
<td>sla</td>
<td>Shift left arithmetic</td>
</tr>
<tr>
<td>sra</td>
<td>Shift right arithmetic</td>
</tr>
<tr>
<td>rol</td>
<td>Rotate left</td>
</tr>
<tr>
<td>ror</td>
<td>Rotate right</td>
</tr>
<tr>
<td>not</td>
<td>Logical negation</td>
</tr>
</tbody>
</table>

Table B.2: VHDL Operators

Operator Precedence
The operators are arranged below in order of their increasing precedence i.e. Binary logical operators have lowest while as miscellaneous operators have highest precedence.

1. Binary logical operators: and or nand nor xor xnor
2. Relational operators: = /= < <= > >=
3. Shifts operators: sll srl sla sra rol ror
4. Adding operators: + - & (concatenation)
5. Unary sign operators: + -
6. Multiplying operators: * / mod rem
7. Miscellaneous operators: not abs **

Example
Let if A="110", B="111", C="011000", and D="111011" then 

\[(A \& \text{not } B \text{ or } C \text{ ror } 2 \text{ and } D) = "110010" \] ?

The operators will be applied in the following order: not, & , ror, or, and, =

not B = '000' --bit-by-bit complement
A & not B = "110000" --concatenation
C ror 2 = "000110" --rotate right 2 places
(A & not B) or (C ror 2) = "110110" --bit-by-bit or
(A & not B or C ror 2) and D = "110010" --bit-by-bit and
\[(A \& \neg B \text{ or } C \text{ ror } 2 \text{ and } D) = \text{“110010”}\]=TRUE --This is the result

**Template for a VHDL file**

VHDL files are composed of Entity-Architecture pairs. The Entity portion of the file is analogous to a symbol for the design. It describes all of the external connections to the design. The Architecture portion of the file is analogous to the circuit diagram of the design. It defines the implementation of the design. Every VHDL design will have the following general appearance:

Figure B.1: VHDL file template

```vhdl
-- Library and package declarations
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_arith.all;

-- The following is the Entity portion
ENTITY name IS
  -- Keywords are capitalized(optional)
  PORT( list of all external connections);
END name;

-- The following is the Architecture portion
ARCHITECTURE anyname OF name IS
  -- This is the declarative part of the Architecture
  -- Declare signals, enumeration types, constants here
BEGIN
  -- This is where the implementation is described. Concurrent signal
  -- assignments go here. Therefore this is called the concurrent part.
  -- statements’ order does not matter since they all are executed concurrently.
  PROCESS
    -- The architecture may contain zero or more processes.
    -- This is the declarative part of the process.
    -- Variables used in the process are declared here.
    BEGIN
      -- Beginning of the process implementation.
      -- The process is implemented using sequential statements.
      -- For example, FOR LOOP, IF-THEN-ELSE, CASE
  END PROCESS;
END anyname;
```
Example

Write a VHDL module for half adder for which block diagram and logic diagram of a half adder are shown below in figure B.2. Use dataflow abstraction level.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Halfadder is
port (a,b : in std_logic;
      sum, c_out : out std_logic;
end Halfadder;

architecture dataflow of Halfadder is
begin
  sum <= (a xor b) after 5 ns;
  c_out <= (a and b) after 5 ns;
end dataflow;

--Half Adder in VHDL
```

Note that in the above example:
1. As in Verilog, the <= operator denotes a concurrent signal assignment (CSA). Moreover, evaluating a block of CSA’s is done similarly as in Verilog. Namely, the signals and variables on the right side of the CSA’s are first sampled at some time t, followed by an update of the signals and variables on the left side using the sampled values. Thus, the updates are performed concurrently.
2. The xor and and bit operators are written explicitly. The same holds for the other logical operators: or, nand, nor, xnor.
3. The after keyword is used to delay a CSA. In this case, the delays are 5 ns. In general, the delay can be any arithmetic expression.
LEVELS OF ABSTRACTION
VHDL in the same way as Verilog supports same four levels of abstractions. For theoretical concepts refer to the appendix-A. Here our focus will be on examples of dataflow, behavioral, and structural levels of abstraction. Single entity in VHDL may have different architectures like one for behavioral another for dataflow etc.

Example: Dataflow Modeling
See the above example of Half adder. In this module, entity HalfAdder is implemented using dataflow level of abstraction. This can also be listed as below:

```
architecture dataflow of Halfadder is
begin
  sum <= (not a and b) or (a and not b);
  c_out <= a and b;
  -- delays are excluded just for simplicity
end dataflow;
-- Remember: not operator have higher precedence than and operator
```

Example: Behavioral Modeling
-- Same entity Halfadder is used
```vhdl
architecture behavior of Halfadder is
begin
  hadd: process (a, b)
  begin
    if a='1' then
      sum <= not b;
      c_out <= b;
    else
      sum <= b;
      c_out <= '0';
    end if;
    -- delays are excluded just for simplicity
  end process hadd;
end behavior;
```

Example: Structural Modeling
-- Same entity Halfadder is used again
```vhdl
architecture structure of Halfadder is
begin
  u1: entity xor_2 port map (i1 => a, i2 => b, o1 => sum);
  u2: entity and_2 port map (i1 => a, i2 => b, o1 => c_out);
  -- Assume xor_2, and_2 are two existing entities
  -- ‘=>' is an association symbol used for instantiation
end structure;
```

Instantiation
Similar to Verilog HDL, VHDL uses two methods for instantiation called named and positional or ordered association.

Examples
-- Named association
```vhdl
u1: entity xor_2 port map (i1 => a, i2 => b, o1 => sum);
-- Ordered/ Positional association (assuming order in original entity is known)
```
u1: entity xor_2 port map (a, b, sum);

**CONDITIONAL STRUCTURES**

VHL supports a number of conditional structures. Mainly it includes the following:

1. Conditional CSAs
2. Selected Signal Assignments
3. If Statement
4. CASE statement

**Conditional CSAs**
Also called when-else statement, allows a signal to be assigned a value based on a set of conditions. The conditions are expressions involving relational operators. The basic syntax is given below:

```
conditional_signal_assignment ::= 
target <= { value_expression when condition else }
value_expression when condition;
```

**Example**
```
target <= in0 after 5 ns when s0 = '0' and s1 = '0' else
in1 after 5 ns when s0 = '1' and s1 = '0' else
in2 after 5 ns when s0 = '0' and s1 = '1' else
in3 after 5 ns when s0 = '1' and s1 = '1' else
"00000000000000" after 5 ns;
```

-- Note: Last value_expression is not associated with when condition

**Selected Signal Assignments**
Also called with-select-when statement, allows one of several possible values to be assigned to a signal based on a select expression. The basic syntax is given below:

```
select_signal_assignment ::= 
with expression select
        target <= { value_expression when choices, }
        value_expression when choices;
```

**Example**
```
with province select
capital <= Karachi when “Sindh”,
            Lahore when "Punjab",
            Quetta when "Balochistan",
            Peshawar when “Pakhtoonkhuwah”,
            Nothing when others;
```

**If Statement**
The if statement allows selection of statements to execute depending on one or more conditions. If statement is used within a process. The syntax is:

```
if_statement ::= 
if condition then
    sequence_of_statements
elsif condition then
    sequence_of_statements
else
    sequence_of_statements
end if;
```
Example

output: process(R,S,Clk) is
begin
  if (R = '0') then
    Q <= '0' after 5 ns;
    Qbar <= '1' after 5 ns;
  elsif S = '0' then
    Q <= '1' after 5 ns;
    Qbar <= '0' after 5 ns;
  elsif(rising_edge(Clk)) then
    Q <= D after 5 ns;
    Qbar <= (not D) after 5 ns;
  end if;
end process output;

Case Statement
The case statement allows selection of statements to execute depending on the value of a selection expression. Like if statement case statement is also used within a process. The syntax is:

case_statement ::= case expression is
  when choices => sequence_of_statements
  { when choices => sequence_of_statements }
end case;

choices ::= choice { | choice }

Example

case instruction is
  when load_accum =>
    accum <= data;
  when store_accum =>
    data_out <= accum;
  when load|store =>
    -- Multiple choices
    process_IO(addr);
  when others =>
    -- Function call
    process_error(instruction);
end case;

TYPE QUALIFICATION
Type qualification is one of the method used to convert from one data type to another allowable data type.

Example

std_logic_vector'(a, b, c);
-- Assume a,b and c are 1 bit signals of std_logic data type
-- The above statement concatenates values of three signals and then
-- changes data type to std_logic_vector
real'(average(points));
-- Converts value returned by average function to real data type
PROCESS
A process is an architectural construct that allows for the use of conventional programming language concepts, such as variables, loops, and other control-flow statements. There exist two different ways to define a process.
1. With a sensitivity list: As with Verilog cyclic behaviors, the process is executed only when an event occurs on one or more signals in the sensitivity list.
2. Without a sensitivity list. In this case a process is inherently cyclic, and therefore continues repeating indefinitely.

General syntax for a process is given below:

```
[label:] process (sensitive signals)
 variable declarations
 constant declarations
...
 begin
 statements
 ...
 end process;
```

---

Label is an optional

Example(s)

```
sum_proc: process(x,y)
 begin
  if(x=y) then sum <= '0' after 5 ns;
  else sum <= (x or y) after 5 ns;
  end if:
 end process sum_proc;
clk_process: process(clk)
 begin
  if(rising_edge(clk)) then
    xr <= not xr;
  end if:
 end process clk_process
```
Digital circuits may be sequential or combinational. When designing these circuits in an HDL, some rules must be followed to get good synthesis output.

**COMBINATIONAL LOGIC**

Combinational logic is used in computer circuits to do Boolean algebra on input signals and on stored data. Logic is combinational if outputs at a specified time are a function only of the inputs at that time. In Verilog, combinational circuits can be designed using assign and always blocks. In VHDL, combinational logic is implemented with concurrent signal assignment statements or with process statements that describe pure combinational behavior. Both concurrent signal statements and process statements should be placed in an Architecture Body, as shown below:

```vhdl
ARCHITECTURE a OF and-gate IS
BEGIN
  <concurrent signal assignments>
  <process statements>
  <other concurrent statements>
END a;
```

Below are some examples to compare and contrast Verilog and VHDL coding techniques for combinational circuits.

**4-bit Full Adder**

A 4-bit full adder performs arithmetic sum of two 4-bit numbers and a previous carry, to produce a 4-bit resulting sum and a carry bit. The example below shows how a 4-bit full adder can be designed using the dataflow description in Verilog and VHDL.

**Example C.1: Design of 4-bit Full Adder in Verilog & VHDL**

Verilog:

```verilog
module fulladd4
  (sum, c_out, a, b, c_in);
Output [3:0] sum;
Output c_out;
input [3:0] a, b;
input c_in;
assign {c_out, sum} = a + b + c_in;
endmodule
// Full Adder in Verilog
```

VHDL:

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

entity adder4 is
  port (a : in std_logic_vector(3 downto 0);
b : in std_logic_vector(3 downto 0);
c_in : in std_logic;
sum : out std_logic_vector(3 downto 0);
c_out : out std_logic;
end adder4;

architecture rtl of adder4 is
begin
  sum <= (a xor b) xor c_in;
c_out <= (a and b) or (c_in and a) or (c_in and b);
end rtl;
-- Full Adder in VHDL
```

70
2-to-4 Decoder
A 2-to-4 decoder takes as input a 2-bit binary number and produces an output on one of the 4 output lines. Below we present an example to show how a 2-to-4 decoder can be modeled using behavioral coding in Verilog and VHDL.

**Example C.2: Design of 2-to-4 Decoder in Verilog & VHDL**

```verilog
library IEEE;
use IEEE.std_logic_1164.all;
entity Decoder2to4 is
port(A,B, en : in std_logic;
    Y3, Y2, Y1, Y0: out std_logic);
end Decoder2To4;
architecture DecIns of Decoder2To4 is
begin
  case y:
    process
    begin
      Y0<='1'; Y1<='1'; Y2<='1'; Y3<='1';
      case std_logic_vector'(A, B, en);
      when "000" => Y0 <= '0';
      when "000" => Y0 <= '0';
      when "000" => Y0 <= '0';
      when "000" => Y0 <= '0';
      end case
    end process;
  end case
end DecIns;
```

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
entity decoder2to4 is
  port(Y3, Y2, Y1, Y0: out std_logic;
       A, B, en: in std_logic);
end decoder2to4;
architecture DecIns of decoder2to4 is
begin
  {Y0,Y1,Y2,Y3} = 4'b1111;
  case ( {en,A,B} )
  when 3'b000 => Y0 <= '0';
  when 3'b001 => Y1 <= '0';
  when 3'b010 => Y2 <= '0';
  when 3'b011 => Y3 <= '0';
  endcase
end//always
endmodule
//2 to 4 Decoder in Verilog
```

**2 to 4 Decoder in VHDL**

```verilog
module mux4to1 ( select, A, B, d, q );
input select, A, B;
input[3:0] d;
output q;
reg q;
assign q = ~select & ( (~A & ~B & d[0]) |(~A & B & d[1]) |(A & ~B & d[2]) |(A & B & d[3]));
end
endmodule
//4 to 1 MUX in Verilog
```

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
entity MUX41 is
port --define inputs and outputs
  ( select, A, B: in std_logic;
    d: in std_logic_vector(3 downto 0);
    q: out std_logic);
end
architecture logic of MUX41 is
begin
  q <= not select and ( not a and not b and d(0))
    or (not a and b and d(1))
    or (a and not b and d(2))
    or (a and b and d(3)));
end MUX41;
```

4-to-1 Multiplexer
A multiplexer connects data from one of the $n$ sources to the output. A number of select inputs determine which data source is connected to the output. In Verilog, the logic equations of the multiplexer can be modeled using assignment statements, as shown below.

**Example C.3: Design of 4-to-1 MUX in Verilog & VHDL**

```verilog
module decoder2to4 ( Y3, Y2, Y1, Y0, A, B, en);
output Y3, Y2, Y1, Y0;
input A, B;
input en;
reg Y3, Y2, Y1, Y0;
always @(A or B or en)
begin
  {Y0,Y1,Y2,Y3} = 4'b1111;
end
endmodule
```

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
entity MUX41 is
port --define inputs and outputs
  ( select, A, B: in std_logic;
    d: in std_logic_vector(3 downto 0);
    q: out std_logic);
end
architecture logic of MUX41 is
begin
  q <= not select and ( not a and not b and d(0))
    or (not a and b and d(1))
    or (a and not b and d(2))
    or (a and b and d(3)));
end MUX41;
```

```verilog
module mux4to1 ( select, A, B, d, q );
input select, A, B;
input[3:0] d;
output q;
reg q;
assign q = ~select & ( (~A & ~B & d[0]) |(~A & B & d[1]) |(A & ~B & d[2]) |(A & B & d[3]));
end
endmodule
//4 to 1 MUX in Verilog
```

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
entity MUX41 is
port --define inputs and outputs
  ( select, A, B: in std_logic;
    d: in std_logic_vector(3 downto 0);
    q: out std_logic);
end
architecture logic of MUX41 is
begin
  q <= not select and ( not a and not b and d(0))
    or (not a and b and d(1))
    or (a and not b and d(2))
    or (a and b and d(3)));
end MUX41;
```

```verilog
library IEEE;
use IEEE.std_logic_1164.all;
entity MUX41 is
port --define inputs and outputs
  ( select, A, B: in std_logic;
    d: in std_logic_vector(3 downto 0);
    q: out std_logic);
end
architecture logic of MUX41 is
begin
  q <= not select and ( not a and not b and d(0))
    or (not a and b and d(1))
    or (a and not b and d(2))
    or (a and b and d(3)));
end MUX41;
```

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
entity MUX41 is
port --define inputs and outputs
  ( select, A, B: in std_logic;
    d: in std_logic_vector(3 downto 0);
    q: out std_logic);
end
architecture logic of MUX41 is
begin
  q <= not select and ( not a and not b and d(0))
    or (not a and b and d(1))
    or (a and not b and d(2))
    or (a and b and d(3)));
end MUX41;
```
SEQUENTIAL LOGIC
In sequential logic, outputs specified at a given time are a function of the inputs at that time and at all preceding times. Hence, sequential logic is used to construct computer memory, delay and storage elements and finite state machines. When using Verilog, sequential logic circuits are modeled using edge sensitive elements in the sensitivity list of always block. Non-blocking assignments are used for sequential circuits. In VHDL, sequential logic is implemented using process statements. These statements direct the compiler to create logic circuitry that is controlled by the Process Statement’s clock signal.

D Flip Flop
The flip flop is the basic building block of any digital circuit. In the example below, we consider Verilog and VHDL code for modeling the classic D flip flop in which the output Q follows the input D as long as the clock is high.

Example C.4: Design of Asynchronous D-Flip Flop in Verilog & VHDL

Verilog Code:

```verilog
module dff_asynch( D, clk, reset, set,
Q, Qbar);

input D, clk, reset, set;
output Q, Qbar;
reg clk, reset, set, D;
reg Q, Qbar;

always@ (posedge clk or posedge reset or posedge set)
if (reset) begin
  Q <= 0;
  Qbar <= 1;
end else if (set) begin
  Q <= 1;
  Qbar <= 0;
else begin
  Q <= D;
  Qbar <= ~D;
end
endmodule
```

VHDL Code:

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

entity dff_asyn is
port( D, Clk, Reset, Set: IN STD_LOGIC;
Q, Qbar: OUT STD_LOGIC);
end dff_asynch;

architecture behavioral of dff_asyn is
begin
pr: process(Clk, Reset, Set)
begin
  if Reset = '1' then
    Q <= '0';
    Qbar <= '1';
  else if Set = '1' then
    Q <= '1';
    Qbar <= '0';
  else if (Clk'event) and (Clk='1') then
    if Set = '1' then
      Q <= '1';
      Qbar <= '0';
    else
      Q <= D;
      Qbar <= not D;
    end if;
  end if;
end process;
end behavioral;
```

-- Asynchronous D-Flip Flop in VHDL
3 bit Counter
We now compare Verilog and VHDL code for designing a synchronous 3-bit counter.

Example C.5: Design of 3-bit Synchronous Counter in Verilog & VHDL

```verilog
module counter3bit(clk,reset,out);
input clk,reset;
output [2:0] q;
reg [2:0] q;
always @(posedge clk) begin
if (reset == 1'b1) q <= 3'b000;
else q <= q+1'b1;
end
endmodule
```

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity Count3bit is
port(
  clk : in std_logic;
  reset: in std_logic;
  q : out std_logic_vector(2 downto 0)
);
architecture behavioral of Count3bit is
begin
pr: process(clk)
begin
if(rising_edge(clk)) then
  if reset = '1' then
    q <= '0';
  else
    q <= q+1;
  end if;
end if;
end process;
end behavioral;
```

-- 3-bit counter in VHDL