**DAY 17 ASSIGNMENT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **05-06-2020** | **Name:** | **Ashish Shanbhag** |
| **Course:** | **DIGITAL DESIGN USING HDL** | **USN:** | **4AL16EC008** |
| **Topic:** | 1. **Verilog Tutorials and practice programs** 2. **Building/ Demo projects using FPGA** | **Semester & Section:** | **8th A** |
| **Github Repository:** | **Ashish Shanbhag** |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS**    **Verilog Tutorials and practice programs**  Verilog is a Hardware DEscription Language (HDL). A hardware description Language is a language used to describe a digital system, for example, a network switch, a microprocessor or a memory or a simple flip−flop. This just means that, by using a HDL one can describe any hardware (digital) at any level.  **Design Styles**  Verilog like any other hardware description language, permits the designers to design a design in either Bottom−up or Top−down methodology.  **Bottom−Up Design**  The traditional method of electronic design is bottom−up. Each design is performed at the gate−level using the standard gates. With increasing complexity of new designs this approach is nearly impossible to maintain. New systems consist of ASIC or microprocessors with a complexity of thousands of transistors. These traditional bottom−up designs have to give way to new structural, hierarchical design methods. Without these new design practices it would be impossible to handle the new complexity.  **Top−Down Design**  The desired design−style of all designers is the top−down design. A real top−down design allows early testing, easy change of different technologies, a structured system design and offers many other advantages. But it is very difficult to follow a pure top−down design. Due to this fact most designs are mix of both the methods, implementing some key elements of both design styles.  **Abstraction Levels of Verilog**  **Verilog supports a design at many different levels of abstraction. Three of them are very important:**  **• Behavioral level**   * **Register−Transfer Level**   **• Gate Level**  **Behavioral level**  This level describes a system by concurrent algorithms (Behavioral). Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other. Functions, Tasks and Always blocks are the main elements. There is no regard to the structural realization of the design.  **Register−Transfer Level**  Designs using the Register−Transfer Level specify the characteristics of a circuit by operations and the transfer of data between the registers. An explicit clock is used. RTL design contains exact timing possibility, operations are scheduled to occur at certain times. Modern definition of a RTL code is "Any code that is synthesizable is called RTL code".  **Gate Level**  Within the logic level the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (`0', `1', `X',`Z`). The usable operations are predefined logic primitives (AND, OR, NOT etc gates). Using gate level modeling might not be a good idea for any level of logic design. Gate level code is generated by tools like synthesis tools and this netlist is used for gate level simulation and for backend.  **Example**  module gates();  wire out0;  wire out1;  wire out2;  reg in1,in2,in3,in4;  not U1(out0,in1);  and U2(out1,in1,in2,in3,in4);  xor U3(out2,in1,in2,in3);  initial begin  1 $monitor( "in1 = %b in2 = %b in3 = %b in4 = %b out0 = %b out1 = %b out2 = %b"  ,in1,in2,in3,in4,out0,out1,out2);  in1 = 0;  in2 = 0;  in3 = 0;  in4 = 0;  #1 in1 = 1;  #1 in2 = 1;  #1 in4 = 1;  #1 $finish;  end  endmodule  **OUTPUT**  in1 = 0 in2 = 0 in3 = 0 in4 = 0 out0 = 1 out1 = 0 out2 = 0  in1 = 1 in2 = 0 in3 = 0 in4 = 0 out0 = 0 out1 = 0 out2 = 1  in1 = 1 in2 = 1 in3 = 0 in4 = 0 out0 = 0 out1 = 0 out2 = 0  in1 = 1 in2 = 1 in3 = 1 in4 = 0 out0 = 0 out1 = 0 out2 = 1  in1 = 1 in2 = 1 in3 = 1 in4 = 1 out0 = 0 out1 = 1 out2 = 1  module transmission\_gates();  reg data\_enable\_low, in;  wire data\_bus, out1, out2;  bufif0 U1(data\_bus,in, data\_enable\_low);  buf U2(out1,in);  not U3(out2,in);  initial begin  $monitor( "in = %b data\_enable\_low = %b out1 = %b out2 = %b" ,in,data\_enable\_low, out1, out2);  data\_enable\_low = 0;  in = 0;  #4 data\_enable\_low = 1;  #8 $finish;  end  always #2 in = ~in;  endmodule  **OUTPUT**  in = 0 data\_enable\_low = 0 out1 = 0 out2 = 1  in = 1 data\_enable\_low = 0 out1 = 1 out2 = 0  in = 0 data\_enable\_low = 1 out1 = 0 out2 = 1  in = 1 data\_enable\_low = 1 out1 = 1 out2 = 0  in = 0 data\_enable\_low = 1 out1 = 0 out2 = 1  in = 1 data\_enable\_low = 1 out1 = 1 out2 = 0 |

**DAY 17 ASSIGNMENT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **05-06-2020** | **Name:** | **Ashish Shanbhag** |
| **Course:** | **PYTHON** | **USN:** | **4AL16EC008** |
| **Topic:** | **Build a Data Collector Web App with PostGreSQL and Flask** | **Semester & Section:** | **8th A** |
| **Github Repository:** | **Ashish Shanbhag** |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS**      **Data Collector Web App with PostGreSQL and Flask**  [Flask](http://flask.pocoo.org/) is a Python web developpement framework to build web applications. It comes with [jinja2](http://jinja.pocoo.org/docs/2.9/), a [templating language](https://en.wikipedia.org/wiki/Web_template_system) for Python, and [Werkzeug](http://werkzeug.pocoo.org/), a [WSGI](https://www.python.org/dev/peps/pep-0333/) utility module. [PostgreSQL](https://www.postgresql.org/) is an open source [relational database system](https://en.wikipedia.org/wiki/Relational_database) which, as its name suggests, uses SQL. [SQLAlchemy](http://www.sqlalchemy.org/) is an Object Relational Mapper ([ORM](https://en.wikipedia.org/wiki/Object-relational_mapping)), it is a layer between object oriented Python and the database schema of Postgres.  [Alembic](http://alembic.zzzcomputing.com/en/latest/) is a useful module to manage migrations with SQLAlchemy in Python. Migrations occur when one wants to change the database schema linked to the application, like adding a table or removing a column from a table. It can also be used to write or delete data in a table. Alembic enables developers not to manually upgrade their database and to easily revert any change: migrations go up and down. It is also useful to recreate databases from scratch, by following the migration flow.  Create an app.py file which will define and run the application. It is the entry point of the application. With Flask, it is as easy as importing the Flask class and initialize an instance with:  app = Flask(\_\_name\_\_)   * Add:   if \_\_name\_\_ = '\_\_main\_\_':  app.run()  in app.py file and then enter python app.py in a terminal to get your app running. Easy, but it does not do many things yet...   * So far, if you want something else than an error 404 when accessing the application, create the first route which will return Hello World! at the root of the application. To do so, add the following piece of code after the definition of the application instance.   @app.route('/')  def main():  return 'Hello World!'   * Set the application in debug mode so that the server is reloaded on any code change and provides detailed error messages, otherwise it should be restarted manually. In app.py, before app.run():   app.config['DEBUG'] = True   * Initialize a database object from Flask-Alchemy with db = SQLAlchemy() to control the SQLAlchemy integration to the Flask applications. You might put it directly in the app.py or in another file usually called models.py.   from flask\_sqlalchemy import SQLAlchemy  db = SQLAlchemy()  # define your models classes hereafter   * Configure Flask by providing the PostgreSQL URI so that the app is able to connect to the database, through : app.config['SQLALCHEMY\_DATABASE\_URI'] = 'postgresql://DB\_USER:PASSWORD@HOST/DATABASE' where you have to replace all the parameters in capital letters (after postgresq://). |