**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | 2 June 2020 | **Name:** | Anupama J S |
| **Course:** | Digital design using HDL | **USN:** | 4AL16EC005 |
| **Topic:** | FPGA Basics: Architecture,  Applications and Uses  Verilog HDL Basics by  Intel  Verilog Testbench code to  verify the design under  test  (DUT) | **Semester & Section:** | 8th sem “A”section |
| **Github Repository:** | AnupamaJS |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| **C:\Users\User\Desktop\WhatsApp Image 2020-06-02 at 7.53.53 PM.jpeg**  The field-programmable gate array (FPGA) is an integrated circuit that consists of internal hardware blocks with user-programmable interconnects to customize operation for a specific application.  **What is FPGA?**  The field-programmable gate array (FPGA) is an integrated circuit that consists of internal hardware blocks with user-programmable interconnects to customize operation for a specific application. The interconnects can readily be reprogrammed, allowing an FPGA to accommodate changes to a design or even support a new application during the lifetime of the part.  The FPGA has its roots in earlier devices such as programmable read-only memories (PROMs) and programmable logic devices (PLDs). These devices could be programmed either at the factory or in the field, but they used fuse technology (hence, the expression “burning a PROM”) and could not be changed once programmed. In contrast, FPGA stores its configuration information in a re-programmable medium such as static RAM (SRAM) or flash memory. FPGA manufacturers include Intel, Xilinx, Lattice Semiconductor, Microchip Technology and Microsemi.  Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs), which are custom manufactured for specific design tasks. Although one-time programmable (OTP) FPGAs are available, the dominant types are SRAM based which can be reprogrammed as the design evolves. FPGA Applications Due to their programmable nature, FPGAs are an ideal fit for many different markets. As the industry leader, Xilinx provides comprehensive solutions consisting of FPGA devices, advanced software, and configurable, ready-to-use IP cores for markets and applications such as:   * [Aerospace & Defense](https://www.xilinx.com/applications/aerospace-and-defense.html) - Radiation-tolerant FPGAs along with intellectual property for image processing, waveform generation, and partial reconfiguration for SDRs. * [ASIC Prototyping](https://www.xilinx.com/applications/emulation-prototyping.html) - ASIC prototyping with FPGAs enables fast and accurate SoC system modeling and verification of embedded software * [Audio](https://www.xilinx.com/applications/audio.html) - Xilinx FPGAs and targeted design platforms enable higher degrees of flexibility, faster time-to-market, and lower overall non-recurring engineering costs (NRE) for a wide range of audio, communications, and multimedia applications. * [Automotive](https://www.xilinx.com/applications/automotive.html) - Automotive silicon and IP solutions for gateway and driver assistance systems, comfort, convenience, and in-vehicle infotainment. - [Learn how Xilinx FPGA's enable Automotive Systems](https://www.xilinx.com/training/automotive-fpga-training.htm) * [Broadcast & Pro AV](https://www.xilinx.com/applications/broadcast.html) - Adapt to changing requirements faster and lengthen product life cycles with Broadcast Targeted Design Platforms and solutions for high-end professional broadcast systems. * [Consumer Electronics](https://www.xilinx.com/applications/consumer-electronics.html) - Cost-effective solutions enabling next generation, full-featured consumer applications, such as converged handsets, digital flat panel displays, information appliances, home networking, and residential set top boxes. * [Data Center](https://www.xilinx.com/applications/data-center.html) - Designed for high-bandwidth, low-latency servers, networking, and storage applications to bring higher value into cloud deployments. * [High Performance Computing and Data Storage](https://www.xilinx.com/applications/high-performance-computing.html) - Solutions for Network Attached Storage (NAS), Storage Area Network (SAN), servers, and storage appliances. * [Industrial](https://www.xilinx.com/applications/industrial.html) - Xilinx FPGAs and targeted design platforms for Industrial, Scientific and Medical (ISM) enable higher degrees of flexibility, faster time-to-market, and lower overall non-recurring engineering costs (NRE) for a wide range of applications such as industrial imaging and surveillance, industrial automation, and medical imaging equipment. * [Medical](https://www.xilinx.com/applications/medical.html) - For diagnostic, monitoring, and therapy applications, the Virtex FPGA and Spartan® FPGA families can be used to meet a range of processing, display, and I/O interface requirements. * Security - Xilinx offers solutions that meet the evolving needs of security applications, from access control to surveillance and safety systems. * [Video & Image Processing](https://www.xilinx.com/applications/video-and-imaging.html) - Xilinx FPGAs and targeted design platforms enable higher degrees of flexibility, faster time-to-market, and lower overall non-recurring engineering costs (NRE) for a wide range of video and imaging applications. * [Wired Communications](https://www.xilinx.com/applications/wired-communications.html) - End-to-end solutions for the Reprogrammable Networking Linecard Packet Processing, Framer/MAC, serial backplanes, and more * [Wireless Communications](https://www.xilinx.com/applications/wireless-communications.html) - RF, base band, connectivity, transport and networking solutions for wireless equipment, addressing standards such as WCDMA, HSDPA, WiMAX and others.   **FPGA architecture**  C:\Users\User\Downloads\WhatsApp Image 2020-06-02 at 8.09.12 PM.jpeg  A basic FPGA architecture consists of thousands of fundamental elements called configurable logic blocks (CLBs) surrounded by a system of programmable interconnects, called a fabric, that routes signals between CLBs. Input/output (I/O) blocks interface between the FPGA and external devices.  Depending on the manufacturer, the CLB may also be referred to as a logic block (LB), a logic element (LE) or a logic cell (LC).  An individual CLB is made up of several logic blocks. A lookup table (LUT) is a characteristic feature of an FPGA. An LUT stores a predefined list of logic outputs for any combination of inputs: LUTs with four to six input bits are widely used. Standard logic functions such as multiplexers (mux), full adders (FAs) and flip-flops are also common. The number and arrangement of components in the CLB varies by device; the simplified example in Figure 2 contains two three-input LUTs (1), an FA (3) and a D-type flip-flop (5), plus a standard mux (2) and two muxes, (4) and (6), that are configured during FPGA programming. This simplified CLB has two modes of operation. In normal mode, the LUTs are combined with Mux 2 to form a four-input LUT; in arithmetic mode, the LUT outputs are fed as inputs to the FA together with a carry input from another CLB. Mux 4 selects between the FA output or the LUT output. Mux 6 determines whether the operation is asynchronous or synchronized to the FPGA clock via the D flip-flop. Current-generation FPGAs include more complex CLBs capable of multiple operations with a single block; CLBs can combine for more complex operations such as multipliers, registers, counters and even digital signal processing (DSP) functions.  C:\Users\User\Downloads\WhatsApp Image 2020-06-02 at 8.22.24 PM (4).jpeg  **C:\Users\User\Downloads\WhatsApp Image 2020-06-02 at 8.22.24 PM (3).jpeg**  **HDL**  In computer engineering, a hardware description language (HDL) is a specialized computer language used to describe the structure and behavior of electronic circuits, and most commonly, digital logic circuits. A hardware description language enables a precise, formal description of an electronic circuit that allows for the automated analysis and simulation of an electronic circuit. It also allows for the synthesis of an HDL description into a netlist (a specification of physical electronic components and how they are connected together), which can then be placed and routed to produce the set of masks used to create an integrated circuit. A hardware description language looks much like a programming language such as C or ALGOL; it is a textual description consisting of expressions, statements and control structures. One important difference between most programming languages and HDLs is that HDLs explicitly include the notion of time. HDLs form an integral part of electronic design automation (EDA) systems, especially for complex circuits, such as application-specific integrated circuits, microprocessors, and programmable logic devices. Verilog Introduction Verilog is a HARDWARE DESCRIPTION LANGUAGE (HDL). It is a language used for describing a digital system like a network switch or a microprocessor or a memory or a flip−flop. It means, by using a HDL we can describe any digital hardware at any level. Designs, which are described in HDL are independent of technology, very easy for designing and debugging, and are normally more useful than schematics, particularly for large circuits.  Verilog supports a design at many levels of abstraction. The major three are −   * Behavioral level * Register-transfer level * Gate level   **Behavioral level**  This level describes a system by concurrent algorithms (Behavioural). Every algorithm is sequential, which means it consists of a set of instructions that are executed one by one. Functions, tasks and 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 using operations and the transfer of data between the registers. Modern definition of an RTL code is "Any code that is synthesizable is called RTL code".  **Gate Level**  Within the logical 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 (basic gates). Gate level modelling may not be a right idea for logic design. Gate level code is generated using tools like synthesis tools and his netlist is used for gate level simulation and for backend.  C:\Users\User\Downloads\WhatsApp Image 2020-06-02 at 8.22.24 PM (2).jpeg  C:\Users\User\Downloads\WhatsApp Image 2020-06-02 at 8.22.24 PM (1).jpeg  **Test benches** are used to simulate your design without the need of any physical hardware. The biggest benefit of this is that you can actually inspect every signal that is in your design. This definitely can be a time saver when your alternatives are staring at the code, or loading it onto the FPGA and probing the few signals brought out to the external pins. However, you don't get this all for free. Before you can simulate your design you must first write a **test bench**. What is a Test Bench What exactly is a **test bench**? A test bench is actually just another Verilog file! However, the Verilog you write in a test bench is not quite the same as the Verilog you write in your designs. This is because all the Verilog you plan on using in your hardware design **must be**synthesizable, meaning it has a hardware equivalent. The Verilog you write in a test bench does not need to be synthesizable because you will only ever simulate it  **TODAY’S TASK :**  I**mplement a 4:1 MUX and write the test bench code to verify the module**  STRUCTURAL:  module and\_gate(output a, input b, c, d);  assign a = b & c & d;  endmodule  module not\_gate(output f, input e);  assign e = ~ f;  endmodule  module or\_gate(output l, input m, n, o, p);  assign l = m | n | o | p;  endmodule  module m41(out, a, b, c, d, s0, s1);  output out;  input a, b, c, d, s0, s1;  wire s0bar, s1bar, T1, T2, T3;  not\_gate u1(s1bar, s1);  not\_gate u2(s0bar, s0);  and\_gate u3(T1, a, s0bar, s1bar);  and\_gate u4(T2, b, s0, s1bar);  and\_gate u5(T3, c, s0bar, s1);  and\_gate u6(T4, d, s0, s1);  or\_gate u7(out, T1, T2, T3, T4);  endmodul  **TESTBENCH:**  module top;  wire out;  reg a;  reg b;  reg c;  reg d;  reg s0, s1;  m41 name(.out(out), .a(a), .b(b), .c(c), .d(d), .s0(s0), .s1(s1));  initial  begin  a=1'b0; b=1'b0; c=1'b0; d=1'b0;  s0=1'b0; s1=1'b0;  #500 $finish;  end  always #40 a=~a;  always #20 b=~b;  always #10 c=~c;  always #5 d=~d;  always #80 s0=~s0;  always #160 s1=~s1;  always@(a or b or c or d or s0 or s1)  $monitor("At time = %t, Output = %d", $time, out);  endmodule; |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | 2 June 2020 | **Name:** | Anupama J S |
| **Course:** | Python | **USN:** | 4AL16EC005 |
| **Topic:** | 1. Interactive data visualization with bokeh, webscraping with python beautiful soup. | **Semester & Section:** | 8th sem “A”section |
| **Github Repository:** | AnupamaJS |  |  |
| **AFTERNOON SESSION DETAILS** | | | |
| From Data to VisualizationBuilding a visualization with Bokeh involves the following steps:Prepare the dataDetermine where the visualization will be renderedSet up the figure(s)Connect to and draw your dataOrganize the layoutPreview and save your beautiful data creationLet’s explore each step in more detail.Prepare the DataAny good data visualization starts with—you guessed it—data. If you need a quick refresher on handling data in Python, definitely check out the growing number of excellent Real Python tutorials on the subject.This step commonly involves data handling libraries like Pandas and Numpy and is all about taking the required steps to transform it into a form that is best suited for your intended visualization.Determine Where the Visualization Will Be RenderedAt this step, you’ll determine how you want to generate and ultimately view your visualization. In this tutorial, you’ll learn about two common options that Bokeh provides: generating a static HTML file and rendering your visualization inline in a Jupyter Notebook.Set up the Figure(s)From here, you’ll assemble your figure, preparing the canvas for your visualization. In this step, you can customize everything from the titles to the tick marks. You can also set up a suite of tools that can enable various user interactions with your visualization.Connect to and Draw Your DataNext, you’ll use Bokeh’s multitude of renderers to give shape to your data. Here, you have the flexibility to draw your data from scratch using the many available marker and shape options, all of which are easily customizable. This functionality gives you incredible creative freedom in representing your data.Additionally, Bokeh has some built-in functionality for building things like stacked bar charts and plenty of examples for creating more advanced visualizations like network graphs and maps.Organize the LayoutIf you need more than one figure to express your data, Bokeh’s got you covered. Not only does Bokeh offer the standard grid-like layout options, but it also allows you to easily organize your visualizations into a tabbed layout in just a few lines of code.In addition, your plots can be quickly linked together, so a selection on one will be reflected on any combination of the others.Preview and Save Your Beautiful Data CreationFinally, it’s time to see what you created.Whether you’re viewing your visualization in a browser or notebook, you’ll be able to explore your visualization, examine your customizations, and play with any interactions that were added.If you like what you see, you can save your visualization to an image file. Otherwise, you can revisit the steps above as needed to bring your data vision to reality.That’s it! Those six steps are the building blocks for a tidy, flexible template that can be used to take your data from the table to the big screen:"""Bokeh Visualization TemplateThis template is a general outline for turning your data into avisualization using Bokeh."""# Data handlingimport pandas as pdimport numpy as np# Bokeh librariesfrom bokeh.io import output\_file, output\_notebookfrom bokeh.plotting import figure, showfrom bokeh.models import ColumnDataSourcefrom bokeh.layouts import row, column, gridplotfrom bokeh.models.widgets import Tabs, Panel# Prepare the data# Determine where the visualization will be renderedoutput\_file('filename.html') # Render to static HTML, oroutput\_notebook() # Render inline in a Jupyter Notebook# Set up the figure(s)fig = figure() # Instantiate a figure() object# Connect to and draw the data# Organize the layout# Preview and saveshow(fig) # See what I made, and save if I like itSome common code snippets that are found in each step are previewed above, and you’ll see how to fill out the rest as you move through the rest of the tutorial!Remove adsGenerating Your First FigureThere are multiple ways to output your visualization in Bokeh. In this tutorial, you’ll see these two options:output\_file('filename.html') will write the visualization to a static HTML file.output\_notebook() will render your visualization directly in a Jupyter Notebook.It’s important to note that neither function will actually show you the visualization. That doesn’t happen until show() is called. However, they will ensure that, when show() is called, the visualization appears where you intend it to.By calling both output\_file() and output\_notebook() in the same execution, the visualization will be rendered both to a static HTML file and inline in the notebook. However, if for whatever reason you run multiple output\_file() commands in the same execution, only the last one will be used for rendering.This is a great opportunity to give you your first glimpse at a default Bokeh figure() using output\_file():# Bokeh Librariesfrom bokeh.io import output\_filefrom bokeh.plotting import figure, show# The figure will be rendered in a static HTML file called output\_file\_test.htmloutput\_file('output\_file\_test.html',title='Empty Bokeh Figure')# Set up a generic figure() objectfig = figure()# See what it looks likeshow(fig) As you can see, the result is the same, just rendered in a different location.  More information about both output\_file() and output\_notebook() can be found in the [Bokeh official docs](https://bokeh.pydata.org/en/latest/docs/reference/io.html" \l "bokeh-io-output).  # Import reset\_output (only needed once)  from bokeh.plotting import reset\_output  # Use reset\_output() between subsequent show() calls, as needed  reset\_output()  Before moving on, you may have noticed that the default Bokeh figure comes pre-loaded with a toolbar. This is an important sneak preview into the interactive elements of Bokeh that come right out of the box. | | | |