DAILY ASSESSMENT FORMAT

Date:	**27/6/2020**	**Name:**	**Samruddi Shetty**
Course:	**Python**	**USN:**	**4AL16IS047**
Topic:	**DAY 8**	**Semester & Section:**	**8th sem**
Github Repository:			

FORENOON SESSION DETAILS

Interactive Data Visualization in Python With Bokeh

Bokeh prides itself on being a library for interactive data visualization.

Unlike popular counterparts in the Python visualization space, like Matplotlib and Seaborn, Bokeh renders its graphics using HTML and JavaScript. This makes it a great candidate for building web-based dashboards and applications. However, it's an equally powerful tool for exploring and understanding your data or creating beautiful custom charts for a project or report.

Using a number of examples on a real-world dataset, the goal of this tutorial is to get you up and running with Bokeh.

You'll learn how to:

1. Transform your data into visualizations, using Bokeh
2. Customize and organize your visualizations
3. Add interactivity to your visualizations

From Data to Visualization

Building a visualization with Bokeh involves the following steps:

1. Prepare the data
2. Determine where the visualization will be rendered
3. Set up the figure(s)
4. Connect to and draw your data
5. Organize the layout
6. Preview and save your beautiful data creation

Let's explore each step in more detail.

Prepare the Data

Any 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 Rendered

At 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 Data

Next, 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 Layout

If 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 Creation

Finally, 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 Template

This template is a general outline for turning your data into a
visualization using Bokeh.
"""
# Data handling
import pandas as pd
import numpy as np

# Bokeh libraries
from bokeh.io import output_file, output_notebook
from bokeh.plotting import figure, show
from bokeh.models import ColumnDataSource
from bokeh.layouts import row, column, gridplot
from bokeh.models.widgets import Tabs, Panel

# Prepare the data

# Determine where the visualization will be rendered
output_file('filename.html')  # Render to static HTML, or
output_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 save
show(fig)  # See what I made, and save if I like it
```

Some 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!

Generating Your First Figure

There are multiple ways to output your visualization in Bokeh. In this tutorial, you'll see these two options:

1. **output_file('filename.html')** will write the visualization to a static HTML file.
2. **output_notebook()** will render your visualization directly in a Jupyter Notebook.

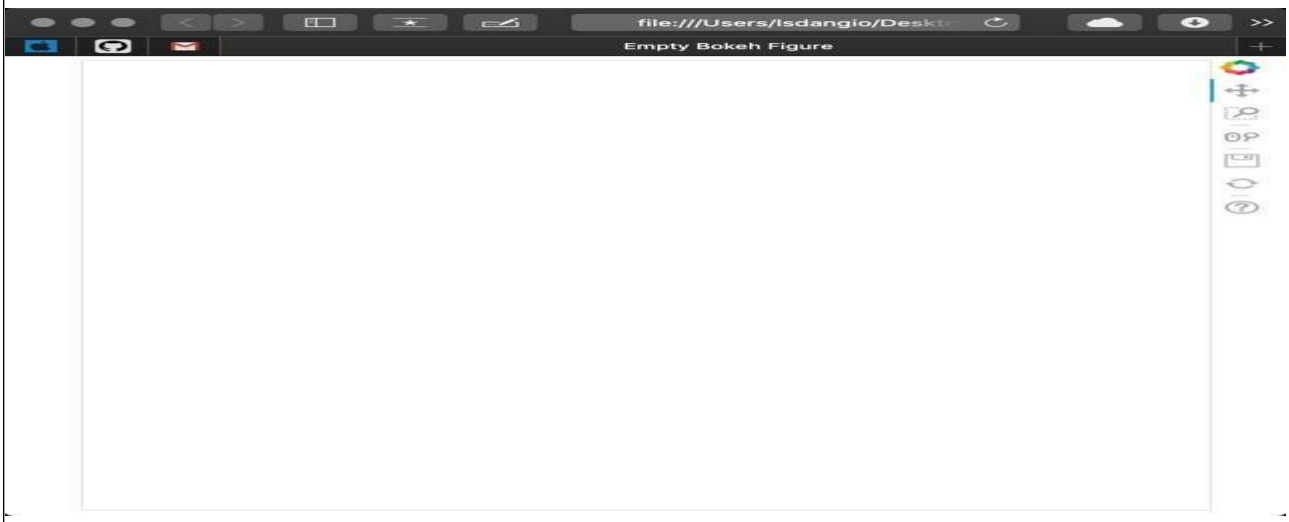
It's important to note that neither function will actually show you the visualization. That doesn't happen until show() is called. However, they will ensure that, when show() is called, the visualization appears where you intend it to.

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():

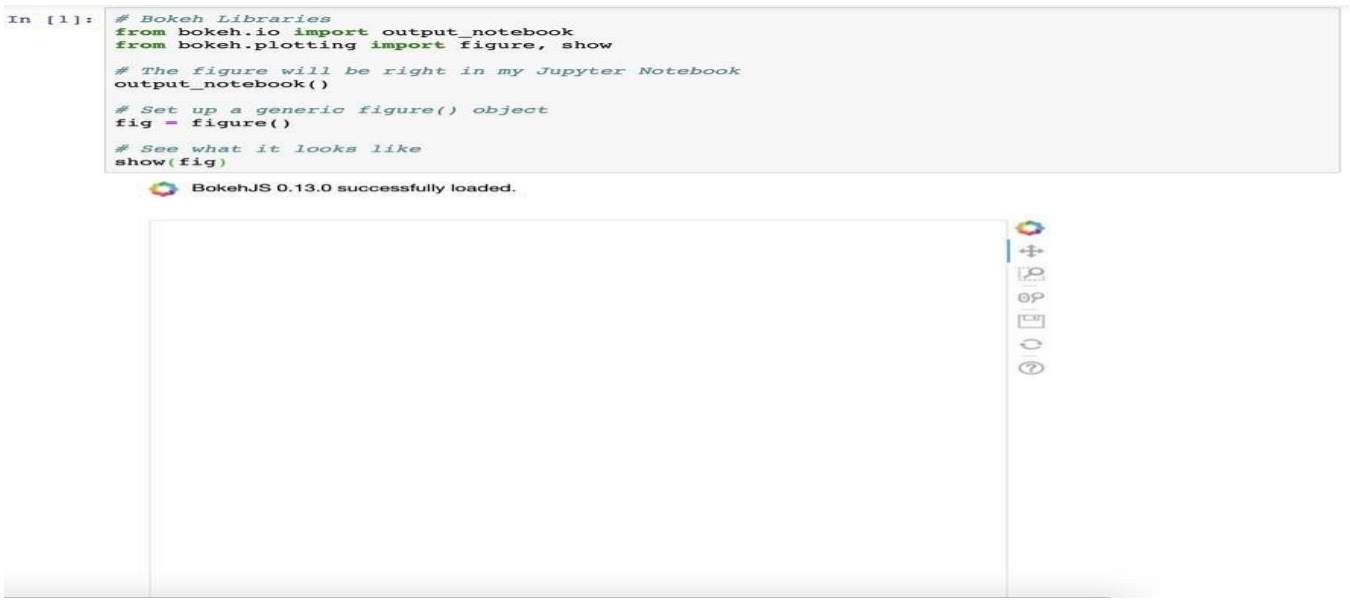
```
# Bokeh Libraries
from bokeh.io import output_file
from bokeh.plotting import figure, show

# The figure will be rendered in a static HTML file called output_file_test.html
output_file('output_file_test.html',
            title='Empty Bokeh Figure')

# Set up a generic figure() object
fig = figure()

# See what it looks like
```

```
show(fig)
```



If you were to run the same code snippet with `output_notebook()` in place of `output_file()`, assuming you have a Jupyter Notebook fired up and ready to go, you will get the following:

```
# Bokeh Libraries
from bokeh.io import output_notebook
from bokeh.plotting import figure, show

# The figure will be right in my Jupyter Notebook
output_notebook()

# Set up a generic figure() object
fig = figure()

# See what it looks like
show(fig)
```



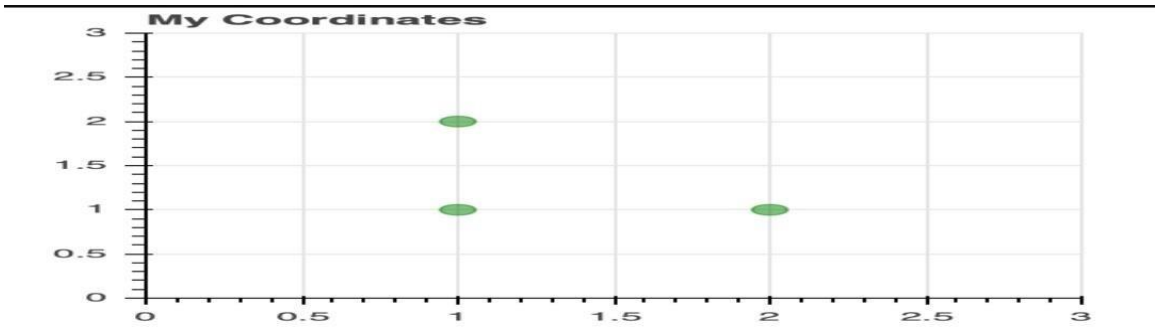
Drawing Data With Glyphs

An empty figure isn't all that exciting, so let's look at glyphs: the building blocks of Bokeh visualizations. A glyph is a vectorized graphical shape or marker that is used to represent your data, like a circle or square. More examples can be found in the Bokeh gallery. After you create your figure, you are given access to a bevy of configurable glyph methods.

Let's start with a very basic example, drawing some points on an x-y coordinate grid:

```
# Bokeh Libraries
from bokeh.io import output_file
from bokeh.plotting import figure, show

# My x-y coordinate data
x = [1, 2, 1]
y = [1, 1, 2]

# Output the visualization directly in the notebook
output_file('first_glyphs.html', title='First Glyphs')

# Create a figure with no toolbar and axis ranges of [0,3]
fig = figure(title='My Coordinates',
             plot_height=300, plot_width=300,
             x_range=(0, 3), y_range=(0, 3),
```

```
            toolbar_location=None)

# Draw the coordinates as circles
fig.circle(x=x, y=y,
           color='green', size=10, alpha=0.5)

# Show plot
show(fig)
```



Here are a few categories of glyphs:

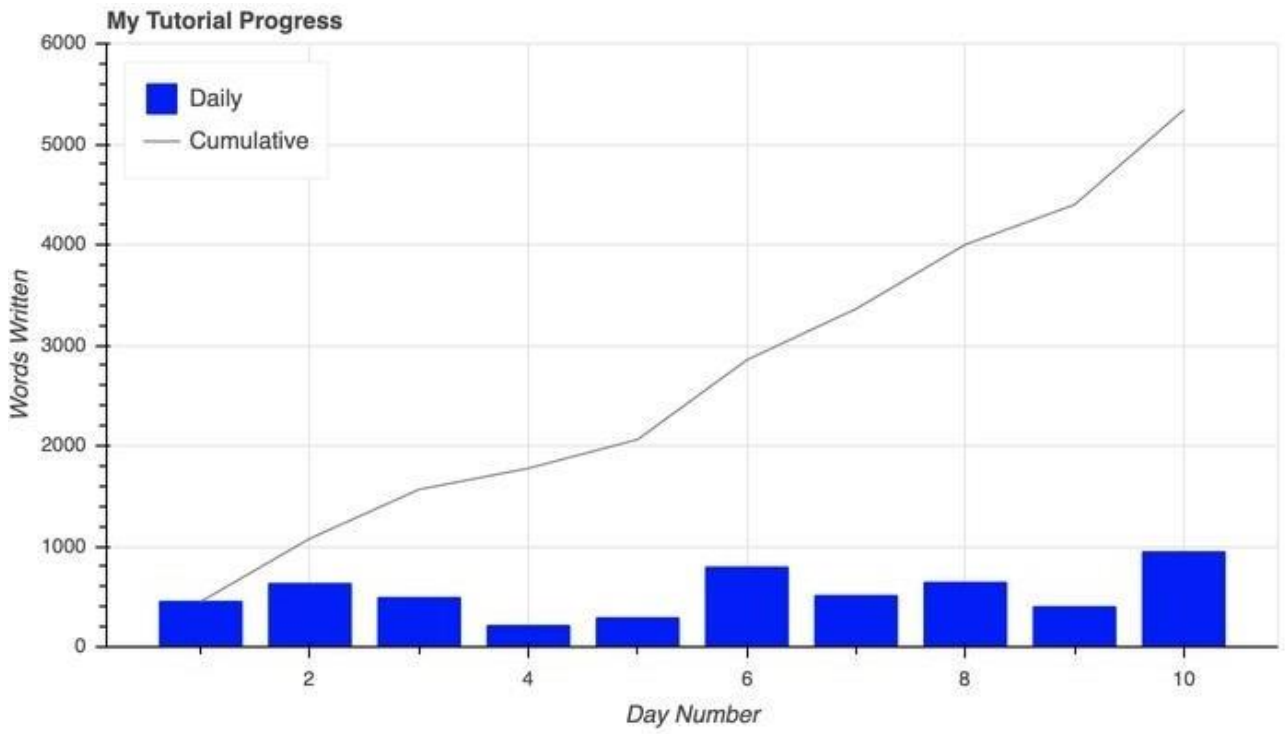
1. Marker includes shapes like circles, diamonds, squares, and triangles and is effective for creating visualizations like scatter and bubble charts.
2. Line covers things like single, step, and multi-line shapes that can be used to build line charts.
3. Bar/Rectangle shapes can be used to create traditional or stacked bar (hbar) and column (vbar) charts as well as waterfall or gantt charts.

Information about the glyphs above, as well as others, can be found in Bokeh's Reference Guide.
These glyphs can be combined as needed to fit your visualization needs. Let's say I want to create a visualization that shows how many words I wrote per day to make this tutorial, with an overlaid trend line of the cumulative word count:

```
import numpy as np



# Bokeh libraries

from bokeh.io import output_notebook

from bokeh.plotting import figure, show
```

```
# My word count data
day_num_=_np.linspace(1,_10,_10)
daily_words_=_[450,_628,_488,_210,_287,_791,_508,_639,_397,_943]
cumulative_words_=_np.cumsum(daily_words)

# Output the visualization directly in the notebook
output_notebook()

# Create a figure with a datetime type x-axis
fig_=_figure(title='My Tutorial Progress',
_____________plot_height=400,_plot_width=700,
_____________x_axis_label='Day Number',_y_axis_label='Words Written',
_____________x_minor_ticks=2,_y_range=(0,_6000),
_____________toolbar_location=None)

# The daily words will be represented as vertical bars (columns)
fig.vbar(x=day_num,_bottom=0,_top=daily_words,
_________color='blue',_width=0.75,
_________legend='Daily')
```

```
# The cumulative sum will be a trend line

fig.line(x=day_num,_y=cumulative_words,

_____color='gray',_line_width=1,

_____legend='Cumulative')



# Put the legend in the upper left corner

fig.legend.location_=_'top_left'



# Let's check it out

show(fig)
```



Web scraping

Is the process of gathering information from the Internet. Even copy-pasting the lyrics of your favorite song is a form of web scraping! However, the words “web scraping” usually refer to a process that involves automation. Some websites don’t like it when automatic scrapers gather their data, while others don’t mind.

If you’re scraping a page respectfully for educational purposes, then you’re unlikely to have any problems. Still, it’s a good idea to do some research on your own and make sure that you’re not violating any Terms of Service before you start a large-scale project. To learn more about the legal aspects of web scraping, check out Legal Perspectives on Scraping Data From The Modern Web.

Why Scrape the Web?
Say you’re a surfer (both online and in real life) and you’re looking for employment. However, you’re not looking for just *any* job. With a surfer’s mindset, you’re waiting for the perfect opportunity to roll your way!

There’s a job site that you like that offers exactly the kinds of jobs you’re looking for. Unfortunately, a new position only pops up once in a blue moon. You think about checking up on it every day, but that doesn’t sound like the most fun and productive way to spend your time.

Thankfully, the world offers other ways to apply that surfer’s mindset! Instead of looking at the job site every day, you can use Python to help automate the repetitive parts of your job search. **Automated web scraping** can be a solution to speed up the data collection process. You write your code once and it will get the information you want many times and from many pages.

InIn contrast, when you try to get the information you want manually, you might spend a lot of time clicking, scrolling, and searching. This is especially true if you need large amounts of data from websites that are regularly

updated with new content. Manual web scraping can take a lot of time and repetition.There's so much information on the Web, and new information is constantly added. Something among all that data is likely of interest to you, and much of it is just out there for the taking. Whether you're actually on the job hunt, gathering data to support your grassroots organization, or are finally looking to get all the lyrics from your favorite artist downloaded to your computer, automated web scraping can help you accomplish your goals.

DAILY ASSESSMENT FORMAT

Date:	**02/6/2020**	**Name:**	**Ramanath Naik**
Course:	**HDL**	**USN:**	**4AL16EC054**
Topic:	**DAY 2**	**Semester & Section:**	**8th sem B sec**
Github Repository:			

FPGA Basics: Architecture, Applications and Uses

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**

FPGA Architecture

A basic FPGA architecture (Figure 1) 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).

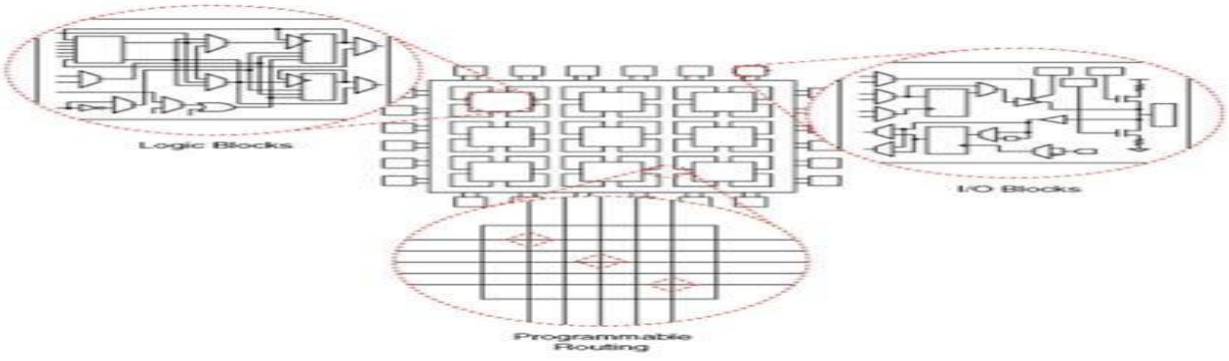


Figure 1: The fundamental FPGA architecture (Image Source: National Instruments)

An individual CLB (Figure 2) 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.

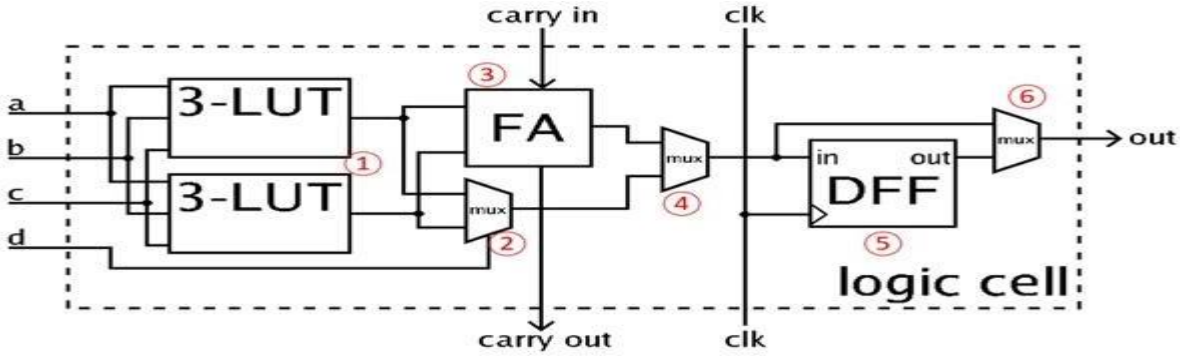


Figure 2: A simplified CLB: The four-input LUT is formed from two three-input units. (Image source: Wikipedia)

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.

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.

Lexical Tokens

Verilog language source text files are a stream of lexical tokens. A token consists of one or more characters, and each single character is in exactly one token.

The basic lexical tokens used by the Verilog HDL are similar to those in C Programming Language. Verilog is case sensitive. All the key words are in lower case.

White Space

White spaces can contain characters for spaces, tabs, new-lines and form feeds. These characters are ignored except when they serve to separate tokens.

White space characters are Blank space, Tabs, Carriage returns, New line, and Form feeds.

Comments

There are two forms to represent the comments

- 1) Single line comments begin with the token // and end with carriage return.

Ex.: //this is single line syntax

- 2) Multiline comments begins with the token /* and end with token */

Ex.: /* this is multiline Syntax*/

Numbers

You can specify a number in binary, octal, decimal or hexadecimal format. Negative numbers are represented in 2’s compliment numbers. Verilog allows integers, real numbers and signed & unsigned numbers.

The syntax is given by − <size> <radix> <value>

Size or unsized number can be defined in <Size> and <radix> defines whether it is binary, octal, hexadecimal or decimal.

Identifiers

Identifier is the name used to define the object, such as a function, module or register. Identifiers should begin with an alphabetical characters or underscore characters. Ex. A_Z, a_z,_

Identifiers are a combination of alphabetic, numeric, underscore and $ characters. They can be up to 1024 characters long.

Operators

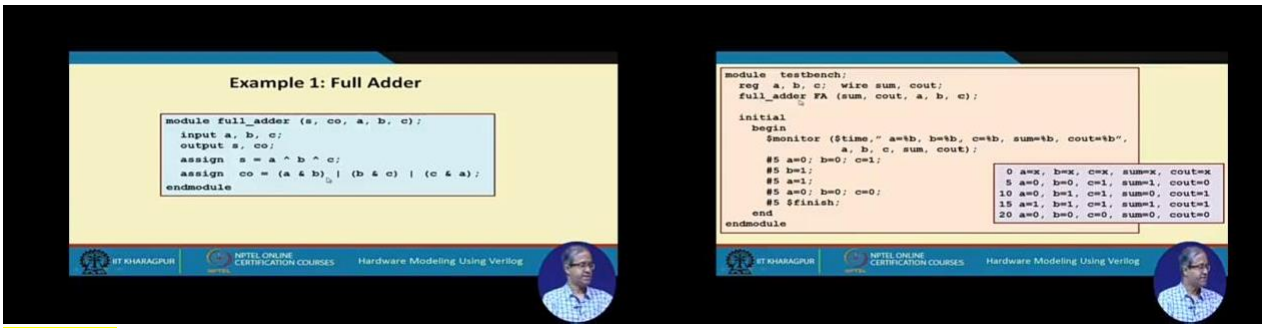
Operators are special characters used to put conditions or to operate the variables. There are one, two and sometimes three characters used to perform operations on variables.

Ex. >, +, ~, &! =.

Verilog Keywords

Words that have special meaning in Verilog are called the Verilog keywords. For example, assign, case, while, wire, reg, and, or, nand, and module. They should not be used as identifiers. Verilog keywords also include compiler directives, and system tasks and functions.

Verilog test bench code to verify the design under test:



Example

y = (b' · c') + (a · b')

```
module sillyfunction(input a, b, c,
output y);
assign y = ~b & ~c | a & ~b;
endmodule
```

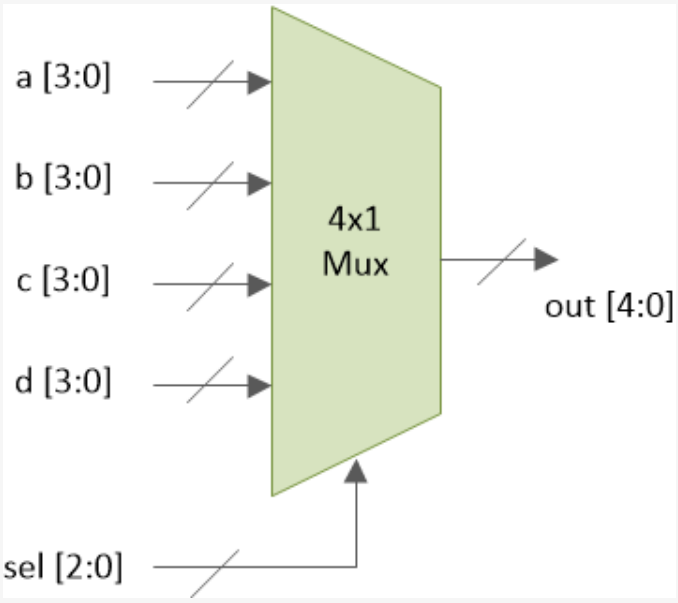
Test bench code

```
module testbench1();
reg a, b, c;
```

```
wire y;
sillyfunction dut (.a(a), .b(b), .c(c), .y(y) );d
initial begin
a = 0; b = 0; c = 0; #10;
c = 1; #10;
b = 1; c = 0; #10;
c = 1; #10;
a = 1; b = 0; c = 0; #10;
end
endmodule
```

Task 2

Implement a 4:1 MUX and write the test bench code to verify the module




```
module mux_4to1_assign ( input [3:0] a, b, c, d, input [1:0] sel, output [3:0] out);
 assign out = sel[1] ? (sel[0] ? d : c) : (sel[0] ? b : a);
endmodule
```

Testbench

```
module tb_4to1_mux;
  reg [3:0] a, b, c, d;
  wire [3:0] out;
  reg [1:0] sel;
  integer i;
mux_4to1_case mux0 ( .a (a), .b (b), .c (c), .d (d), .sel (sel), .out (out));
  initial begin
    $monitor ("[%0t] sel=0x%0h a=0x%0h b=0x%0h c=0x%0h d=0x%0h out=0x%0h", $time, sel, a, b, c,
d, out);
   sel <= 0; a <= $random; b <= $random; c <= $random; d <= $random;
```

```
    for (i = 1; i < 4; i=i+1) begin
      #5 sel <= i;
    end
 #5 $finish;
  end
endmodule
```