DAILY ASSESSMENT

Date:	**1-6-2020**	**Name:**	**Kavyashree m**
Course:	**Digital design using HDL**	**USN:**	**4al15ec036**
Topic:	**FPGA Basics: Architecture,Applications and Uses, Verilog HDL Basics by intel , Verilog Testbench code to verify the design under test**	**Semester & Section:**	**8th A**
Github Repository:	**kavya**		

FORENOON SESSION DETAILS

Image of session

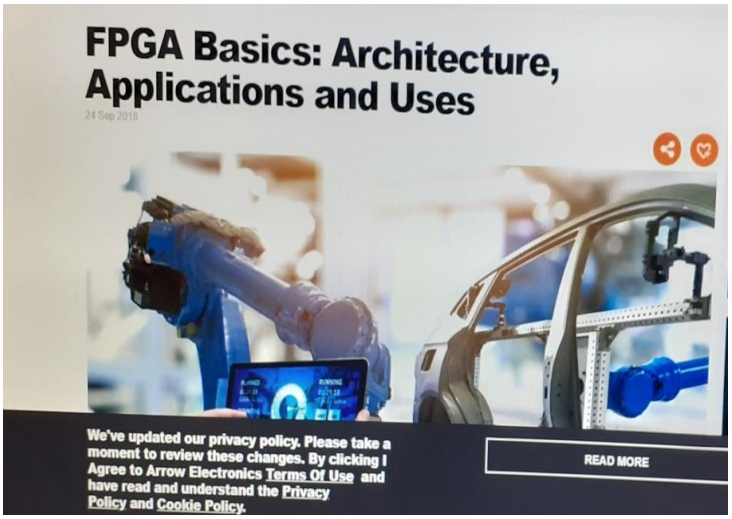


Fig 1: FPGA Basics

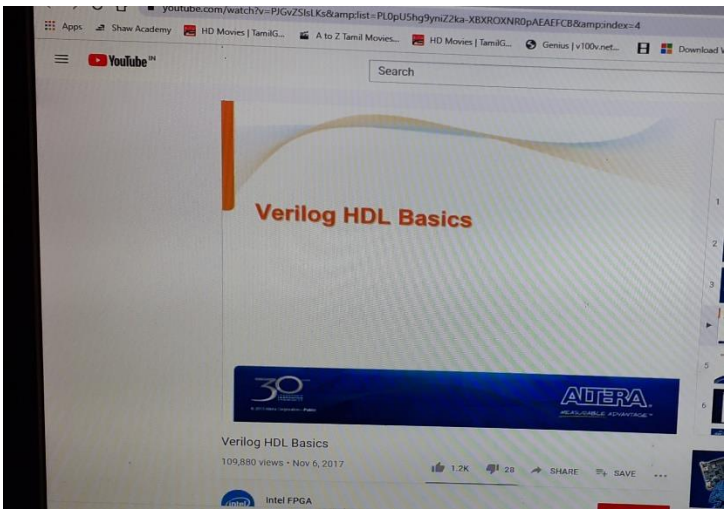


Fig 2 : Verilog HDL Basics by intel

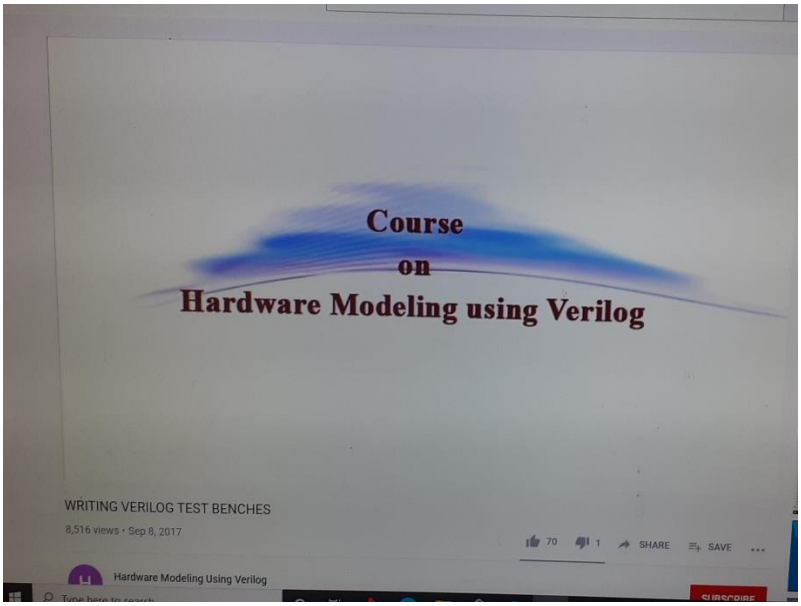
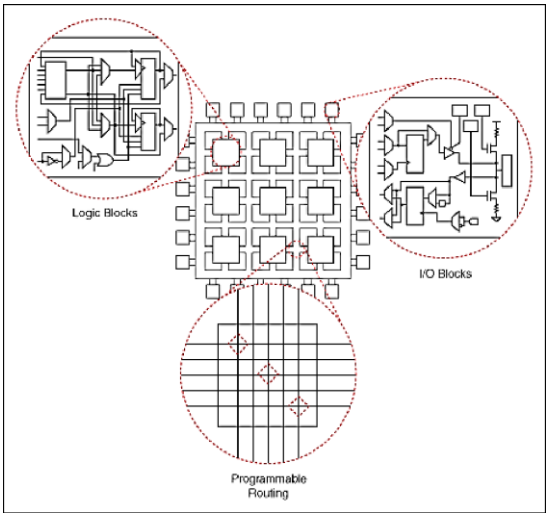


Fig 3 : Verilog Testbench code

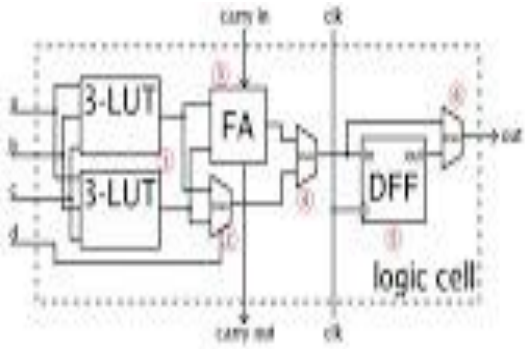
FPGA Basics

FPGA Architecture

A basic FPGA architecture consists of thousands of fundamental elements called configurable logic blocks surrounded by a system of programmable interconnects, called a fabric, that routes signals between CLBs. Input/output blocks interface between the FPGA and external devices.Depending on the manufacturer, the CLB may also be referred to as a logic block, a logic element or a logic cell .



An individual CLB is made up of several logic blocks. A lookup table 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 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 functions.

CPLD vs FPGA

Originally, FPGAs included the blocks in Figure 1 and little else, but now designers can choose from products with a large range of features. Less complex devices such as

simple programmable logic device and complex programmable logic devices (bridge the gap between discrete logic devices and entry-level FPGAs.

Entry-level FPGAs emphasize low power consumption, low logic density and low complexity per chip. Higher-function devices add functional blocks dedicated to specific functions: Examples include clock management components, phase-locked loops ,high-speed serializers and deserializers, Ethernet MACs, PCI express controllers and high-speed transceivers. These blocks can either be implemented with CLBs termed soft IP or designed as separate circuits; i.e., hard IP. Hard IP blocks gain performance at the expense of reconfigurability.

At the high end, the FPGA product family includes complex system-on-chip (SoC) parts that integrate the FPGA architecture, hard IP and a microprocessor CPU core into a single component. Compared to separate devices, a SoC FPGA provides higher integration, lower power, smaller board size and higher-bandwidth communication between the core and other blocks.

SoC FPGAs

SoC FPGAs include a wide range of processing capabilities to suit different applications. A low-cost, low-power SoC FPGA such as Intel's Cyclone V, for example, targets high-volume applications such as industrial motor control drives, protocol bridging, video processing cards and handheld devices. The device has two distinct parts: the FPGA portion and a hard processor system based around a single- or dual-core 32-bit Arm Cortex-A9 MPCORE running at 925 MHz. Each part contains its own set of peripherals, which includes hard IP from third-party vendors.

FPGA Design

How do we transform this collection of thousands of hardware blocks into the correct configuration to execute the application? An FPGA-based design begins by defining the

required computing tasks in the development tool, then compiling them into a configuration file that contains information on how to hook up the CLBs and other modules. The process is similar to a software development cycle except that the goal is to architect the hardware itself rather than a set of instructions to run on a predefined hardware platform.

Designers have traditionally used a hardware description language (HDL) such as VHDL or Verilog to design the FPGA configuration.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity signed_adder is
6   port
7   (
8     aclr : in   std_logic;
9     clk  : in   std_logic;
10    a    : in   std_logic_vector;
11    b    : in   std_logic_vector;
12    q    : out  std_logic_vector
13  );
14 end signed_adder;
15
16 architecture signed_adder_arch of signed_adder is
17   signal q_s : signed(a'high+1 downto 0); -- extra bit wide
18
19 begin -- architecture
20   assert(a'length >= b'length)
21     report "Port A must be the longer vector if different sizes!"
22     severity FAILURE;
23   q <= std_logic_vector(q_s);
24
25   adding_proc:
26   process (aclr, clk)
27     begin
28       if (aclr = '1') then
29         q_s <= (others => '0');
30       elsif rising_edge(clk) then
31         q_s <= ('0'&signed(a)) + ('0'&signed(b));
32       end if; -- clk'd
33     end process;
34
35 end signed_adder_arch;
```

As a result, vendors are offering software development kits that allow designers to develop FPGA solutions in popular high-level languages such as C/C++, Python and OpenCL. High-level synthesis design tools are also available; these run on a framework such as National Instrument's LabVIEW and feature graphical block diagrams instead of lines of code.

FPGA Applications

Many applications rely on the parallel execution of identical operations; the ability to configure the FPGA's CLBs into hundreds or thousands of identical processing blocks has applications in image processing, artificial intelligence (AI), data center hardware accelerators, enterprise networking and automotive advanced driver assistance systems (ADAS).

Many of these application areas are changing very quickly as requirements evolve and new protocols and standards are adopted. FPGAs enable manufacturers to implement systems that can be updated when necessary.

A good example of FPGA use is high-speed search: Microsoft is using FPGAs in its data centers to run Bing search algorithms. The FPGA can change to support new algorithms as they are created. If needs change, the design can be repurposed to run simulation or modeling routines in an HPC application. This flexibility is difficult or impossible to achieve with an ASIC.

Other FPGA uses include aerospace and defense, medical electronics, digital television, consumer electronics, industrial motor control, scientific instruments, cybersecurity systems and wireless communications.

Verilog HDL Basics by intel

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

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.

Gate Level Modelling

Verilog has built-in primitives like logic gates, transmission gates and switches. These are rarely used for design work but they are used in post synthesis world for modelling of ASIC/FPGA cells.

Gate level modelling exhibits two properties −

Drive strength − The strength of the output gates is defined by drive strength. The output is strongest if there is a direct connection to the source. The strength decreases if the connection is via a conducting transistor and least when connected via a pull-up/down resistive. The drive strength is usually not specified, in which case the strengths defaults to strong1 and strong0.

Delays − If delays are not specified, then the gates do not have propagation delays; if two delays are specified, then first one represents the rise delay and the second one, fall delay; if only one delay is specified, then both, rise and fall are equal. Delays can be ignored in synthesis.

Gate Primitives

The basic logic gates using one output and many inputs are used in Verilog. GATE uses one of the keywords - and, nand, or, nor, xor, xnor for use in Verilog for N number of inputs and 1 output.

Example:

```
Module gate()
Wire ot0;
Wire ot1;
Wire ot2;

Reg in0,in1,in2,in3;
Not U1(ot0,in0);
Xor U2(ot1,in1,in2,in3);
```

```
And U3(ot2, in2,in3,in0)
```

Transmission Gate Primitives

Transmission gate primitives include both, buffers and inverters. They have single input and one or more outputs. In the gate instantiation syntax shown below, GATE stands for either the keyword buf or NOT gate.

Example: Not, buf, bufif0, bufif1, notif0, notif1

Not – n outout inverter

Buf – n output buffer

Bufifo – tristate buffer, active low enable

Bufif1 – tristate buffer, active high enable

Notifo – tristate inverter, active low enable

Notif1 – tristate inverter, active high enable

Example:

```
Module gate()
Wire out0;
Wire out1;

Reg in0,in1;
Not U1(out0,in0);
Buf U2(out0,in0);
```

Data Types

Value Set

Verilog consists of, mainly, four basic values. All Verilog data types, which are used in Verilog store these values −

0 (logic zero, or false condition)

1 (logic one, or true condition)

x (unknown logic value)

z (high impedance state)

use of x and z is very limited for synthesis.

Wire

A wire is used to represent a physical wire in a circuit and it is used for connection of gates or modules. The value of a wire can only be read and not assigned in a function or block. A wire cannot store value but is always driven by a continuous assignment statement or by connecting wire to output of a gate/module. Other specific types of wires are −

Wand (wired-AND) − here value of Wand is dependent on logical AND of all the device drivers connected to it.

Wor (wired-OR) − here value of a Wor is dependent on logical OR of all the device drivers connected to it.

Tri (three-state) − here all drivers connected to a tri must be z, except only one (which determines value of tri).

Example:

```
Wire [msb:lsb] wire_variable_list;
Wirec // simple wire
Wand d;

Assign d = a; // value of d is the logical AND of
Assign d = b; // a and b
Wire [9:0] A; // a cable (vector) of 10 wires.

Wand [msb:lsb] wand_variable_list;
```

Wor [msb:lsb] wor_variable_list;

Tri [msb:lsb] tri_variable_list;

Register

A reg is a data object, which is holding the value from one procedural assignment to next one and are used only in different functions and procedural blocks. A reg is a simple Verilog, variable-type register and can't imply a physical register. In multi-bit registers, the data is stored in the form of unsigned numbers and sign extension is not used.

Example :

```
reg c; // single 1-bit register variable
reg [5:0] gem; // a 6-bit vector;
reg [6:0] d, e; // two 7-bit variables
```

Input, Output, Inout

These keywords are used to declare input, output and bidirectional ports of a task or module. Here input and inout ports, which are of wire type and output port is configured to be of wire, reg, wand, wor or tri type. Always, default is wire type.

Example:

```
Module sample(a, c, b, d);
Input c;   // An input where wire is used.

Output a, b;  // Two outputs where wire is used.
Output [2:0] d;  /* A three-bit output. One must declare type in a separate statement. */
reg [1:0] a;  // The above 'a' port is for declaration in reg.
```

Integer

Integers are used in general-purpose variables. They are used mainly in loops-indicies, constants, and parameters. They are of 'reg' type data type. They store data as signed numbers whereas explicitly declared reg types store them as an unsigned data. If the integer is not defined at the time of compiling, then the default size would be 32 bits. If an integer holds a constant, the synthesizer adjusts them to the minimum width needed at the time of compilation.

Example

```
Integer c;   // single 32-bit integer
Assign a = 63;  // 63 defaults to a 7-bit variable.
```

Supply0, Supply1

Supply0 define wires tied to logic 0 (ground) and supply1 define wires tied to logic 1 (power).

Example

```
supply0 logic_0_wires;
supply0 gnd1;  // equivalent to a wire assigned as 0
supply1 logic_1_wires;
supply1 c, s;
```

Time

Time is a 64-bit quantity that can be used in conjunction with the $time system task to hold simulation time. Time is not supported for synthesis and hence is used only for simulation purposes.

Example

```
time time_variable_list;
time c;
c = $time;   //c = current simulation time
```

Parameter

A parameter is defining a constant which can be set when you use a module, which allows customization of module during the instantiation process.

Example

```
Parameter add = 3'b010, sub = 2'b11;
Parameter n = 3;
Parameter [2:0] param2 = 3'b110;

reg [n-1:0] jam; /* A 3-bit register with length of n or above. */
always @(z)
y = {{(add - sub){z}};

if (z)
begin
  state = param2[1];
else
  state = param2[2];
end
```

Operators

Arithmetic Operators

These operators is perform arithmetic operations. The + and −are used as either unary (x) or binary (z−y) operators.

The Operators which are included in arithmetic operation are −

+ (addition), −(subtraction), * (multiplication), / (division), % (modulus)

Example :

```
parameter v = 5;
```

```
reg[3:0] b, d, h, i, count;
h = b + d;
i = d - v;
cnt = (cnt +1)%16; //Can count 0 thru 15.
```

Relational Operators

These operators compare two operands and return the result in a single bit, 1 or 0.

Wire and reg variables are positive. Thus (−3'd001) = = 3'd111 and (−3b001)>3b110.

The Operators which are included in relational operation are −

- == (equal to)
- != (not equal to)
- > (greater than)
- >= (greater than or equal to)
- < (less than)
- <= (less than or equal to)

Example

```
if (z = = y) c = 1;
  else c = 0; // Compare in 2's compliment; d>b
reg [3:0] d,b;

if (d[3]= = b[3]) d[2:0] > b[2:0];
  else b[3];
```

Equivalent Statement

```
e = (z == y);
```

Bit-wise Operators

Bit-wise operators which are doing a bit-by-bit comparison between two operands.

The Operators which are included in Bit wise operation are −

- & (bitwise AND)
- | (bitwiseOR)
- ~ (bitwise NOT)
- ^ (bitwise XOR)
- ~^ or ^~(bitwise XNOR)

Example

```
module and2 (d, b, c);
input [1:0] d, b;
output [1:0] c;
assign c = d & b;
end module
```

Logical Operators

Logical operators are bit-wise operators and are used only for single-bit operands. They return a single bit value, 0 or 1. They can work on integers or group of bits, expressions and treat all non-zero values as 1. Logical operators are generally, used in conditional statements since they work with expressions.

The operators which are included in Logical operation are −

- ! (logical NOT)
- && (logical AND)
- || (logical OR)

Example

```
wire[7:0] a, b, c; // a, b and c are multibit variables.
reg x;

if ((a == b) && (c)) x = 1; //x = 1 if a equals b, and c is nonzero.
```

else x = !a; // x =0 if a is anything but zero.

Reduction Operators

Reduction operators are the unary form of the bitwise operators and operate on all the bits of an operand vector. These also return a single-bit value.

The operators which are included in Reduction operation are −

- & (reduction AND)
- | (reduction OR)
- ~& (reduction NAND)
- ~| (reduction NOR)
- ^ (reduction XOR)
- ~^ or ^~(reduction XNOR)

Example

```
Module chk_zero (x, z);
Input [2:0] x;
Output z;
Assign z = & x; // Reduction AND
End module
```

Shift Operators

Shift operators, which are shifting the first operand by the number of bits specified by second operand in the syntax. Vacant positions are filled with zeros for both directions, left and right shifts (There is no use sign extension).

The Operators which are included in Shift operation are −

- << (shift left)
- >> (shift right)

Example

```
Assign z = c << 3; /* z = c shifted left 3 bits;
Vacant positions are filled with 0's */
```

Concatenation Operator

The concatenation operator combines two or more operands to form a larger vector.

The operator included in Concatenation operation is − { }(concatenation)

Example

```
wire [1:0] a, h; wire [2:0] x; wire [3;0] y, Z;
assign x = {1'b0, a}; // x[2] = 0, x[1] = a[1], x[0] = a[0]
assign b = {a, h}; /* b[3] = a[1], b[2] = a[0], b[1] = h[1],
b[0] = h[0] */
assign {cout, b} = x + Z; // Concatenation of a result
```

Replication Operator

The replication operator are making multiple copies of an item.

The operator used in Replication operation is − {n{item}} (n fold replication of an item)

Example

```
Wire [1:0] a, f; wire [4:0] x;
Assign x = {2{1'f0}, a}; // Equivalent to x = {0,0,a }
Assign y = {2{a}, 3{f}}; //Equivalent to y = {a,a,f,f}
```

For synthesis, Synopsis did not like a zero replication.

For example:-

```
Parameter l = 5, k = 5;
Assign x = {(l-k){a}}
```

Conditional Operator

Conditional operator synthesizes to a multiplexer. It is the same kind as is used in C/C++ and evaluates one of the two expressions based on the condition.

The operator used in Conditional operation is −

(Condition) ? (Result if condition true) −

(result if condition false)

Example

Assign x = (g) ? a : b;

Assign x = (inc = = 2) ? x+1 : x-1;

/* if (inc), x = x+1, else x = x-1 */

Verilog Testbench code to verify the design under test

HDL code written to test another HDL module, the device under test , also called the unit under test ′ Not synthesizeable ′ Types of testbenches:

- Simple testbench
- Self-checking testbench
- Self-checking testbench with testvectors

Example

Write Verilog code to implement the following function in hardware:

$y = (b \cdot c) + (a \cdot b)$

.

AFTERNOON SESSION DETAILS

Date:	**2-6-2020**	**Name:**	**Kavyashree m**
Course:	**Python programming**	**USN:**	**4al15ec036**
Topic:	**Interactive Data Visualization with Bokeh, Webscraping with Python Beautiful Soup**	**Semester & Section:**	**8th A**
Github Repository:	**kavya**		

Image of session

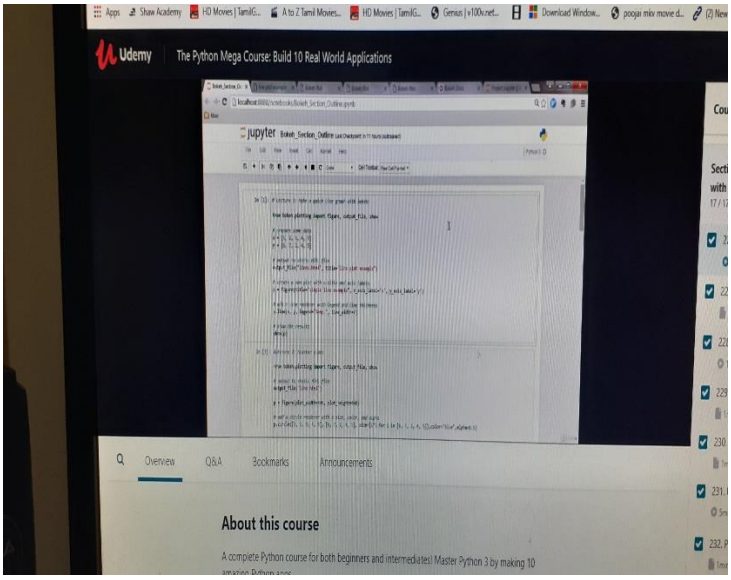


Fig 1: Interactive Data Visualization with Bokeh

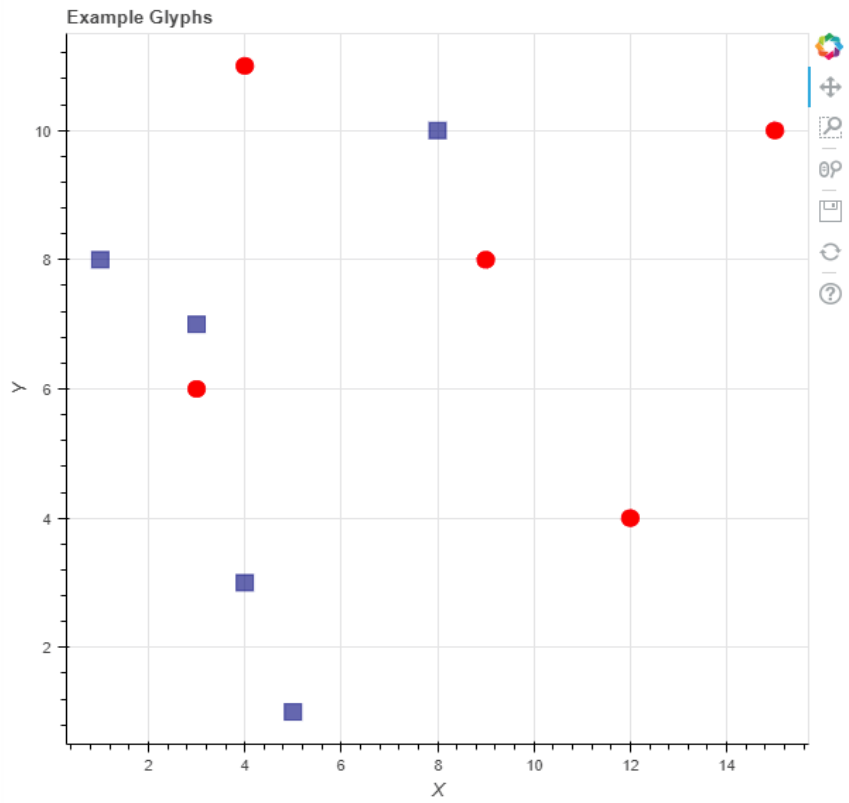
Interactive Data Visualization with Bokeh

The major concept of Bokeh is that graphs are built up one layer at a time. We start out by creating a figure, and then we add elements, called glyphs. Glyphs can take on many shapes depending on the desired use: circles, lines, patches, bars, arcs, and so on. Let's illustrate the idea of glyphs by making a basic chart with squares and circles.

```
#bokehbasics
        from bokeh.plotting import figure
```

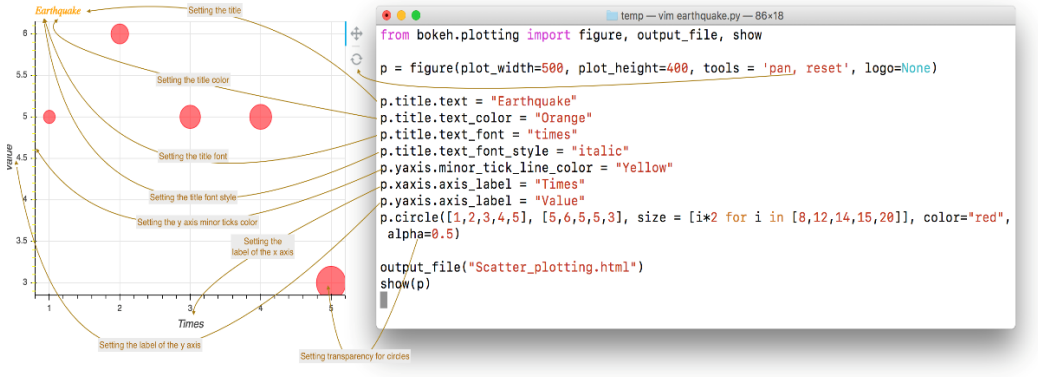
```
from bokeh.io import show, output_notebook

# Create a blank figure with labels
p = figure(plot_width = 600, plot_height = 600,
        title = 'Example Glyphs',
        x_axis_label = 'X', y_axis_label = 'Y')

# Example data
squares_x = [1, 3, 4, 5, 8]
squares_y = [8, 7, 3, 1, 10]
circles_x = [9, 12, 4, 3, 15]
circles_y = [8, 4, 11, 6, 10]

# Add squares glyph
p.square(squares_x, squares_y, size = 12, color = 'navy', alpha = 0.6)
# Add circle glyph
p.circle(circles_x, circles_y, size = 12, color = 'red')

# Set to output the plot in the notebook
output_notebook()
# Show the plot
show(p)
```

This generates the slightly uninspiring plot below:



Visual Attributes

Once you have built a basic plot, you can customize its visual attributes including changing the title color and font, adding labels for xaxis and yaxis, changing the color of the axis ticks, etc. All these properties are illustrated in the diagram below:



And here is the code if you want to play around with it:

1. from bokeh.plotting import figure, output_file, show

```
2. p = figure(plot_width=500, plot_height=400, tools = 'pan, reset')
3. p.title.text = "Earthquakes"
4. p.title.text_color = "Orange"
5. p.title.text_font = "times"
6. p.title.text_font_style = "italic"
7. p.yaxis.minor_tick_line_color = "Yellow"
8. p.xaxis.axis_label = "Times"
9. p.yaxis.axis_label = "Value"
10. p.circle([1,2,3,4,5], [5,6,5,5,3], size = [i*2 for i in [8,12,14,15,20]],
    color="red", alpha=0.5)
11. output_file("Scatter_plotting.html")
12. show(p)
```

For a complete list of visual attributes, see the Styling Visual Attributes documentation page of Bokeh.

Webscraping with Python Beautiful Soup

The incredible amount of data on the Internet is a rich resource for any field of research or personal interest. To effectively harvest that data, you'll need to become skilled at web scraping.

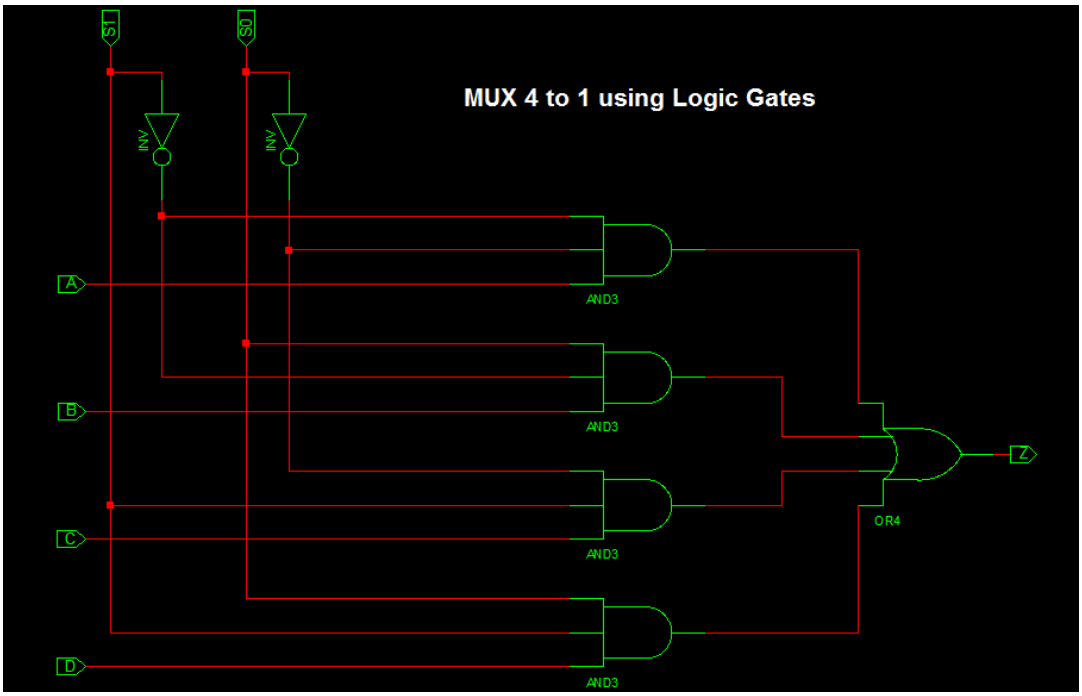
Task 2

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

Testbench

```
module tb_4to1_mux;
  // Declare internal reg variables to drive design inputs
  // Declare wire signals to collect design output
  // Declare other internal variables used in testbench
  reg [3:0] a;
  reg [3:0] b;
  reg [3:0] c;
  reg [3:0] d;
  wire [3:0] out;
  reg [1:0] sel;
  integer i;

  // Instantiate one of the designs, in this case, we have used the design with case statement
  // Connect testbench variables declared above with those in the design
  mux_4to1_case  mux0 (   .a (a),
                          .b (b),
                          .c (c),
                          .d (d),
                          .sel (sel),
                          .out (out));

  // This initial block is the stimulus
  initial begin
```

```
    // Launch a monitor in background to display values to log whenever a/b/c/d/sel/out changes
    $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);

     // 1. At time 0, drive random values to a/b/c/d and keep sel = 0
    sel <= 0;
    a <= $random;
    b <= $random;
    c <= $random;
    d <= $random;

    // 2. Change the value of sel after every 5ns
    for (i = 1; i < 4; i=i+1) begin
      #5 sel <= i;
    end

    // 3. After Step2 is over, wait for 5ns and finish simulation
    #5 $finish;
  end
endmodule
```


MUX 4 to 1 using Logic Gates
S1
S0
INV
INV
A
B
C
D
AND3
AND3
AND3
AND3
OR4
Z