Experiment 5
Study of Basic operators and data types in Verilog HDL

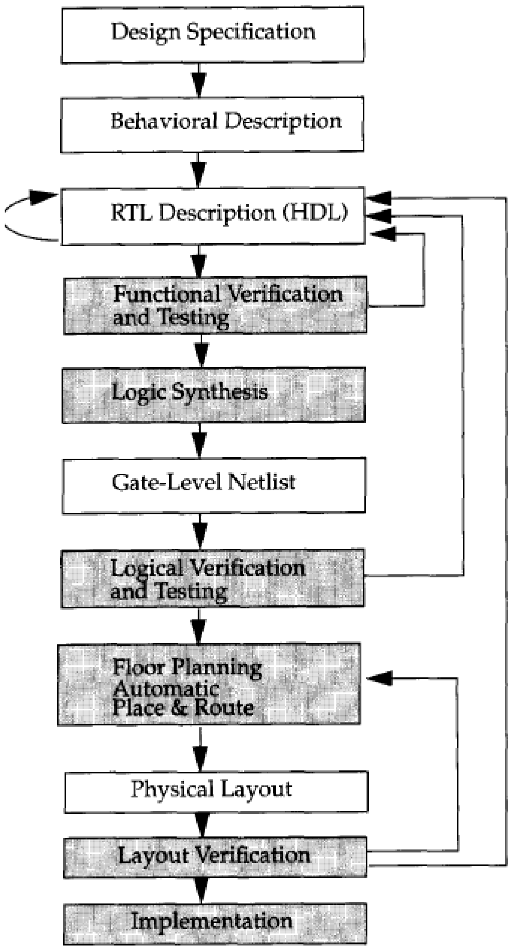
Aim:

- Introduction to Digital system design with Verilog HDL
- Familiarizing with operators and data types

Introduction:
The world around us has become digital. Hence, digital systems have become the dominant part of our lives. Although most of us enjoy the benefits offered by digital systems, it is the duty of a candidate engineer to learn how to design and analyze them. Besides, digital design concepts have become topics of interest to a hobbyist and the maker community due to their power in implementing systems. There are several ways to implement a digital system and implementation by Field Programmable Gate Arrays (FPGAs) will be the prime focus. FPGA can be taken as a generic platform such that a digital system can be implemented on it. Evaluation boards using such chips have become widespread and Basys3 is one among them. Basys3 boards have their FPGA from the Xilinx Artix-7 XC7A35T family, specifically XC7A35TCPG236-1. The details of the board can be referred from the Appendix A.

Typical design flow:

In any design, specifications are written first which describe abstractly the functionality, interface, and overall architecture of the digital circuit to be designed. A behavioral description is then created to analyze the design in terms of functionality, performance, compliance to standards, and other high-level issues. The behavioral description is manually converted to an RTL description in an HDL. The designer has to describe the data flow that will implement the desired digital circuit. From this point onward, the design process is done with the assistance of Computer-Aided Design (CAD) tools. Logic synthesis tools convert the RTL description to a gate-level netlist. A gate level netlist is a description of the circuit in terms of gates and connections between them. The gate-level netlist is input to an Automatic Place and Route tool, which creates a layout. The layout is verified and then fabricated on chip.



Hardware Description Languages:

For a long time, programming languages such as FORTRAN, Pascal, and C were being used to describe computer programs that were sequential in nature. Similarly, in the digital design field, designers felt the need for a standard language to describe digital circuits. Thus, Hardware Description Languages (HDLs) came into existence. HDLs allowed the designers to model the concurrency of processes found in hardware elements. Hardware description languages such as Verilog HDL and VHDL became popular. Both Verilog and VHDL simulators to simulate large digital circuits.
Even though HDLs were popular for logic verification, designers had to manually translate the HDL-based design into a schematic circuit with interconnections between gates. The advent of logic synthesis in the late 1980s changed the design methodology radically. Digital circuits could be described at a register transfer level (RTL) by use of an HDL. Thus, the designer had to specify how the data flows between registers and how the design processes the data. The details of gates and their interconnections to implement the circuit were automatically extracted by logic synthesis tools from the RTL description. Designers no longer had to manually place gates to build digital circuits. They could describe complex circuits at an abstract level in terms of functionality and data flow by designing those circuits in HDLs. Logic synthesis tools would implement the specified functionality in terms of gates and gate

interconnections. HDLs also began to be used for system-level design. HDLs were used for simulation of system boards, interconnect buses, FPGAs (Field Programmable Gate Arrays), and PALS (Programmable Array Logic).

Structure of a Verilog module

Verilog module has declaration and body. In the declaration, the name, inputs, and outputs of the module are entered. The body shows the relationship between the inputs and the outputs.

```
module {module_name} (port_list);
//Port definitions
//Description of the digital system
statement1
statement2
statement3
………………
endmodule
```

The name of the module is user-defined. Verilog HDL is case-sensitive. The name of the module should start with an alphabetical letter and can include the special character underscore (_). The declaration of the module starts with the predefined word module followed by the user-selected name. The names of the inputs and outputs (they are called input and output ports) follow the same guidelines as the module's name. They are written inside parentheses separated by a comma. The parenthesis is followed by a semicolon. The order of writing the input and output ports inside the parentheses is irrelevant. Following the module statement, the input and output port modes are declared. More than one input or output could be entered on the same line by using a comma (,) to separate each input or output. The double slash (//) is a comment command where a comment can be entered. If the comment takes more than one line, a double slash or pair (/*……..*/) can be used. The module is concluded by the predefined word endmodule. Leaving blank lines is allowed in the module; also, spaces between two words or at the beginning of the line are allowed.

Styles (types) of description

a) Data Flow description
   Data flow describes how the system's signals flow from the inputs to the outputs. Usually, the description is done by writing the Boolean function of the outputs. The data-flow statements are concurrent; their execution is controlled by events.
b) Behavioral Description
   A behavioral description models the system as to how the outputs behave with the inputs. The behavioral description is the one where the module contains the

keyword **always or initial**. Behavioral description is usually used when the Boolean function or the digital logic of the system is hard to obtain.

c) Structural Description
Structural description models the system as components or gates. In this method, each element to be used in the description statement should have been defined under Verilog as a structure. This description is identified by the presence of the gates construct such as **and**, **or**, and **not** in the module.

Verilog HDL programming examples:

1. Write a Verilog program employing the dataflow method of description to provide input through switches and observe the corresponding outputs at the respective LEDs.

   Program:
   ```
   module sample1(out1,out2,in1,in2);
   input in1,in2;
   output out1,out2;
   wire and_out,or_out;
   assign and_out=in1 & in2;
   assign or_out=in1 | in2;
   assign out1=and_out ^ or_out;
   assign out2=~in2;
   endmodule
   ```

2. Write a Verilog program employing the behavioral method of description to provide input through switches and observe the corresponding outputs at the respective LEDs.

   Program:
   ```
   module sample2(out1,out2,in1,in2);
   input in1,in2;
   output out1,out2;
   reg out1,out2;
   initial
   begin
   out1=0;
   out2=0;
   end
   always @(in1,in2)
   begin
   out1=(in1 & in2) ^ (in1 | in2);
   out2=~in2;
   ```

```
end
endmodule
```

3. Write a Verilog program employing the structural method of description to provide input through switches and observe the corresponding outputs at the respective LEDs.

   Program:

```
module sample2(out1,out2,in1,in2);
input in1,in2;
output out1,out2;
wire and_out,or_out;
and gate_and(and_out,in1,in2);
or gate_or(or_out,in1,in2);
xor gate_xor(out1,and_out,or_out);
not gate_not(out2,in2);
endmodule
```

4. Design an ALU to Perform the Following Operations: - And, Or, Xor, Nand, Nor, Xnor, Not. Use A Control Line Input to Switch Between the Operations. Program it Using Dataflow Modelling.

   Program:

```
module alu(i_alu1,i_alu2, i_control, o_aluout);
input i_alu1, i_alu2;
input [2:0] i_control;
output reg o_aluout;
always @(i_control, i_alu1, i_alu2)
begin
case (i_control)
3’b000 : o_aluout <= i_alu1& i_alu2;
3’b001 : o_aluout <= i_alu1 | i_alu2;
3’b010 : o_aluout <= i_alu1 ^ i_alu2;
3’b011 : o_aluout <= ~ (i_alu1 & i_alu2);
3’b100 : o_aluout <= ~( i_alu1| i_alu2);
3’b101 : o_aluout <= ~( i_alu1^ i_alu2);
3’b110 : o_aluout <= ~ i_alu1;
3’b111 : o_aluout <=~ i_alu2;
endcase
end
endmodule
```

5. Write a complete Verilog description for the following operation:
   Four switches on the Basys3 board (sw[0], sw[1], sw[14], sw[15]) will control the pattern of 16 LEDs (from led[0] to led[15]). Here
   • led[7] and led[8] will turn on when all switches are in off condition.
   • led[0] and led[1] will turn on when only sw[0] is on.
   • led[1] and led[2] will turn on when only sw[1] is on.
   • led[13] and led[14] will turn on when only sw[14] is on.
   • led[14] and led[15] will turn on when only sw[15] is on.
   • led[7] and led[8] will turn on for all other combinations of these switches.

Experiment 6
Combinational Circuit Design

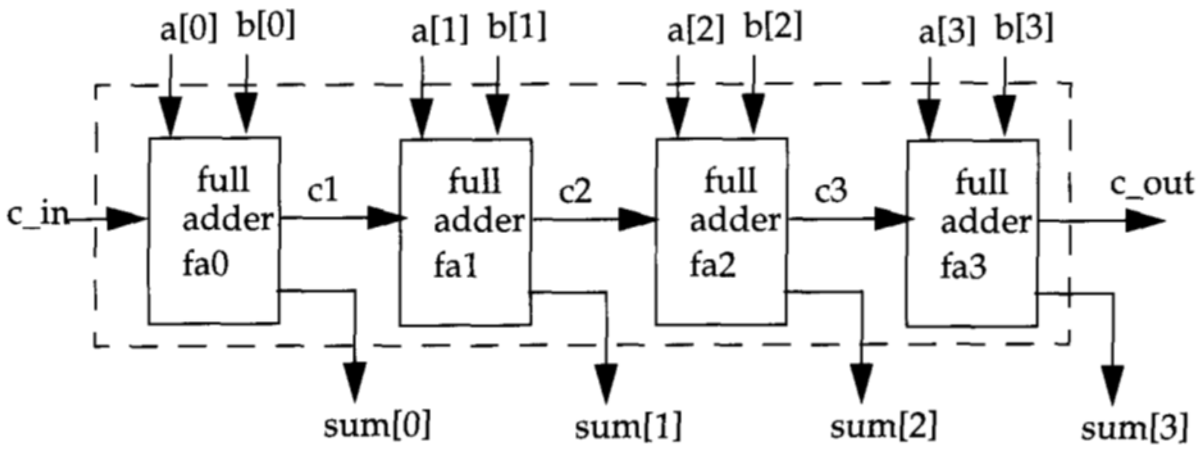
Aim:

- To design and implement the following:
  (a) Adders (b) Multiplexers (c) Decoders (d) Encoders

Part 1: Full adder: Full adder adds two bits and also another bit called carry-in that is propagated from the previous stage. In effect, a full adder adds three bits to give out two outputs – sum and carry out. A,B and C_{in} are the three inputs, out of which C_{in} is to be considered as carry propagated from the previous stage. S and C_{out} are the outputs.

(i) Implement a full adder employing the dataflow method of description.

Part 2: 4-bit Ripple carry adder: A 4-bit ripple carry full adder can be constructed from four 1-bit full adders, as shown in the figure below:



Implement the 4-bit ripple carry adder using 4 full adders employing the structural mode of description.

Part 3: Multiplexer:

Theory:

Multiplexers – A multiplexer or data selector is a logic circuit that accepts several data inputs and allows only one of them at a time to get through the output. The routing of the desired data input to the output is controlled by SELECT inputs. The multiplexer acts like a digitally controlled multi position switch. The digital code applied to the SELECT inputs determines which data input to be switched to the output.

1. Implement a 4:1 MUX employing the behavioral method of description.

Program:

```
module 4to1mux (D, S, Y);
input [3:0] D;
input [1:0] S;
output reg Y;
always @ (S or D)
case (S)
2’b00: Y=D[0];
2’b01: Y=D[1];
2’b10: Y=D[2];
2’b11: Y=D[3];
default: Y=1’bx;
endcase
endmodule
```

2. Implement a 4:1 MUX using conditional assignment.

Part 4: Decoder

Theory:

Decoder – It is a logic circuit that converts an N-bit binary input code into M output lines such that only one line is activated for each one of those possible combinations of inputs.

1. Implement a 3:8 decoder using the case statement.

```
module 3to8decoder (y,x);
input [2:0] x;
output reg [7:0] y;
initial
y=8’b0;
always @(x)
case(x)
3’b000: y=8’b00000001;
3’b001: y=8’b00000010;
3’b010: y=8’b00000100;
3’b011: y=8’b00001000;
3’b100: y=8’b00010000;
3’b101: y=8’b00100000;
```

```
3’b110: y=8’b01000000;
3’b111: y=8’b10000000;
endcase
endmodule
```

Part 5: Priority encoder

Theory:

Priority Encoder-Priority encoders can be used to reduce the number of wires needed in a particular circuits or application that have multiple inputs. The encoder generates an error output when more than one input is high.

1. Implement a 4:2 priority encoder.

```
module priorityencoder (i_priority, o_encoded);
input [3:0] i_priority;
output reg [1:0] o_encoded;
always @(i_priority)
casex (i_priority)
4’b1xxx: o_encoded = 2’b11;
4’b01xx: o_encoded = 2’b10;
4’b001x: o_encoded = 2’b01;
4’b0001: o_encoded = 2‘b00;
default: o_encoded = 2’bx;
endcase
endmodule
```

Experiment 7
Sequential Circuit Design

Aim:

- To design a clock divider circuit to deliver the desired clock frequency.
- To design a sequence code detector using the Mealy and Moore model which will detect the sequence of 1010 in overlapping input sequence.

Part1: Clock divider: A clock signal is needed in order for sequential circuits to function. Usually the clock signal comes from a crystal oscillator on-board. The oscillator used on the Basys3 board usually operates at 100 MHz; however, some peripheral controllers do not need such a high frequency to operate. Hence clock dividers may be required to divide the frequency to the required frequency.

Implement a clock divider which divides the clock frequency to any desired frequency.

Program:

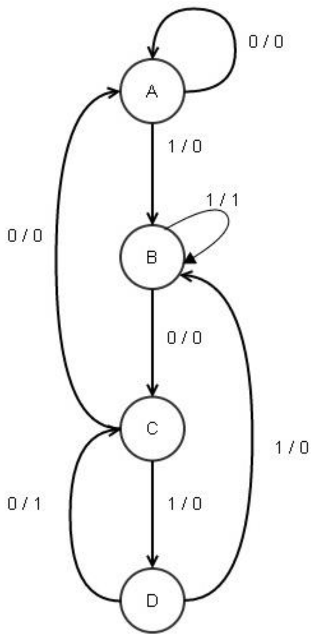
```
module clock_div(
  input clk,
  input rst,
  output reg new_clk
  );

  // The constant that defines the clock speed.
  // Since the system clock is 100MHZ,
  // define_speed = 100MHz/(2*desired_clock_frequency)
  localparam define_speed = 26'd5000000; // Here the desired_clock_frequency is 10 Hz.

  // Count value that counts to define_speed
  reg [25:0] count;

  // Run on the positive edge of the clk and rst signals
  always @ (posedge(clk),posedge(rst))
  begin
    // When rst is high set count and new_clk to 0
    if (rst == 1'b1)
    begin
      count = 26'b0;
      new_clk = 1'b0;
    end
    // When the count has reached the constant
    // reset count and toggle the output clock
    else if (count == define_speed)
    begin
      count = 26'b0;
      new_clk = ~new_clk;
    end
    // increment the clock and keep the output clock
```

```
        // the same when the constant hasn't been reached
        else
        begin
            count = count + 1'b1;
            new_clk = new_clk;
        end
    end
endmodule
```

Part 2: Sequence Detector:

(a) Implement a sequence code detector using the Mealy model to detect the overlapping sequence “1010”.

The state diagram and the state table are as follows:



State	Present State A	Present State B	Input	Next State A_n	Next State B_n	Output
0	0	0	0	0	0	0
1	0	0	1	0	1	0
2	0	1	0	1	1	0
3	0	1	1	0	1	0
6	1	1	0	0	0	0
7	1	1	1	1	0	0
4	1	0	0	1	1	1
5	1	0	1	0	1	0

Program:

```
module mealy1010(sysclk,rst,inp,outp,outclk);
input sysclk,rst,inp;
output reg outp;
output wire outclk;
reg [1:0] state;
reg [1:0] next_state;
parameter S0=0, S1=1, S2=2, S3=3;
reg div_clk;
reg [25:0] delay_count;
wire clk1;

//////////clock division block///////////////////
always @(posedge clk1 or posedge rst)
begin
```

```
   if(rst)
   begin
      delay_count<=26'd0;
      div_clk <= 1'b0;
   end
   else
      if(delay_count==26'd33554432)
      begin
         delay_count<=26'd0;
         div_clk <= ~div_clk;
      end
   else
   begin
      delay_count<=delay_count+1;
   end
end

  always @ (state or inp)
begin
     case(state)
        S0: if(inp)
             next_state<=S1;
           else
             next_state<=S0;
        S1: if(inp)
             next_state<=S1;
           else
             next_state<=S2;
        S2: if(inp)
             next_state<=S3;
           else
             next_state<=S0;
        S3: if(inp)
             next_state<=S1;
           else
             next_state<=S2;
     endcase
     end

always @(posedge div_clk or posedge rst)
begin
      if(rst)
         state<=S0;
      else
         state <= next_state;
      end
end
```

```
//
always @(state,inp)
begin
   case(state)
        S0: if(inp)
               outp<=0;
           else
               outp<=0;
        S1: if(inp)
               outp<=0;
            else
               outp<=0;
        S2: if(inp)
               outp<=0;
            else
               outp<=0;
        S3: if(inp)
               outp<=0;
            else
               outp<=1;
   endcase
end

    clk_wiz_0 inst
    (
    // Clock out ports  //instantiation of clock IP from the block design
    .clk_out1(clk1),
   // Clock in ports
    .clk_in1(sysclk)
    );

    assign outclk = div_clk;

endmodule
```

(b) Implement a sequence code detector using the Moore model to detect the overlapping sequence “1010”.