

B.M.S. College of Engineering
(Autonomous Institution affiliated to VTU, Belagavi)

Department of Computer Science and Engineering



AAT

**Verilog Laboratory
Report**

23CS3PCLOD

(September 2025-January 2026)

B.M.S. College of Engineering

Department of Computer Science and Engineering



Laboratory Certificate

This is to certify that Chetan Barakki Satish Kumar, Chinmay G Hegde, D Murali Satya Suhas, Darshan R Palrecha satisfactorily completed the course of Experiments in Practical Logic Design (Verilog) prescribed by the Department during the odd semester 2025-26.

Name of the Candidate: Chetan Barakki Satish Kumar, Chinmay G Hegde, D Murali Satya Suhas, Darshan R Palrecha

USN No.: **1BM24CS081, 1BM24CS083, 1BM24CS086, 1BM24CS091**

Semester: **III**

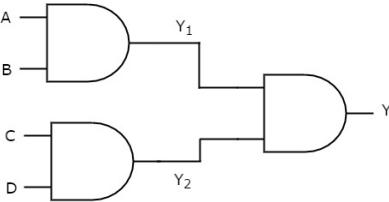
Section: **G**

Marks	
Max. Marks	Obtained
10	
Marks in Words	

Signature of the staff in-charge

Head of the Department

Date:

Serial No.	Title	Page No
	CYCLE I Structural Modelling	
1.	<p>Write HDL implementation for the following Logic</p> <p>a. AND/OR/NOT</p> <p>Simulate the same using structural model and depict the timing diagram for valid inputs.</p>	4
2.	<p>Write HDL implementation for the following Logic</p> <p>a. NAND/NOR</p> <p>Simulate the same using structural model and depict the timing diagram for valid inputs.</p>	6
3.	<p>Write HDL implementation for the following AND-OR Combinational Logic</p>  <p>Simulate the same using structural model and depict the timing diagram for valid inputs.</p>	8
4.	Write HDL implementation for a 4:1 Multiplexer. Simulate the same using structural model and depict the timing diagram for valid inputs.	10
5.	Write HDL implementation for a 2-to-4 decoder. Simulate the same using structural model and depict the timing diagram for valid inputs.	12
6.	Write HDL implementation for a 4-to-2 encoder. Simulate the same using structural model and depict the timing diagram for valid inputs.	14
	CYCLE II Behavior Modeling	
7.	Write HDL implementation for a RS flip-flop using behavioral model. Simulate the same using structural model and depict the timing diagram for valid inputs.	16
8.	Write HDL implementation for a JK flip-flop using behavioral model. Simulate the same using structural model and depict the timing diagram for valid inputs.	18
9.	Write HDL implementation for a 3-bit up-counter using behavioral model. Simulate the same using structural model and depict the timing diagram for valid inputs.	20
	CYCLE III Dataflow Modeling	
10.	Write HDL implementation for AND/OR/NOT gates using data flow model. Simulate the same using structural model and depict the timing diagram for valid inputs	22

CYCLE 1: STRUCTURAL MODELLING

Experiment 1: Write HDL implementation for the following Logic AND/OR/NOT. Simulate the same using structural model and depict the timing diagram for valid inputs.

```
module and_gate_struct (
    input a,
    input b,
    output y
);
    and (y, a, b);
endmodule

module or_gate_struct (
    input a,
    input b,
    output y
);
    or (y, a, b);
endmodule

module not_gate_struct (
    input a,
    output y
);
    not (y, a);
endmodule

// testbench
module struct_basic_gates_tb;

    reg a, b;
    wire y_and, y_or, y_not;

    and_gate_struct op1 (.a(a), .b(b), .y(y_and));
    or_gate_struct op2 (.a(a), .b(b), .y(y_or));
    not_gate_struct op3 (.a(a), .y(y_not));

    initial begin
        $dumpfile("struct_basic_gates.vcd");
```

```

$dumpvars(0, struct_basic_gates_tb);
a = 0; b = 0;
#10 a = 0; b = 1;
#10 a = 1; b = 0;
#10 a = 1; b = 1;
#10 $finish;
end
endmodule

```

Compilation, Execution and Result of Simulation

The screenshot displays the workflow for a Verilog simulation. On the left, the code editor shows the source files: 'struct_basic_gates.v' and 'struct_basic_gates_tb.v'. The terminal window in the center shows the command-line interface for compilation and execution. The right window is a waveform viewer (GTKWave) showing the timing of signals over a 40-second period.

struct_basic_gates.v:

```

module and_gate_struct (
    input a,
    input b,
    output y
);
    and (y, a, b);
endmodule

module or_gate_struct (
    input a,
    input b,
    output y
);
    or (y, a, b);
endmodule

module not_gate_struct (
    input a,
    output y
);
    not (y, a);
endmodule

```

struct_basic_gates_tb.v:

```

module struct_basic_gates_tb;
    reg a, b;
    wire y_and, y_or, y_not;

    and_gate_struct op1 (.a(a), .b(b), .y(y_and));
    or_gate_struct op2 (.a(a), .b(b), .y(y_or));
    not_gate_struct op3 (.a(a), .y(y_not));

    initial begin
        $dumpfile("struct_basic_gates.vcd");
        $dumpvars(0, struct_basic_gates_tb);
        a = 0; b = 0;
        #10 a = 0; b = 1;
        #10 a = 1; b = 0;
        #10 a = 1; b = 1;
        #10 $finish;
    end
endmodule

```

Terminal Output:

```

PS C:\Users\Admin\Desktop\ld-aat> iverilog -o struct_basic_gates.out struct_basic_gates.v struct_basic_gates_tb.v
PS C:\Users\Admin\Desktop\ld-aat> vvp struct_basic_gates.out
VCD info: dumpfile struct_basic_gates.vcd opened for output.
struct_basic_gates_tb.v:17: $finish called at 40 (1s)
PS C:\Users\Admin\Desktop\ld-aat> gtkwave struct_basic_gates.vcd

```

GTKWave Analyzer v3.3.100 (w)1999-2019 BSI

[0] start time.
[40] end time.

The waveform viewer shows the following signals over time (0 to 40 seconds):

- Time:** The current time is 0 sec.
- Signals:** The signals listed are Time, a, y_not, y_and, a, b, y_or.
- Waves:** The waveforms show the state of each signal over time. The 'a' signal toggles between 0 and 1 at specific intervals. The 'b' signal is constant at 0. The 'y_and' signal is 1 whenever 'a' and 'b' are both 1. The 'y_or' signal is 1 whenever either 'a' or 'b' is 1. The 'y_not' signal is the inverse of 'a'.

Experiment 2: Write HDL implementation for the following Logic NAND/NOR. Simulate the same using structural model and depict the timing diagram for valid inputs.

```
module nand_struct (
    input A,
    input B,
    output Y
);
    nand (Y, A, B);
endmodule

module nor_struct (
    input A,
    input B,
    output Y
);
    nor (Y, A, B);
endmodule

// testbench
module nand_nor_tb;
    reg A, B;
    wire Y_nand_struct;
    wire Y_nor_struct;

    nand_struct U2 (.A(A), .B(B), .Y(Y_nand_struct));
    nor_struct U4 (.A(A), .B(B), .Y(Y_nor_struct));

    initial begin
        $dumpfile("nand_nor.vcd");
        $dumpvars(0, nand_nor_tb);

        // Apply all input combinations
        A = 0; B = 0;
        #10 A = 0; B = 1;
        #10 A = 1; B = 0;
        #10 A = 1; B = 1;

        #10 $finish;
    end
endmodule
```

Compilation, Execution and Result of Simulation

The screenshot displays three windows related to the simulation of a Nand NOR module:

- nand_nor.v**: Verilog source code for the Nand and NOR structures.
- Windows PowerShell**: Command-line interface showing the compilation process using iverilog and vvp, followed by gtkwave.
- GTKWave - nand_nor.vcd**: Waveform viewer showing the simulation results for signals A, B, Y_nand_struct, and Y_nor_struct over a 40-second period.

Verilog Source Code (nand_nor.v):

```
module nand_struct (
    input A,
    input B,
    output Y
);
    nand (Y, A, B);
endmodule

module nor_struct (
    input A,
    input B,
    output Y
);
    nor (Y, A, B);
endmodule
```

Simulation Results (nand_nor_tb.v):

```
module nand_nor_tb;
    reg A, B;
    wire Y_nand_struct;
    wire Y_nor_struct;

    nand_struct U2 (.A(A), .B(B), .Y(Y_nand_struct));
    nor_struct U4 (.A(A), .B(B), .Y(Y_nor_struct));

    initial begin
        $dumpfile("nand_nor.vcd");
        $dumpvars(0, nand_nor_tb);

        // Apply all input combinations
        A = 0; B = 0;
        #10 A = 0; B = 1;
        #10 A = 1; B = 0;
        #10 A = 1; B = 1;

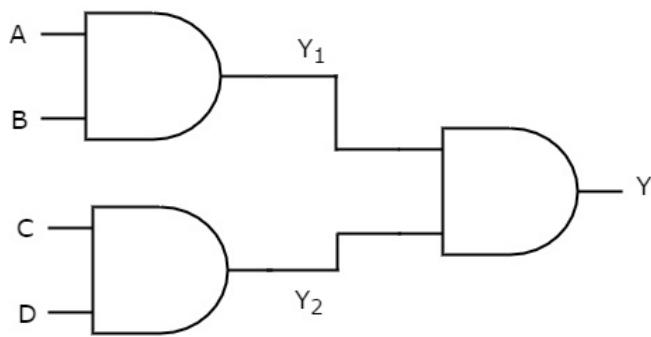
        #10 $finish;
    end
endmodule
```

Simulation Waveform (GTKWave - nand_nor.vcd):

The waveform viewer shows four signals over time (0 to 40 seconds):

- A**: A digital signal starting at 0, transitioning to 1 at 10s, and back to 0 at 20s.
- B**: A digital signal starting at 0, transitioning to 1 at 20s, and back to 0 at 30s.
- Y_nand_struct**: The output of the Nand structure. It is high (1) from 0s to 10s, low (0) from 10s to 20s, high (1) from 20s to 30s, and low (0) from 30s to 40s.
- Y_nor_struct**: The output of the NOR structure. It is low (0) from 0s to 10s, high (1) from 10s to 20s, low (0) from 20s to 30s, and high (1) from 30s to 40s.

Experiment 3: Write HDL implementation for the following AND-OR Combinational Logic $Y = (AB)(CD)$. Simulate the same using structural model and depict the timing diagram for valid inputs.



```

module comb_struct (
    input A,
    input B,
    input C,
    input D,
    output Y
);

    wire Y1, Y2;

    and (Y1, A, B);
    and (Y2, C, D);
    or (Y, Y1, Y2);

endmodule

// testbench
module comb_tb;

    reg A, B, C, D;
    wire Y_df, Y_struct;

    // Instantiate structural model
    comb_struct U2 (
        .A(A), .B(B), .C(C), .D(D),
        .Y(Y_struct)
    );

    initial begin
        $dumpfile("comb.vcd");
        $dumpvars(0, comb_tb);

        // Apply valid input combinations
        A = 0; B = 0; C = 0; D = 0;
    end

```

```

#10 A = 1; B = 1; C = 0; D = 0;
#10 A = 0; B = 0; C = 1; D = 1;
#10 A = 1; B = 1; C = 1; D = 1;
#10 A = 1; B = 0; C = 1; D = 0;
#10 $finish;
end
endmodule

```

Compilation, Execution and Result of Simulation

The screenshot displays four windows illustrating the workflow:

- combination.v**: The Verilog source code for the module. It defines a structural model named `comb_struct` with inputs A, B, C, D and output Y. It also includes an initial block with stimulus and a `#10 $finish;` statement.
- Windows PowerShell**: The terminal window shows the command-line steps: `iverilog -o combination.out combination.v combination_tb.v`, `vvp combination.out`, and `gtkwave comb.vcd`. The output indicates a dumpfile named `comb.vcd` was opened for output, and `$finish` was called at 50 seconds.
- combination_tb.v**: The testbench Verilog code. It instantiates the `comb_struct` module, sets initial values for A, B, C, D, and performs a `$dumpfile` and `$dumpvars` operation to capture waveforms. It also applies valid input combinations and ends with `#10 $finish;`.
- GTKWave - comb.vcd**: The waveform viewer window titled "GTKWave - comb.vcd". It shows four signals over time (0 to 50 seconds): A, B, C, and D. All four signals are high (1) from 10 to 20 seconds. Signal Y_struct is low (0) from 0 to 10 seconds and high (1) from 20 to 50 seconds.

Experiment 4: Write HDL implementation for a 4:1 Multiplexer. Simulate the same using

```
module mux_4to1(
    input I0, I1, I2, I3,
    input S1, S0,
    output Y
);
    wire nS1, nS0;
    wire w0, w1, w2, w3;

    not (nS1, S1);
    not (nS0, S0);
    and (w0, I0, nS1, nS0);
    and (w1, I1, nS1, S0);
    and (w2, I2, S1, nS0);
    and (w3, I3, S1, S0);

    or (Y, w0, w1, w2, w3);
endmodule

// testbench
module mux_4to1_tb;
    reg I0, I1, I2, I3;
    reg S1, S0;
    wire Y;

    mux_4to1 op (
        .I0(I0), .I1(I1), .I2(I2), .I3(I3),
        .S1(S1), .S0(S0),
        .Y(Y)
    );
    initial begin
        $dumpfile("mux_4to1.vcd");
        $dumpvars(0, mux_4to1_tb);
        I0 = 0; I1 = 1; I2 = 0; I3 = 1;
        S1 = 0; S0 = 0; // Select I0
        #10 S1 = 0; S0 = 1; // Select I1
        #10 S1 = 1; S0 = 0; // Select I2
        #10 S1 = 1; S0 = 1; // Select I3

        #10 $finish;
    end
endmodule
```

Compilation, Execution and Result of Simulation

The screenshot shows a desktop interface with several windows open:

- A Notepad window titled "mux_4to1.v" containing Verilog code for a 4-to-1 multiplexer.
- Two "Windows PowerShell" windows showing the command-line process: iverilog -o mux_4to1.out mux_4to1.v mux_4to1_tb.v, followed by vvp mux_4to1.out. The output indicates a VCD dumpfile was opened.
- A "GTKWave Analyzer v3.3.100 (w)1999-2019 BSI" window titled "GTKWave - mux_4to1.vcd". It displays waveforms for signals I0, I1, I2, I3, S0, S1, and Y over a 40-second period. The S0 signal is asserted at time 20 sec, and the Y signal changes from 0 to 1 at time 20 sec.
- A Notepad window titled "mux_4to1_tb.v" containing the testbench code for the mux_4to1 module.

Verilog code in mux_4to1.v:

```
module mux_4to1(
    input I0, I1, I2, I3,
    input S1, S0,
    output Y
);

    wire nS1, nS0;
    wire w0, w1, w2, w3;

    not (nS1, S1);
    not (nS0, S0);

    and (w0, I0, nS1, nS0);
    and (w1, I1, nS1, S0);
    and (w2, I2, S1, nS0);
    and (w3, I3, S1, S0);

    or (Y, w0, w1, w2, w3);

endmodule
```

Verilog code in mux_4to1_tb.v:

```
module mux_4to1_tb;

    reg I0, I1, I2, I3;
    reg S1, S0;
    wire Y;

    mux_4to1 op (
        .I0(I0), .I1(I1), .I2(I2), .I3(I3),
        .S1(S1), .S0(S0),
        .Y(Y)
    );

    initial begin
        $dumpfile("mux_4to1.vcd");
        $dumpvars(0, mux_4to1_tb);
        I0 = 0; I1 = 1; I2 = 0; I3 = 1;
        S1 = 0; S0 = 0; // Select I0
        #10 S1 = 0; S0 = 1; // Select I1
        #10 S1 = 1; S0 = 0; // Select I2
        #10 S1 = 1; S0 = 1; // Select I3
        #10 $finish;
    end

endmodule
```

Experiment 5: Write HDL implementation for a 2-to-4 decoder. Simulate the same using structural model and depict the timing diagram for valid inputs.

```
module decoder_2to4 (
    input D0, input D1,
    output Y0, output Y1,
    output Y2, output Y3
);

    wire D0_bar, D1_bar;

    not (D0_bar, D0);
    not (D1_bar, D1);

    and (Y0, D0_bar, D1_bar);
    and (Y1, D0_bar, D1);
    and (Y2, D0, D1_bar);
    and (Y3, D0, D1);

endmodule

// testbench
module decoder_2to4_tb;

    reg D0, D1;
    wire Y0, Y1, Y2, Y3;

    decoder_2to4 dut (
        .D0(D0), .D1(D1),
        .Y0(Y0), .Y1(Y1), .Y2(Y2), .Y3(Y3)
    );

    initial begin
        $dumpfile("decoder_2to4.vcd");
        $dumpvars(0, decoder_2to4_tb);

        D0 = 0; D1 = 0; #10;
        D0 = 0; D1 = 1; #10;
        D0 = 1; D1 = 0; #10;
        D0 = 1; D1 = 1; #10;

        $finish;
    end
endmodule
```

Compilation, Execution and Result of Simulation

The terminal window shows the following command and its output:

```
PS C:\Users\Admin\Desktop\ld-aat\decoder_2to4> iverilog -o decoder_2to4.out decoder_2to4.v decoder_2to4_tb.v
PS C:\Users\Admin\Desktop\ld-aat\decoder_2to4> vvp decoder_2to4.out
VCD info: dumpfile decoder_2to4.vcd opened for output.
decoder_2to4_tb.v:20: $finish called at 40 (1s)
PS C:\Users\Admin\Desktop\ld-aat\decoder_2to4> gtkwave decoder_2to4.vcd
```

GTKWave Analyzer v3.3.100 (w)1999-2019 BSI

[0] start time.
[40] end time.

The screenshot also shows two code editors. The top editor contains the Verilog module `decoder_2to4.v`:

```
module decoder_2to4 (
    input D0, input D1,
    output Y0, output Y1,
    output Y2, output Y3
);
    wire D0_bar, D1_bar;
    not (D0_bar, D0);
    not (D1_bar, D1);
    and (Y0, D0_bar, D1_bar);
    and (Y1, D0_bar, D1);
    and (Y2, D0, D1_bar);
    and (Y3, D0, D1);
endmodule
```

The bottom editor contains the Verilog testbench `decoder_2to4_tb.v`:

```
module decoder_2to4_tb;
    reg D0, D1;
    wire Y0, Y1, Y2, Y3;

    decoder_2to4 dut (
        .D0(D0), .D1(D1),
        .Y0(Y0), .Y1(Y1), .Y2(Y2), .Y3(Y3)
    );

    initial begin
        $dumpfile("decoder_2to4.vcd");
        $dumpvars(0, decoder_2to4_tb);

        D0 = 0; D1 = 0; #10;
        D0 = 0; D1 = 1; #10;
        D0 = 1; D1 = 0; #10;
        D0 = 1; D1 = 1; #10;

        $finish;
    end

endmodule
```

A GTKWave window is open, showing waveforms for signals D0, D1, Y0, Y1, Y2, and Y3 over a 40-second period. The waveforms show the expected output of a 2-to-4 decoder for various input combinations.

Experiment 6: Write HDL implementation for a 4-to-2 encoder. Simulate the same using structural model and depict the timing diagram for valid inputs.

```
module encoder_4to2 (
    input D0, D1, D2, D3,
    output Y0, Y1
);

    or (Y0, D1, D3);
    or (Y1, D2, D3);

endmodule

// testbench
module encoder_4to2_tb;

    reg D0, D1, D2, D3;
    wire Y0, Y1;

    encoder_4to2 op (
        .D0(D0), .D1(D1), .D2(D2), .D3(D3),
        .Y0(Y0), .Y1(Y1)
    );

    initial begin
        $dumpfile("encoder_4to2.vcd");
        $dumpvars(0, encoder_4to2_tb);
        D3=0; D2=0; D1=0; D0=1; // 00
        #10 D3=0; D2=0; D1=1; D0=0; // 01
        #10 D3=0; D2=1; D1=0; D0=0; // 10
        #10 D3=1; D2=0; D1=0; D0=0; // 11
        #10 $finish;
    end

endmodule
```

Compilation, Execution and Result of Simulation

The terminal window shows the following command and its output:

```
PS C:\Users\Admin\Desktop\ld-aat\encoder_4to2> iverilog -o encoder_4to2.out encoder_4to2.v encoder_4to2_tb.v
PS C:\Users\Admin\Desktop\ld-aat\encoder_4to2> vvp encoder_4to2.out
VCD info: dumpfile encoder_4to2.vcd opened for output.
encoder_4to2_tb.v:18: $finish called at 40 (1s)
PS C:\Users\Admin\Desktop\ld-aat\encoder_4to2> gtkwave encoder_4to2.vcd
```

The terminal also displays the GTKWave Analyzer version and usage information:

```
GTKWave Analyzer v3.3.100 (w)1999-2019 BSI
[0] start time.
[40] end time.
```

The screenshot also includes a screenshot of the GTKWave application window titled "GTKWave - encoder_4to2.vcd". It shows a waveform for four inputs (D0, D1, D2, D3) and two outputs (Y0, Y1) over a 40-second period. The signals are represented by colored lines: D0 (green), D1 (blue), D2 (red), D3 (yellow), Y0 (orange), and Y1 (purple). The waveforms show the logic levels changing over time, corresponding to the simulation results shown in the terminal.

Experiment 7: Write HDL implementation for a RS flip-flop using behavioral model. Simulate the same using structural model and depict the timing diagram for valid inputs.

```
module rs_ff_beh (
    input S,
    input R,
    output reg Q,
    output reg Qbar
);
    always @ (S or R) begin
        if (S == 1 && R == 0) begin
            Q = 1;
            Qbar = 0;
        end
        else if (S == 0 && R == 1) begin
            Q = 0;
            Qbar = 1;
        end
        else if (S == 0 && R == 0) begin
            Q = Q;      // Hold state
            Qbar = Qbar;
        end
        // S = 1, R = 1 is invalid: not modeled
    end
endmodule

// testbench
module beh_rsff_tb;
    reg S, R;
    wire Q, Qbar;
    rs_ff_beh op (.S(S), .R(R), .Q(Q), .Qbar(Qbar));
    initial begin
        $dumpfile("beh_rsff.vcd");
        $dumpvars(0, beh_rsff_tb);
        S = 0; R = 0;      // Hold
        #10 S = 1; R = 0; // Set
        #10 S = 0; R = 0; // Hold
        #10 S = 0; R = 1; // Reset
        #10 S = 0; R = 0; // Hold
        #10 $finish;
    end
endmodule
```

Compilation, Execution and Result of Simulation

The terminal window shows the following command sequence:

```
PS C:\Users\Admin\Desktop\ld-aat\rsff> iverilog -o beh_rsff.out beh_rsff.v beh_rsff_tb.v
PS C:\Users\Admin\Desktop\ld-aat\rsff> vvp beh_rsff.out
VCD info: dumpfile beh_rsff.vcd opened for output.
beh_rsff_tb.v:18: $finish called at 50 (1s)
PS C:\Users\Admin\Desktop\ld-aat\rsff> gtkwave beh_rsff.vcd
```

The terminal also displays the GTKWave Analyzer version information and timing markers:

GTKWave Analyzer v3.3.100 (w)1999-2019 BSI
[0] start time.
[50] end time.

The screenshot also includes a screenshot of the GTKWave interface showing the waveform for signals R, S, Q, and Qbar over a 50-second period. The signals are defined in the beh_rsff_tb module:

```
module beh_rsff_tb;
    reg S, R;
    wire Q, Qbar;
    rs_ff_beh op (.S(S), .R(R),
                  .Q(Q), .Qbar(Qbar));
    initial begin
        $dumpfile("beh_rsff.vcd");
        $dumpvars(0, beh_rsff_tb);
        S = 0; R = 0; // Hold
        #10 S = 1; R = 0; // Set
        #10 S = 0; R = 0; // Hold
        #10 S = 0; R = 1; // Reset
        #10 S = 0; R = 0; // Hold
        #10 $finish;
    end
endmodule
```

Experiment 8: Write HDL implementation for a JK flip-flop using behavioral model. Simulate the same using structural model and depict the timing diagram for valid inputs.

```
module jk_ff_beh ( input J, input K, input clk, output reg Q, output reg Qbar );
    initial begin
        Q = 0;
        Qbar = 1;
    end

    always @(posedge clk) begin
        Q = 1;
        Qbar = 0;
        case ({J, K})
            2'b00: begin
                Q <= Q;      // Hold
                Qbar <= Qbar;
            end
            2'b01: begin
                Q <= 0;      // Reset
                Qbar <= 1;
            end
            2'b10: begin
                Q <= 1;      // Set
                Qbar <= 0;
            end
            2'b11: begin
                Q <= ~Q;     // Toggle
                Qbar <= ~Qbar;
            end
        endcase
    end
endmodule

// testbench
module beh_jkff_tb;

    reg J, K, clk;
    wire Q, Qbar;

    jk_ff_beh op (.J(J), .K(K), .clk(clk), .Q(Q), .Qbar(Qbar));

    // Clock generation
```

```

always #5 clk = ~clk;

initial begin
    $dumpfile("beh_jkff.vcd");
    $dumpvars(0, beh_jkff_tb);
    clk = 0;
    J = 0; K = 0; // Hold
    #10 J = 0; K = 1; // Reset
    #10 J = 1; K = 0; // Set
    #10 J = 1; K = 1; // Toggle
    #20 $finish;
end
endmodule

```

Compilation, Execution and Result of Simulation

The screenshot displays three windows illustrating the workflow:

- Top Window (Windows PowerShell):** Shows the command-line interface used for compilation and simulation. It includes:
 - Compiling Verilog code: PS C:\Users\Admin\Desktop\ld-aat\jkff> iverilog -o beh_jkff.out beh_jkff.v beh_jkff_tb.v
 - Running the testbench: PS C:\Users\Admin\Desktop\ld-aat\jkff> vvp beh_jkff.out
 - Opening the dumpfile: VCD info: dumpfile beh_jkff.vcd opened for output.
 - Termination: beh_jkff_tb.v:21: \$finish called at 50 (1s)
 - Opening the simulation waveforms: PS C:\Users\Admin\Desktop\ld-aat\jkff> gtkwave beh_jkff.vcd
- Middle Window (GTKWave Analyzer v3.3.100):** Displays the simulation waveforms for the JK flip-flop. The signals shown are CLK, J, K, Q, and Qbar. The simulation spans from 0 to 50 seconds. The CLK signal is a square wave. The J and K signals are held at 0. The Q signal toggles between 1 and 0 at each clock edge. The Qbar signal is the complement of the Q signal.
- Bottom Window (beh_jkff.v and beh_jkff_tb.v):** Shows the Verilog source code for the behavioral model and its testbench. The behavioral module (beh_jkff.v) defines a JK flip-flop with logic for Hold, Reset, Set, and Toggle modes. The testbench (beh_jkff_tb.v) initializes the environment, sets up stimulus, and performs a \$finish operation after 20 time units.

Experiment 9: Write HDL implementation for a 3-bit up-counter using behavioral model. Simulate the same using structural model and depict the timing diagram for valid inputs.

```
module up_counter_3bit_beh (
    input clk,
    input reset,
    output reg [2:0] Q
);

initial begin
    Q = 3'b000;
end

always @(posedge clk or posedge reset) begin
    if (reset)
        Q <= 3'b000;
    else
        Q <= Q + 1'b1;
end

endmodule

// testbench
module beh_up_counter_tb;

reg clk;
wire [2:0] Q;

up_counter_3bit_beh op (.clk(clk), .Q(Q));

// Clock generation
always #5 clk = ~clk;

initial begin
    $dumpfile("beh_up_counter.vcd");
    $dumpvars(0,beh_up_counter_tb);
    clk = 0;
    #80 $finish;
end

endmodule
```

Compilation, Execution and Result of Simulation

The screenshot shows a development environment with three main windows:

- Top Left Window:** A code editor showing the Verilog source code for `beh_up_counter.v`. It contains a module definition for `up_counter_3bit_beh` with an initial state of `Q = 3'b000` and a logic block for incrementing `Q` on each rising edge of `clk`.
- Top Middle Window:** A Windows PowerShell window showing the compilation process. It runs `iverilog -o beh_up_counter.out beh_up_counter.v beh_up_counter_tb.v`, then `vvp beh_up_counter.out`, and finally `gtkwave beh_up_counter.vcd`. The output includes VCD dumpfile information and a `$finish` call at 80 seconds.
- Bottom Right Window:** A GTKWave Analyzer window titled "GTKWave - beh_up_counter.vcd". It displays a waveform for the signal `Q[2:0]` over time from 0 to 80 seconds. The waveform shows a binary counter starting at 000 and incrementing by 1 every second. The clock signal `clk` is also shown, with its period being approximately 1 second.
- Bottom Left Window:** A code editor showing the Verilog source code for `beh_up_counter_tb.v`. It defines a testbench module `beh_up_counter_tb` with a clock generation block and a simulation setup block that calls `$dumpvars` and `$finish` after 80 seconds.

Experiment 10: Write HDL implementation for AND/OR/NOT gates using data flow model. Simulate the same using structural model and depict the timing diagram for valid inputs.

```
module and_gate_df (
    input a,
    input b,
    output y
);
    assign y = a & b;
endmodule

module or_gate_df (
    input a,
    input b,
    output y
);
    assign y = a | b;
endmodule

module not_gate_df (
    input a,
    output y
);
    assign y = ~a;
endmodule

// testbench
module df_basic_gates_tb;
    reg a, b;
    wire y_and, y_or, y_not;

    // Instantiate structural models
    and_gate_df op1 (.a(a), .b(b), .y(y_and));
    or_gate_df op2 (.a(a), .b(b), .y(y_or));
    not_gate_df op3 (.a(a), .y(y_not));

    initial begin
        $dumpfile("df_basic_gates.vcd");
        $dumpvars(0, df_basic_gates_tb);
        a = 0; b = 0;
    end

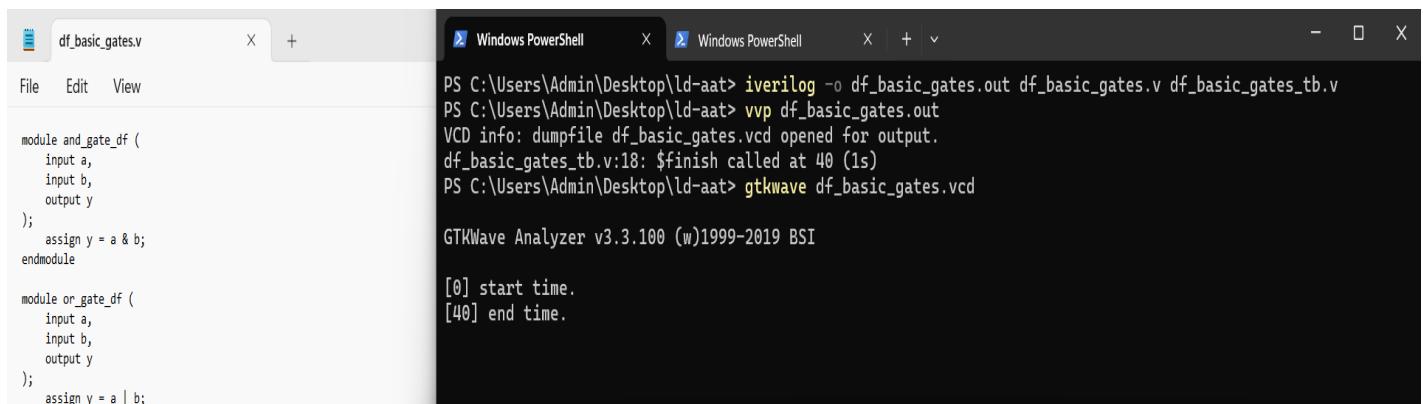
```

```

#10 a = 0; b = 1;
#10 a = 1; b = 0;
#10 a = 1; b = 1;
#10 $finish;
end
endmodule

```

Compilation, Execution and Result of Simulation



The terminal window shows the following command sequence:

```

PS C:\Users\Admin\Desktop\ld-aat> iverilog -o df_basic_gates.out df_basic_gates.v df_basic_gates_tb.v
PS C:\Users\Admin\Desktop\ld-aat> vvp df_basic_gates.out
VCD info: dumpfile df_basic_gates.vcd opened for output.
df_basic_gates_tb.v:18: $finish called at 40 (1s)
PS C:\Users\Admin\Desktop\ld-aat> gtkwave df_basic_gates.vcd

```

GTKWave Analyzer v3.3.100 (w)1999-2019 BSI

[0] start time.
[40] end time.

GTKWave - df_basic_gates.vcd

The GTKWave window displays waveforms for signals over a 40-second period. The signals shown are Time, a, y_not, a, b, y_and, a, b, y_or. The waveforms show the logic levels of the inputs and outputs of the three basic gates (NOT, AND, OR) over time.

df_basic_gates.v

```

module and_gate_df (
    input a,
    input b,
    output y
);
    assign y = a & b;
endmodule

module or_gate_df (
    input a,
    input b,
    output y
);
    assign y = a | b;
endmodule

module not_gate_df (
    input a,
    output y
);
    assign y = ~a;
endmodule

```

df_basic_gates_tb.v

```

module df_basic_gates_tb;

reg a, b;
wire y_and, y_or, y_not;

// Instantiate structural models
and_gate_df op1 (.a(a), .b(b), .y(y_and));
or_gate_df op2 (.a(a), .b(b), .y(y_or));
not_gate_df op3 (.a(a), .y(y_not));

initial begin
    $dumpfile("df_basic_gates.vcd");
    $dumppars(0, df_basic_gates_tb);
    a = 0; b = 0;
    #10 a = 0; b = 1;
    #10 a = 1; b = 0;
    #10 a = 1; b = 1;
    #10 $finish;
end
endmodule

```