MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

B. Tech. IN ELECTRONICS AND COMMUNICATION ENGINEERING- VLSI

COURSE PLAN: LABORATORY COURSE

Department:	**ECE (VLSI)**	
Course Name & code:	**FPGA-based system design using Verilog lab & ECE2243**	
Semester & branch:	**IV**	**Electronics and Communication Engineering (VLSI)**
Name of the faculty:	**CSR, MK, SKT, BM**	

Instructions to the students

1. Students should carry the lab manual and observation book to every lab session.
2. Be on time and follow the institution's dress code.
3. You should try to analyze and understand the solved problems and then try to solve all the exercise problems of the experiment in the lab.
4. Maintaining an observation copy is compulsory for all, where the results of all the problems solved in the lab should be appropriately noted down.
5. You must get your results verified and observation copies checked by the instructor before leaving the lab for the day.
6. You should maintain a folder of all the programs you do in the lab on the computer you used by your registration number. You are also advised to keep a backup of it.
7. Use of external storage media during lab is not allowed.
8. Maintain the timings and the discipline of the lab.

Evaluation plan

- Internal assessment marks: 60% (60 marks)
  - Continuous evaluation component (for each experiment): 10 marks
    - Assessment is based on preparation, conduction of each experiment, exercise problems, maintaining the observation note, and answering the questions related to the experiment.
    - Total marks of the 9 regular experiments scaled to 50 marks + mini project carries 10 marks

Note: Follow the code of conduct (punctual, disciplined, and sincere)

- End semester exam assessment: 40% (40 marks)
  - Write up: 12 marks
  - Conduction: 12 marks
  - Results: 8 marks
  - Viva-voce: 8 marks

Experiment No. 2

Verilog Dataflow Modeling

OBJECTIVE: To understand the concepts related to dataflow modeling style and write Verilog Programs on it.

THEORY: *Modules* are the basic building blocks for modeling. The module is the principal design entity in Verilog.

Module Declaration:

The first line of a module declaration specifies the *module name* and *port list* (arguments). The next few lines specify the *i/o type* (input, output or inout) and *width* of each port.

Syntax:

```
module module_name (port_list);
       input [msb:lsb] input_port_list;
       output [msb:lsb] output_port_list;
       inout [msb:lsb] inout_port_list;
               ... statements...
  endmodule
```

Dataflow modeling:

The data-flow model uses concurrent signal assignment statements (The order of assign statements does not matter). Dataflow modeling uses continuous assignment statements with keyword `assign`.

`assign Y =` *Boolean Expression using variables and operators***.**

A dataflow description is based on function rather than structure and uses several bit-wise operators.

Bitwise Verilog Operator	Symbol
NOT	~
AND	&
OR	|
XOR	^
XNOR	^~ or ~^

EXAMPLE 2.1: Write a dataflow Verilog code to realize the given logic function in SOP form y(a, b, c) = Σ(1, 4, 7) and verify the design by simulation.

Solution: y(a, b, c) = $\bar{a}.\bar{b}.c + a.\bar{b}.\bar{c} + a.b.c$

Verilog Code:

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

Simulation Results:

Input: a = 1, b = 1, c=1

Output: y = 1

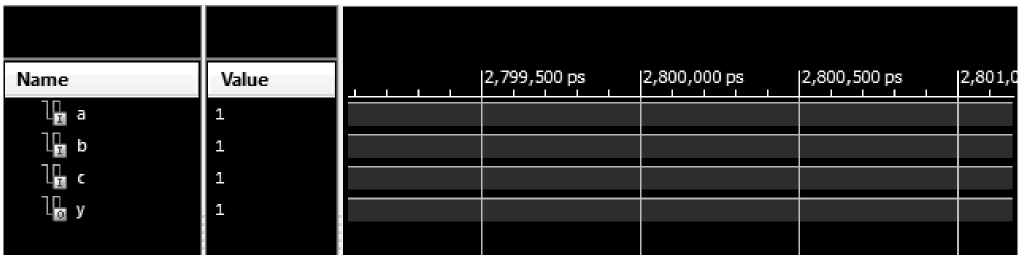


Fig. 2.1: Simulation results of example 2.1

EXAMPLE 2.2: Write a dataflow Verilog code for a 2-to-4 decoder with active low enable input and active low outputs and verify the design by simulation.

Solution:

Truth Table of 2-to-4 decoder with active low enable input and active low outputs.

Input			Output			
E	**B(MSB)**	**A**	**D3(MSB)**	**D2**	**D1**	**D0**
1	x	x	1	1	1	1
0	0	0	1	1	1	0
0	1	0	1	1	0	1
0	0	1	1	0	1	1
0	1	1	0	1	1	1

Verilog Code:

```
module decoder(
input A,B,E,
output D0,D1,D2,D3
    );
  assign D0= A|B|E;
  assign D1= A|~B|E;
  assign D2= ~A|B|E;
  assign D3= ~A|~B|E;
endmodule
```

Simulation Results:

Input: E = 0, B (MSB) = 1, A (LSB) = 0

Output: D [3:0] = 1011

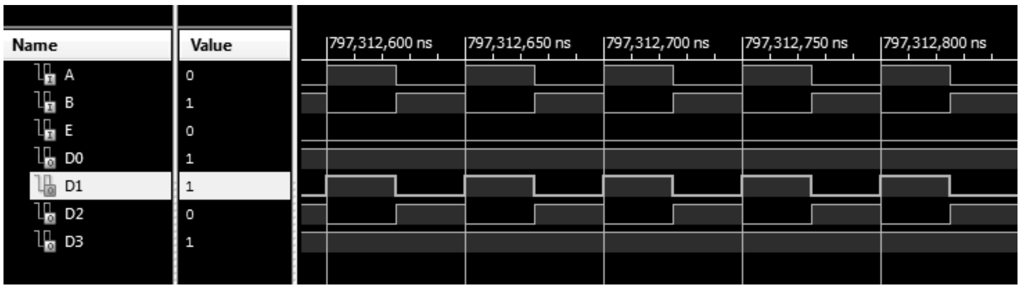


Fig. 2.2: Simulation results of example 2.2

EXAMPLE 2.3: Write a dataflow Verilog code for 8- to-1 multiplexer with active low select input and verify the design by simulation.

Solution: The truth table of an 8- to-1 multiplexer, where **select**[2:0] are the select lines, **d**[7:0] are the input lines and **q** is the output line, is as follows:

select[2]	**select[1]**	**select[0]**	**q**
0	0	0	**d[7]**
0	0	1	**d[6]**
0	1	0	**d[5]**
0	1	1	**d[4]**
1	0	0	**d[3]**
1	0	1	**d[2]**
1	1	0	**d[1]**
1	1	1	**d[0]**

Verilog Code:

```
module mux1( select, d, q );
    input [2:0] select;
    input [7:0] d;
    output q;
    assign q = d [~select];
endmodule
```

Simulation Results:

Input: select[2:0]=101;

d[7:0]=10101010

Output: q = 0

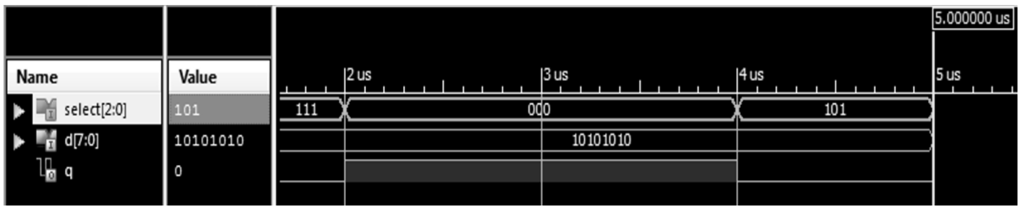
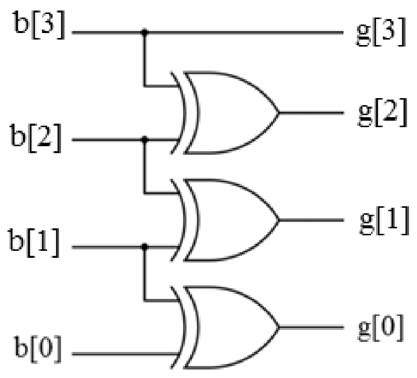


Fig. 2.3: Simulation results of example 2.3

EXAMPLE 2.4: Write a dataflow Verilog code for a 4-bit binary-to-gray code converter and verify the design by simulation.

Solution:



Verilog Code:

```
module binarytogray(
input [0:3] b,
output [0:3] g
    );
    assign g[3]=b[3];
    assign g[2]=b[3]^b[2];
    assign g[1]=b[2]^b[1];
    assign g[0]=b[0]^b[1];
endmodule
```

Simulation Results:

Input: b [3:0] = 0101_2

Output: g [3:0] = 0111_2

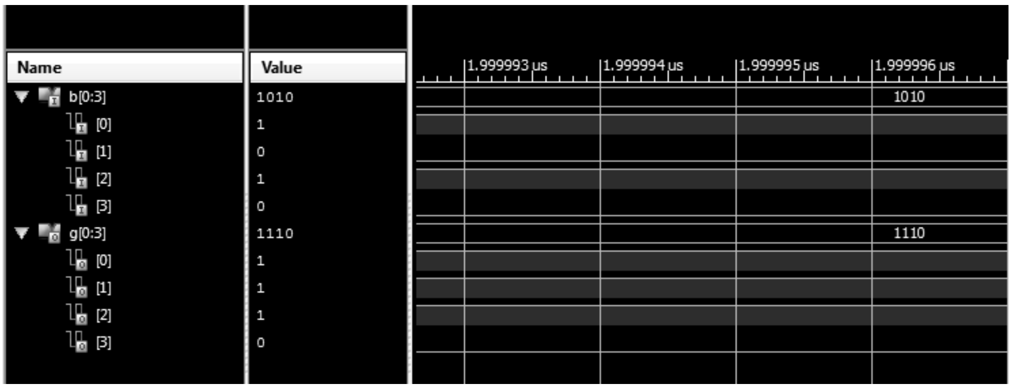


Fig. 2.4: Simulation results of example-2.4

EXERCISE PROBLEMS:

1. Write a dataflow Verilog code to realize the given logic function in POS form and verify the design by simulation.

$$F = (\overline{A}+\overline{B}+C).(\overline{A}+\overline{B}+\overline{C}).(A+B+C).(A+B+\overline{C})$$

2. Write a dataflow Verilog code for the following digital building blocks and verify the design by simulation: [i]. full adder, [ii]. full subtractor, [iii]. three variable majority function, [iv]. three input ex-nor function, [v]. two-bit equality detector.
3. Write a dataflow Verilog code for an 8- to-3 encoder with enable input and verify the design by simulation.
4. Write a dataflow Verilog code for an 8-to-3 priority encoder and verify the design by simulation.
5. Write a dataflow Verilog code for a 4-bit gray-to-binary code converter and verify the design by simulation.
6. Write a dataflow Verilog code for the 8421 to 2421 code converter and verify the design by simulation.
7. Write a dataflow Verilog code for a 1-bit magnitude comparator and verify the design by simulation.
8. Write a dataflow Verilog code for the N-bit magnitude comparator and verify the design by simulation.
9. Write a dataflow Verilog code for a 4-bit adder and verify the design by simulation.

Experiment No. 3

Verilog Sequential Modeling

OBJECTIVE: To understand the concepts related to sequential modeling style and write Verilog programs using the same.

THEORY: To model the behavior of a digital description using sequential modeling, the following two statements are primarily used:

i) Initial statement
ii) Always statement

Initial statement: An `initial` statement executes only once. It begins its execution at the start of the simulation, which is at time t = 0.

Syntax:

```
initial

[timing_control] procedural_statement
```

Always statement: An `always` statement executes repeatedly. Just like the `initial` statement, an `always` statement also begins execution at time t = 0.

Syntax:

```
always

[timing_control] procedural_statement
```

Only a *register* data type can be assigned a value in either of these statements. Such data type retains its value until a new value is assigned. All `initial` and `always` statements begin execution at time t = 0 concurrently. If no delays are specified in a procedural assignment, zero delay is the default; that is, the assignment occurs instantaneously.

EXAMPLE 3.1: Write a sequential Verilog code for an 8-to-3 priority encoder with active high enable input and verify the design by simulation.

Solution: Truth Table of 8-to-3 priority encoder with active high enable input

Input									**Output**		
E	D7	D6	D5	D4	D3	D2	D1	D0	Q2	Q1	Q0
0	X	X	X	X	X	X	X	X	X	X	X
1	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	1	X	0	0	1
1	0	0	0	0	0	1	X	X	0	1	0
1	0	0	0	0	1	X	X	X	0	1	1
1	0	0	0	1	X	X	X	X	1	0	0
1	0	0	1	X	X	X	X	X	1	0	1
1	0	1	X	X	X	X	X	X	1	1	0
1	1	X	X	X	X	X	X	X	1	1	1

Verilog Code:

```
    module encoder(D,Q,E);
        input [7:0] D;
        input E;
        output [2:0] Q;
        reg [2:0] Q;
        always @(D or E)
        begin
            if (E = = 1)
                casez (D)
                    8'b00000001: Q=3'b000;
                    8'b0000001?: Q=3'b001;
                    8'b000001??: Q=3'b010;
                    8'b00001???: Q=3'b011;
                    8'b0001????: Q=3'b100;
                    8'b001?????: Q=3'b101;
                    8'b01??????: Q=3'b110;
                    8'b1???????: Q=3'b111;
                endcase
            else
                Q=3'bX;
        end
    endmodule
```

Simulation Results:

Input: D [7:0] = 00100 010,

E = 1

Output: Q [2:0] = 101

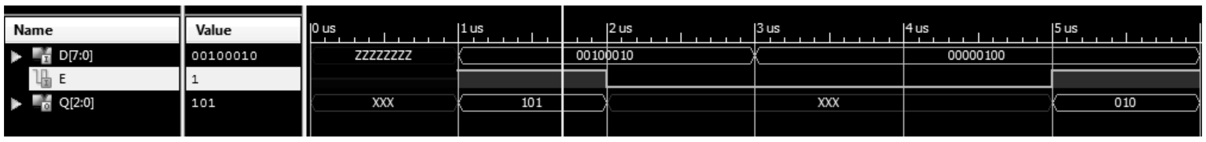


Fig. 3.1: Simulation results of example 3.1

EXAMPLE 3.2: Write a sequential Verilog code for a 3-bit binary ripple-up counter and verify the design by simulation.

Verilog Code:

```
    `timescale 1ns / 1ps
```

```
module counter( clk, count );
    input clk;
    output [2:0] count;
    reg [2:0] count;
    wire clk;
    initial
        count = 3'b0;
    always @( negedge clk )
        count[0] <= ~count[0];
    always @( negedge count[0] )
        count[1] <= ~count[1];
    always @( negedge count[1] )
        count[2] <= ~count[2];
endmodule
```

Testbench:

```
module counter_tb;
    reg clk;
    wire [2:0] count;
    counter cnter( .clk(clk), .count( count ) );
    initial
        begin
            clk = 0;
            #200 $finish;
        end
    always
        begin
            #2 clk = ~clk;
        end
    always @( posedge clk)
        $display("Count = %b", count );
endmodule
```

Simulation Results:

Output: 000 - - 001 - - 010 - - 011 - - 100 - - 101 - - 110 - - 111

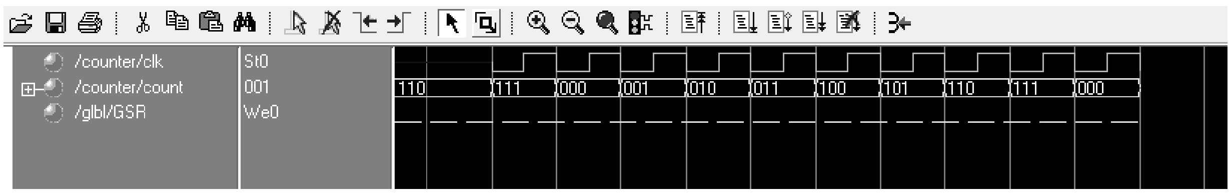
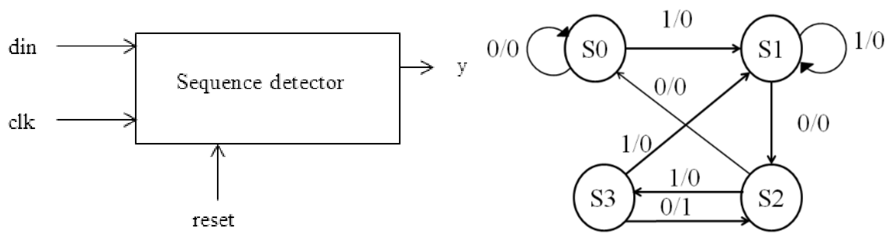


Fig. 3.2: Simulation results of example 3.2

EXAMPLE 3.3: Write a sequential Verilog code for 1010 overlapping sequence detector with active low reset and positive edge triggered clock (use parameter declaration) and verify the design by simulation.

Solution:



Overlapping 1010 sequence detector block and state diagram [Format: din/ y]

Verilog Code:

```
module melfsm(din, reset, clk, y);
  input din;
  input clk;
  input reset;
  output reg y;
  reg [1:0] cst, nst;
  parameter S0 = 2'b00, //all state
            S1 = 2'b01,
            S2 = 2'b10,
            S3 = 2'b11;
  always @(cst or din)
    begin
      case (cst)
          S0: if (din == 1'b1)
                begin
                    nst = S1;
                    y=1'b0;
                end
      else
          begin
               nst = cst;
               y=1'b0;
          end
S1: if (din = = 1'b0)
          begin
               nst = S2;
               y=1'b0;
          end
     else
          begin
               y=1'b0;
               nst = cst;
          end
S2: if (din = = 1'b1)
          begin
               nst = S3;
               y=1'b0;
          end
     else
          begin
               nst = S0;
               y=1'b0;
          end
S3: if (din = = 1'b0)
          begin
               nst = S2;
               y=1'b1;
          end
     else
          begin
               nst = S1;
               y=1'b0;
```

```
                end
          default:
              nst = S0;
      endcase
    end
  always@(posedge clk)
    begin
      if (reset)
          cst<= S0;
      else
          cst<= nst;
      end
    end
endmodule
```

Simulation Results:

Input: din :1010

Output: y : 0 0 0 1

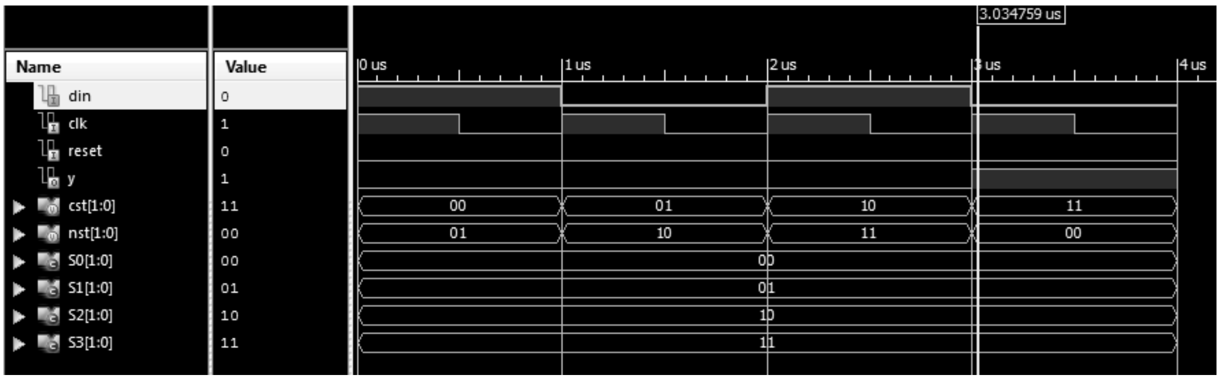


Fig. 3.3: Simulation results of example 3.3

EXAMPLE 3.4: Write a sequential Verilog code for a 4-bit ring counter and verify the design by simulation.

Solution:

Table showing output sequence of the 4-bit ring counter

Count Order	Sequence
0	1000
1	0100
2	0010
3	0001

Verilog Code:

```
module Ringcounter(q,clk,clr);
```

```
        input clk,clr;
        output [3:0] q;
        reg [3:0] q;
        always @(posedge clk)
            if(clr= =1)
                q<=4'b1000;
            else
                begin
                    q[3]<=q[0];
                    q[2]<=q[3];
                    q[1]<=q[2];
                    q[0]<=q[1];
                end
        end
    endmodule
```

Test Bench:

```
    module ringtest;
        // Inputs
        reg clk;
        reg clr;
        // Outputs
        wire [3:0] q;
        // Instantiate the Unit Under Test (UUT)
        Ringcounter uut (
            .q(q),
            .clk(clk),
            .clr(clr)
        );
        always
        begin
            #50 clk=1'b1;
            #50 clk=1'b0;
        end
    initial
        begin
        // Initialize Inputs
            clk = 0;
            clr = 0;
            #50 clr = 1'b1;
            #100 clr = 1'b0;
        // Wait 100 ns for global reset to finish
            #100;
        end
endmodule
```

Simulation Results:

Output: 1000 --- 0100 --- 0010 --- 0001--- 1000 ----

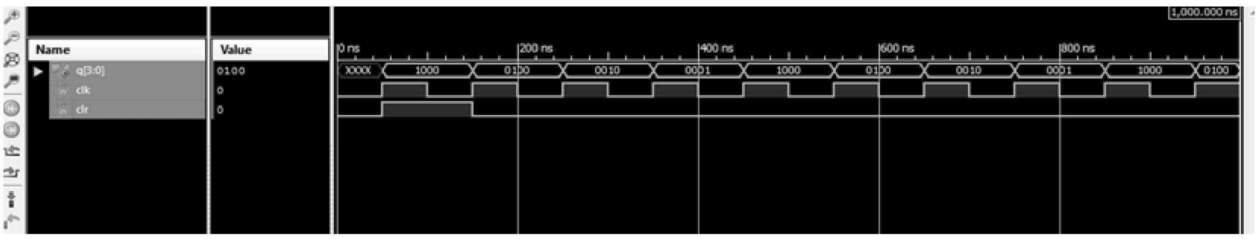


Fig. 3.4: Simulation results of example 3.4

EXERCISE PROBLEMS:

1. Write the sequential Verilog code for N bit full adder (assume N = 4 and use for-loop statement) and verify the design by simulation.
2. Write the sequential Verilog code for the synchronous mod 5 counter and verify the design by simulation.
3. Write a sequential Verilog code for a 4-bit priority encoder and verify the design by simulation.
4. Write the sequential Verilog code for Master-Slave JK flip-flop (assume delay of master and slave as 2 ns and 1ns respectively) and verify the design by simulation.
5. Write sequential Verilog code for 4-bit universal shift register and verify the design by simulation.
6. Write sequential Verilog code to model ACTEL ACT 1 Logic Module (Use initial statement) and verify the design by simulation.

Experiment. No. 4

Verilog Structural Modeling

OBJECTIVE: To understand the concepts related to structural modeling style and write Verilog programs on it.

THEORY: Structures can be described in Verilog HDL using

i) Built-in gate primitives (at the gate level)
ii) Switch level primitives (at the transistor level)
iii) User-defined primitives (at the gate level)
iv) Module primitives (to create hierarchy)

A module can be instantiated in another module, thus creating a hierarchy. A module instantiation statement is of the form:

```
Module_name instance_name(port_association);
```

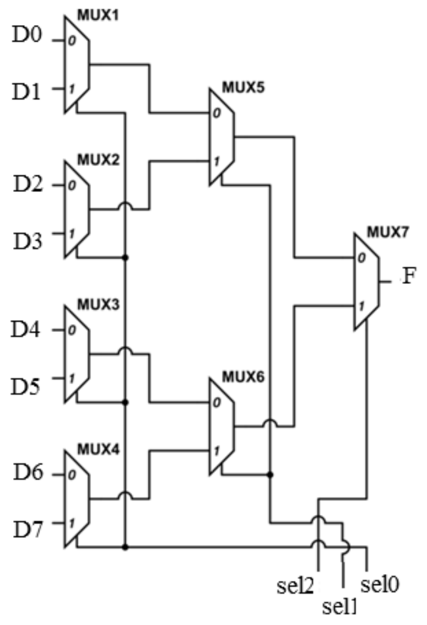
Port association can be by position or by name, however associations cannot be mixed. A port association is of the form.

```
port_expr
.portname(port_expr)
```

In positional association, the port expressions connect to the module's ports in the specified order. In association by name, the connection between the module port and the port expression is explicitly specified, and thus, the order of port associations is not important.

EXAMPLE 4.1: Write structural Verilog code for 8:1 multiplexer using 2:1 multiplexers and verify the design by simulation.

Solution:



Structure of 8:1 multiplexer using 2:1 multiplexers

Verilog Code:

```
module mux_2to1(
input A,B,S,
output Y
    );
    wire Sbar;
    assign S bar=~S;
    assign Y=((Sbar& A)|(S & B));
endmodule

module mux8to1(D,sel,F);
    input [7:0] D;
    input [2:0] sel;
    output F;
    wire W [6:1];
    mux_2to1 M1(D[0],D[1],sel[0],W[1]);
    mux_2to1 M2(D[2],D[3],sel[0],W[2]);
    mux_2to1 M3(D[4],D[5],sel[0],W[3]);
    mux_2to1 M4(D[6],D[7],sel[0],W[4]);
    mux_2to1 M5(W[1],W[2],sel[1],W[5]);
    mux_2to1 M6(W[3],W[4],sel[1],W[6]);
    mux_2to1 M7(W[5],W[6],sel[2],F);
endmodule
```

Simulation Results:

Input: D[7:0] = 0001 0000_2 ; sel[2:0] = 100_2 ;

Output: F = 1

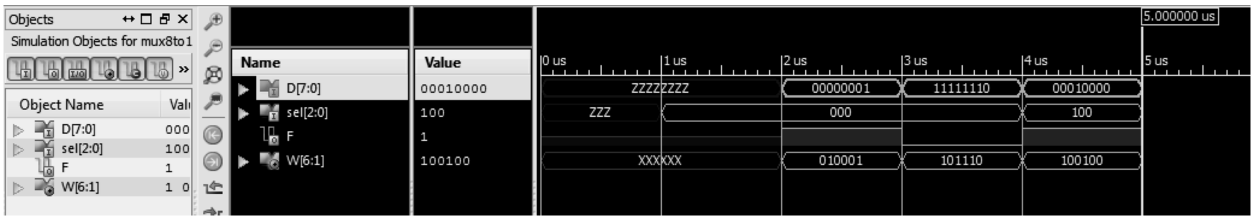
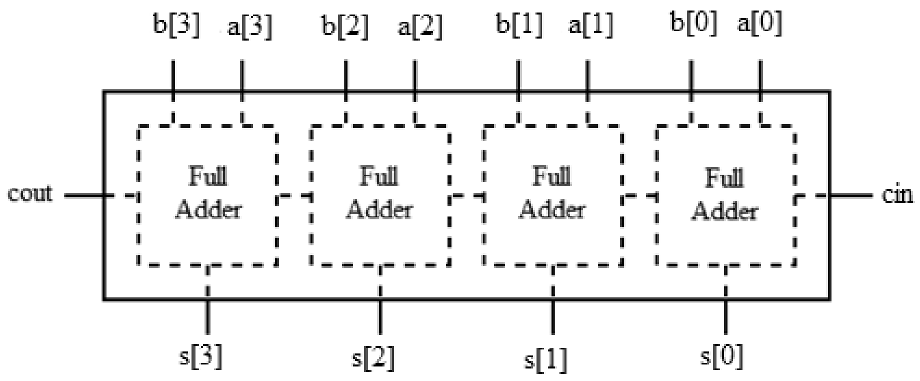


Fig. 4.1: Simulation results of example 4.1

EXAMPLE 4.2: Write hierarchical structural Verilog code for 4-bit ripple carry adder using full-adder component and verify the design by simulation.

Solution:



4 bit ripple carry adder using full-adder blocks

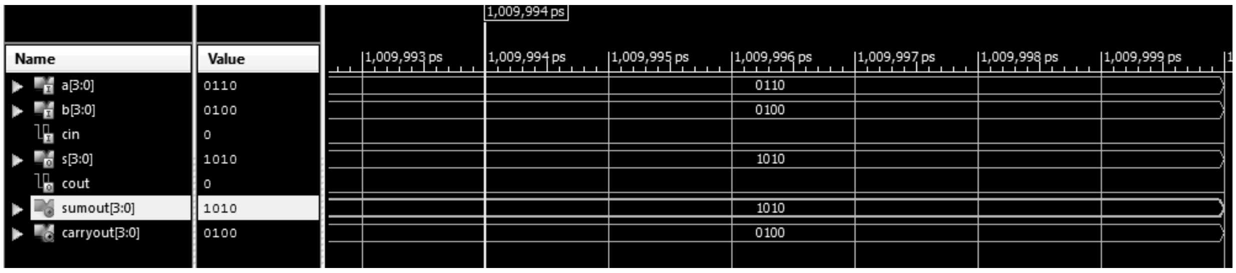
Verilog Code:

```
module adder(input a,input b,input cin,output s,output cout);
   assign s=a^b^cin;
   assign cout=(a&b)|(b&cin)|(cin&a);
endmodule

module rippleadd(input [3:0] a, input [3:0] b,input
cin,output [3:0] s,output cout);
   wire [3:0] sumout;
   wire [3:0] carryout;
   adder fa1(a[0],b[0],cin, sumout[0],carryout[0]);
   adder fa2(a[1],b[1],carryout[0],sumout[1], carryout[1]);
   adder fa3(a[2],b[2],carryout[1],sumout[2], carryout[2]);
   adder fa4(a[3],b[3],carryout[2],sumout[3], carryout[3]);
   assign s= sumout;
   assign cout = carryout[3];
endmodule
```

Simulation Results:

Input: a[3:0] = 0110_2 , b[3:0] = 0100_2, cin = 0_2

Output: s[3:0] = 1010_2, cout = 0_2

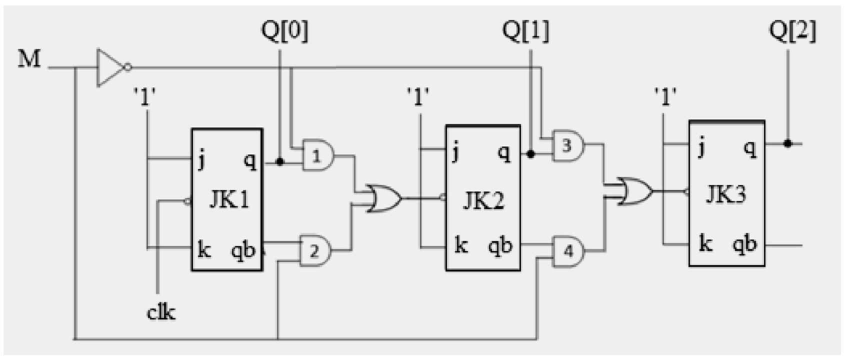


```
                     1,009,994 ps
Name            Value   |1,009,993 ps |1,009,994 ps |1,009,995 ps |1,009,996 ps |1,009,997 ps |1,009,998 ps |1,009,999 ps
a[3:0]          0110                               0110
b[3:0]          0100                               0100
cin             0
s[3:0]          1010                               1010
cout            0
sumout[3:0]     1010                               1010
carryout[3:0]   0100                               0100
```

Fig. 4.2: Simulation results of example 4.2

EXAMPLE 4.3: Write structural Verilog code for a 3-bit ripple up/down counter and verify the design by simulation.

Solution:



3-bit ripple up/down counter realization using JK flip-flops

Verilog Code:

```
`timescale 1ns / 1ps
module Jk_FF(j,k,clock,q,qb);
  input j,k,clock;
  output reg q,qb;
  initial
      begin
          q=1;
          qb=0;
      end
  always@(posedge clock)
      begin
          case({j,k})
          2'b00 :q=q;
          2'b01 :q=0;
          2'b10 :q=1;
          2'b11 :q=~q;
          default :q=0;
          endcase
          qb<=~q;
      end
endmodule
```

```
module jk_up_down_counter(input clk,input M,output
[2:0]Q);
  wire S1,S2,S3,S4,S5,S6,S7,S8,S9;
  Jk_FF    JK1(1'b1,1'b1,clk,Q[0],S1),
           JK2(1'b1,1'b1,S4,Q[1],S5),
           JK3(1'b1,1'b1,S8,Q[2],);
  and A1(S2,S9,Q[0]),A2(S3,S1,M),A3(S7,Q[1],S9),
      A4(S6,S5,M);
  or or1 (S4,S2,S3),or2 (S8,S7,S6);
  not not1(S9,M);
endmodule
```

Simulation Results:

Input: M = 1 (UP mode)

Output: 000 - - 001 - - 010 - - 011 - - 100 - - 101 - -

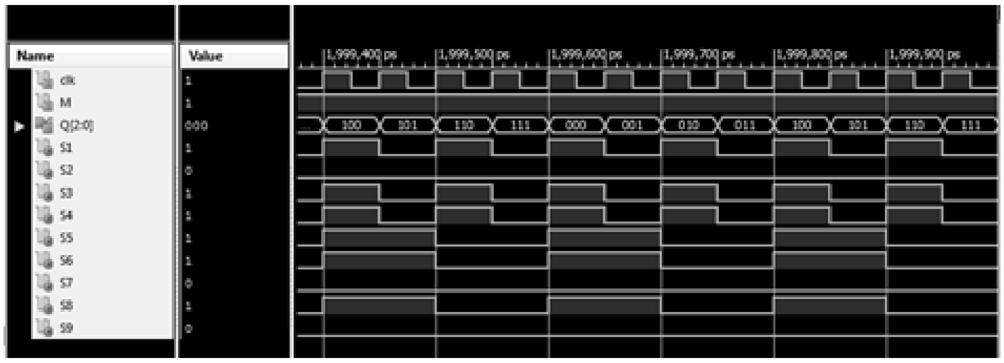
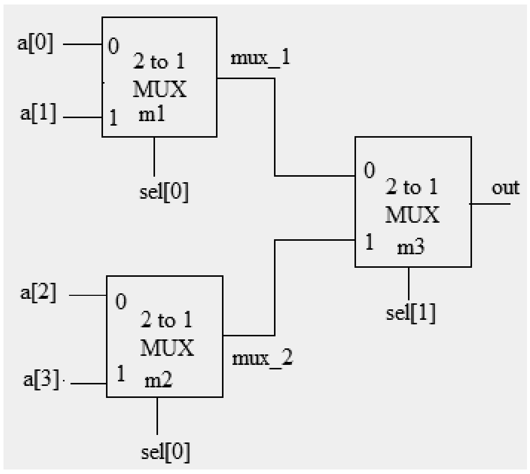


Figure 4.3: Simulation results of example 4.3

EXAMPLE 4.4: Write structural Verilog code for 4:1 multiplexer using 2:1 multiplexer and verify the design by simulation.

Solution:



4:1 multiplexer using 2:1 multiplexers

Verilog Code:

```
module mux4to1(a,sel,out);
  input [3:0] a;
  input [1:0] sel;
  output out;
  wire mux[2:0];

  Mux2to1 m1(a[0], a[1], sel[0], mux_1);
  Mux2to1 m2(a[2], a[3], sel[0], mux_2);
  Mux2to1 m3(mux_1,mux_2,sel[1],out);
endmodule
```

Simulation Results:

Input: sel[1:0] = 10_2 , a[3] = 0, a[2] = 1, a[1] = 0, a[0] = 1

Output: out = 1

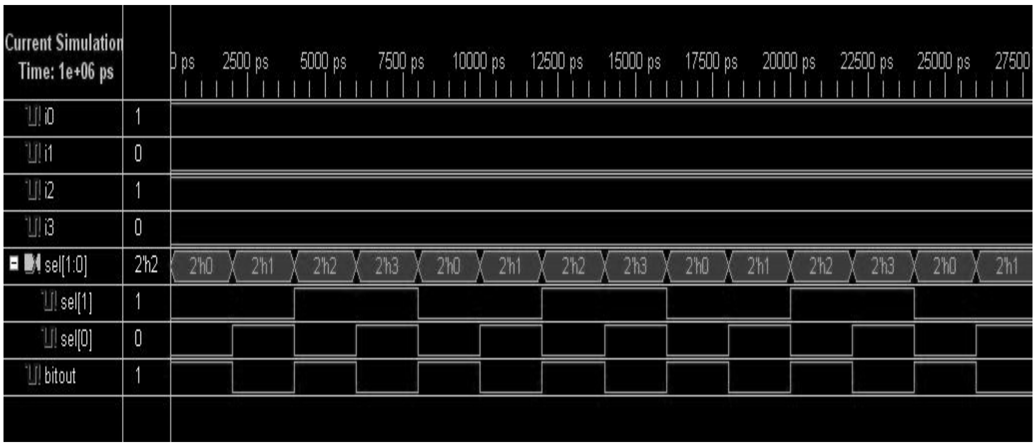


Figure 4.4: Simulation results of example-4.4

EXERCISE PROBLEMS:

1. Write structural Verilog code for mod-10 ripple counter and verify the design by simulation.
2. Write structural Verilog code for (a) 4-bit SIPO shift register and (b) 4-bit PISO shift register and verify the design by simulation.
3. Write structural Verilog code for 4-bit carry look-ahead adder and verify the design by simulation.
4. Write structural Verilog code for 4-bit carry save multiplier and verify the design by simulation.
5. Write structural Verilog code for a 4-bit binary-to-gray code converter and verify the design by simulation.

Experiment No. 5

Verilog Switch Level and Mixed-mode Modeling

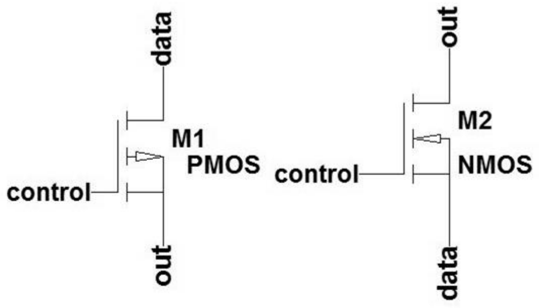
OBJECTIVE: To study switch level and mixed mode style of Verilog with examples

THEORY: Usually, transistor level modeling is referred to as modeling the hardware structures using transistor models with analog inputs and outputs. On the other hand, gate level modeling refers to modeling hard-ware structures using gate models with digital input and output signal values. Between these two modeling schemes is what is referred to as switch level modeling. At this level, a hardware component is described at the transistor level, but transistors only exhibit digital behavior and their input, and output signal values are only limited to digital values. At the switch level, transistors behave as on-off switches. Verilog uses a 4-value logic value system, so Verilog switch input and output signals can take any of the four 0, 1, Z, and X logic values.

Syntax:

```
nmos n1(out, data, control);
pmos p1(out, data, control);
```

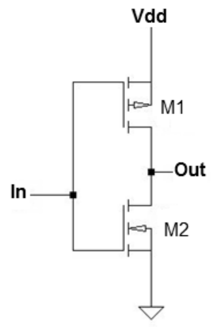
The two MOS switches, namely `nmos` and `pmos,` are used to model NMOS and PMOS transistors, respectively, and their symbols are as follows:



Symbol for NMOS and PMOS transistor

EXAMPLE 5.1: Write switch level Verilog description of the following and verify the design by simulation: [i] CMOS inverter, [ii] 3 input CMOS NOR gate.

Solution: [i] CMOS inverter



CMOS Inverter

Verilog Code:

```
module cmos1(out,in);
    output out;
    input in;
    supply1 vdd;
    supply0 gnd;
    wire out;

    pmos M1(out,vdd,in);
    nmos M2(out,gnd,in);
endmodule
```

Test bench:

```
module cmos1_test;
    // Inputs
    reg in;
    // Outputs
    wire out;
    // Instantiate the Unit Under Test (UUT)
    cmos1 uut (.out(out), .in(in));

    initial
        begin
            // Input Stimuli
            in = 1'b0;
            #100;
            in = 1'b1;
            #100;
        end
endmodule
```

Simulation Results:

Input: in = 1

Output: out = 0

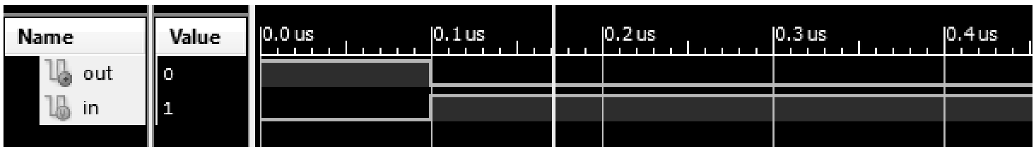
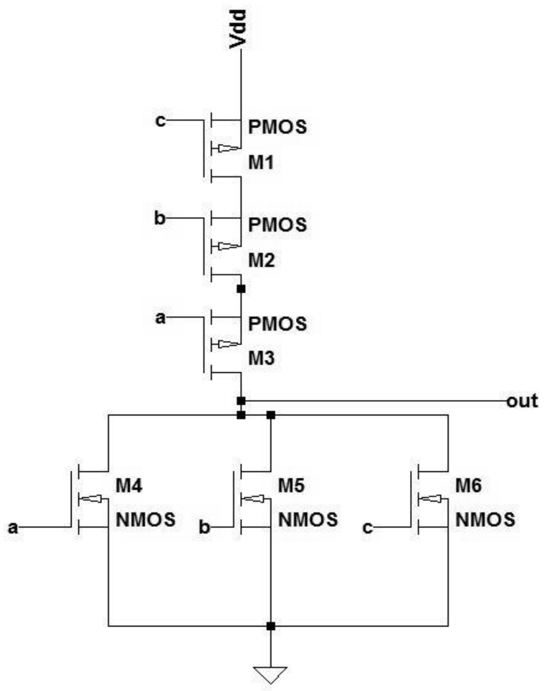


Fig. 5.1(a): Simulation results of example 5.1[i]

[ii] 3 input CMOS NOR gate



Three input CMOS NOR gate

Verilog Code:

```
module nor_3_cmos(out,a,b,c);
    output out;
    input a,b,c;

    supply1 vdd;
    supply0 gnd;

    pmos M1(e,vdd,c);
    pmos M2(d,e,b);
    pmos M3(out,d,a);
    nmos M4(out,gnd,a);
    nmos M5(out,gnd,b);
    nmos M6(out,gnd,c);
endmodule
```

Testbanch:

```
module nor_3_tb;
    // Inputs
    reg a;
    reg b;
```

```
        reg c;
        // Outputs
        wire out;
        // Instantiate the Unit Under Test (UUT)
        nor_3_cmosuut (.out(out),.a(a),.b(b),.c(c));
        initial
            begin
                // Input Stimuli
                a=1'b0;b=1'b0;c=1'b0;
                #5 a=1'b0;b=1'b0;c=1'b1;
                #5 a=1'b0;b=1'b1;c=1'b0;
                #5 a=1'b0;b=1'b1;c=1'b1;
                #5 a=1'b1;b=1'b0;c=1'b0;
                #5 a=1'b1;b=1'b0;c=1'b1;
                #5 a=1'b1;b=1'b1;c=1'b0;
                #5 a=1'b1;b=1'b1;c=1'b1;
            end
        initial
        $monitor($time,"out=%b,a=%b,c=%b",out,a,b,c);
    endmodule
```

Simulation Results:

Input: a = 1, b = 1, c = 1

Output: out = 0

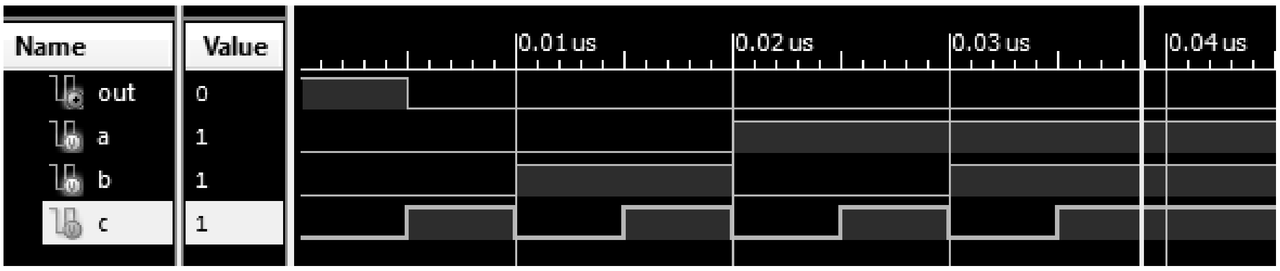
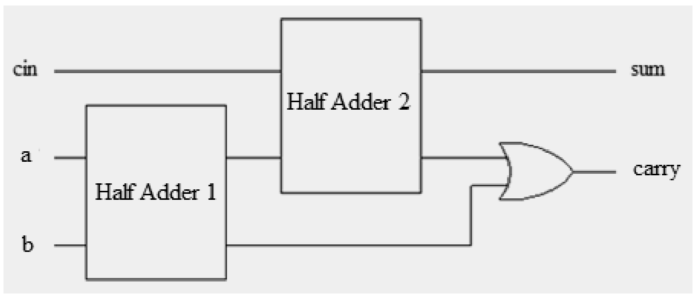


Fig. 5.1(b): Simulation results of example 5.1[ii]

EXAMPLE 5.2: Write a Verilog code for 1-bit full adder using mixed style of modeling and verify the design by simulation.

Solution:



Full adder in terms of half adders

Verilog Code:

```
module fulladder_task(a,b,cin,sum,carry);
  input a,b,cin;
  output sum,carry;
  reg sum,carry;
  reg s1,s2,s3,s4,s5,s6,s7;

  always@(a or b or cin)
  begin
      s4=a;
      s5=b;
      s6=cin;
      halfadder_task(s4,s5,s1,s2);
      halfadder_task(s1,s6,s7,s3);
      carry=s2|s3;
      sum=s7;
      $display("sum=%b carry=%b",sum,carry);
  end

  task halfadder_task;
      input l,m;
      output y,z;
      begin
          y=l^m;
          z=l&m;
      end
  endtask
endmodule
```

Test bench:

```
module full_test;
     // Inputs
     reg a;
     reg b;
     reg cin;
     // Outputs
     wire sum;
     wire carry;
     // Instantiate the Unit Under Test (UUT)
     fulladder_task uut (.a(a),.b(b),.cin(cin),
                    .sum(sum), .carry(carry));

     initial
     begin
          // Input Stimuli
          $monitor($time,"a=%b b=%b cin=%b sum=%b
                  carry=%b",a,b,cin,sum,carry);

          a=0; b=0; cin=0;
          #10 a=0; b=0; cin=1;
          #10 a=0; b=1; cin=0;
          #10 a=0; b=1; cin=1;
          #10 a=1; b=0; cin=0;
          #10 a=1; b=0; cin=1;
          #10 a=1; b=1; cin=0;
          #10 a=1; b=1; cin=1;
     end
endmodule
```

Simulation Results:

Input: a = 1, b = 0, cin = 0

Output: Sum = 1, carry = 0

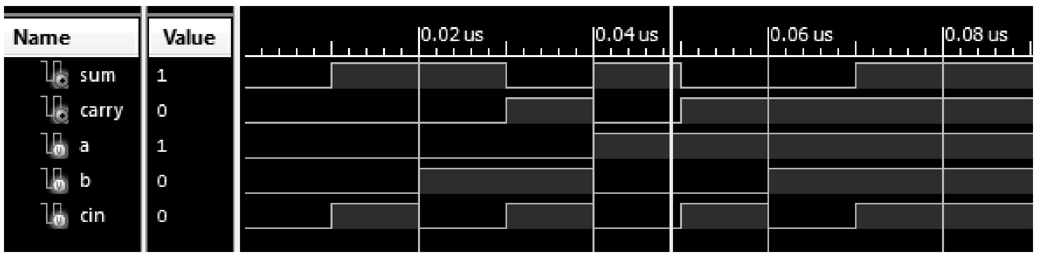
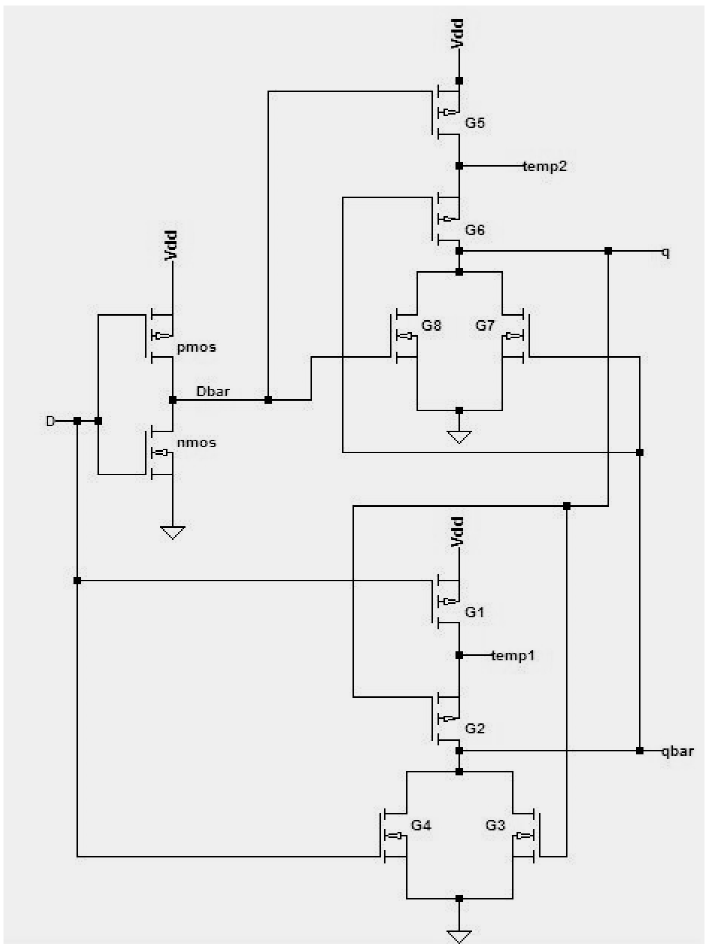


Fig. 5.2: Simulation results of example 5.2

EXAMPLE 5.3: Write switch level Verilog code of a D-latch using PMOS and NMOS switches and verify the design by simulation.

Solution:



CMOS circuit of D latch

Verilog Code:

```
module d_latch(d,q,qbar);
    input d;
    output q;
    output qbar;
```

```
        wire temp1,temp2,dbar;
        supply1 vdd;
        supply0 gnd;

        pmos p1(dbar,vdd, d);
        nmos n1(dbar,gnd,d);
        pmos g5(temp2,vdd,dbar);
        nmos g8(q,gnd,dbar);
        pmos g6(q,temp2,qbar);
        nmos g7(q,gnd,qbar);
        pmos g1(temp1,vdd,d);
        pmos g2(qbar,temp1,q);
        nmos g4(qbar,gnd,d);
        nmos g3(qbar,gnd,q);
    endmodule
```

Test bench:

```
      module dlatch_tb;
        reg d;
        wire q;
        wire qbar;
        d_latch uut (.d(d), .q(q), .qbar(qbar));
        initial
            begin
                d=1;
                #100;
                d=0;
            end
    endmodule
```

Simulation Results:

Input: d = 1

Output: q = 1, qbar = 0

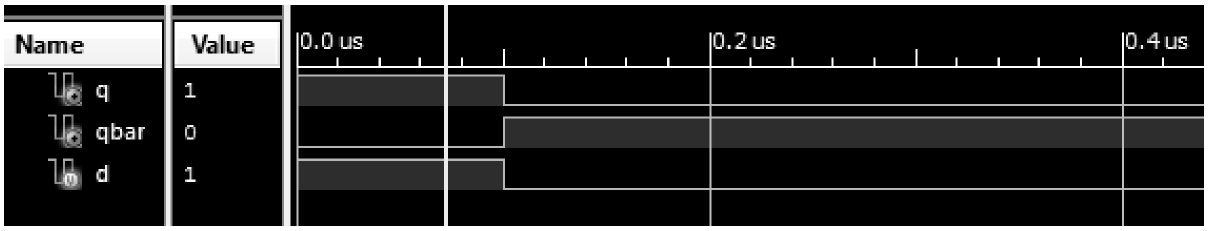
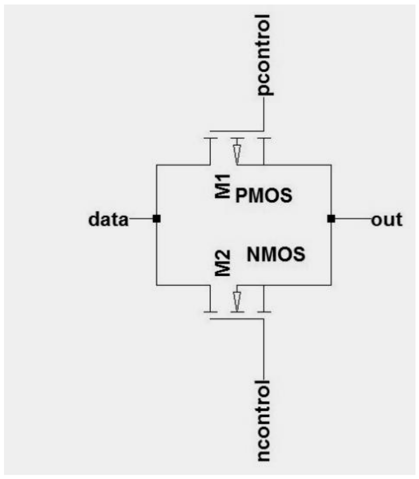


Fig. 5.3: Simulation results of example-5.3

EXAMPLE 5.4: Write switch level Verilog code for 1-bit CMOS shift register celland verify the circuit operation by simulation.

Solution:



1-bit CMOS shift register cell

Verilog Code:

```
    module sr(data, pcontrol, ncontrol, out);
        input data;
        input pcontrol;
        input ncontrol;
        output out;

        nmos (out,data,ncontrol);
        pmos (out,data,pcontrol);
    endmodule
```

Test bench:

```
    module CMOS_Cell_tb;
        reg data; // Inputs
        reg pcontrol;
        reg ncontrol;
    wire out; // Output

    // Instantiate the Unit Under Test (UUT)
    sr uut (.data(data), .pcontrol(pcontrol),
            .ncontrol(ncontrol), .out(out));


    initial
        begin
            // Input Stimuli
            data = 0;
            pcontrol = 0;
            ncontrol = 1;
            #100;

            data = 1;
            pcontrol = 0;
            ncontrol = 1;
            #100;

            data = 1;
            pcontrol = 0;
            ncontrol = 0;
            #100;

            data = 1;
            pcontrol = 1;
            ncontrol = 1;
            #100;

            data = 1;
            pcontrol = 1;
            ncontrol = 0;

        end
endmodule
```

Simulation Results:

Input: data = 1, pcontrol = 0, ncontrol = 1

Output: out = 1

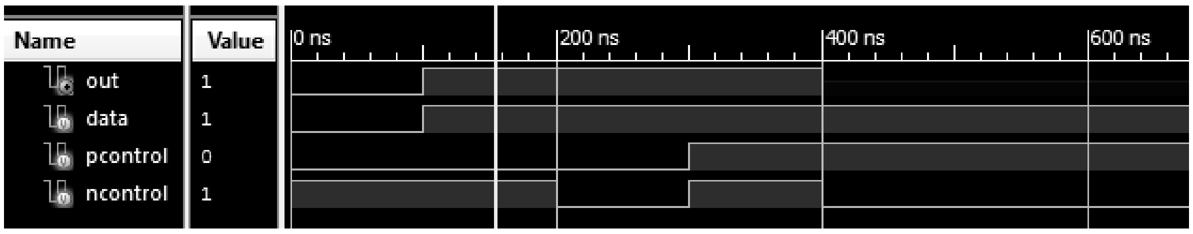


Fig. 5.4: Simulation results of example-5.4

EXERCISE PROGRAMS:

1. Write a switch level Verilog code for the following combinational logic using both gate based, and TG based approach

   $Y = \overline{(AB+CD)}$

2. Write switch level Verilog code for a 3 input CMOS NAND gate with test benches.
3. Write a Verilog code for ALU using mixed style of modelling. Model the addition operation using carry-look ahead adders. The operation code for selection is given below.

Operation Code	**Operation**
00	Addition
01	Multiplication
10	Integer Division
11	No operation

Experiment No. 6

Verilog examples using tasks, functions, and user defined primitives

OBJECTIVE: To study switch level and mixed mode style of Verilog with examples

THEORY:

Task: A task provides the ability to execute common pieces of code from several different places. This common piece of code is written as task so it can be called from different places in the design discerption.

A task is delimited by the keywords `task` and `endtask`. The syntax for a task declaration is as follows:

```
task task_name
  input arguments
  output arguments
  inout arguments
   …task declarations…
   …local variable declarations…
  begin
     …statements…
  end
endtask
```

Function: Functions are behavioral statements. Functions must be called within always or initial. Functions take one or more inputs, and, in contrast to task, they return only a single output value. Functions are delimited by the keywords `function` and `endfunction` and are used to implement combinational logic; therefore, functions cannot contain event controls or timing controls.

```
function [range or type] function name
  input declaration
  …other declarations…
  begin
     …statement…
  end
endfunction
```

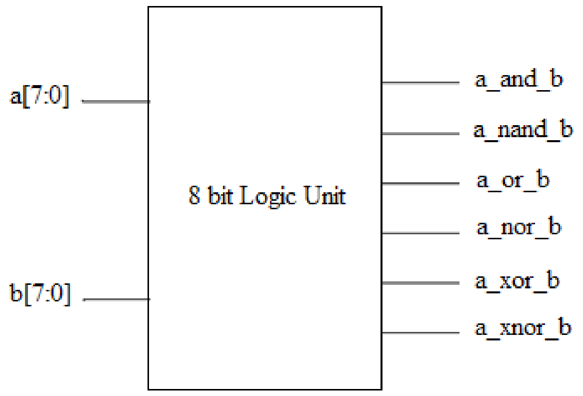
User defined primitive: The syntax for a user defined primitive (UDP) is similar to that for declaring a module. The definition begins with the keyword `primitive` and ends with the keyword `endprimitive`. The UDP contains a name and a list of ports, which are declared as `input` or `output`. For a sequential UDP, the output port is declared as `reg`. UDPs can have one or more scalar inputs, but only one scalar output. UDPs do not support `inout` ports

```
primitive udp_name (output,input_1,input_2, … , input_n);
```

```
        output output;
        input input_1, input_2, … , input_n;
        regs equential_output; //for sequential UDPs
        initial //for sequential UDPs
          table
          …state table entries…
          endtable
     endprimitive
```

EXAMPLE 6.1: Write a Verilog code using task to perform logical operations on two 8-bit vectors a[7:0] and b[7:0]. The logical operations are: AND, NAND, OR, NOR, exclusive-OR, and exclusive-NOR.

Solution:



Block performing logical operations on two 8-bit data

Verilog Code

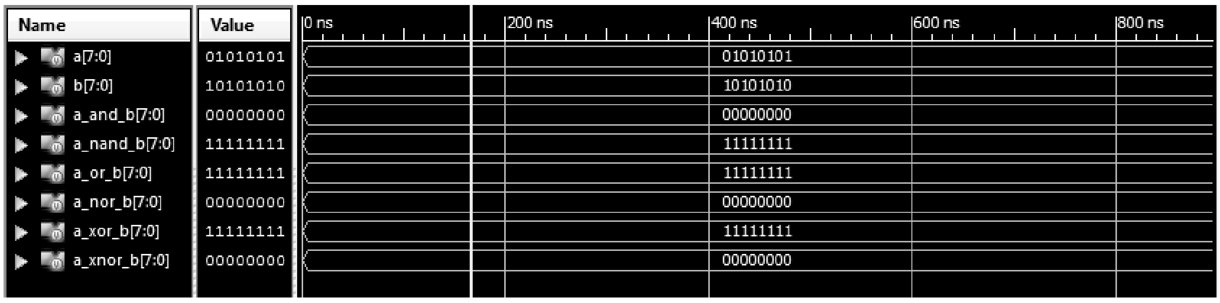
```
//module to illustrate a task for logical operations
module task_logical;
     reg[7:0] a, b;
     reg[7:0] a_and_b, a_nand_b, a_or_b, a_nor_b,
              a_xor_b, a_xnor_b;

     initial
          begin
          a=8'b1010_1010; b=8'b1100_1100;
          logical (a, b, a_and_b, a_nand_b, a_or_b,
                    a_nor_b, a_xor_b, a_xnor_b);
```

```
    //invoke the task
    a=8'b1110_0111; b=8'b1110_0111;
    logical (a, b, a_and_b, a_nand_b, a_or_b,
            a_nor_b,a_xor_b, a_xnor_b);

    //invoke the task
    a=8'b0000_0111; b=8'b0000_0111;
    logical (a, b, a_and_b, a_nand_b, a_or_b,
            a_nor_b,a_xor_b, a_xnor_b);

    //invoke the task
    a=8'b0101_0101; b=8'b1010_1010;
    logical (a, b, a_and_b, a_nand_b, a_or_b,
            a_nor_b,a_xor_b, a_xnor_b);

end

task logical;
    input [7:0] a, b;
    output [7:0] a_and_b, a_nand_b, a_or_b,
            a_nor_b,a_xor_b, a_xnor_b;
    begin
    a_and_b = a & b;
    a_nand_b = ~(a & b);
    a_or_b = a | b;
    a_nor_b = ~(a | b);
    a_xor_b = a ^ b;
    a_xnor_b = ~(a ^ b);

    $display ("a=%b, b=%b, a_and_b=%b,
    a_nand_b=%b,
            a_or_b=%b, a_nor_b=%b, a_xor_b=%b,
            a_xnor_b=%b", a, b, a_and_b,
                    a_nand_b, a_or_b, a_nor_b, a_xor_b,
                    a_xnor_b);
          end
     endtask
endmodule
```

Simulation Results:

Input: a[7:0] = 0101 0101_2, b[7:0] = 1010 1010_2

Output: a_and_b[7:0] = 0000 0000_2; a_nand_b[7:0] = 11111111_2; a_or_b[7:0] = 11111111_2; a_nor_b[7:0] = 00000000_2; a_xor_b[7:0] = 11111111_2; a_xnor_b[7:0] = 00000000_2

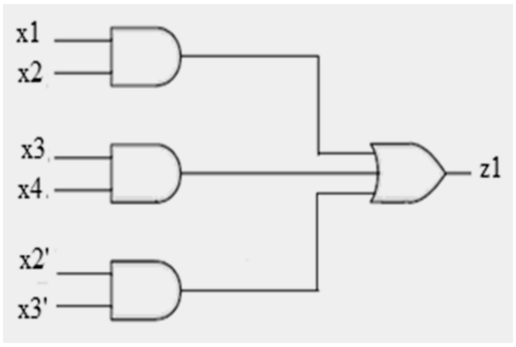


```
Name          Value
a[7:0]        01010101
b[7:0]        10101010
a_and_b[7:0]  00000000
a_nand_b[7:0] 11111111
a_or_b[7:0]   11111111
a_nor_b[7:0]  00000000
a_xor_b[7:0]  11111111
a_xnor_b[7:0] 00000000
```

Fig. 6.1: Simulation results of example 6.1

EXAMPLE 6.2: Write a Verilog code for the expression $z_1 = x_1\ x_2 + x_3\ x_4 + x_2'\ x_3'$ using the user defined AND, and OR gate primitives.

Solution:



SOP expression using AND gate and OR gate as UDP

Verilog Code:

```
//UDP for a 2-input AND gate
primitive udp_and2 (z1, x1, x2); //output is listed first
     input x1, x2;
     output z1;

     //define state table
     table
     //inputs are the same order as the input list
```

```
         // x1 x2 : z1; comment is for readability
             0 0 : 0;
             0 1 : 0;
             1 0 : 0;
             1 1 : 1;
        endtable
   endprimitive

  //UDP for a 3-input OR gate
  primitive udp_or3 (z1, x1, x2, x3); //output is listed
first
     input x1, x2, x3;
     output z1;

     //define state table
     table
     //inputs are the same order as the input list
     // x1 x2 x3 : z1; comment is for readability
        0 0 0 : 0;
        0 0 1 : 1;
        0 1 0 : 1;
        0 1 1 : 1;
        1 0 0 : 1;
        1 0 1 : 1;
        1 1 0 : 1;
        1 1 1 : 1;
     endtable
  endprimitive
  //sum of products using UDPs for the AND gate and OR gate
  module udp_sop (x1, x2, x3, x4, z1);
        input x1, x2, x3, x4;
        output z1;

        //define internal nets
        wire net1, net2, net3;

        //instantiate the udps
        udp_and2 (net1, x1, x2);
        udp_and2 (net2, x3, x4);
        udp_and2 (net3, ~x2, ~x3);
        udp_or3 (z1, net1, net2, net3);
    endmodule
```

Simulation Results:

Input: $x_1 = 1_2$; $x_2 = 1_2$; $x_3 = 0_2$; $x_4 = 1_2$

Output: z_1=1

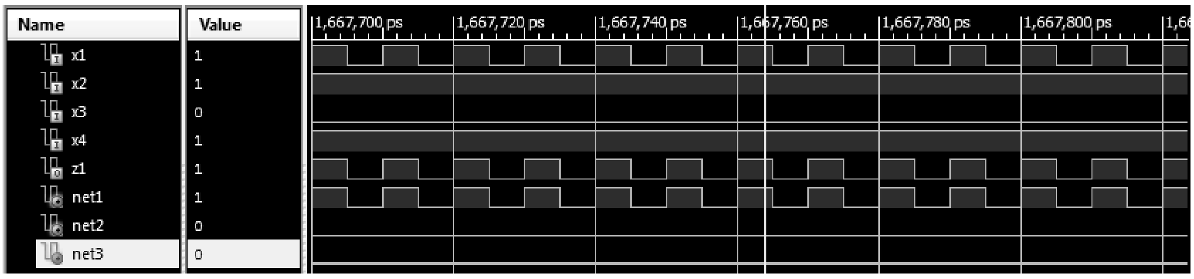
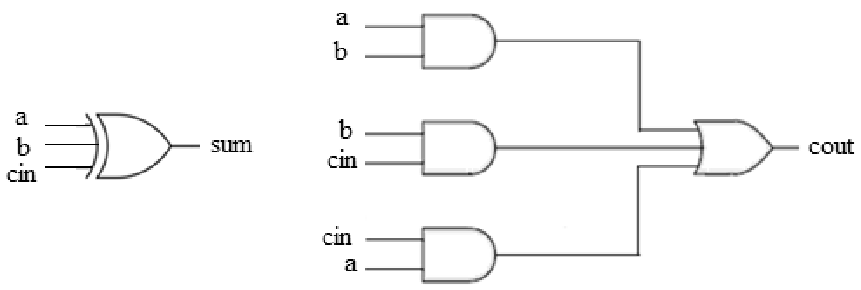


Fig. 6.2: Simulation results of example 6.2

EXAMPLE 6.3: Write a Verilog code for full adder using 3 input XOR gate as UDP and, AND and OR gates as built-in primitives.

Solution:



Full adder using basic gates

Verilog Code:

```
//UDP for a 3-input exclusive-OR
primitive udp_xor2 (z1, x1, x2,x3);
  input x1, x2, x3;
  output z1;
  //define state table
  table
    //inputs are in the same order as the input list
    // x1 x2 x3 : z1; comment is for readability
      0 0 0 : 0;
      0 0 1 : 1;
      0 1 0 : 1;
      0 1 1 : 0;
      1 0 0 : 1;
      1 0 1 : 0;
      1 1 0 : 0;
      1 1 1 : 1;
  endtable
endprimitive

//full adder using a UDP and built-in primitives
module full_adder_udp (a, b, cin, sum, cout);
  input a, b, cin;
  output sum, cout;

  //define internal nets
  wire net1, net2, net3;

  //instantiate the udps and built-in primitive
  udp_xor2 (sum, a, b, cin);
  and inst1 (net1, a, b);
  and inst2 (net2, b, cin);
  and inst3 (net3, a, cin);
  or inst3 (cout, net3, net2, net1);
```

```
endmodule
```

Simulation Results

Input: a = 1_2 ;b = 1_2 ; cin = 1_2

Output: sum =1_2 ; cout=1_2

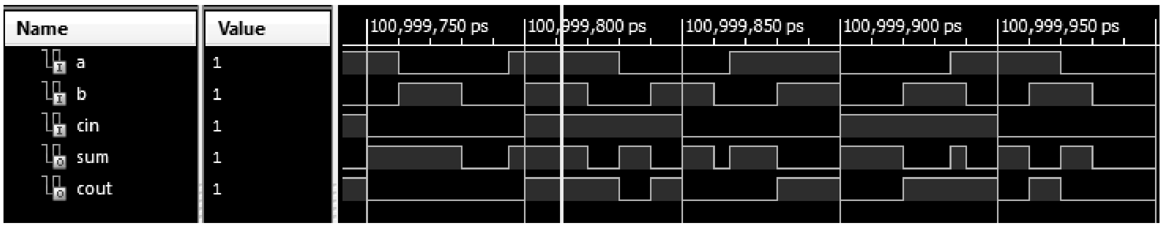
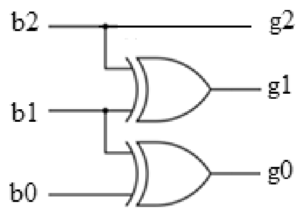


Fig. 6.3: Simulation results of example 6.3

EXAMPLE 6.4: Write a sequential Verilog code for 3-bit binary-to-gray code converter using function that evaluates the two-input EX-OR expression and verify the code by simulation.

Solution:



Binary-to-gray code converter

Verilog Code:

```
module Func_exm (b0, b1,b2,g0,g1,g2);
  input b0, b1,b2;
  output g0,g1,g2;
  reg g0,g1,g2;
  always @ (b0,b1,b2)
      begin
      g0= exp (b0, b1);
      g1= exp (b1, b2);
      g2= exp (0, b2);
      end


  function exp ;
      input a, b;
      begin
          exp = a ^ b;
      end
  endfunction
endmodule
```

Simulation Results:

Input: b[3:0] = 101_2

Output: g[3:0] =111_2

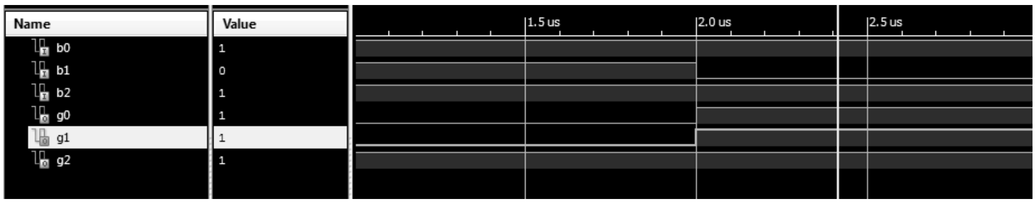


Fig. 6.4: Simulation results of example 6.4

EXERCISE PROBLEMS:

1. Write a Verilog code of 4-to-1 multiplexer as UDP and verify the design description by simulation.
2. Write a Verilog code for 4-bit binary-to-gray code converter using two-input xor gate UDP and verify the design by simulation.
3. Write a Verilog code to define and call the function that evaluates the two-input xor expression and verify the code by simulation.
4. Write a Verilog code for half-adder using task and then describe the behaviour of full-adder from two half-adders and verify the design by simulation.
5. Write a Verilog code for the positive-edge-triggered D flip-flop as UDP and verify the design by simulation.

Experiment No. 7

Utilization of on-board resources of BASYS-3 FPGA Kit

OBJECTIVE: To realize digital logic through Verilog programs targeting the on-board peripherals of Basys-3 FPGA kit.

THEORY: The Basys-3 FPGA board has the following features:

- 1,800 Kbits of fast block RAM
- 33,280 logic cells in 5200 slices (each slice contains four 6-input LUTs and 8 flip-flops)
- Five clock management tiles, each with a phase-locked loop (PLL)
- 90 DSP slices
- Internal clock speeds exceeding 450MHz
- On-chip analog-to-digital converter (XADC)
- 16 user switches
- 16 user LEDs
- 5 user pushbuttons
- 4-digit 7-segment display
- Three Pmod connectors
- Pmod for XADC signals
- 12-bit VGA output
- USB-UART bridge
- Serial Flash
- Digilent USB-JTAG port for FPGA programming and communication
- USB HID Host for mice, keyboards and memory sticks

EXAMPLE 7.1: Write a sequential Verilog code for an 8-bit ALU, verify the design by simulation, and then implement it on the Basys-3 FPGA kit.

Solution: **Truth Table of 8-bit ALU**

Input A	Input B	Select lines ALU_Sel	Operation	Output ALU_Out	CarryOut
00000011	00000011	000	Addition	00000110	0
00000011	00000011	001	Subtraction	00000000	0
00000011	00000011	010	Multiplication	00001001	0
00000011	00000011	011	Division	00000001	0
00000011	00000011	100	Logical left shift of A	00000110	0
00000011	00000011	101	Logical right shift of A	00000001	0
00000011	00000011	110	Rotate left of A	00000110	0
00000011	00000011	111	Rotate right of A	10000001	0

Verilog Code:

```
module ALU( input [7:0] A,B,              // ALU 8-bit Inputs
      input [2:0] ALU_Sel,                // ALU Selection
      output [7:0] ALU_Out,               // ALU 8-bit Output
      output CarryOut);                   // Carry Out Flag
    reg [7:0] ALU_Result;
    wire [8:0] tmp;
    assign ALU_Out = ALU_Result;
    assign tmp = {1'b0,A} + {1'b0,B};
    assign CarryOut = tmp[8];
    always @(*)
    begin
        case(ALU_Sel)
        4'b000: // Addition
           ALU_Result = A + B ;
        4'b001: // Subtraction
           ALU_Result = A - B ;
        4'b010: // Multiplication
           ALU_Result = A * B;
        4'b011: // Division
           ALU_Result = A/B;
        4'b100: // Logical shift left
           ALU_Result = A<<1;
         4'b101: // Logical shift right
           ALU_Result = A>>1;
         4'b110: // Rotate left
           ALU_Result = {A[6:0],A[7]};
         4'b111: // Rotate right
           ALU_Result = {A[0],A[7:1]};
         default: ALU_Result = A + B ;
        endcase
    end
endmodule
```

Simulation Results:

Input: A [7:0] = 0000 0011, B [7:0] = 0000 0011, ALU_Sel[2:0]=010

Output: ALU_Out[7:0] = 0000 0001, CarryOut=0

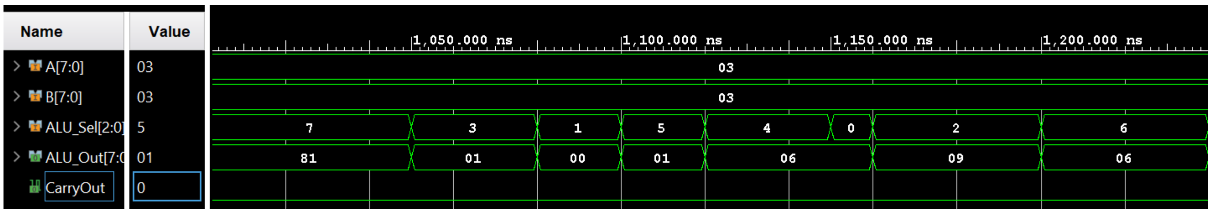


Fig. 7.1: Simulation results of example 7.1

IO Pin Assignment:

Bit	Input A	Input B	ALU_Sel	Output ALU_Out	CarryOut
0	V17	V2	T17	V13	U16
1	V16	T3	U18	V3	
2	W16	T2	W19	W3	
3	W17	R3		U3	
4	W15	W2		P3	
5	V15	U1		N3	
6	W14	T1		P1	
7	W13	R2		L1	

Hardware Results:

Input: A[7:0]

SW7	SW6	SW5	SW4	SW3	SW2	SW1	SW0
OFF	OFF	OFF	OFF	OFF	OFF	ON	ON

Input: B[7:0]

SW15	SW14	SW13	SW12	SW11	SW10	SW9	SW8
OFF	OFF	OFF	OFF	OFF	OFF	ON	ON

Input: ALU_Sel[2:0]

BTNL	BTNC	BTNR
ON	OFF	ON

Output: ALU_Out[7:0]

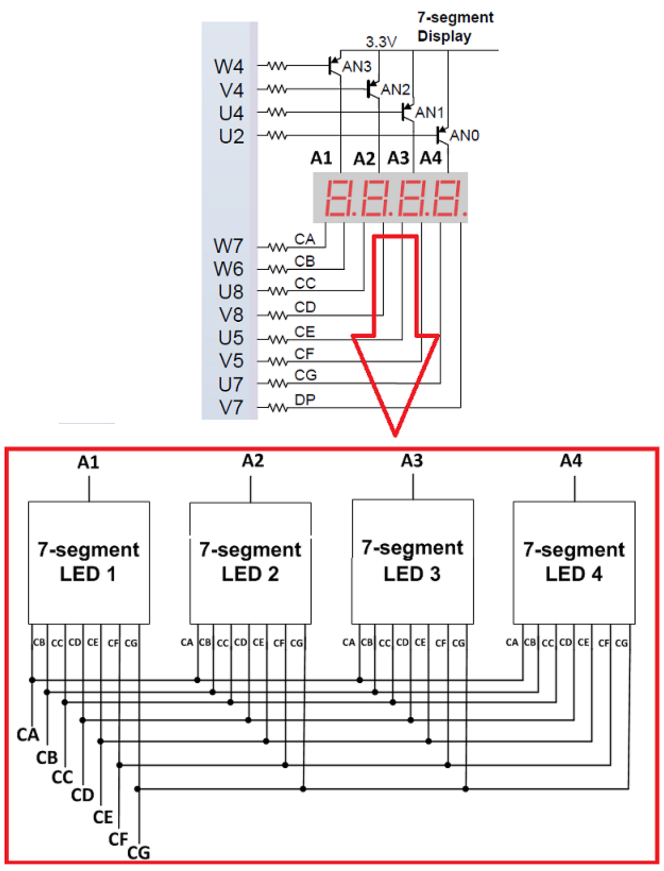
LD15	LD14	LD13	LD12	LD11	LD10	LD9	LD8
OFF	OFF	OFF	OFF	OFF	OFF	OFF	ON

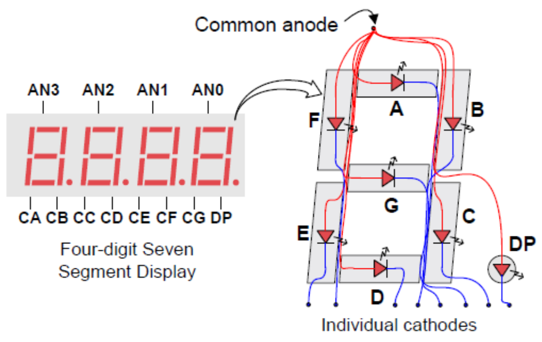
Output: CarryOut

LD0
OFF

EXAMPLE 7.2: Write a Verilog code to display the hexadecimal equivalent of the 16-bit binary input, given through the user switches, on the seven-segment display of the Basys-3 FPGA kit.

Solution:





Verilog Code:

```
module Seven_segment(
    input clock_100Mhz,  // 100 Mhz clock source on Basys 3 FPGA
    input reset,  // Reset
    input [15:0] switch,  // Binary input
    output reg [3:0] Anode_Activate,  // anode signals of  7-segment LED display
    output reg [6:0] LED_out    // cathode patterns of the 7-segment LED display
    );

    reg [3:0] LED_BCD;
    reg [19:0] refresh_counter;
    wire [1:0] Digit_activator;

   always @(posedge clock_100Mhz or posedge reset)
    begin
        if(reset==1)
            refresh_counter <= 0;
        else
            refresh_counter <= refresh_counter + 1;
    end

    assign Digit_activator = refresh_counter[19:18];

    always @(*)
    begin
        case(Digit_activator)
        2'b00: begin
                  Anode_Activate = 4'b0111;
                                      // activate display1 and deactivate display2, 3, 4
                  LED_BCD = switch[15:12];
                                      // the first digit of the 16-bit number
              end
        2'b01: begin
                  Anode_Activate = 4'b1011;
                                  // activate display2 and deactivate display1, 3, 4
                  LED_BCD = switch[11:8];
                                  // the second digit of the 16-bit number
              end
        2'b10: begin
                  Anode_Activate = 4'b1101;
                                  // activate display 3 and deactivate display2, 1, 4
                  LED_BCD = switch[7:4];
                                  // the third digit of the 16-bit number
               end
        2'b11: begin
                  Anode_Activate = 4'b1110;
                                  // activate LED4 and Deactivate LED2, LED3, LED1
                  LED_BCD = switch[3:0]; // the fourth digit of the 16-bit number
               end
        endcase
    end
    // Cathode patterns of the 7-segment LED display
    always @(*)
    begin
        case(LED_BCD)
        4'b0000: LED_out = 7'b0000001; // "0"
        4'b0001: LED_out = 7'b1001111; // "1"
        4'b0010: LED_out = 7'b0010010; // "2"
        4'b0011: LED_out = 7'b0000110; // "3"
        4'b0100: LED_out = 7'b1001100; // "4"
        4'b0101: LED_out = 7'b0100100; // "5"
        4'b0110: LED_out = 7'b0100000; // "6"
        4'b0111: LED_out = 7'b0001111; // "7"
        4'b1000: LED_out = 7'b0000000; // "8"
        4'b1001: LED_out = 7'b0000100; // "9"
        4'b1010: LED_out = 7'b0001000; // "A"
        4'b1011: LED_out = 7'b1100000; // "b"
        4'b1100: LED_out = 7'b0110001; // "C"
        4'b1101: LED_out = 7'b1000010; // "d"
        4'b1110: LED_out = 7'b0110000; // "E"
        4'b1111: LED_out = 7'b0111000; // "F"
        default: LED_out = 7'b1111111;
        endcase
    end
 endmodule
```

Simulation Results:

Input: switch[15:0] = 0100 0101 0110 0011, reset = 1→0, clock=100MHz

Output:

Clock_pulses	Digit_activator	LED_BCD	LED_out (7-segment code)
at 1st pulse	0 (1st 7-seg display)	4	1001100
at (1x2^{18})th pulse	1 (2nd 7-seg display)	5	0100100
at (2x2^{18})th pulse	2 (3rd 7-seg display)	6	0100000
at (3x2^{18})th pulse	3 (4th 7-seg display)	3	0000110

The above sequence keeps repeating at an interval of 2^{18} clock pulses

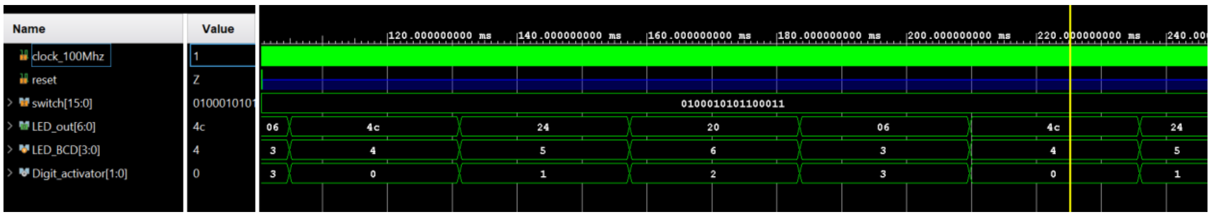


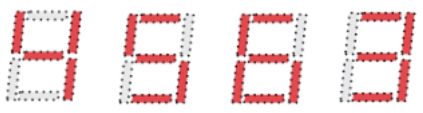
Fig. 7.2: Simulation results of example 7.2

Hardware Results:

Input: switch[15:0]

SW 15	SW 14	SW 13	SW 12	SW 11	SW 10	SW 9	SW 8	SW 7	SW 6	SW 5	SW 4	SW 3	SW 2	SW 1	SW 0
OFF	ON	OFF	OFF	OFF	ON	OFF	ON	OFF	ON	ON	OFF	OFF	OFF	ON	ON

Output:



EXAMPLE 7.3: Write a Verilog code to display the decimal equivalent of the 4-bit binary number, input through the user switches, on the seven-segment display of the Basys 3 kit.

Solution:

Binary Input	Decimal digit - 1	Decimal digit - 0
0000	0	0
0001	0	1
0010	0	2
0011	0	3
0100	0	4
0101	0	5
0110	0	6
0111	0	7
1000	0	8
1001	0	9
1010	1	0
1011	1	1
1100	1	2
1101	1	3
1110	1	4
1111	1	5

Verilog Code:

```
    module Bin_2_Dec(
        input clock_100Mhz,  // 100 Mhz clock source on Basys 3 FPGA
        input reset,  // reset
        input [3:0] switch,  //4-bit Binary input
        output reg [3:0] Anode_Activate,
                                                    // anode signals of the 7-segment LED display
        output reg [6:0] LED_out    // cathode patterns of the 7-segment LED display
    );

    reg Z;
    reg [3:0] LED_BCD;
    reg [19:0] refresh_counter;
    wire [1:0] Digit_activator;

   always @(posedge clock_100Mhz or posedge reset)
    begin
        if(reset==1)
            refresh_counter <= 0;
        else
            refresh_counter <= refresh_counter + 1;
    end

   assign Digit_activator = refresh_counter[18];

   always @(*)
    begin
        Z = (switch>4'b1001)?1:0;
        case(Digit_activator)
        2'b0: begin
                Anode_Activate = 4'b1101;
                LED_BCD = {3'b000,Z};
              end
        2'b1: begin
            Anode_Activate = 4'b1110;
                if(Z==1)
                    LED_BCD = switch-4'b1010;
                else
                    LED_BCD = switch;
                end
        endcase
    end
    // Cathode patterns of the 7-segment LED display
    always @(*)
    begin
        case(LED_BCD)
        4'b0000: LED_out = 7'b0000001; // "0"
        4'b0001: LED_out = 7'b1001111; // "1"
        4'b0010: LED_out = 7'b0010010; // "2"
        4'b0011: LED_out = 7'b0000110; // "3"
        4'b0100: LED_out = 7'b1001100; // "4"
        4'b0101: LED_out = 7'b0100100; // "5"
        4'b0110: LED_out = 7'b0100000; // "6"
        4'b0111: LED_out = 7'b0001111; // "7"
        4'b1000: LED_out = 7'b0000000; // "8"
        4'b1001: LED_out = 7'b0000100; // "9"
        default: LED_out = 7'b1111111; // "0"
        endcase
    end
endmodule
```

Simulation Results:

Input: switch[3:0]=1100, reset = 1→0, clock=100MHz

Output:

Clock_pulses	Digit_activator	LED_BCD	LED_out (7-segment code)
at 1st pulse	0 (3rd 7-seg display)	1	1001111
at $(1x2^{18})$th pulse	1 (4th 7-seg display)	2	0010010

The above sequence keeps repeating at an interval of 2^{18} clock pulses

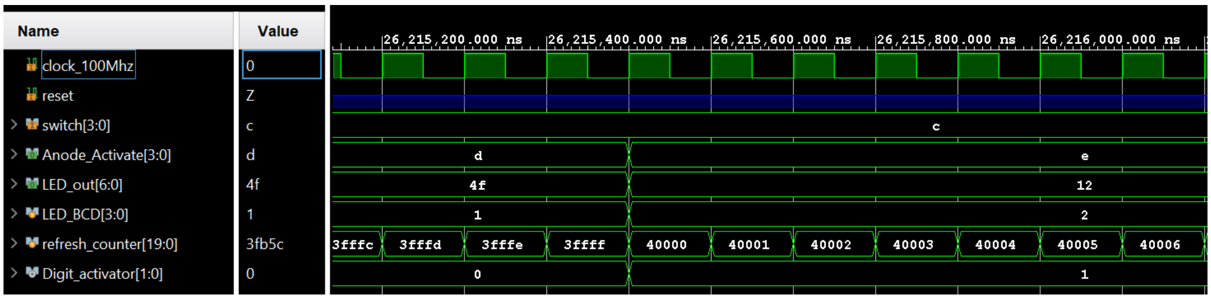


Fig. 7.3: Simulation results of example 7.3

IO Pin Assignment:

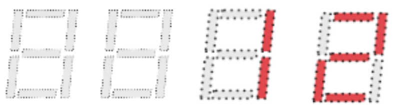
Switch[3:0]		Anode_Activate [3:0]		LED_out [6:0]			
Bit 3	W17	Bit 3: Display 1	W4	Bit 6: Seg a	W7	Bit 2: Seg e	U5
Bit 2	W16	Bit 2: Display 2	V4	Bit 5: Seg b	W6	Bit 1: Seg f	V5
Bit 1	V16	Bit 1: Display 3	U4	Bit 4: Seg c	U8	Bit 0: Seg g	U7
Bit 0	V17	Bit 0: Display 4	U2	Bit 3: Seg d	V8		
Reset: T17				**Clock:** W5			

Hardware Results:

Input: switch[3:0]

SW 3	SW 2	SW 1	SW 0
ON	ON	OFF	OFF

Output:



EXAMPLE 3.4: Write a Verilog code to perform the 4-bit BCD addition and display the results on the seven-segment display of the Basys 3 FPGA kit.

Solution:

BCD inputs (Input1+Input2+Carry)	**BCD sum**	**Binary sum**
0000+0101+0	Carry=0, sum=0101	Carry=0, sum=0101
0001+0101+0	Carry=0, sum=0110	Carry=0, sum=0110
0010+0101+0	Carry=0, sum=0111	Carry=0, sum=0111
0011+0101+0	Carry=0, sum=1101	Carry=0, sum=1000
0100+0101+0	Carry=0, sum=1001	Carry=0, sum=1001
0101+0101+0	Carry=1, sum=0000	Carry=0, sum=1010
0110+0101+0	Carry=1, sum=0001	Carry=0, sum=1011
0111+0101+0	Carry=1, sum=0010	Carry=0, sum=1100
1000+0101+0	Carry=1, sum=0011	Carry=0, sum=1101
1001+0101+0	Carry=1, sum=0100	Carry=0, sum=1110

On the 4-digit segment display of the Basys 3 kit, the binary sum is programmed to display on the leftmost two digits, and the BCD sum to be displayed on the rightmost two digits.

Verilog Code:

```
module BCD_Addition(
        input clock_100Mhz,  // 100 Mhz clock source on Basys 3 FPGA
        input reset,  // reset
        input [3:0] a,b,  // two BCD inputs
        input carry_in,  // Initial carry
        output reg [3:0] Anode_Activate,
                                        // anode signals of the 7-segment LED display
        output reg [6:0] LED_out  // cathode patterns of the 7-segment LED display
    );
```

```
 reg [3:0] sum;
 reg carry;
 reg bin_carry;
 reg [3:0] bin_sum;
 reg [3:0] LED_BCD;
 reg [19:0] refresh_counter;
 wire [1:0] Digit_activator;

 always @(a,b,carry_in)
 begin
     {bin_carry, bin_sum} = a+b+carry_in;  //add all the inputs
     if(bin_sum > 9)
      begin
         carry = 1;                    //set the carry output
         sum = bin_sum+6;              //add 6, if result is more than 9.
      end
      else
      begin
         carry = 0;
         sum = bin_sum[3:0];
     end
 end

always @(posedge clock_100Mhz or posedge reset)
 begin
     if(reset==1)
         refresh_counter <= 0;
     else
         refresh_counter <= refresh_counter + 1;
 end

assign Digit_activator = refresh_counter[19:18];

always @(*)
 begin
     case(Digit_activator)
     2'b00: begin
             Anode_Activate = 4'b1110;
             LED_BCD = sum;
           end
     2'b01: begin
         Anode_Activate = 4'b1101;
         LED_BCD = {3'b000,carry};
         end
```

```
        2'b10: begin
                Anode_Activate = 4'b1011;
                LED_BCD = bin_sum;
              end
        2'b11: begin
            Anode_Activate = 4'b0111;
            LED_BCD = {3'b000,bin_carry};
            end
        endcase
    end
```

// Cathode patterns of the 7-segment LED display

```
    always @(*)
    begin
        case(LED_BCD)
        4'b0000: LED_out = 7'b0000001; // "0"
        4'b0001: LED_out = 7'b1001111; // "1"
        4'b0010: LED_out = 7'b0010010; // "2"
        4'b0011: LED_out = 7'b0000110; // "3"
        4'b0100: LED_out = 7'b1001100; // "4"
        4'b0101: LED_out = 7'b0100100; // "5"
        4'b0110: LED_out = 7'b0100000; // "6"
        4'b0111: LED_out = 7'b0001111; // "7"
        4'b1000: LED_out = 7'b0000000; // "8"
        4'b1001: LED_out = 7'b0000100; // "9"
        4'b1010: LED_out = 7'b0001000; // "A"
        4'b1011: LED_out = 7'b1100000; // "b"
        4'b1100: LED_out = 7'b0110001; // "C"
        4'b1101: LED_out = 7'b1000010; // "d"
        4'b1110: LED_out = 7'b0110000; // "E"
        4'b1111: LED_out = 7'b0111000; // "F"
        default: LED_out = 7'b1111111; // "0"
        endcase
    end
 endmodule
```

Simulation Results:

Input: a[3:0] = 1000, b[3:0]=0101, carry_in=0, reset = 1→0, clock=100MHz

Output:

Clock_pulses	Digit_activator	LED_BCD	LED_out (7-segment code)
at 1st pulse	0 (1st 7-seg display)	0	0000001
at $(1x2^{18})$th pulse	1 (2nd 7-seg display)	d	1000010

at (2x2^{18})th pulse	2 (3rd 7-seg display)	1	1001111
at (3x2^{18})th pulse	3 (4th 7-seg display)	3	0000110

The above sequence keeps repeating at an interval of 2^{18} clock pulses

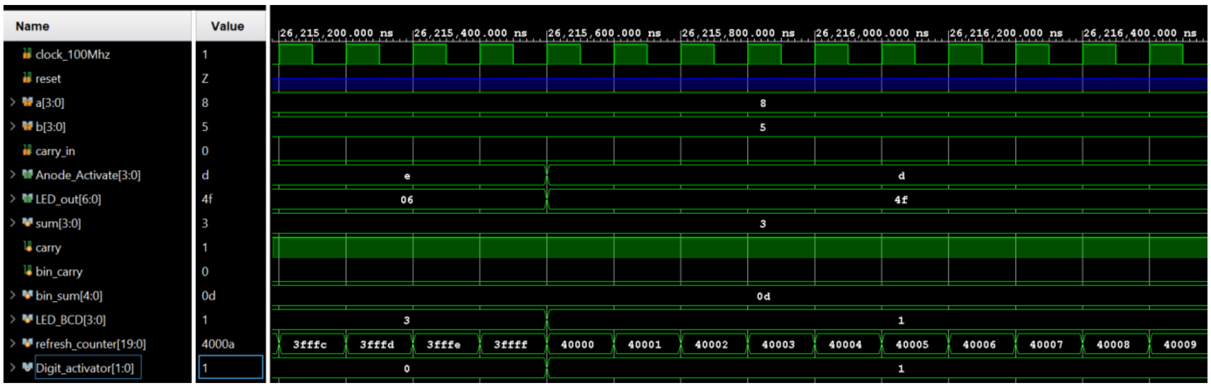


Fig. 7.4: Simulation results of example 7.4

IO Pin Assignment:

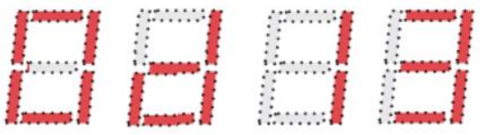
a[3:0]	B[3:0]	Anode_Activate [3:0]		LED_out [6:0]			
W17	W13	Bit 3: Display 1	W4	Bit 6: Seg a	W7	Bit 2: Seg e	U5
W16	W14	Bit 2: Display 2	V4	Bit 5: Seg b	W6	Bit 1: Seg f	V5
V16	V15	Bit 1: Display 3	U4	Bit 4: Seg c	U8	Bit 0: Seg g	U7
V17	W15	Bit 0: Display 4	U2	Bit 3: Seg d	V8		
Reset: T17		**carry_in:** R2		**Clock:** W5			

Hardware Results:

Input:

carry_in	b[3:0]				a[3:0]			
SW 15	**SW 7**	**SW 6**	**SW 5**	**SW 4**	**SW 3**	**SW 2**	**SW 1**	**SW 0**
OFF	**OFF**	**ON**	**OFF**	**ON**	**ON**	**OFF**	**OFF**	**OFF**

Output:

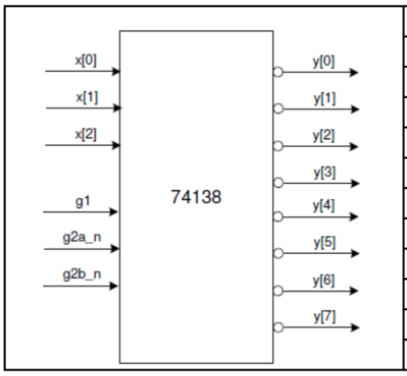


EXERCISE PROBLEMS:

1. Write the structural Verilog code to realize a 3x1 multiplexer using 2x1 multiplexers, synthesize, and implement it through the onboard user switches and LEDs of the Basys 3 FPGA kit.
2. Write a Verilog code to display the push button's status on the seven-segment display of the Basys 3 kit as follows:

Pressed Push Button	**Character to be displayed on the seven segment display**
BTNR	P
BTNU	U
BTNL	L
BTND	d
BTNC	C

3. Write a data flow Verilog code to create a 4-bit ripple carry adder, synthesize, and implement it on Basys 3 hardware.
4. Design and implement a popular IC, 74138, functionality using dataflow modeling of Verilog. The IC symbol and truth table are given below. Verify the functionality on Basys 3 hardware.



g_1 g_{2a_n} g_{2b_n}	x_0 x_1 x_2	y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7
0 x x	x x x	1 1 1 1 1 1 1 1
x 1 x	x x x	1 1 1 1 1 1 1 1
x x 1	x x x	1 1 1 1 1 1 1 1
1 0 0	0 0 0	0 1 1 1 1 1 1 1
1 0 0	0 0 1	1 0 1 1 1 1 1 1
1 0 0	0 1 0	1 1 0 1 1 1 1 1
1 0 0	0 1 1	1 1 1 0 1 1 1 1
1 0 0	1 0 0	1 1 1 1 0 1 1 1
1 0 0	1 0 1	1 1 1 1 1 0 1 1
1 0 0	1 1 0	1 1 1 1 1 1 0 1
1 0 0	1 1 1	1 1 1 1 1 1 1 0

5. Design a comparator that compares two 2-bit numbers (A and B) and assert outputs indicating whether the decimal equivalent of word A is less than, greater than, or equal to that of word B. Realize it on Basys 3 hardware.

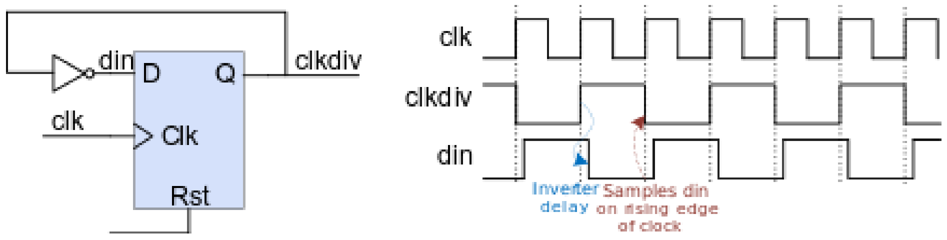
Experiment No. 8

Clocking of Artix-7 FPGA on Basys3 Kit

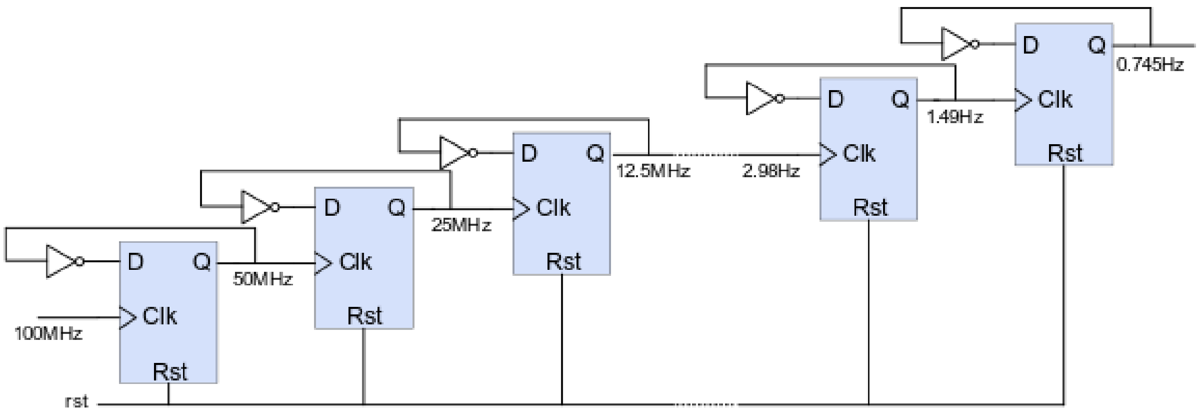
OBJECTIVE: To understand the clocking mechanism for Artix-7 FPGA on Basys3 kit by realizing various sequential circuits through Verilog.

THEORY: Sequential digital circuits need a clock signal for them to function. Usually, the clock signal comes from a crystal oscillator on-board. The Basys3 board includes a single 100MHz oscillator connected to pin W5. However, some peripheral controllers do not need such a high frequency to operate. Therefore, the frequency of the clock must be slowed down.

There is a simple circuit that can divide the clock frequency by half:



The clock can further be divided by cascading the above circuit:



Each stage divided the frequency by 2.

After the first stage, the frequency becomes: $\frac{100MHz}{2} = 50MHz.$

After the second stage, the frequency becomes: $\frac{100MHz}{2*2} = 25MHz.$

After the third stage, the frequency becomes: $\frac{100MHz}{2*2*2} = 12.5MHz.$

…….

Like this, after the n^{th} stage, the frequency becomes:

$$\frac{100MHz}{\underbrace{2\cdot 2\cdot \ldots \cdot 2}_{n}} = \frac{100,000,000}{2^n}Hz$$

For example, the output of a 3-stage clock divider circuit can be seen as a 3-bit up counter. At each bit, the output is a square wave of frequency $\frac{f_{clk}}{2^{(bit\ index)+1}}$, which can be used as a slowed-down clock:

	Stage-3 output: Square wave of frequency $\frac{f_{clk}}{2^{(2)+1}}$	**Stage-2 output:** Square wave of frequency $\frac{f_{clk}}{2^{(1)+1}}$	**Stage-1 output:** Square wave of frequency $\frac{f_{clk}}{2^{(0)+1}}$
f_{clk}	Q[2]	Q[1]	Q[0]
0→1	0	0	0
0→1	0	0	1
0→1	0	1	0
0→1	0	1	1
0→1	1	0	0
0→1	1	0	1
0→1	1	1	0
0→1	1	1	1
0→1	0	0	0

EXAMPLE 8.1:

Write a Verilog description to slow down the input clock frequency of 100MHz to blink the onboard LED of Basys3 kit at 0.745 Hz.

Solution:

$$\frac{Input\ clock\ frequency}{2^{(bit\ index)+1}} = desired\ frequency$$

$$\frac{100\ MHz}{2^{(bit\ index)+1}} = 0.745\ Hz$$

$$2^{(bit\ index)+1} = 134228187.9\ Hz$$

$$(bit\ index) + 1 = \frac{log\ 134228187.9}{log\ 2}$$

$$(bit\ index) + 1 = 27$$

$$bit\ index = 26$$

Therefore, design a 27-bit up counter and consider its MSB (i.e., 26th bit) output as the slowed-down clock to blink the onboard LED

Verilog Code:

```
  module Clk_Div(input clk, reset, output counter);
   reg [26:0] counter_up;

   // up counter
   always @(posedge clk or posedge reset)
   begin
       if(reset)
           counter_up <= 0;
       else
           counter_up <= counter_up + 1;
   end
   assign counter = counter_up[26];
endmodule
```

Simulation Results:

Input: clk = 10ns (100MHz), reset = 1→0

Output: counter = 1.34s (0.745Hz) square wave

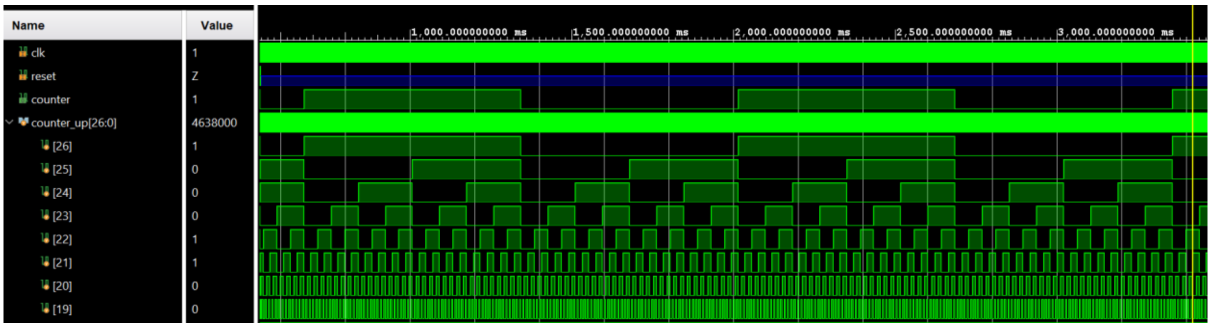


Fig. 8.1: Simulation results of example 8.1

IO pin assignment:

clk: W5	reset: T17	counter: U16

Hardware results:

LED "LD0" of BASYS3 blinks at the rate of 1.34 seconds.

EXAMPLE 8.2:

Write a Verilog description to blink the onboard LED of the Basys3 kit at the rate of 1 second.

Solution:

The relation between the input frequency and the desired frequency, given in example 8.1, doesn't always result in the "*bit index,*" an integer.

For example, for the desired frequency of 1Hz (or 1s):

$$\frac{100\ \mathrm{MHz}}{2^{(\mathrm{bit\ index})+1}} = 1\mathrm{Hz}$$

$$2^{(\mathrm{bit\ index})+1} = 100\ \mathrm{MHz}$$

$$(\mathrm{bit\ index}) + 1 = \frac{\log 100000000}{\log 2}$$

$$(\mathrm{bit\ index}) + 1 = 26.57$$

$$\mathrm{bit\ index} = 25.57$$

where the "*bit index*" must be adjusted to either 25 (which corresponds to 1.49Hz or 0.67s) or 26 (which corresponds to 0.745 Hz or 1.34s). Therefore, this example gives an alternative method of generating the desired clock of period 1 second.

Verilog Code:

```
module Clk_Div(input clk, reset, output counter);
  reg [26:0] one_second_counter;
  // up counter
  always @(posedge clk or posedge reset)
  begin
    if(reset==1)
        begin
        one_second_counter = 0;
        end
    else
        begin
         one_second_counter = one_second_counter + 1;
         if(one_second_counter==99999999)
           begin
           one_second_counter = 0;
           end
        end
   end
   assign counter = (one_second_counter>49999999)?1:0;
endmodule
```

Simulation Results:

Input: clk = 10ns (100MHz), reset = 1→0; **Output**: counter = 1s (1Hz) square wave

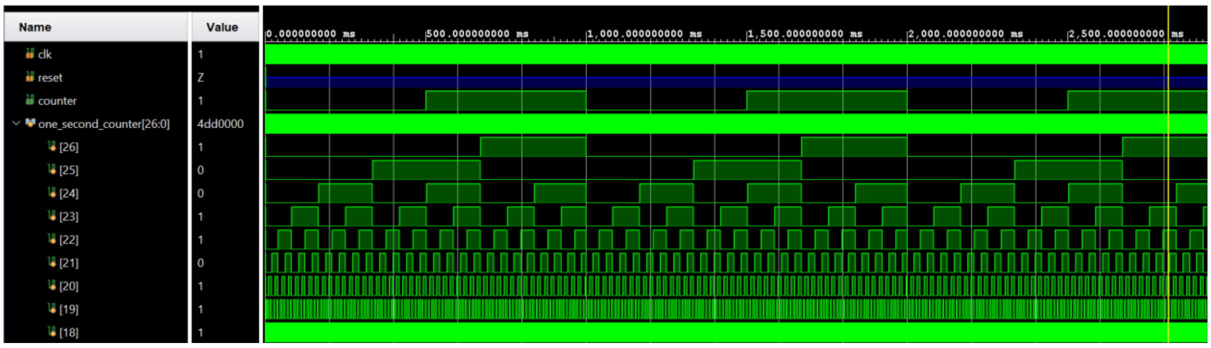


Fig. 8.2: Simulation results of example 8.2

IO pin assignment:

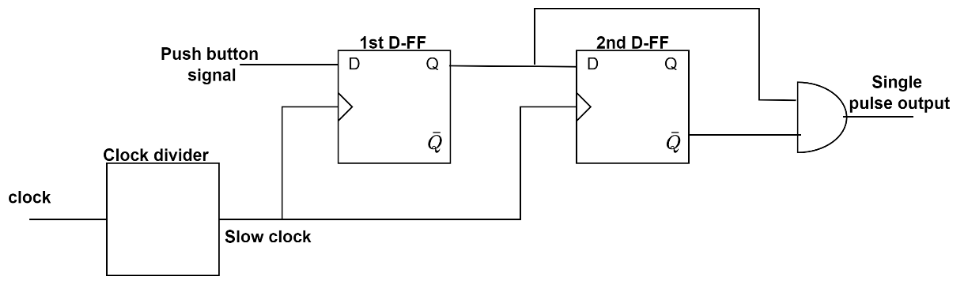
clk: W5	reset: T17	counter: U16

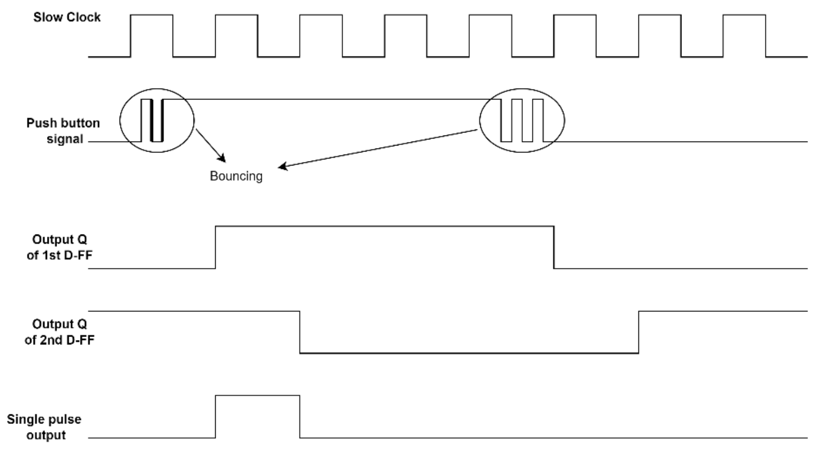
Hardware results:

LED “LD0” of BASYS3 blinks at the rate of 1 second.

EXAMPLE 8.3: Write a Verilog code to debounce the on-board push button of the Basys 3 FPGA kit.

Solution: Mechanical buttons cause an unpredictable bounce in the signal when toggled. Therefore, a simple debouncing circuit that can generate only a single pulse when the button on FPGA is pressed is:





Verilog Code:

```
module debounce(input pb_1,clk,
     output pb_out,slow_clk,Q1,Q2,Q2_bar);

     wire slow_clk;
     wire Q1,Q2,Q2_bar;
     clock_div u1(clk,slow_clk);

     my_dff d1(slow_clk, pb_1,Q1 );
     my_dff d2(slow_clk, Q1,Q2 );
     assign Q2_bar = ~Q2;
     assign pb_out = Q1 & Q2_bar;
endmodule
```

```
// Slow clock for debouncing
module clock_div(input Clk_100M, output reg slow_clk);
     reg [26:0]counter=0;

     always @(posedge Clk_100M)
      begin
         counter <= (counter>=249999)?0:counter+1;
                                                    //Uncomment it for simulation
         slow_clk <= (counter <125000)?1'b0:1'b1;
                                                    //Uncomment it for simulation
       //counter <= (counter>=99999999)?0:counter+1;
                                                    // Uncomment it for HW implementation
       //  slow_clk <= (counter <49999999)?1'b0:1'b1;
                                                    // Uncomment it for HW implementation
      end
endmodule
```

```
// D-flip-flop for debouncing module
module my_dff(input DFF_CLOCK, D, output reg Q);
    always @ (posedge DFF_CLOCK) begin
        Q <= D;
    end
endmodule
```

Test bench:

```
`timescale 1ns / 1ps
module tb_button;
     reg pb_1;
     reg clk;
```

```
    wire pb_out;
    // Instantiate the debouncing Verilog code
    debounce uut (
    .pb_1(pb_1),
    .clk(clk),
    .pb_out(pb_out),
    .slow_clk(slow_clk),
    .Q1(Q1),
    .Q2(Q2),
    .Q2_bar(Q2_bar)
    );

initial
    begin
    clk = 0;
    forever #5 clk = ~clk;
    end

initial
    begin
    pb_1 = 0;
    #3000000;  //3ms
    pb_1=1;
    #100000;  //0.1ms
    pb_1 = 0;
    #100000;  //0.1ms
    pb_1=1;
    #100000;  //0.1ms
    pb_1 = 0;
    #100000;   //0.1ms
    pb_1=1;
    #200000;  //0.2ms
    pb_1 = 0;
    #100000;//0.1ms
    pb_1=1;
    #200000;  //0.2ms
    pb_1 = 0;
    #100000;//0.1ms
    pb_1=1;
    #10000000;  //10ms
    pb_1 = 0;
    #100000;  //0.1ms
    pb_1=1;
    #200000;//0.2ms
```

```
      pb_1 = 0;
      #100000;//0.1ms
      pb_1=1;
      #100000;  //0.1ms
      pb_1 = 0;
      #100000;//0.1ms
      pb_1=1;
      #100000;//0.1ms
      pb_1 = 0;
      #20000000;  //20ms
      pb_1=1;
      #100000;  //0.1ms
      pb_1 = 0;
      #100000;//0.1ms
      pb_1=1;
      #100000;  //0.1ms
      pb_1 = 0;
      #100000;//0.1ms
      pb_1=1;
      #200000;//0.2ms
      pb_1 = 0;
      #100000;//0.1ms
      pb_1=1;
      #200000;  //0.2ms
      pb_1 = 0;
      #100000;//0.1ms
      pb_1=1;
      #10000000;  //10ms
      pb_1 = 0;
      #100000;  //0.1ms
      pb_1=1;
      #200000;//0.2ms
      pb_1 = 0;
      #100000;//0.1ms
      pb_1=1;
      #100000;  //0.1ms
      pb_1 = 0;
      #100000;//0.1ms
      pb_1=1;
      #100000;//0.1ms
      pb_1 = 0;
    end
  endmodule
```

Simulation Results:

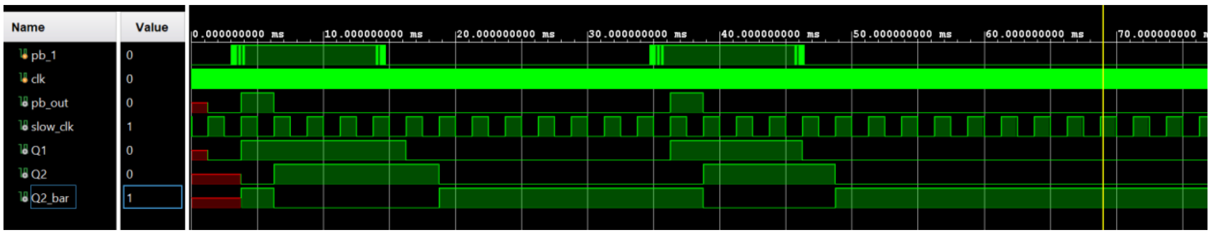


Fig. 8.3: Simulation results of example 8.3

IO Pin assignment:

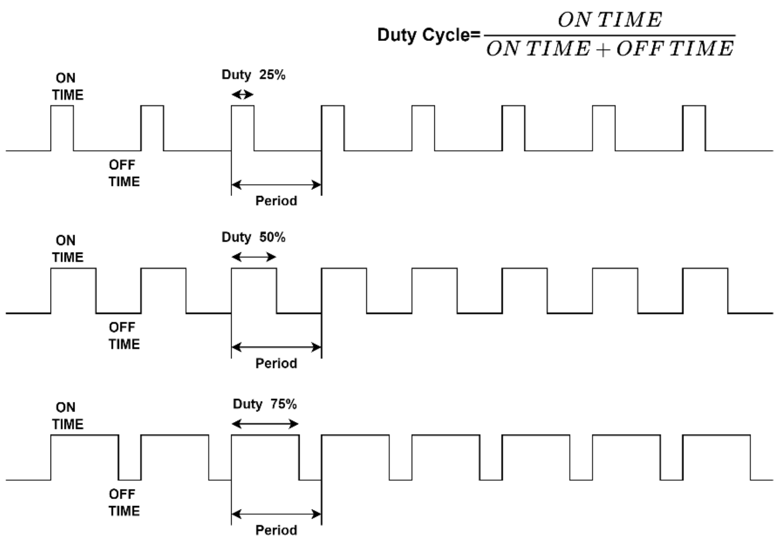
Input	Pin No	Output	Pin No
pb_1	T17	pb_out	U16
clk	W5	slow_clk	V14
		Q1	L1
		Q2	P1
		Q2_bar	N3

Hardware Results: Variation of output due to button BTNR press

BTNR	**Before press**	**Press and Hold**	**Release**
LD7	Blinks at rate of 1s	Blinks at rate of 1s	Blinks at a rate of 1s
LD15	OFF	ON	OFF at next active edge of slow_clk
LD14	OFF	ON at the next active edge of slow_clk	OFF at the second active edge of slow_clk
LD13	ON	OFF at the next active edge of slow_clk	ON at the second active edge of slow_clk
LD0	OFF	ON at the next active edge of slow_clk and then immediately OFF at the next active edge	OFF

EXAMPLE 8.4: Realize a pulse width modulated(PWM) wave of varying duty cycles on the onboard LED of basys-3 through the Verilog description

Solution:



Verilog Code:

```
module PWM
 (
 input clk, // 100MHz clock input
 input increase_duty,  // input to increase 10% duty cycle
 input decrease_duty,  // input to decrease 10% duty cycle
 output duty_inc,
 output duty_dec,
 output PWM_OUT,  // 1Hz PWM output signal
 output PWM_clk,
 output DUTY_CYCLE,
 output reg [3:0] Anode_Activate,  // anode signals of  7-segment LED display
 output reg [6:0] LED_out    // cathode patterns of the 7-segment LED display
    );

 reg [26:0] one_second_counter=0;
 wire PWM_clk;
 wire tmp1,tmp2,duty_inc;// temporary flip-flop signals for debouncing the increasing button
 wire tmp3,tmp4,duty_dec;// temporary flip-flop signals for debouncing the decreasing button
 reg[3:0] counter_PWM=0;  // counter for creating PWM signal
 reg[3:0] DUTY_CYCLE=5;  // initial duty cycle is 50%

 // PWM clock
 always @(posedge clk)
  begin
        one_second_counter = one_second_counter + 1;
   //    if(one_second_counter==2)  //Comment it for FPGA implementation
        if(one_second_counter>99999999)  //Comment it for simulation
          begin
          one_second_counter = 0;
          end
   end
 //assign PWM_clk = (one_second_counter>0)?1:0;
                                              //Comment it for FPGA implementation
   assign PWM_clk = (one_second_counter>49999999)?1:0;
                                                   // Comment it for simulation

 // debouncing FFs for increasing button
 DFF_PWM PWM_DFF1(PWM_clk,increase_duty,tmp1);
 DFF_PWM PWM_DFF2(PWM_clk,tmp1, tmp2);
 assign duty_inc =  tmp1 & (~ tmp2);
 // debouncing FFs for decreasing button
 DFF_PWM PWM_DFF3(PWM_clk,decrease_duty, tmp3);
 DFF_PWM PWM_DFF4(PWM_clk,tmp3, tmp4);
 assign duty_dec =  tmp3 & (~ tmp4);

// vary the duty cycle using the debounced buttons above
 always @(posedge PWM_clk)
 begin
   if(duty_inc==1 && DUTY_CYCLE <= 9)
    DUTY_CYCLE <= DUTY_CYCLE + 1; // increase duty cycle by 10%
   else if(duty_dec==1 && DUTY_CYCLE>=1)
    DUTY_CYCLE <= DUTY_CYCLE - 1; //decrease duty cycle by 10%
 end

// Create PWM signal with variable duty cycle controlled by 2 buttons
 always @(posedge PWM_clk)
 begin
   counter_PWM <= counter_PWM + 1;
   if(counter_PWM>=9)
    counter_PWM <= 0;
 end
 assign PWM_OUT = counter_PWM < DUTY_CYCLE ? 1:0;

 always @(*)
    begin
        Anode_Activate = 4'b1110;
        case(DUTY_CYCLE)
        4'b0000: LED_out = 7'b0000001; // "0"
        4'b0001: LED_out = 7'b1001111; // "1"
        4'b0010: LED_out = 7'b0010010; // "2"
        4'b0011: LED_out = 7'b0000110; // "3"
        4'b0100: LED_out = 7'b1001100; // "4"
        4'b0101: LED_out = 7'b0100100; // "5"
        4'b0110: LED_out = 7'b0100000; // "6"
        4'b0111: LED_out = 7'b0001111; // "7"
        4'b1000: LED_out = 7'b0000000; // "8"
        4'b1001: LED_out = 7'b0000100; // "9"
        default: LED_out = 7'b0000001; // "0"
        endcase
    end
endmodule

// Debouncing DFFs for push buttons on FPGA
module DFF_PWM(clk,D,Q);
input clk,D;
output reg Q;
always @(posedge clk)
```

```
begin
  Q <= D;
end
endmodule
```

Test bench:

```
`timescale 1ns / 1ps
module tb_PWM_Generator_Verilog;
 // Inputs
 reg clk;
 reg increase_duty;
 reg decrease_duty;
 // Outputs
 wire PWM_OUT;
 wire PWM_clk;
 wire duty_inc;
 wire duty_dec;
 wire [3:0]Anode_Activate;
 wire [6:0]LED_out;
 wire [3:0]DUTY_CYCLE;

// Instantiate the PWM Generator with variable duty cycle in Verilog
 PWM PWM_Generator_Unit(
  .clk(clk),
  .increase_duty(increase_duty),
  .decrease_duty(decrease_duty),
  .PWM_OUT(PWM_OUT),
  .PWM_clk(PWM_clk),
  .duty_inc(duty_inc),
  .duty_dec(duty_dec),
  .Anode_Activate(Anode_Activate),
  .LED_out(LED_out),
  .DUTY_CYCLE(DUTY_CYCLE)
 );

// Create 100Mhz clock
 initial begin
 clk = 0;
 forever #5 clk = ~clk;
 end
 initial begin
  increase_duty = 0;
  decrease_duty = 0;
  #1000;
```

```
     increase_duty = 1;
    #100;  // increase duty cycle by 10%
      increase_duty = 0;
    #100;
      increase_duty = 1;
    #100;  // increase duty cycle by 10%
      increase_duty = 0;
    #100;
      increase_duty = 1;
    #100;  // increase duty cycle by 10%
      increase_duty = 0;
    #100;
      decrease_duty = 1;
    #100;  // decrease duty cycle by 10%
      decrease_duty = 0;
    #100;
      decrease_duty = 1;
    #100;  // decrease duty cycle by 10%
      decrease_duty = 0;
    #100;
      decrease_duty = 1;
    #100;  // decrease duty cycle by 10%
      decrease_duty = 0;
    #100;
      decrease_duty = 1;
    #100;  // decrease duty cycle by 10%
      decrease_duty = 0;
    #100;
      decrease_duty = 1;
    #100;  // decrease duty cycle by 10%
      decrease_duty = 0;
  end
endmodule
```

Simulation Results:

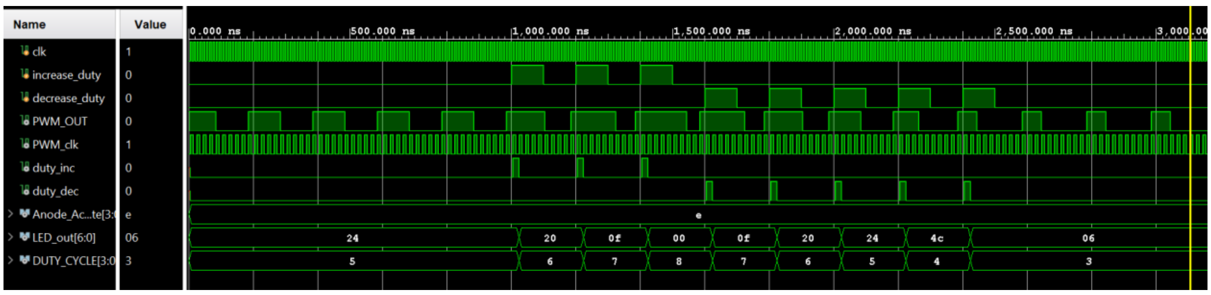


Fig. 8.4: Simulation results of example 8.4

IO Pin Assignment:

		Anode_Activate [3:0]		**LED_out [6:0]**			
Increase_duty	T18	Bit 3: Display 1	W4	Bit 6: Seg a	W7	Bit 2: Seg e	U5
Decrease_duty	U17	Bit 2: Display 2	V4	Bit 5: Seg b	W6	Bit 1: Seg f	V5
Clock	W5	Bit 1: Display 3	U4	Bit 4: Seg c	U8	Bit 0: Seg g	U7
Duty_inc	N3	Bit 0: Display 4	U2	Bit 3: Seg d	V8		
Duty_dec	P3						
PWM_clk	L1		**DYTY_CYCLE [3:0]**				
PWM_OUT	V3		Bit3: V19	Bit2: U19	Bit1: E19	Bit0: U16	

Hardware Results:

Initial state		**BTNU press**	**BTNU press**	**BTND press**	**BTND press**	**BTND press**	**BTND press**
LD5	Blinks at rate of 1s	Blinks at rate of 1s	Blinks at rate of 1s	Blinks at rate of 1s	Blinks at rate of 1s	Blinks at rate of 1s	Blinks at rate of 1s
SSD	5	6	7	6	5	4	3
LD9	PWM output of 50%duty cycle	PWM output of 60% duty cycle	PWM output of 70% duty cycle	PWM output of 60% duty cycle	PWM output of 50% duty cycle	PWM output of 40% duty cycle	PWM output of 30% duty cycle
LD13	OFF	ON for 1 PWM clock cycle	ON for 1 PWM clock cycle	OFF	OFF	OFF	OFF
LD12	OFF	OFF	OFF	ON for 1 PWM clock cycle	ON for 1 PWM clock cycle	ON for 1 PWM clock cycle	ON for 1 PWM clock cycle
V19	OFF	OFF	OFF	OFF	OFF	OFF	OFF
U19	ON	ON	ON	ON	ON	ON	OFF
E19	OFF	ON	ON	ON	OFF	OFF	ON
U16	ON	OFF	ON	OFF	ON	OFF	ON

EXERCISE PROBLEMS:

1. Write a Verilog description for the “1011” sequence detector using Moore FSM. Simulate, synthesize, and implement it on the on-board peripherals of Basys3 FPGA kit.
2. Write a Verilog description to design an 8-bit counter using T flip-flops. Your design needs to be hierarchical, using a T flip-flop in behavioral modeling, and the rest should be either in dataflow or gate-level modeling. Develop a testbench and validate the design. Assign Clock input, Clear_n, Enable, and Q. Implement the design and verify the functionality in hardware.
3. Write a Verilog description to implement a 4-digit ring counter on the 7-segment display. Verify its functionality on Basys3 FPGA kit.

Experiment No. 9

FPGA System Design using Vivado IP Integrator

OBJECTIVE: To construct the block-level digital system design using the Vivado IP integrator (IPI) and realize them on the Basys3 FPGA kit.

THEORY: This experiment is designed to introduce students to the digital design flow using Vivado IPI for Artix-7 FPGAs.

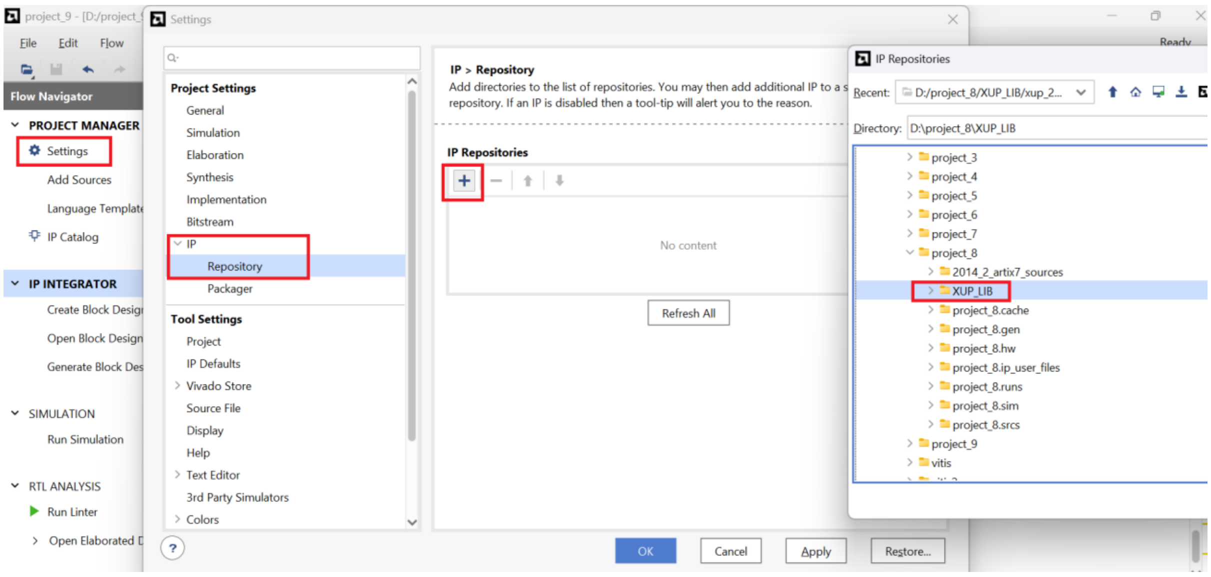
The Vivado IPI allows the creation of digital designs in schematic view through the basic functional IP blocks. The basic functional IP blocks are available on your desktop under "XUP_LIB" directory. Extract it into your working directory.

The design flow to create a digital circuit using Vivado IPI is detailed here, and summarized below for quick reference:

Step 1: Create a new project following the steps detailed in Exp.1. However, do not add any source files while creating the project.

Step 2: In the flow navigator, click "create block design," assign a meaningful name to your design, and click "OK."

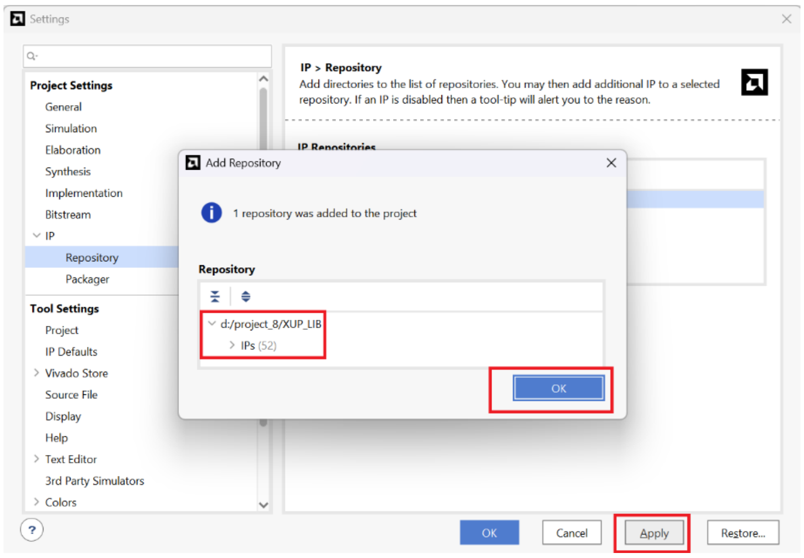
Step 3: Click "Settings" in the flow navigator, click "IP" and then "Repository", click "+", browse to the "XUP_LIB" directory, and then select it.



IP > Repository
Add directories to the list of repositories. You may then add additional IP to a s
repository. If an IP is disabled then a tool-tip will alert you to the reason.

Recent: D:/project_8/XUP_LIB/xup_2...
Directory: D:\project_8\XUP_LIB

The added repository will be shown on the pop-up; check it once and then click "OK" followed by a click on "Apply" and "OK."



IP > Repository

Add directories to the list of repositories. You may then add additional IP to a selected repository. If an IP is disabled then a tool-tip will alert you to the reason.

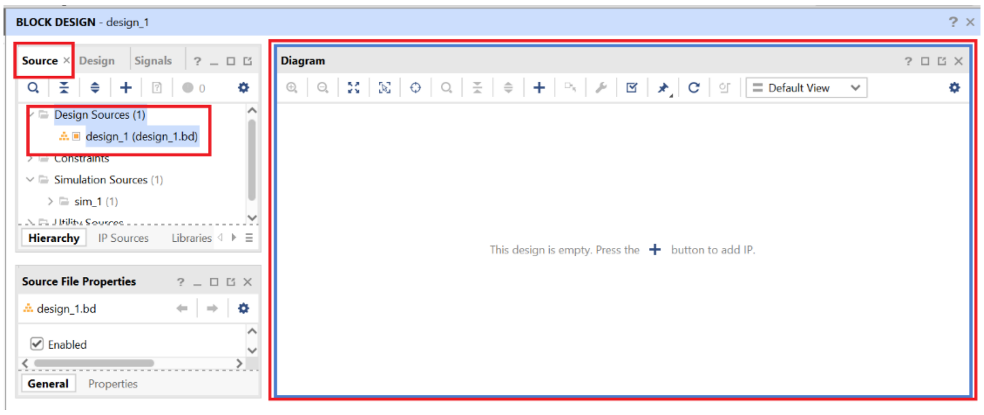
1 repository was added to the project

Repository

d:/project_8/XUP_LIB

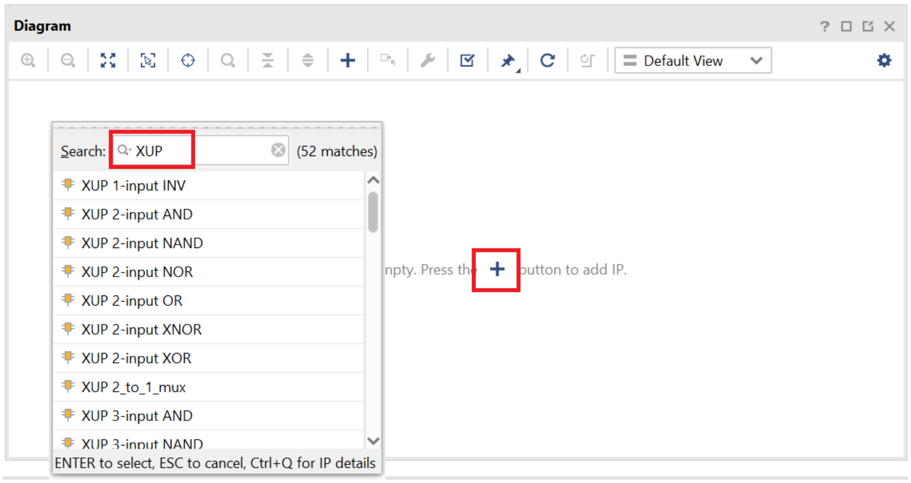
IPs (52)

Step 4: Under the "Source" tab, expand the "Design sources" tab to see the created block design file (with .bd extension). Double-click it to open its design canvas.



This design is empty. Press the + button to add IP.

Step 5: Click "+" and type "XUP" in the search tab. You can see all the IP blocks available in the XUP directory that you can utilize for your design.



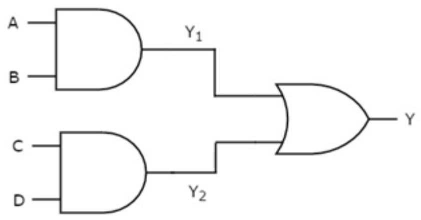
Search: XUP (52 matches)

XUP 1-input INV
XUP 2-input AND
XUP 2-input NAND
XUP 2-input NOR
XUP 2-input OR
XUP 2-input XNOR
XUP 2-input XOR
XUP 2_to_1_mux
XUP 3-input AND
XUP 3-input NAND

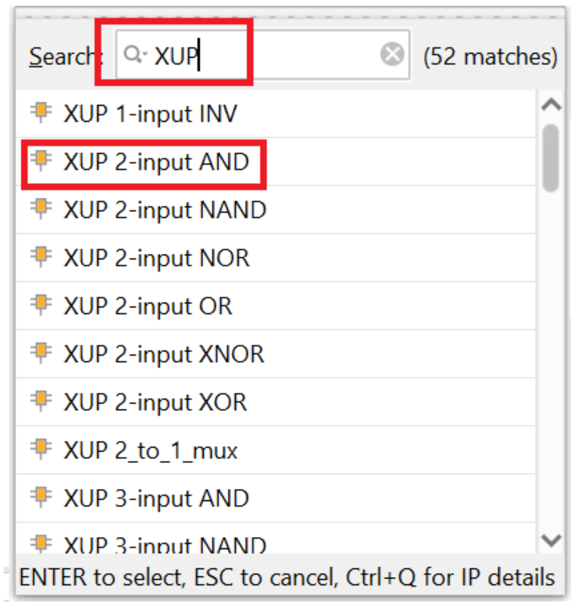
ENTER to select, ESC to cancel, Ctrl+Q for IP details

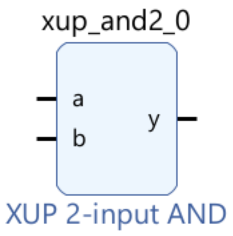
npty. Press the + button to add IP.

Step 6: Design a simple example shown below using the IPI:

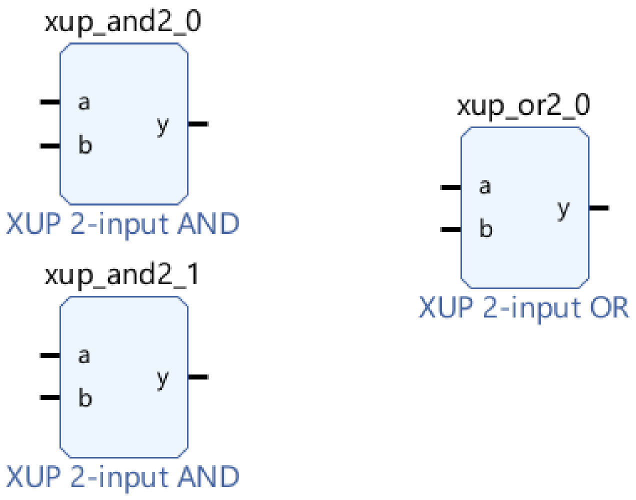


Double-click on “XUP 2-input AND” to add it to the design canvas:

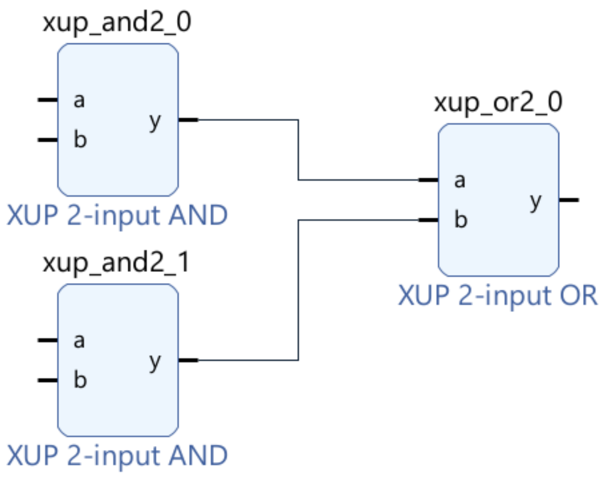




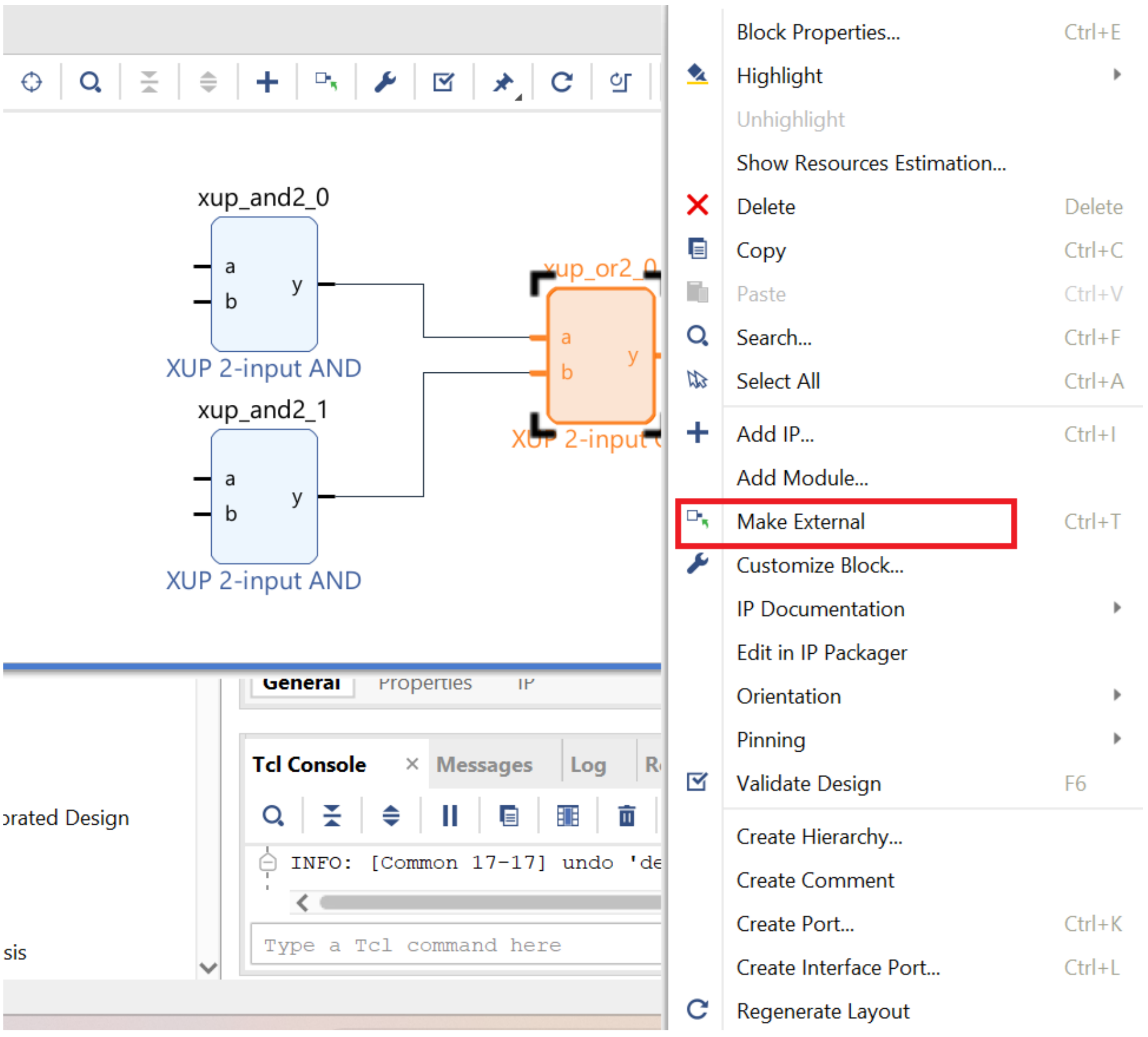
Similarly, add another instance of “XUP 2-input AND”, and an instance of “XUP 2-input OR”:



Interconnect the outputs of 2-input AND gates and the inputs of 2-input:



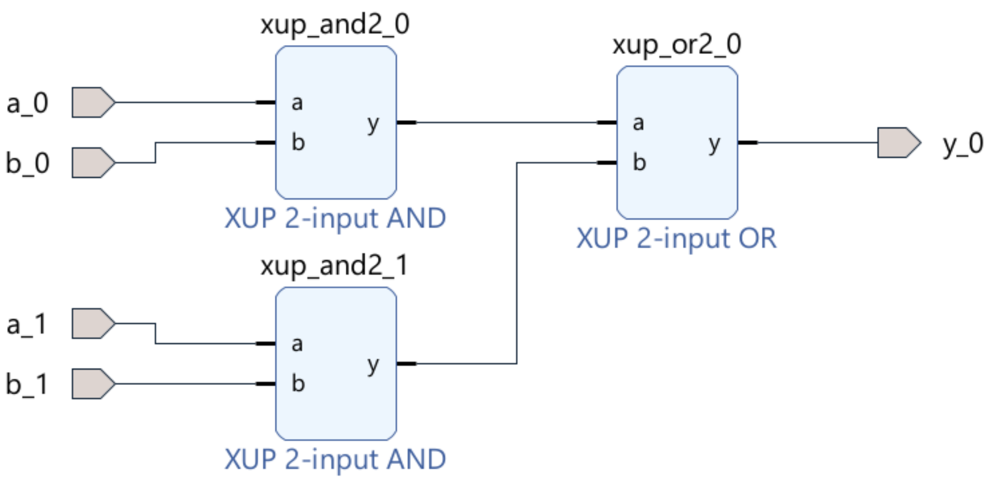
Right-click at the inputs of AND gates as well as at the output of OR gate one by one, and click "make external" to add external ports to them:



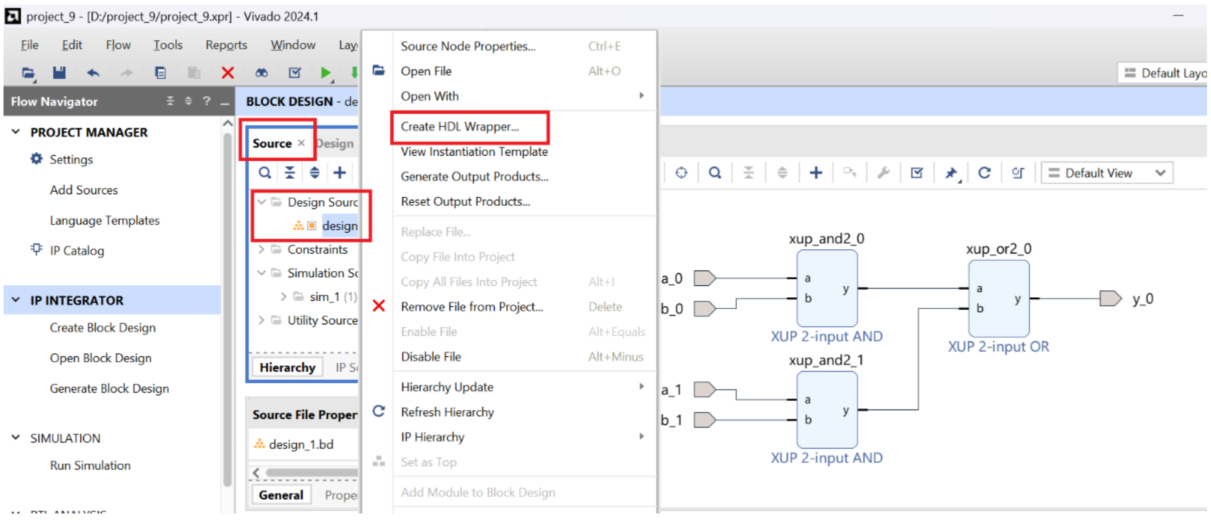
```
INFO: [Common 17-17] undo 'de
```

```
Type a Tcl command here
```

The completed design should look like this:



Step 7: Right-click on the block design(.bd) file under the "Source" tab, then click "Create HDL wrapper" and then click OK.



This creates the equivalent HDL code to the created design, and the associated files will be listed under the "Source" tab.

Step 8: Run the simulation, synthesis, and implementation following the steps detailed in Exp.7 and Exp.8.

Step 9: Generate the bitstream and download it into Basys3.

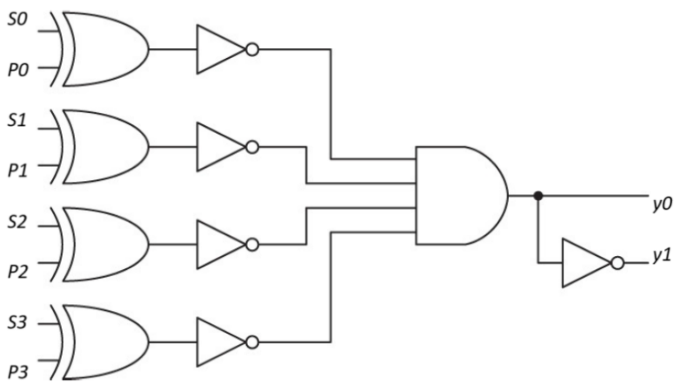
EXAMPLE 9.1:

Design a digital safe system using combinational circuits on Basys 3. Assume that the system has a four-bit predefined password. If the user input to the system matches the predefined password, turn an LED ON. Otherwise, turn another LED ON.

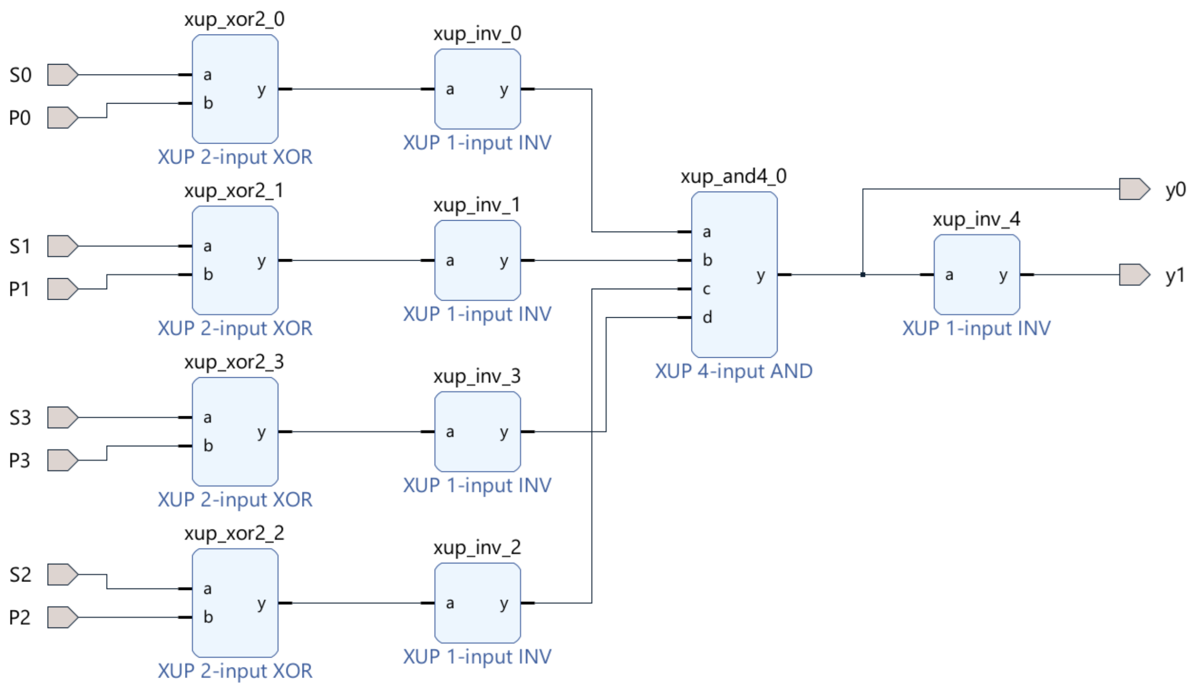
Solution:

The digital safe can be implemented using an XOR gate followed by a NOT gate for each bit to be tested. Therefore, if the input bit matches the corresponding password bit, then the XOR gate followed by NOT will give logic level 1. If all input bits match corresponding predefined password bits this way, the output will have logic level 1. Defining a second output by simply inverting the actual output can indicate the incorrect password.

Let's define S0, S1, S2, and S3 as the user inputs and P0, P1, P2, and P3 are the predefined password bits. y0 as the correct password indicator and y1 as the incorrect password indicator.



Block Design:



Simulation Results:

Input: Set P0=1, P1=0, P2=1, P3=1

Feed S0=1, S1= 100ns clock, S2=1, S3=1

Output: y0 and y1 varies based on correct and incorrect password

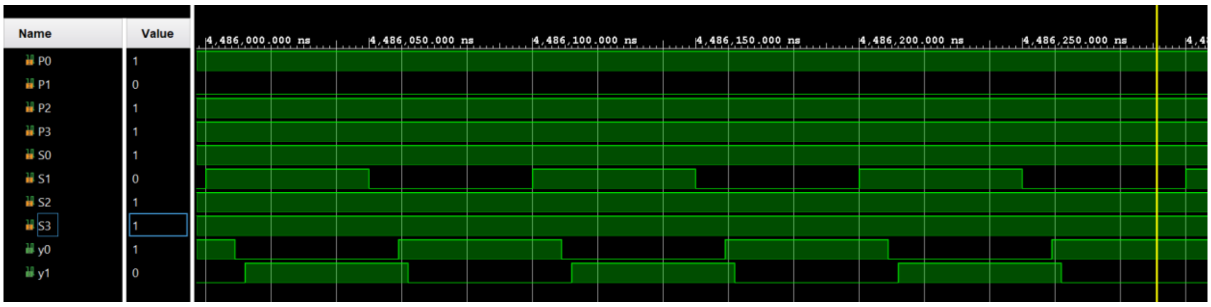


Fig. 9.1: Simulation results of example 9.1

IO pin assignment:

P0:R2	P2:U1	S0:W17	S2:V16	y0:V13
P1:T1	P3:W2	S1:W16	S3:V17	y1:V14

Hardware results:

Set password:

SW14	**SW13**	**SW12**	**SW11**
ON	OFF	ON	ON

Enter password:

SW3	**SW2**	**SW1**	**SW0**	**LD8**	**LD7**
ON	OFF	ON	ON	ON	OFF
For other combinations				OFF	ON

EXAMPLE 9.2:

Design a car park-occupied slot counting system, which counts how many slots are occupied at any given time and displays its count on a seven-segment display of Basys 3.

Solution:

Assume there is a car park with three slots, and we would like to know how many slots are occupied at a given time. Within the design, occupied slot locations are not necessary. Assume that we placed a sensor over each slot, which provides output logic level 1 when the slot is occupied. If the slot is empty, the sensor provides output logic level 0.

Let label the output of sensors as binary variables S_0, S_1, and S_2. The designed digital circuit will provide the output as a two-bit binary number C_1 (MSB) and C_0 (LSB). Therefore, we should cover all input combinations in terms of a truth table as:

Inputs			Outputs	
S0	**S1**	**S2**	**C1**	**C0**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0

1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The Boolean equation for the above table is:

$$C_0 = \overline{S_o}.\overline{S_1}.S_2 + \overline{S_o}.S_1.\overline{S_2} + S_0.\overline{S_1}.\overline{S_2} + S_0.S_1.S_2$$

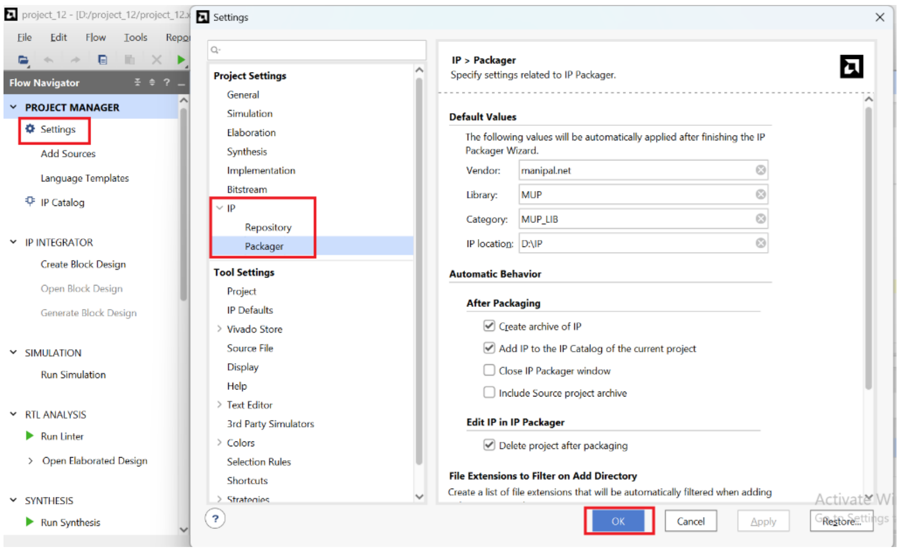
$$C_1 = \overline{S_o}.S_1.S_2 + S_0.\overline{S_1}.S_2 + S_0.S_1.\overline{S_2} + S_0.S_1.S_2$$

Now, the 2-bit binary output must be displayed on the seven-segment display. However, the Vivado in-built repositories don't have the IP for the segment-segment display HDL description. Therefore, a custom IP needs to be created using the steps below(Ref):

Step 1: Create a Vivado project following the steps described in Exp.1.

Step 2: Create an empty Verilog source file and name it "Seven_Seg.v". Copy the seven-segment HDL code from Example.7.1. Although it is optional, recommended to simulate it to verify the logical correctness of your HDL description.

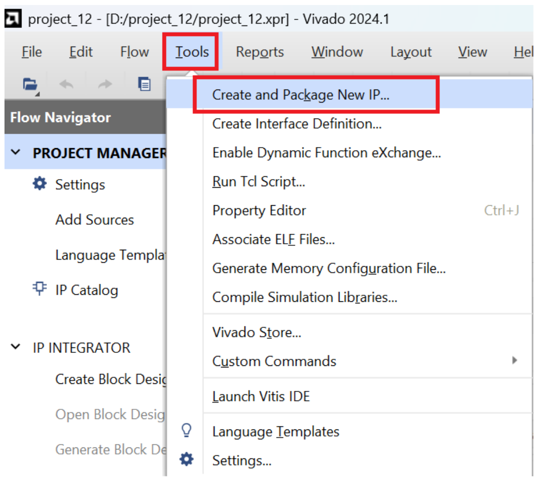
Step 3: In the flow navigator, open "settings" and then click "IP" → "packager," set the fields as below, and click "OK."


IP > Packager
Specify settings related to IP Packager.

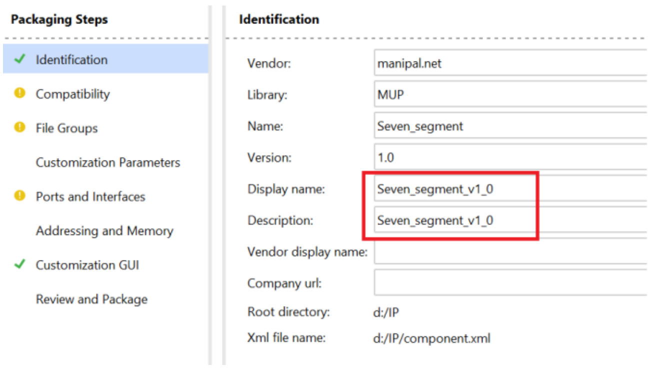
Default Values
The following values will be automatically applied after finishing the IP Packager Wizard.
Vendor: manipal.net
Library: MUP
Category: MUP_LIB
IP location: D:\IP

Automatic Behavior
After Packaging
☑ Create archive of IP
☑ Add IP to the IP Catalog of the current project
☐ Close IP Packager window
☐ Include Source project archive
Edit IP in IP Packager
☑ Delete project after packaging
File Extensions to Filter on Add Directory
Create a list of file extensions that will be automatically filtered when adding

Step 4: Select "tools" and click "Create and Package New IP", click "Next", select "Package your current project", click "Next", give the IP location, click "Next", and "Finish".



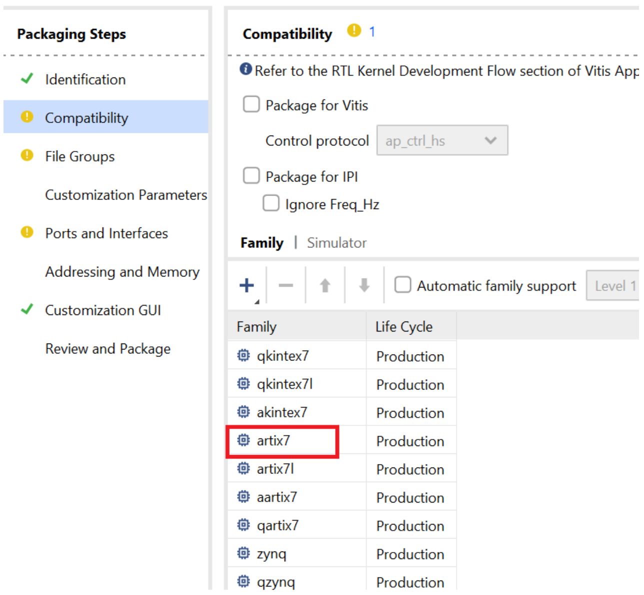
Step 5: Under "packaging steps", select "Identification" and give a meaningful name and description to your IP:



Identification

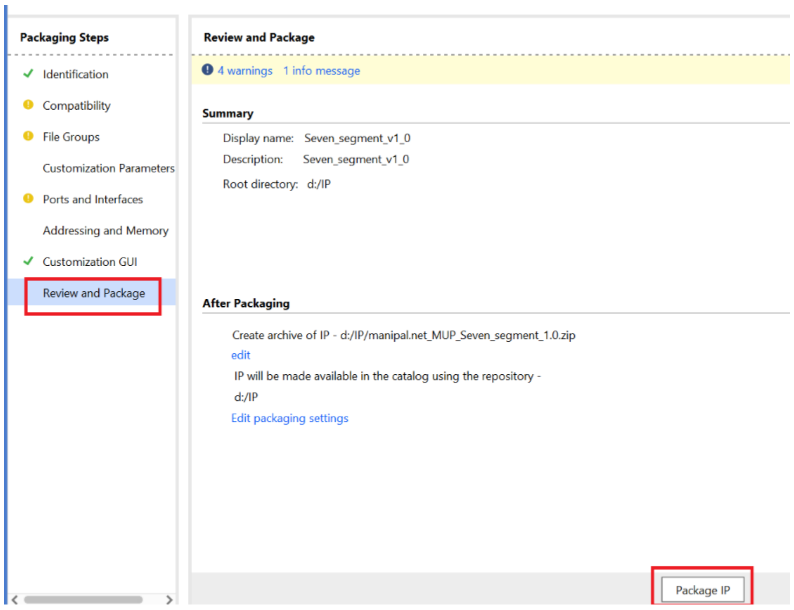
Vendor:	manipal.net
Library:	MUP
Name:	Seven_segment
Version:	1.0
Display name:	Seven_segment_v1_0
Description:	Seven_segment_v1_0
Vendor display name:	
Company url:	
Root directory:	d:/IP
Xml file name:	d:/IP/component.xml

In "compatibility," check if the Artix-7 family is added:



Family	Life Cycle
qkintex7	Production
qkintex7l	Production
akintex7	Production
artix7	Production
artix7l	Production
aartix7	Production
qartix7	Production
zynq	Production
qzynq	Production

Go to "Review and Package" and click "Package IP":



Review and Package

4 warnings 1 info message

Summary

Display name: Seven_segment_v1_0
Description: Seven_segment_v1_0
Root directory: d:/IP

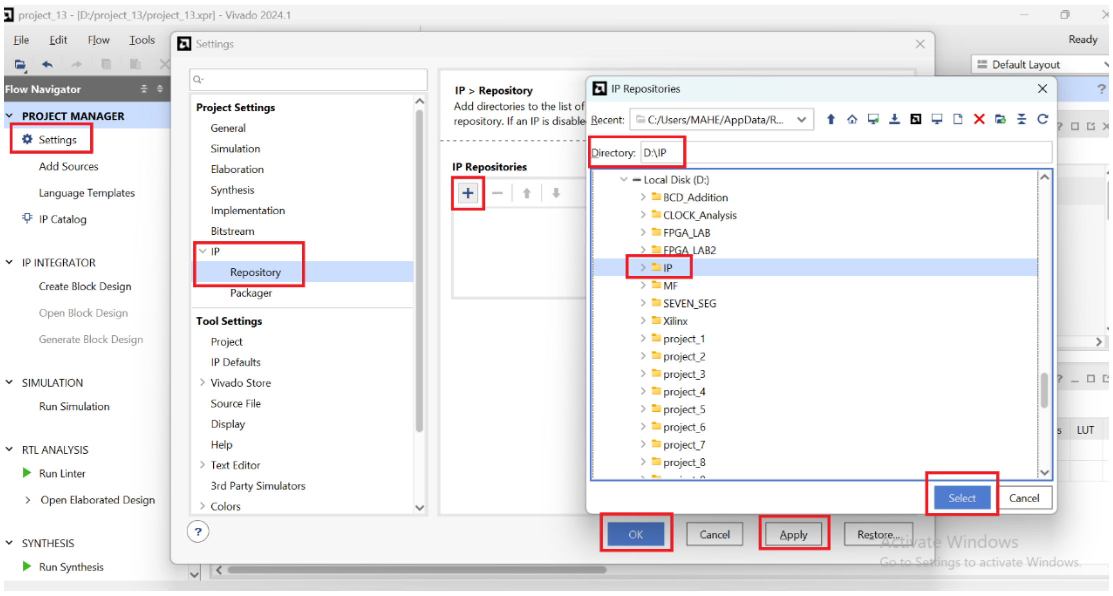
After Packaging

Create archive of IP - d:/IP/manipal.net_MUP_Seven_segment_1.0.zip
edit
IP will be made available in the catalog using the repository - d:/IP
Edit packaging settings

Your IP is created with the name "Seven_segment_v1_0" and saved at the given IP location.

Block design:

Create a new Vivado project, go to "settings", click "IP" and then click "Repository", click "+" and then browse to your "IP location" and "select" it. Click "OK", "Apply" and "OK".



Add directories to the list of repository. If an IP is disable

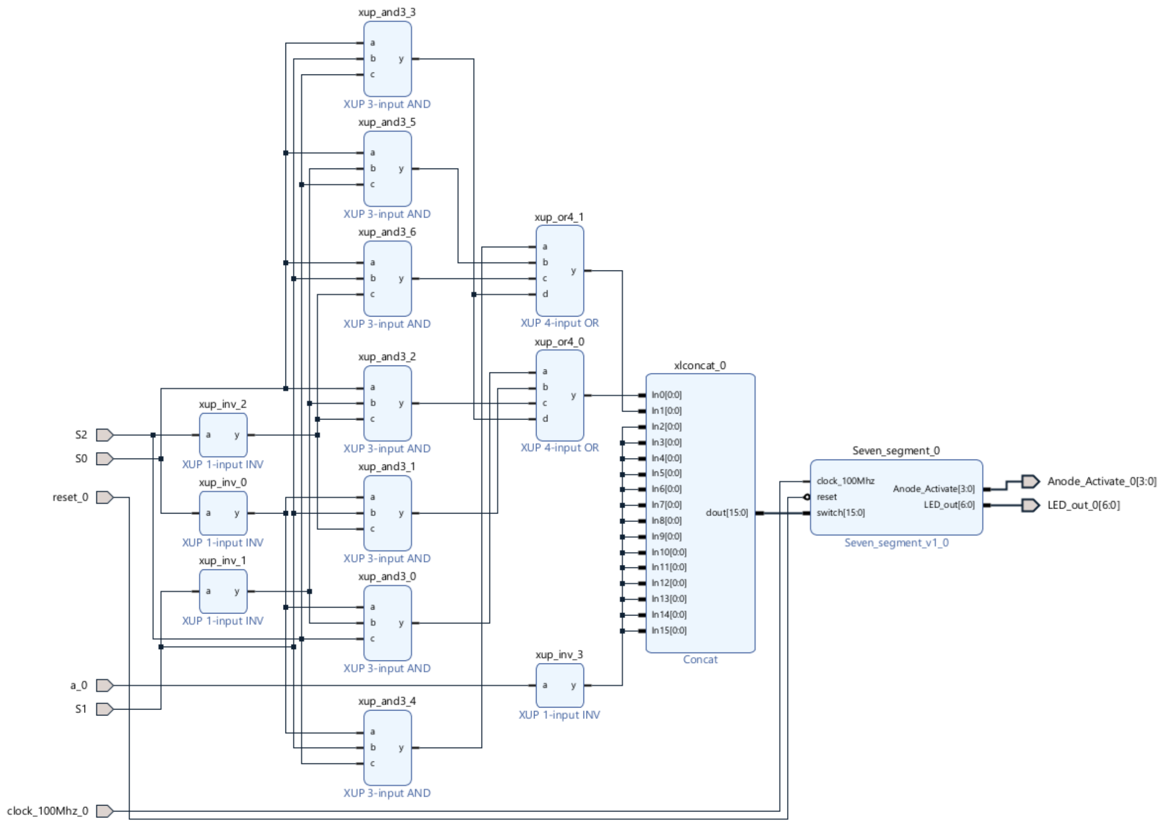
Directory: D:\IP

Similarly, add the "XUP_LIB" repository.

In the flow navigator, click "Create Block design" and select the "Seven_segment_v1_0" component in the search space of the design canvas. Add it to your design along with other necessary components listed in below table for the implementation of our Boolean equation:

Component	Library	Operation	Description	Quantity
xup_inv_2	XUP_LIB	NOT	For compliment operation of Boolean Equation	4
xup_and_3	XUP_LIB	AND	For product operation of Boolean Equation	7
xup_or_3	XUP_LIB	OR	For sum operation of Boolean Equation	2
xlconcat_0	in-built	Concatenate	To mask the 14 higher input bits to seven segment display IP	1
Seven_seg_v1_0	Custom	Binary to hex conversion	To receive 16-bit binary input and display the equivalent 4-digit Hexadecimal on SSD	1

Interconnect the components as shown in the schematic below, create the HDL wrapper, synthesize and implement it, and generate the bitstream to download it onto Basys 3.



IO pin assignment:

Inputs		Anode_Activate [3:0]		LED_out [6:0]			
S0	V17	Bit 3: Display 1	W4	Bit 6: Seg a	W7	Bit 2: Seg e	U5
S1	V16	Bit 2: Display 2	V4	Bit 5: Seg b	W6	Bit 1: Seg f	V5

S2	W16	Bit 1: Display 3	U4	Bit 4: Seg c	U8	Bit 0: Seg g	U7
a_0	R2	Bit 0: Display 4	U2	Bit 3: Seg d	V8		
Reset_0: T17				**Clock:** W5			

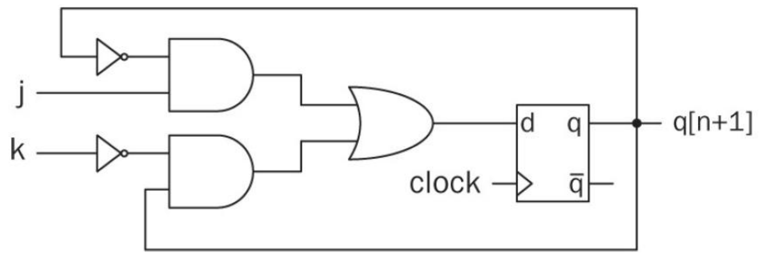
Hardware results:

SW15	SW2	SW1	SW0	Display on SSD
ON	OFF	OFF	OFF	**0**
ON	OFF	OFF	ON	**1**
ON	OFF	ON	OFF	**1**
ON	OFF	ON	ON	**2**
ON	ON	OFF	OFF	**1**
ON	ON	OFF	ON	**2**
ON	ON	ON	OFF	**2**
ON	ON	ON	ON	**3**

EXAMPLE 8.3: Design JK flipflop using D flipflop and implement it on Basys3 using Vivado IPI

Solution:

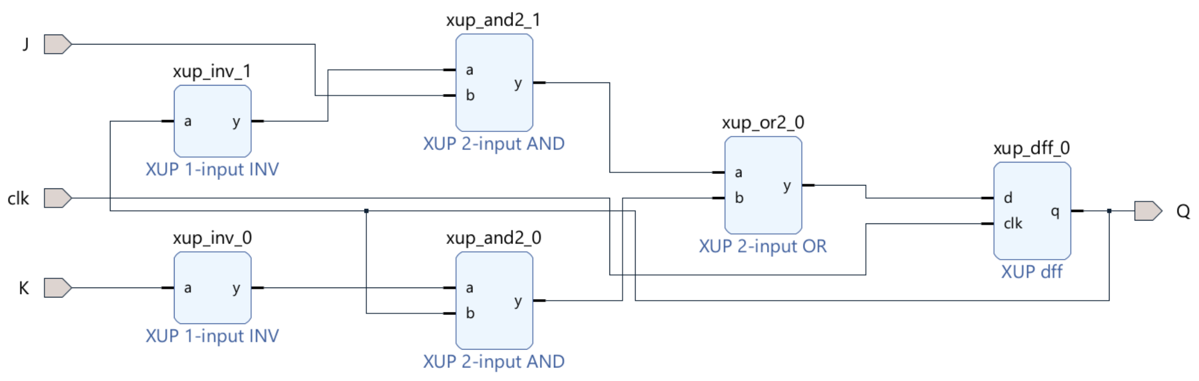
Logic diagram:



Truth table:

Inputs		Output
J	K	Q[n+1]
0	0	Q[n]
0	1	0
1	0	1
1	1	$\bar{Q}[n]$

Block design:



IO Pin assignment:

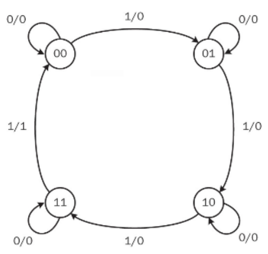
Input	**Pin No**	**Output**	**Pin No**
J	V16	Q	L1
K	V17		
clk	W5		

Hardware Results:

SW1	SW2	LD15
OFF	ON	OFF
OFF	OFF	OFF
ON	OFF	ON
OFF	OFF	ON

EXERCISE PROBLEMS:

1. Construct a block design for 2 to 4 decoder using Vivado IPI and realize it on Basys3.
2. Construct a block design for 4 to 2 priority encoder using Vivado IPI and realize it on Basys3.
3. Construct a block design for 3-bit parity generator and 3-bit parity checker using Vivado IPI and realize it on Basys3.
4. Construct a block design for 2 asynchronous counter using Vivado IPI and realize it on Basys3.
5. Construct a block design for the below state diagram using Vivado IPI and realize it on Basys3.


0/0
1/0
0/0
00
01
1/1
1/0
11
10
0/0
1/0
0/0