ELL201 – Digital Electronics

Project:

A comprehensive design of the Electronic Voting Machine & Real-Time Data Logger using Universal Shift Registers

IIT Delhi

Indian Institute of Technology Delhi

By:

Pushkar Shah -2022PH11838

Group Members :

Pritish Mahajan – 2022PH11835

Syed Nabhan – 2022PH11842

Sujit Kumar – 2022PH11244

Electronic Voting Machine: A Comprehensive Design

In this report, we present a detailed design for an electronic voting machine that allows people to vote between two teams, A and B. The system features a secure and efficient voting process, where each vote is stored and tallied using a robust 8-bit register system. The design includes a comprehensive implementation methodology, Verilog code, and test bench waveforms to validate the functionality of the voting machine.

Voting Process Overview

The electronic voting machine is designed to allow people to cast their votes between two teams, A and B. The voting process is simple and straightforward. Each voter must physically switch (or "store") a button from the off position to the on position, which triggers the voting mechanism. This process is supervised to ensure the integrity of the vote.

When a voter stores the button, the current vote count is retrieved from an 8-bit register, incremented by 1 using an 8-bit CLA, and then stored back into the register. This process ensures that each vote is accurately recorded and the total vote count is continuously updated.

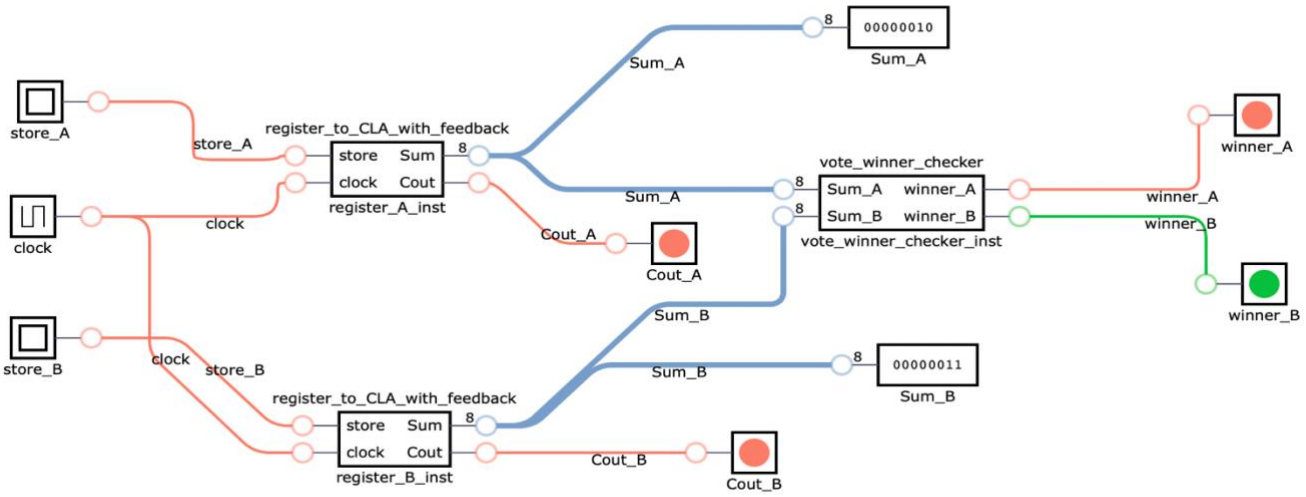
Verilog Code:

```
module register_1bit(input data_in, input store, input clock, output reg data_out);

    // Initialize data_out to zero at time t=0
    initial begin
        data_out = 1'b0;
    end

    always @(posedge clock) begin
        if (store == 1) begin
            data_out <= data_in;
        end
    end

endmodule

module register_8bit(input [7:0] data_in, input store, input clock, output reg [7:0] data_out);
    reg [7:0] reg_data;

    // Individual 1-bit registers
    register_1bit bit0(.data_in(data_in[0]), .store(store), .clock(clock), .data_out(reg_data[0]));
    register_1bit bit1(.data_in(data_in[1]), .store(store), .clock(clock), .data_out(reg_data[1]));
    register_1bit bit2(.data_in(data_in[2]), .store(store), .clock(clock), .data_out(reg_data[2]));
    register_1bit bit3(.data_in(data_in[3]), .store(store), .clock(clock), .data_out(reg_data[3]));
    register_1bit bit4(.data_in(data_in[4]), .store(store), .clock(clock), .data_out(reg_data[4]));
    register_1bit bit5(.data_in(data_in[5]), .store(store), .clock(clock), .data_out(reg_data[5]));
    register_1bit bit6(.data_in(data_in[6]), .store(store), .clock(clock), .data_out(reg_data[6]));
    register_1bit bit7(.data_in(data_in[7]), .store(store), .clock(clock), .data_out(reg_data[7]));

    assign data_out = reg_data;
endmodule

module carry_lookahead_adder(
    input [7:0] A,
    input [7:0] B,
    output [7:0] Sum,
```

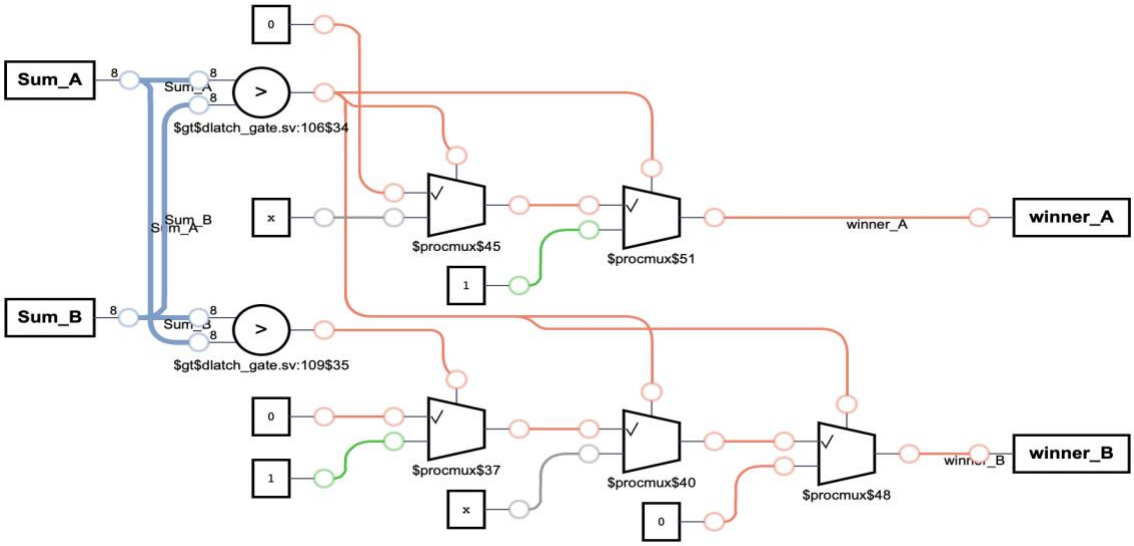
```
    output Cout
);

    wire [7:0] P, G;
    wire [7:0] generate_carry;
    wire [7:0] propagate_carry;

    assign P = A ^ B;
    assign G = A & B;

    assign generate_carry[0] = G[0];
    assign propagate_carry[0] = P[0];

    assign generate_carry[1] = G[1] | (P[1] & generate_carry[0]);
    assign propagate_carry[1] = P[1] | (P[0] & propagate_carry[0]);

    assign generate_carry[2] = G[2] | (P[2] & generate_carry[1]);
    assign propagate_carry[2] = P[2] | (P[1] & propagate_carry[1]);

    assign generate_carry[3] = G[3] | (P[3] & generate_carry[2]);
    assign propagate_carry[3] = P[3] | (P[2] & propagate_carry[2]);

    assign generate_carry[4] = G[4] | (P[4] & generate_carry[3]);
    assign propagate_carry[4] = P[4] | (P[3] & propagate_carry[3]);

    assign generate_carry[6] = G[6] | (P[6] & generate_carry[5]);
    assign propagate_carry[6] = P[6] | (P[5] & propagate_carry[5]);

    assign generate_carry[7] = G[7] | (P[7] & generate_carry[6]);
    assign propagate_carry[7] = P[7] | (P[6] & propagate_carry[6]);

    assign Sum = A + B;
    assign Cout = generate_carry[7] | (propagate_carry[7] & 0);

endmodule

module register_to_CLA_with_feedback(

    input store,
    input clock,
    output reg [7:0] Sum,
    output reg Cout
);

    reg [7:0] registered_data;
    wire [7:0] CLA_output_Sum;
    wire CLA_output_Cout;



    // Instantiate the carry-lookahead adder (CLA) with registered data and constant '00000001' as inputs
    carry_lookahead_adder CLA_inst(.A(registered_data), .B(8'b00000001), .Sum(CLA_output_Sum), .Cout(CLA_output_Cout));

   // Instantiate the 8-bit register
    register_8bit register_inst(.data_in(CLA_output_Sum), .store(store), .clock(clock), .data_out(registered_data));

    // Output CLA results
    assign Sum = CLA_output_Sum;
    assign Cout = CLA_output_Cout;


endmodule

module vote_winner_checker(
```

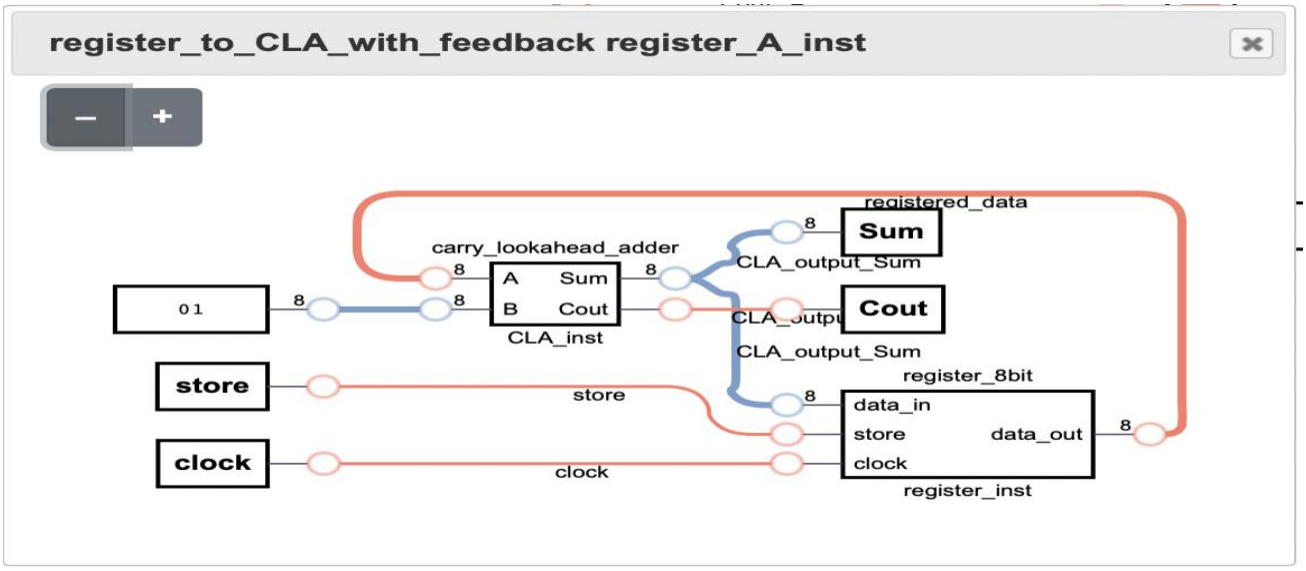
```
    input wire [7:0] Sum_A,
    input wire [7:0] Sum_B,
    output reg winner_A,
    output reg winner_B
);

    always @* begin
        if (Sum_A > Sum_B) begin
            winner_A = 1'b1; // A wins
            winner_B = 1'b0; // B loses
        end else if (Sum_B > Sum_A) begin
            winner_A = 1'b0; // A loses
            winner_B = 1'b1; // B wins
        end else begin
            winner_A = 1'b0; // Tie for A
            winner_B = 1'b0; // Tie for B
        end
    end

endmodule


module vote(
    input wire store_A,
    input wire store_B,
    input wire clock,
    output reg [7:0] Sum_A,
    output reg Cout_A,
    output reg [7:0] Sum_B,
    output reg Cout_B,
    output reg winner_A,  // Output for winner of vote A
    output reg winner_B  // Output for winner of vote B
);

    // Instantiate the register_to_CLA_with_feedback module for voting A
    register_to_CLA_with_feedback register_A_inst(
        .store(store_A),
        .clock(clock),
        .Sum(Sum_A),
        .Cout(Cout_A)
    );

    // Instantiate the register_to_CLA_with_feedback module for voting B
    register_to_CLA_with_feedback register_B_inst(
        .store(store_B),
        .clock(clock),
        .Sum(Sum_B),
        .Cout(Cout_B)
    );

    // Instantiate the vote_winner_checker module to determine the winner
    vote_winner_checker vote_winner_checker_inst(
        .Sum_A(Sum_A),
        .Sum_B(Sum_B),
        .winner_A(winner_A),
        .winner_B(winner_B)
    );

endmodule
```

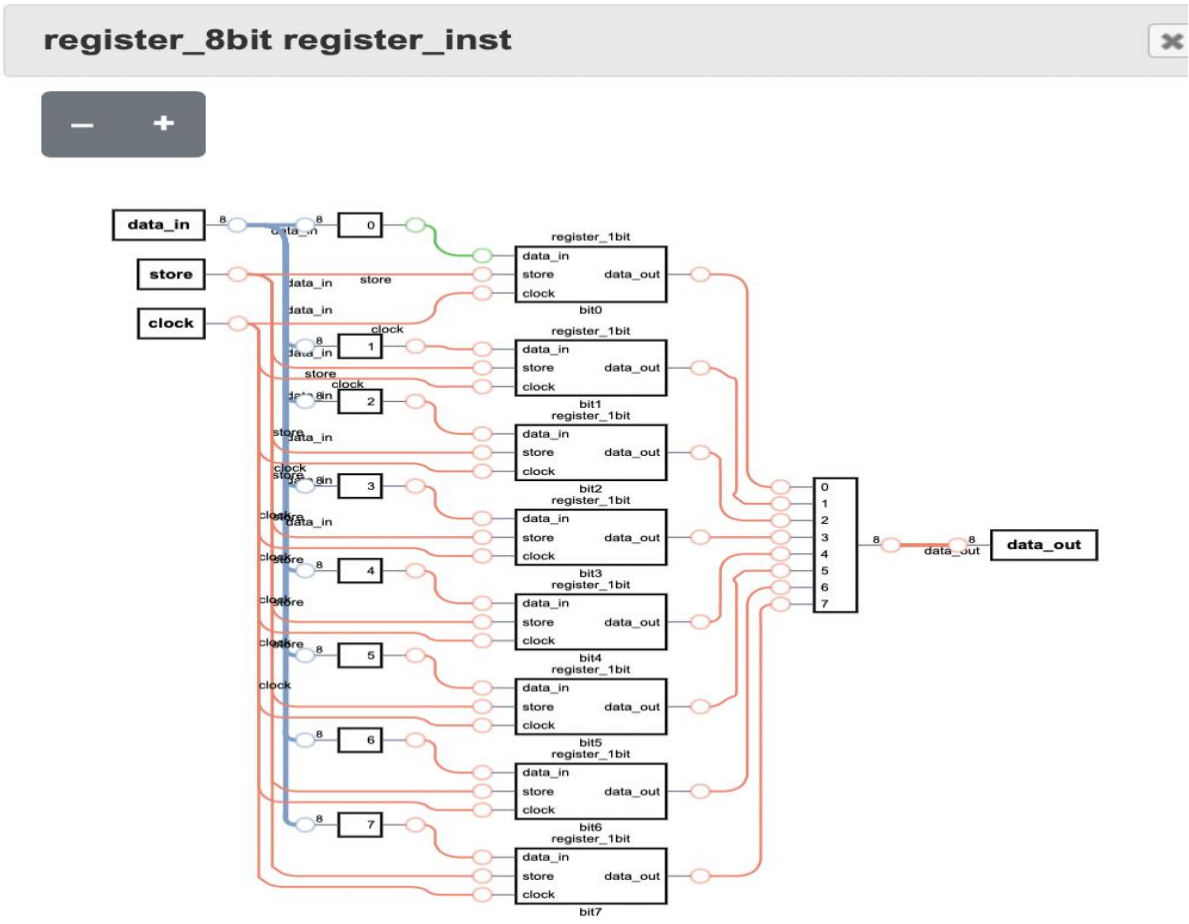
Diagrams:



Vote_winner_checker:




register_to_CLA_with_feedback register_A_inst
registered_data
Sum
carry_lookahead_adder
A Sum
B Cout
CLA_inst
CLA_output_Sum
Cout
CLA_output_Sum
register_8bit
data_in
store
clock
data_out
register_inst
01
store
clock


register_8bit register_inst
data_in
store
clock
register_1bit
data_in
store
clock
data_out
bit0
bit1
bit2
bit3
bit4
bit5
bit6
bit7
data_out

Truth Table:

Register:

Clock (↑)	Store	Data In	Data Out (at next clock edge)
0	0	X	Previous Data Out
1	0	0	Previous Data Out
1	0	1	Previous Data Out
1	1	X	Data In

1 bit vote_winner_checker (this can be extrapolated to 8 bit vote_winner_checker):

A	B	Winner_A	Winner_B
0	0	0	0
0	1	0	1
1	0	1	0
1	1	0	0

1-bit register with feedback to a carry lookahead adder (this can be extrapolated to 8 bit vote_winner_checker):

Clock (↑)	Store	Data In	Data Out	Carry In	Sum (at next clock edge)	Carry Out
0	0	X	Previous Data Out	X	Previous Sum	Previous Carry Out
1	0	0	Previous Data Out	X	Previous Sum	Previous Carry Out
1	0	1	Previous Data Out	X	Previous Sum	Previous Carry Out
1	1	X	Data In	X	Data In + Previous Data Out	Previous Carry Out
1	1	X	Data In	0	Data In + Previous Data Out	0
1	1	X	Data In	1	Data In + Previous Data Out + 1	1

Waveform Analysis:

The test bench generates detailed waveforms that allow the design to be analysed and debugged. These waveforms show the evolution of the input signals, the internal register values, and the output vote counts, ensuring the voting machine operates as

expected under various conditions. But due to technical difficulties and time constraints, we were unable to generate detailed waveforms using the test bench.
Initially we were getting errors on Continuous counting but after some improvements fixed the issue below is the complete working code, for the voting machine with test bench output and link for the repository
I have made a EDA Playground Repository For the updated Code..
The Link is -> https://www.edaplayground.com/x/PxpX

Updated Code:

```
// Description: Design of a 3 candidate voting machine using state machine.

module voting_machine #(
parameter idle =  2'b00,                                    // states and their corresponding numbers
parameter vote =  2'b01,
parameter hold =  2'b10,
parameter finish = 2'b11
)(
   input clk,
   input rst,                                                       // input High to reset counting  (active high)
   input i_candidate_1,                                 // input to vote for candidate 1
   input i_candidate_2,                                 // input to vote for candidate 2
   input i_candidate_3,                                 // input to vote for candidate 3
   input i_voting_over,                                 // input high to get total votes after voting is over

   output reg [31:0] o_count1,                          // output for total number of votes of candidate 1
   output reg [31:0] o_count2,                          // output for total number of votes of candidate 2
   output reg [31:0] o_count3                           // output for total number of votes of candidate 3

);

reg [31:0] r_cand1_prev;                                    // store previous value of input for candidate 1
reg [31:0] r_cand2_prev;                                    // store previous value of input for candidate 2
reg [31:0] r_cand3_prev;                                    // store previous value of input for candidate 3

reg [31:0] r_counter_1;                                         // counting register for candidate 1
reg [31:0] r_counter_2;                                         // counting register for candidate 2
reg [31:0] r_counter_3;                                         // counting register for candidate 3

reg [1:0] r_present_state, r_next_state;            // present state and next state registers
//reg [1:0] r_state_no;                                             // store state number
reg [3:0] r_hold_count;                 //counter for hold state

////////////// always block that assigns next state & internal reg operations ///////////////////////////////////
always @(posedge clk or negedge rst)
        begin
                case (r_present_state)

                        idle: if (!rst)
                // idle state operations
                                        begin
                                                r_next_state <= vote;
                                // assign next state vote when reset low
                                                //r_state_no <= 2'b01;
                                        end

                                else
                                        begin
                                        //r_present_state = idle;
                        // present state at the beginning
                                                r_counter_1 <= 32'b0;
                                        // clear counting registers
                                                r_counter_2 <= 32'b0;
                                                r_counter_3 <= 32'b0;
```

```
                                        r_hold_count <= 4'b0000;

                                        r_next_state <= idle;
                                // assign next state as idle till reset not low
                                        //r_state_no <= 2'b0;
                                end


                        vote: if (i_voting_over == 1'b1)
        // check if voting is over
                                begin
                                        r_next_state <= finish;
                        // if over is high go to finish state
                                        //r_state_no <= 2'b11;
                                end

// if over is low contiue counting
                                else if (i_candidate_1 == 1'b0 && r_cand1_prev == 1'b1)
 // check falling edge of input candidate1 so only single input is regeistered
                                begin
                                        r_counter_1 <= r_counter_1 + 1'b1;
                                // increment counter for candidate 1
                                        //r_counter_2 <= r_counter_2;
                                // keep previous value of counter
                                        //r_counter_3 <= r_counter_3;
                                // keep previous value of counter

                                        r_next_state <= hold;
                                // got to hold state
                                        //r_state_no <= 2'b10;
                                end

                                else if (i_candidate_2 == 1'b0 && r_cand2_prev == 1'b1)
// check falling edge of input candidate2 so only single input is regeistered
                                begin
                                        //r_counter_1 <= r_counter_1;
                                // keep previous value of counter
                                        r_counter_2 <= r_counter_2 + 1'b1;
                                // increment counter for candidate 2
                                        //r_counter_3 <= r_counter_3;
                                // keep previous value of counter

                                        r_next_state <= hold;
                                // got to hold state
                                        //r_state_no <= 2'b10;
                                end

                                else if (i_candidate_3 == 1'b0 && r_cand3_prev == 1'b1)
// check falling edge of input candidate3 so only single input is regeistered
                                begin
                                        //r_counter_1 <= r_counter_1;
                                // keep previous value of counter
                                        //r_counter_2 <= r_counter_2;
                                // keep previous value of counter
                                        r_counter_3 <= r_counter_3 + 1'b1;
                                // increment counter for candidate 3

                                        r_next_state <= hold;
                                // got to hold state
                                        //r_state_no <= 2'b10;
                                end
                                else
                                        // none of the input present or more than 1 input present at same time
                                        begin
```

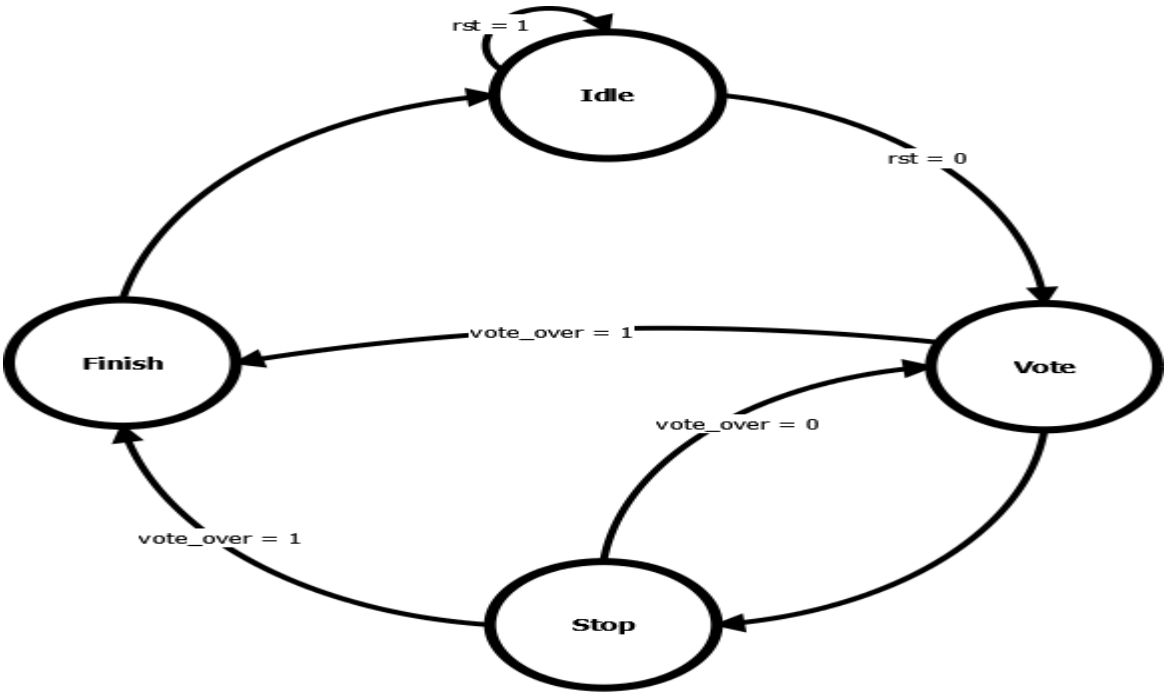
```
                                        r_counter_1 <= r_counter_1;
                                // keep previous value of counter
                                        r_counter_2 <= r_counter_2;

                                        r_counter_3 <= r_counter_3;

                                        r_next_state <= vote;
                                        //r_state_no <= 2'b01;
                                end

                hold: if (i_voting_over == 1'b1)
        // check if over input is high
                                begin
                                        r_next_state <= finish;
                        // go to finish state
                                        //r_state_no <= 2'b11;
                                end

                                else
                                        begin
                                        if (r_hold_count != 4'b1111) begin
                                                r_hold_count = r_hold_count + 1'b1;
                                        end
                                        else begin
                                           r_next_state <= vote;
                        // if over is low go to vote state
                                        end
                                        //r_state_no <= 2'b01;
                                end
                finish: if (i_voting_over == 1'b0)

                                begin
                                        r_next_state <= idle;
                        // if over is low go to idle state
                                        //r_state_no <= 2'b0;
                                end

                                else
                                        begin
                                        r_next_state <= finish;
                        // remain in finish state if over is high
                                        //r_state_no <= 2'b11;
                                end
                default:
                        begin
                                r_counter_1 <= 32'b0;
                        // default values for resgisters
                                r_counter_2 <= 32'b0;
                                r_counter_3 <= 32'b0;
                                r_hold_count <= 4'b0000;

                                r_next_state <= idle;
                // by default go to idle state at the begining
                                //r_state_no <= 2'b0;
                        end
            endcase
        end

////////////// always block that performs assignment of regsiters and output on clock signal
//////////////////////////////
always @(posedge clk or negedge rst)        // work on positive edge of clock
        begin
    if (rst == 1'b1)
                begin
```

```
                        r_present_state <= idle;
                // remain in idle state when reset is high

                        o_count1 <= 32'b0;
                        // reset final output count
                        o_count2 <= 32'b0;
                        o_count3 <= 32'b0;
                        r_hold_count <= 4'b0000;
                end

        else if (rst == 1'b0 && i_voting_over == 1'b1)
                        // if voting process is i.e.over is high
                begin
                        o_count1 <= r_counter_1;
                        // provide value of counting registers at output
                        o_count2 <= r_counter_2;
                        o_count3 <= r_counter_3;
                end

        else
                begin
                        r_present_state <= r_next_state;
        // if reset is low keep assigning next state to present state
                        r_cand1_prev <= i_candidate_1;
        // keep assigning input of candidate 1 to internal register
                        r_cand2_prev <= i_candidate_2;
                        r_cand3_prev <= i_candidate_3;
                end
    end
endmodule
```

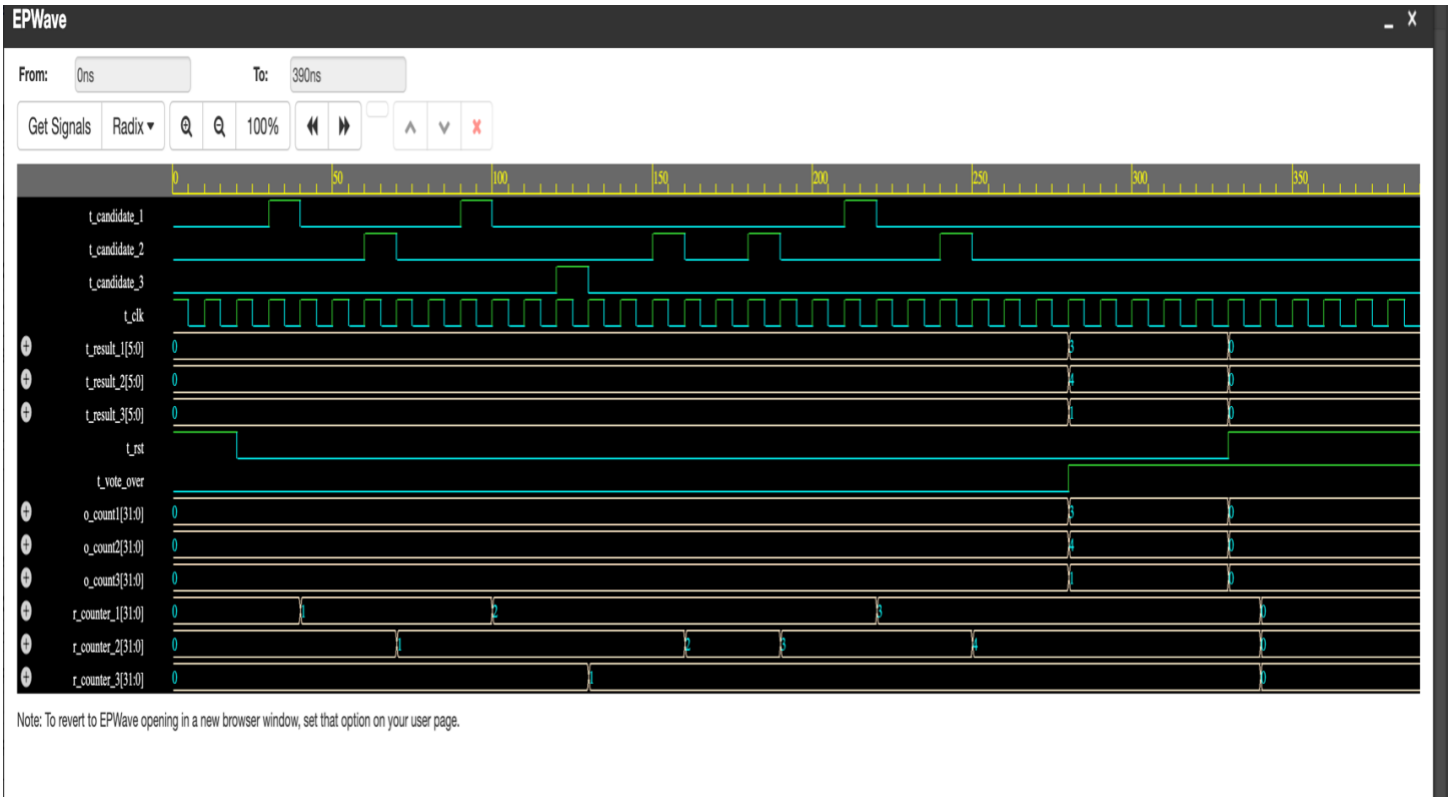
The Wave Forms And Testbench Pictures and state Diagram :

1) The voting machine is designed for 3 candidates with output size of 32 bits. Used the state machine approach comprising of 4 states idle, vote, stop and finish.
2) Runs on clock frequency 10MHz. There is another combinational always block that runs on clock signal whereas the always block of state machine works on state and inputs.

State Diagram. ->



The Wave Forms of testbench..



From the wave forms:

After the voting is over the Candidates have Votes :

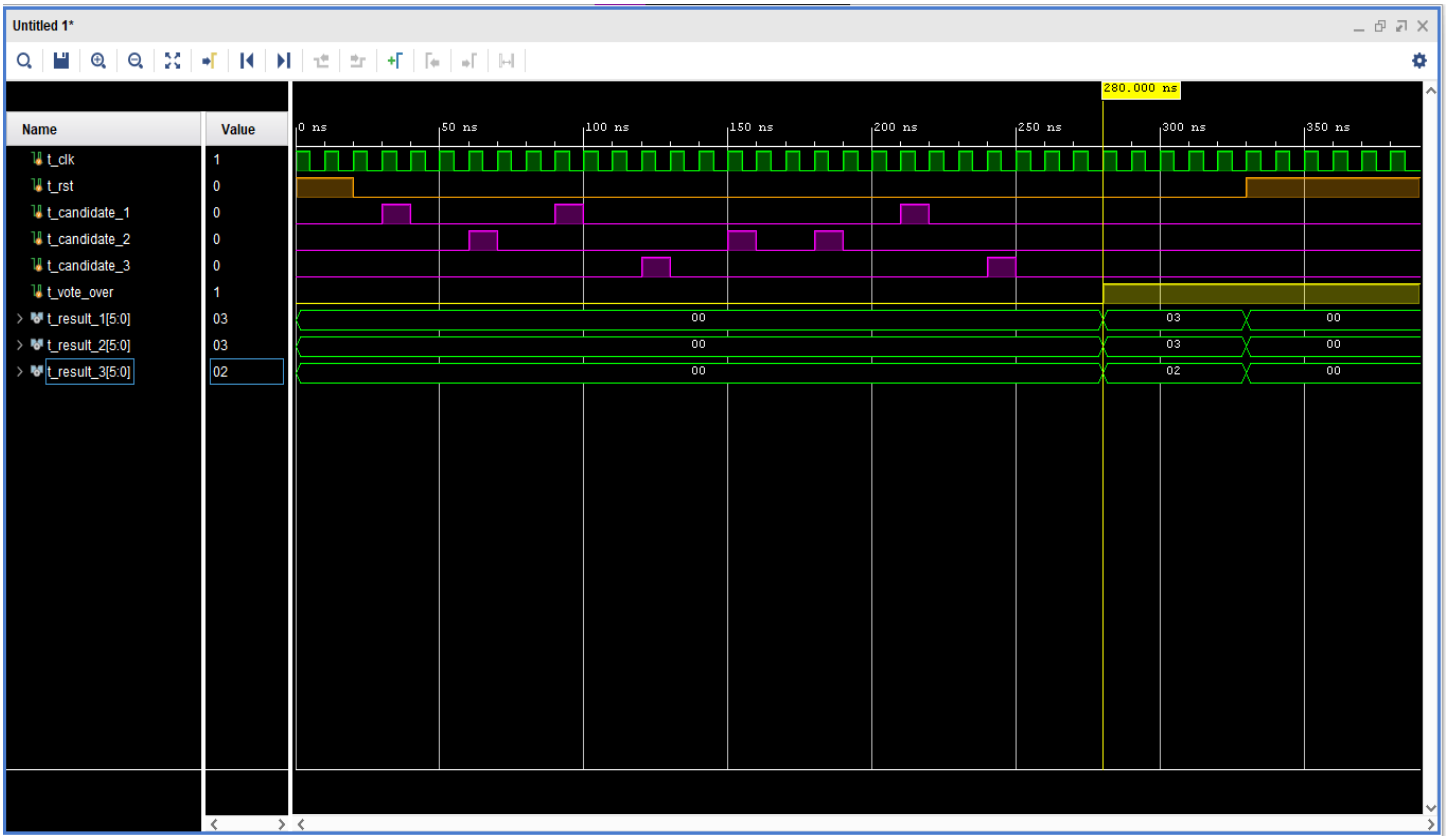
1. Candidate 1 : 3 votes.
2. Candidate 2 : 4 votes.
3. Candidate 3 : 1 votes.

The machine can store votes upto a 32 bits Integer .
Voting counters : r_counter which can been seen changing after every vote is casted.
Result Register : t_result represent final votes for each candidate.

But due to time constrain in the lab we weren't able to show this on CPLD board..

Another Test Case :

Candidate 1 – 3 votes
Candidate 2 – 3 votes
Candidate 3 – 2 votes



Next We would Discuss about : Real-Time Data Logger using Universal Shift Registers.

Real-Time Data Logger using Universal Shift Registers.

This project analyses and simulates the operations of a 4-bit Universal Shift Register. The Register can take data and control inputs from the user and execute data operations according to the mode of operation specified.

USR: Introduction

A Universal Shift Register is a register with both right shift and left shift with parallel load capabilities. Universal Shift Registers are used as memory elements in computers. A Unidirectional Shift Register shifts in only one direction whereas a Bidirectional Shift Register is capable of shifting in both the directions. The design of Universal Shift Register is a combination of Bidirectional Shift Register and a Unidirectional Shift Register with provision for parallel. A 4-bit Universal Shift Register consists of 4 flip-flops and 4 4×1 multiplexers. All the 4 multiplexers share the same select lines (S1 and S0) which select the mode of operation for the shift register. The select line inputs choose the suitable input for the flip-flops.

Circuit Design

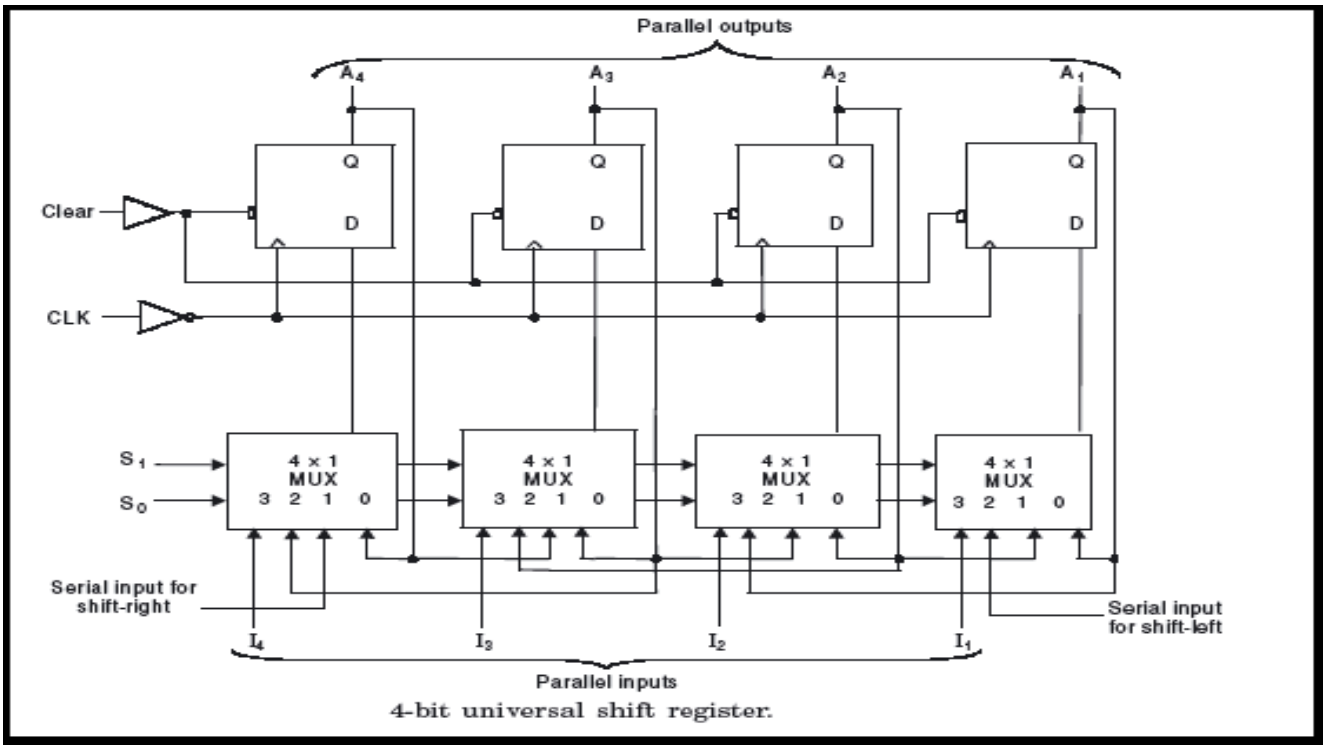
This model has the following connections:

1. The first input is connected to the output pin of the corresponding flip-flop.

2. The second input is connected to the output of the very-previous flip flop which initiates the right shift.
3. The third input is connected to the output of the very-next flip-flop which facilitates the left shift.
4. The fourth input is connected to the individual bits of the input data which helps in parallel loading.

The working of the Universal Shift Register depends on the inputs given to the select lines.

Circuit Diagram :



4-bit universal shift register.

Universal Shift Register Working

1. From the above figure, selected pins the mode of operation of the universal shift register. Serial input shifts the data towards the right and left and stores the data within the register.
2. Clear pin and CLK pin are connected to the flip-flop.
3. M0, M1, M2, M3 are the parallel inputs while F0, F1, F2, F3 are the parallel outputs of flip-flops.
4. When the input pin is active HIGH, then the universal shift register loads / retrieve the data in parallel. In this case, the input pin is directly connected to 4×1 MUX
5. When the input pin (mode) is active LOW, then the universal shift register shifts the data. In this case, the input pin is connected to 4×1 MUX via NOT gate.
6. When the input pin (mode) is connected to GND (Ground), then the universal shift register acts as a Bi-directional shift register.
7. To perform the shift-right operation, the input pin is fed to the 1st AND gate of the 1st flip-flop via serial input for shit-right.
8. To perform the shift-left operation, the input pin is fed to the 8th AND gate of the last flip-flop via input M.
9. If the selected pins S0= 0 and S1 = 0, then this register doesn't operate in any mode. That means it will be in a Locked state or no change state even though the clock pulses are applied.
10. If the selected pins S0 = 0 and S1 = 1, then this register transfers or shifts the data to left and stores the data.
11. If the selected pins S0 = 1 and S1 = 0, then this register shifts the data to right and hence performs the

shift-right operation.

12· If the selected pins S0 = 1 and S1 = 1, then this register loads the data in parallel. Hence it performs the parallel loading operation and stores the data.

Modes of Operation:

According to the inputs to the select lines, the following modes can be implemented in a Universal Shift Register:

1. The input '00' to the select lines refers to "locked state" wherein the register contents remain unchanged.
2. The input '01' refers to "right shift" meaning that the register contents will be shifted towards the right.
3. The input '10' indicates "left shift" which shifts the contents of the register to the left.
4. The input '11' to the select line reflects parallel loading of data into the register.

The register operations performed for the various inputs of select lines are as follows:

S1	s0	Register operation
0	0	No changes
0	1	Shift right
1	0	Shift left
1	1	Parallel load

Advantages of Universal Shift Register

1. Has the ability to perform 3 operations: shift-left, shift-right, and parallel loading.
2. Temporary storage of data within register.
3. Capable of performing serial to serial, serial to parallel, parallel to serial and parallel to parallel operations.
4. Acts as an interface between devices during data transfer.

Applications:

1. Used in micro-controllers for I/O expansion
2. Used as a serial-to-serial, parallel-to-parallel, serial-to-parallel data converter respectively.
3. Used in parallel and serial to serial data transfer.
4. Used as a memory element in computers.
5. Used in time delay and data manipulation applications.
6. Used in frequency counters, binary counters and digital clocks.

Verilog Code for CPLD :

```
module universal_shift_reg(
  input clk, rst_n,
  input [1:0] select, // select operation
  input [3:0] p_din,  // parallel data in
  input s_left_din,   // serial left data in
  input s_right_din,  // serial right data in
  output reg [3:0] p_dout, //parallel data out
  output s_left_dout, // serial left data out
```

```
  output s_right_dout // serial right data out
);
  always@(posedge clk) begin
    if(!rst_n) p_dout <= 0;
    else begin
      case(select)
        2'h1: p_dout <= {s_right_din,p_dout[3:1]}; // Right Shift
        2'h2: p_dout <= {p_dout[2:0],s_left_din};  // Left Shift
        2'h3: p_dout <= p_din; // Parallel in - Parallel out
        default: p_dout <= p_dout; // Do nothing
      endcase
    end
  end
  assign s_left_dout = p_dout[0];
  assign s_right_dout = p_dout[3];
endmodule
```

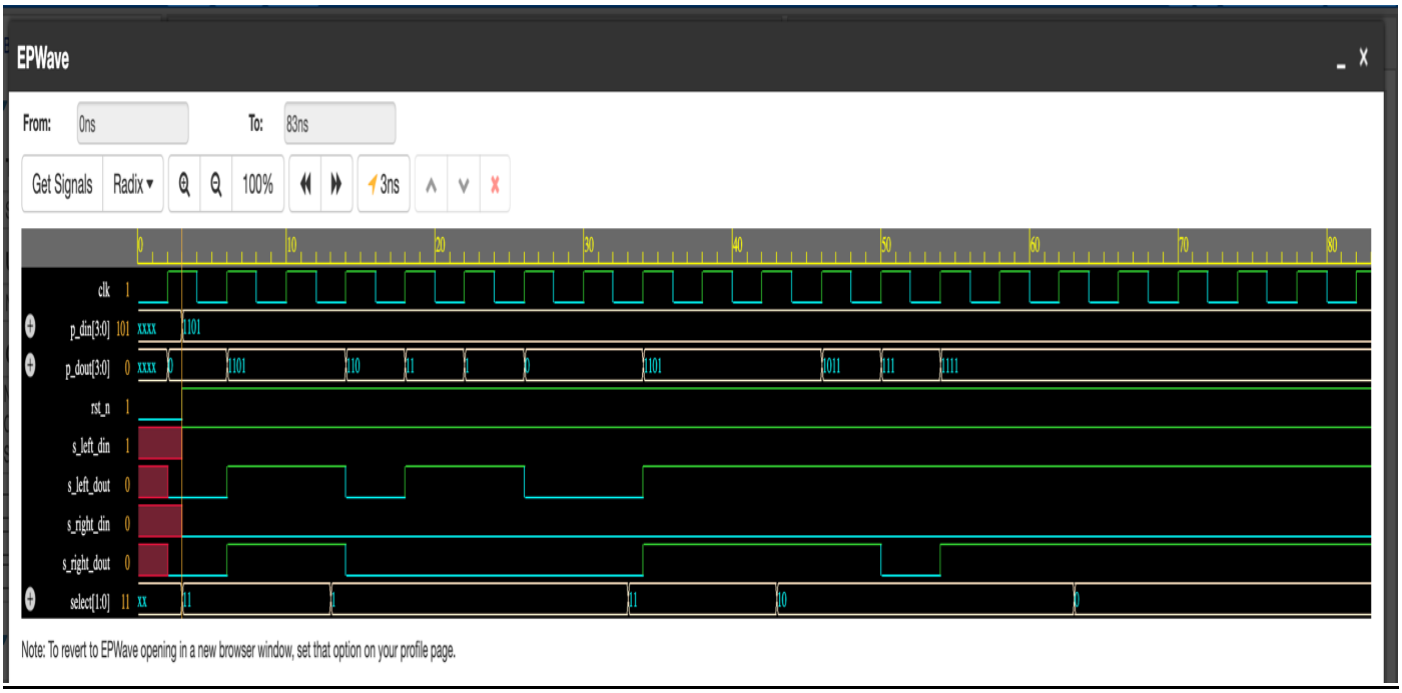
The EDA Implementation link for this : https://www.edaplayground.com/x/crpG

The WaveForms For the TeshBench :

TestBench :

```
module TB;
  reg clk, rst_n;
  reg [1:0] select;
  reg [3:0] p_din;
  reg s_left_din, s_right_din;
  wire [3:0] p_dout; //parallel data out
  wire s_left_dout, s_right_dout;

  universal_shift_reg usr(clk, rst_n, select, p_din, s_left_din, s_right_din, p_dout, s_left_dout, s_right_dout);

  always #2 clk = ~clk;
  initial begin
    $monitor("select=%b, p_din=%b, s_left_din=%b, s_right_din=%b --> p_dout = %b, s_left_dout = %b, s_right_dout =
%b",select, p_din, s_left_din, s_right_din, p_dout, s_left_dout, s_right_dout);
    clk = 0; rst_n = 0;
    #3 rst_n = 1;

    p_din = 4'b1101;
    s_left_din = 1'b1;
    s_right_din = 1'b0;

    select = 2'h3; #10;
    select = 2'h1; #20;
    p_din = 4'b1101;
    select = 2'h3; #10;
    select = 2'h2; #20;
    select = 2'h0; #20;

    $finish;
  end
  // To enable waveform
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
  end

endmodule
```

The WaveForms :



Conclusion :

The Universal Shift Register provides a flexible and efficient solution for data manipulation tasks in digital systems. Its ability to perform shifts in both directions with configurable shift amounts makes it a valuable component in a wide range of applications, from data processing algorithms to communication protocols. Through rigorous testing and verification, the USR design demonstrates reliable and accurate shift operations, meeting the requirements of modern digital systems.