**COAL Lab # 01**

**Verilog (HDL) Programing Fundamentals**

**------------------------------------------------------**

**Name: Saad Nisar Butt**

**Reg. no: cs211246**

**Class: BS-CS 3C-1**

**------------------------------------------------------**

**Lab Exercises:**

**Combinational Circuits:**

**Literature Review:**

Combinational circuits are made up of basic logic gates such as and, or and not gates. A combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs. There operations are determined by set of Boolean functions. Outputs of the combinational logic circuits depend only on the present inputs. Here, we have three inputs, two are XORed and that two are ANDed and both outputs with another input are ORed to perform certain logic.

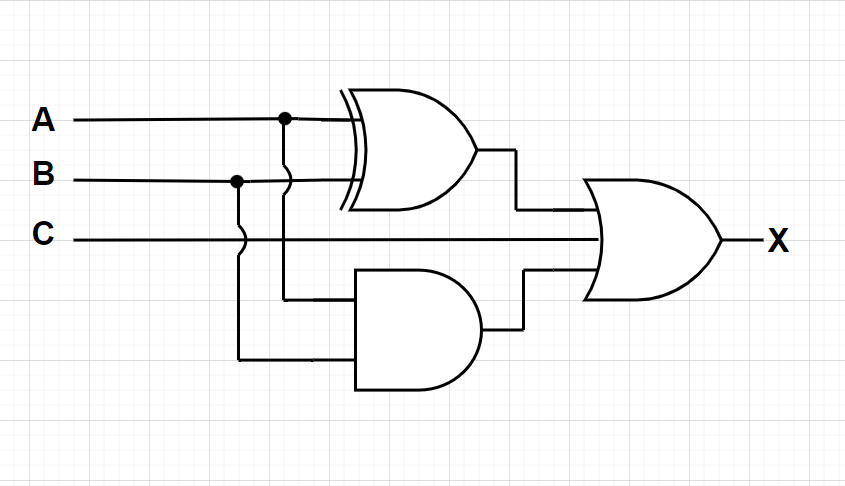
**Task 01:**

Implement combinational logic circuit using Verilog HDL.

**Truth Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **A** | **B** | **C** | **X** |
| 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 |

**Block Diagram:**

****

**Boolean Equation:**

X = A ⊕ B + A . B + C

**Verilog Code:**

design.v

module combinational\_circuit(A, B, C, Q);

    //inputs

    input A, B, C;

    output Q;

    //interim signals

    wire A\_xor\_B;

    wire A\_and\_B;

    //logic designing

    assign A\_xor\_B = A ^ B;

    assign A\_and\_B = A & B;

    // final output

    assign Q = C | A\_xor\_B | A\_and\_B;

endmodule

// assign Q = (A ^ B) | (A & B) | C

testbench.v

module tb();

    reg A, B, C;

    wire Q;

    // module declaration

    combinational\_circuit dut (

        .A(A), .B(B), .C(C), .Q(Q)

    );

    initial begin

      A <= 1'b0;

      B <= 1'b0;

      C <= 1'b0;

      #100;

      A <= 1'b0;

      B <= 1'b0;

      C <= 1'b1;

      #100;

      A <= 1'b0;

      B <= 1'b1;

      C <= 1'b0;

      #100;

      A <= 1'b0;

      B <= 1'b1;

      C <= 1'b1;

      #100;

      A <= 1'b1;

      B <= 1'b0;

      C <= 1'b0;

      #100;

      A <= 1'b1;

      B <= 1'b0;

      C <= 1'b1;

      #100;

      A <= 1'b1;

      B <= 1'b1;

      C <= 1'b0;

      #100;

      A <= 1'b1;

      B <= 1'b1;

      C <= 1'b1;

      #100;

    end

    initial begin

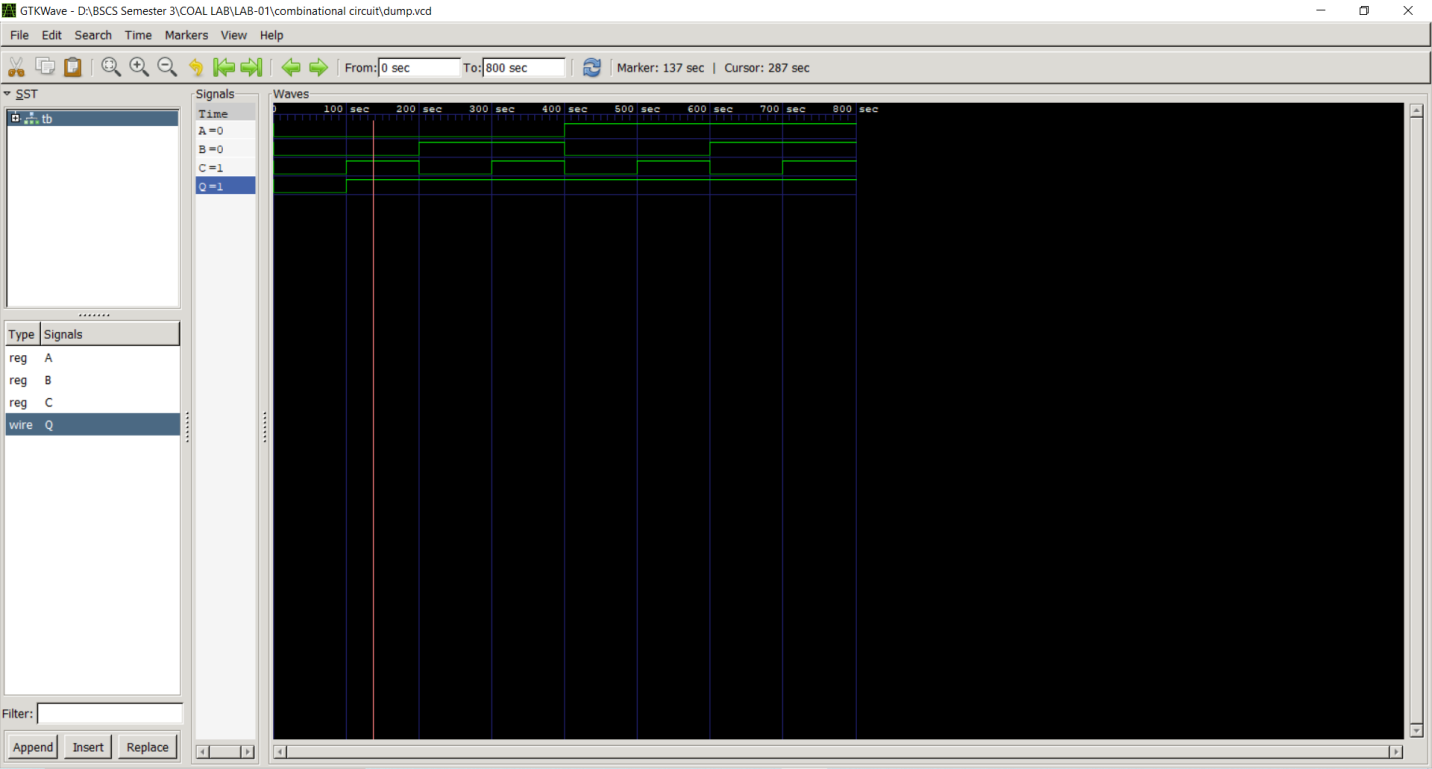
      $dumpfile("dump.vcd");

      $dumpvars(0);

    end

endmodule

**Waveforms:**

****

**-------------------------------------**

**Task 02:**

Implement sequential logic circuit using Verilog HDL.

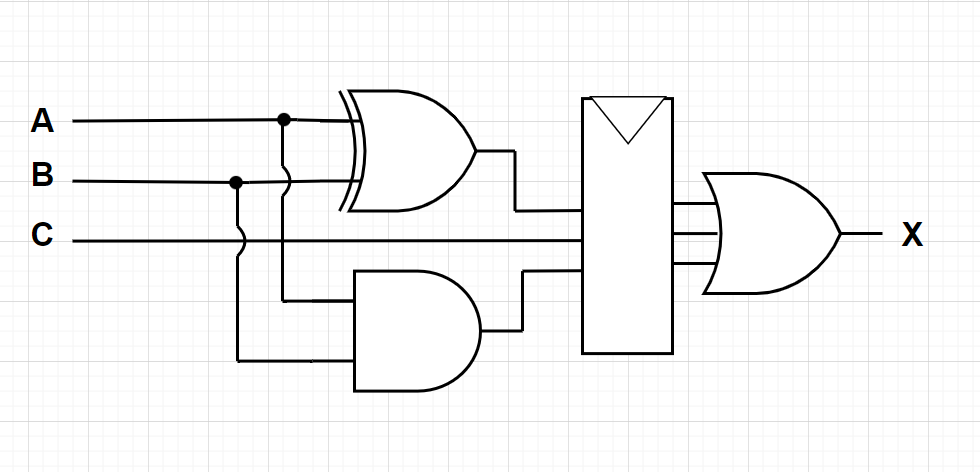
**Literature Review:**

Sequential Circuits have memory elements in addition to logic gates. They possess the ability to store the output for the computation of logic. Their outputs are a function of the inputs and the state of the storage elements. They depend upon current inputs as well as past inputs. They also include a time sequence of inputs and internal states. In this task, a combinational circuit is designed as a sequential circuit in which the output is stored in a flop just to show an example of sequential circuits.

**Truth Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **A** | **B** | **C** | **X** |
| 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 |

**Block Diagram:**

****

**Verilog Code:**

design.v

module sequential\_circuit(A, B, C, clk, reset, Q);

    // inputs

    input A, B, C, clk, reset;

    // output

    output Q;

    //interim signals

    wire A\_xor\_B;

    wire A\_and\_B;

    wire A\_or\_B\_C;

    reg flop;

    //logic designing

    assign A\_xor\_B = A ^ B;

    assign A\_and\_B = A & B;

    assign A\_or\_B\_C = C | A\_xor\_B | A\_and\_B;

    always @(posedge clk) begin

        if(reset == 1'b1)

        begin

            flop <= 1'b0;

        end

        else begin

          flop <= A\_or\_B\_C;

        end

    end

    assign Q = flop;

endmodule

// assign Q = (A ^ B) | (A & B) | C

testbench.v

module tb();

    reg A, B, C, clk, reset;

    wire Q;

    // module declaration

    sequential\_circuit dut (

        .A(A), .B(B), .C(C), .clk(clk), .reset(reset), .Q(Q)

    );

    always begin

        clk <= 1'b0;

        #50;

        clk <= 1'b1;

        #50;

    end

    initial begin

      reset <= 1'b1;

      #100;

      reset <= 1'b0;

      A <= 1'b0;

      B <= 1'b0;

      C <= 1'b0;

      #100;

      A <= 1'b0;

      B <= 1'b0;

      C <= 1'b1;

      #100;

      A <= 1'b0;

      B <= 1'b1;

      C <= 1'b0;

      #100;

      A <= 1'b0;

      B <= 1'b1;

      C <= 1'b1;

      #100;

      A <= 1'b1;

      B <= 1'b0;

      C <= 1'b0;

      #100;

      A <= 1'b1;

      B <= 1'b0;

      C <= 1'b1;

      #100;

      A <= 1'b1;

      B <= 1'b1;

      C <= 1'b0;

      #100;

      A <= 1'b1;

      B <= 1'b1;

      C <= 1'b1;

      #100;

      $finish;

    end

    initial begin

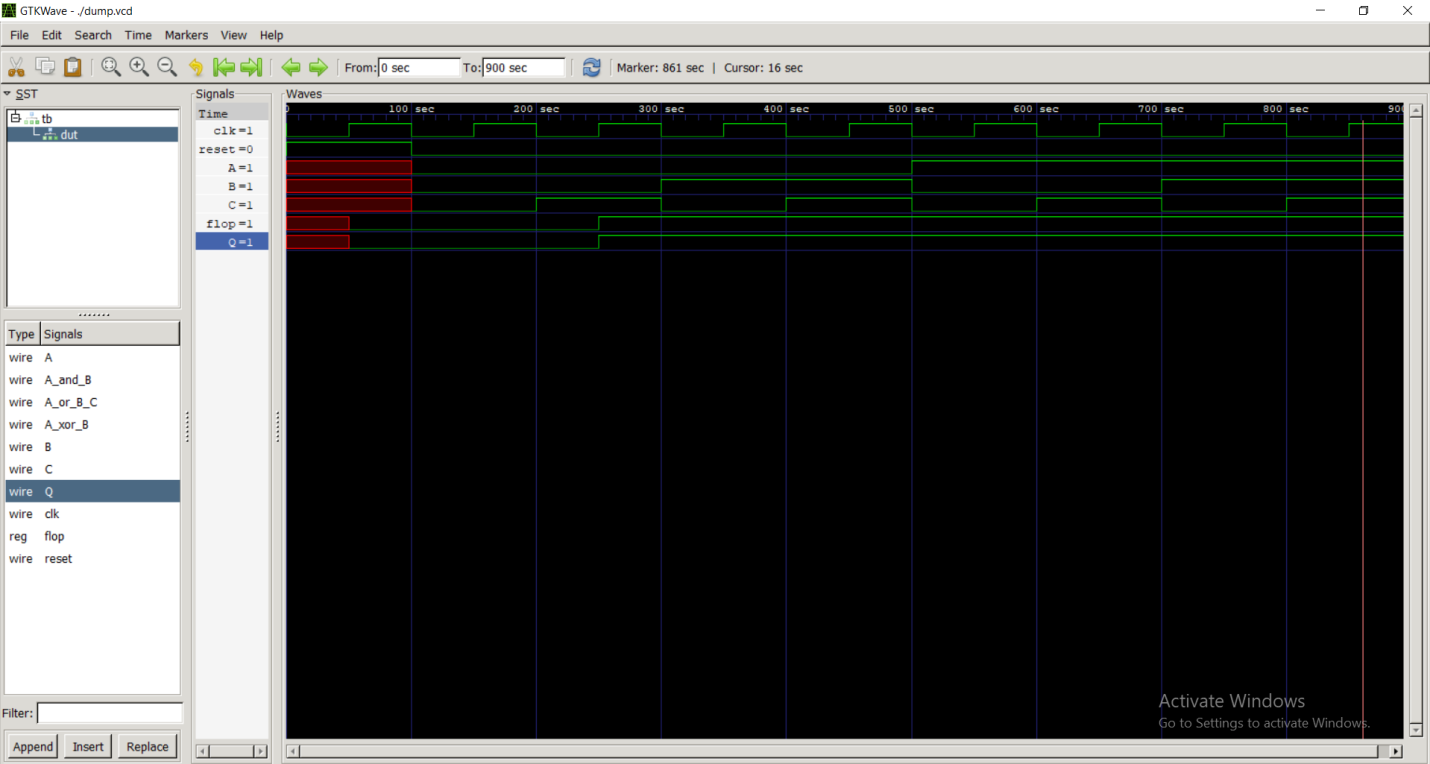
      $dumpfile("dump.vcd");

      $dumpvars(0);

    end

endmodule

**Waveforms:**

****

**-------------------------------------**

**In Lab Tasks:**

**Task 03:**

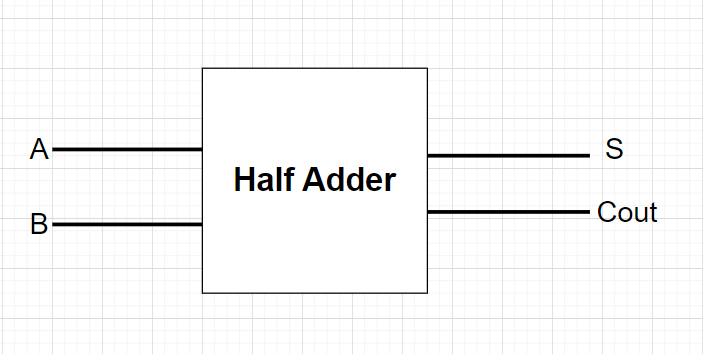
Create a Verilog module for half adder. Make a test bench for all the possible input combinations and attach the waveforms.

**Half Adder**

**Literature Review:**

Half adder is a combinational and arithmetic logic circuit which is used to add two binary digits. The adder accepts two inputs and produces tow outputs, one is sum and a carry out. Inputs are XORed to produce sum and then ANDed to produce carry out. If two half adders are cascaded, one full adder can be formed which can compute more than two bits addition.

**Block Diagram:**

****

**Truth Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **A** | **B** | **S** | **C** |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |

**Boolean Equation:**

S = A **⊕** B

C = A . B

**Verilog Code:**

design.v

module half\_adder(A, B, S, Cout);

    // inputs

    input A, B, C;

    // outputs

    output S, Cout;

    //logic designing

    assign S = A ^ B;

    assign Cout = A & B;

endmodule

testbench.v

module tb();

    reg A, B;

    wire S, Cout;

    // module declaration

    half\_adder dut (

        .A(A), .B(B), .S(S), .Cout(Cout)

    );

    initial begin

      A <= 1'b0;

      B <= 1'b0;

      #100;

      A <= 1'b1;

      B <= 1'b0;

      #100;

      A <= 1'b0;

      B <= 1'b1;

      #100;

      A <= 1'b1;

      B <= 1'b1;

      #100;

    end

    initial begin

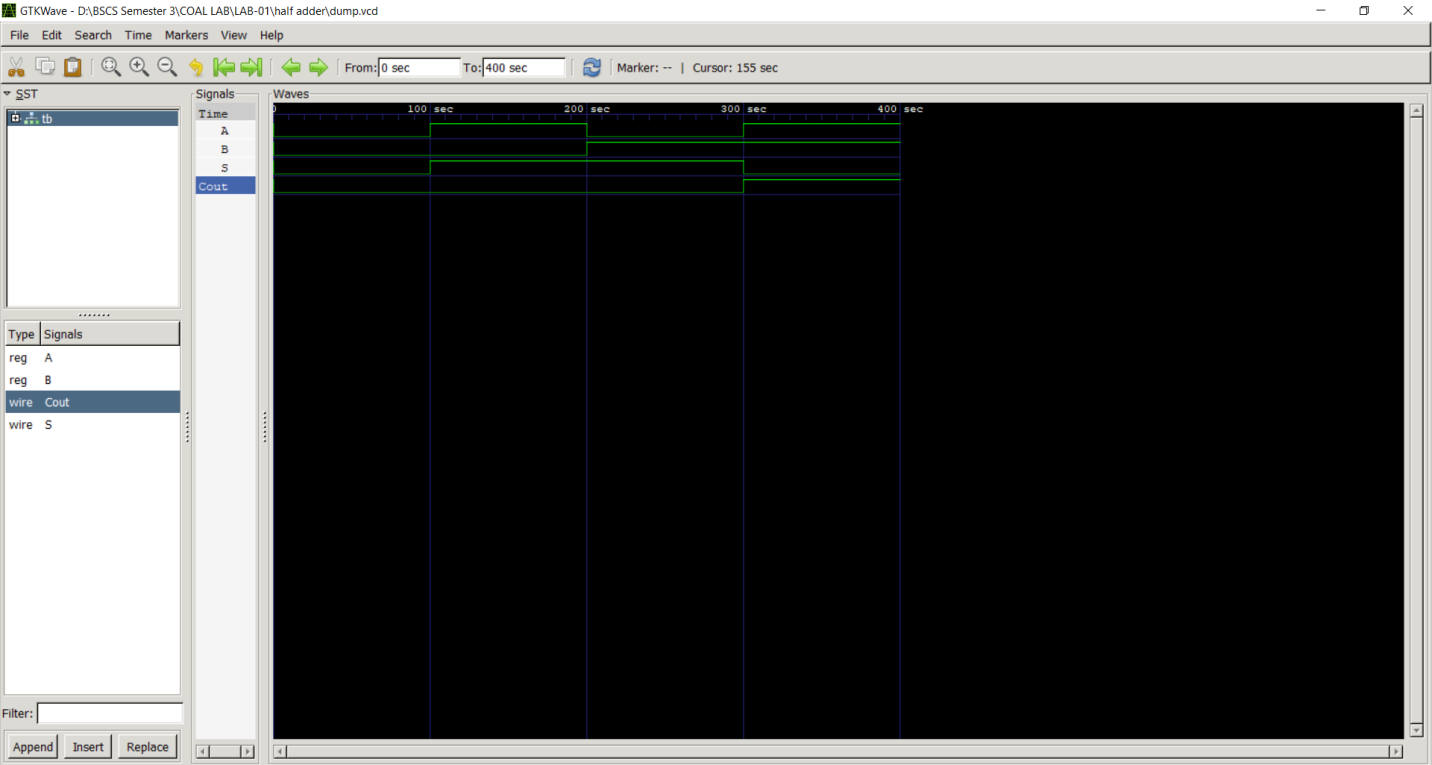
      $dumpfile("dump.vcd");

      $dumpvars(0);

    end

endmodule

**Waveforms:**

****

**-------------------------------------**

**Task 04:**

Create a two bit counter. Make a test bench for all the possible input combinations and attach the waveforms.

**Sequential Circuit:**

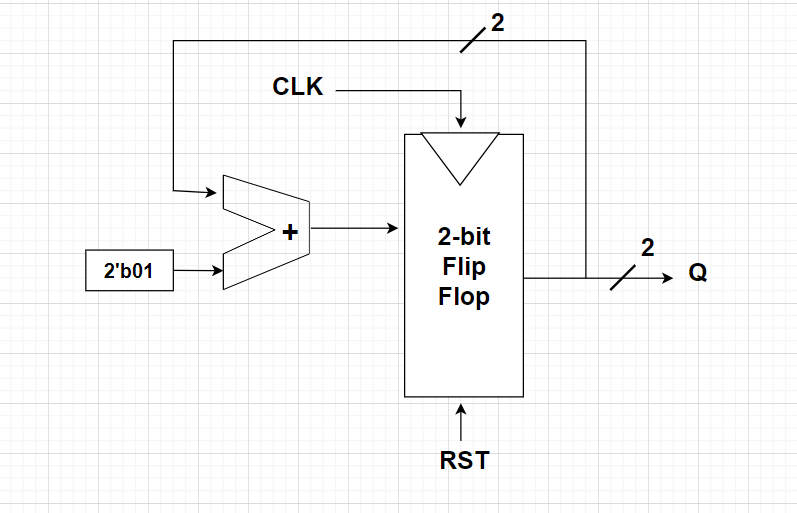
**Literature Review:**

Sequential circuits have certain applications, one of them is 2-bit counter. Counter simply increments one in the previous value and store it for the next computation. On every clock cycle, under defined conditions, one will add in the input and it will be stored in memory element and then for next cycle it will repeat the process until it reaches the limit of bits storage. And if the delay will still be there the cycle will restart from zero and keep on incrementing one.

**Truth Table:**

|  |  |  |
| --- | --- | --- |
| **CLK** | **Q1** | **Q2** |
| Initial | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 1 | 0 |
| 3 | 1 | 1 |
| 4 | 0 | 0 |

**Block Diagram:**

****

**Verilog Code:**

design.v

module two\_bit\_counter(input clk, reset,  output[1:0] Q);

    //input

    reg[1:0] flop;

    always @(posedge clk) begin

        if(reset == 1'b1)

        begin

            flop <= 2'b00;

        end

        else begin

          flop <= flop + 2'b01;

        end

    end

    assign Q = flop;

endmodule

testbench.v

module tb();

    reg clk, reset;

    wire[1:0] Q;

    // module declaration

    two\_bit\_counter dut (

        .clk(clk), .reset(reset), .Q(Q)

    );

    // clock cycle

    always begin

        clk <= 1'b0;

        #50;

        clk <= 1'b1;

        #50;

    end

    initial begin

      reset <= 1'b1;

      #100;

      reset <= 1'b0;

      #500;

      $finish;

    end

    initial begin

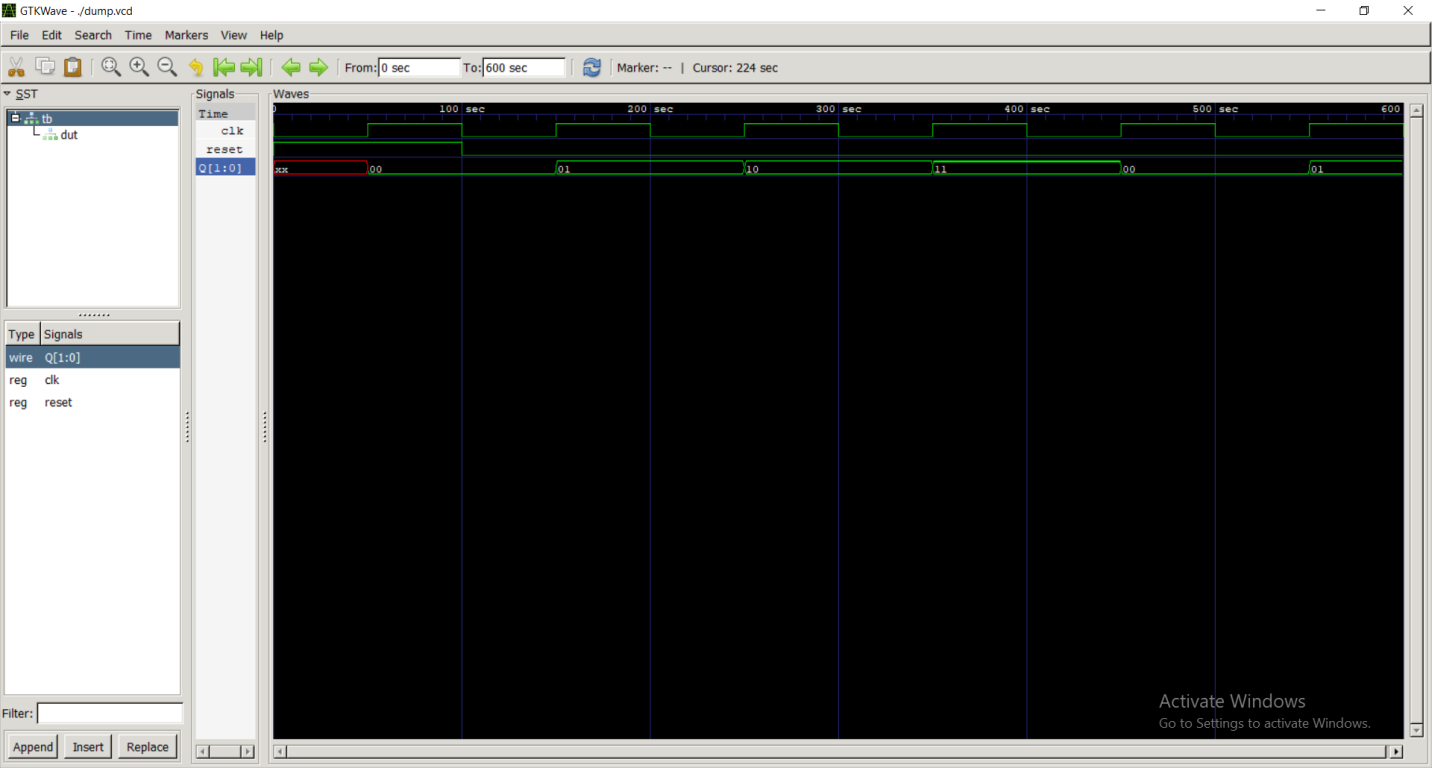
      $dumpfile("dump.vcd");

      $dumpvars(0);

    end

endmodule

**Waveforms:**

****

**-------------------------------------**

**Post Lab Task:**

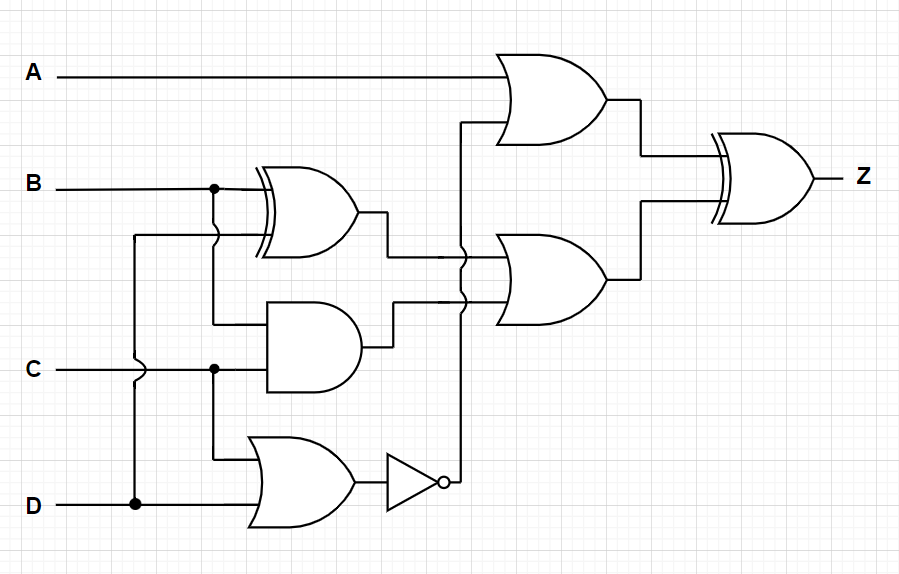
**Task 05:**

Below is the gate level circuit for the combinational circuit. Write down a data flow model for it. Hint: First write the Boolean expression for the below circuit.

**Literature Review:**

Combinational circuits are made up of basic logic gates such as and, or and not gates. A combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs. There operations are determined by set of Boolean functions. Mainly, combinational circuits are used to design complex logics such as alarm system for stuck elevator or traffic lights etc. Here, we have four inputs, A,B,C and D. which are computed logically using AND, OR, NOT and XOR gate performing certain logic.

**Block Diagram:**



**Truth Table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **B** | **C** | **D** | **Z** |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |

**Boolean Equation:**

Z = A+ ⊕ (B ⊕ D) + B.C

**Verilog Code:**

design.v

module combinational\_circuit(A, B, C, D, Z);

    // inputs

    input A, B, C, D;

    // interim wires

    wire B\_xor\_D;

    wire B\_and\_C;

    wire C\_or\_D;

    wire not\_C\_or\_D;

    wire A\_or\_not\_C\_or\_D;

    wire B\_xor\_D\_or\_B\_and\_C;

    // output

    output Z;

    // assignment with logical operations

    assign B\_xor\_D = B ^ D;

    assign B\_and\_C = B & C;

    assign C\_or\_D = C | D;

    assign not\_C\_or\_D = !(C\_or\_D);

    assign A\_or\_not\_C\_or\_D = A | not\_C\_or\_D;

    assign B\_xor\_D\_or\_B\_and\_C = B\_xor\_D | B\_and\_C;

    // final output

    assign Z = A\_or\_not\_C\_or\_D ^ B\_xor\_D\_or\_B\_and\_C;

endmodule

// assign Z = (A | (!(C | D))) ^ ((B ^ D) | (B & C));

testbench.v

module tb();

    reg A, B, C, D;

    wire Z;

    // module declaration

    combinational\_circuit dut (

        .A(A), .B(B), .C(C), .D(D), .Z(Z)

    );

    initial begin

      A <= 1'b0;

      B <= 1'b0;

      C <= 1'b0;

      D <= 1'b0;

      #100;

      A <= 1'b0;

      B <= 1'b0;

      C <= 1'b0;

      D <= 1'b1;

      #100;

      A <= 1'b0;

      B <= 1'b0;

      C <= 1'b1;

      D <= 1'b0;

      #100;

      A <= 1'b0;

      B <= 1'b0;

      C <= 1'b1;

      D <= 1'b1;

      #100;

      A <= 1'b0;

      B <= 1'b1;

      C <= 1'b0;

      D <= 1'b0;

      #100;

      A <= 1'b0;

      B <= 1'b1;

      C <= 1'b0;

      D <= 1'b1;

      #100;

      A <= 1'b0;

      B <= 1'b1;

      C <= 1'b1;

      D <= 1'b0;

      #100;

      A <= 1'b0;

      B <= 1'b1;

      C <= 1'b1;

      D <= 1'b1;

      #100;

      A <= 1'b1;

      B <= 1'b0;

      C <= 1'b0;

      D <= 1'b0;

      #100;

      A <= 1'b1;

      B <= 1'b0;

      C <= 1'b0;

      D <= 1'b1;

      #100;

      A <= 1'b1;

      B <= 1'b0;

      C <= 1'b1;

      D <= 1'b0;

      #100;

      A <= 1'b1;

      B <= 1'b0;

      C <= 1'b1;

      D <= 1'b1;

      #100;

      A <= 1'b1;

      B <= 1'b1;

      C <= 1'b0;

      D <= 1'b0;

      #100;

      A <= 1'b1;

      B <= 1'b1;

      C <= 1'b0;

      D <= 1'b1;

      #100;

      A <= 1'b1;

      B <= 1'b1;

      C <= 1'b1;

      D <= 1'b0;

      #100;

      A <= 1'b1;

      B <= 1'b1;

      C <= 1'b1;

      D <= 1'b1;

      #100;

    end

    initial begin

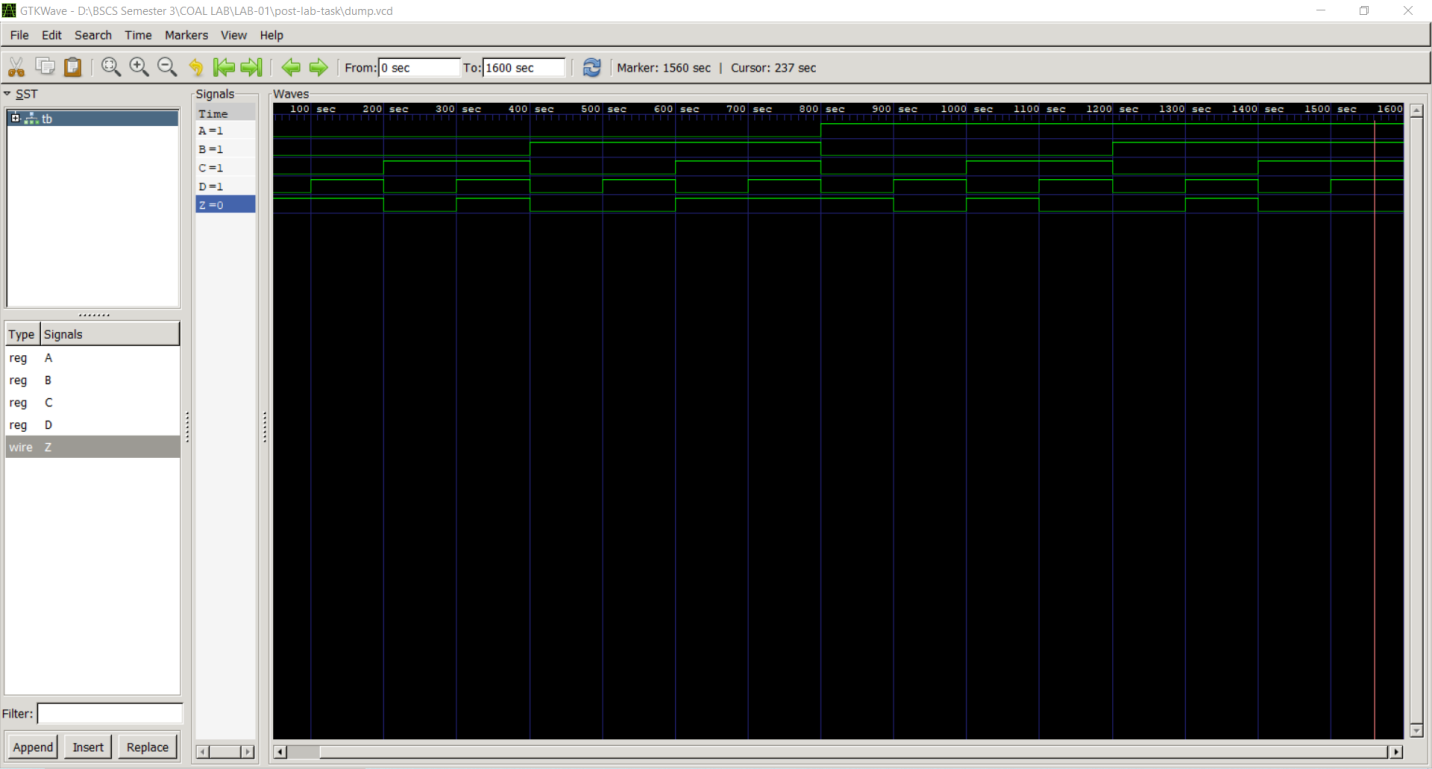
      $dumpfile("dump.vcd");

      $dumpvars(0);

    end

endmodule

**Waveform:**

****

**\* -------------------------------------- \***