LAB 1

Team 6

鄭聰明 **Jason Theodorus Pratama (109006236)**

鐵特德 **Tuguldur Tserenbaljir (109006271)**

I . 4bit 1x4 DMUX (using 3 1x2 DMUX)

a. Design

Firstly to design the DMUX, we need to know the number of select lines.

$n = 4$ // Number of outputs

$m = log_2 n$ // Number of select lines

$m = 2$ //Number of select lines

Then we can get the following truth table.

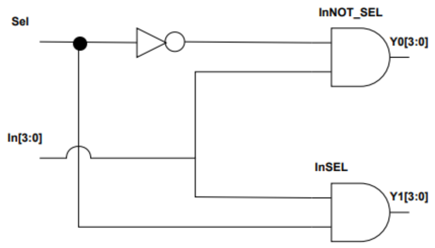
Select	**Select**	**Data**	**Output**	**Output**	**Output**	**Output**
S1	**S0**	**In**	**a**	**b**	**c**	**d**
0	0	**0**	**0**	0	0	0
0	0	**1**	**1**	0	0	0
0	1	**0**	0	**0**	0	0
0	1	**1**	0	**1**	0	0
1	0	**0**	0	0	**0**	0
1	0	**1**	0	0	**1**	0
1	1	**0**	0	0	0	**0**
1	1	**1**	0	0	0	**1**

Thus, from the table we can get our logic expression for the output where

$a = \bar{S}_1 \bar{S}_0 In$, $b = \bar{S}_1 S_0 In$, $c = S_1 \bar{S}_0 In$, $d = S_1 S_0 In$. // logic expression

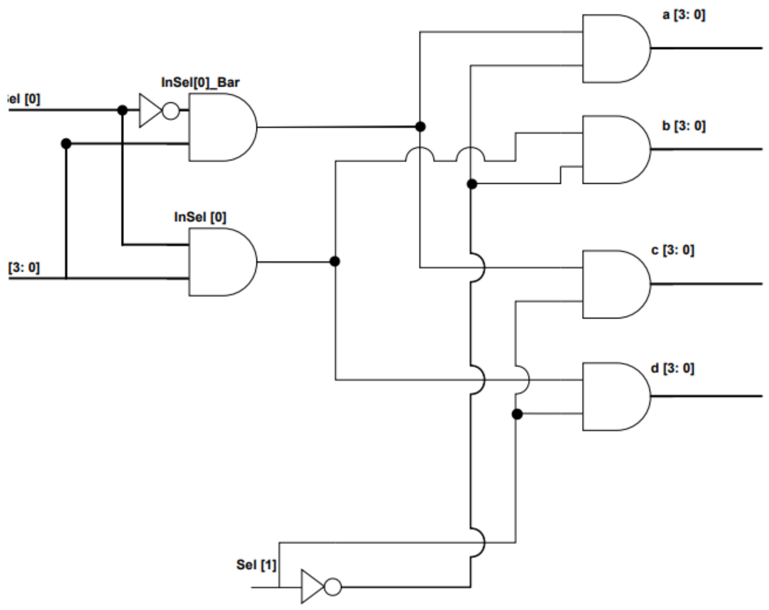
Following the base design of a 1x2 DMUX (**Figure 1.1**), we can get our final design of a 1x4 DMUX (**Figure 1.2**) using only gate-level circuits. For the basic 1x2 DMUX, we only need 1 select line whereas the outputs will be Y_0 and Y_1. Since the input is 4-bit, the outputs will also be 4-bit.

$Y_0 = \bar{S}_0 In$, $Y_1 = S_0 In$



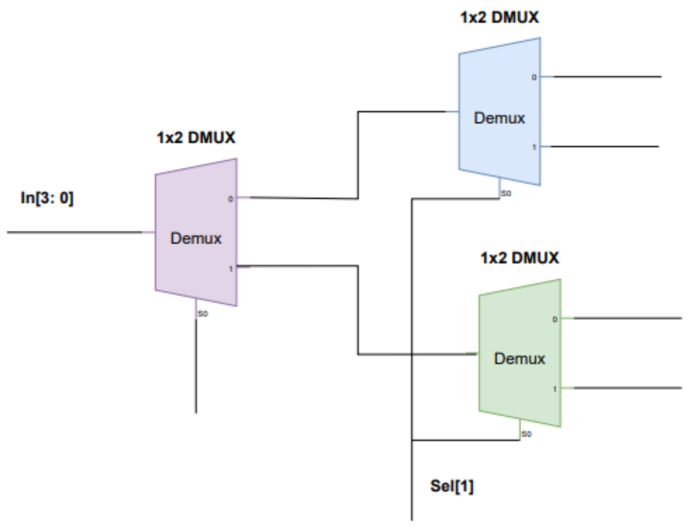
(Figure 1.1 Base 1x2 DMUX design)

Now that we have done our first part, we can add the final select design and finish the full logic circuit design by following our logic expression from above. After we have our final design of the 1x4 DMUX, we can test it by inputting 1's and 0's in our logic circuit and see if it is faulty or not from our table which we got earlier. (Since the project is 4-bit, the input and the output will be 4-bit.)



(Figure 1.2 1x4 DMUX using three 1x2 DMUX)

Moreover, we can simplify our gate-level circuits in the following picture (**Figure 1.3**).



(Figure 1.3 Simplified picture of 1x4 DMUX from three 1x2 DMUX)

b. Circuit

```
`timescale 1ns/1ps

module Dmux_1x4_4bit(in, a, b, c, d, sel);

    input [4-1:0] in;
    input [2-1:0] sel;
    output [4-1:0] a, b, c, d;

    wire [3:0] w1,w2;

    Dmux_1x2_4bit DMUX1(w1, w2, sel, in);
    Dmux_1x2_4bit DMUX2(a, b, sel, w1);
    Dmux_1x2_4bit DMUX3(c, d, sel, w2);

endmodule

module Dmux_1x2_4bit(a,b,sel,in);
```

```
   input [3:0] in;
   input [1:0] sel;
   output [3:0] a, b;

   wire notsel;

   not not1(notsel, sel);

   // 4bit outputs
   and and1(a[3], notsel, in);
   and and2(b[3], sel, in);

   and and3(a[2], notsel, in);
   and and4(b[2], sel, in);

   and and5(a[1], notsel, in);
   and and6(b[1], sel, in);

   and and7(a[0], notsel, in);
   and and8(b[0], sel, in);

endmodule
```

Our Vivado Verilog code for the lab will have 2 modules which are **module Dmux_1x4_4bit** and **module Dmux_1x2_4bit**.

Module Dmux_1x4_4bit:

The module will have an input of 4-bit, 2 selection lines, and four(a,b,c,d) 4-bit outputs. We wire w1 and w2, so that they can be used to save the value after the first 1x2 DMUX execution ($Y_0 = \bar{S_0}In$, $Y_1 = S_0 In$). Since we have to use three **1x2 4-bit Dmux**, the module **Dmux_1x2_4bit** will be called 3 times. Following the logic circuit design from above, after calculating w1 and w2, then it will pass a and b, and, c and d respectively to the module **Dmux_1x2_4bit**.

Module Dmux_1x2_4bit:

The module takes a 4-bit input and calculates for each input which the output is also 4-bit with the help of w1 and w2. Before calculating each bit for each output, we wire another (notsel). Which is crucial for our design to work properly.

For example:

To make sure our code follows the logical way of our logic expression, let's take a and b output as an example. The output a is $a = \bar{S}_1\bar{S}_0 In$ and $b = \bar{S}_1 S_0 In$. After the first call of module **Dmux_1x2_4bit** w1 will be $w_1 = \bar{S}_0 In$ and w2 will be $w_2 = S_0 In$. On the second call of the module, it will calculate for every bit for a and b starting from 0 to 4 will have be completed with the second selection line.

c. Testbench

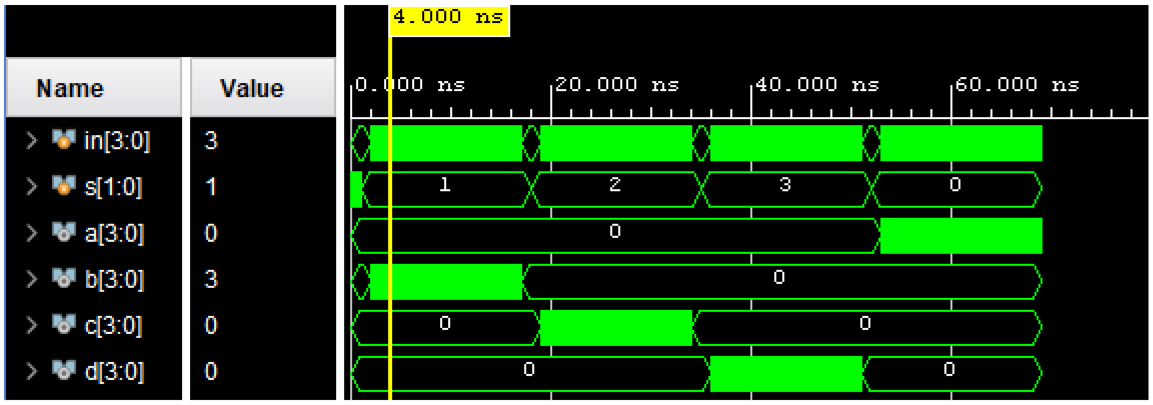
```
`timescale 1ns / 1ps

module Dmux_1x4_4bit_t;
   reg [3:0] in;
   reg [1:0] s;
   wire [3:0] a, b, c, d;

   Dmux_1x4_4bit dmux1(
      .in(in),
      .sel(s),
      .a(a),
      .b(b),
      .c(c),
      .d(d)
   );

   initial begin
      s = 2'b0; //declare 00
      in = 4'b0; // declare 0000
```

```
    repeat (2 ** 2) begin // 2bit
      #1 s = s + 2'b1; // 1ns delay
      in = 4'b0; // declare in as 0000
      repeat (2 ** 4) begin //4bit
        #1 in = in + 4'b1; // 1ns delay, for loop for adding 0000, 0001,0010,0011 until
1111,overflow 0000
      end
    end
    #1 $finish; //1ns delay
  end

endmodule
```

First, we define the 4-bit 1-to-4 DMUX with it's corresponding 4-bit input, selection line and four 4-bit outputs.Then we iniliaze the selection value to '00' and input to '0000'. Since the selection line is only two bits there are 4 possibility and the input will have 16 possibilities. To make it easier to code, we use two repeater, one for selection line and one for input. Each loop will have a 1 nanosecond delay and it will keep adding up until '1111' from '0000'. After '1111', the function will overflow as the next value will be '0000' and end.

Finally, after making sure our Verilog transformation of the gate-level logic circuits, we can also see our waveforms and double check our work. Our testbench checks all the possible values of outputs from input and selection start to finish, using three 1-to-2 DMUX for 1-to-4 DMUX.

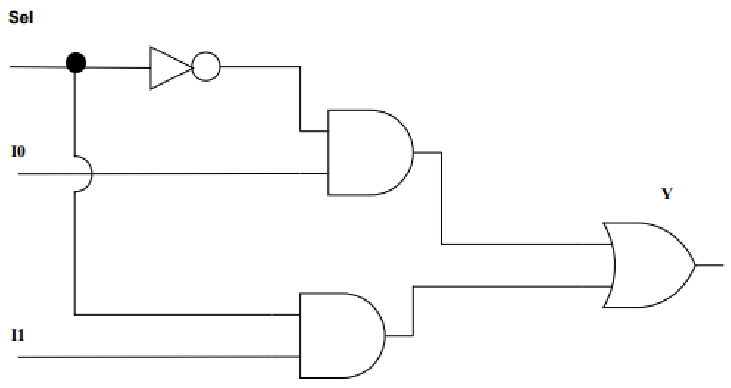
d. Waveform



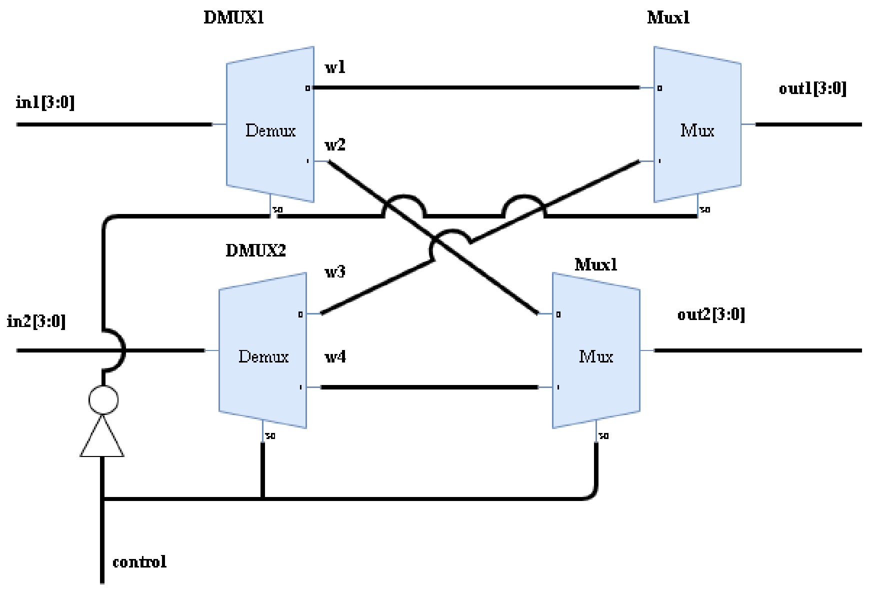
II. 4bit Simple Crossbar Switch

a. Design

For a simple 4-bit crossbar switch, we followed the pdf file where we reused our 2-to-1 MUX and 1-to-2-DMUX modules. We added our own parameters to the design shown below(**Figure 2.2)**. As the DMUX takes 4-bit inputs, each 4-bit outputs are the input of the 2-to-1 MUX. The crossbar has a control where if it is equal to 1'b1 it is the design to be crossed (the two 4-bit output values will be dependent on w2 and w3), but if the control is 1'b0 it is a bar (the two 4-bit output will be dependent on w1 and w4). But before it shows the outputs, the 2-to-1 MUX module calculates it. The control controls the selection line for every module with a bit. For the 4-bit 2-to-1 MUX design, we used the following logic gates **(Figure 2.1).**

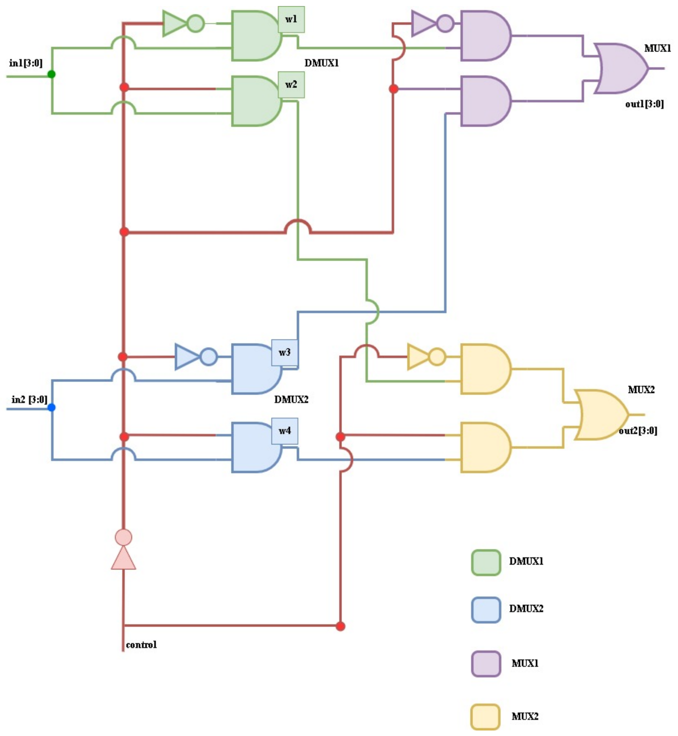


(Figure 2.1 basic 2-to-1 MUX)



(Figure 2.2 Crossbar switch)

In addition, if we were to put the modules into a basic logic circuit, the transformation is shown below **(Figure 2.3)**.



(Figure 2.3 basic logic gate Crossbar switch)

b. Circuit

```
module Crossbar_2x2_4bit(in1, in2, control, out1, out2);

    input [4-1:0] in1, in2;
    input control;
    output [4-1:0] out1, out2;

    wire notcontrol;
    wire [3:0] w1, w2, w3, w4;

    not not1(notcontrol, control);

    Dmux_1x2 Dmux1(in1, w1, w2, control);
    Dmux_1x2 Dmux2(in2, w3, w4, notcontrol);

    Mux_2x1 Mux1(w1,w3,control,out1);
    Mux_2x1 Mux2(w2,w4,notcontrol,out2);

endmodule
```

To start the Crossbar_2x2_4but module, we are given 4-bit inputs and outputs and one control input. Thus, we wire notcontrol for the selection line simplicity, and w1,w2,w3, and w4 to save the 1-to-2 DMUX 4-bit outputs to be 4-bit inputs for the 2-to-1 MUX. The control will serve as a selection line for our modules and since our modules of 1-to-2 DMUX and 2-to-1 MUX both have a logic of not for the selection line, to make it work properly we will pass on the opposite of control into the modules. After each 1-to-2 DMUX calculates w1,w2,w3, and w4 respectively, it will pass into the 2-to-1 MUX module. Finally, it will output the two 4-bit values.

c. Testbench

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

```
module Crossbar_2x2_4bit_t;
   reg[3:0]
   in1 = 4'b0,
   in2 = 4'b0;
   reg
   control = 1'b0;
   wire[3:0] out1, out2;

   Crossbar_2x2_4bit SCB(
      .control(control),
      .in1(in1),
      .in2(in2),
      .out1(out1),
      .out2(out2)
   );

   initial begin
      repeat (2) begin
         #1 control = control + 1'b1;
         repeat (2 ** 4) begin
            #1 in1 = in1 + 4'b1;
            repeat (2 ** 4) begin
               #1 in2 = in2 + 4'b1;
            end
         end
      end
      #1 $finish;
   end
endmodule
```

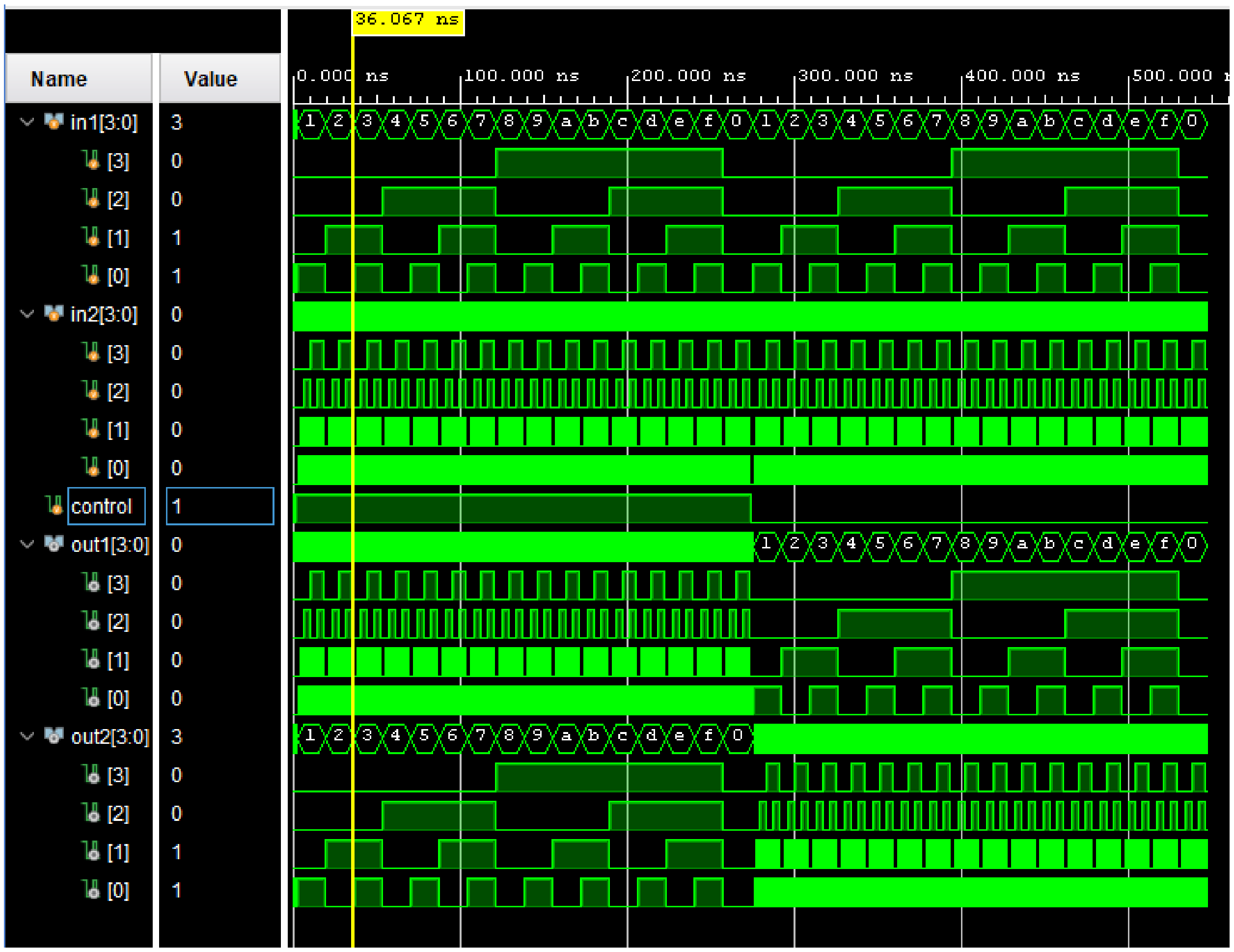
To start, we declare the inputs and outputs of the Crossbar module with the simulation Unit and accuracy above. We reg 4-bit in1 and in2, wire 4-bit outputs and reg a control line.

After, declaring our module, we instantiate our Cross_2x2_4bit design and give a name (SCB). Then, the initial conditions are written as the codes from above.

In the initial conditions, two inputs will have each 16 possibilities and there are two types of control line. To satisfy this logic statement, we use a repeater for control and a repeater for each input until it calculates all the corresponding outputs and end. Every loop has a 1 nanosecond delay and before finishing it will also have a 1 nanosecond delay as well.

After, Vivado compiles successfully we can see our waveform and see if it matches our logic circuit gate-level design from above.

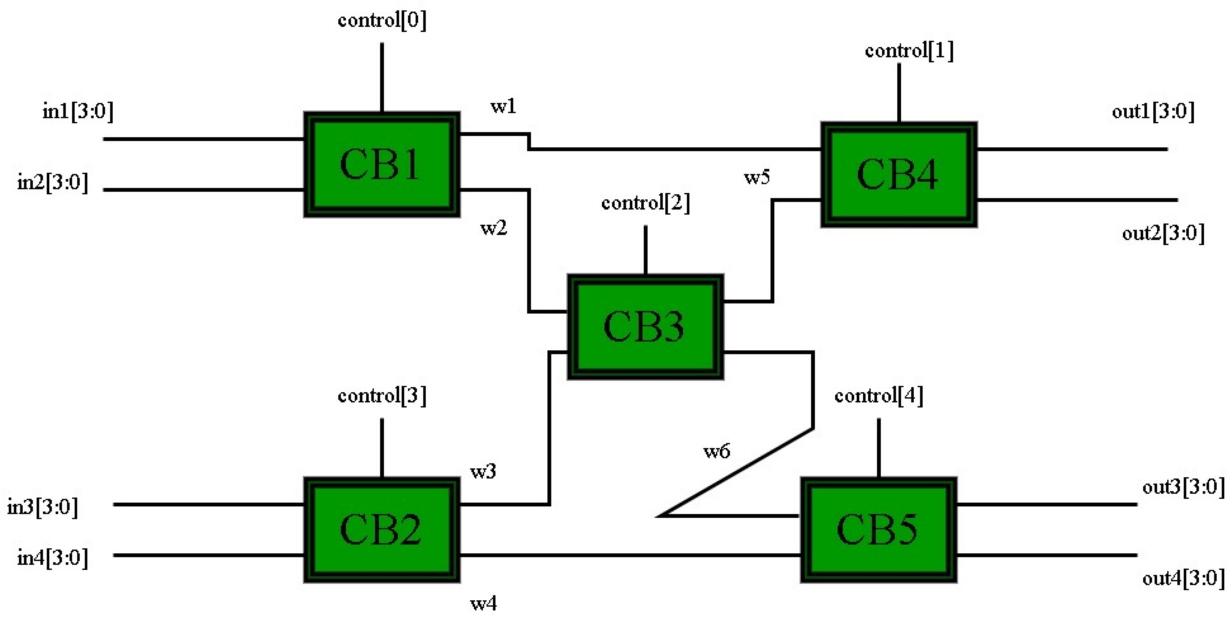
d. Waveform



III. 4bit 4x4 Crossbar with Crossbar Switch

a. Design

Following to slides from the pdf file, we named our wires to w1,w2,w3,w4,w5, and w6 respectively to each 2x2 crossbar **(Figure 3.1)**. Since we already know what a 2x2 crossbar with a simple crossbar switch looks and behaves from our last question, we can easily calculate how a 4bit 4x4 crossbar with a simple crossbar switch algorithm works. Given four 4-bit inputs, each 2x2 crossbar with control input will take two inputs respectively. After receiving the inputs, the first 2 2x2 crossbar will generate 4 outputs w1,w2 from crossbar 1 and, w3 and w4 from crossbar 2. As w2 and w3 will serve as 4-bit inputs to crossbar 3 with its own control (control 2). Crossbar 3 will calculate the final w5 and w6 whereas w5 and w1 will be 4-bit inputs for crossbar 4, and crossbar 5 will have w6 and w4 as inputs. Finally, the last two crossbars with their own control will calculate the four 4-bit outputs for our system. However, due to the control line, the logic will either be bar or cross. Much like the 2x2 crossbar switch system, they will look a cross or a bar depending on the control line. Hence, there will be some combinations of input and output that cannot be achieved by this crossbar switch due to the control line controlling the system to either be a cross or a bar. They are [(in1, out3), (in2, out4), (in3, out1), (in4, out2)]).



(Figure 3.1 4 bit 4x4 Crossbar with simple crossbar switch)

In addition, if we were to draw the 4 bit 4x4 crossbar from basic logic gates, it would like this with our implementation (**Figure 3.2**) .

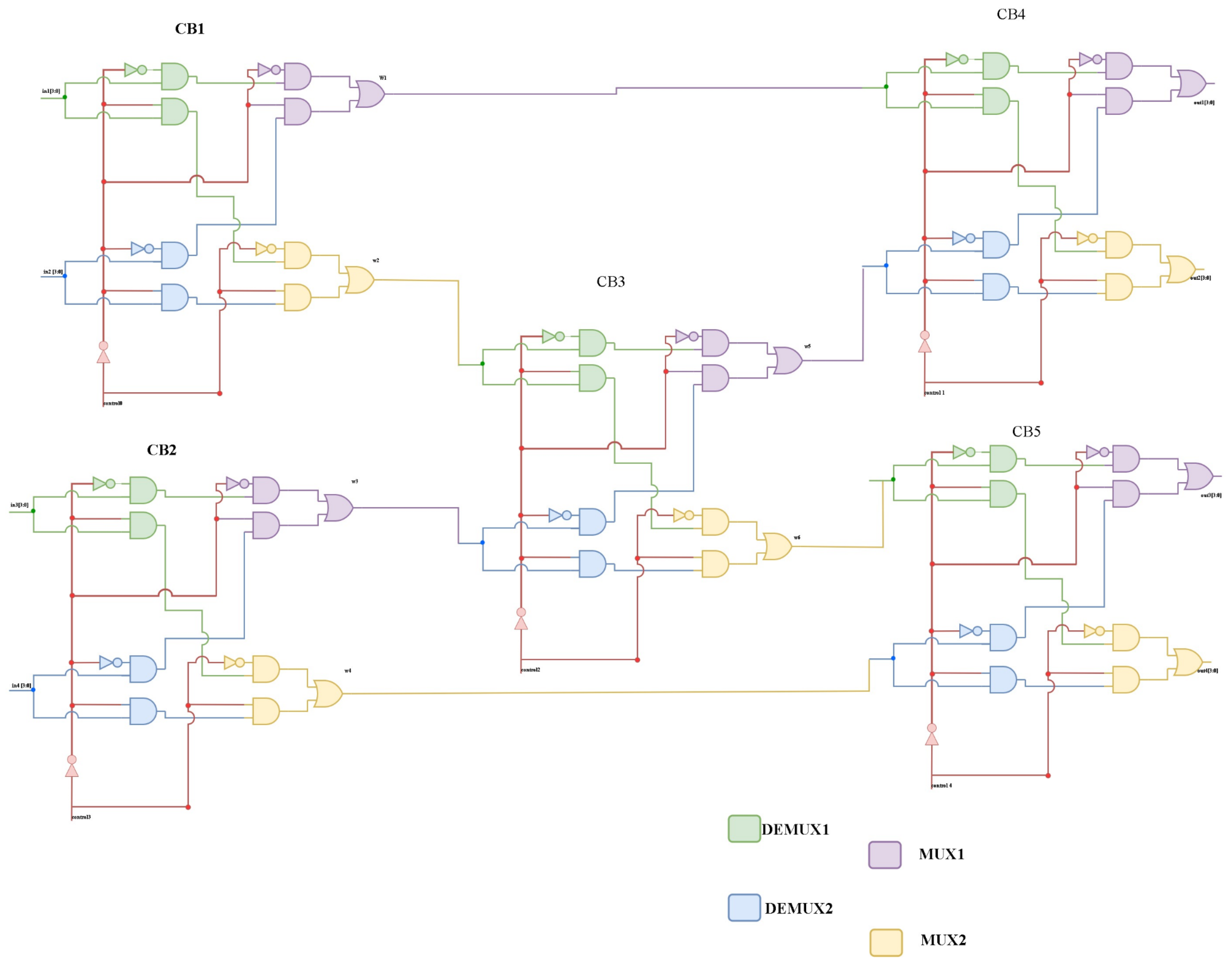


Figure 3.2 4 bit 4x4 Crossbar with simple crossbar switch basic gate)

b. Circuit

```
module Crossbar_4x4_4bit(in1, in2, in3, in4, out1, out2, out3, out4, control);
   input [4-1:0] in1, in2, in3, in4;
   input [5-1:0] control;
   output [4-1:0] out1, out2, out3, out4;

   wire[3:0] w1,w2,w3,w4,w5,w6;

   Crossbar_2x2_4bit CB1(in1, in2, control[0], w1 ,w2);
   Crossbar_2x2_4bit CB2(in3, in4, control[3], w3, w4);

   Crossbar_2x2_4bit CB3(w2, w3, control[2], w5, w6);

   Crossbar_2x2_4bit CB4(w1, w5, control[1], out1, out2);
   Crossbar_2x2_4bit CB5(w6, w4, control [4], out3, out4);

endmodule
```

To follow the flow of the logical system from above, we have four 4-bit inputs, control lines, and four 4-bit outputs. Thus, we also wire our w1,w2,w3,w4,w5 and w6 to save the logical calculations of the 2x2 crossbars. We call the module Crossbar_2x2_4bit from previous question for each crossbar, in total 5. Moreover, each crossbar module will calculate it's 1-to-2 DMUX and 2-to-1 MUX from the modules we also implemented from earlier.

c. **Testbench**

```
`timescale 1ns / 1ps



module Crossbar_4x4_4bit_t;
   reg[3:0]
   in1 = 4'b0001,
   in2 = 4'b0010,
   in3 = 4'b0100,
   in4 = 4'b1000;

   reg [4:0]
   control  = 5'b0;

   wire [3:0] out1, out2, out3, out4;

   Crossbar_4x4_4bit CB4x41(
      .control(control),
    .in1(in1),
    .in2(in2),
    .in3(in3),
    .in4(in4),
    .out1(out1),
    .out2(out2),
    .out3(out3),
    .out4(out4)
    );

  initial begin
    repeat (2 ** 5) begin
      #1 control = control + 5'b1;
      repeat(2 ** 4) begin
        #1 in1 = in1 + 4'b1;
        repeat(2 ** 4) begin
          #1 in2 = in2 + 4'b1;
          repeat(2 ** 4) begin
            #1 in3 = in3 + 4'b1;
            repeat(2 ** 4) begin
              #1 in4 = in4 + 4'b1;
            end
          end
        end
      end
    end
    #1 $finish;
  end

endmodule
```

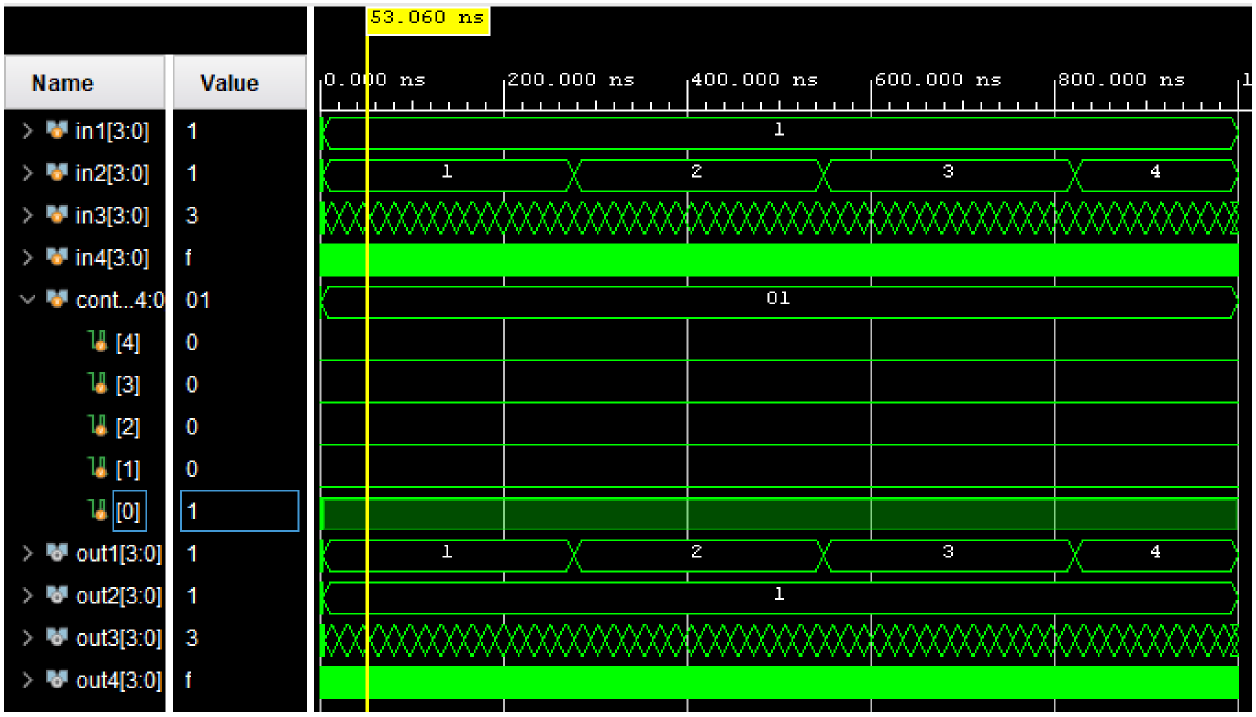
To start, we declare the inputs and outputs of the Crossbar module with the simulation Unit and accuracy above. We reg 4-bit in1,in2,in3 and in4, wire 4-bit out1,out2,out3 and out4 and reg 5 control lines for each crossbar. .

After, declaring our module, we instantiate our Cross_4x4_4bit design and give a name (**CB4x41**). Then, the initial conditions are written as the codes from above.

In this case, we start by putting the control in repeater function where it will change value everytime when it is called. To continue with the code, since there are 4-bit 4 inputs to test for each and every value, we will need 5 repeaters for each one. Thus, each repeater has a 1 nanosecond delay.

After, Vivado compiles successfully we can see our waveform and see if it matches our logic circuit gate-level design from above.

d. <u>Waveform</u>



IV. Toggle Flip Flop

a. Design

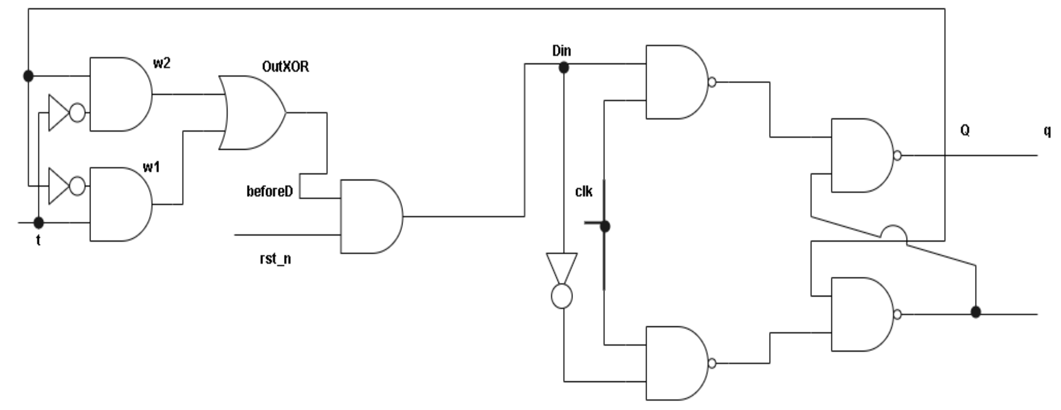
To implement the Toggle Flip Flop we reused our DFF design. We design a XOR gate made from basic gate-leveling where it takes w2 and w1. Furthermore,

$OutXOR = w_1 \oplus w_2$ // naming of wires for future simplicity

$OutXOR = t\bar{q} \oplus q\bar{t}$

$D_{in} = and(OutXOR, rst_n)$ //DFF input

Following the logic design from above we can design our gate-level basic logic circuit **(Figure 4.1)**.



(Figure 4.1 TFF basic gate-level logic circuit)

b. Circuit

```
`timescale 1ns / 1ps


module Toggle_Flip_Flop(clk, q, t, rst_n);
  input clk;
  input t;
  input rst_n;
  output q;

  wire nott,notq;
  wire w1,w2;
  wire XOROut, Din;

  //XOR
  not not1(nott, t);
  not not2(notq, q);
```

```
    and and1(w1, t, notq);
    and and2(w2, nott, q);

    or OutXOR(XOROut, w1, w2);

    and beforeD (Din, XOROut, rst_n);

    D_Flip_Flop D(clk, Din, q);

endmodule
```

For the Toggle Flip Flop module, we take clk, t, rst_n as input and q as an output. We wire the following values nott,notq,w1,w2,XOROut and Din for future use. The values of nott and notq will be not of t and q respectively. We calculate the XOR gate first were we use 2 and gates, and save the value to w1 and w2. Later before going into the D_Flip_Flop module, our code calculates Din by and(XOROut,rst_n) because we need the Din value for our DFF design to work. Finally the final D_FLIP_FLOP module takes the clk,Din,q and in this module it calculates the final part of the gate-level logic design that we have created.

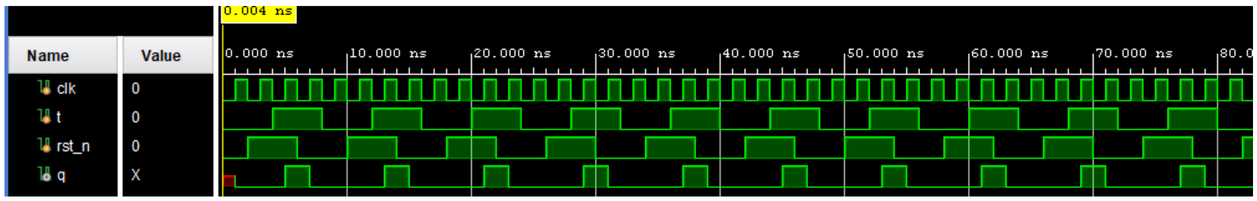
c. Testbench

```
`timescale 1ns / 1ps

module Toggle_Flip_Flop_t;
    reg
    clk = 1'b0,
    t = 1'b0,
    rst_n = 1'b0;
    wire q;

    Toggle_Flip_Flop T(
    .clk(clk),
    .t(t),
    .rst_n(rst_n),
    .q(q)
  );

  always#(1) clk = clk + 1'b1;

  always begin
    @(negedge clk) begin
      #2 rst_n = rst_n + 1'b1;
      #2 t = t + 1'b1;
    end
  end

endmodule
```

For the testbench, we reg and wire our input and outputs with the simulation unit and accuracy. Then instantiate our Toggle Flip Flop.

Then we have two always block. One for the clock to tick every 1 nanosecond. Another one for our rst_n and t to test every possible value as their delay is 2 seconds.

d. Waveform

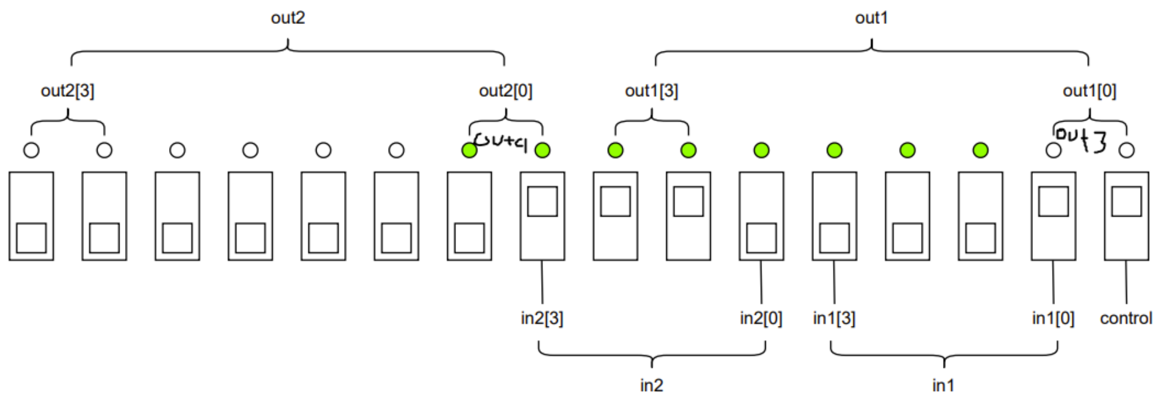


V.FPGA Implementation Code 4 2x2 Crossbar:

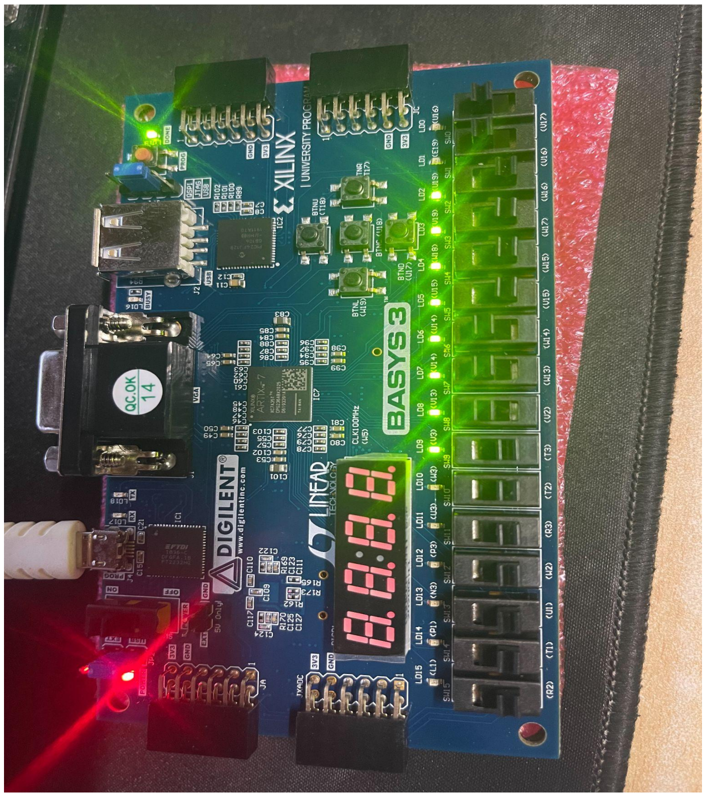
We use 4 Mux because 1 output, can't be directed into 2 lights, so we make it there are 4 outputs.

So every LED get a single variable output.

```
module Crossbar_2x2_4bit(in1, in2, control, out1, out2,out3,out4);

   input [4-1:0] in1, in2;
   input control;
   output [4-1:0] out1, out2, out3, out4;

   wire notcontrol;
   wire [3:0] w1, w2, w3, w4;

   not not1(notcontrol, control);

   Dmux_1x2_4bit Dmux1(w1, w2, control, in1);
   Dmux_1x2_4bit Dmux2(w3, w4, notcontrol, in2);

   Mux_2x1_4bit Mux1( w1, w3, control, out1);
   Mux_2x1_4bit Mux2( w2, w4, notcontrol, out2);
   Mux_2x1_4bit Mux3(w1, w3, control , out3);
   Mux_2x1_4bit Mux4(w2, w4, notcontrol, out4);

endmodule
```



Since the 1 LED can only handle 1 out variable, every pair gets a output of its own, for example, the LED E19, gets out3[0], and U16, gets out1[0], and so on for the rest of the outputs. Which we implemented with 4 MUX. and the result of it would be on the picture below



VI. What we have learned from Lab 1

Verilog, a program we both haven't written in 2 years because we took Logic Design in the 2nd Semester, so we had to restudy everything. In the 1st Lab in class, we couldn't do a single bit of Verilog, so we went on to study those introductory labs, and try to create a circuit of it in Verilog, which helps a lot when trying to come onto these Advanced Questions which was quite challenging for us.

In addition to gaining more experience in Verilog, we learned to work as a team and share the workload. Setting a certain deadline for each questionnaire and meeting the deadline within time. Coding Verilog in Vivado was challenging but as a few weeks passed and doing the lab, we became much more familiar with it. Also, coded our first FPGA board.

VII. Our sources

-https://app.diagrams.net

-https://www.youtube.com/watch?v=oR1Gec5dX9g

-https://www.youtube.com/watch?v=vHnqabIzFU8

-Google

-Stackoverflow

-Wikipedia

Vlll. List of Contributions

We divided the work into a few parts where Jason was in charge of question number 4 and Tuguldr was in charge of question 1. We gave each other until Sunday to try to do it ourselves in Lab1 and agreed to do question number 2 and 3 on Monday and Tuesday. Finally, give ourselves one full day on Wednesday to work on writing the report. Since we live in the same dorm, it was easy to meet up and discuss the Lab.