Hardware Design

Lab 1 Report

Gate-Level Verilog

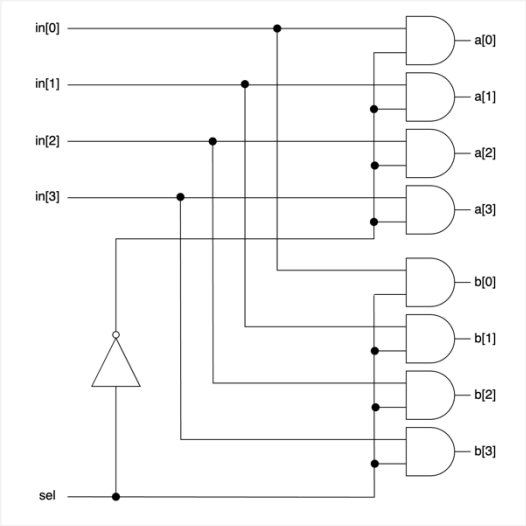
Team 01

112062144 謝佳晉　112062144 范升維

Table of Contents:

Q1. 1x4_4bit DMUX

A. 1x2_4bit DMUX

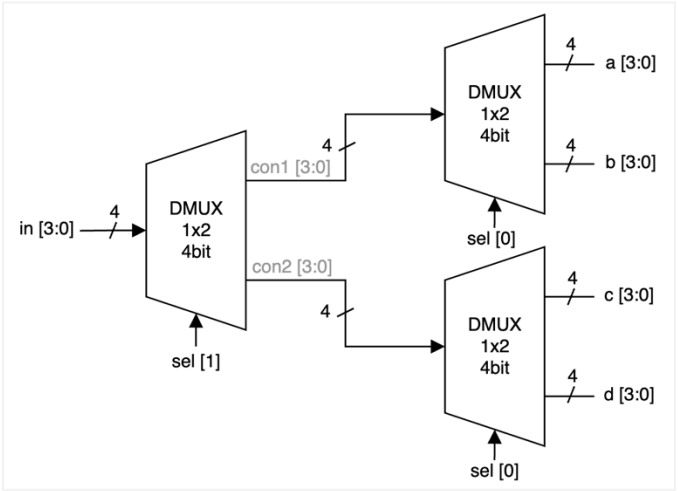


▲ Figure 1.1

- 1 input: *in[3:0]*
- 1 control signal: *sel*
- 2 outputs: *a[3:0], b[3:0]*

First, we construct module 1x2 DMUX. This module can **route the input**, *in[3:0]*, to either a or b **depending** on the **control signal**, *sel*. When *sel* is 0, the upper four AND gates have one of the input 1, so their outputs will be the same as input. Since, the lower four have one of the input 0, their outputs will all be 0. Vice versa.

B. 1x4_4bit DMUX

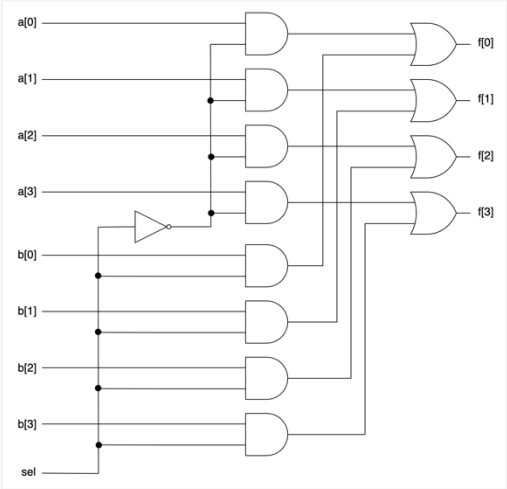


▲ Figure 1.2

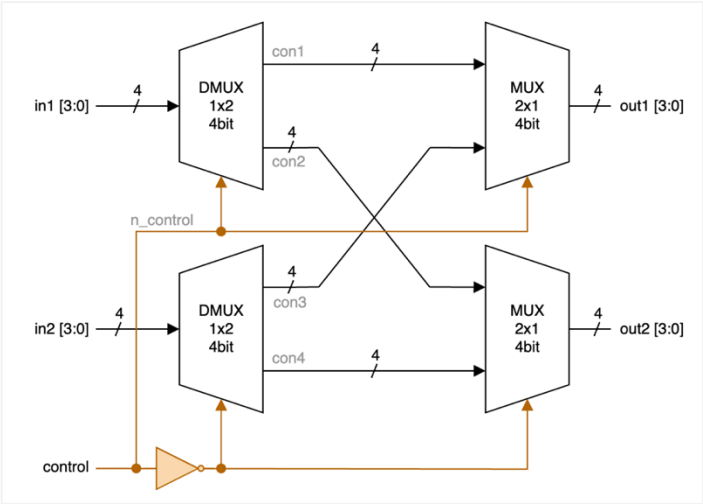
- 1 input: *in[3:0]*
- 1 control signal: *sel[1:0]*
- 4 outputs: *a[3:0], b[3:0], c[3:0], d[3:0]*

Now we have 1x2 DMUX, so we can construct module 1x4 DMUX by **combining three** 1x2 DMUX. We can use the left 1x2 DMUX with $sel[1]$ as its control signal to preliminarily route the input to a/b or c/d. Next, we use the right two 1x2 DMUX with $sel[0]$ as its control signal to route the input to a or b, and c or d. Therefore, we get one of $a/b/c/d$ to be the same as input, and other three outputs be 4'b0000.

Q2. 2x2_4bit Crossbar Switch



▲ Figure 2.1: 2x1_4bit MUX



▲ Figure 2.2: 2x2_4bit Crossbar Switch

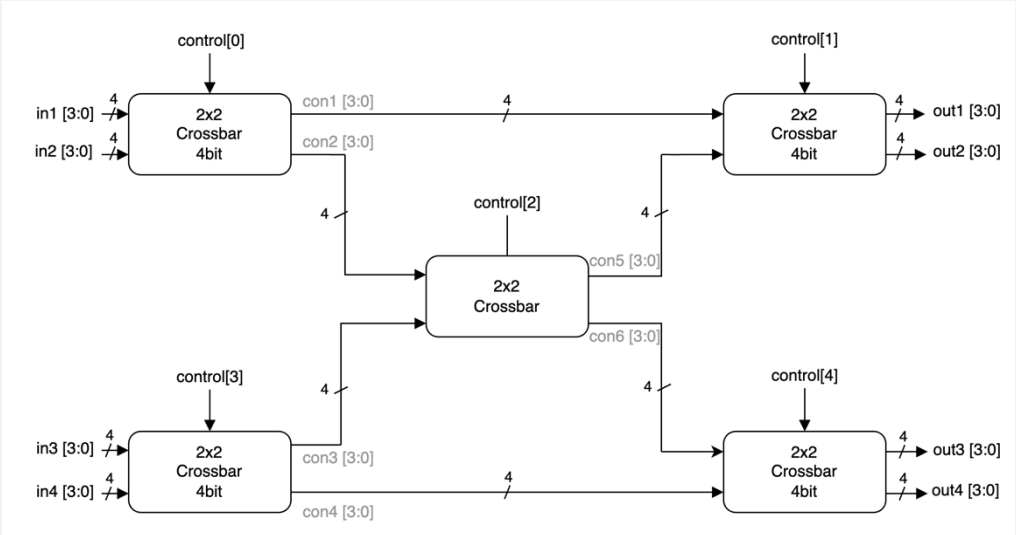
- 2 inputs: $in1[3:0]$, $in2[3:0]$
- 1 control signal: $control$
- 2 outputs: $out1[3:0]$, $out2[3:0]$

We use two 1x2 DMUX, from Q1, and two 2x1 MUX, from Lab 1 basic question, to construct a 2x2 Crossbar Switch.

When $control$ is 0, $in1 \rightarrow con1 \rightarrow out1$, and $in2 \rightarrow con4 \rightarrow out2$, like parallel.

When control is 1, $in1 \rightarrow con2 \rightarrow out2$, and $in2 \rightarrow con3 \rightarrow out1$, like cross.

Q3. 4x4_4bit Crossbar Switch



▲ Figure 3

- 4 inputs: *in1[3:0], in2[3:0], in3[3:0], in4[3:0]*
- 1 control signal: *control[4:0]*
- 4 outputs: *out1[3:0], out2[3:0], out3[3:0], out4[3:0]*

The 4x4 Crossbar Switch consists of five 2x2 Crossbar Switch. It acts like an enhanced version of 2x2 Crossbar Switch, but there has a limit. Some of the inputs are impossible to route to some specific outputs. The following are the routes that **cannot be implemented** by this 4x4 Crossbar Switch:

[(*in1*, *out3*), (*in2*, *out4*), (*in3*, *out1*), (*in4*, *out2*)],

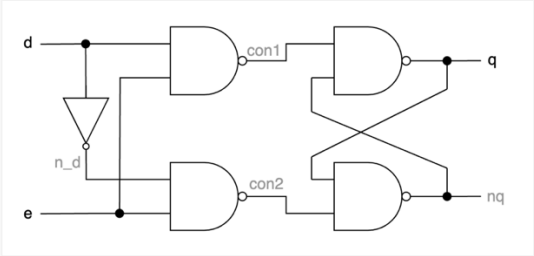
[(*in1*, *out4*), (*in2*, *out3*), (*in3*, *out1*), (*in4*, *out2*)],

[(*in1*, *out3*), (*in2*, *out4*), (*in3*, *out2*), (*in4*, *out1*)],

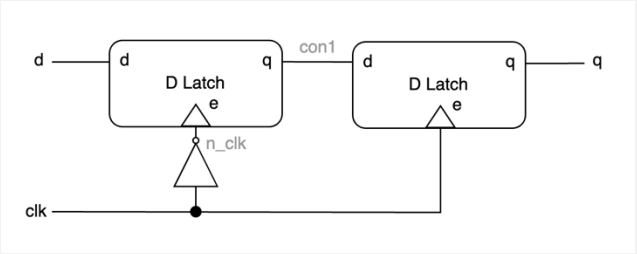
[(*in1*, *out4*), (*in2*, *out3*), (*in3*, *out2*), (*in4*, *out1*)].

The above limits exist because **only one** of the *in1*/*in2* **can be routed** to *out3*/*out4*, and at the same time, only one of the *in3*/*in4* can be routed to *out1*/*out2*. To handle the problem, we can **add an additional 2x2 Crossbar** which takes *con1* and *con4* as input.

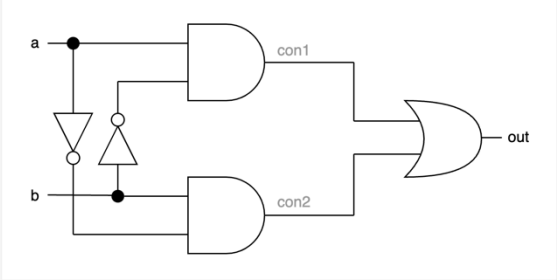
Q4. Toggle Flip-Flop



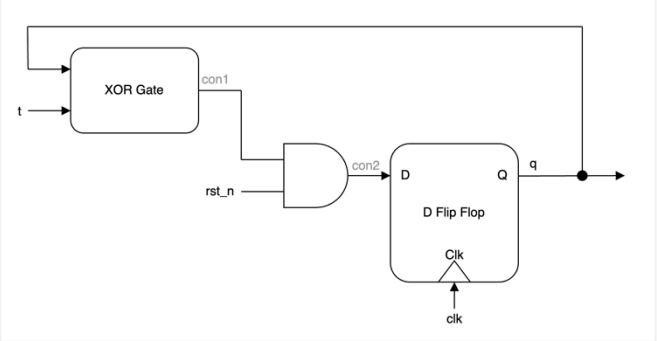
▲ Figure 4.1: D-Latch



▲ Figure 4.2: D Flip-Flop



▲ Figure 4.3: XOR Gate



▲ Figure 4.4: Toggle Flip-Flop

- 3 inputs: clk, t, rst_n
- 1 output: q

Given that we cannot utilize built-in XOR gate, Figure 4.3 shows how we construct the XOR gate ourselves. After we have the XOR gate and D Flip-Flop with D-Latch, shown in Figure 4.1 and 4.2, which we first constructed them in Lab 1 basic question 2, we can further get a Toggle Flip Flop as shown in Figure 4.4.

When rst_n is 0, and on the **positive edge**, q will be reset to 0.

When rst_n is 1, and on the positive edge:

When t is **stably 1** at that time, q will be toggled into $\sim q$.

If t is 0 or t isn't **stable** at its **setup time** and **hold time**, q will not be changed.

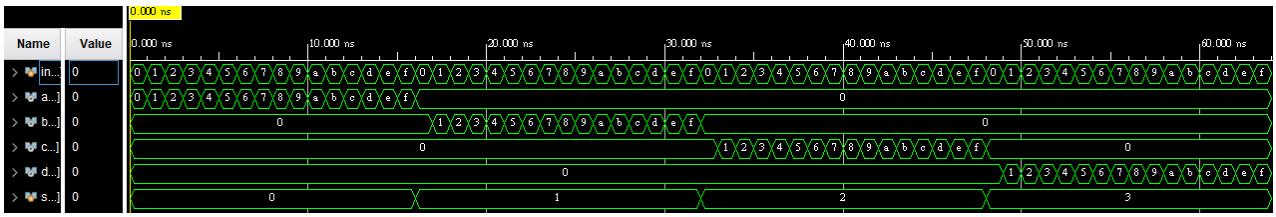
Testbenches

A. 1x4_4bit DMUX

```
module Dmux_1×4_4bit_t();
    reg  [4-1:0] in;
    wire [4-1:0] a, b, c, d;
    reg  [2-1:0] sel = 2'b0000;

    Dmux_1×4_4bit DMUX1(in, a, b, c, d, sel);

    initial begin
        repeat (2**2) begin
            in = 4'b0000;
            repeat (2**4 - 1) #1 in = in + 4'b0001;
            #1 sel = sel + 2'b01;
        end
        #1 $finish;
    end
endmodule
```



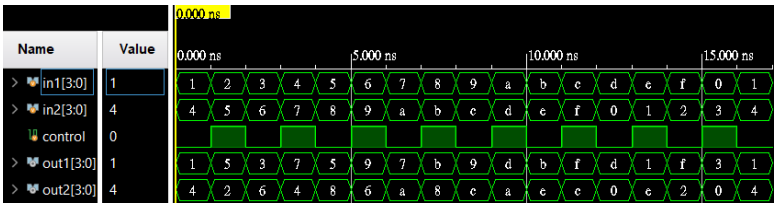
We can test the module by adding up *sel*'s value and inspect whether the outputs change correspondingly, with ever-changing input value to demonstrate its flexibility.

B. 2x2_4bit Crossbar Switch

```
module Crossbar_2×2_4bit_t;
    reg [4-1:0] in1 = 4'b0001;
    reg [4-1:0] in2 = 4'b0100;
    reg control = 1'b0;
    wire [3:0] out1, out2;

    Crossbar_2×2_4bit CB2×2_0(in1, in2, control, out1, out2);

    initial begin
        repeat (2 ** 4) begin
            #1;
            control = control + 2'b01;
            in1 = in1 + 4'b0001;
            in2 = in2 + 4'b0001;
        end
        #1 $finish;
    end
endmodule
```



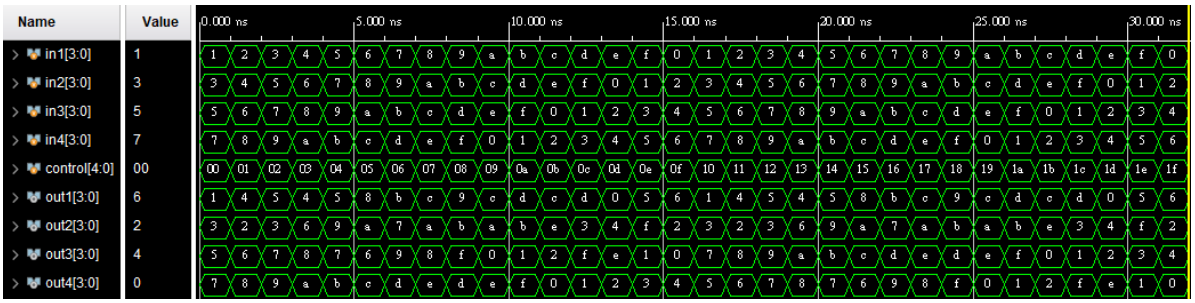
We can test this module by repeatedly changing the value of control to see if the inputs are routed to the correct output ports. Similarly, we keep changing the input values to demonstrate its flexibility.

C. 4x4_4bit Crossbar Switch

```
module Crossbar_4×4_4bit_t;
    reg [4-1:0] in1 = 4'b0001;
    reg [4-1:0] in2 = 4'b0011;
    reg [4-1:0] in3 = 4'b0101;
    reg [4-1:0] in4 = 4'b0111;
    reg [5-1:0] control = 5'b00000;
    wire [4-1:0] out1, out2, out3, out4;

    Crossbar_4×4_4bit CB4×4_0(in1, in2, in3, in4, out1, out2, out3, out4, control);

    initial begin
        $monitor("%h%h%h%h", out1, out2, out3, out4);
        repeat (2 ** 5) begin
            #1;
            control = control + 5'b00001;
            in1 = in1 + 4'b0001;
            in2 = in2 + 4'b0001;
            in3 = in3 + 4'b0001;
            in4 = in4 + 4'b0001;
        end
        #1 $finish;
    end
endmodule
```



We can test this module by changing control's value to see if the inputs have been routed to the correct output ports, just like 2x2_4bit Crossbar Switch, along with changing input value to demonstrate its flexibility. **To find the impossible routes easily**, we can set input values to be a, b, c, d and use **$monitor** command.

◎ **Difference between $display and $monitor:**

$display: It's like printf in C. It output the message once the statement is executed.

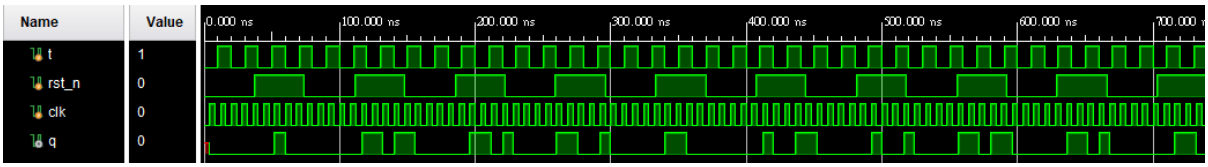
$monitor: Once being set, every time the signal changed, it will output the message.

D. Toggle Flip Flop

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

    Toggle_Flip_Flop TFF1(clk, q, t, rst_n);

    initial begin
        repeat (185) #4 clk = ~clk;
        $finish;
    end
    always #37 rst_n = ~rst_n;
    always #10 t = ~t;
endmodule
```



We can test this module by setting the proper staggered interval(4, 37, 10) so that we can simulate more cases that it may meet.

What we have learned from Lab1?

We began writing Verilog for the first time, as last semester's logic design course only involved "looking" at it, without hands-on writing experience. In this lab, we reviewed some previous concepts such as latches and flip-flops, and learned how to implement them using Verilog. We also discovered that using gate-level descriptions to write buses requires manually typing many lines of code. Additionally, we learned how to write testbenches and use Vivado simulation to verify if our modules were functioning correctly. Moreover, we used draw.io for the first time, which took some time to become familiar with in order to create neat, concise, and visually appealing circuit diagrams. Lastly, we spent considerable time learning how to create a more formal report with a table of contents. In conclusion, we learned a great deal of content through this lab experience.

Contributions

謝佳晉：

wrote Q1~Q4 Verilog
wrote testbench Q1~Q4
performed Q1~Q4 simulation on Vivado
drew diagram Q3, Q4
constructed report foundation with images and descriptions
made FPGA demonstration

范升維：

wrote Q1~Q4 Verilog
wrote testbench Q1~Q4
performed Q1~Q4 simulation on Vivado
drew diagram Q1, Q2
revised diagram Q3, Q4
organized, beautified report, added more descriptions and revised images
implemented FPGA demonstration