Hardware Design

**Lab 2 Report**

Advanced Gate-Level Verilog

**Team 01**

112062144 謝佳晉 112062144 范升維

Table of Contents:

[Basic Q1. Nand\_Implement 1](#_Toc178101178)

[Basic Q3. Difference between Half Adder and Full Adder 1](#_Toc178101179)

[Advanced Q1. 8-bit RCA 2](#_Toc178101180)

[Advanced Q2. Decode and execute 3](#_Toc178101181)

[Advanced Q3. 8-bit CLA 4](#_Toc178101182)

[Advanced Q4. 4-bit Multiplier 5](#_Toc178101183)

[Advanced Q5. Exhaustive testbench design 6](#_Toc178101184)

[Advanced Q4. 4-bit Multiplier 7](#_Toc178101185)

[Testbenches 8](#_Toc178101186)

[A. 1x4\_4bit DMUX 8](#_Toc178101187)

[B. 2x2\_4bit Crossbar Switch 8](#_Toc178101188)

[C. 4x4\_4bit Crossbar Switch 9](#_Toc178101189)

[D. Toggle Flip Flop 10](#_Toc178101190)

[What we have learned from Lab1? 10](#_Toc178101191)

[Contributions 11](#_Toc178101192)

# Basic Q1. Nand\_Implement

一張含有 圖表, 文字, 寫生, 方案 的圖片

自動產生的描述

# Basic Q3. Difference between Half Adder and Full Adder

Half Adder takes two inputs, a and b, to calculate the sum and carry of them. However, Full Adder takes three inputs, a, b, and cin, to calculate the sum and carry. To construct a multiple bits adder, we usually use a series connection of adders. Except for the first adder in series, we have to take the carry output by the adder before. Because Half Adder can only take two inputs, we have to use two Half Adders to calculate one bit of sum and carry. For this reason, we use Full Adder instead of Half Adder to connect each other in series.

# Advanced Q1. 8-bit RCA

▲ 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]*

To construct a 8-bit RCA, we use eight Full Adders connecting each other to calculate each sum and carry. The first(rightmost), second, third, …, seventh Full Adders produce carry for the next Full Adder to take as cin, and the eighth Full Adder produce the overall carry of the final answer. Each bit of sum is calculated by each Full Adder respectively.

# Advanced Q2. Decode and execute

▲ 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*)],

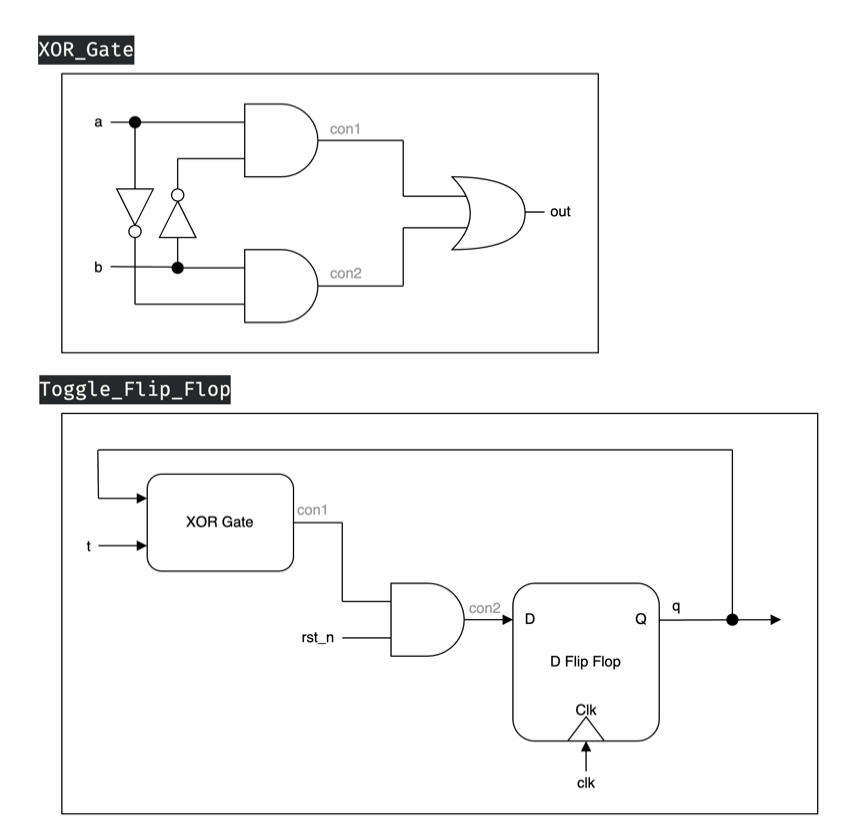
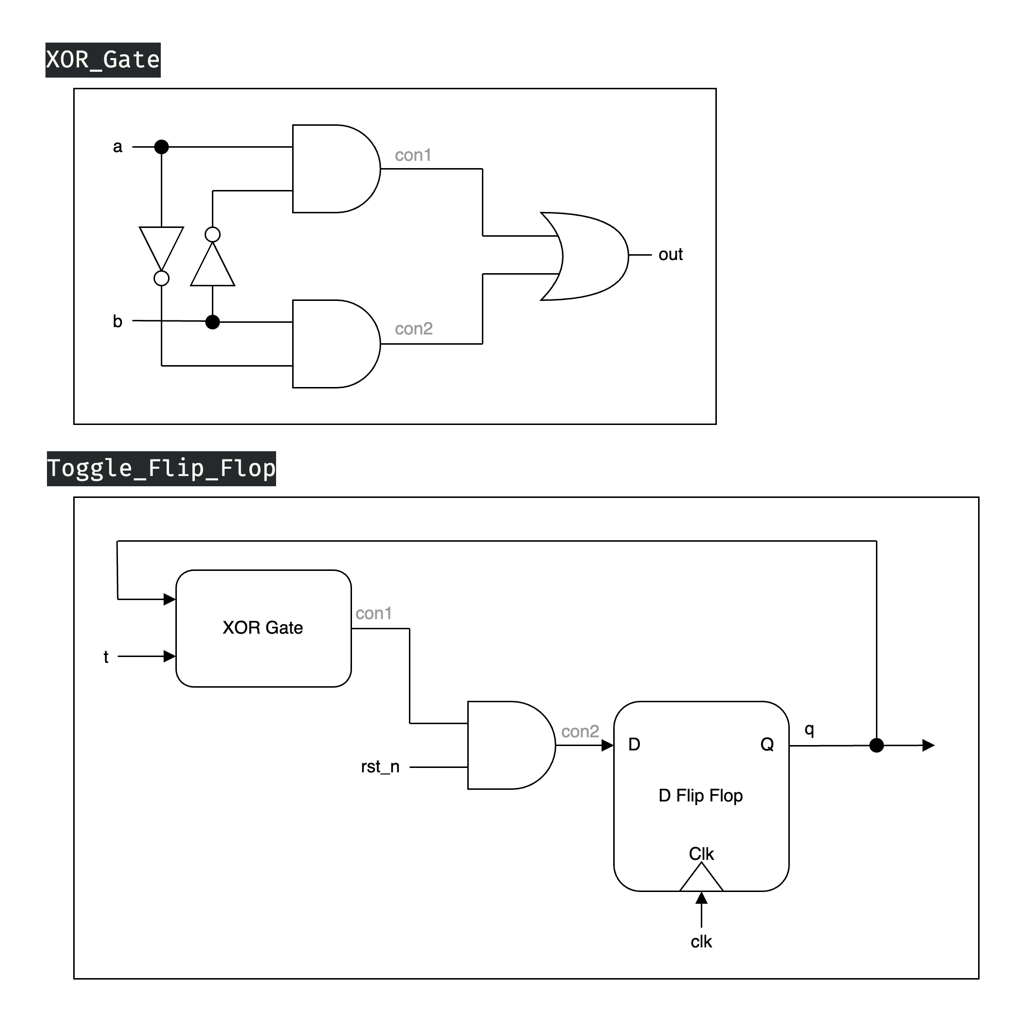
[(*in1*, *out4*), (*in2*, *out3*), (*in3*, *out1*), (*in4*, *out2*)],

[(*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.

# Advanced Q3. 8-bit CLA

 ▲ 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 *~q.*

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

# 

# 一張含有 圖表, 文字, 工程製圖, 方案 的圖片 自動產生的描述一張含有 圖表, 工程製圖, 寫生, 方案 的圖片 自動產生的描述Advanced Q4. 4-bit Multiplier

一張含有 圖表, 文字, 工程製圖, 方案 的圖片

自動產生的描述一張含有 圖表, 文字, 方案, 工程製圖 的圖片

自動產生的描述 ▲ 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 *~q.*

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

# Advanced Q5. Exhaustive testbench design

一張含有 文字, 螢幕擷取畫面, 字型, 軟體 的圖片

自動產生的描述 ▲ Figure 5.1: Our testbench design

To design a proper exhaustive testbench, we use “Two-level nested loop” to run all the number combinations we will encounter when using a 4-bit RCA. When the sum produced by the 4-bit RCA isn’t equal to the actual value, the error will be set to 1’b1 one nanosecond later and last for five nanoseconds. After the nested loop run out, we set done to 1’b1 in order to show that all of the combinations have been tested.

# Testbenches

## 一張含有 螢幕擷取畫面, 文字, 行 的圖片 自動產生的描述A. 1x4\_4bit DMUX

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

**一張含有 螢幕擷取畫面, 鍵盤, 電腦 的圖片

自動產生的描述**

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.

## 一張含有 螢幕擷取畫面, 文字 的圖片 自動產生的描述C. 4x4\_4bit Crossbar Switch

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

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