Hardware Design

**Lab 2 Report**

Advanced Gate-Level Verilog

**Team 01**

112062144 謝佳晉 112062144 范升維

Table of Contents:

[Q1. 1x4\_4bit DMUX 2](#_Toc178076429)

[A. 1x2\_4bit DMUX 2](#_Toc178076430)

[B. 1x4\_4bit DMUX 2](#_Toc178076431)

[Q2. 2x2\_4bit Crossbar Switch 3](#_Toc178076432)

[Q3. 4x4\_4bit Crossbar Switch 4](#_Toc178076433)

[Q4. Toggle Flip-Flop 5](#_Toc178076434)

[Testbenches 6](#_Toc178076435)

[A. 1x4\_4bit DMUX 6](#_Toc178076436)

[B. 2x2\_4bit Crossbar Switch 6](#_Toc178076437)

[C. 4x4\_4bit Crossbar Switch 7](#_Toc178076438)

[D. Toggle Flip Flop 8](#_Toc178076439)

[What we have learned from Lab1? 8](#_Toc178076440)

[Contributions 9](#_Toc178076441)

# 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.

# Advaced Q2. Decode and execute

**#TODO**

# Advanced Q3. 8-bit CLA

**#TODO**

# Advanced Q4. 4-bit Multiplier

**#TODO**

# 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 “Three-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. Lastly, we reset done to 1’b0 after five nanoseconds.

# Advanced Q5. FPGA─Decode and execute

In order to light the rightmost 7-segment display, we set AN[0] to 1’b0 and AN[3:1] to 1’b1. 一張含有 文字, 圖表, 螢幕擷取畫面, 行 的圖片

自動產生的描述

In the 7-segment display, we have to set the value of each segment port respectively according to the value of rd[3:0]. Below is the design of number displays.

一張含有 Rectangle, 圖表, 寫生, 行 的圖片

自動產生的描述 一張含有 圖表, 設計 的圖片

自動產生的描述

To set the value correctly, we have to draw K-map of each segment to get Boolean expressions. Take segment ‘a’ for example, we want to light up segment ‘**a**’ when rd[3:0] equals to 2, 3, 5, 6, 7, 8, 9, a, c, e, f, so we construct a K-map like this:

一張含有 數字, 文字, 行 的圖片

自動產生的描述

From this K-map, we can know the boolean expression of segment ‘**a**’ is:  
(rd[0]&(!rd[1])&(!rd[2])&(!rd[3]))|((!rd[0])&(!rd[1])&rd[2]&(!rd[3])) |(rd[0]&(!rd[1])&rd[2]&rd[3])|(rd[0]&rd[1]&(!rd[2])&rd[3])

We can get other 6 segments’ boolean expressions likewise:

**b:** (rd[0]&(!rd[1])&rd[2]&(!rd[3]))|((!rd[0]&rd[1]&rd[2])|(rd[0]&rd[1]&rd[3])| ((!rd[0]&rd[2]&rd[3])

**c:** ((!rd[0])&rd[1]&(!rd[2])&(!rd[3]))|(rd[1]&rd[2]&rd[3])|((!rd[0])&rd[2]&rd[3])

**d:** (rd[0]&(!rd[1])&(!rd[2])&(!rd[3]))|((!rd[0])&(!rd[1])&rd[2]&(!rd[3]))| ((!rd[0])&rd[1]&(!rd[2])&rd[3])|(rd[0]&rd[1]&rd[2])

**e:** (rd[0]&(!rd[3]))|(rd[0]&(!rd[1])&(!rd[2]))|((!rd[1])&rd[2]&(!rd[3]))

**f:** (rd[0]&(!rd[1])&rd[2]&rd[3])|(rd[0]&(!rd[2])&(!rd[3]))|(rd[1]&(!rd[2])&(!rd[3]))| (rd[0]&rd[1]&(!rd[3]))

**g:** (rd[0]&rd[1]&rd[2]&(!rd[3]))|((!rd[0])&(!rd[1])&rd[2]&rd[3])|((!rd[1])&(!rd[2])&(!rd[3]))

After we have all segments’ boolean expressions, we can construct the FPGA module with primitive logic gates.  
(output: light[3:0] ⭢AN[3:0], display[6:0] ⭢segment ‘g’, ‘f’, ‘e’, ‘d’, ‘c’, ‘b’, ‘a’)

# Testbenches

## A. 8-bit RCA

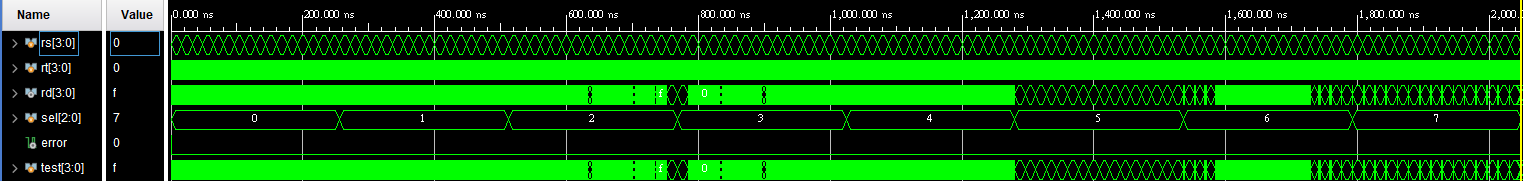
## 一張含有 文字, 螢幕擷取畫面, 軟體, 陳列 的圖片 自動產生的描述

We can test the module by using three level nested loop like advanced Q5 to run all the number combinations and see whether the output is correct.

## B. Decode and execute

一張含有 文字, 螢幕擷取畫面, 電腦, 設計 的圖片

自動產生的描述一張含有 文字, 螢幕擷取畫面, 字型 的圖片

自動產生的描述

Steps:

Initialize⭢SUB⭢reset⭢ADD⭢reset⭢Bitwise OR⭢reset⭢Bitwise AND⭢reset⭢Right Shift⭢reset⭢Left Shift⭢reset⭢Compare LT⭢reset⭢Compare EQ⭢finish

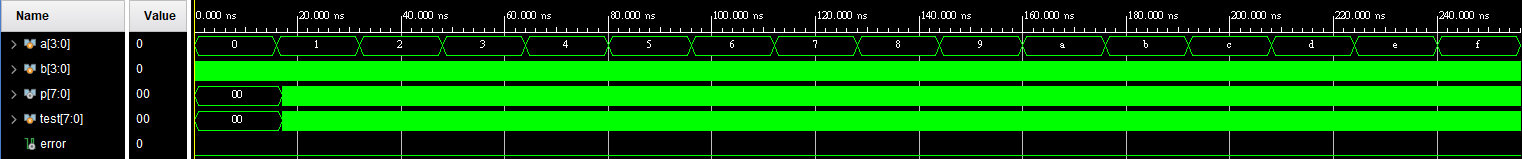
## C. 8-bit CLA

**#TODO**

## D. 4-bit Multiplier

一張含有 文字, 螢幕擷取畫面, 軟體, 字型 的圖片

自動產生的描述



We can test the module by using two level nested loop to run all the number combinations and use signal ‘error’ to detect whether the output is correct.

# What we have learned from Lab2?

**#TODO**

# 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