Digital Laboratory Lab07 Report

(Experiment Record Template)

**FPGA Experiment-701: Digital clock**

**Create a digital clock which is satisfied the following requirements:**

1. Modeling style: behavior
2. Input: FPGA built-in CMOS oscillator (i.e., pin E3)
3. Output: **8-digit** 7 segment display
4. Here are the guidelines for the marquee:
   1. The marquee displays digital clock content in the format of YYYY-MM-DD-HH-MM-SS-blank.
   2. The marquee can move in two directions: MSB to LSB or LSB to MSB.
   3. The clock has two refresh rates: high frequency to check for hour changes and low frequency to check for second/minute changes.
   4. There are two stop functions: one for stopping the movement of the marquee and another for stopping the clock from counting."

sample-701 <https://youtu.be/yNNW6fVzuVA>

Experiment Data

RTL Schematic

Simulation results

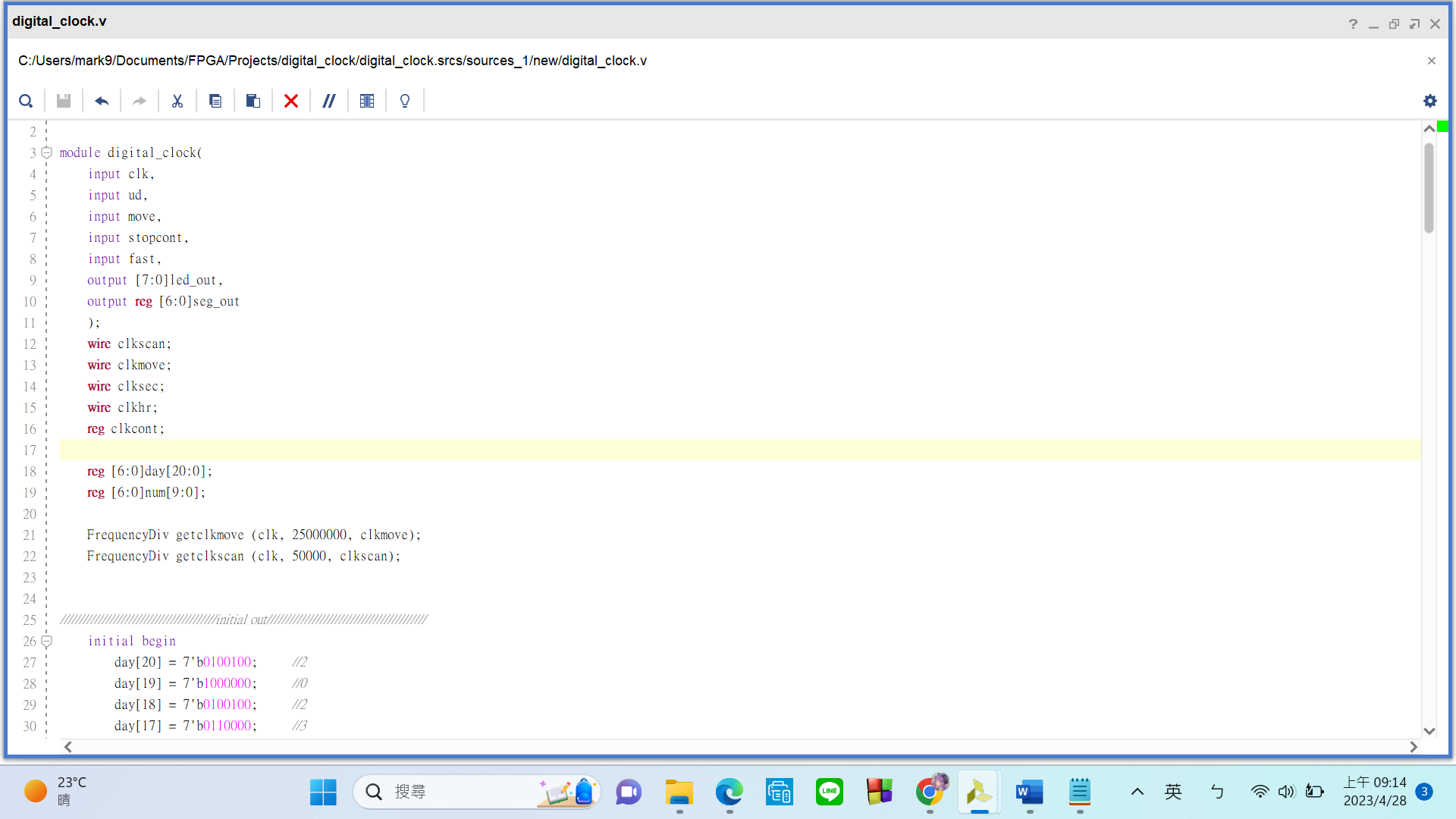
(Design your own testbench)

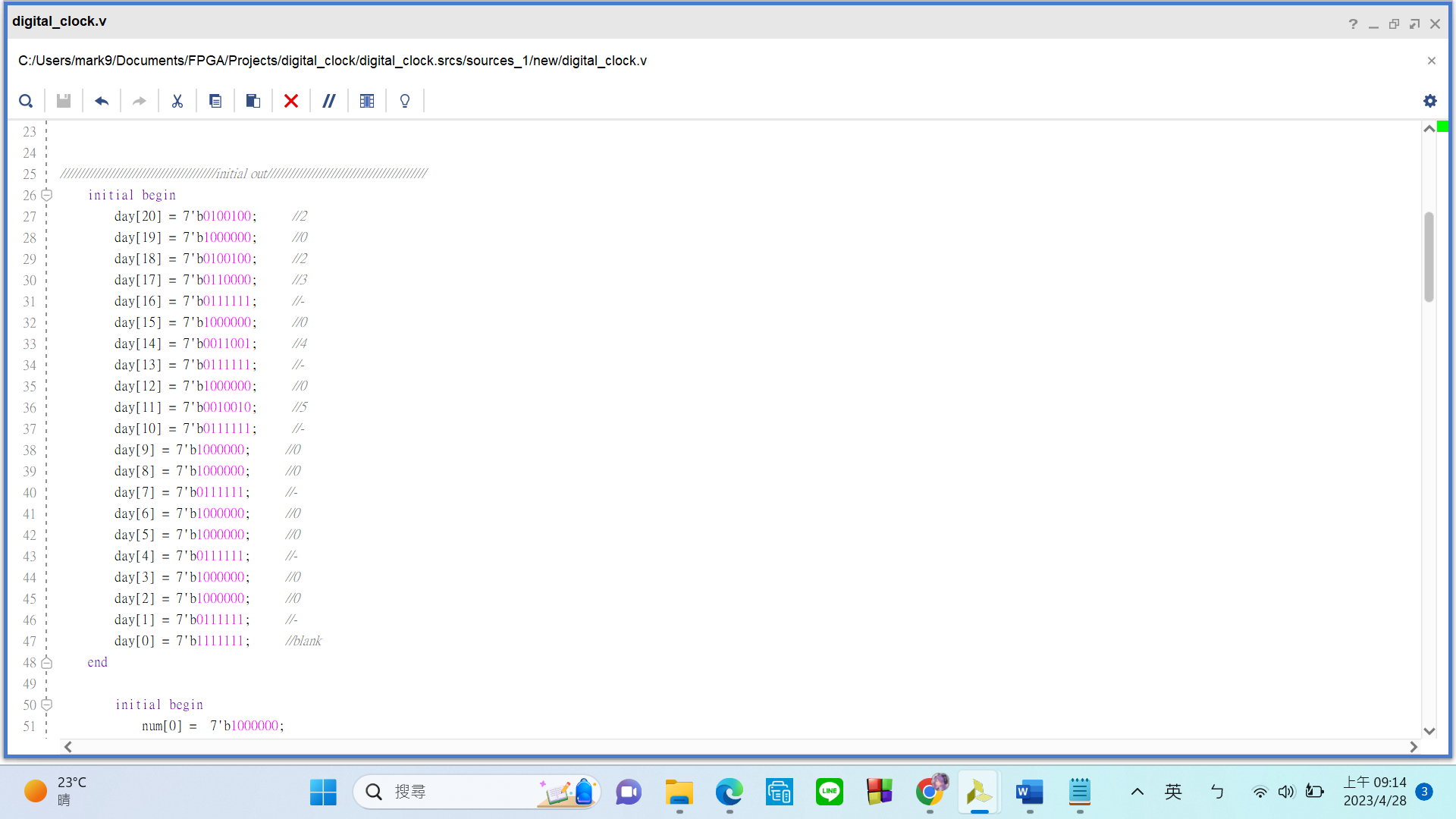
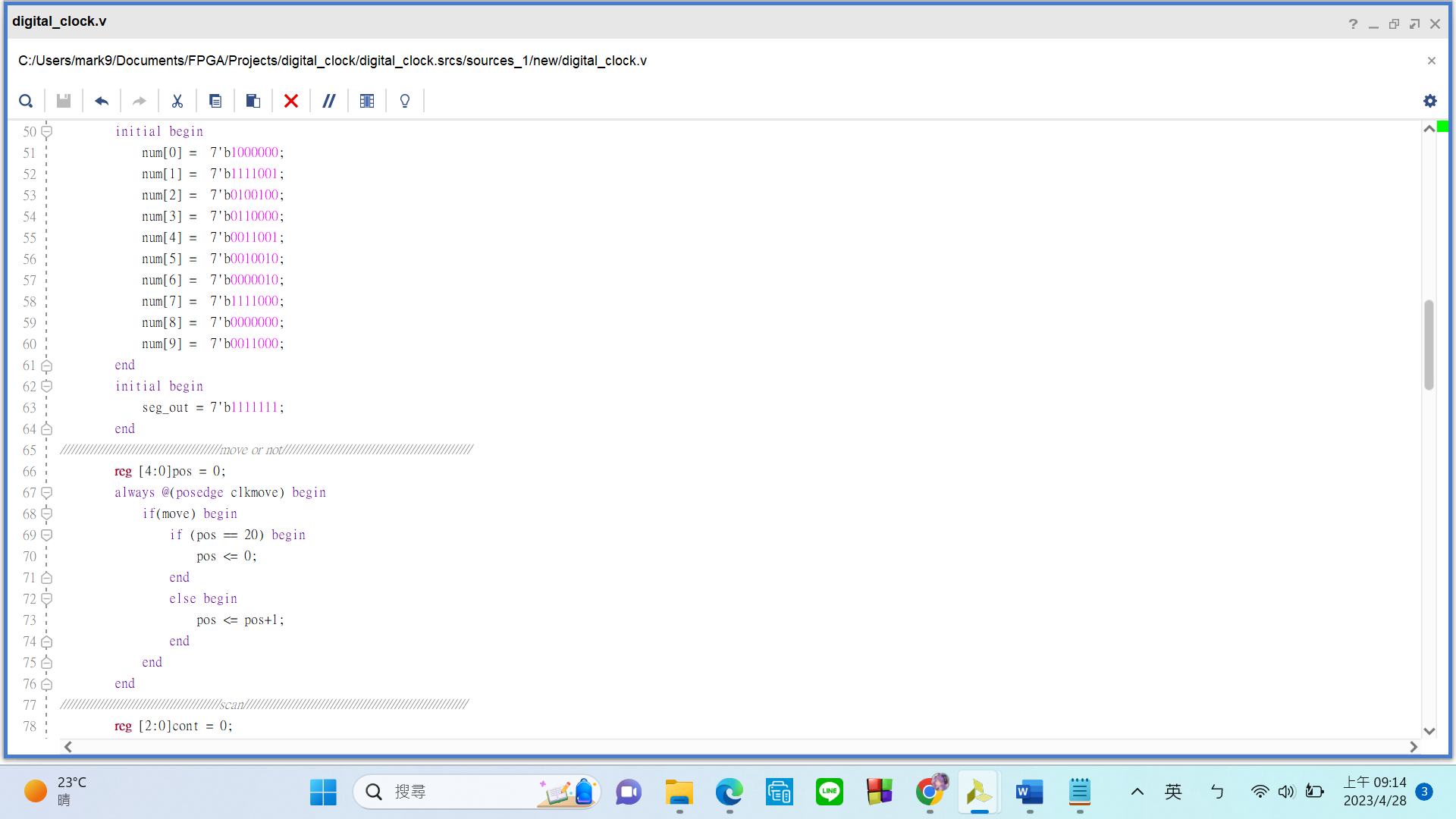
Practical results on the FPGA board

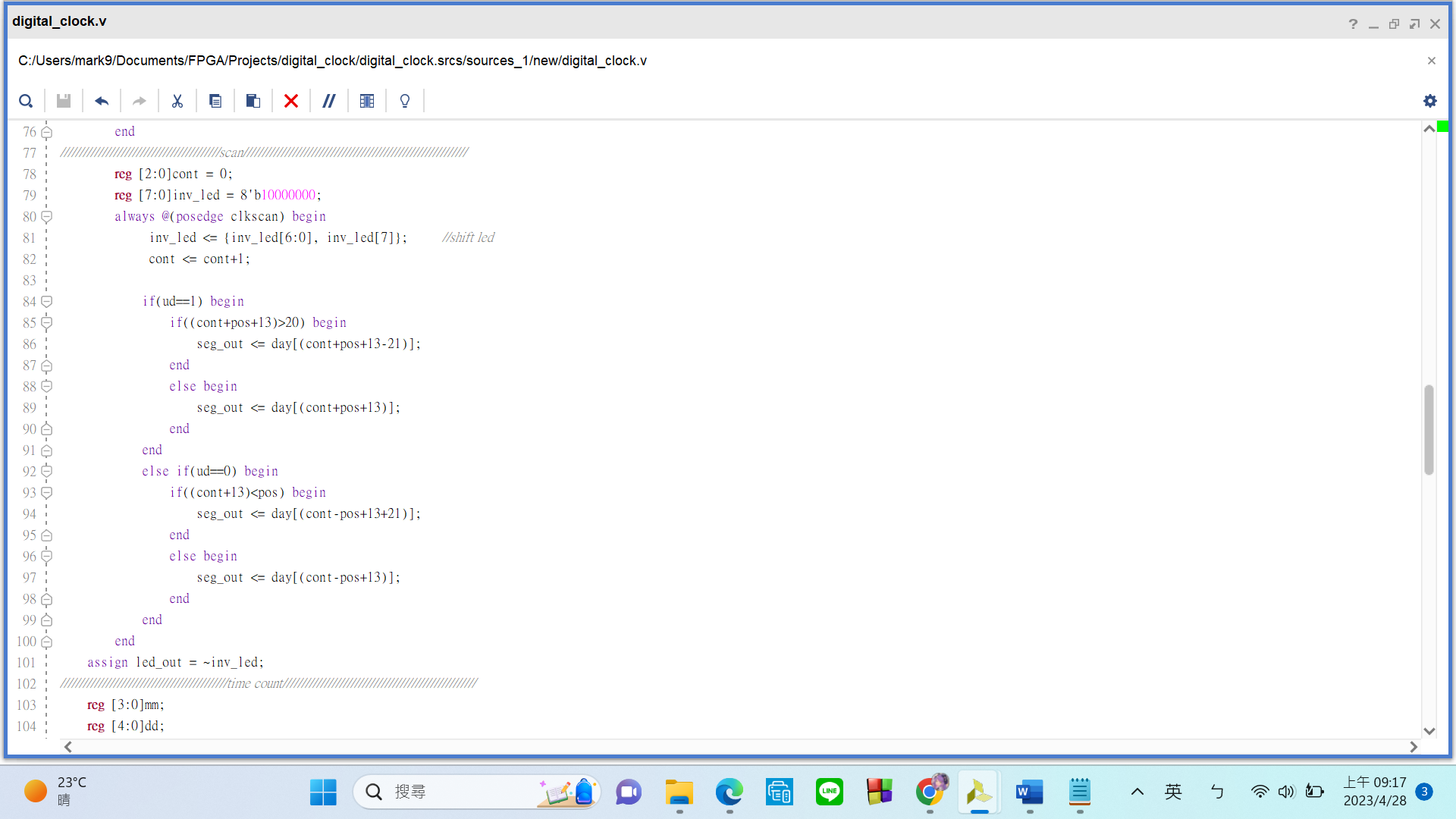
static record (photo, screen capture) or dynamic record (video link)

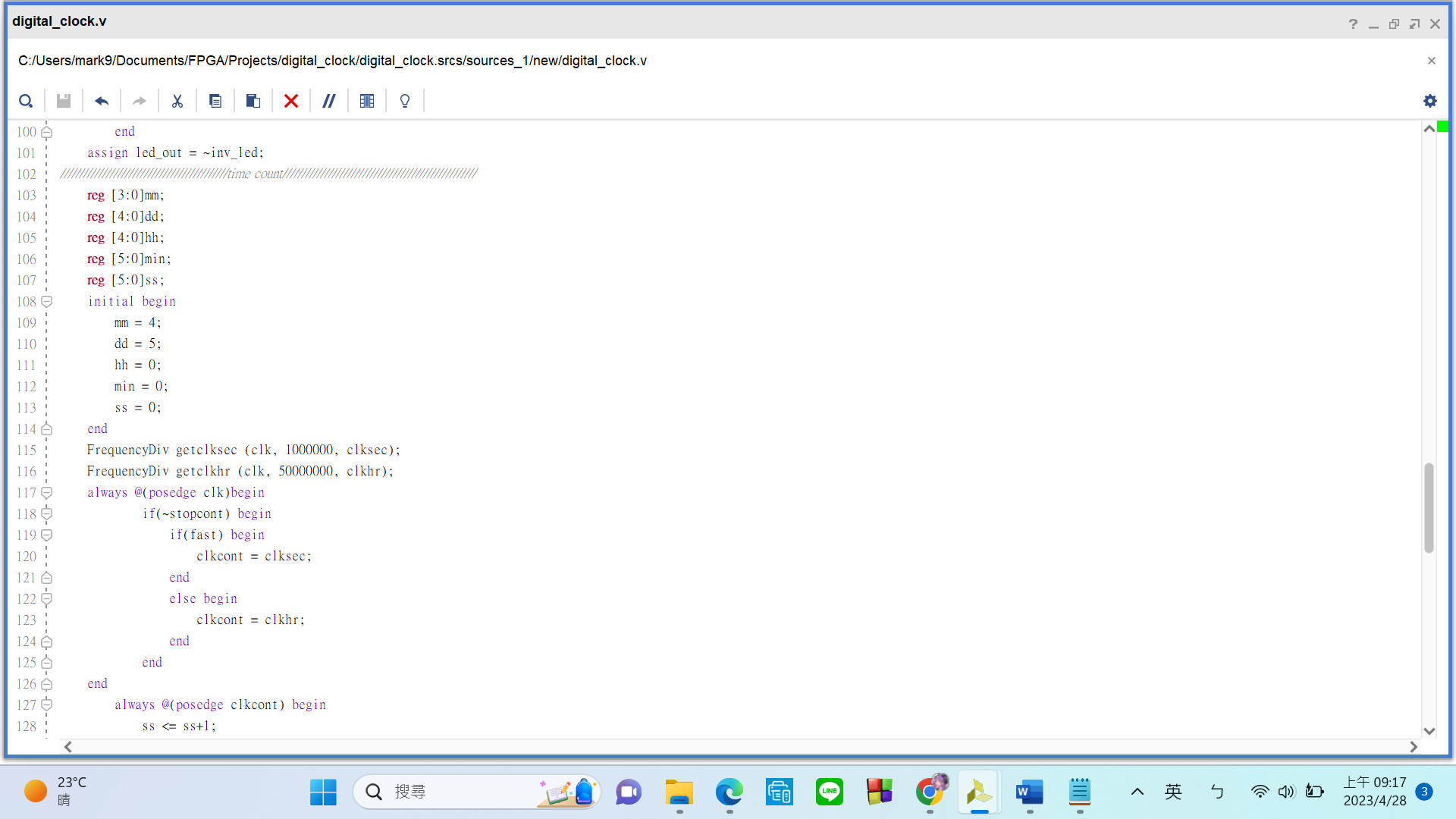
<https://youtu.be/N2RN6qhfMEM>

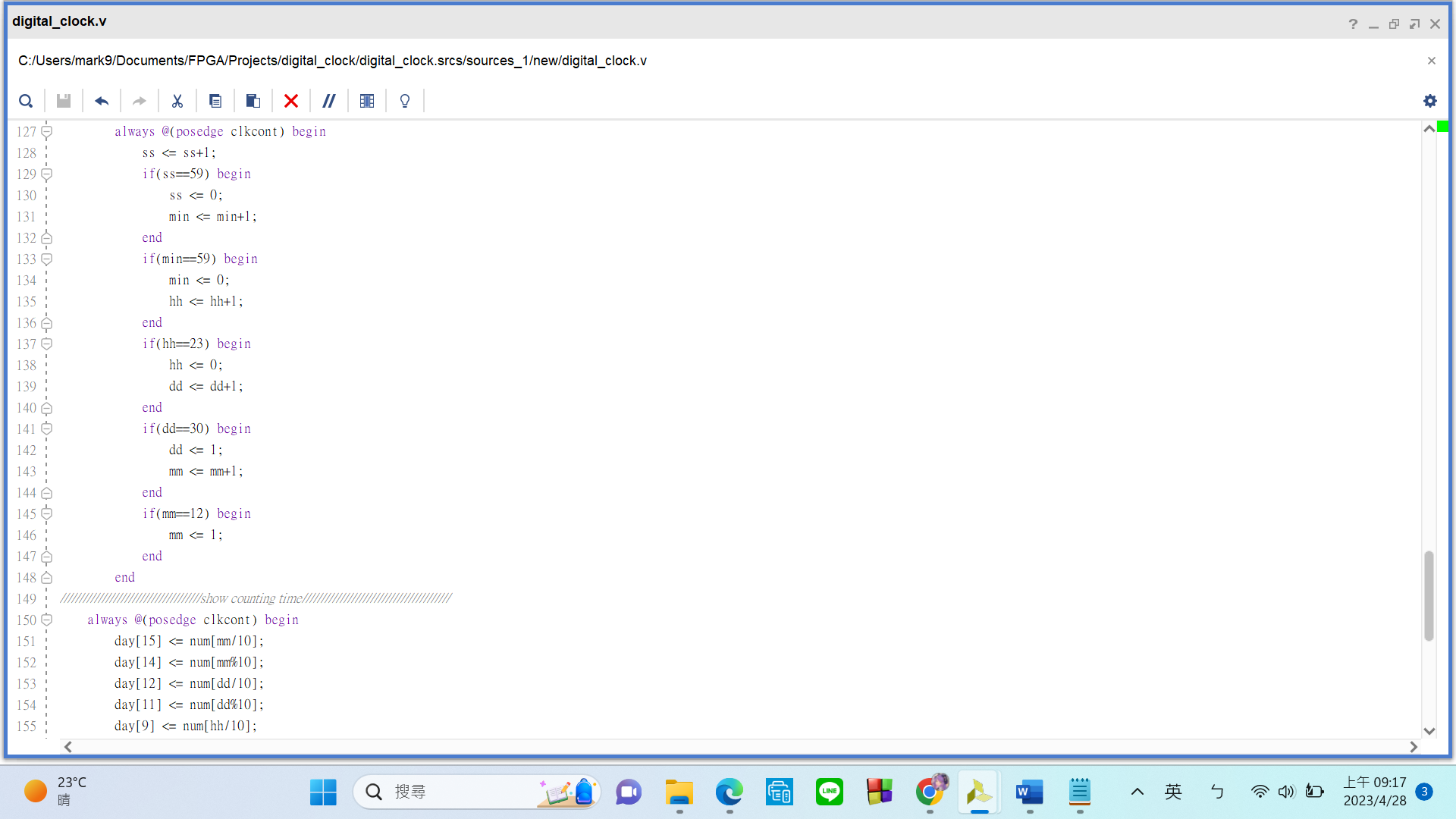
OPTIONAL: design process

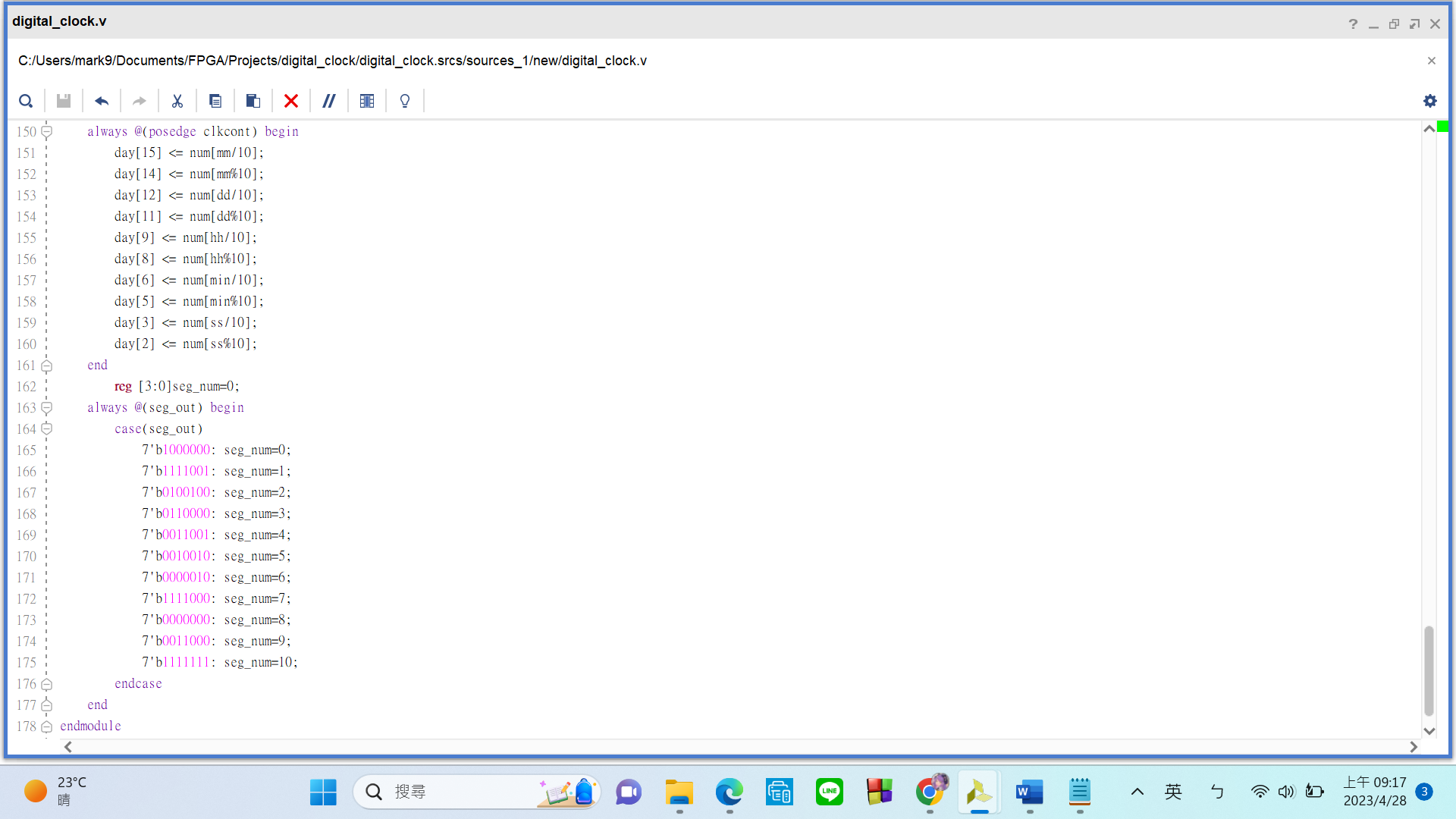






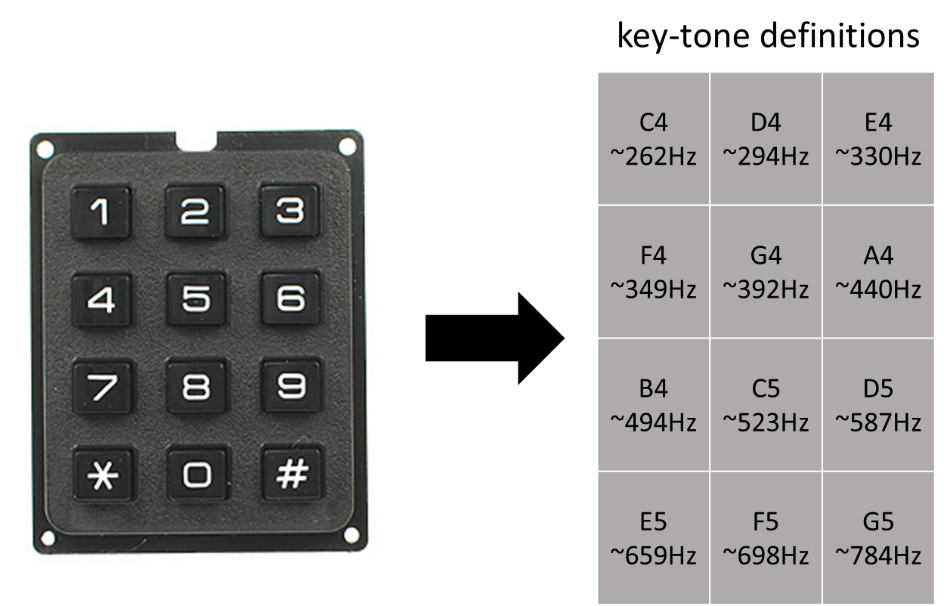




**FPGA Experiment-702: Sound Keyboard**

**Create a sound keyboard which is satisfied the following requirements:**

1. Modeling style: behavior
2. Input: keypad (7-pin or 8-pin)
3. Output: speaker
4. Press various keys to produce different tones. ([ref](https://en.wikipedia.org/wiki/Piano_key_frequencies).)

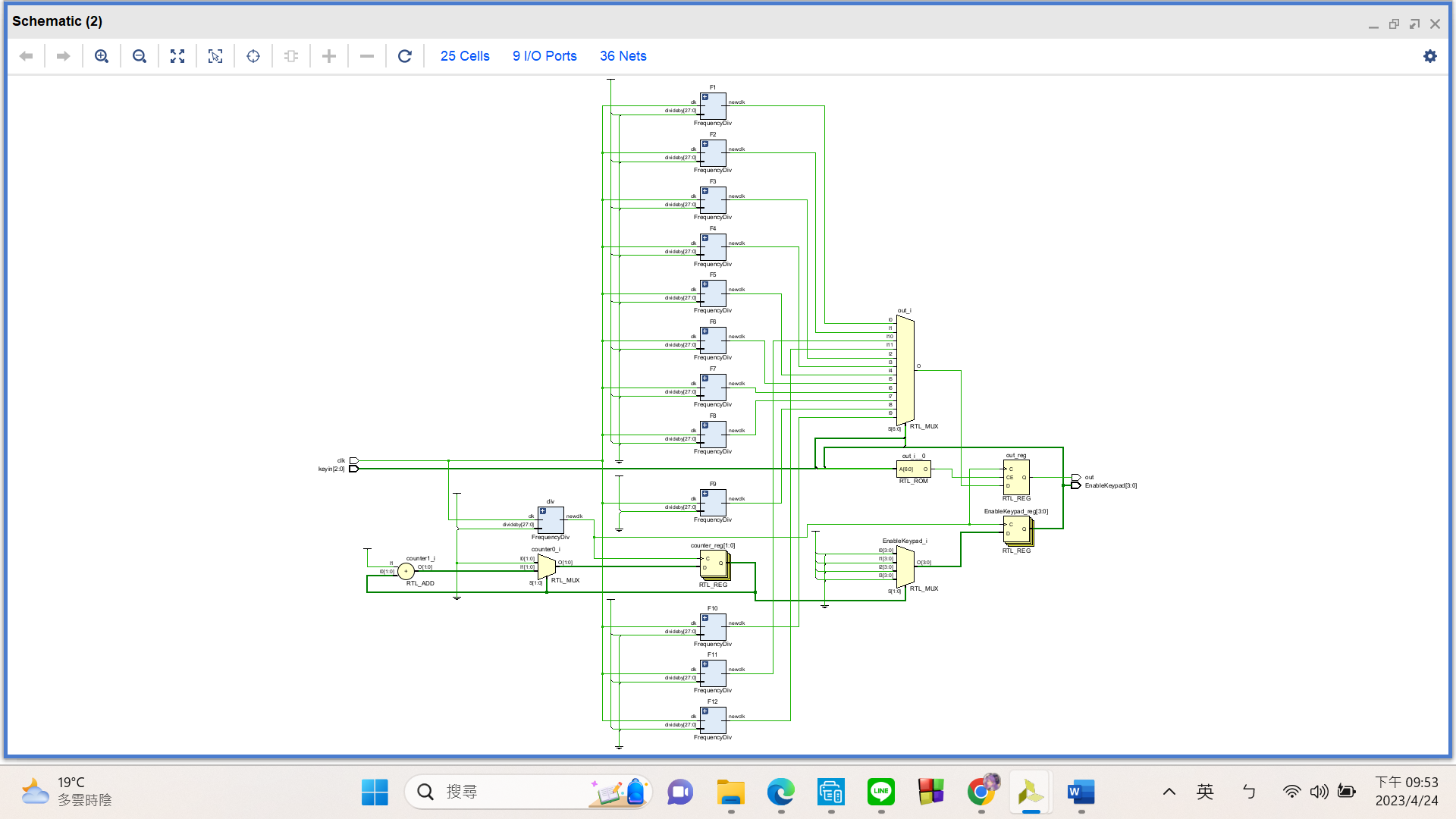


Note: **Press** and **click** are two different actions that can be performed on a computer mouse or a touchpad. Press means to push down on something with force, while click means to make a short and sharp sound by pressing and releasing a button.

sample-702 <https://youtu.be/uaSHE_uVfeQ>

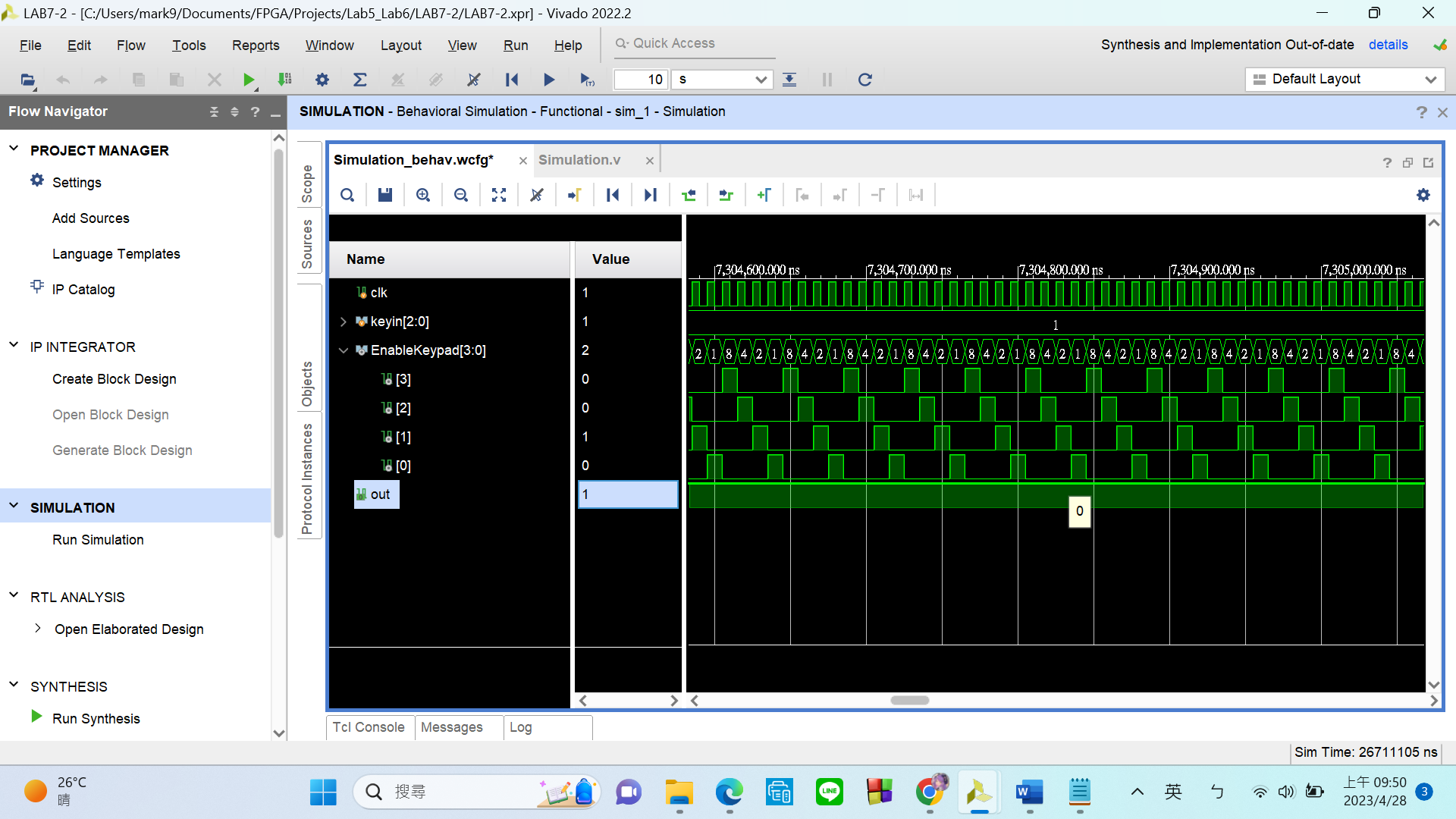
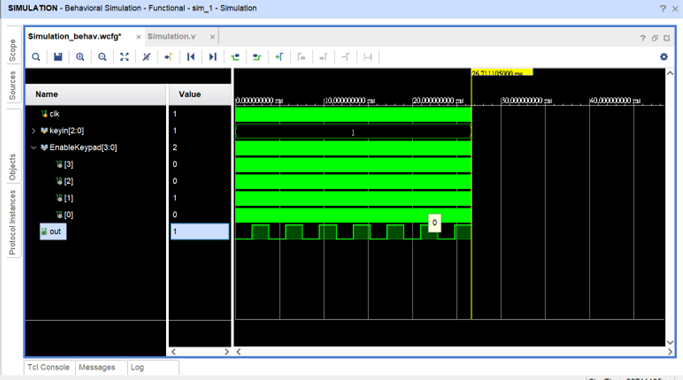
Experiment Data

RTL Schematic



Simulation results

(Design your own testbench)

Practical results on the FPGA board

static record (photo, screen capture) or dynamic record (video link)

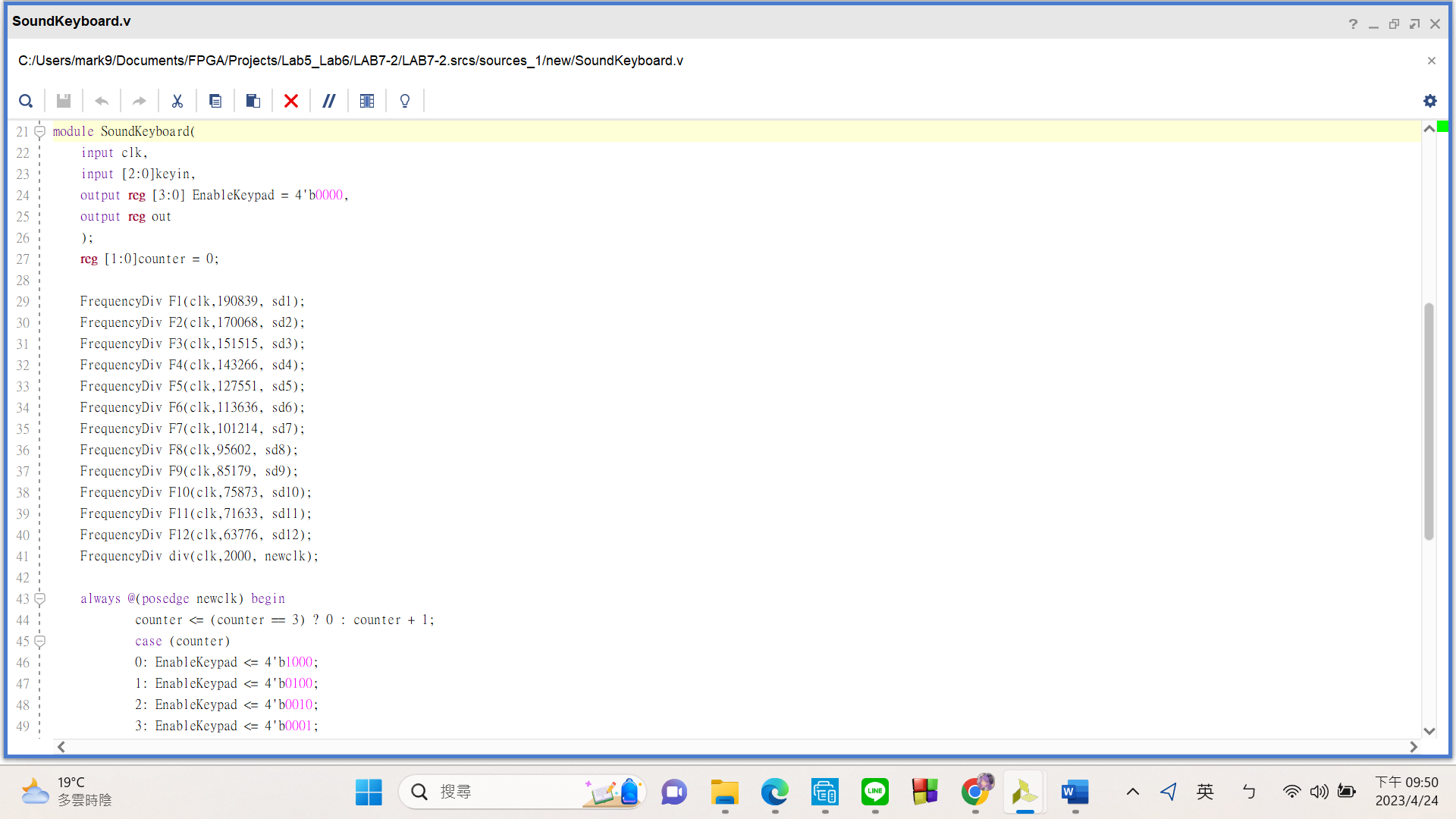
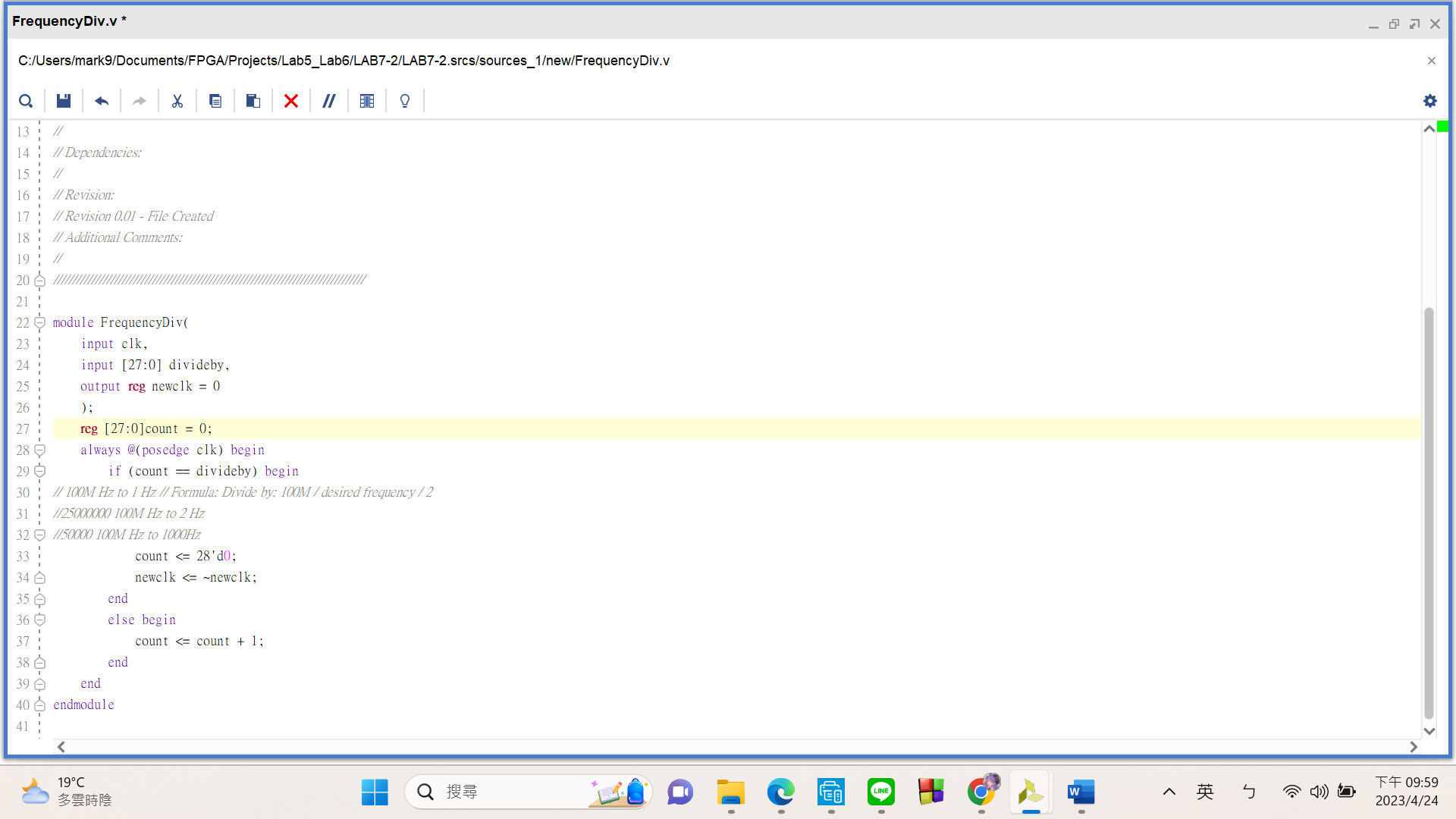
<https://youtu.be/PERtqM5YTRg>

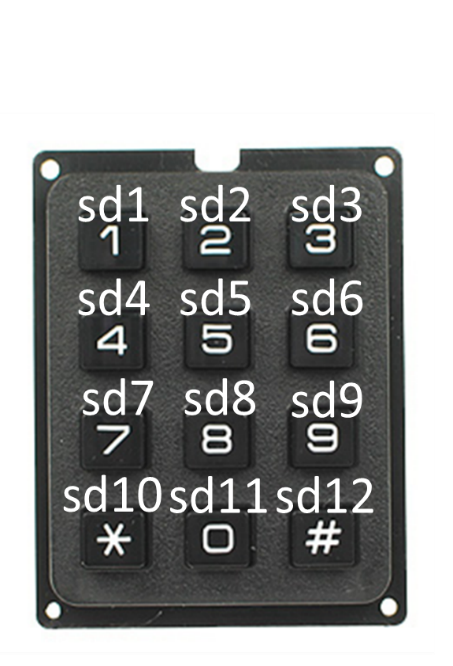
OPTIONAL: design process

**Let’s look at my code in 2 parts:**

1. **Get Frequency:**

I use my **“FrequencyDiv” module to get the desired frequency** of the 12 sounds we want the speaker to produce. FrequencyDiv takes 2 inputs: the first is the 100MHz clk from E3, the second number is obtained from the **formula: 100M / ( 2 \* desired frequency)**. This formula gives us **the number that the counter inside FrequencyDiv has to count to before toggling its divided clk output**. The calculations have already been done by myself.



The outputs of these twelve FrequencyDiv modules (sd1 to sd2) corresponds to the keypad as **shown in the following picture:**

1. **Output to Speaker**



The**first always block** simply sends enable signals through each row at a high frequency, allowing each key on the keypad to **activate**.

In the **second always block**, I used these curly brackets to **group together the rows register and columns register of the keypad{EnableKeypad(ROWS), keyin(COLUMNS)}**. Please refer to the following picture for

clarification. Once I group the regs together, I can use a **case statement** to determine which key was pressed. **For example, “1” would be 7’b1000100, “2” would be 7’b1000010, “4” would be 7’b0100100 etc.**

In the case statement, **whenever “1” is pressed I would set output reg “out” to the divided clk signal “sd1”. If “2” gets pressed, output “sd2”, and so on. (\* is “sd10”, 0 is “sd11”, # is “sd12”)**

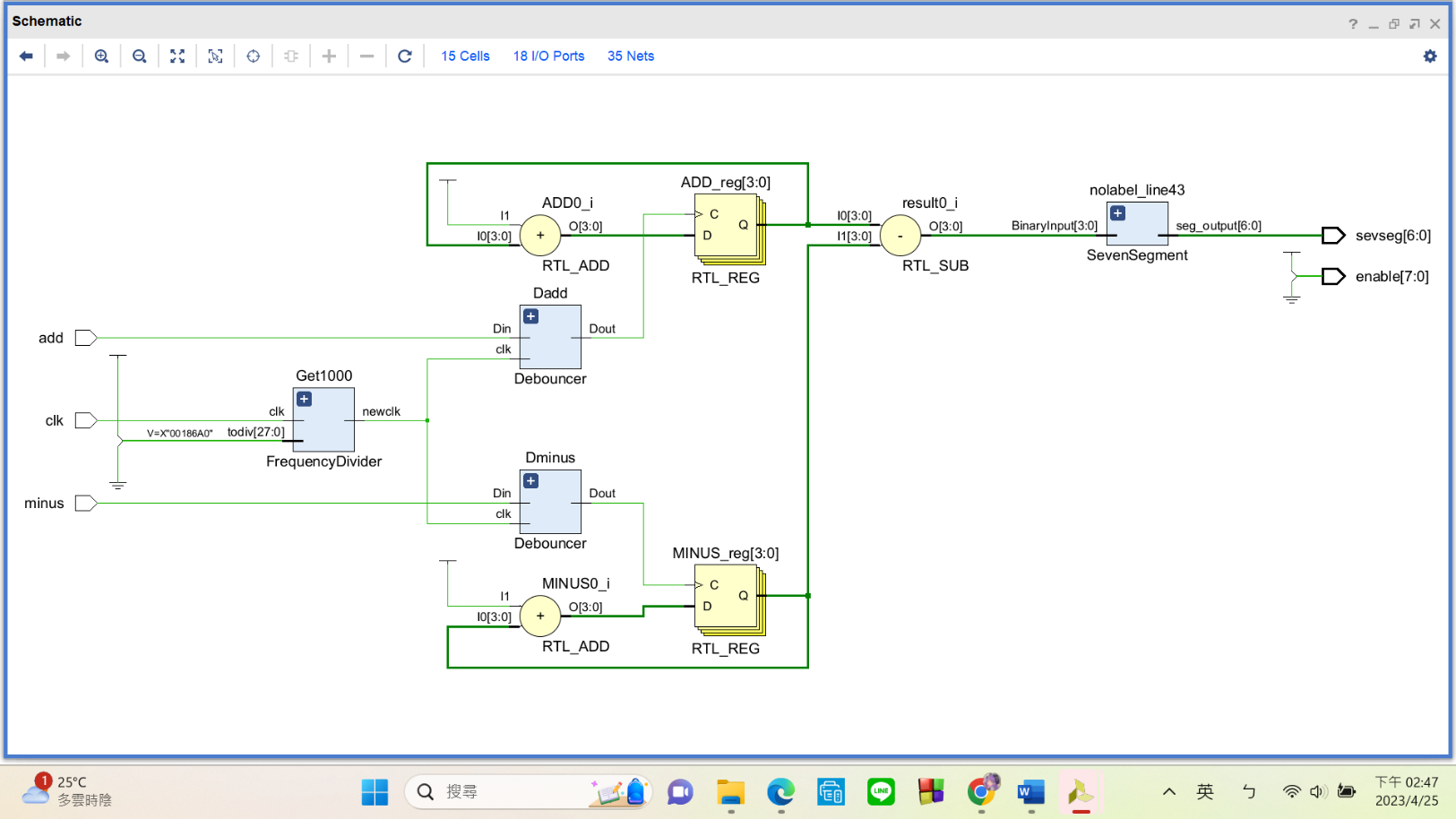
**FPGA Experiment-703: Manual Counter**

**Create a manual counter which is satisfied the following requirements:**

1. Modeling style: behavior
2. Input: button
3. Output: **exactly one** 7 segment display = 1-digit number
4. The display value starts from 0.
5. The display value ranges from 0 to 15 in a 7-segment display format.
6. To increase the current value by 1, click the 'up' button once.
7. To decrease the current value by 1, click the 'down' button once.
8. You need to solve the problem of button bounce, which occurs when a single click is read as multiple clicks."

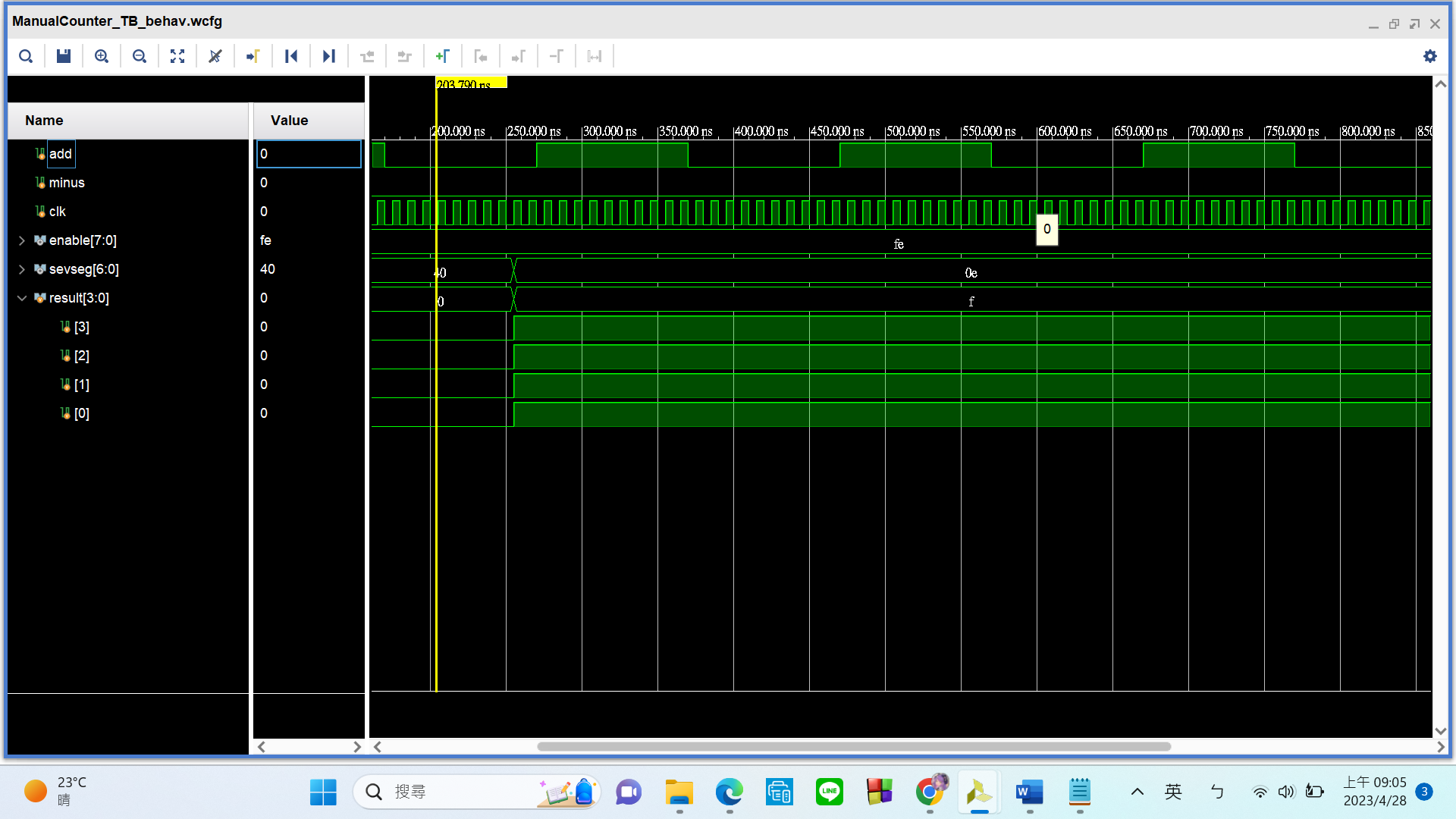
Experiment Data

RTL Schematic



Simulation results

(Design your own testbench)



Practical results on the FPGA board

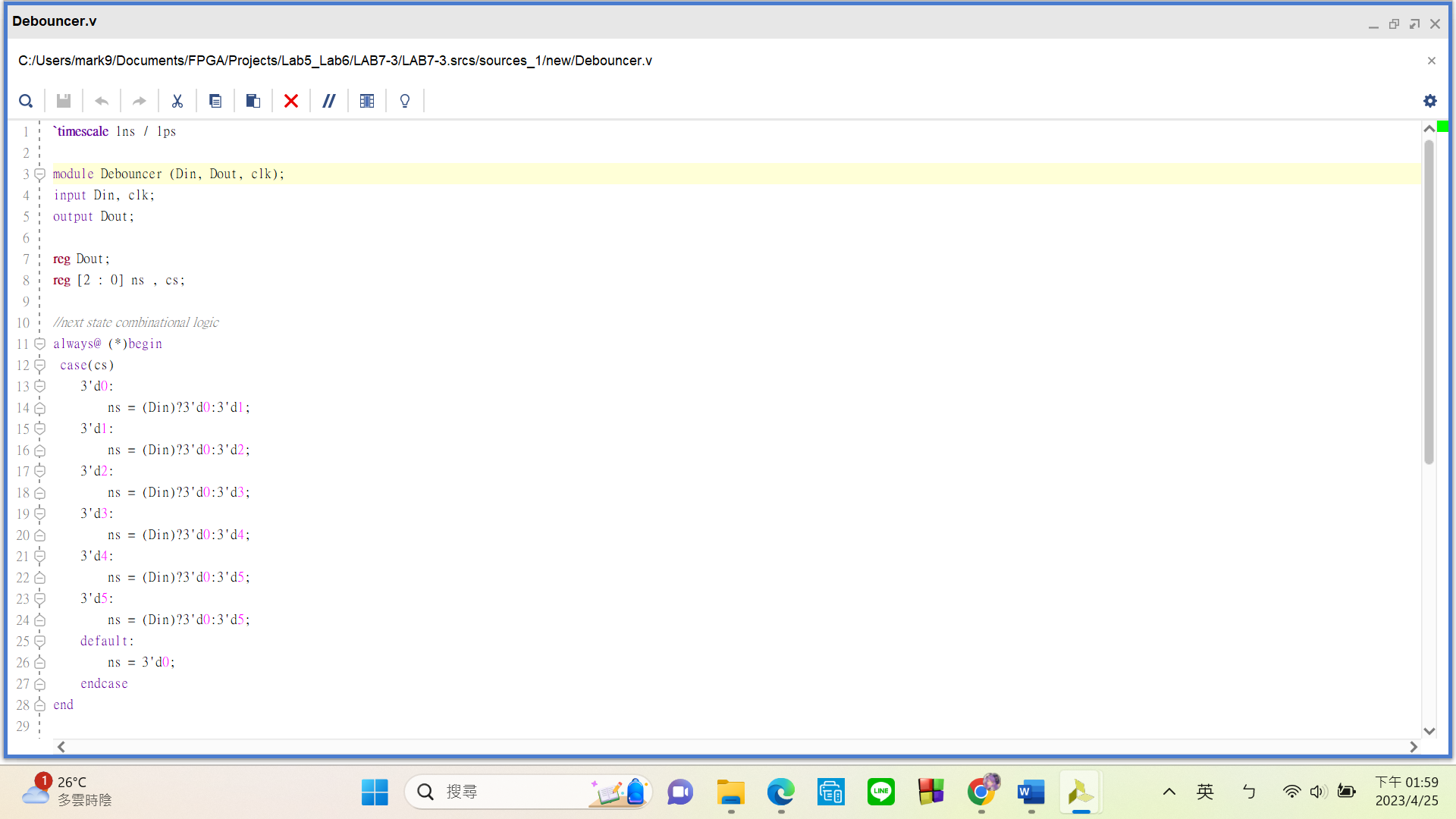
static record (photo, screen capture) or dynamic record (video link)

<https://youtu.be/c_miq4v_Y3k>

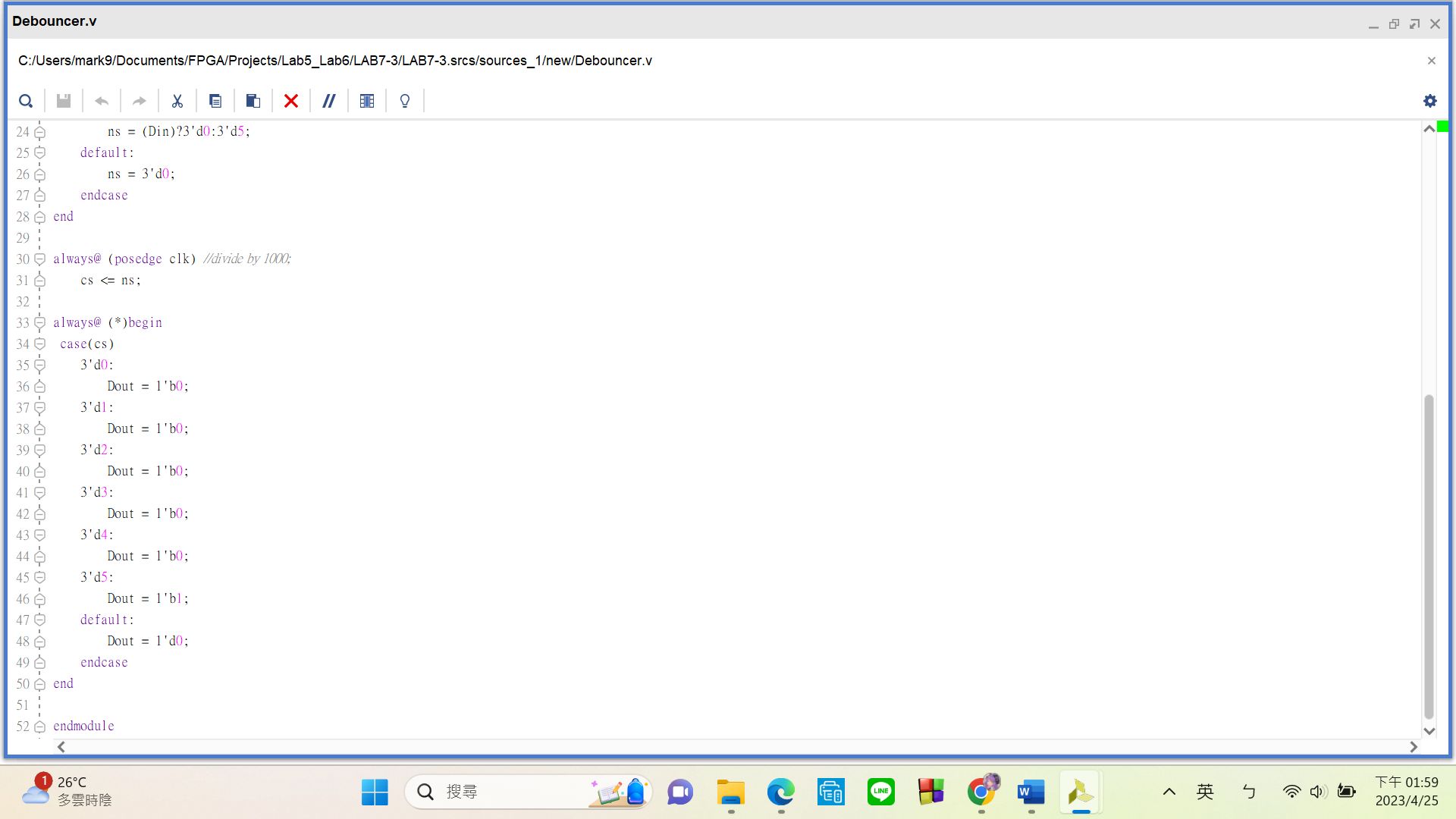
OPTIONAL: design process

**Let’s look at my code in 3 parts.**

1. **Deboucer Part 1**

In my debouncer module, I have **2 inputs**(Din, clk) and **1 output**(Dout). **Din will take the original signal from the button, clk** will take a **1000Hz clk signal**, **Dout** sends out the **debounced signal**.

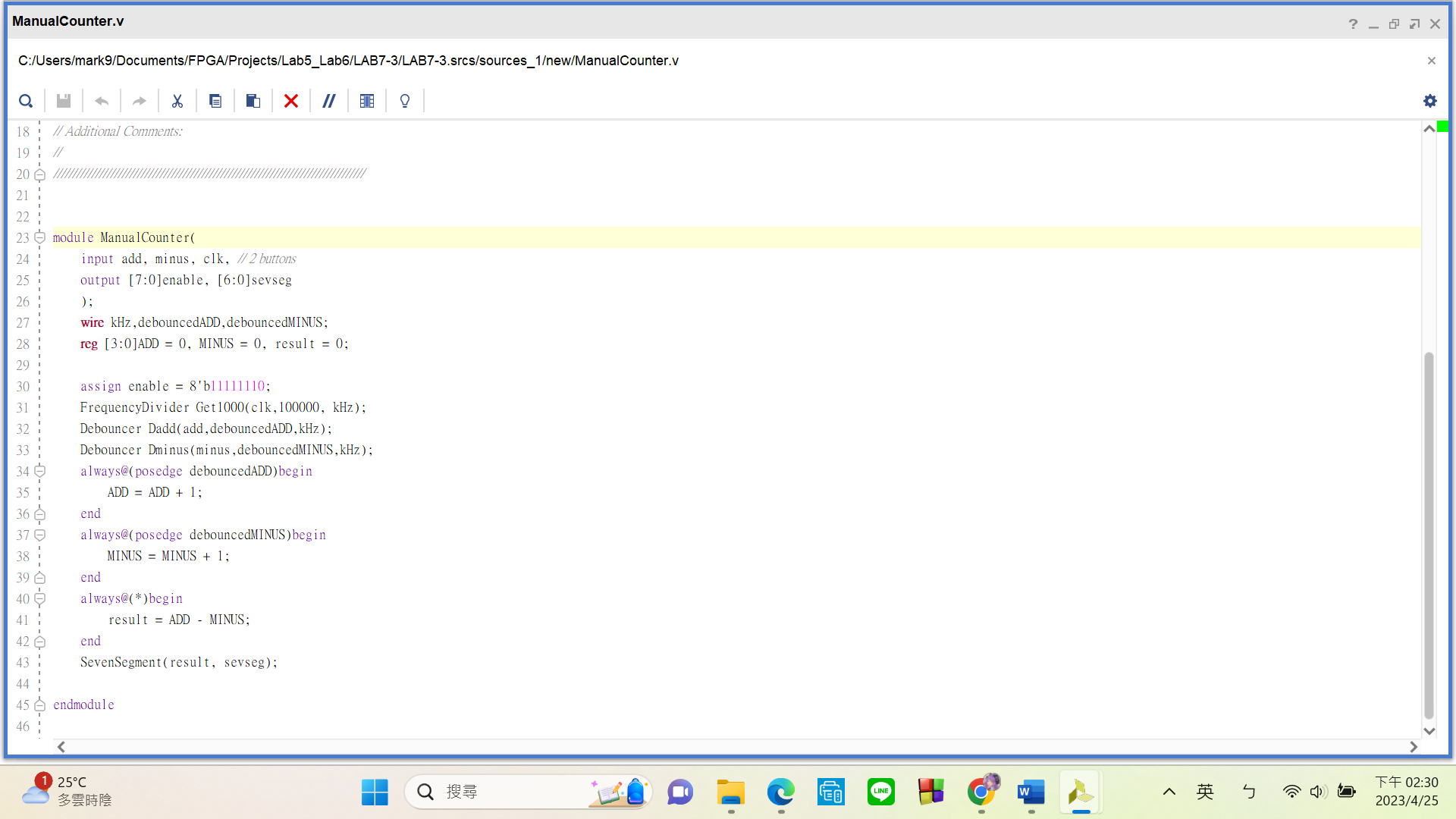
**Debouncing checks if the signal is constant for an extended period of time.** I achieved this by using two regs, cs(current state) and ns(next state) to create a **state machine**. Simply put, my code is going to **check if “Din” is constantly 1. By asking “Are you sure you are still 1?” with 6 ternary statements doing the same thing.** In the always block “ns = Din? 0 : cs + 1” is asked 6 times every time cs updates (\* means that the always block gets triggered whenever a variable inside of it gets updated). The reason behind using a 1000Hz clk this is that if the **clock is too fast(100MHz)**, it would **get through these 6 states too soon to know if the signal is actually stable**. If the **clock is too slow**(1Hz, 2Hz), we would have to **press the button longer to see the number increase because the states are getting checked too slowly**.

1. **Debouncer Part 2**

In this second part, the **first always block** here **updates** the current state to the **next state** whenever the 1000Hz clk hits a positive edge.

The second always block **sets “Dout” to 0 for State 0 to 4**. It only **sends out 1 when we hit State 5.**

1. **Top Module**

.

After finishing the debouncer works, the rest is fairly simple. I have **three submodules** in this top module, “Debouncer”, “FrequencyDiv” and “SevenSegment”. Here’s their purpose:

Debouncer: **debounces** the signal from the buttons

FrequencyDiv: simple frequency divider

SevenSegment: **converts** binary numbers into what we see on **seven segment displays**

I did encounter **one problem** while working on this part. You can easily tell that my first always block is used to add 1 whenever the button is pressed, and that the second always block is used to minus 1. I originally put my “result” reg (representing the current number) in both of these always blocks. However, I got the **“reg has multiple drivers” error**. I looked it up and found that **having the same reg in two always block is illegal** because both blocks are trying to drive the same reg at the same time. This will make its value ambiguous and unable to be determined. I **fixed this by assigning a reg ADD that is responsible only for adding and a reg MINUS that is responsible only for deduction. “result” is the difference between these two regs.**