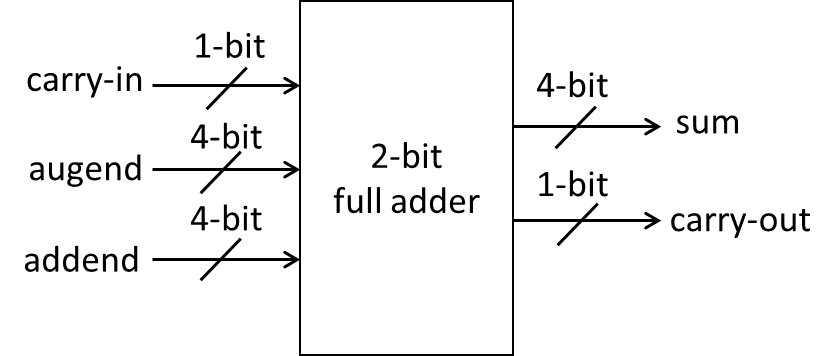
Digital Laboratory Lab05 Report

(Experiment Record Template)

**FPGA Experiment-501: Full Adder**

**Design a 4-bit full adder which is satisfied the following requirements:**

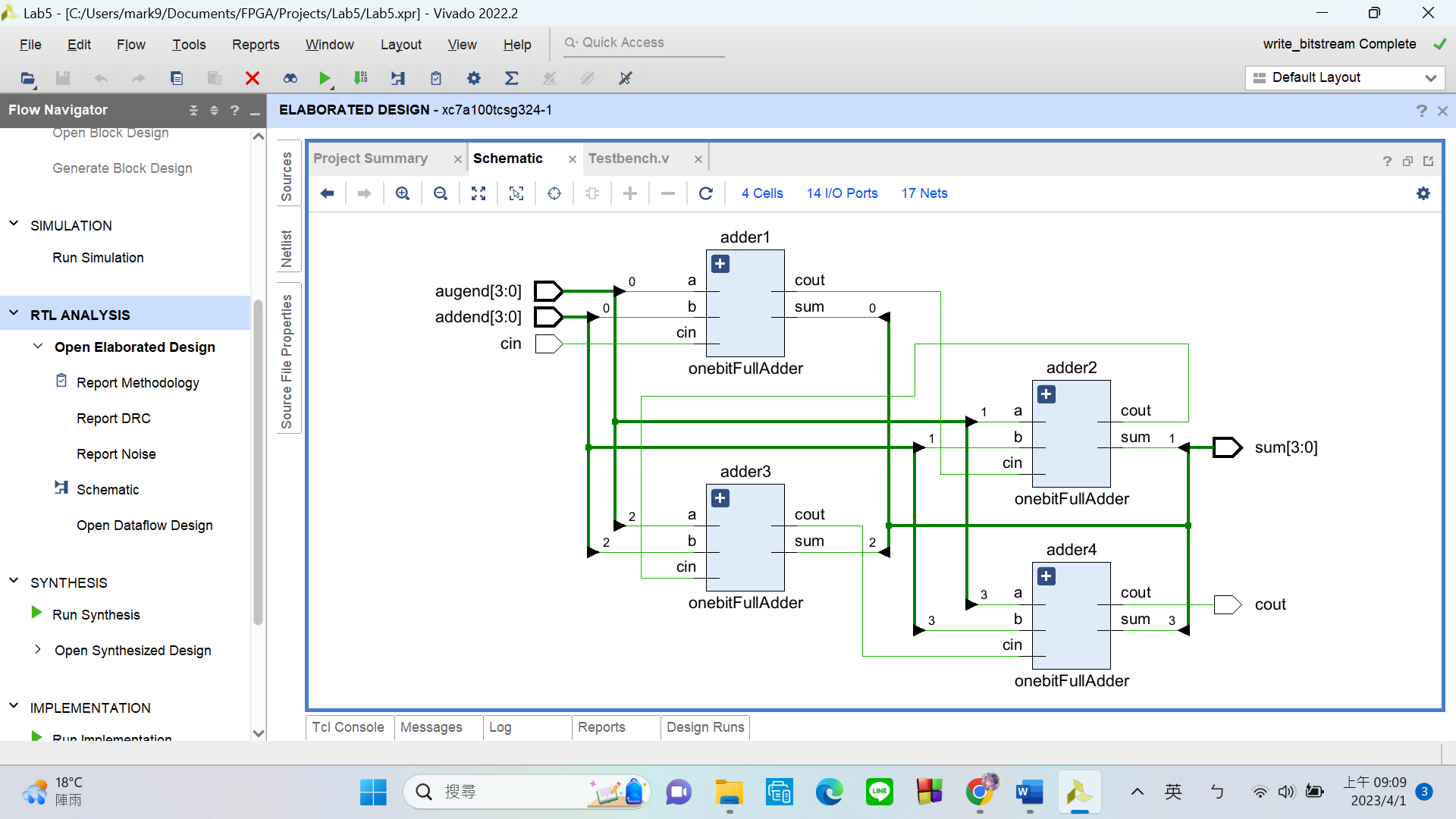
1. modeling style: gate-level
2. input: slide switch (or AD2 pattern generator input through Pmod)
3. output: led
4. arithmetic function block



carry-in + augend + addend = {carry-out, sum}

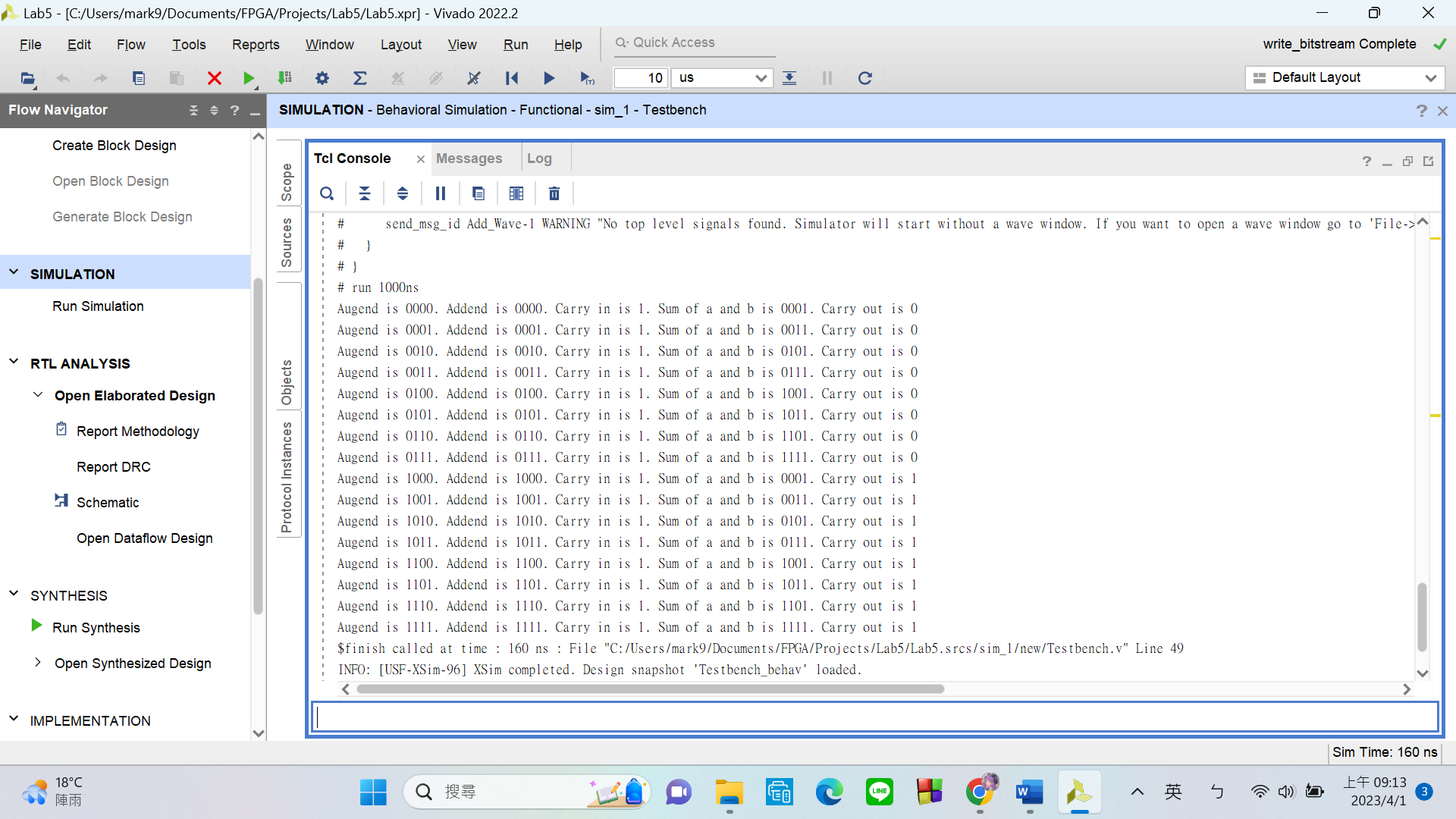
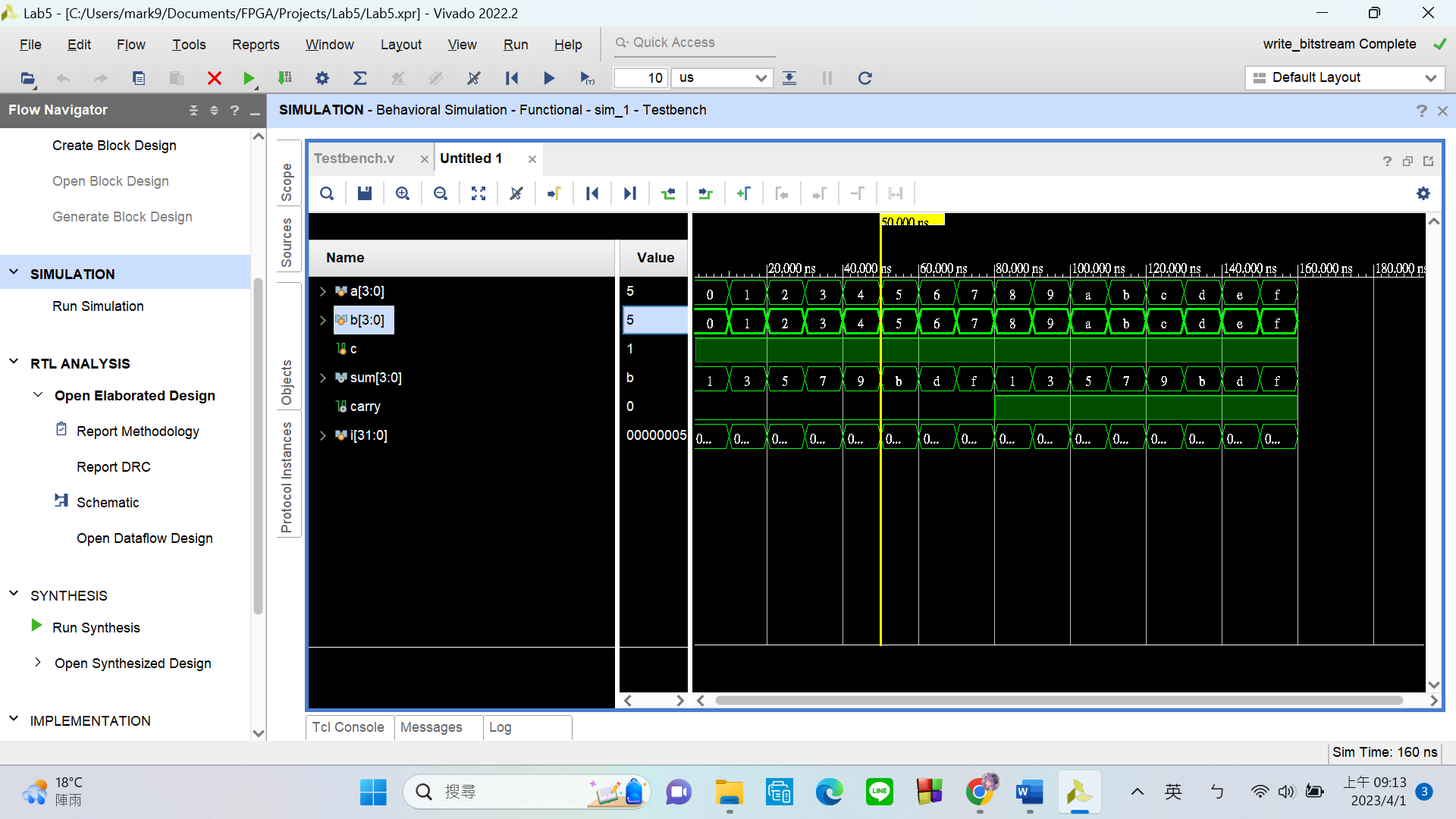
Experiment Data

RTL Schematic



Simulation results

(Design your own testbench)



From the simulation interface and the result printed out in the console, we can see that the 4-bit adder is working as intended.

Practical results on the FPGA board

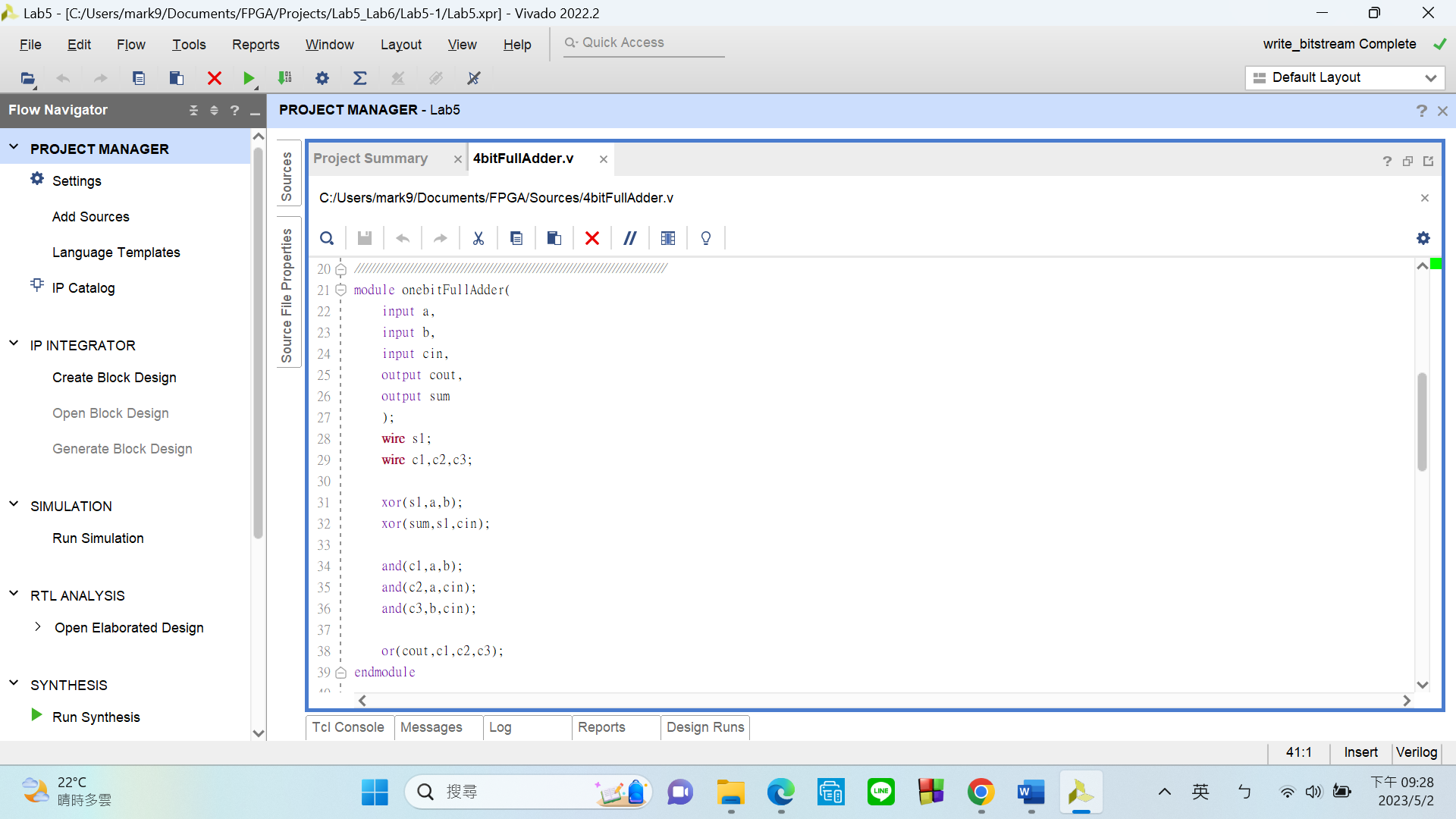
static record (photo, screen capture) or dynamic record (video link)

<https://www.youtube.com/watch?v=e9fZghnsEyQ>

OPTIONAL: design process

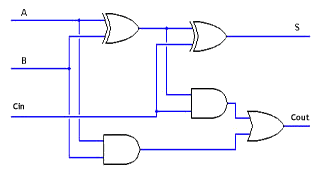
**Let’s look at my code in 2 parts:**

1. **Submodule – One-Bit Full Adder**



In this module, I implemented a one-bit full adder using the following gate logic.

As we all know, the **SUM** of a one-bit full adder is: **A ⊕ B ⊕ Cin**, and the **CARRY OUT** is: **AB + ACin + BCin**.



1. **Top module – Four-Bit Full Adder**

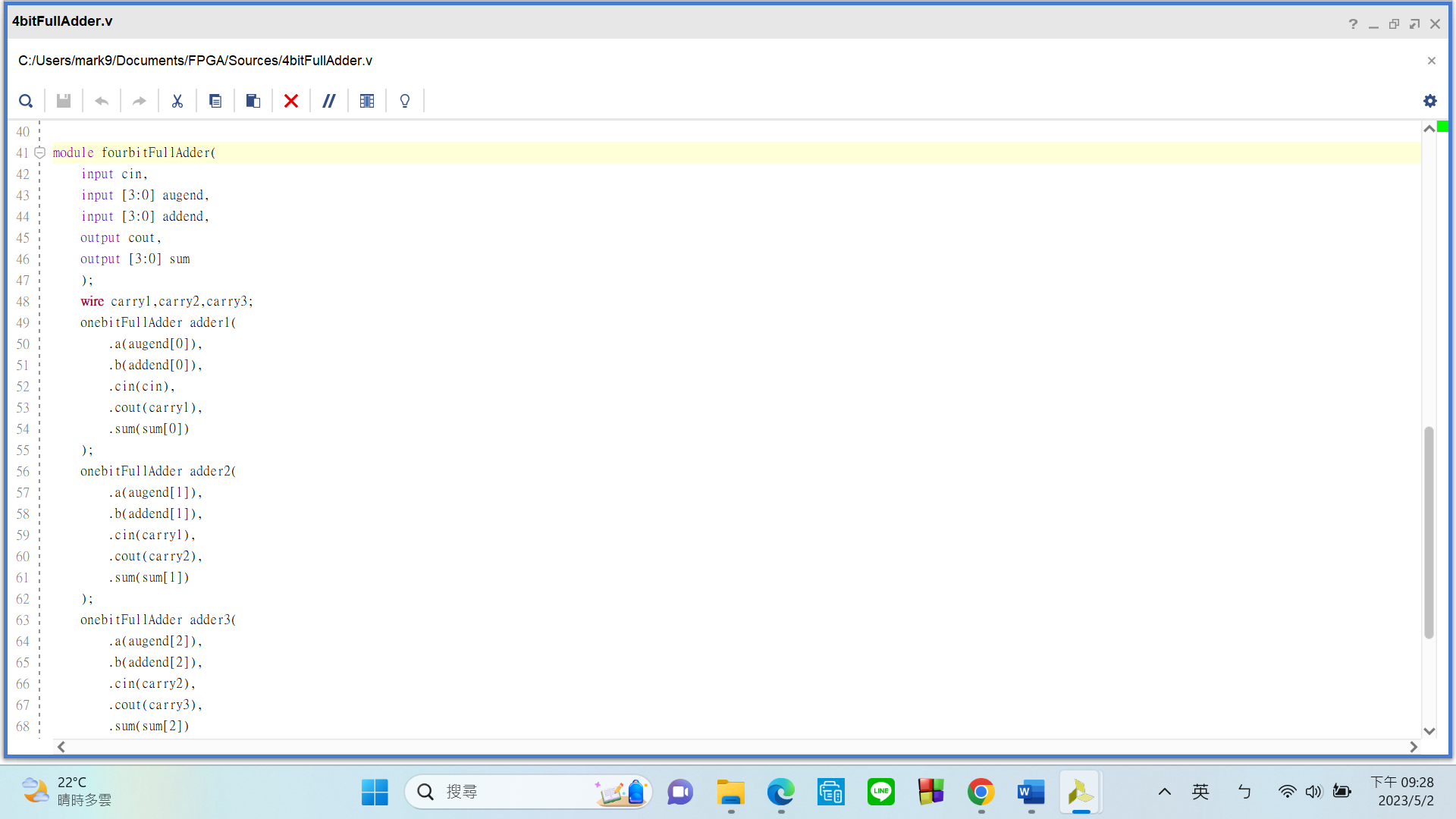
My top module has

input:

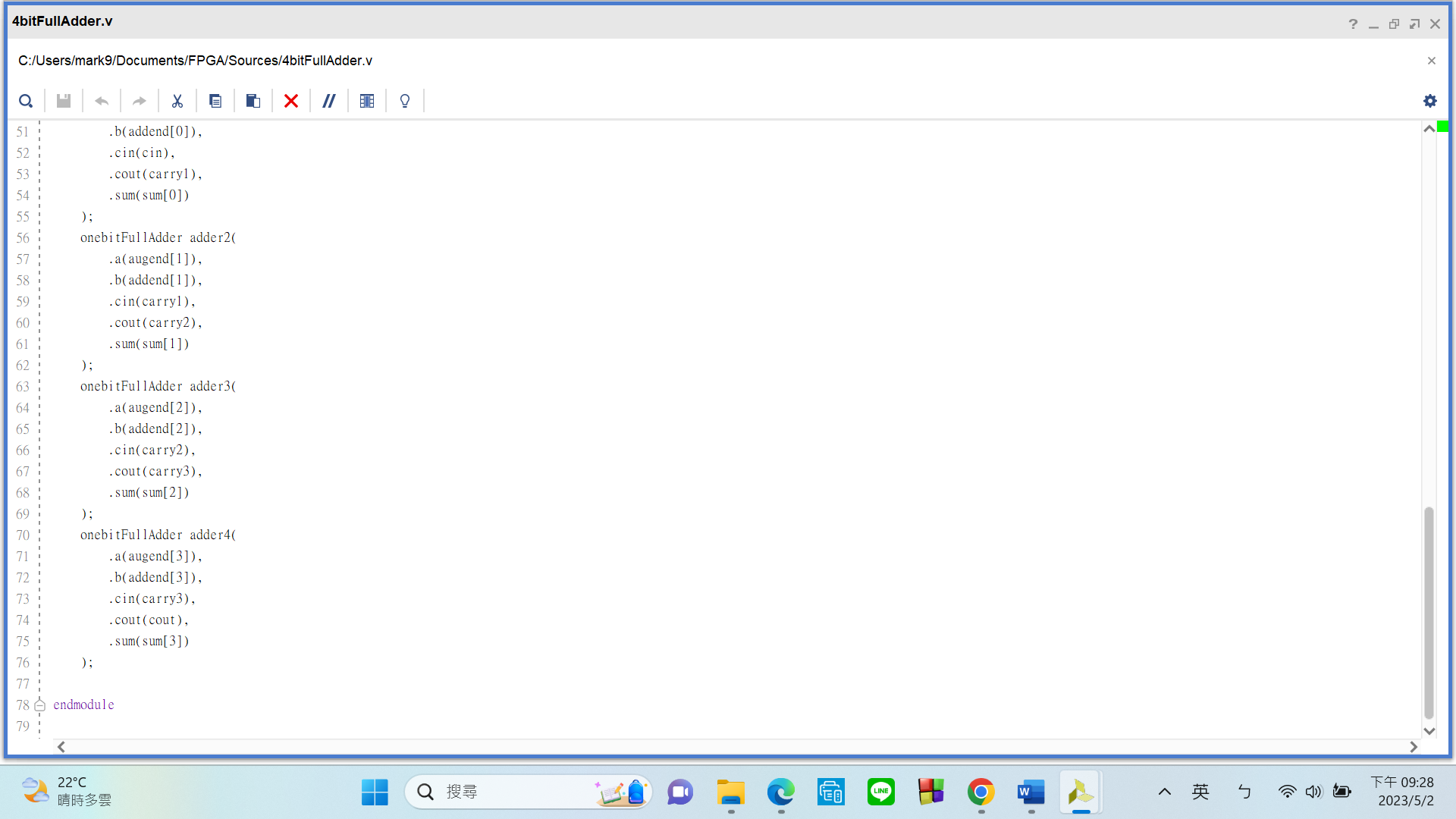
* cin: carry in
* augend: a 4-bit-number
* addend: a 4-bit-number

output:

* cout: carry out
* sum



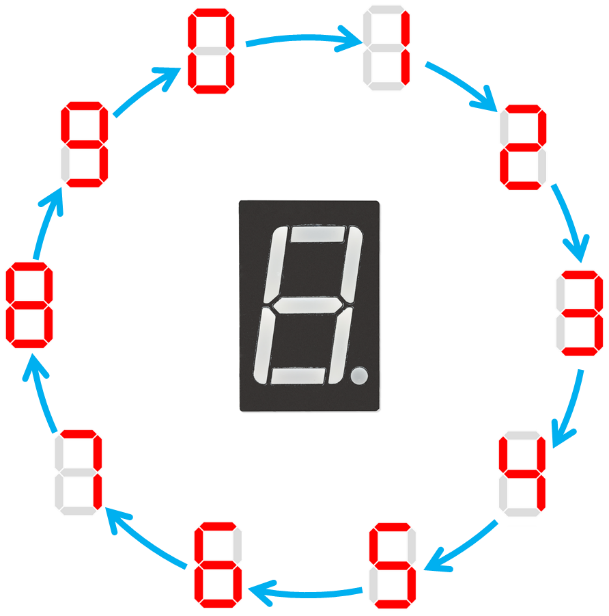
To create a four-bit full adder, I simply **linked 4 one-bit adders together**, and voila! It’s done.



**FPGA Experiment-502: BCD Sequence Generator**

**Design a sequence generator which is satisfied the following requirements:**

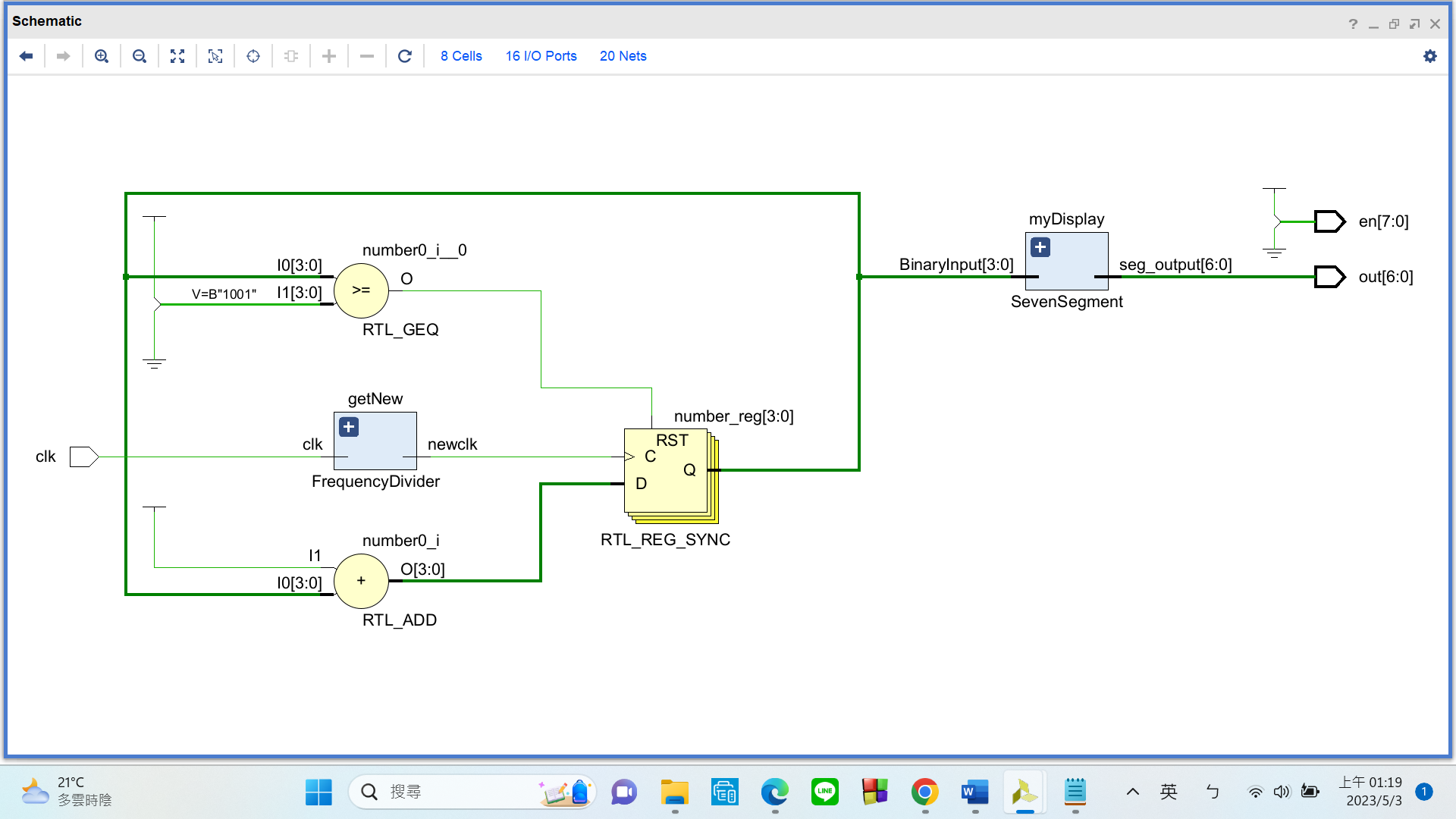
1. modeling style: gate-level
2. input: FPGA built-in CMOS oscillator (i.e., pin E3)
3. output: **exactly one** 7 segment display = 1-digit number
4. sequence example



* sequence contains 10 elements
* direction: count up (or you can design both directions up and down)

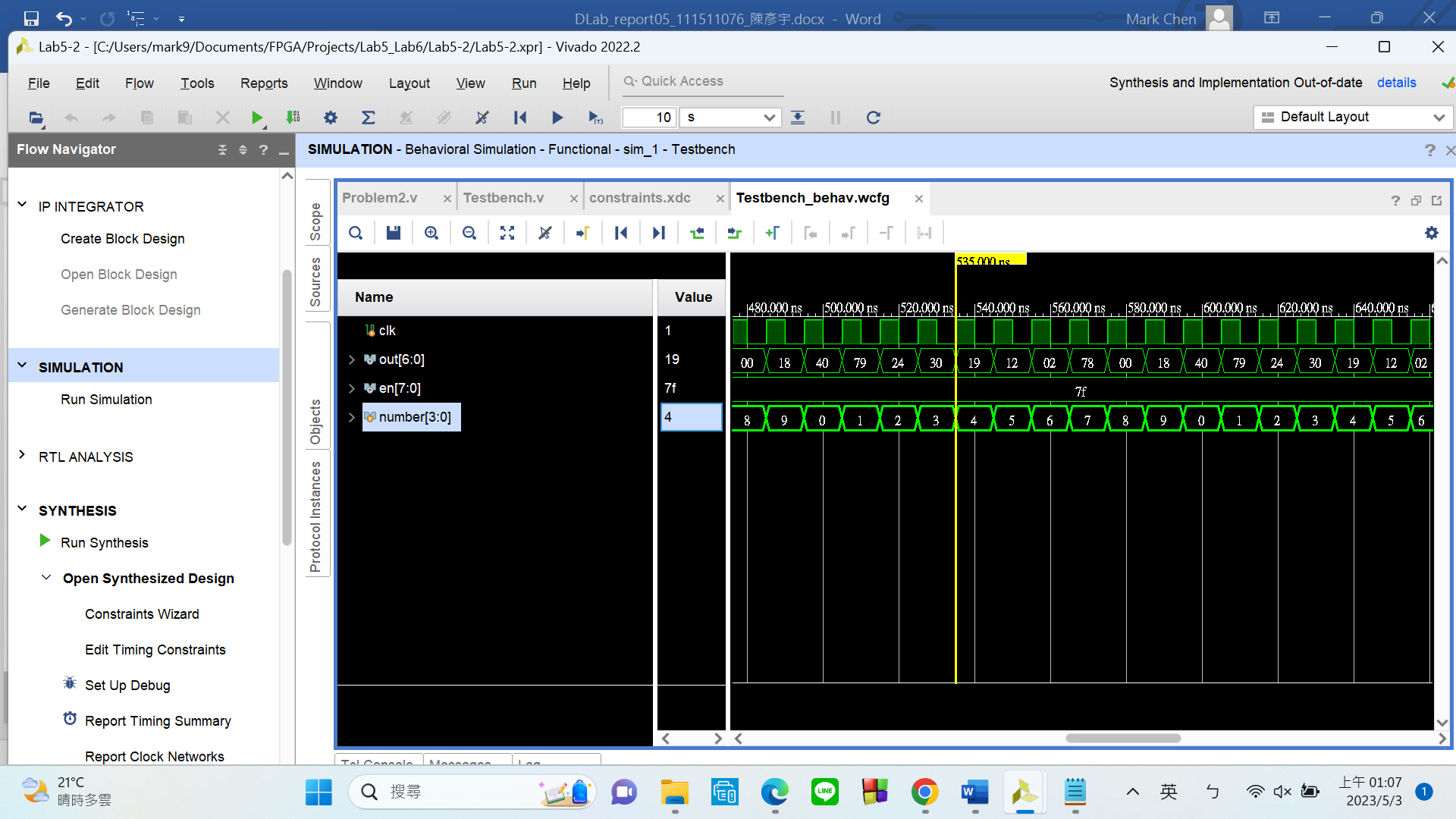
Experiment Data

RTL Schematic



Simulation results

(Design your own testbench)



As you can see, the output is correct. My “number” reg repeats the process of counting from 0 to 9 and then reset back to 0.

Practical results on the FPGA board

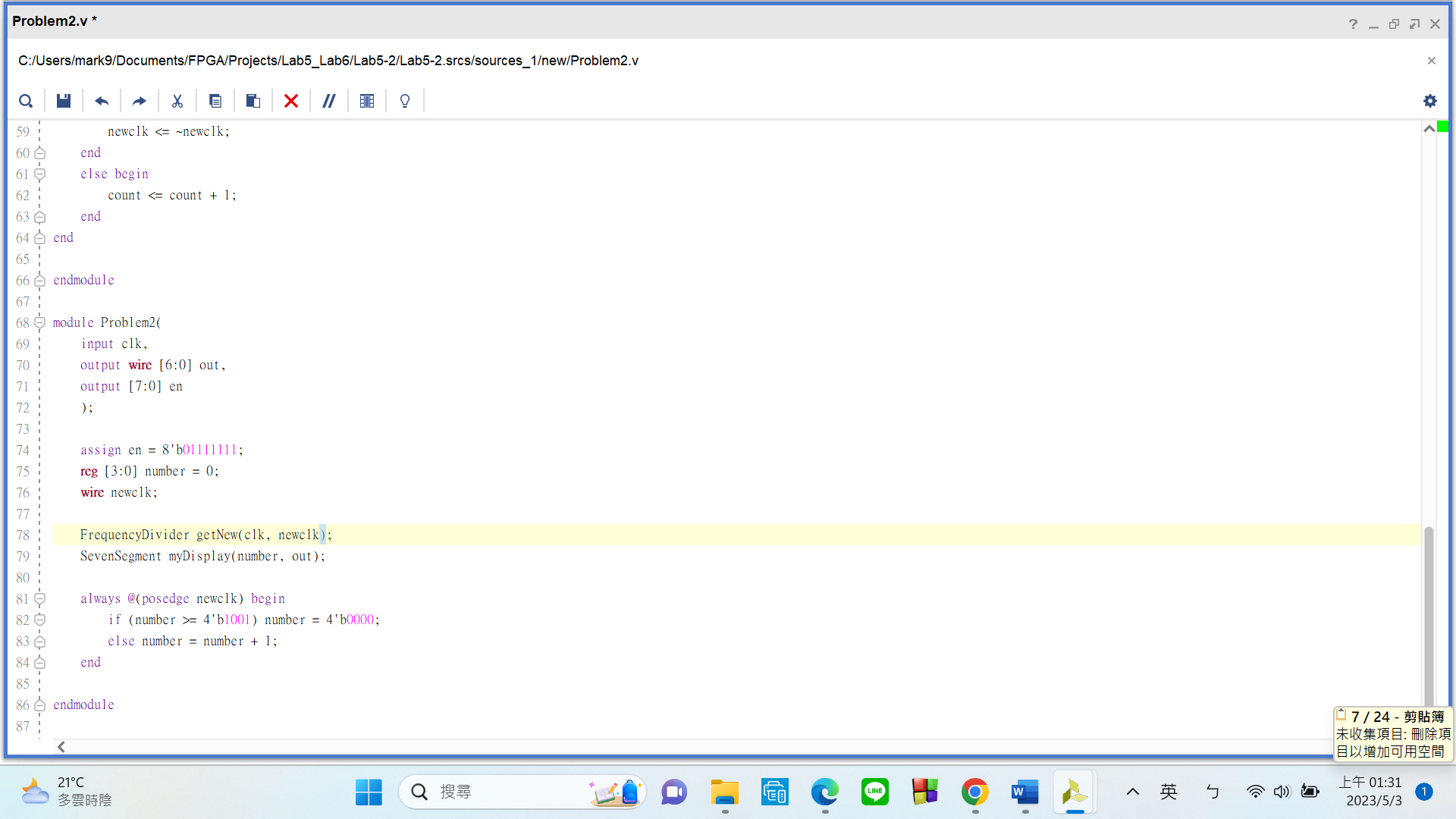
static record (photo, screen capture) or dynamic record (video link)

<https://www.youtube.com/watch?v=vLWa761XUfQ>

OPTIONAL: design process

**Let’s look at my code in 3 parts:**

1. **Top Module**



My top module has

input:

* clk: FPGA built-in CMOS oscillator (pin E3)

output:

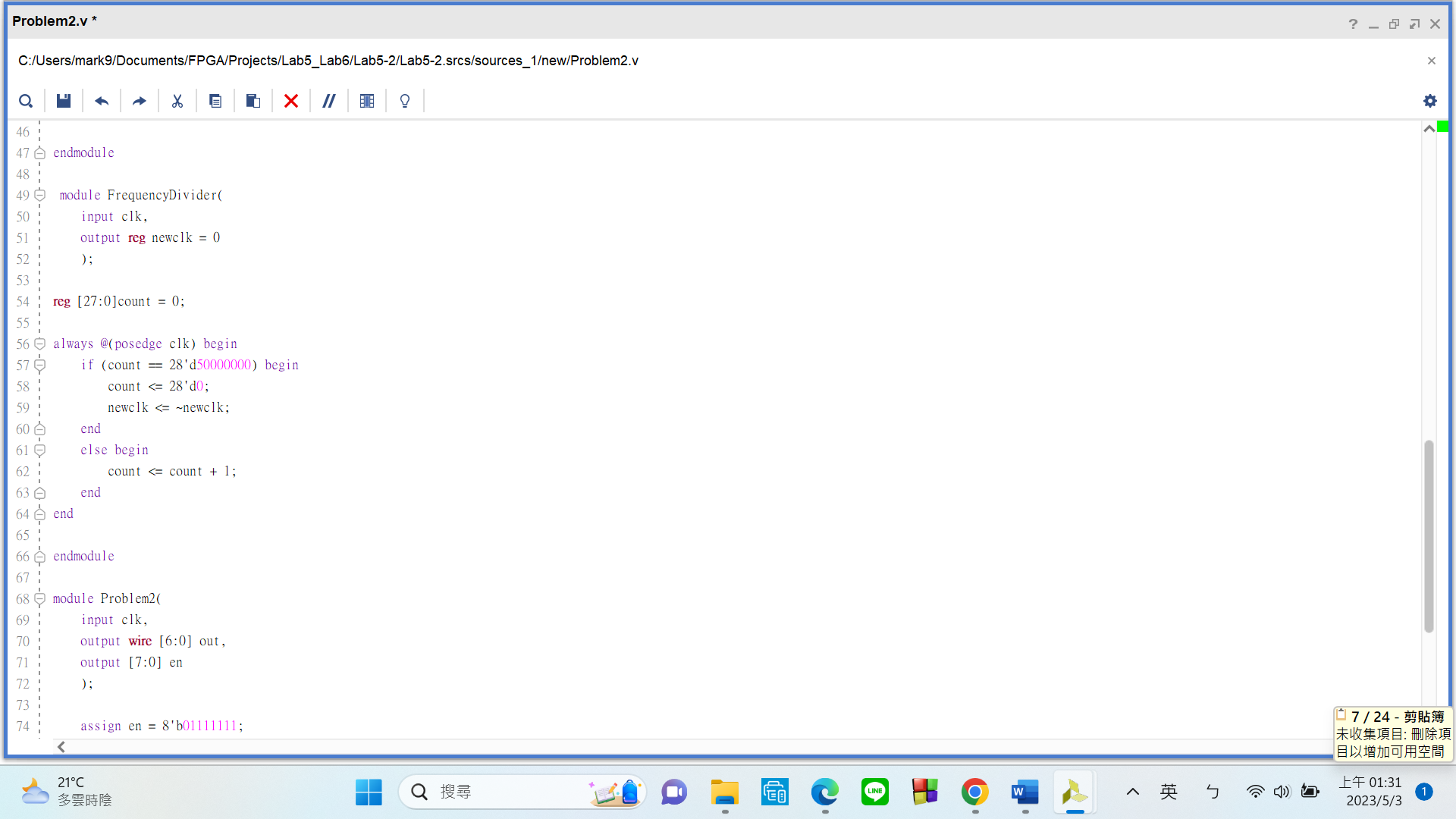
* en: Enable signal that turns on the seven segment displays
* out: controls how the seven segment display should glow (A ~ G)

There are 2 submodules in my top module:

* FrequencyDiv: This is a standard frequency divider that takes the 100MHz from pin E3 and obtain a new clk signal of our own choice.
* SevenSegment: Matches the numbers to what the seven segment display should show.

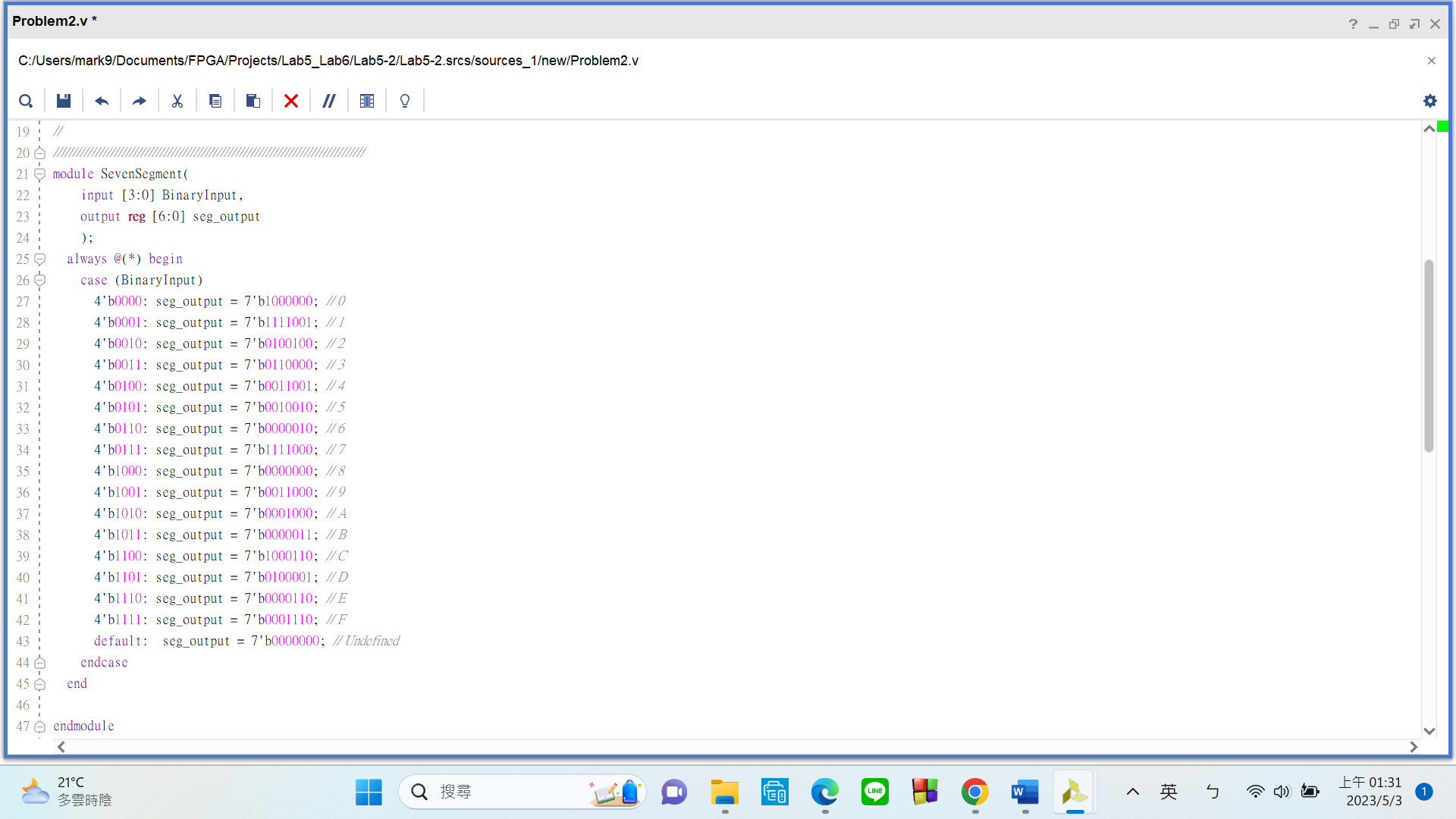
I have a 4-bit-reg that is initialized to 0. It will increase by 1 every time the divided clk signal hits a positive edge. The reg **will count from 0 to 9 and then reset back to 0** when it reaches 9.

1. **Submodule – FrequencyDiv**



Please take a look at the always block. My frequency divider works by **counting up to 50M** before inverting the signal of the “newclk” output, this should create a **1Hz** new clk signal.

1. **Submodule – SevenSegment**



This module uses a **case statement to match numbers to how the seven segment display should glow**. This module will be very handy in the future, considering how we are going to use seven segment displays a lot.

**FPGA Experiment-503: ID Sequence Generator**

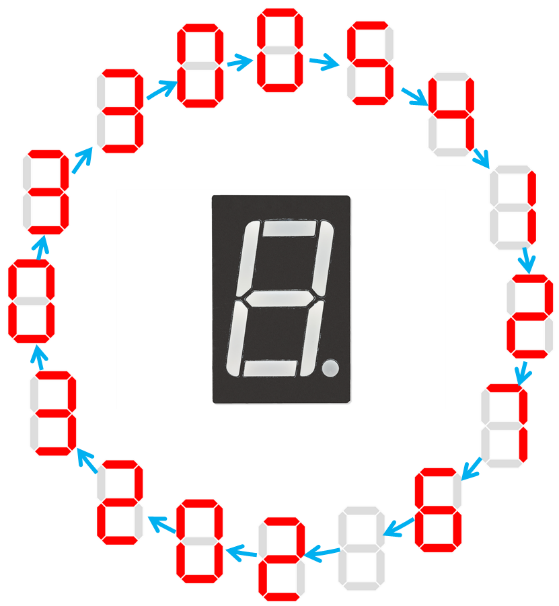
**Create a sequence generator which is satisfied the following requirements:**

1. modeling style: behavior
2. input: FPGA built-in CMOS oscillator (i.e., pin E3)
3. output: **exactly one** 7 segment display = 1-digit number
4. sequence elements corresponding to your student ID

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| element number | | | | | | | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | | 12 | 13 | 14 | 15 |
| 7-digit ID number | | | | | | | 1-digit  blank | 4-digit  birth year | | | | | 2-digit  birth month | | 2-digit  birthday | |
| 8-digit ID number | | | | | | | | 1-digit  blank | 2-digit  birth month | | 1-digit  blank | | 2-digit  birthday | | 2-digit  table  number | |
| 9-digit ID number | | | | | | | | | 2-digit  birth month | | | 2-digit  birthday | | 1-digit  blank | 2-digit  table  number | |

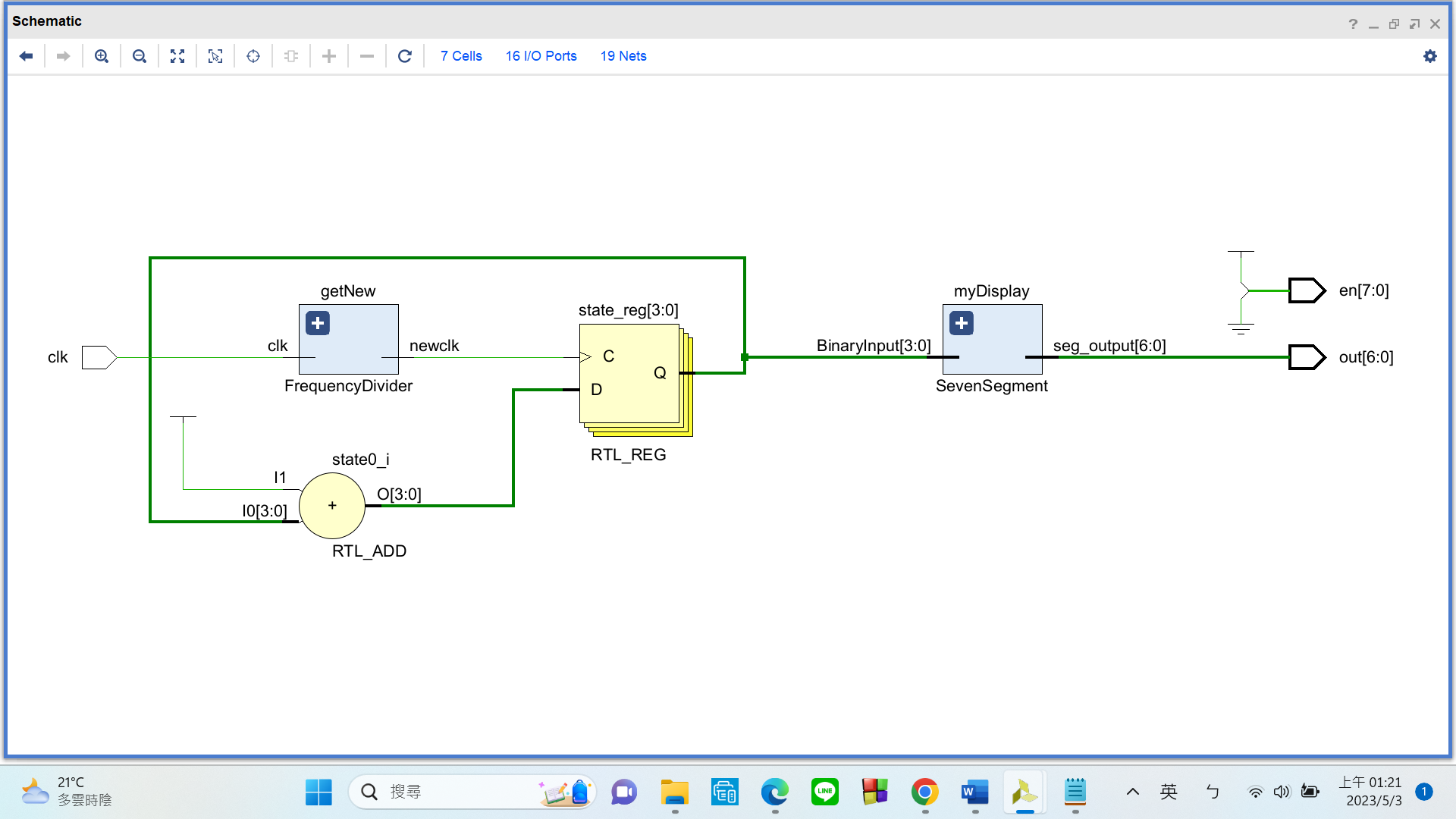
1. example

ID = 05412176, birthday = 03/30, table number = 01 🡪 sequence = 0541276□20230330



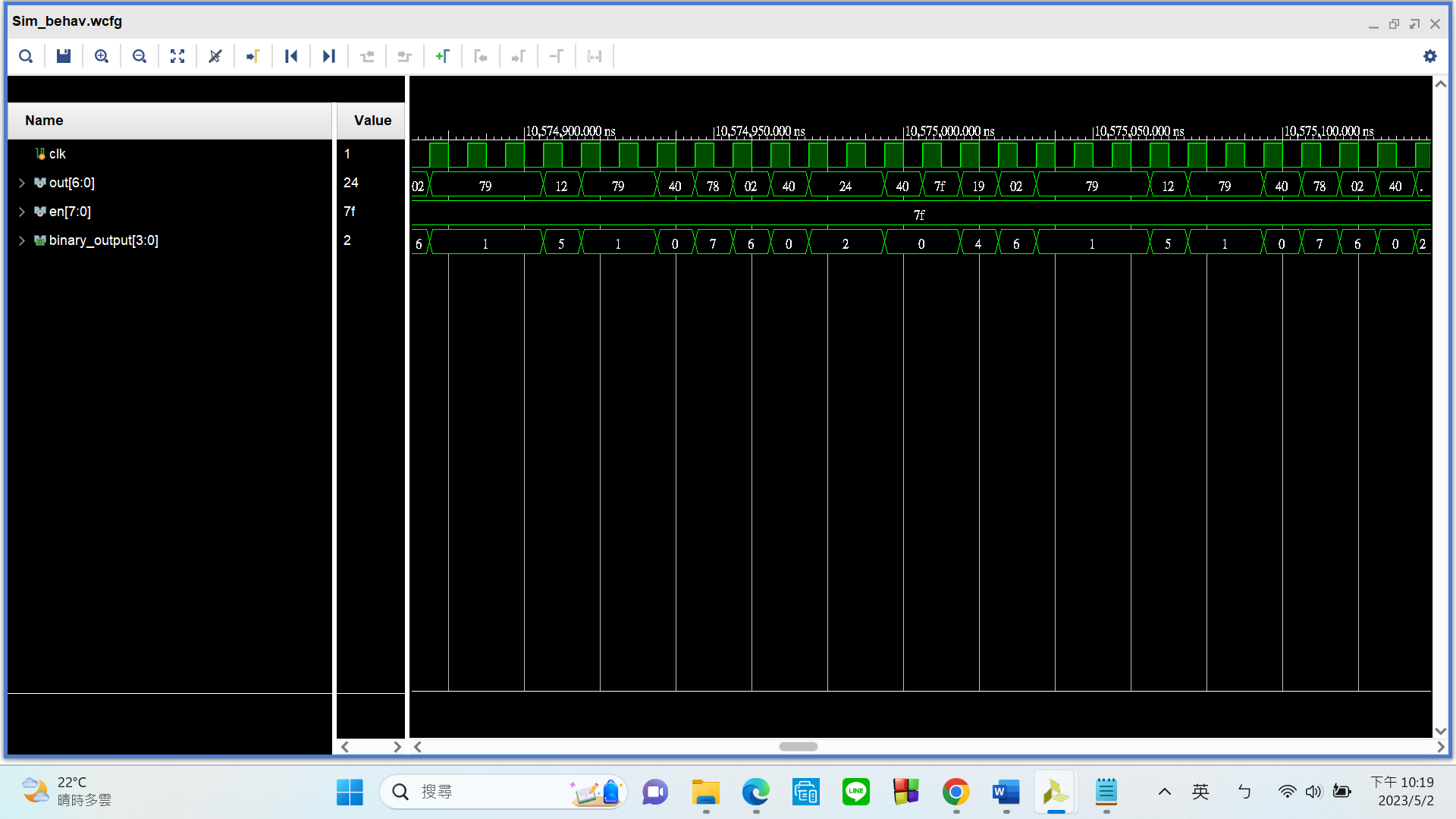
Experiment Data

RTL Schematic



Simulation results

(Design your own testbench)



My ID: 111511076

My birthday: 0220

My table number: 46

Output: 1115110760220\_46

As you can see, the output is correct.

Practical results on the FPGA board

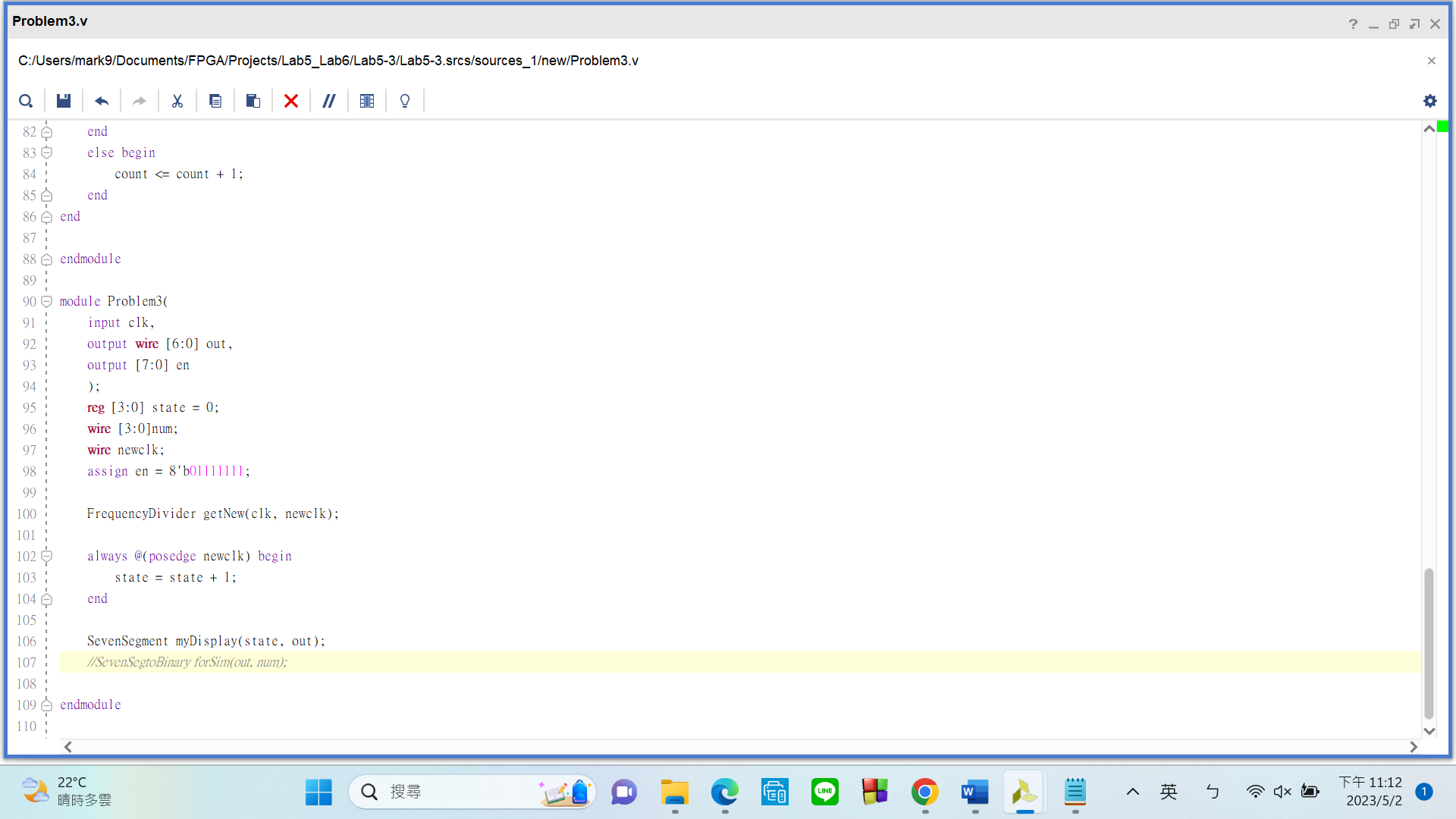
static record (photo, screen capture) or dynamic record (video link)

<https://www.youtube.com/watch?v=NdZVBX4277A>

OPTIONAL: design process

**Let’s look at my code in 3 parts:**

1. **Top Module**



My top module has

input:

* clk: FPGA built-in CMOS oscillator (pin E3)

output:

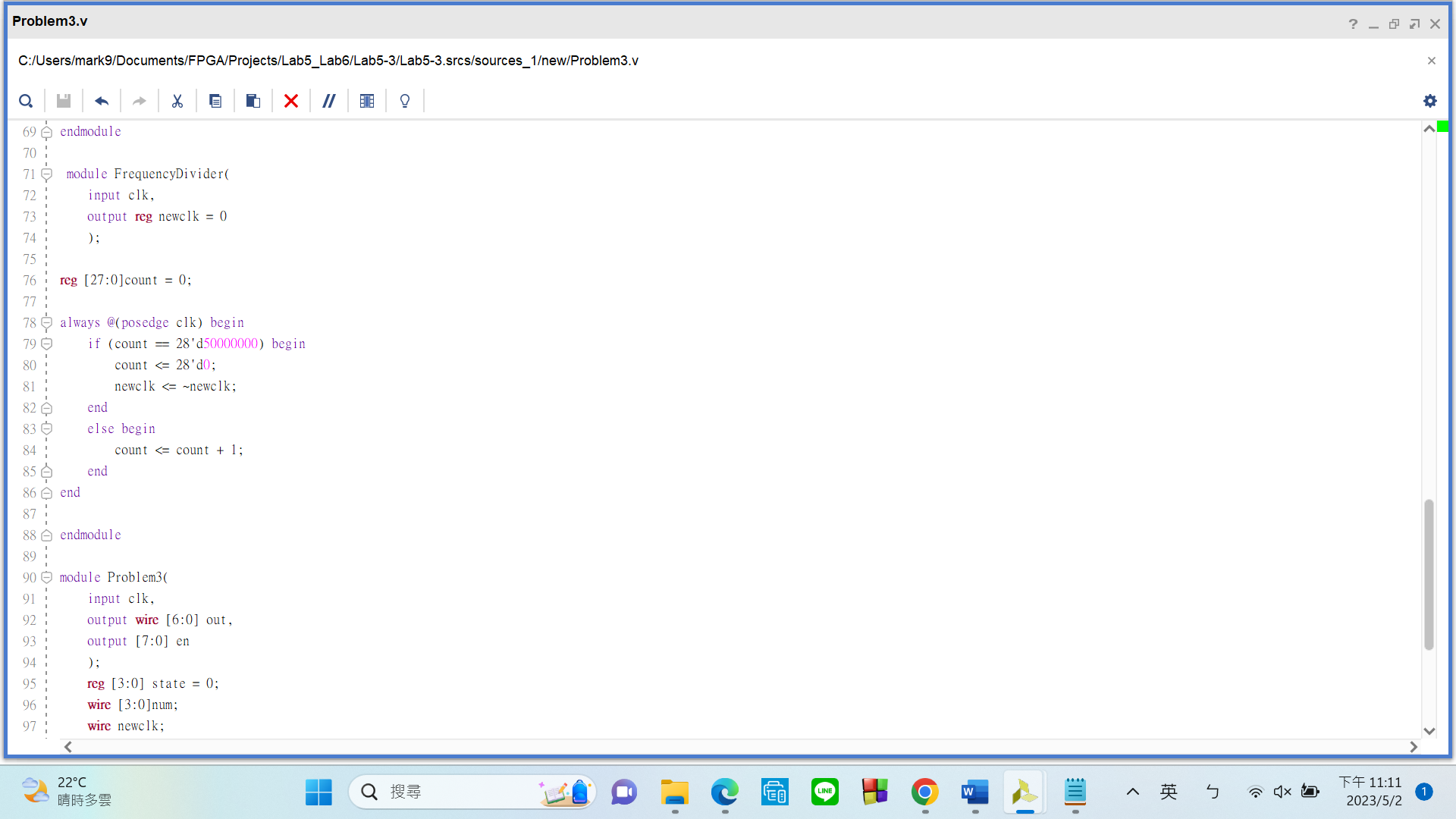
* en: Enable signal that turns on the seven segment displays
* out: controls how the seven segment display should glow (A ~ G)

There are 2 submodules in my top module:

* FrequencyDiv: This is a standard frequency divider that takes the 100MHz from pin E3 and obtain a new clk signal of our own choice.
* SevenSegment: Matches each of the **16 states** (explained in detail later) to what the seven segment display should show.

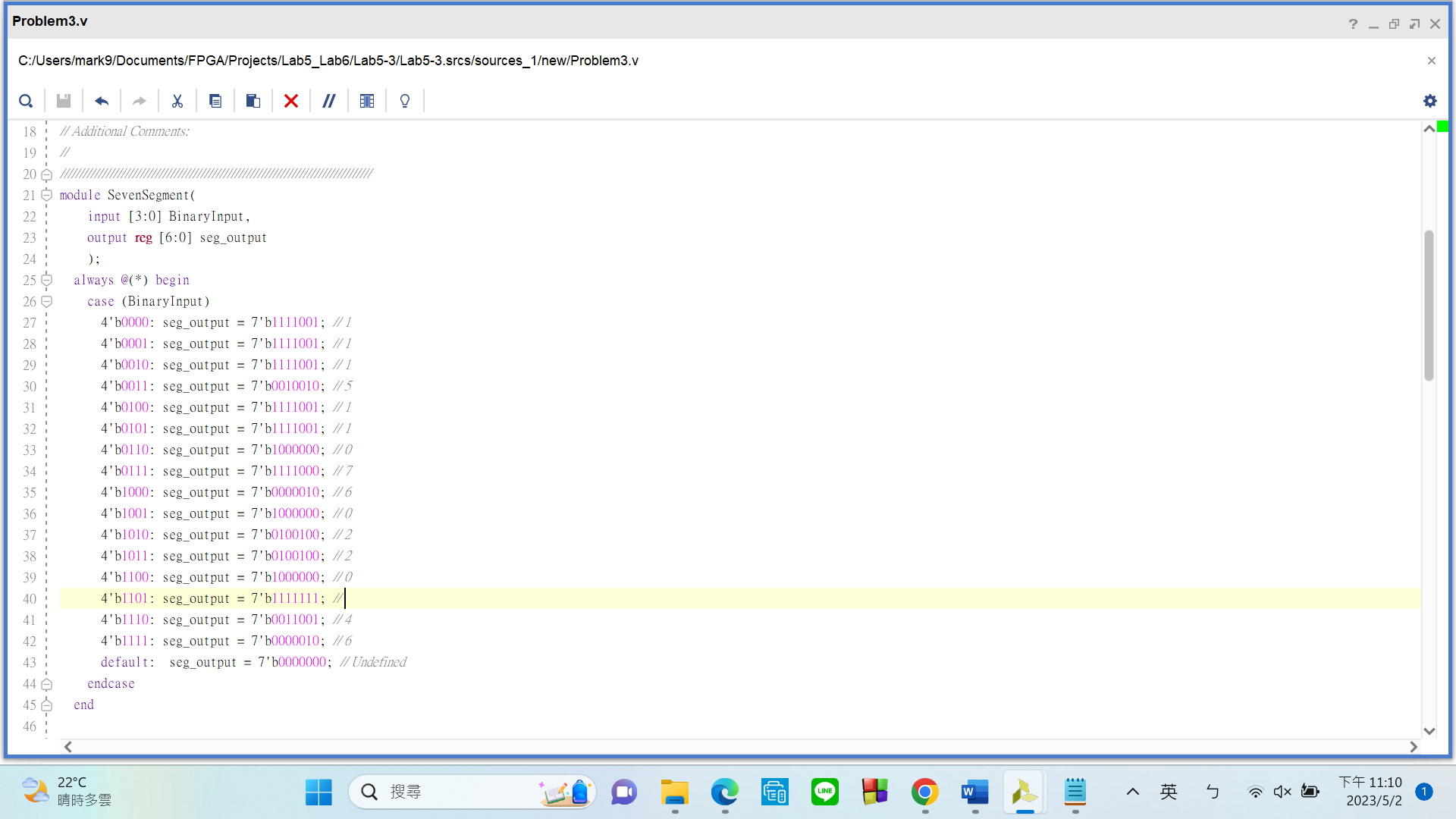
I have a 4-bit-reg named “state” initialized to 0. This reg will represent the current state. **Since there are 16 possible display outputs, there are going to be 16 states**. The current state **becomes the next state every** time the divided clk “newclk” hits a **positive edge**, causing the number to change.

1. **Submodule – FrequencyDiv**



Please take a look at the always block. **My frequency divider works by counting up to 50M** before **inverting** the signal of the “newclk” output, this should create a **1Hz** new clk signal.

1. **Submodule – SevenSegment**



This SevenSegment module is **modified from the one used in 5-2**, so the variable names are all the same. Instead of matching actual numbers to their seven segment display**, this version is matching states to what should be displayed**.

**Problems Encountered**

* **Module outputs cannot take reg variables**

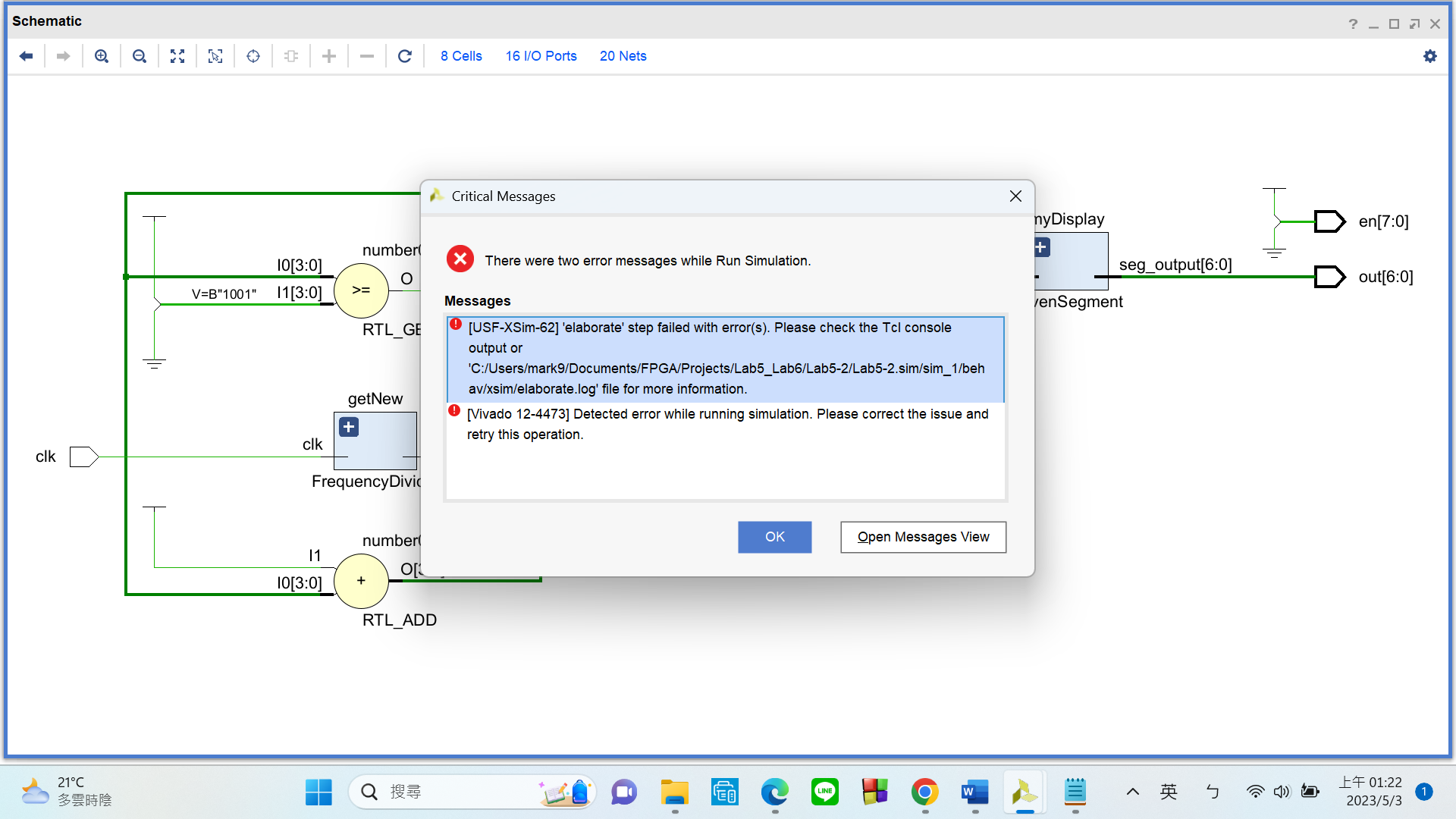
Solution: change “reg” to “wire”

* Frequency divider working as intended on FPGA, yet failed during simulation

Solution: It turns out I forgot to give my counter and output “newclk” **initial values**. Do that and it’s fixed.

* Can’t find simulation error messages

Solution: Follow the path (**project\_name.sim/sim\_1/behave/xsim/elaborate.log**) to the designated directory to find the error message.



* Simulation fails if you leave modules unnamed

Solution: give the module a **name**

This isn’t really an excuse, but I only figured these solutions out around the time we reached Lab 7. It would have been nice if there was a tutorial that detailed these things.