**LAB REPORT**

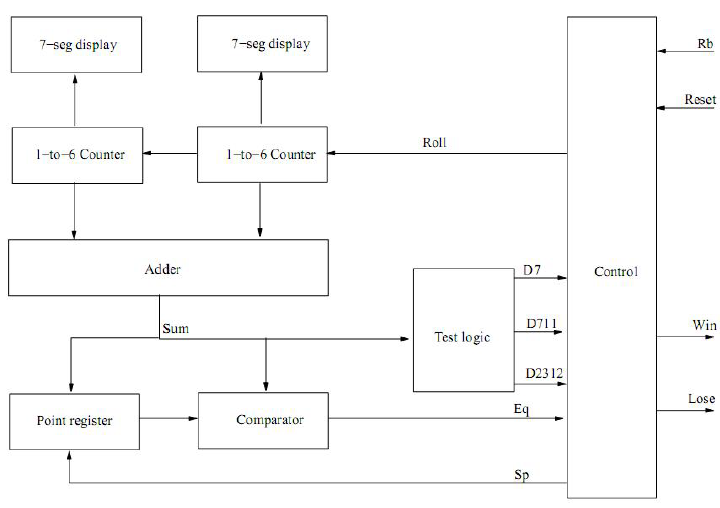
**Laboratory exercise 7: A Dice Game**

3220104688 杨佳昕

19 Dec, 2024

**1. Problem Description**

When the roll button is pressed, two dices (shown on two hexes) begin to scroll from 1 to 6 randomly. When the roll button stopped for the first time, the player wins if the sum of two dices is 7 or 11, and loses if 2, 3 or 12. Otherwise, the game continues. When the button stopped again, the player wins if the sum equals to the sum in the first turn, and loses if 7 and so on. The flow chart is as follows.



*NB: Rb represents roll button.*

Match two hexes to two dices, KEY0 to reset, KEY1 to roll button, green LED to winning display and red LED to losing display.

**2. Design Formulation**

Six states for the problem:

\* **Idle:** just after initializing.

\* **Rolling:** before the first falling edge of roll triggers.

\* **First check:** when the first falling edge of roll triggers, determine whether the sum meets the requirements.

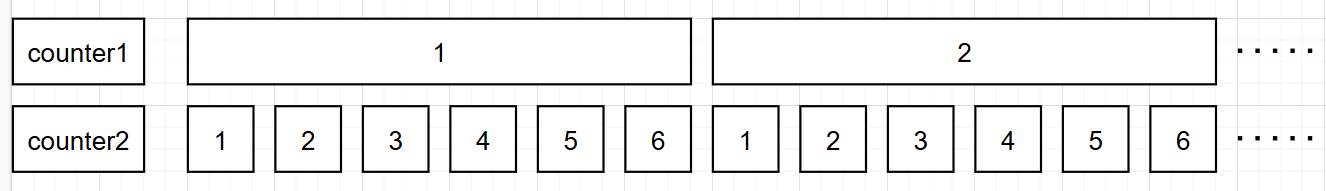
\* **Other check:** after the first trigger, when trigger again, determine whether the sum meets the requirements.

\* **Win:** the green LED lights, and nothing will be changed until reset is pressed.

\* **Lose:** the red LED lights, and nothing will be changed until reset is pressed.

*NB: The first-check state is not declared in the code, because the code directly contains the content of the first check.*

In the *rolling* state, we need to define the random method of counter1 and counter2 (two dices). Because no random function is included in VHDL, we need to design a pseudo-random algorithm for the two variables. The algorithm is as follows.



When roll=1, the algorithm begins. First, counter2 rolls from 1 to 6. Then counter2 is reset and counter1 adds 1 to itself. When counter1 reaches 6, it is reset to 1 as well. With this method, we achieved the pseudo-random of two variables.

In the *first-check* state, when the first falling edge of roll triggers, determine whether the sum meets the winning or losing requirement. If so, go to *win* or *lose* state. If not, go to *other-check* state and save the sum of the two variables for the determination in the following parts.

In the *other-check* state, check if the sum equals to the sum saved above or equals to 7. If so, go to *win* or *lose* state. If not, go to *other-check* state again.

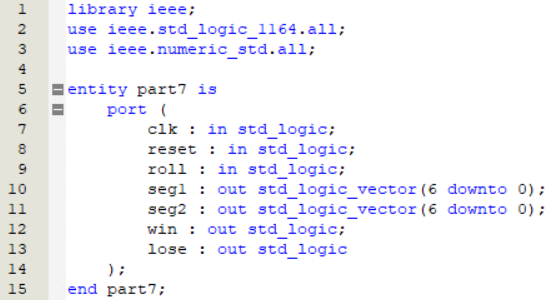
*NB: In fact, the determination check of sum is included in an if-else judgement. The other-check state is only used for hex display.*

In the *win* or *lose* state, we define a variable *finish* which determines whether the game ends or not. And what’s more, we copy the value of two counters and display them on two hexes. Thus, the value on both hexes will not change until reset is triggered.

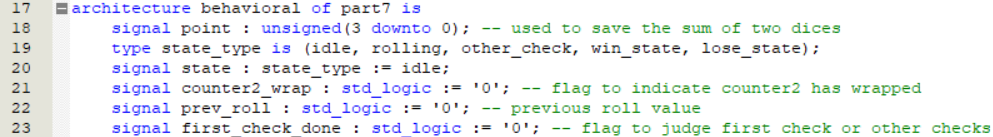
Finally, add an if-else expression for *reset*. If *reset* is triggered, set all variables to the initial state.

**3. Design Entry**

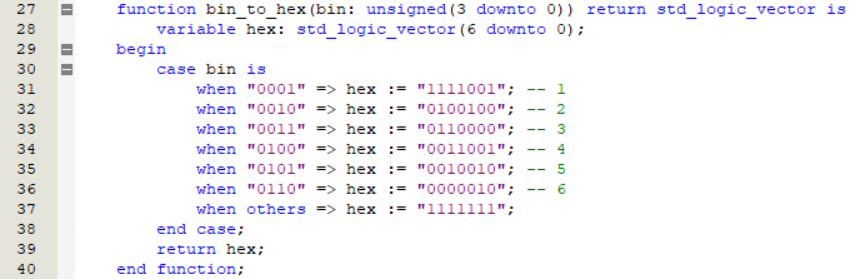
Entity:



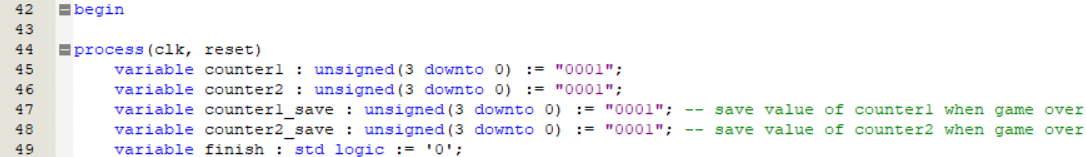
Definition of variables in *signal* form:



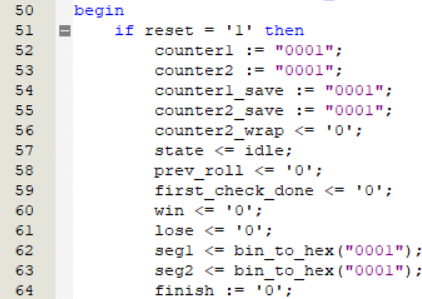
Function used to transfer *counter1* and *counter2* in binary form to hex display:



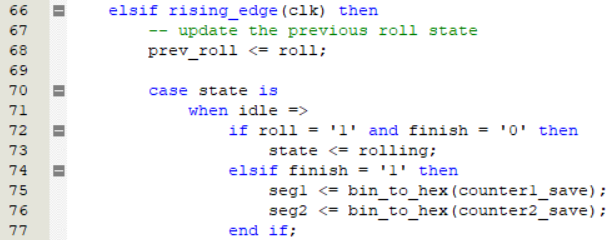
Definition of variables in *variable* form:



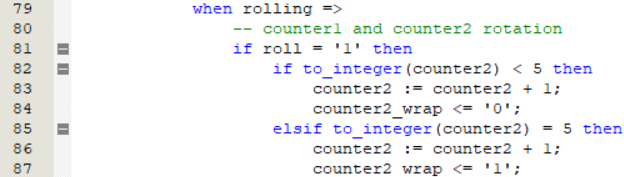
*Reset* initialization:

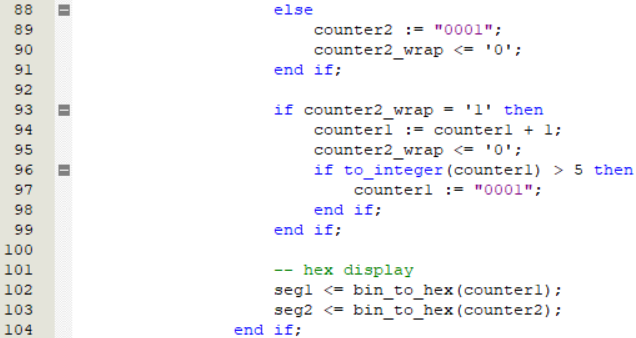


*Idle* state:

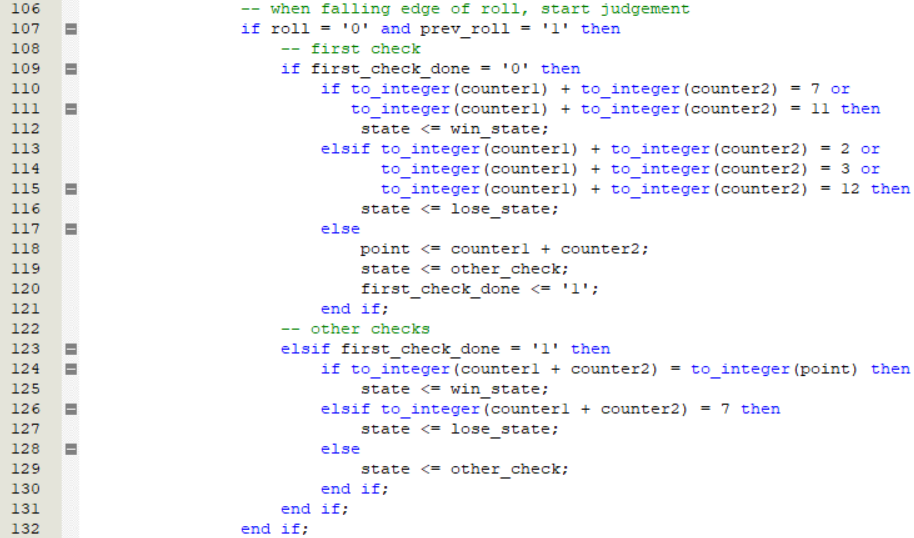


*Counter1* and *counter2* rotation:

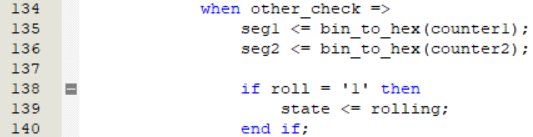




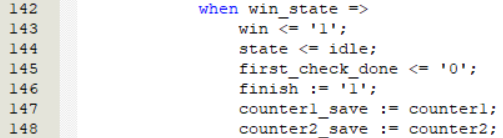
Determination of sum, no matter the first falling edge or other falling edges:

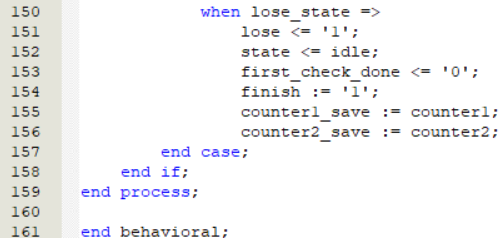


*Other-check* state, just for hex display:



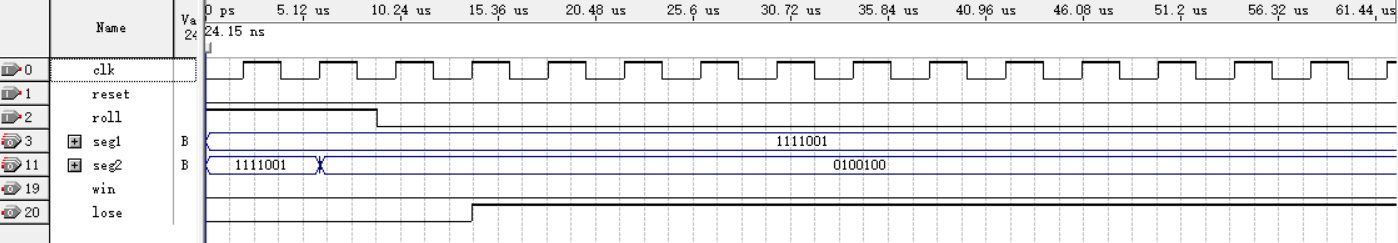
*Win* and *lose* state:



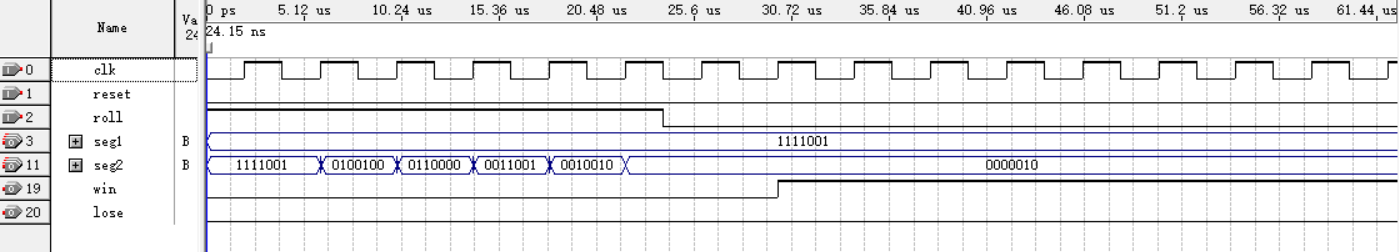


**4. Simulation and Synthesis Results:**

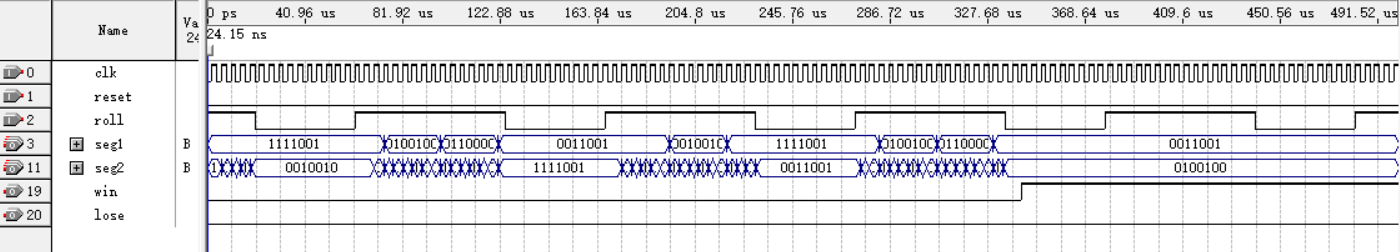
Win state of the first falling edge: *counter1* = 1, *counter2* = 2.



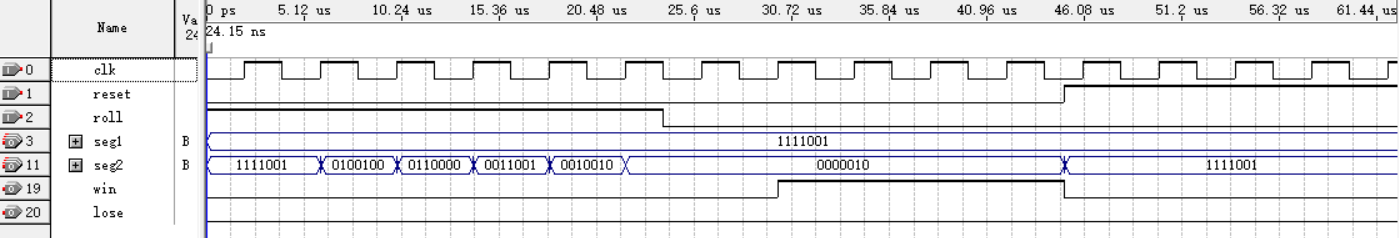
Lose state of the first falling edge: *counter1* = 1, *counter2* = 6.



Win state of not the first falling edge: The sums are 6, 5, 5, 6 respectively.



Reset:

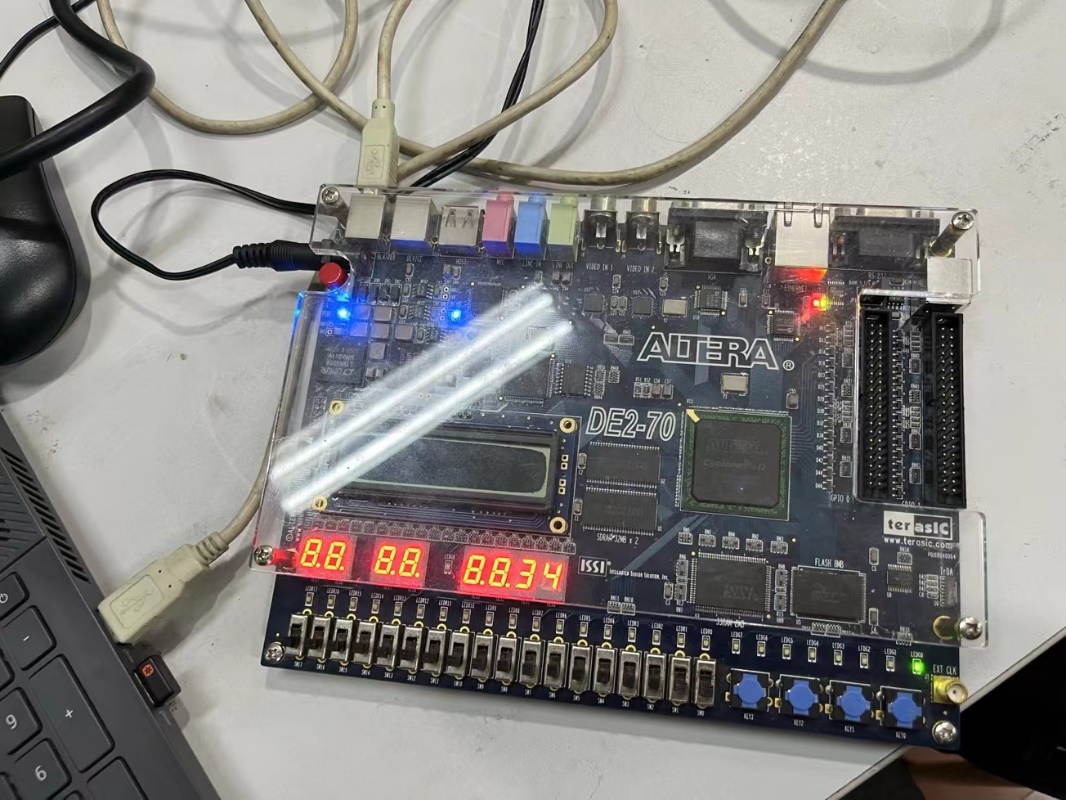


**5. Experimental Results**

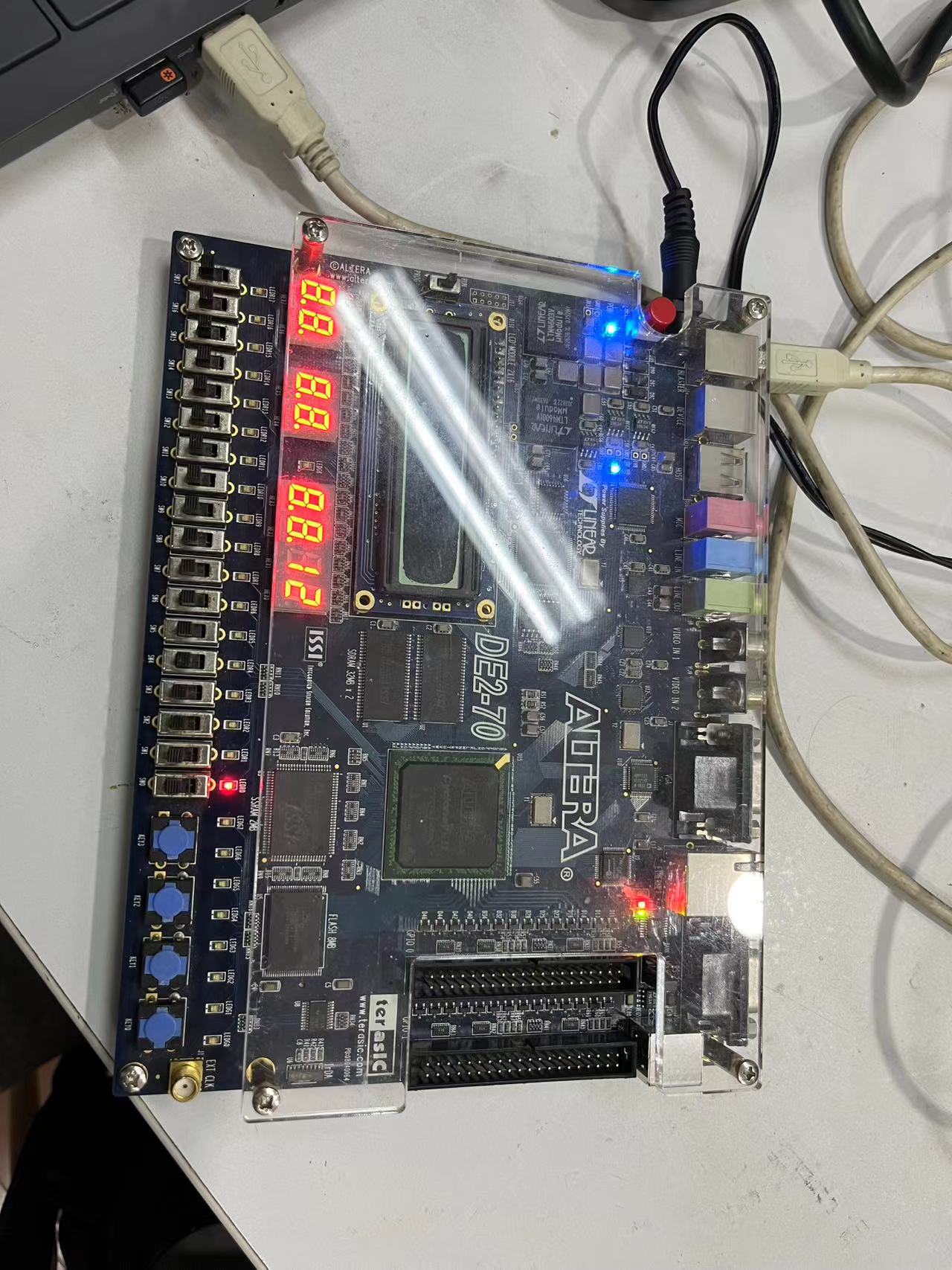
Note: (1) Key press KEY1 is unavailable, and thus use SW0 and SW1 to substitute KEY0 and KEY1.

(2) The code has been slightly changed in order to eliminate the point display in each hex.

Win state of the first falling edge:

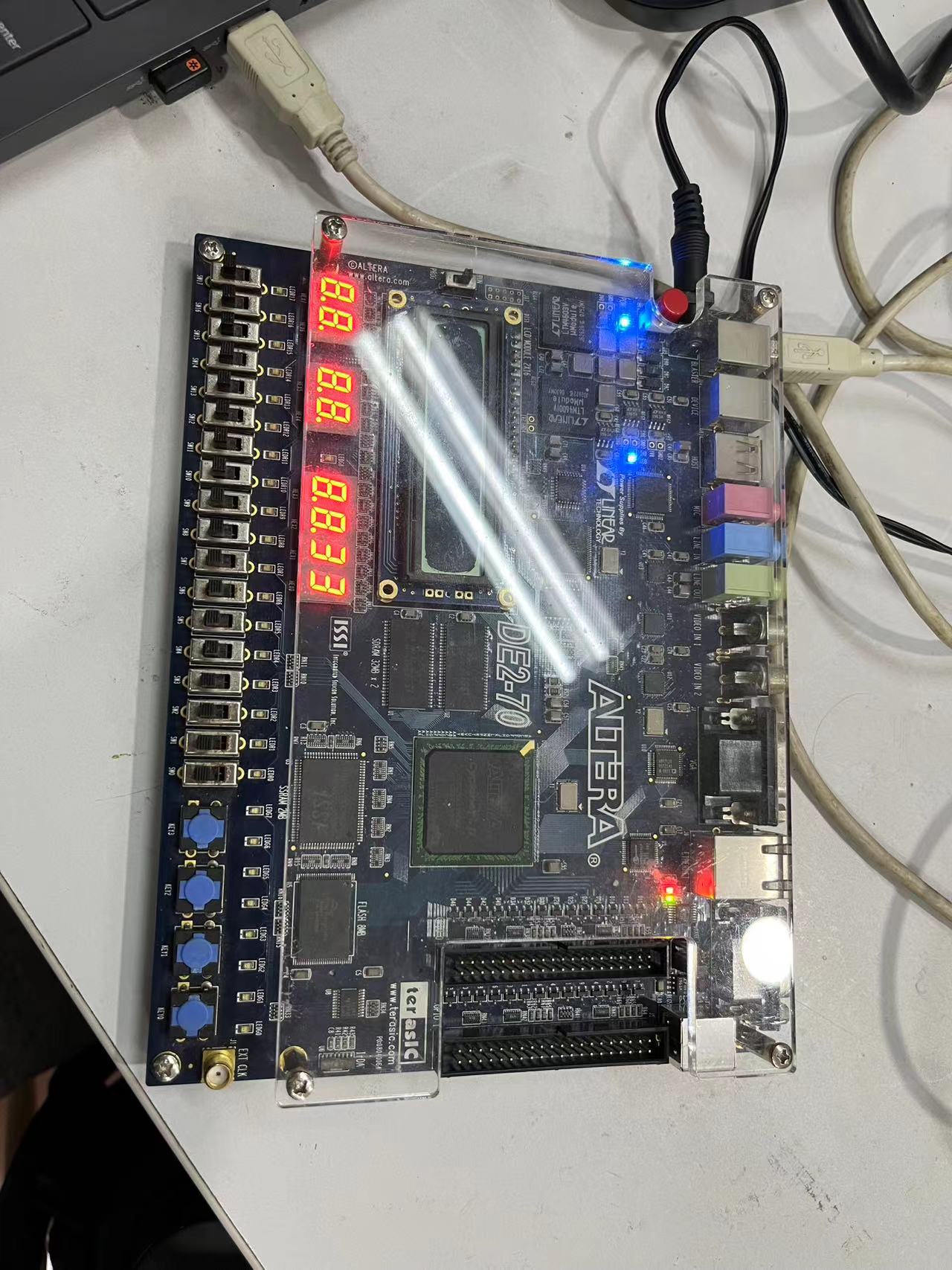


Lose state of the first falling edge:

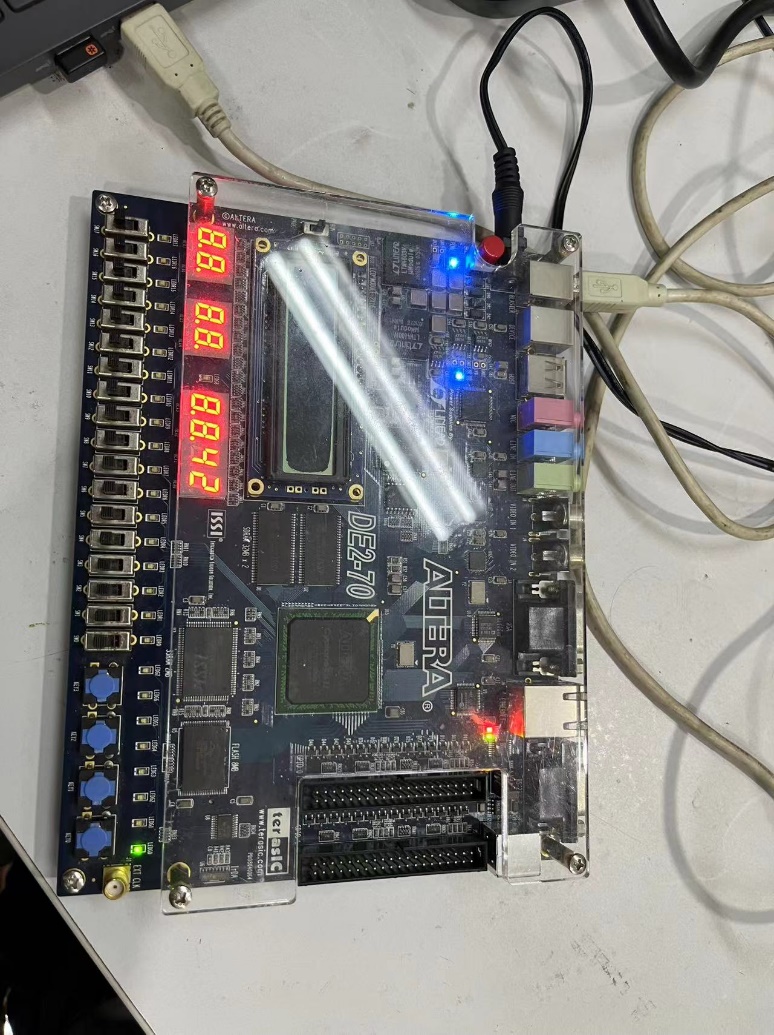


Win state of not the first falling edge:

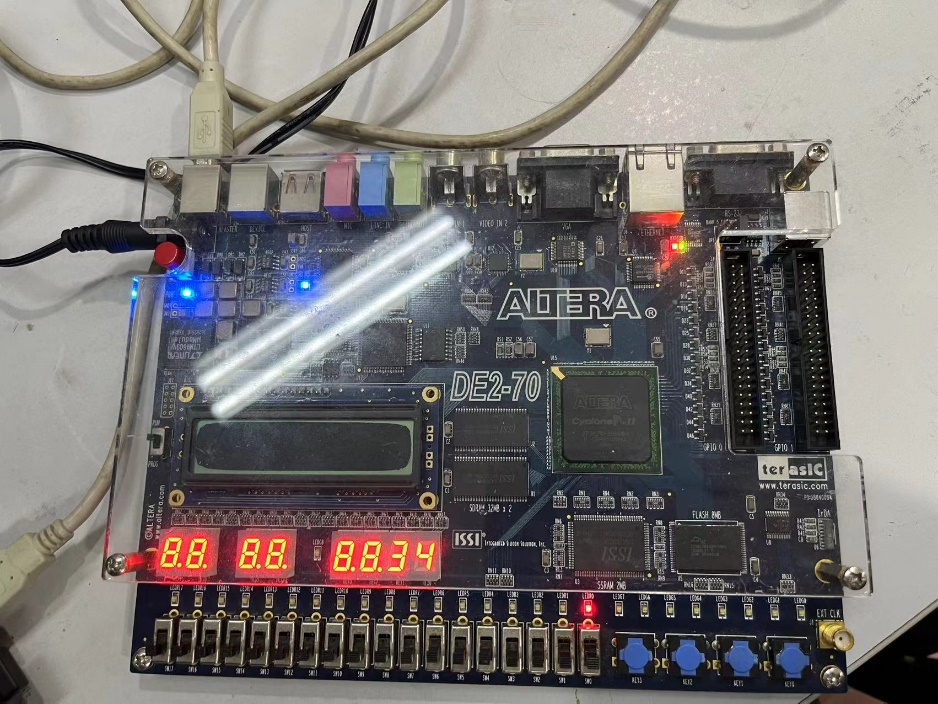
*First try:*



*After several tries:*



Lose state of not the first falling edge:



**6. Discussion and Conclusion**

**Discussion I**

In this experiment, I match the 50MHz clock with the clk input in the code. My initial purpose is to make the values on both hexes pseudo-random, however, they are too random to examine whether the code is right or not in both simulation and physical display. Matching SW3-0 and SW7-4 to *counter1* and *counter2* respectively is probably better.

However, it’s better to make both values random using the code above when considering the real application scenarios.

**Discussion II**

Hex(4) is darker than other hex displays when rolling. Now we list the correspondence of each dice digit to hex(i):

Hex(6) lights: 2, 3, 4, 6

Hex(5) lights: 4, 5, 6

Hex(4) lights: 2, 6

Hex(3) lights: 2, 3, 5, 6

Hex(2) lights: 1, 3, 4, 5, 6

Hex(1) lights: 1, 2, 3, 4

Hex(0) lights: 2, 3, 5, 6

Notice that hex(4) owns the least numbers. Due to the pseudo-random algorithm (looping from 1 to 6 infinitely) mentioned above, hex(4) is indeed darker than other hex bits. This proves the darkness of hex(4) is not a fault.

**Discussion III**

Although the physical display is passed, the code has some shortages still. Look at this:

*counter1\_save* := *counter1;*

*counter2\_save* := *counter2;*

The code above is in the *win* and *lose* state part. The initial purpose is to lock the value of two counters when winning or losing the game, and to display both locked value on two hexes. This is totally correct in high clock frequency because before the counter value delay happened, the clock rising edge has refreshed the counter state. However, if the clock frequency is low, the *counter* value is firstly copied to *counter\_save*, yet the value of *counter* would change due to counter delay.

When simulating, the error above has happened once, i.e., although the display is 1 and 5, the simulation result is winning because the actual value is 1 and 6. However, in real application scenarios, the error could never happen, and thus we can ignore the error in the simulation.

**Conclusion:**

In the lab exercise, a dice game was achieved using VHDL, featuring pseudo-random dice rolls and win/lose conditions based on the sum of two dices. Challenges include managing clock speed for observable randomness and addressing display inconsistencies due to segment usage display. Minor simulation faults happened, but it did not affect physical performance. Overall, the experiment validated the design's functionality, demonstrating effective state management and output control in response to user inputs.