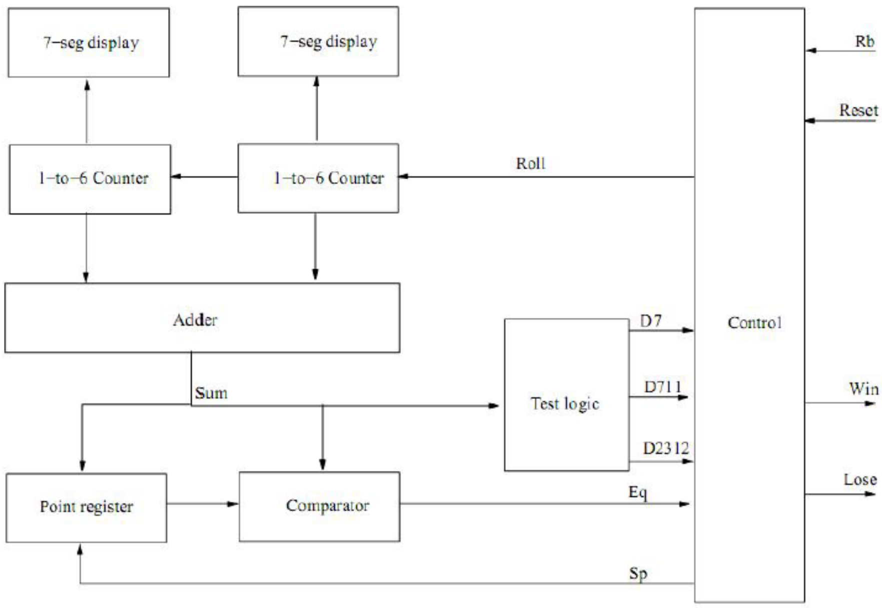
LAB REPORT
Laboratory exercise 7: A Dice Game

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.

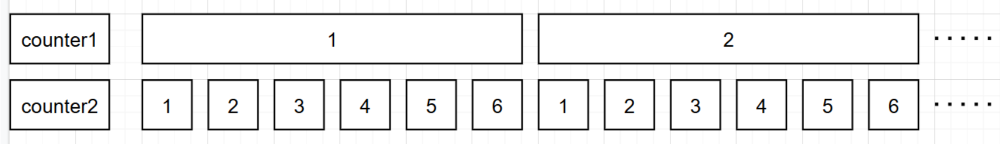
2. Design Formulation

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:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity part7 is
    port (
        clk : in std_logic;
        reset : in std_logic;
        roll : in std_logic;
        seg1 : out std_logic_vector(6 downto 0);
        seg2 : out std_logic_vector(6 downto 0);
        win : out std_logic;
        lose : out std_logic
    );
end part7;
```

Definition of variables in *signal* form:

```vhdl
architecture behavioral of part7 is
    signal point : unsigned(3 downto 0); -- used to save the sum of two dices
    type state_type is (idle, rolling, other_check, win_state, lose_state);
    signal state : state_type := idle;
    signal counter2_wrap : std_logic := '0'; -- flag to indicate counter2 has wrapped
    signal prev_roll : std_logic := '0'; -- previous roll value
    signal first_check_done : std_logic := '0'; -- flag to judge first check or other checks
```

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

```
    function bin_to_hex(bin: unsigned(3 downto 0)) return std_logic_vector is
        variable hex: std_logic_vector(6 downto 0);
    begin
        case bin is
            when "0001" => hex := "1111001"; -- 1
            when "0010" => hex := "0100100"; -- 2
            when "0011" => hex := "0110000"; -- 3
            when "0100" => hex := "0011001"; -- 4
            when "0101" => hex := "0010010"; -- 5
            when "0110" => hex := "0000010"; -- 6
            when others => hex := "1111111";
        end case;
        return hex;
    end function;
```

Definition of variables in *variable* form:

```
begin

process(clk, reset)
    variable counter1 : unsigned(3 downto 0) := "0001";
    variable counter2 : unsigned(3 downto 0) := "0001";
    variable counter1_save : unsigned(3 downto 0) := "0001"; -- save value of counter1 when game over
    variable counter2_save : unsigned(3 downto 0) := "0001"; -- save value of counter2 when game over
    variable finish : std_logic := '0';
```

Reset initialization:

```
begin
    if reset = '1' then
        counter1 := "0001";
        counter2 := "0001";
        counter1_save := "0001";
        counter2_save := "0001";
        counter2_wrap <= '0';
        state <= idle;
        prev_roll <= '0';
        first_check_done <= '0';
        win <= '0';
        lose <= '0';
        seg1 <= bin_to_hex("0001");
        seg2 <= bin_to_hex("0001");
        finish := '0';
```

Idle state:

```
    elsif rising_edge(clk) then
        -- update the previous roll state
        prev_roll <= roll;

        case state is
            when idle =>
                if roll = '1' and finish = '0' then
                    state <= rolling;
                elsif finish = '1' then
                    seg1 <= bin_to_hex(counter1_save);
                    seg2 <= bin_to_hex(counter2_save);
                end if;
```

Counter1 and *counter2* rotation:

```
            when rolling =>
                -- counter1 and counter2 rotation
                if roll = '1' then
                    if to_integer(counter2) < 5 then
                        counter2 := counter2 + 1;
                        counter2_wrap <= '0';
                    elsif to_integer(counter2) = 5 then
                        counter2 := counter2 + 1;
                        counter2_wrap <= '1';
```

```
    else
        counter2 := "0001";
        counter2_wrap <= '0';
    end if;

    if counter2_wrap = '1' then
        counter1 := counter1 + 1;
        counter2_wrap <= '0';
        if to_integer(counter1) > 5 then
            counter1 := "0001";
        end if;
    end if;

    -- hex display
    seg1 <= bin_to_hex(counter1);
    seg2 <= bin_to_hex(counter2);
end if;
```

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

```
-- when falling edge of roll, start judgement
if roll = '0' and prev_roll = '1' then
    -- first check
    if first_check_done = '0' then
        if to_integer(counter1) + to_integer(counter2) = 7 or
           to_integer(counter1) + to_integer(counter2) = 11 then
            state <= win_state;
        elsif to_integer(counter1) + to_integer(counter2) = 2 or
              to_integer(counter1) + to_integer(counter2) = 3 or
              to_integer(counter1) + to_integer(counter2) = 12 then
            state <= lose_state;
        else
            point <= counter1 + counter2;
            state <= other_check;
            first_check_done <= '1';
        end if;
    -- other checks
    elsif first_check_done = '1' then
        if to_integer(counter1 + counter2) = to_integer(point) then
            state <= win_state;
        elsif to_integer(counter1 + counter2) = 7 then
            state <= lose_state;
        else
            state <= other_check;
        end if;
    end if;
end if;
```

Other-check state, just for hex display:

```
when other_check =>
    seg1 <= bin_to_hex(counter1);
    seg2 <= bin_to_hex(counter2);

    if roll = '1' then
        state <= rolling;
    end if;
```

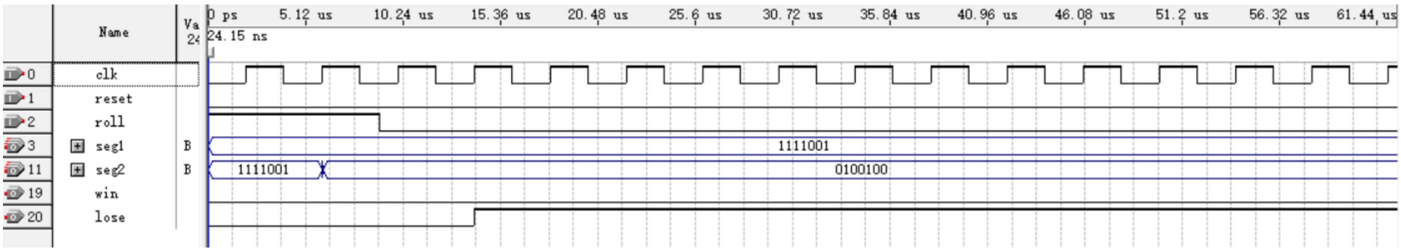
Win and *lose* state:

```
when win_state =>
    win <= '1';
    state <= idle;
    first_check_done <= '0';
    finish := '1';
    counter1_save := counter1;
    counter2_save := counter2;
```

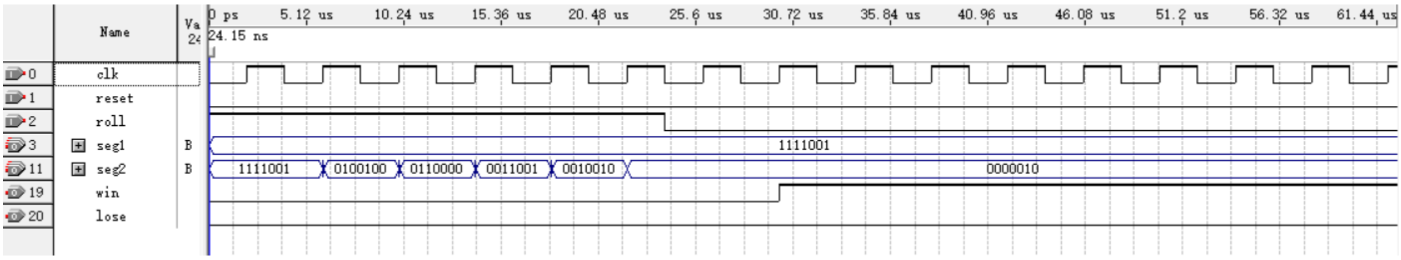
```
                when lose_state =>
                    lose <= '1';
                    state <= idle;
                    first_check_done <= '0';
                    finish := '1';
                    counter1_save := counter1;
                    counter2_save := counter2;
            end case;
        end if;
end process;

end behavioral;
```

4. Simulation and Synthesis Results:

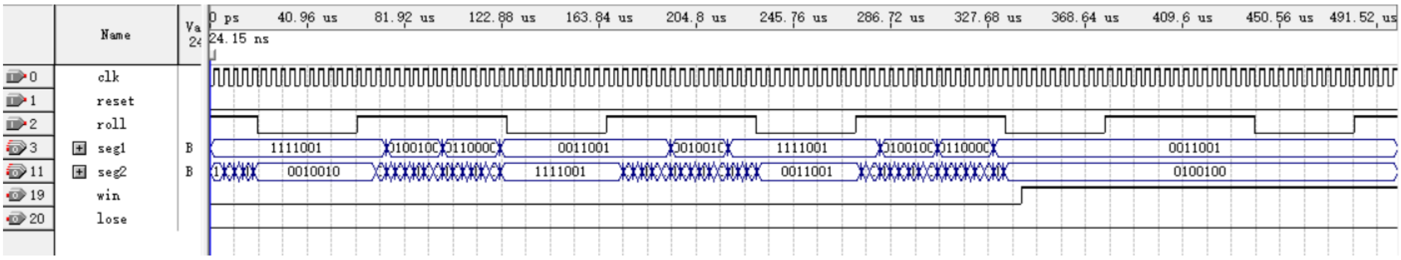
Win state of the first falling edge: *counter1* = 1, *counter2* = 2.



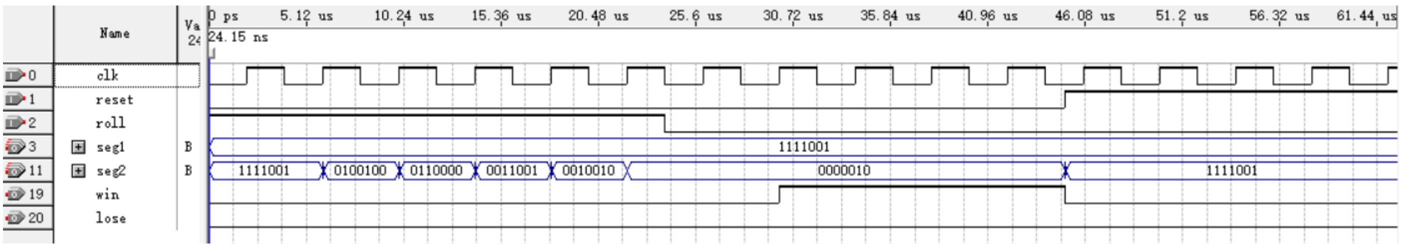
Lose state of the first falling edge: *counter1* = 1, *counter2* = 6.



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



Reset:

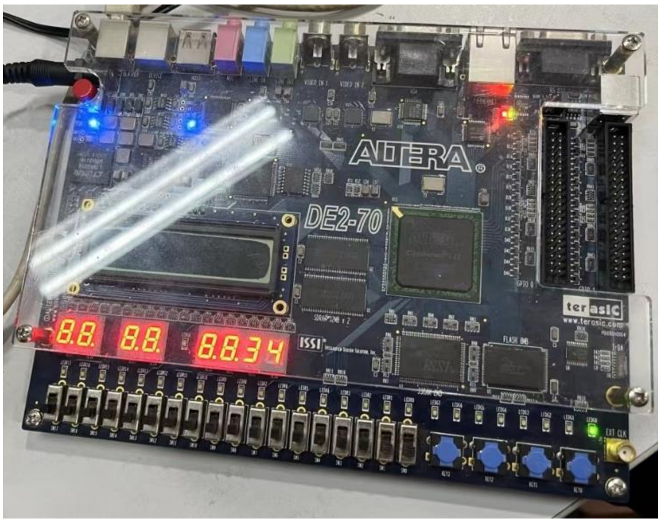


5. Experimental Results

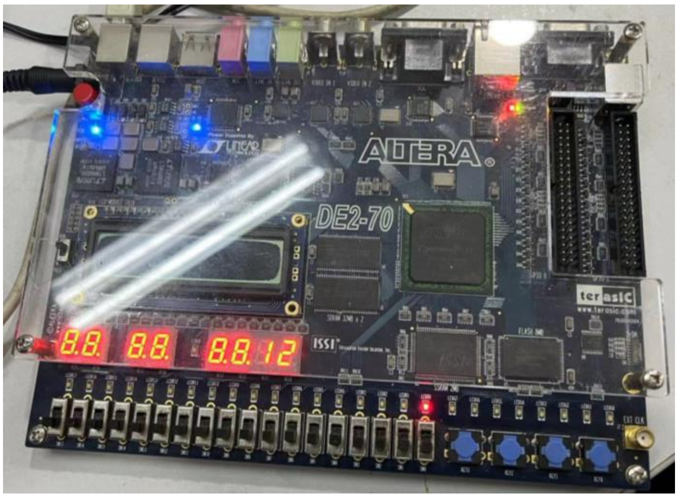
Note: (1) Key press KEY1 is unavailable, and thus use SW0 and SW1 to substitute KEY0 and KEY1.

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

Win state of the first falling edge:

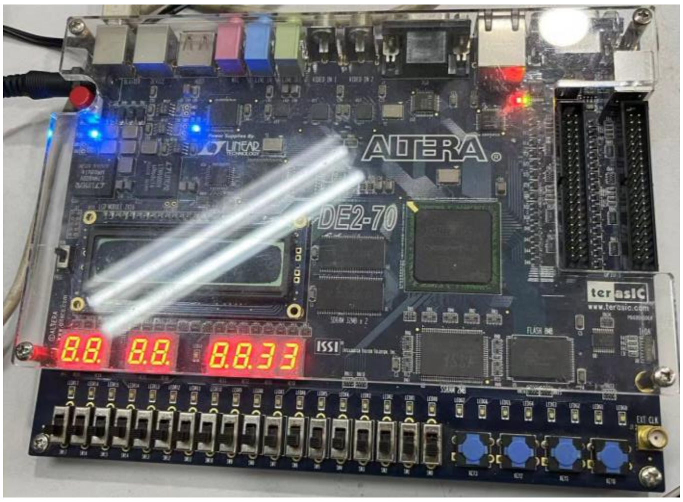


Lose state of the first falling edge:

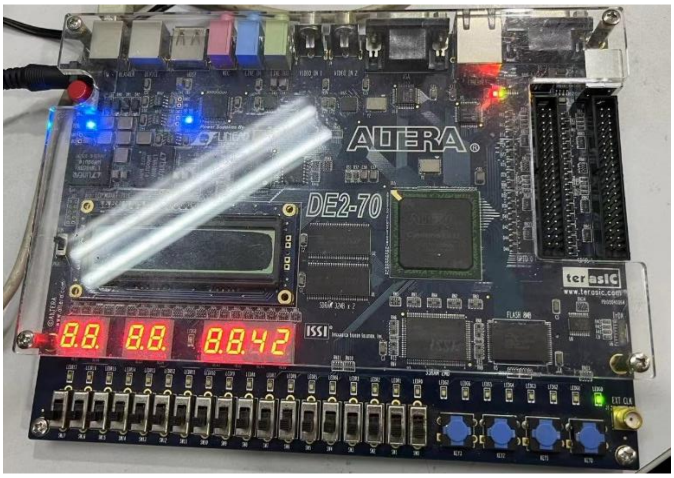


Win state of not the first falling edge:

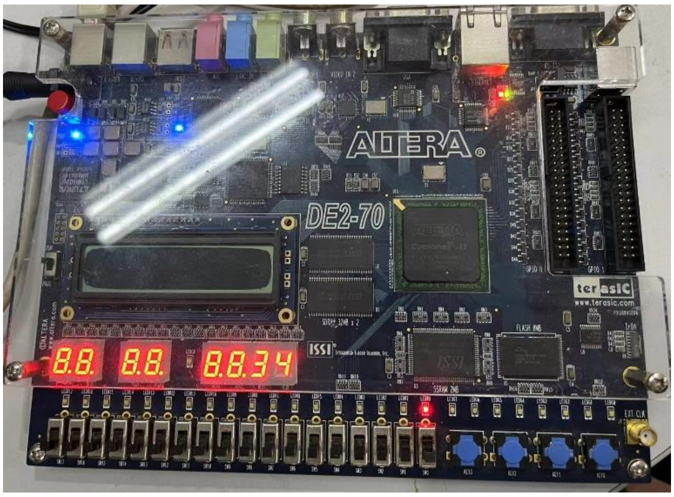
First try:



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.