Implementation of a Three-digit Decimal Counter

Wang Zhuoyang[1]

[1] *12112907, Department of Electrical and Electronic Engineering, SUSTech. Email: glverfer@outlook.com*

Abstract

Lazy me. No abstract.

Keywords: Decimal Counter; VHDL

Contents

1 Introduction

The purpose of this experiment is to create a three-digit decimal counter, with each digit represented by a four-bit BCD code for decimal numbers from 0 to 9. The counter should have the following functions:

1. Synchronous clock input (clk).
2. Reset input (rst).
3. Enable input (count).
4. Load mode selection input, along with corresponding data input (three BCD codes representing digits, totaling 12 bits).
5. A set of data outputs (three BCD codes representing digits, totaling 12 bits).
6. Count direction selection input (countup).

As stated above, these functions are encapsulated in a module **three_digit_decimal_counter** with pins represented as shown in Figure 1. Based on this, a complete peripheral circuit is established with the following requirements:

Digital System Designing Laboratory Reports (2023)

Corresponding author
Wang Zhuoyang

1. The system uses the onboard 100MHz clock signal, and the counter is required to count once per second.
2. Connect the load signal and the countup signal to the switches (with the aid of indicator LEDs).
3. Connect the 12-bit load mode data input to other switches.
4. Connect the rst signal to a button.
5. Display the current three-digit number of the counter on the seven-segment display (originally required to use LED display).

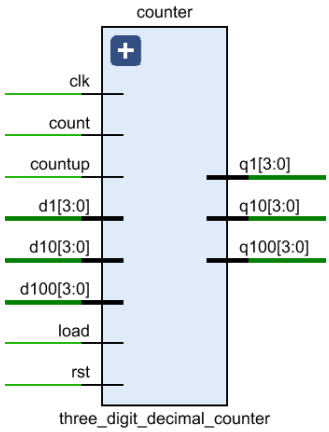

Figure 1. The model and I/O descriptions for the 3-digit decimal counter.

2 Pre-lab Exercises

The requirements mention that there is only a 100MHz clock source on the board, and the counting signal frequency is 1Hz. Therefore, a frequency divider module called **freq_divider** is required. We hope that this module can output a one-cycle-length pulse every 100 million system clock cycles, which will serve as the enable signal (count) for the counter rather than as a clock. The specific reason can be found in the experiment requirements documentation.

2.1 The Conceptual Diagram

The frequency divider can be implemented using a simple sequential logic circuit, as shown in the conceptual diagram in Figure 2.

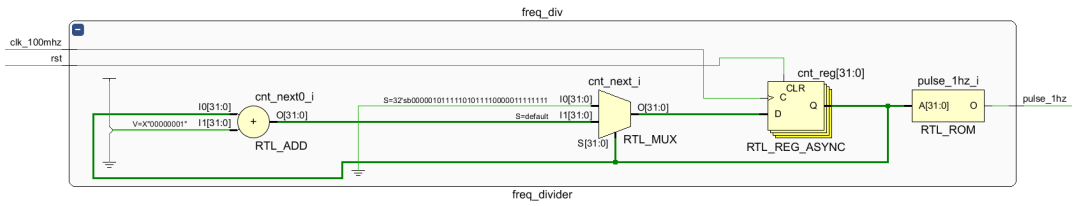

Figure 2. The conceptual diagram of the frequency divider. The 32-bit binary codes represent 99,999,999, i.e. 100M - 1.

2.2 The VHDL Code

And here is the VHDL Implementation for the module (Code 1).

```
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.std_logic_arith.all;
```

```vhdl
use IEEE.std_logic_unsigned.all;

entity freq_divider is
    port (
        clk_100mhz, rst: in std_logic;
        pulse_1hz: out std_logic
    );
end freq_divider;

architecture Behavioral of freq_divider is
    constant cnt_max: integer := 100000000;
    signal cnt, cnt_next: integer;
begin
    process (clk_100mhz, rst) is
    begin
        if rst = '1' then
            cnt <= 0;
        elsif clk_100mhz'event and clk_100mhz = '1' then
            cnt <= cnt_next;
        end if;
    end process;
    cnt_next <= 0 when cnt = (cnt_max - 1) else cnt + 1;
    pulse_1hz <= '1' when cnt = (cnt_max - 1) else '0';
end Behavioral;
```

Listing 1. The frequency divider (1Hz pulse generator).

3 Design and Implementation

3.1 Frequency Divider (1Hz Pulse Generator)

The description of the frequency divider can be found in the Pre-lab Exercises and will not be repeated here.

3.2 Three-digit Decimal Counter

Similarly, the conceptual diagram of the three-digit decimal adder module is as follows (Figure 1).

And here is the code (Code 2) for the three-digit decimal adder module.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity three_digit_decimal_counter is
    port (
        clk, rst, count: in std_logic;
        load: in std_logic;                                    -- synchronous
    load
        countup: in std_logic;
        d100, d10, d1: in std_logic_vector(3 downto 0);        -- data in
        q100, q10, q1: out std_logic_vector(3 downto 0)        -- values of
    registers
    );
end three_digit_decimal_counter;

architecture Behavioral of three_digit_decimal_counter is
    signal r100, r10, r1: std_logic_vector(3 downto 0);
    signal r100_next, r10_next, r1_next: std_logic_vector(3 downto 0);
begin
    process (clk, rst)
    begin
```

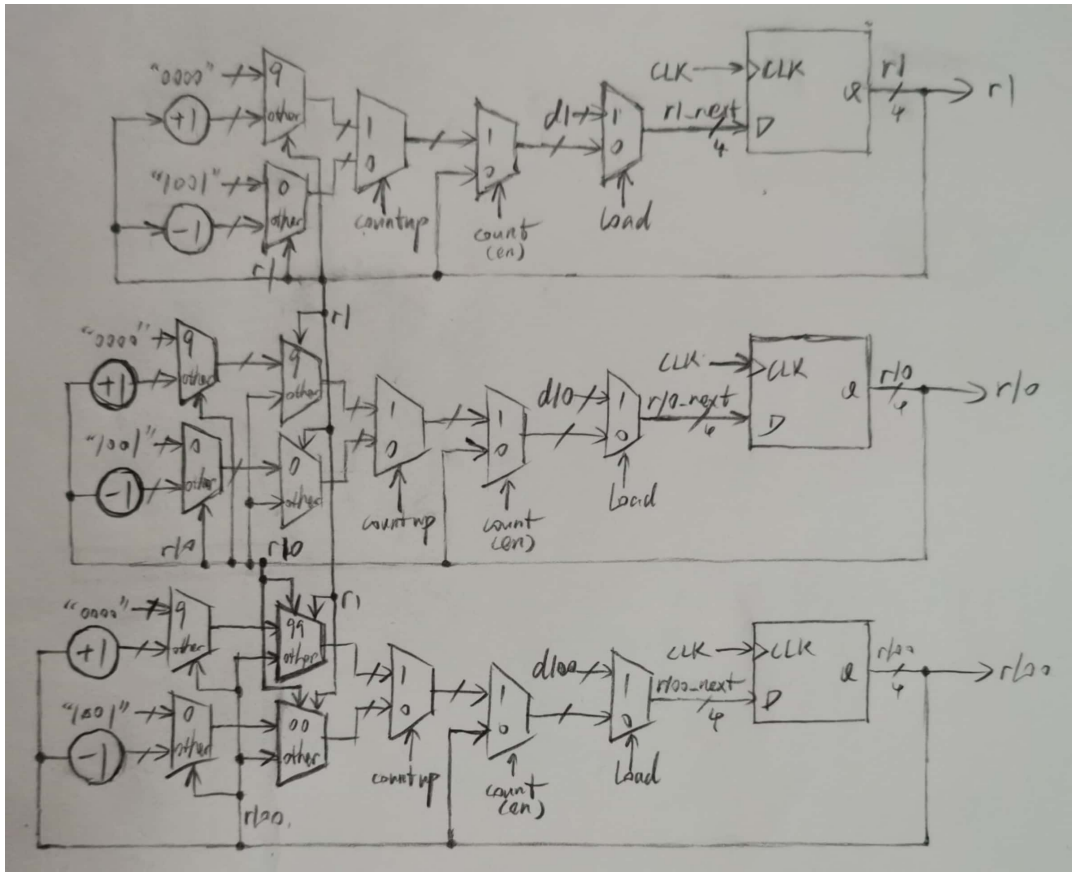

Figure 3. The conceptual diagram for the 3-digit decimal counter module.

```
22          if rst = '1' then
23              r100 <= "0000";
24              r10 <= "0000";
25              r1 <= "0000";
26          elsif clk'event and clk = '1' then
27              r100 <= r100_next;
28              r10 <= r10_next;
29              r1 <= r1_next;
30          end if;
31      end process;
32
33      r1_next <= d1           when load = '1' else
34                 r1           when count = '0' else
35                 "0000"       when countup = '1' and r1 = 9 else
36                 "1001"       when countup = '0' and r1 = 0 else
37                 r1 + 1       when countup = '1' else
38                 r1 - 1       when countup = '0' else
39                 r1;
40
41      r10_next <= d10         when load = '1' else
42                  r10         when count = '0' else
43                  "0000"      when countup = '1' and r10 = 9 and r1 = 9 else
44                  "1001"      when countup = '0' and r10 = 0 and r1 = 0 else
45                  r10 + 1     when countup = '1' and r1 = 9 else
46                  r10 - 1     when countup = '0' and r1 = 0 else
47                  r10;
48
49      r100_next <= d100       when load = '1' else
```

```
                r100        when count = '0' else
                "0000"      when countup = '1' and r100 = 9 and r10 = 9 and
    r1 = 9 else
                "1001"      when countup = '0' and r100 = 0 and r10 = 0 and
    r1 = 0 else
                r100 + 1    when countup = '1' and r10 = 9 and r1 = 9 else
                r100 - 1    when countup = '0' and r10 = 0 and r1 = 0 else
                r100;

    q100 <= r100;
    q10 <= r10;
    q1 <= r1;
end Behavioral;
```

Listing 2. The three-digit decimal counter.

3.3 Seven-segment Display Controller

The seven-segment display control module will also be used in the next experiment and will not be detailed here.

3.4 The Overall Design

We can draw the conceptual diagram of the entire top-level module (Figure 4).

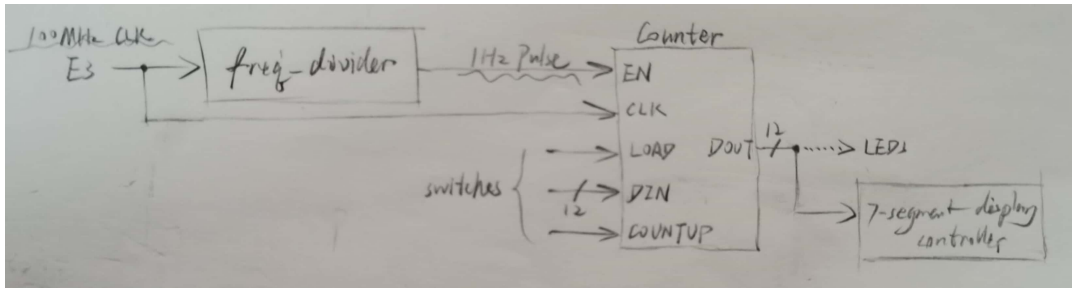

Figure 4. The top level structural diagram.

And the code is as below (Code 3).

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity top_module is
    port (
        clk_100mhz, rst: in std_logic;
        load, count_mode: in std_logic;
        data_in: in std_logic_vector(11 downto 0);
        load_led, count_mode_led: out std_logic;
        an: out std_logic_vector(7 downto 0);
        cathodes: out std_logic_vector(0 to 7)
    );
end top_module;

architecture Behavioral of top_module is
    component freq_divider is
        port (
            clk_100mhz, rst: in std_logic;
            pulse_1hz: out std_logic
```

```
        );
    end component;

    component three_digit_decimal_counter is
        port (
            clk, rst, count: in std_logic;
            load: in std_logic;                                    -- synchronous load
            countup: in std_logic;
            d100, d10, d1: in std_logic_vector(3 downto 0);      -- data in
            q100, q10, q1: out std_logic_vector(3 downto 0)      -- values of registers
        );
    end component;

    component seven_segments_display_decoder is
        port (
            bcd: in std_logic_vector(3 downto 0);
            en_dp: in std_logic;
            segs_n: out std_logic_vector(0 to 7)                -- 0 to 7: dp, a, b, c, d, e, f, g
        );
    end component;

    constant en_dp: std_logic := '0';

    signal pulse_en: std_logic;
    signal bcd100, bcd10, bcd1: std_logic_vector(3 downto 0);
    signal seg100, seg10, seg1: std_logic_vector(0 to 7);

    signal delay_cnt, delay_cnt_next: integer range 0 to (100000 - 1);
    signal scan_cnt, scan_cnt_next: integer range 0 to 2;
begin
    freq_div: freq_divider
        port map (
            clk_100mhz => clk_100mhz,
            rst => rst,
            pulse_1hz => pulse_en
        );

    counter: three_digit_decimal_counter
        port map (
            clk => clk_100mhz,
            rst => rst,
            count => pulse_en,
            load => load,
            countup => count_mode,
            d100 => data_in(11 downto 8),
            d10 => data_in(7 downto 4),
            d1 => data_in(3 downto 0),
            q100 => bcd100,
            q10 => bcd10,
            q1 => bcd1
        );
    load_led <= load;
    count_mode_led <= count_mode;

    seg100_disp: seven_segments_display_decoder port map (bcd => bcd100, en_dp => en_dp, segs_n => seg100);
    seg10_disp: seven_segments_display_decoder port map (bcd => bcd10, en_dp => en_dp, segs_n => seg10);
    seg1_disp: seven_segments_display_decoder port map (bcd => bcd1, en_dp
```

```
    => en_dp, segs_n => seg1);

    process (clk_100mhz, rst) is
    begin
        if rst = '1' then
            delay_cnt <= 0;
            scan_cnt <= 0;
        elsif clk_100mhz'event and clk_100mhz = '1' then
            delay_cnt <= delay_cnt_next;
            scan_cnt <= scan_cnt_next;
        end if;
    end process;

    delay_cnt_next <= delay_cnt + 1;
    scan_cnt_next <= scan_cnt + 1 when delay_cnt = 0 else scan_cnt;

    an(7 downto 3) <= (others => '1');
    an(2) <= '0' when scan_cnt = 2 else '1';
    an(1) <= '0' when scan_cnt = 1 else '1';
    an(0) <= '0' when scan_cnt = 0 else '1';
    cathodes <= seg100 when scan_cnt = 2 else
                seg10 when scan_cnt = 1 else
                seg1;
end Behavioral;
```

Listing 3. The top module.

4 Simulation

Now we can write a testbench to perform Post-Implementation Timing Simulation on the counter module. And here is the simulation result (Figure 5).

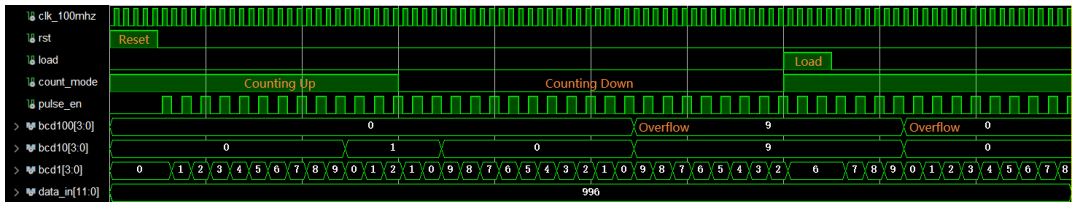

Figure 5. The result of the Post-Implementation Timing Simulation. The diagram contains different inputs and they are all commented on the figure.

We can see that it includes reset, count up, count down, overflow, underflow, and read data.

5 Program and Run

Then we program the board to test our design. First, when we press reset button (N17) the counter will be reseted to zero (Figure 6).

And this time we turn on the first switch (V10) which is connected to the countup signal, which refers to the counting up mode (the LED V11 will be lit up). Then we can see the number growing (Figure 7, 8 and 9).

When we turn on the second switch (U11), the LED V12 will also be lit up, while the counter will enter the Load Mode. The data '1001 1001 0110', i.e., 996 will be loaded into the counter (Figure 10).

As the counting continues, the numbers will increase up to 999 and then overflow and jump back to 0 (Figure 11, 12 and 13).

Also, we can change to the counting down mode, which is omitted here.

6 Conclusion

First let us summarize the lab content. In this lab, we built a three-digit decimal counter with functions such as reset, read, counting mode selection, and enable. Additionally, we implemented a frequency divider to generate counting pulses that are easily recognizable by the human eye. Furthermore, we implemented a seven-segment display, eliminating the hassle of binary observation.

Through this experiment, we have gained the following insights:

- Sequential logic or state machines must be written in standard two-section format, otherwise it is very easy to make mistakes. That is, the state transition occupies one section and the combinational logic starts a new section.
- Synchronous timing circuits are best synchronized with a global clock and use enable signals to transmit signals and actions, which is more standardized.

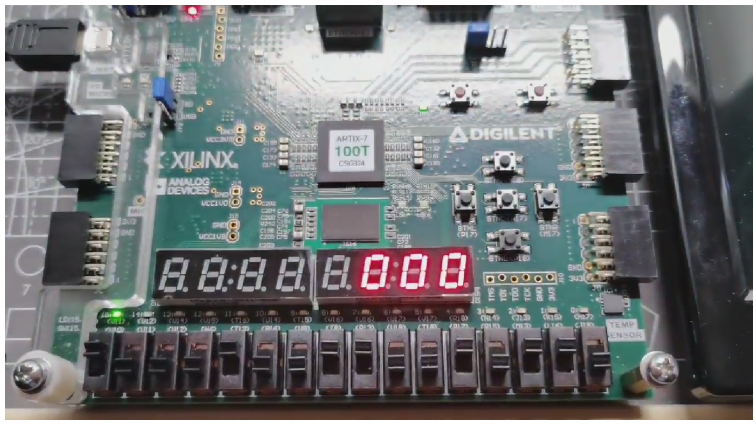

Figure 6. After reset.

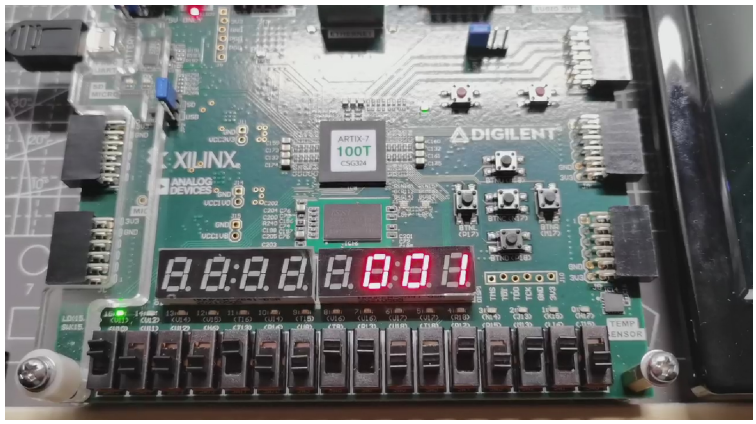

Figure 7. The first second after reset.

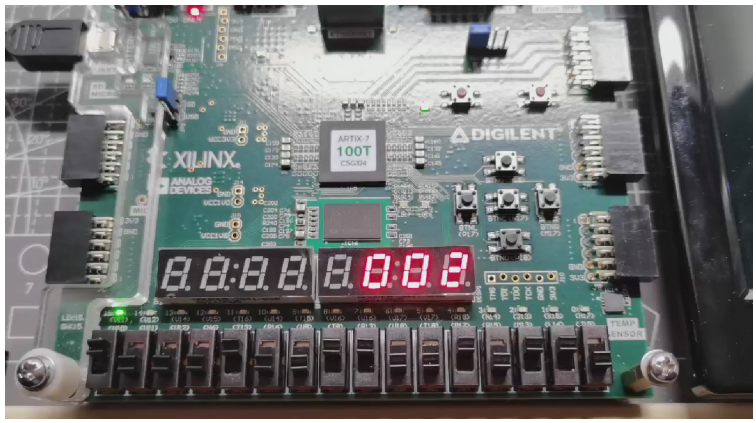

Figure 8. The second second after reset.

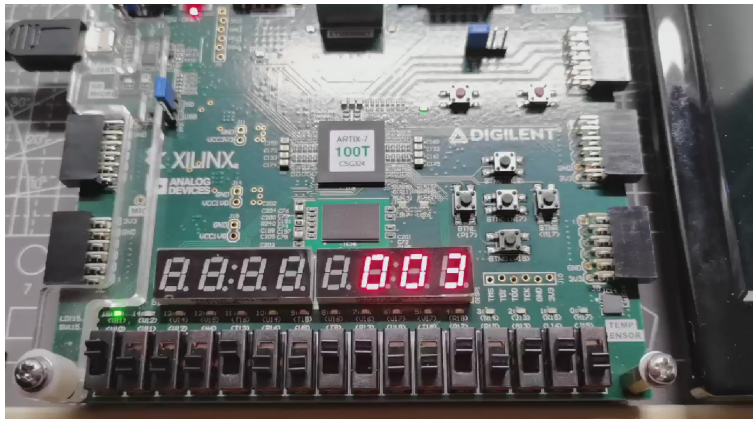

Figure 9. The third second after reset.

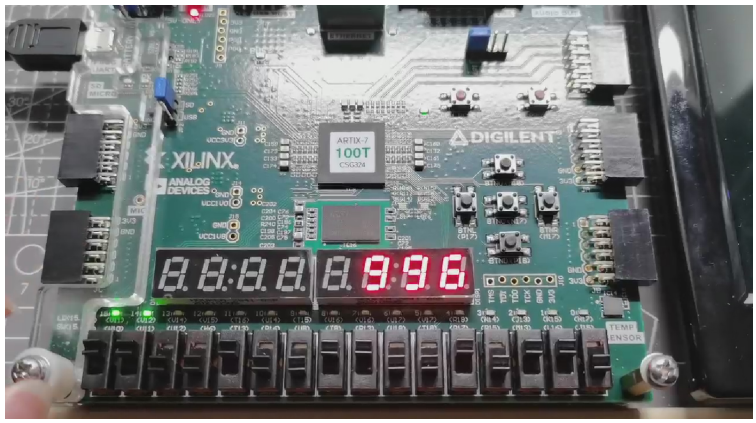

Figure 10. The Load Mode.

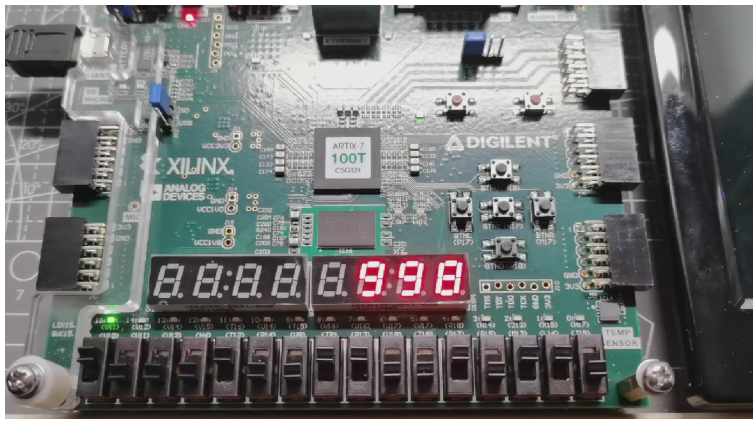

Figure 11. Two second before overflowing.

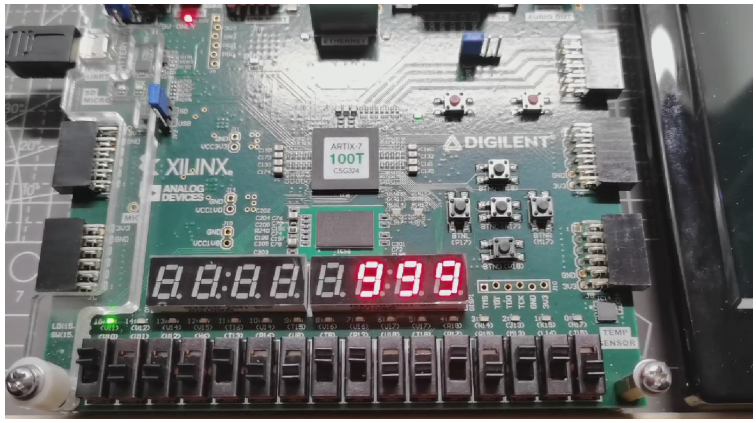

Figure 12. One second before overflowing.

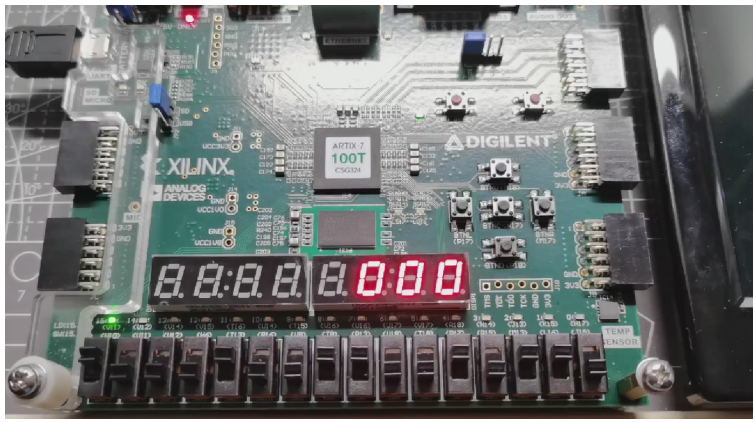

Figure 13. Overflow.