Flashing Displays

Toki Nishikawa

October 25 2022
6-8pm Tuesday lab with Kim Nguyen/Nick Boudreau

The objective of the lab is to display a 2-digit number in base 10 on a seven-segment display.

1 Design

The first step in the design process is to build a VHDL module that alternately flashes two LEDs. A schematic of the top-level module is shown in **Figure 1**. The HSOSC module represents the high speed oscillator within the iCE40 FPGA and produces a 48MHz clock signal. The 'counter logic' is designed to increment a 26-bit binary number, called *a* in the code, by one on the rising edge of the clock. When it reaches its maximum value, it will roll over to zero and start again. The final code for the top-module can be found in **Section 5** – its entity name is *lab6*.

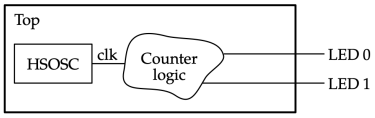

Figure 1: Schematic of top-level module with HSOSC module and counter logic

Since the counter rolls over, the most significant bit will alternate between zero and one. This is the mechanism we will use to simultaneously display two independently-controlled digits on the seven-segment display. Since the two digits share the cathode wires, we are unable to keep both of them on and control them independently. However, we can display one digit at a time and alternate between them really fast so that it looks like they are both on. We will achieve this by wiring up the anode pins to outputs that switch between a high and low voltage at a frequency set by our counter. The corresponding output in the code is *anode*, which is a 2-bit standard logic vector. Each bit will be wired up to each anode on the seven-segment display, and so *anode* will alternate between '01' and '10'. The frequency of the alternating bits is dependent on which bit we select from the 26-bit binary number, with less significant bits yielding a higher frequency. Thus, in order to identify an appropriate flashing frequency for our seven-segment display, we will use a bit that alternates quickly enough so that our eyes cannot see any flashing but slowly enough so that we minimize energy costs.

The next step is to create a dual-decimal-digit-driver (DDDD) module that will take a 6-bit unsigned integer and produces two 7-bit outputs corresponding to the two digits on the seven-segment display. A schematic of the DDDD module is shown in **Figure 2**.

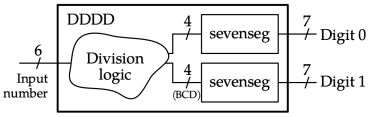

Figure 2: Schematic of DDDD module

This module instantiates a sevenseg module – the code for the sevenseg module and the DDDD module can be found in **Section 5**. The 'division logic' in **Figure 2** refers to converting the 6-bit unsigned integer into two 4-bit unsigned integers, one representing the tens place, *digit1_bcd* in the code, and the other representing the ones place, *digit0_bcd* in the code. The naming 'bcd' stands for binary coded decimal, indicating that we are representing a decimal number with two binary numbers. The largest integer that can be obtained with a 6-bit binary number is 63, and so our seven-segment display should be able to display all integers in the interval 0 to 63. The signal *digit0_bcd* is obtained using the mod operator. The input number, *count* in the code, mod 10 will yield the ones place. In order to obtain *digit1_bcd*, we need to divide *count* by 10. However, dividing by factors other than 2 is challenging, and thus we will instead multiply by 52 and divide by 512 which is equivalent to dividing by 10.16. The signals *digit0_bcd* and *digit1_bcd* are then inputted into the sevenseg module which outputs two 7-bit standard logic vectors, *digit0* and *digit1*, respectively.

Next, we will instantiate the DDDD in our top module according to the schematic shown in **Figure 3**. The input to our top-module will be a 6-bit unsigned integer, *dip_value* in the code, which is port-mapped to *count*, the input to our DDDD module. We will use a DIP switch to specify the input number, although we could have used any number generation mechanism.

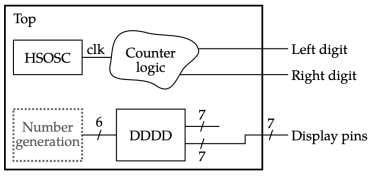

Figure 3: Schematic of top-module with DDDD

Now, we must integrate the counter logic with the outputs of the DDDD in order to alternate the two digits on the seven-segment display. We do this using a 2:1 multiplexer as shown in **Figure 4**. If we take a look at the code for our top-module, there are two outputs, a 7-bit standard logic vector named *sevenseg* and *anode* which we mentioned earlier. The output *sevenseg* is mapped to the pins that will be wired to the cathode pins of the seven segment display. The output *anode* is mapped to the pins that will be wired to the anode pins of the seven segment display. More specifically, the most significant bit of *anode* is mapped to the anode pin controlling the tens place digit while the least significant bit is mapped to the anode pin controlling the ones place digit.

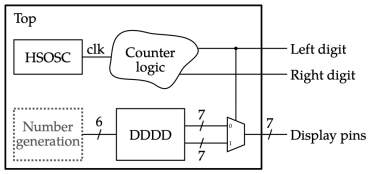

Figure 4: Schematic of final top-module

We will assign signal *digit0_select*, which is mapped to *digit0*, to *sevenseg* when the nineteenth bit of a is 0 and signal *digit1_select*, which is mapped to *digit1*, to *sevenseg* when the nineteenth bit of a is 1. Thus, we want *anode* to have the value '01' when the nineteenth bit of a is 0 and '10' when the the nineteenth bit of a is 1. The choice of the nineteenth bit will be explained in **Section 3**.

2 Implementation

Recall that the first step in our design process was to create a top module that includes the counter logic. It is a good idea to test that the counter logic is functioning correctly before incorporating the DDDD module. Thus, we will flash the design corresponding to **Figure 1** onto our FPGA and test it by connecting the two outputs to LEDs. These outputs correspond to the signals *bit_iszero* and *bit_isone* in the final top-level module code, and they indicate whether a specified bit in a is 1 or 0. In order to test the counter's functionality, we will turn one LED on when the most significant bit of a is 0 and the other LED on when it is 1. The resulting circuit is shown in **Figure 5**. Note that the DIP switch and pushbuttons are not a part of this circuit. The LEDs should alternate with a period of 1.3980 seconds since the HSOSC module produces a 48MHz clock signal and 1 period corresponds to 2^{26} signals.

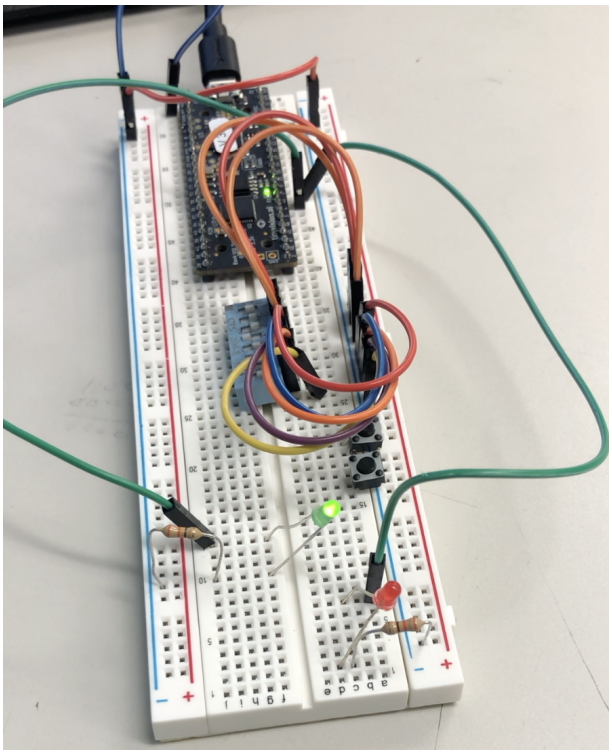

Figure 5: Circuit to test counter logic

In order to configure the *dip_value* input, we will use a DIP switch. Note that we don't need to use pull-up resistors since these are already included in the FPGA. The *sevenseg* and *anode* outputs should be connected to the seven-segment display according to the schematic shown in **Figure 6**.

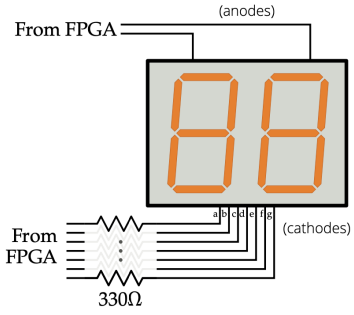

Figure 6: Schematic of seven-segment display

Figure 7 shows the completed circuit.

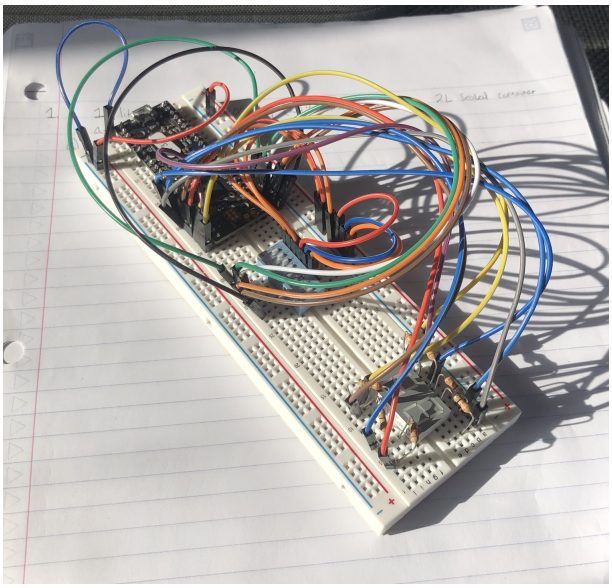

Figure 7: Completed circuit

3 Testing

3.1 Testing counter logic

The first step in the implementation process was to construct a circuit to test the counter logic. As mentioned earlier, we started with just two outputs (which are now signals), *bit_iszero* and *bit_isone*, that describe the state of the most significant bit in a. We also calculated the theoretical period of the flashing LEDs to be 1.3980s. In order to make sure that our circuit is performing as expected, we will time the period of the flashing LEDs with a stopwatch. The results are displayed in **Table 1**.

The average of the results in **Table 1** is 1.304s, which is very close to our theoretical period 1.398s. With that said, our strategy of alternating between the two digits on the seven-segment display will not work very well if we stick with a period of 1.398s. Consequently, we must reduce the period until we can no longer tell that each digit is flashing on and off, and it looks as if it is displaying two digits simultaneously. Recall that the frequency increases as we consider less significant bits in a (this should make sense intuitively). Thus, we will revise the bit in question until the the display seems simultaneous. It turns out that this occurs on the 19th bit.

Time (s)
1.30
1.25
1.18
1.25
1.43
1.33
1.41
1.31
1.30
1.28

Table 1: Testing of counter period

3.2 Testing the completed circuit

The completed circuit, shown in **Figure 7**, was bound not to work on the first try since we just wrote a whole bunch of code. The first test is shown in **Table 2** where *dip_value* is the 6-bit input, # is the corresponding value in base 10, *Exp.sgments* are the segments we expect in the order A-G, and *Meas.sgments* are the segments we observed in the order A-G. For the sake of simplicity, a '1' indicates that a segment is ON while a '0' indicates that a segment is OFF, even though we actually send a high voltage to the segments we want OFF. The letters associated with each segment on the seven-segment display are shown in **Figure 8**. Note that the labeling of the segments on the second digit are analogous to that of the first.

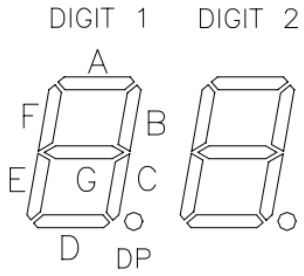

Figure 8: Letters associated with each segment

dip_value	#	Exp.sgmnts(d1)	Meas.sgmnts(d1)	Exp.sgmnts(d2)	Meas.sgmnts(d2)
100010	34	1111001	1100110	0110011	1001111

Table 2: First test of completed circuit

As shown in **Table 2**, neither of the digits were correct. Before checking the code, I decided to make sure that the pin-out was correct, which it was. Then, I went through the code for the DDDD module, which seemed to be okay. Finally, I chose to test out some more numbers to see if that would yield any hints. I started with 0, 1, 2, and 3. The results are displayed in **Table 3**.

dip_value	#	Exp.sgmnts(d1)	Meas.sgmnts(d1)	Exp.sgmnts(d2)	Meas.sgmnts(d2)
000000	00	1111110	0111111	1111110	0111111
000001	01	1111110	0000110	0110000	0111111
000010	02	1111110	1011011	1101101	0111111
000011	03	1111110	1001111	1111001	0111111

Table 3: Second test of completed circuit

These results were a lot more helpful. Since all of the *Meas.sgmnts(d2)* values were the same, and all of the numbers contained a zero, Professor Bell and I hypothesized that the digits were flipped. We reversed the order of the digits by switching the bits in the *anode* output for a given state of the counter. Now, *anode* will have the value '01' when the nineteenth bit of a is 0 and '10' when the nineteenth bit of a is 1. In **Section 1**, I explain why this yields the correct result. The new results for the same four tests are shown in **Table 4**.

dip_value	#	Exp.sgmnts(d1)	Meas.sgmnts(d1)	Exp.sgmnts(d2)	Meas.sgmnts(d2)
000000	00	1111110	0111111	1111110	0111111
000001	01	1111110	0111111	0110000	0000110
000010	02	1111110	0111111	1101101	1011011
000011	03	1111110	0111111	1111001	1001111

Table 4: Third test of completed circuit

Pictures of the results summarized in **Table 4** are shown in **Figures 9-12** below. From the '00' test, Professor Bell and I concluded that segments A and G were switched, and from the '01', '02', and '03' tests, we concluded that segments B and F were switched and segments C and E were switched. From most significant bit to least significant bit, the *sevenseg* output is ordered A through G, meaning *sevenseg(6)*, for example, should be connected to A. However, all of the cathode connections were reversed, that is, *sevenseg(6)* was mapped to G, etc. After altering the pin-out, all of the previous tests worked, along with numerous other tests, all of which are displayed in **Table 5**.

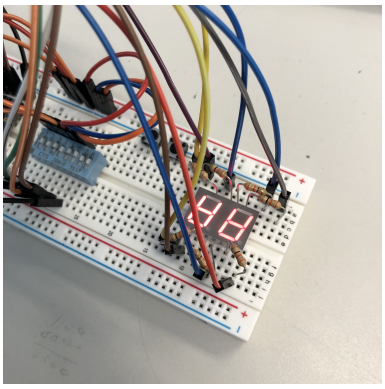

Figure 9: 00 test from **Table 4**

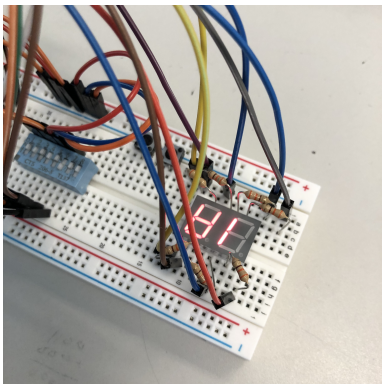

Figure 10: 01 test from **Table 4**

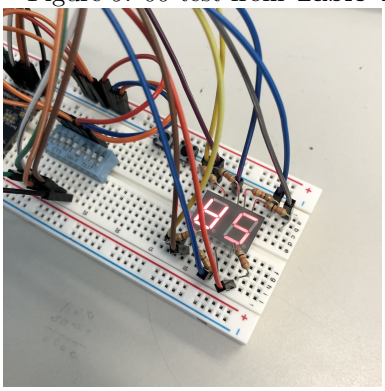

Figure 11: 02 test from **Table 4**

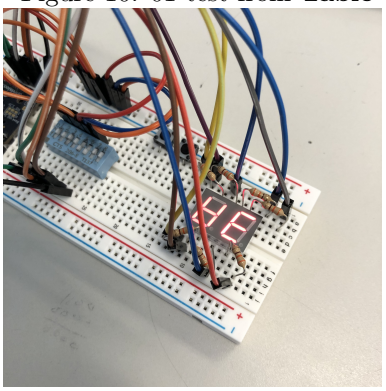

Figure 12: 03 test from **Table 4**

4 Reflection

This lab taught me that there are many creative ways to get more out of the hardware that you have available to you. In this case, we constructed a circuit that seems to output two digits simultaneously, whereas in reality, each digit is just flashing on and off super fast. As a result, we only needed to use one set of cathode pins on the seven-segment display, and by extension, only nine pins on the FPGA (seven cathode plus two anode). Although using an extra seven pins on the FPGA, corresponding to another seven cathode pins for a second digit, may not seem like that big of a deal, suppose we wanted to use ten seven-segment displays. Then, using our trick, we would only have to use seventeen outputs as opposed to eighty, which is a big difference.

I've been struggling with some VHDL syntax, particularly when to use signals. Here is a brief of description of when to use them so that I can refer to it in future projects:

dip_value	#	Exp.sgmnts(d1)	Meas.sgmnts(d1)	Exp.sgmnts(d2)	Meas.sgmnts(d2)
000000	00	1111110	1111110	1111110	1111110
000001	01	1111110	1111110	0110000	0110000
000010	02	1111110	1111110	1101101	1101101
000011	03	1111110	1111110	1111001	1111001
000100	04	1111110	1111110	0110011	0110011
000101	05	1111110	1111110	1011011	1011011
010000	16	0110000	0110000	1011111	1011111
010001	17	0110000	0110000	1110000	1110000
010010	18	0110000	0110000	1111111	1111111
010011	19	0110000	0110000	1111011	1111011
010100	20	1101101	1101101	1111110	1111110
011111	31	1111001	1111001	0110000	0110000
100000	32	1111001	1111001	1101101	1101101
100001	33	1111001	1111001	1111001	1111001
100010	34	1111001	1111001	0110011	0110011
100011	35	1111001	1111001	1011011	1011011
111100	60	1111110	1111110	1111110	1111110
111101	61	1011111	1011111	0110000	0110000
111110	62	1011111	1011111	1101101	1101101
111111	63	1011111	1011111	1111001	1111001

Table 5: Final test of completed circuit

When you instantiate a module, its inputs/outputs can be mapped to the inputs/outputs of the higher-level module, or to a signal that the higher-level module will use in its logic. For example, our top-level module instantiated the DDDD module, which has input *count*, and outputs *digit0* and *digit1*. We want *count* to be directly mapped to the input of our top-level module, *dip_value*, but we want *digit0* and *digit1* to be mapped to signals since we want to use them in combinational logic that will eventually assign something to the outputs of our top-level module.

The lab took ∼5 hours to complete.

List of Tables

5 VHDL code

5.1 Top Module

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity lab6 is
    port(
        dip_value : in unsigned (5 downto 0); -- value set on dip switch
        sevenseg : out std_logic_vector(6 downto 0); -- configuration of sevenseg
        anode : out std_logic_vector(1 downto 0) -- configuration of anode pins
    );
end;

architecture synth of lab6 is
    component HSOSC is
        generic(
         CLKHF_DIV : String := "0b00"
        );
        port(
         CLKHFPU : in std_logic := '1';
         CLKHFEN : in std_logic := '1';
         CLKHF : out std_logic := 'X'
        );
    end component;

    component dddd is
        port(
         count : in unsigned(5 downto 0);
         digit0 : out std_logic_vector(6 downto 0);
         digit1 : out std_logic_vector(6 downto 0)
        );
    end component;

    signal a : unsigned (25 downto 0) := "00000000000000000000000000";
    signal clk : std_logic;
    signal bit_iszero : std_logic := '1';
    signal bit_isone : std_logic := '0';
    signal digit0_select : std_logic_vector(6 downto 0);
    signal digit1_select : std_logic_vector(6 downto 0);
begin
    HSOSC_inst : HSOSC
        port map (
         CLKHF => clk,
```

```
         CLKHFPU => '1',
         CLKHFEN => '1'
        );
    dddd_inst : dddd
        port map(
         count => dip_value,
         digit0 => digit0_select,
         digit1 => digit1_select
        );
    process (clk) begin
        if rising_edge(clk) then
            a <= a + 1;
            if (a(19) = '0') then
                bit_iszero <= '1';
                bit_isone <= '0';
            else
                bit_isone <= '1';
                bit_iszero <= '0';
            end if;
        end if;
    end process;

    sevenseg <= digit0_select when bit_iszero = '1' else digit1_select;
    anode <= "01" when bit_iszero = '1' else "10";

end;
```

5.2 DDDD Module

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity dddd is
    port(
        count : in unsigned(5 downto 0);
        digit0 : out std_logic_vector(6 downto 0);
        digit1 : out std_logic_vector(6 downto 0)
    );
end;

architecture synth of dddd is
    component sevenseg is
        port(
            S : in unsigned(3 downto 0);
            segments : out std_logic_vector(6 downto 0)
```

```
        );
    end component;
    signal digit0_bcd : unsigned (3 downto 0);
    signal digit1_bcd : unsigned (3 downto 0);
    signal intrm : unsigned (12 downto 0); -- intermediate term is 13 bit unsigned
begin
    sevenseg_inst0 : sevenseg
        port map (
            S => digit0_bcd,
            segments => digit0
        );
    sevenseg_inst1 : sevenseg
        port map (
            S => digit1_bcd,
            segments => digit1
        );
    -- mod 10 to find ones place
    digit0_bcd <= count mod 4d"10"; -- ones place is 4 bit unsigned
    -- multiply by 52 and divide by 512 to find tens place
    intrm <= count * 7d"52";
    digit1_bcd <= intrm(12 downto 9); -- tens place is 4 bit unsigned
end;
```

5.3 Seven-segment Module

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity sevenseg is
    port(
        S : in unsigned(3 downto 0);
        segments : out std_logic_vector(6 downto 0)
    );
end sevenseg;

architecture synth of sevenseg is
begin
    process(all) begin
        case S is
            when X"0" => segments <= "0000001";
            when X"1" => segments <= "1001111";
            when X"2" => segments <= "0010010";
            when X"3" => segments <= "0000110";
            when X"4" => segments <= "1001100";
            when X"5" => segments <= "0100100";
```

```
            when X"6" => segments <= "0100000";
            when X"7" => segments <= "0001111";
            when X"8" => segments <= "0000000";
            when X"9" => segments <= "0000100";
            when X"A" => segments <= "0001000";
            when X"B" => segments <= "1100000";
            when X"C" => segments <= "0110001";
            when X"D" => segments <= "1000010";
            when X"E" => segments <= "0110000";
            when X"F" => segments <= "0111000";
            when others => segments <= "1111111";
        end case;
    end process;
end;
```