Università di Pisa

Electronics and Communication Systems
Master Degree in Computer Engineering
year 2018/19

Controller for a 7-segment Display

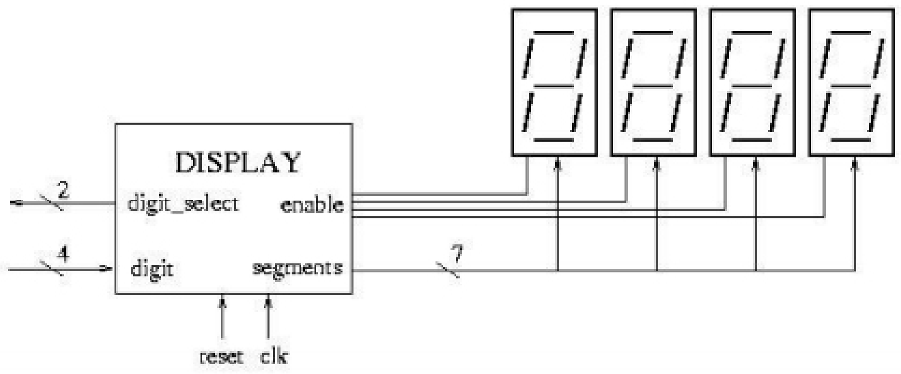
A VHDL implementation and a synthesis by means of Vivado

Riccardo Polini

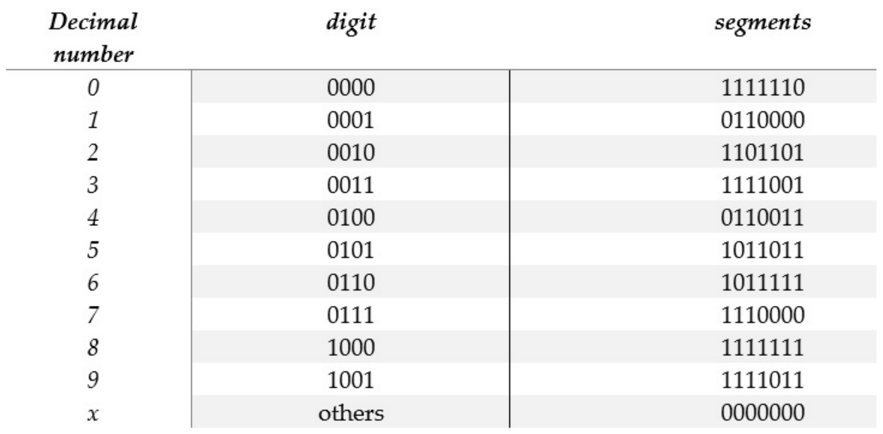
Contents

1 Introduction

The purpose of this project was to implement the description of a controller for a 7-segment display using VHDL code. Furthermore, it was required to synthetize the project by using the Vivado Software.



A user selects which one of the four segments to use through the `digit_select` input and then alters the digit input in order to display the required number. The controller will read the `digit_select` input and correctly enable the required display before setting it through the segments output line. The conversion between the BCD digit and the segments encoding happens according to the following truth table, found in most Electronics textbooks:



Decimal number	*digit*	*segments*
0	0000	1111110
1	0001	0110000
2	0010	1101101
3	0011	1111001
4	0100	0110011
5	0101	1011011
6	0110	1011111
7	0111	1110000
8	1000	1111111
9	1001	1111011
x	others	0000000

As for the enable signal, the truth table is the following:

digit_select	*enable*
00	1000
01	0100
10	0010
11	0001

2 Architecture

The architecture for this system is very simple. The inputs are received and processed according to the truth tables in order to produce the `enable` and the `segments` signals. The network updates its outputs only when the clock signal is on the rising edge and the reset signal is high (1). The final user will be able to attach this controller to a pre-made 7-segment display by just connecting the `enable` signal to the display's voltage port and the `segments` signal to the display's segments ports.

3 Code Description

Declaration of the various inputs/outputs:

```vhdl
entity controller is
    port (
            clk : in std_logic;
            reset : in std_logic;
            digit : in std_logic_vector(3 downto 0);  --BCD input
            digit_select : in std_logic_vector(1 downto 0);
            enable : out std_logic_vector(3 downto 0);
            segments : out std_logic_vector(6 downto 0)  -- 7 bit decoded output.
        );
end controller;
```

Now let's see the behaviour of the controller. A couple of internal signals are used to assign the binary code to the output variables `segments` and `enable`. As mentioned before, this only happens on the rising edge of the clock.

```vhdl
architecture bhv of controller is

signal segments_signal : std_logic_vector(6 downto 0);
signal enable_signal : std_logic_vector(3 downto 0);

begin
    process(clk,reset) is
        begin
                        if(rising_edge(clk) and reset='1') then
                        --modify the signal according to the truth tables
                        case digit is
                         when "0000" => segments_signal <="1111110";
                         when "0001" => segments_signal <= "0110000";
                         when "0010" => segments_signal <= "1101101";
                         when "0011" => segments_signal <= "1111001";
                         when "0100" => segments_signal <= "0110011";
                         when "0101" => segments_signal <= "1011011";
                         when "0110" => segments_signal <= "0011111";
                         when "0111" => segments_signal <= "1110000";
                         when "1000" => segments_signal <= "1111111";
                         when "1001" => segments_signal <= "1110011";
                         when others => segments_signal <="0000000";
                        end case;

                        --update the actual output
                        segments <= segments_signal;

                        case digit_select is
```

```
                         when "00" => enable_signal<="1000";
                         when "01" => enable_signal<="0100";
                         when "10" => enable_signal<="0010";
                         when "11" => enable_signal<="0001";
                         when others => enable_signal<="0000";
                        end case;

                        --update the actual output
                        enable <= enable_signal;
                        end if;
        end process;
end architecture bhv;
```

3.1 Test Plans and Test Bench

In order to properly test for this component, I planned a test which consists in iteratively trying all the possible inputs and check them against the expected outputs.
The initial constants declarations consists in the following:

```
ARCHITECTURE testbench OF controller_tb IS

        --constants declaration
        constant T_CLK   : time := 100 ns;
        constant T_RESET : time := 5 ns;

        -- Simulation time
        constant T_sim   : time := 1000 ns;
        constant N_segments : integer := 7;
        constant N_enable : integer := 4;

        --temporization signals
        signal clk_tb : std_logic := '0';
        signal rst_tb : std_logic := '0';

        --signal to stop the simulation
        signal stop_simulation : std_logic := '1';

        --input signals
           signal digit_tb : std_logic_vector(3 downto 0) := (others => '0');
        signal digit_select_tb : std_logic_vector(1 downto 0);

        --output signals
        signal output_segments_tb : std_logic_vector(6 downto 0);
        signal output_enable_tb : std_logic_vector(3 downto 0);
```

Here is the main body of the `testbench` file:

```
1  begin
2  controller_map:
3          entity work.controller
4          port map (
5                  clk =>clk_tb,
6                  reset =>rst_tb,
7                  digit => digit_tb,
8                  digit_select => digit_select_tb,
9                  enable => output_enable_tb,
10                 segments => output_segments_tb
11         );
12
13    clk_process :process
14    begin
15         rst_tb <= '1' after T_RESET;
16         clk_tb <= '1';
17         wait for T_CLK/2;
18         clk_tb <= '0';
19         wait for T_CLK/2;
20
21    end process;
22         --feed the simulated user inputs
23    stim_proc: process
24    begin
25      for i in 0 to 9 loop
26                 digit_tb <= conv_std_logic_vector(i,4);
27                 digit_select_tb <= conv_std_logic_vector(i mod 4,2);
28                   wait for T_CLK;
29                   --T_CLK for ease of reading in the simulation diagram
30                   --can be any value
31      end loop;
32    end process;
33 end;
```

In order to test the behavior of the controller, I wrote a loop (lines 25-28) in which the input digit is incremented from 0 to 9 and the `digit_select` input is also changed . This way I can verify all the configurations. More specifically, I can check against the truth tables a given input signal matches the output signal.
There is a delay at the end of the loop (line 28) which prevents the simulation from changing the inputs too fast. I re-ran the simulations with varying values of `T_CLK` in order to ensure the correct functioning of the component.

3.2 Output

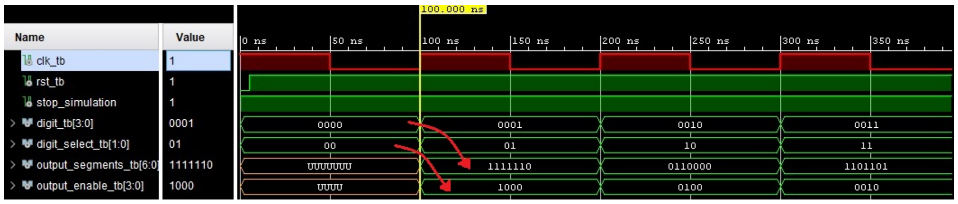
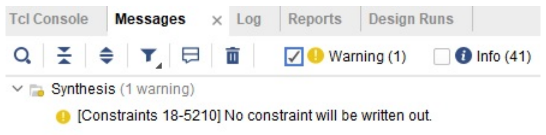


Figure 1: Output waveforms

The above plot shows the result of a simulation of the controller. It is immediate to note that the outputs are updated only on the rising edge of the clock. Also, the correctness of the controller is easily verified by double-checking the outputs with the truth table described in paragraph 1. In this case, the `T_CLK` delay was set equal to a clock period therefore we can see that changes in the input signals are seen in output with a period of delay (red arrows).

3.3 Synthesis

The synthesis of this component was performed by means of the Xilinx Vivado software suite. The only warning received during the **first** synthesis process was the following:



[Constraints 18-5210] No constraint will be written out.

Figure 2: Synthesis Warnings

This warning indicates that there are no constraints specified for this project at this time. On following runs constraints were given therefore no warings were generated.
The synthesis process produced the following RTL structure:

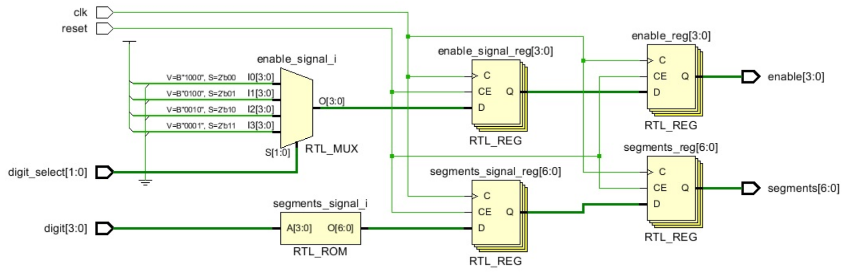
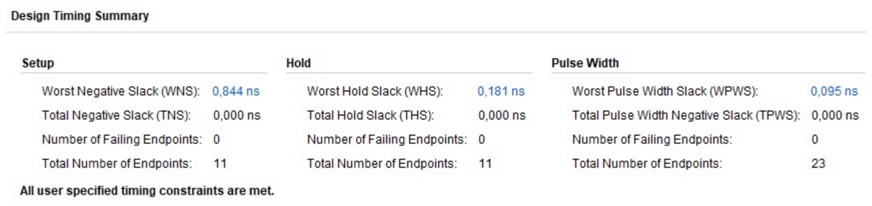


Figure 3: RTL structure

The following step consisted in giving a clock constraint to the network and check the performances. After some trial and error, I gave the following clock constraint:

`create_clock period 2.250 name my_clk waveform {0.000 1.125} [get_ports clk]`
After this, I reloaded the synthesis (no warnings) and checked the Design Timing Summary which I report below.



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,844 ns	Worst Hold Slack (WHS):	0,181 ns	Worst Pulse Width Slack (WPWS):	0,095 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	11	Total Number of Endpoints:	11	Total Number of Endpoints:	23

All user specified timing constraints are met.

Figure 4: Final Design Timing Summary

Where:

- WNS (Worst Negative Slack) is the critical path, corresponds to the worst slack of all the timing paths for max delay analysis.
- TNS (Total Negative Slack) is the sum of negative slack.
- WHS (Worst Hold Slack) corresponds to the worst slack of all the timing paths for min delay analysis.
- THS (Total Hold Slack) is the sum of negative hold slack paths.
- WPWS (Worst Pulse Width Slack) corresponds to the worst pulse width slack for (Min low pulse width, min high pulse width, min period, max period).
- TWPS (Total Worst Pulse Width) is the sum of all WPWS violations.

At this point we can compute the maximum operating frequency which is:

$$f_{\max} = \frac{1}{t_{\text{clk\_min}}}$$

We know that the minimum clock is:

$$t_{\text{clk\_min}} = t_{\text{sup}} + t_{\text{prop}} + t_{\text{c\_q}} + t_{\text{slack}}$$

where:

- t_{sup} is the time during which the input shall be stable before the clock rising edge.
- t_{prop} is the propagation delay.
- $\text{t}_{\text{c\_q}}$ is the time after the clock edge necessary for the stabilization of the output.
- t_{slack} is the time that the data is stable.

Sice we want to minimize the clock we need to set $t_{\text{slack}} = 0$. This brings us to the conclusion that:

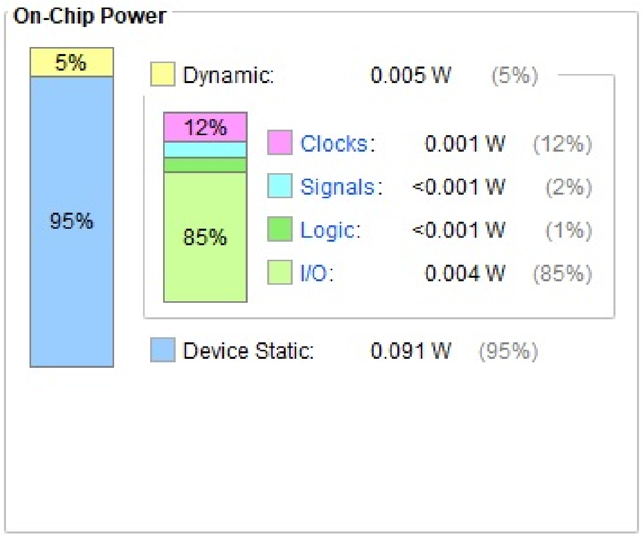
$$t_{\text{clk\_min}} = t_{\text{clk}} - t_{\text{slack}} = 22.5ns - 0.844ns = 21.656ns$$

Finally:

$$f_{\max} = \frac{1}{t_{\text{clk\_min}}} = 46.18MHz$$

4 Conclusions

All the steps were carried out without particular problems. The final product works as intended. To conclude this work, I will report the power utilization of the proposed design.



Dynamic:	0.005 W	(5%)
Clocks:	0.001 W	(12%)
Signals:	<0.001 W	(2%)
Logic:	<0.001 W	(1%)
I/O:	0.004 W	(85%)
Device Static:	0.091 W	(95%)

Figure 5: Final Design Timing Summary