Laboratory 5

Creeping Line

Digital Systems Design with VHDL
IAS0600

Marcel Cases Freixenet

Tallinn,
October 17, 2018

Tallinna Tehnikaülikool
TalTech

Contents

1 Introduction

1.1 Aim

This project consists in the design, simulation and implementation in VHDL of a creeping line. The content will be multiplexed and shown on the 7-segment displays of the Nexys 4 board.

1.2 Background

A 7-segment display [1] consists in eight LEDs that compose a human-readable number:

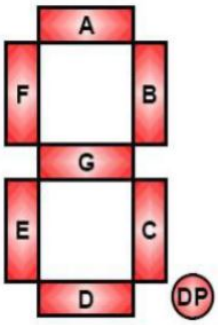

Figure 1.1 *7-segment display*

Each display has eight (one for every LED or segment) and one anode. When more than one 7-segment display are used, then the anodes have to be multiplexed. In order to avoid showing the same value on all of the displays, the cathodes also have to be multiplexed according to the current anode.

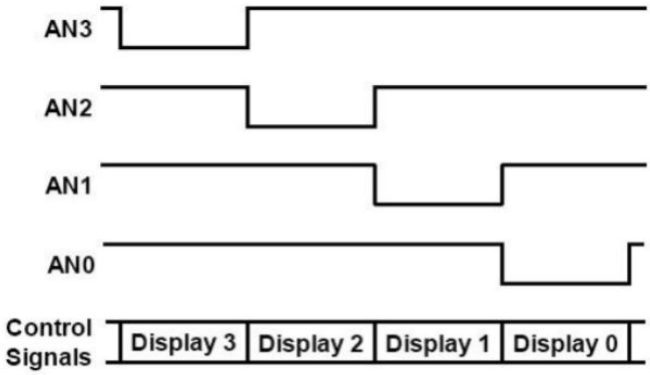

Figure 1.2 *7-segment display anode multiplex*

Only four 7-segment displays will be used the first steps of this Lab, and they will show an hexadecimal value of the binary-input from 16 switches as shown below.

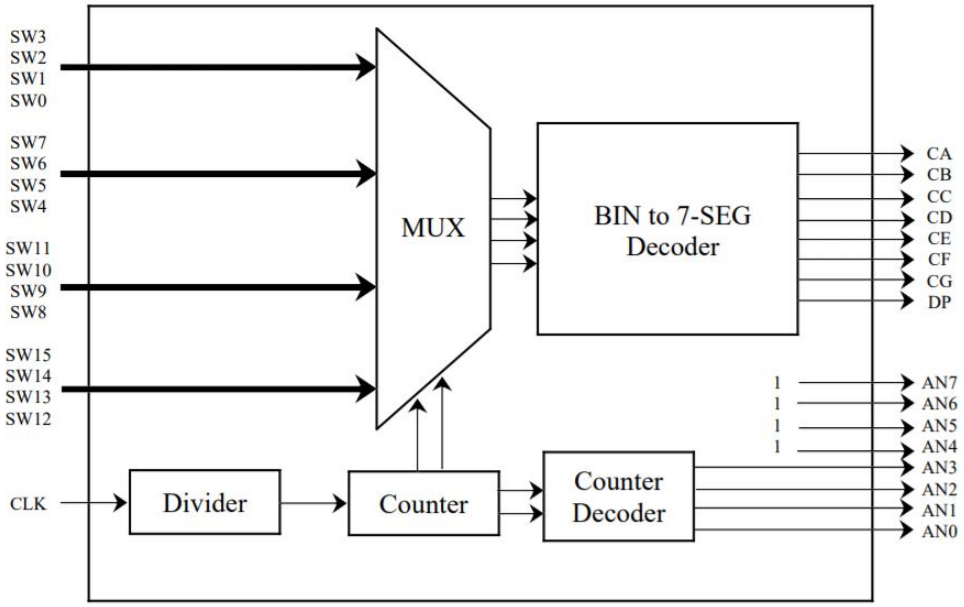

Figure 1.3 *Four 7-segment displays from switches*

Then we will be using eight 7-segment displays to show a code stored in a 32-bit shift register [2] that will shift at a frequency of 2Hz so that the code rotates among the eight 7-segment displays.

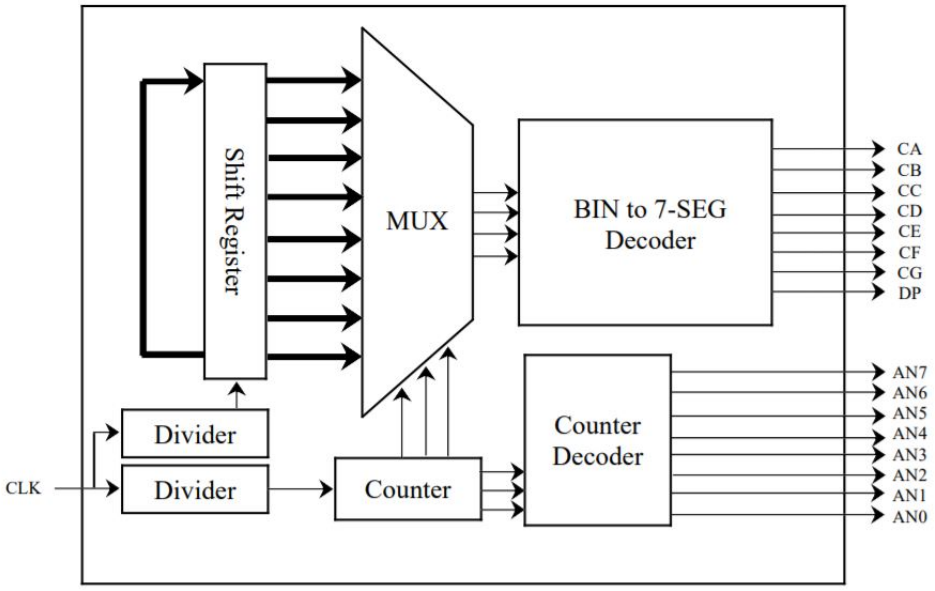

Figure 1.4 *Eight 7-segment displays from a shift register*

2 Workflow

2.1 Top level file

2.1.1 Libraries

Two libraries have been used for this project:

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
```

2.1.2 Entity

A clock and reset signals are necessary for the synchronous processes involved in this Lab.

2.1.2.1 Using switches

When using switches, *sw* is an input vector of 16 bits to indicate the values we want to show on displays.

```
1 entity creeping_line is
2         Port (  clk : in std_logic ; -- Internal clock (100MHz)
3                 reset : in std_logic ; -- Centre button on Nexys 4 board
4                 sw : in std_logic_vector (15 downto 0);
5                 cat : out std_logic_vector (7 downto 0);
6                 an : out std_logic_vector (7 downto 0)
7                 );
8 end creeping_line;
```

2.1.2.2 Using a shift register

When using a 32-bit shift register, a *sw* port is not necessary anymore because a shift register is an internal process and the code we want to show is defined internally.

```
1 entity creeping_line is
2         Port (  clk : in std_logic ; -- Internal clock (100MHz)
3                 reset : in std_logic ; -- Centre button on Nexys 4 board
4                 cat : out std_logic_vector (7 downto 0);
5                 an : out std_logic_vector (7 downto 0)
6                 );
7 end creeping_line;
```

2.1.3 Architecture

2.1.3.1 Using switches

This architecture only uses two components: a *decoder* and a *clock divider*. Then some modules run concurrently.

The 16-bit long physical signal from the switches is multiplexed in groups of 4 bits (one for each display) and stored in *sw_mux*. The switching frequency is defined in the process *Display mux*, which has the behavior of an up-counter at a frequency of *clk_div*. This frequency is set at 1kHz which is enough for a human eye.

The signal is sent to the decoder component and it returns the configuration of cathodes necessary to display a number. The current anode has to be set for every moment. This is done in *gen_an* generate statement.

```
1  architecture Behavioral of creeping_line is
2
3          -- Component *DECODER*
4          component decoder is
5                  port (  sw : in std_logic_vector (3 downto 0);
6                          cat : out std_logic_vector (7 downto 0)
7                          );
8          end component;
9
10         -- Component *CLOCK DIVIDER*
11         component clock_divider is
12                 generic (eoc: integer := 99999); -- 1kHz
13                 Port (  clk : in STD_LOGIC;
14                         reset : in STD_LOGIC;
15                         clk_div : out STD_LOGIC
16                         );
17         end component;
18
19         signal sw_mux : std_logic_vector (3 downto 0);
20         signal current_display : integer range 0 to 3;
21         signal clk_div : std_logic ; --1kHz
22
23 begin
24
25 inst_decoder_display_i : decoder
26         port map (  sw => sw_mux,
27                 cat => cat
28                 );
29
30 -- Clock divider to 1kHz (Display mux shift frequency)
31 inst_clock_divider : clock_divider
32         port map (  clk => clk,
33                 reset => reset,
34                 clk_div => clk_div
35                 );
36
37 -- Switch mux
38 with current_display select
39         sw_mux <=
40                 sw(3 downto 0)      when 0,
```

```
41                sw(7 downto 4)      when 1,
42                sw(11 downto 8)     when 2,
43                sw(15 downto 12)    when 3,
44                "0000"              when others;
45
46  -- Display mux
47  process (clk, reset) begin
48          if rising_edge(clk) then
49                  if reset = '1' then
50                          current_display <= 0;
51                  elsif clk_div = '1' then
52                          current_display <= current_display + 1;
53                  end if;
54          end if;
55  end process;
56
57  -- Anode mux
58  gen_an: for i in 0 to 7 generate
59          an(i) <= '0' when current_display = i else '1';
60  end generate gen_an;
61
62  end Behavioral;
```

2.1.3.2 Using a shift register

When using a shift register instead of switches, a new component has to be declared for this purpose.

The main difference with the switches model is that two clock dividers are required: one for the decoder multiplex (1kHz) and another for shifting the shift register (2Hz).

Since switches are not used anymore, a signal *code* is set in the architecture. This signal is now the input for the parallel shift register.

The *parallell_out* signal from the shift register is routed to *code_shifted*, and the latter is sent to the decoder using a *when ... else* statement that depends on the current enabled signal.

```
1   architecture Behavioral of creeping_line is
2
3           -- Component *DECODER*
4           component decoder is
5                   port (  input : in std_logic_vector (3 downto 0);
6                           cat : out std_logic_vector (7 downto 0)
7                           );
8           end component;
9
10          -- Component *CLOCK DIVIDER*
11          component clock_divider is
12                  generic (eoc: integer := 99999); -- 1kHz
13                  Port (  clk : in STD_LOGIC;
14                          reset : in STD_LOGIC;
15                          clk_div : out STD_LOGIC
16                          );
17          end component;
```

```
18
19          -- Component *SHIFT REGISTER*
20          component shift_register is
21                  generic ( n : integer := 32);
22                  port(   clk: in std_logic;
23                          reset: in std_logic;
24                          enable: in std_logic; --enables shifting
25                          parallel_in: in std_logic_vector(n-1 downto 0);
26                          parallel_out: out std_logic_vector(n-1 downto 0)
27                          );
28          end component;
29
30          signal current_display : integer range 0 to 7;
31          signal clk_div_displays : std_logic ; --1kHz is OK for human eye
32          signal clk_div_shift_register : std_logic ; -- 2Hz for proper reading
33          signal shift_register_buff : std_logic ; -- Shift Register buffer
34          signal sr_mux : std_logic_vector (3 downto 0); -- Shift Register mux
35          signal code, code_shifted : std_logic_vector (31 downto 0); -- Code to
            ↪  show on displays
36
37  begin
38
39  -- *CODE*
40  code <= X"ABCD1234";
41
42  inst_decoder_display_i : decoder
43          port map (  input => sr_mux,
44                  cat => cat
45                  );
46
47  -- Clock divider to 1kHz (Display mux shift frequency)
48  inst_clock_divider_displays : clock_divider
49          generic map (eoc => 99999)
50          port map (  clk => clk,
51                  reset => reset,
52                  clk_div => clk_div_displays
53                  );
54
55  inst_clock_divider_shift_register : clock_divider
56          generic map (eoc => 49999999) -- 2Hz
57          port map (  clk => clk,
58                  reset => reset,
59                  clk_div => clk_div_shift_register
60                  );
61  -- Switch mux
62  with current_display select
63          sr_mux <=
64                  code_shifted(3 downto 0)      when 0,
65                  code_shifted(7 downto 4)      when 1,
66                  code_shifted(11 downto 8)     when 2,
67                  code_shifted(15 downto 12)    when 3,
68                  code_shifted(19 downto 16)    when 4,
69                  code_shifted(23 downto 20)    when 5,
70                  code_shifted(27 downto 24)    when 6,
71                  code_shifted(31 downto 28)    when 7,
72                  "0000"                        when others;
73
74  inst_shift_register : shift_register
75          generic map(n => 32)
```

```
76          port map (  clk => clk,
77                  reset => reset,
78                  enable => clk_div_shift_register,
79                  parallel_in => code,
80                  parallel_out => code_shifted
81                  );
82
83  -- Display mux
84  process (clk, reset) begin
85          if rising_edge(clk) then
86                  if reset = '1' then
87                          current_display <= 0;
88                  elsif clk_div_displays = '1' then
89                          current_display <= current_display + 1;
90                  end if;
91          end if;
92  end process;
93
94  -- Anode mux
95  gen_an: for i in 0 to 7 generate
96          an(i) <= '0' when current_display = i else '1';
97  end generate gen_an;
98
99  end Behavioral;
```

2.2 Components

2.2.1 Decoder

The decoder has the purpose to return the values of the cathodes for every 4-bit input, which represents an hexadecimal character. This component works as a look-up table and works in a combinational way, independently from the clock frequency and the reset.

```
1   entity decoder is
2           Port (  input : in std_logic_vector (3 downto 0);
3                   cat : out std_logic_vector (7 downto 0)
4                   );
5   end decoder;
6
7   architecture Behavioral of decoder is begin
8
9   with input select
10          cat <=
11                  "00000011" when "0000", --0
12                  "10011111" when "0001", --1
13                  "00100101" when "0010", --2
14                  "00001101" when "0011", --3
15                  "10011001" when "0100", --4
16                  "01001001" when "0101", --5
17                  "01000001" when "0110", --6
18                  "00011111" when "0111", --7
19                  "00000001" when "1000", --8
20                  "00011001" when "1001", --9
21                  "00010001" when "1010", --a
22                  "11000001" when "1011", --b
```

```
                "11100101" when "1100", --c
                "10000101" when "1101", --d
                "01100001" when "1110", --e
                "01110001" when "1111", --f
                "11111111" when others;

end Behavioral;
```

2.2.2 Clock Divider

The clock divider is used multiple times in this Lab and it has to be parameterizable in order to give an output for different frequencies. This is why it has a generic parameter, *eoc*, which stands for *End Of Counting*.

```
entity clock_divider is
        generic (eoc: integer := 99999);
        Port (  clk : in STD_LOGIC;
                reset : in STD_LOGIC;
                clk_div : out STD_LOGIC
                );
end clock_divider;


architecture Behavioral of clock_divider is
        signal counter: integer range 0 to eoc;
begin

process (clk) begin
        if clk'event and clk = '1' then
                clk_div <= '0';
                if reset = '1' then
                        counter <= 0;
                elsif counter = eoc then
                        counter <= 0;
                        clk_div <= '1';
                else counter <= counter + 1;
                end if;
        end if;
end process;

end Behavioral;
```

2.2.3 Shift Register

The Shift Register is circular. This means that the input (*parallel_in*) has the same length of the output (*parallel_out*), which is set by another generic parameter *n*. Every time *enable* is activated, it shifts 4 bits so that we can see that the code has been moved one position left on the displays.

```
entity shift_register is
        generic ( n : integer := 32);
        port(   clk: in std_logic;
                reset: in std_logic;
                enable: in std_logic; --enables shifting
                parallel_in: in std_logic_vector(n-1 downto 0);
                parallel_out: out std_logic_vector(n-1 downto 0)
```

```
                );
end shift_register;

architecture behavioral of shift_register is
        signal parallel_out_buff : std_logic_vector (n-1 downto 0);
begin

process (clk) begin
        if clk'event and clk = '1' then
                if reset = '1' then
                        parallel_out_buff <= parallel_in;
                elsif enable = '1' then
                        parallel_out_buff(3 downto 0) <= parallel_out_buff(31
                        ↪  downto 28);
                        parallel_out_buff(31 downto 4) <= parallel_out_buff(27
                        ↪  downto 0);
                end if;
        end if;
end process;

parallel_out <= parallel_out_buff;

end behavioral;
```

2.3 Testbench

For the 16 switches version, the testbench uses four different combinations of switches to perform a simulation. For the Shift Register version, the *stimulus* process is removed because the user does not have the possibility to change the inputs anymore and the code to be shifted is set concurrently in the architecture.

```
clk_tb <= not clk_tb after 5ns; --half_period
reset_tb <= '0' after 10ns;

uut : creeping_line
        port map (  clk => clk_tb,
                reset => reset_tb,
                sw => sw_tb,
                cat => cat_tb,
                an => an_tb
                );

stimulus: process begin

        sw_tb <= X"0000" ;    wait for 10000 us;
        sw_tb <= X"1234" ;    wait for 10000 us;
        sw_tb <= X"ABCD" ;    wait for 10000 us;
        sw_tb <= X"137F" ;    wait for 10000 us;

end process;
```

3 Results and discussion

Before proceeding to the implementation of this project, two simulations have been performed (for switches version and for the shift register one).

3.1 Simulation

As we can see, the current display is the source to multiplex the anodes. When an anode is equal to one, it will not light physically. Only when it is zero it will show the value set by the cathodes. The value established in *sw_tb* is also multiplexed in *sw_mux*.

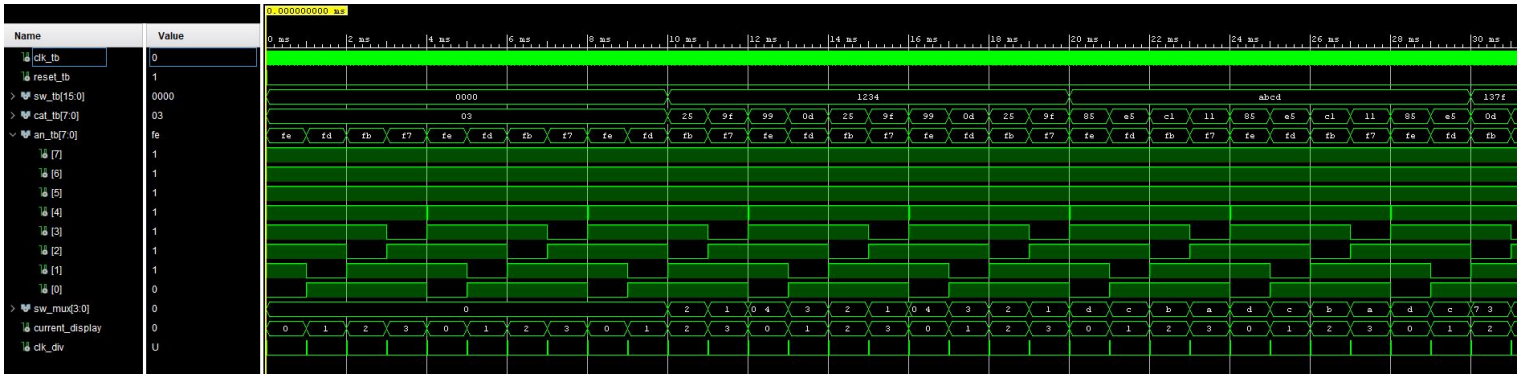

Figure 3.1 *Switches version*

For the Shift Register version, the simulation shows how the *code* signal is scrolled indefinitely.

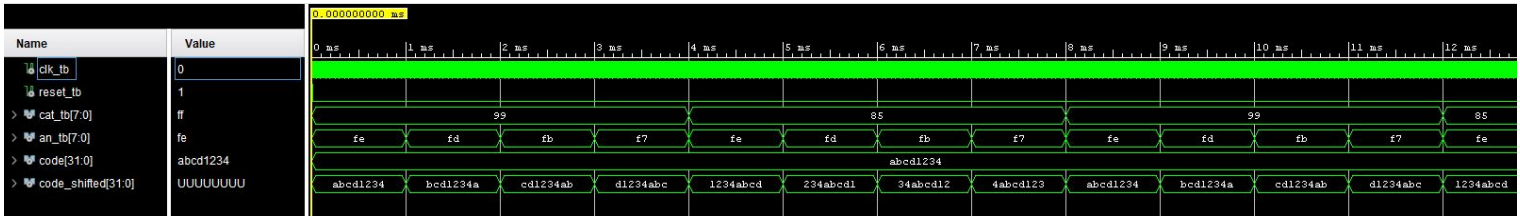

Figure 3.2 *Shift Register version*

4 Conclusion

This Lab has shown the necessity to use a multiplexer when we want to show values on some displays but we have a physical limitation which is a finite number of ports (for cathodes).

It also shows a powerful tool for hardware description languages, which is the *generate* statement. Given a defined integer number previous to the synthesis, this statement will generate as many hardware elements as the given number requires. It has been used to generate the anodes.

We have also experimented with *generic* values, which give the possibility to change the behavior of a component giving a number on the instantiation.

References

[1] Wikipedia. Seven-segment display. https://en.wikipedia.org/wiki/Seven-segment_display, 2018. 1

[2] Wikipedia. Shift register. https://en.wikipedia.org/wiki/Shift_register, 2018. 2