FPGA laboratory report

Diego Figueroa – 2485776

September 16, 2025

Contents

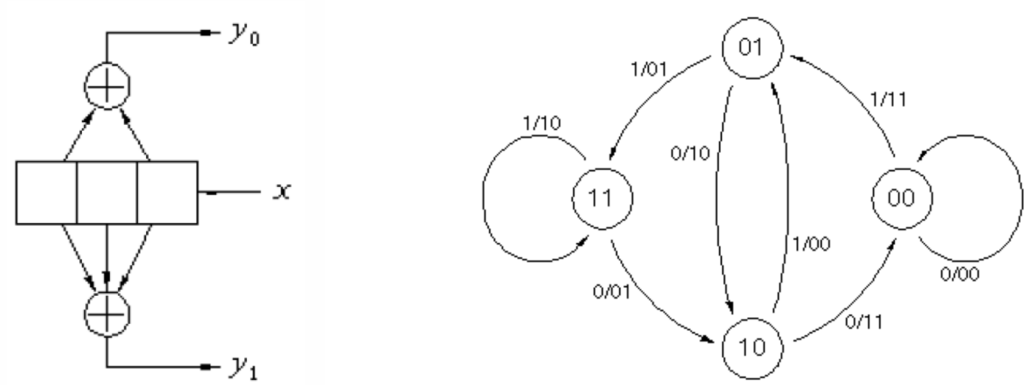
List of Figures

List of Tables

Listings

1 Convolutional encoder

The goal of this practice is to implement a convolutional encoder, of a specific code of rate $R = 1/2$, with memory $m = 2$, and polynomials generators $g_0(x) = x^2 + 1$ and $g_1(x) = x^2 + x + 1$. This specific convolutional encoder can be represented by a shift register and some xor operations, as seen in figure 1a, or as a finite state machine as seen in figure 1b.



(a) Representation as shift register **(b)** Representation as a finite state machine

Figure 1: Possible representations of convolutional encoders.

1.1 Symbol schematic implementation

To implement this encoder in the FPGA, the first Method used was to use the schematic interface of Quartus II. For this propose the representation of a shift register of the convolutional encoder is well suited, so it is implemented as seen in figure 2. There the shift register is done with 3 D-flipflops conected in series, with a common clock and reset, and no preset is used. Finally the two output bits are generated with xor gates using the signals stored in the registers.

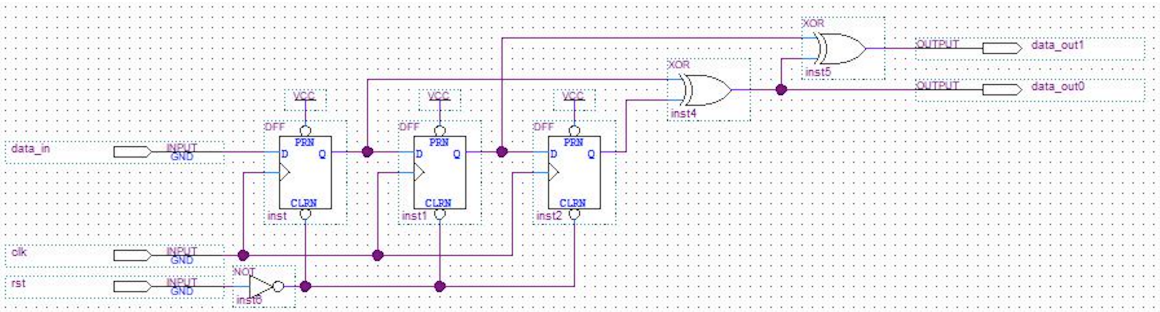


Figure 2: Schematic of the convolutional encoder in Quartus II.

Finally the circuit is simulated, and the results can be seen in section 1.4.

1.2 VHDL implementation

For VHDL the representation of the code as a FSM is used. The code describing this machine can be seen listing 1 in the Annex, this code is based in an example of a state machine in [1].

In the architecture declaration, the first thing done is defining a new enum type to store the states of the state machine, then some signals are defined. Two processes are used, one for the state logic i.e. describing the outputs and what transitions are done in each state; and the other to update the state and read the incoming data on every rising edge of the clock.

Finally the circuit is simulated, and the results can be seen in section 1.4.

1.3 AHDL implementation

For the AHDL implementation a FSM is also used. The code in this case can be seen in listing 2, and is inspired by the example found in [2].

The first three lines correspond to the declaration of the component, then two variables are declared, one of type `MACHINE`, where the states are declared, and one of type `node`. In the description of the behavior, the clock and the reset of the state machine ara assigned, `d_in` is implemented as a D-flipflop to store the incoming data over a clock period, and the behavior of the FSM is described as a table.

It is important to notice that the state machine is implemented with asynchronous outputs, that means that while being in one state, the outputs can change if the inputs change, even before the clock edge. That is why it is important the flip flop for `d_in`, so it has a stable value over a clock period.

Finally the circuit is simulated, and the results can be seen in section 1.4.

1.4 Simulation results

To simulate all the encoders a small schematic impementing all the encoders is done, it can be seen in figure 3. In figure 4 can be seen the simulations result of the encoders, the first two signals (from top to bottom) are the clock and a reference clock with a phase difference to change the data. The third signal is the input binary data, the forth signal is the reset signal.

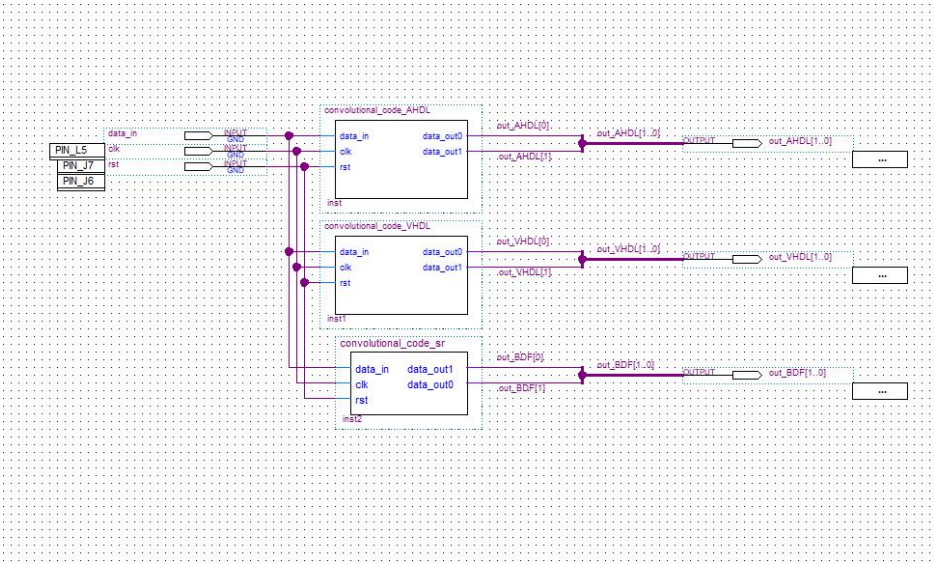


Figure 3: Schematic of all the convolutional encoders in Quartus II.

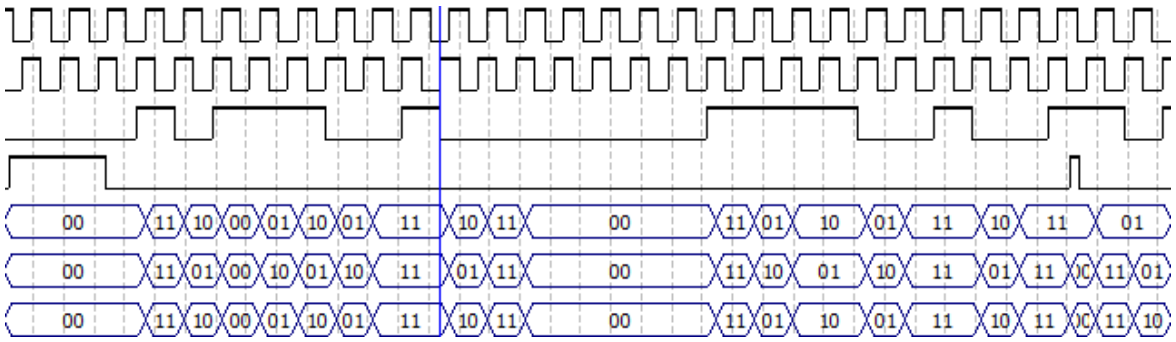


Figure 4: Simulation results of the three convolutional codes encoders.

Finally we see the signals of the tree encoders: AHDL, VHDL and Schematic implementation. The first part of the data is de binary representation of 0x$B9$, and then there is some random data. It is evident that the three encoders have the same output which is also the expected one given the desired code. The only difference is at the end of the simulation when the reset signal goes to one for a small time. Here the two last encoders behave the same, but the first is different, this is because the AHDL implementation has a synchronous reset while the other two have an asynchronous reset.

2 Running light

The goal of this practice is to familiarize with the FPGA board, its built elements, such as leds push buttons, switches and seven segments displays, and program the FPGA to test the results of a circuit in real life

2.1 Functional description

The circuit has a series of functional requirements:

- Using the leds of the board, make a running light that goes from left to right and back, by turning off the current led and turning on the next one in a line, so it seems like a moving light (it is important to take into account the speed of the moving light, so it is possible to perceive the effect with the eyes).
- Use the seven segments to count how many times the light has gone from one side to the other.
- Add the possibility to change the speed of the running light with the use of some switches.
- Add de ability of giving a desired pattern that moves through the leds, by using the switches.
- Change the operation mode of the circuit between **Start**, **Stop**, **Reset, Load pattern**, by using the push buttons.
- Change the direction of the lights between **left**, **right** or **for- and backward**

2.2 Digital Design and VHDL implementation

To accomplish the functional specifications of the circuit, the design is broken into smaller pieces, which are then interconnected in a top level entity. Each of the subsystems and the top level entity are describen next.

2.2.1 lr_ring_reg

To accomplish the moving pattern in both directions the entity `lr_ring_reg` is used. It is a shift register that shift all the values to the left (in a circular manner) on every rising edge of the clock. It also has a `pattern_in` signal that is hard set in the registers if the signal `load` is set to high. Finally the output is a pattern moving left, and other moving right, which is implemented simply by reversing the order of the ring register.

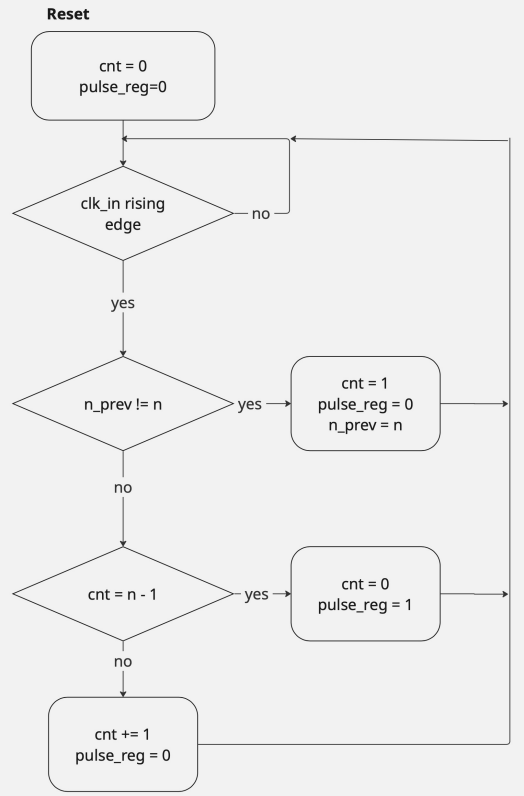
The implementation in VHDL can be seen in listing 3. In line 7 a generic is defined to specify the number of cells in the ring register. In the architecture some intermediate signals are defined. Then in line 16 - 19 a concurrent generate statement is used to define `pattern_out_l` as simply `pattern_out`, and `pattern_out_r` as the reversed version of `pattern_out`. A process is used to do the shift operation, and an other process is used to load a pattern and update `pattern_out`.

2.2.2 clk_div_n

A clock divider is needed for different proposes: first to make the effect of the moving light visible, but also to change the speed of the light. Also a clock that is n_{led} (number of leds) slower than the clock used for the ring register is useful to signal when a light has done a full run from one side to the other (which is needed to count the runs of the light).

For this reason an entity for a clock divider is defined, which can change the division number during operation. The code for this module is in listing 4. In the entity declaration a generic is used to define the counter width. The only process of the architecture resets the counter `cnt` and the pulse to 0 if `rst` is 1. Otherwise on a `clk_in` rising edge if `n` changes, set `cnt` to 1 and `pulse_reg` to 0, if `cnt = n - 1`, `cnt` is set again to 0 and `pulse_reg` to 1, otherwise `cnt` is increased by one and `pulse_reg` set to 0. The result is a small positive pulse every `n` pulses of `clk_in`.

(a) `clk_dic_n` symbol block



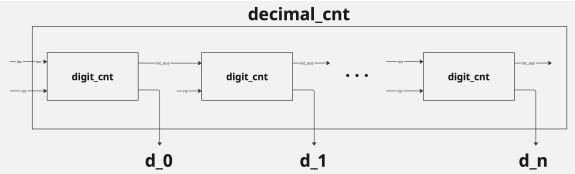
(b) `clk_dic_n` flow diagram

Figure 5: `clk_dic_n` desing

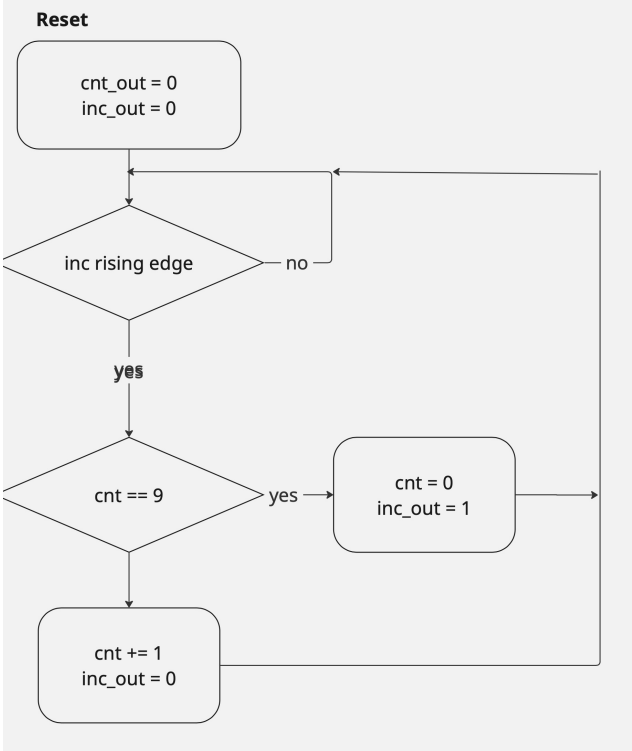
2.2.3 decimal_cnt

This module is used to count the number of runs of the light. It is based on the module `digint_cnt`, and is just a series connection of n `decimal_cnt`'s, as can be seen in listing 5. And `digint_cnt` just counts on every `inc` rising edge, and if gets to 9 it goes back to 0 and generate a pulse on `inc_out` (see code on listing 6).

(a) `digit_cnt` symbol block



(b) `decimal_cnt` schematic



(c) `digit_cnt` flow diagram

Figure 6: `decimal_cnt` design.

2.2.4 seven_segments

This module is really simple, it just outputs the needed signals for a seven segments display depending on the given number it receives, it is just implemented as a look up table in a concurrent statement as seen in listing 7.

2.2.5 speed_rom

To change the speed of the light a ROM is implemented, where different values of n are stored and used as inputs of `clk_div_n`, this file is automatically generated using a python code (see listing 8) depending in some parameters, an example the VHDL rom can be seen in listing 9.

2.2.6 const_types_pkg

A package to define some types and constants is also used to improve the readability of the code in `running_light` (see code in listing 10).

2.2.7 running_light

Finally the top level entity is described in listing 11. A structural schematic of the model can be seen in figure 7. And the different modules are controlled using the FSM seen in figure 8. In general the fast clock of the FPGA is divided by `speed`, this clock goes to the `lr_ring_reg`, and is also divided to count the number of runs whit the module `decimal_cnt`. The signal going to the leds is taken from a multiplexer depending on the current value `dir`, and with the FSM the circuits switches between the modes **Start**, **Stop**, **Reset, Load pattern**.

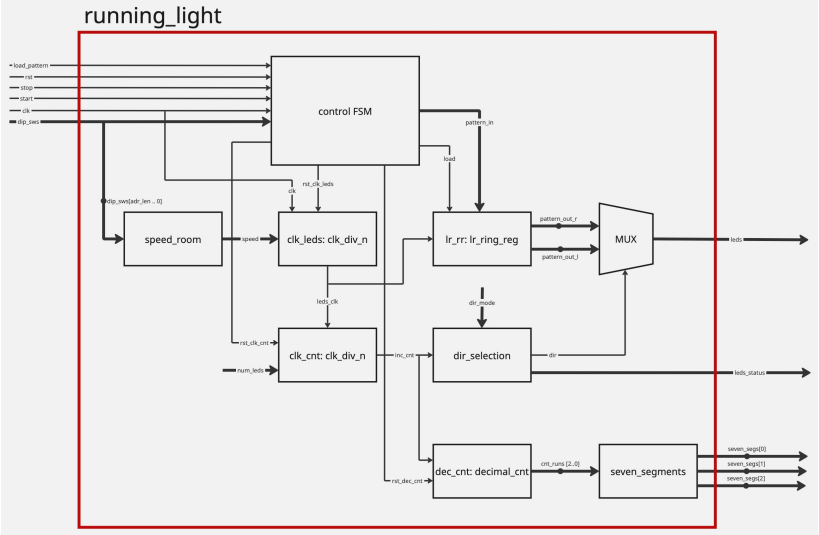


Figure 7: `running_light` schematic

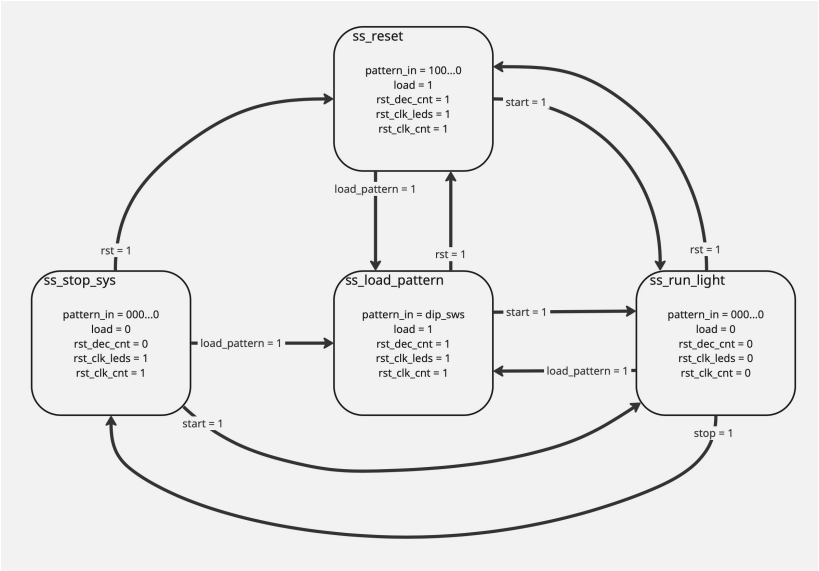


Figure 8: `running_light` control finite state machine.

3 FIR filters

For this experiment the four basic FIR filters are done: low pass, hight pass, band pass, and band stop. All the filters are implemented as a second order FIR, with the coefficients shown in table 1. Additionally the filters are implemented using two possible representations: direct and transposed graph as show in figure 9.

	a_0	a_1	a_2
Low pass	0.25	0.5	0.25
High pass	0.25	-0.5	0.25
Band pass	-0.5	0	0.5
Band stop	0.5	0	0.5

Table 1: Coefficients of the different FIR filters

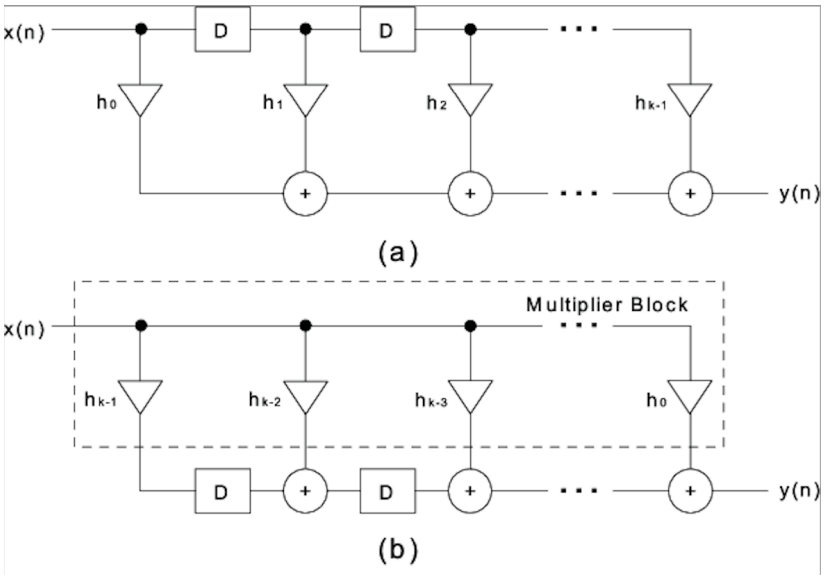


Figure 9: Direct (a) and transposed (b) graphs for a FIR filter.

3.1 Low pass

To implement the Low pass filter the direct representation is used, and the schematic editor of Quartus is used. The final schematic can be seen in figure 10. It is important to notice some modifications to the filter in order to be able to implement it.

First of all, since Quartus schematic editor have only multipliers, the multiplication by 0.25 or 0.5, i.e. division by 4 and 2 can not be implemented. To solve this the coefficients are all multiplied by 4, so the equivalent coefficients are $a_0 = 1$, $a_1 = 2$ and $a_2 = 1$. This does not change the frequency response of the filter, since it is linear, but now the multiplication block can be used.

Second, since there is a multiplication by 4, the output of the multiplication has two bits more than the input. But in our model we wanted to have always the same amount of bits, so the 3 least significant bits are discarded, this is done automatically by the multiplication block.

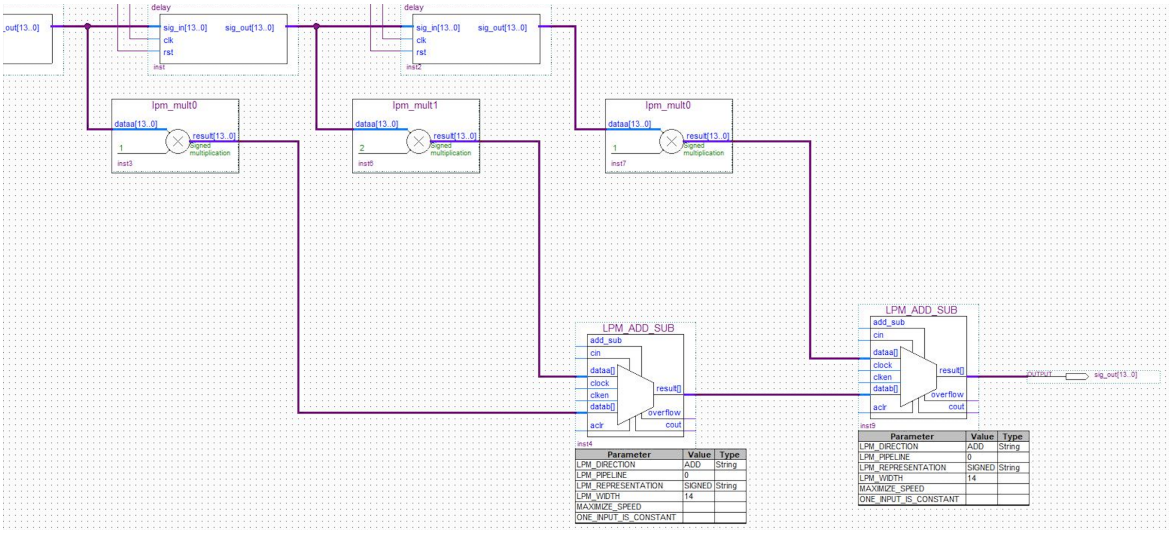
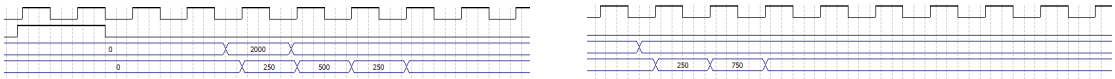


Figure 10: Schematic of the low pass filter in Quartus II

Finally the circuit is simulated under an impulse response and a step response. The results can be seen in figure 11. Notice how the output has a total gain of 1/2, this is because the coefficients are multiplied by 4, but the three LSB are discarded, so a net gain of $4/8 = 1/2$ is obtained.



(a) Simulation of the low pass filter for an impulse response.

(b) Simulation of the low pass filter for a step response.

Figure 11: Low pass filter simulations.

3.2 High pass

The high pass filter is implemented using the transposed representation. The circuit is again described using the Quartus schematic editor, so again the coefficients of 0.25 and 0.5 are implemented by multipliers of weight 1 and 2 respectively, and again the least 3 significant bits of the multipliers are discarded. The result is the expected high pass filter with a net gain of 1/2. The schematic can be seen in figure 12.

3.3 Band pass

The band pass filter is done with VHDL code and the direct representation is used. The code can be seen in listing 13, and consist of a single process, with an asynchronous reset, and on each clock rising edge the output signal is calculated from the registers, and the registers shift the value of the incoming signal. To calculate the output the syntax `taps(0)/ 2` is used, which is allowed by the Quartus synthesizer because the denominator is a power of 2.

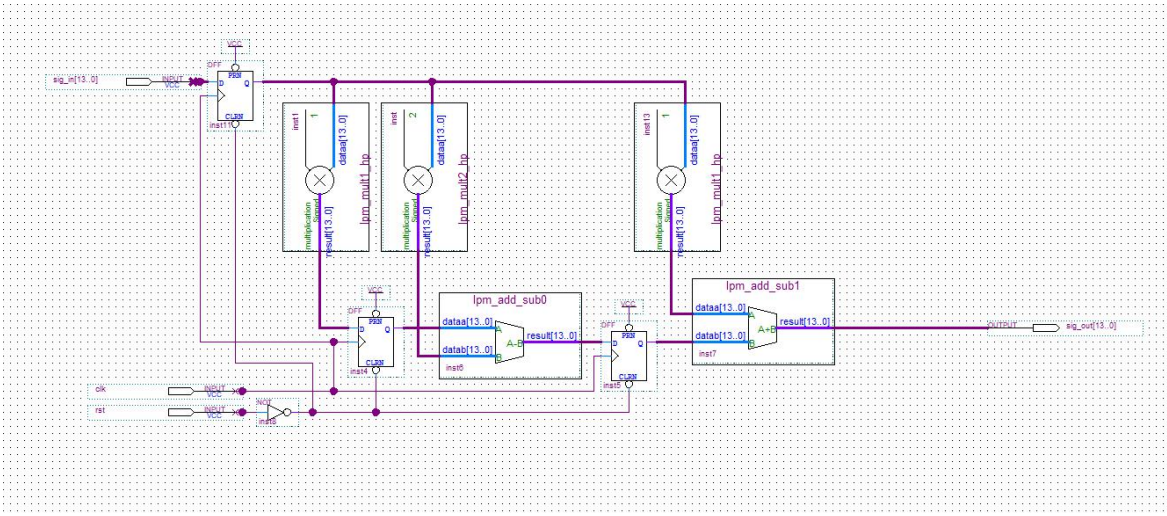


Figure 12: Schematic of the high pass filter in Quartus II

3.4 Band stop

The band stop filter is done with VHDL code and the transposed representation is used. The code can be seen in listing 14, and consist of a single process, with an asynchronous reset, and on each clock rising edge to each register a new value is assigned, which corespond to the sum of the last register and the input signal multiplied with the respective coefficient. Again syntax `taps(0)/ 2` is used to perform the division.

3.5 Project with the 4 filters

Finally to test the filters with the FPGA a small module is written in VHDL that has as inputs the output of each filter, and 4 push buttons used to select one of the filters as the output of the system. The code can be seen in listing 15, it implements a simple process to control the value of the signal `fir_select`, two drivers that perform the multiplexer based on `fir_select` and a final process doing a simple clock divider to lower the sampling frequency of the system, in such a way that the filters can be tested with the 1 MHz signal generator.

All the filters and the multiplexer are connected using the Quartus schematic editor (se figure 13). and testing with a real sinusoidal signal for each filter shows the correct working of the system (see figures 14, 15, 16 and 17. On each figure the input signal is above and the output below)

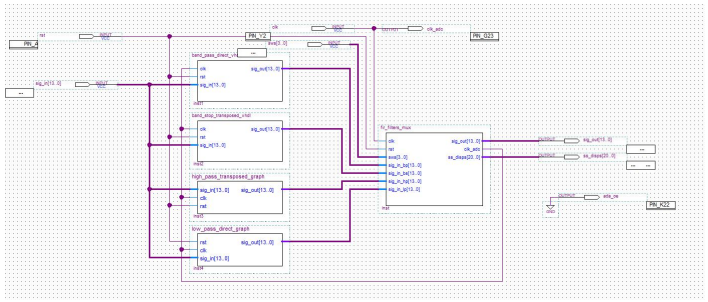
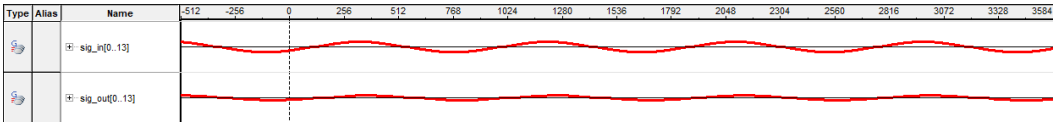
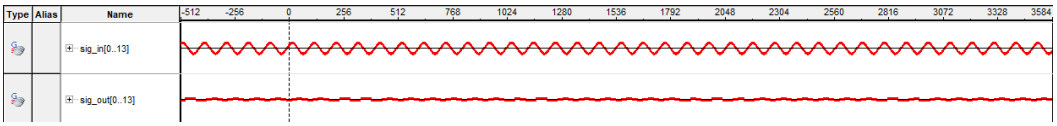


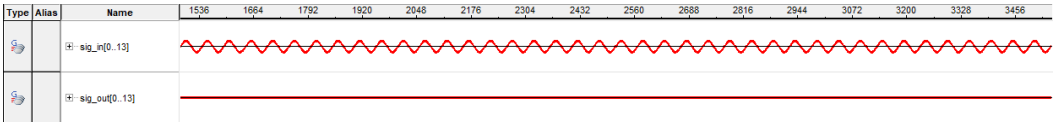
Figure 13: Schematic of all the filters in Quartus II



(a) Low pass filter working at low frequency.

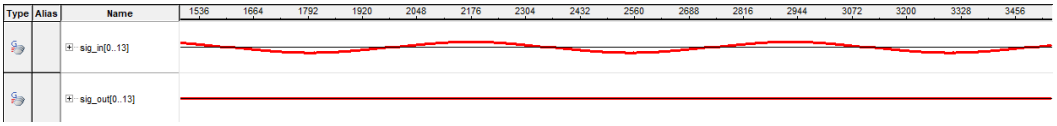


(b) Low pass filter working at intermediate frequency.

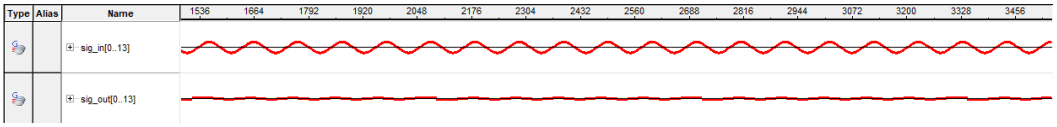


(c) Low pass filter working at high frequency.

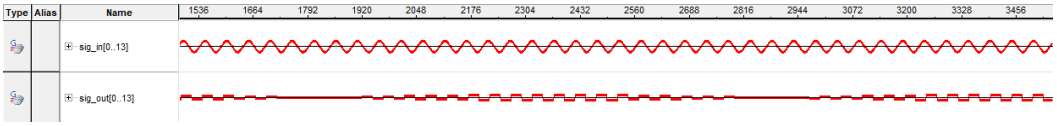
Figure 14: Low pass filter results.



(a) High pass filter working at low frequency.

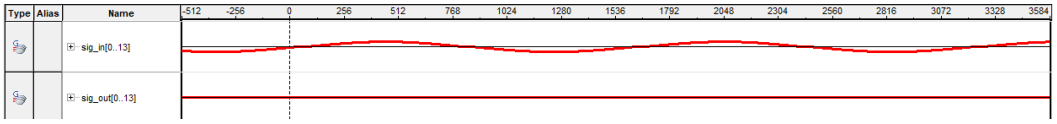


(b) High pass filter working at intermediate frequency.

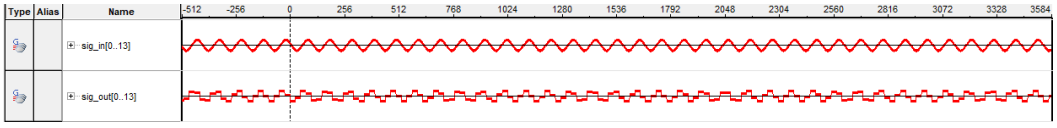


(c) High pass filter working at high frequency.

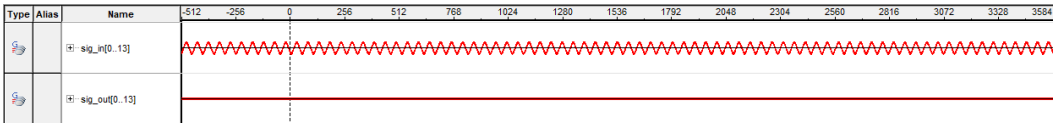
Figure 15: High pass filter results.



(a) Band pass filter working at low frequency.

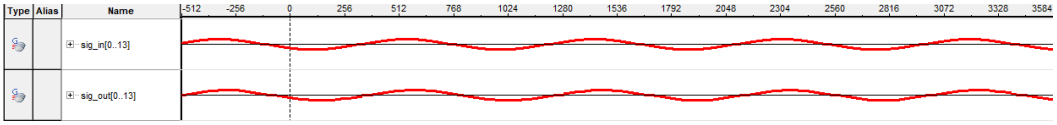


(b) Band pass filter working at intermediate frequency.

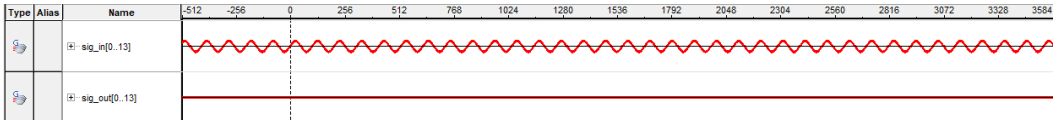


(c) Band pass filter working at high frequency.

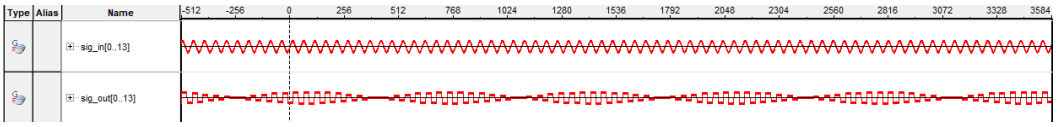
Figure 16: Band pass filter results.



(a) Band stop filter working at low frequency.



(b) Band stop filter working at intermediate frequency.



(c) Band stop filter working at high frequency.

Figure 17: Band stop filter results.

Finally in table 2 we can see the maximum frequency and the number of used logic elements in the implementation of each filter. This shows as that there is a trade off between maximum frequency and the size of the circuit, so the transposed representation is better suited for small circuits, whereas the direct representation is faster at the expenses of using more hardware.

Filter	$F_{\max}$ [MHz]	total logic elements
Low pass	797.45	38
High pass	382.7	36
Band pass	264.69	56
Band stop	404.2	42

Table 2: Limitations of the synthesized filters

References

[1] D. L. Perry, *VHDL /*, 3. ed. New York, NY [u.a.] : McGraw-Hill, 1998, Previous ed.: 1994. [Online]. Available: `http://www.loc.gov/catdir/enhancements/fy0602/98016663-t.html`.

[2] *Designing state machines (ahdl)*. [Online]. Available: `http://www.xilinx.info/_altera/html/_sw/q2help/source/ahdl/ahdl_pro_design_state_machine.htm`.

Annex

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

entity convolutional_code_VHDL is
  port(data_in, clk, rst: in std_logic;
       data_out0, data_out1: out std_logic);
end convolutional_code_VHDL;

architecture state_machine of convolutional_code_VHDL is
  type state is (s00, s01, s10, s11);
  signal curr_st, next_st: state;
  signal d_tmp, d_out0, d_out1: std_logic;
  begin
    process(curr_st, d_tmp) -- state logic
    begin
      case curr_st is
      when s00 =>
        if d_tmp = '0' then
          d_out0 <= '0';
          d_out1 <= '0';
          next_st <= s00;
        else
          d_out0 <= '1';
          d_out1 <= '1';
          next_st <= s01;
        end if;
      when s01 =>
        if d_tmp = '0' then
          d_out0 <= '0';
          d_out1 <= '1';
          next_st <= s10;
        else
          d_out0 <= '1';
          d_out1 <= '0';
          next_st <= s11;
        end if;
      when s10 =>
        if d_tmp = '0' then
          d_out0 <= '1';
          d_out1 <= '1';
          next_st <= s00;
        else
          d_out0 <= '0';
          d_out1 <= '0';
          next_st <= s01;
        end if;
      when s11 =>
        if d_tmp = '0' then
          d_out0 <= '1';
```

```
50            d_out1 <= '0';
51            next_st <= s10;
52          else
53            d_out0 <= '0';
54            d_out1 <= '1';
55            next_st <= s11;
56          end if;
57        end case;
58      end process;
59
60      process(clk, rst) -- go to next state
61      begin
62        if rst = '1' then
63          curr_st <= s00;
64          d_tmp <= '0';
65        elsif (clk = '1' and clk'event) then
66          curr_st <= next_st;
67          d_tmp <= data_in;
68        end if;
69      end process;
70
71      -- concurrent assigment of the output
72      data_out0 <= d_out0;
73      data_out1 <= d_out1;
74    end state_machine;
```

Listing 1: Code in VHDL for the FSM of the convolutional code encoder.

```
1  subdesign convolutional_code_AHDL(
2    data_in, clk, rst: input;
3    data_out0, data_out1: output
4  )
5  variable
6    conv_code_state: MACHINE
7    WITH STATES (
8      s00 = b"00",
9      s01 = b"01",
10     s10 = b"10",
11     s11 = b"11");
12   d_in: node;
13
14 begin
15   conv_code_state.clk = clk;
16   conv_code_state.reset = rst;
17
18   d_in = DFF(data_in, clk, VCC, VCC);
19
20   TABLE
21   conv_code_state, d_in => data_out1, data_out0,
       conv_code_state;
22   s00, 0 => 0, 0, s00;
23   s00, 1 => 1, 1, s01;
```

```
24    s01, 0 => 1, 0, s10;
25    s01, 1 => 0, 1, s11;
26    s10, 0 => 1, 1, s00;
27    s10, 1 => 0, 0, s01;
28    s11, 0 => 0, 1, s10;
29    s11, 1 => 1, 0, s11;
30    END TABLE;
31 end;
```

Listing 2: Code in VHDL for the FSM of the convolutional code encoder.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  use work.const_types_pkg.all;
5
6  entity lr_ring_reg is
7      generic(num_cells: integer := 8);
8      port(clk, load: in std_logic;
9           pattern_in: in std_logic_vector(num_cells - 1 downto
               0);
10          pattern_out_l, pattern_out_r: inout std_logic_vector(
               num_cells - 1 downto 0));
11 end lr_ring_reg;
12
13 architecture behave of lr_ring_reg is
14     signal pattern_out, pattern_shift: std_logic_vector(
          num_cells - 1 downto 0);
15 begin
16     gen_x: for i in pattern_out'range generate
17         pattern_out_l(i) <= pattern_out(i);
18         pattern_out_r(i) <= pattern_out(pattern_out'left +
              pattern_out'right - i);
19     end generate;
20     process (pattern_out)
21     begin
22         pattern_shift(num_cells - 2 downto 0) <= pattern_out(
              num_cells - 1 downto 1);
23         pattern_shift(num_cells - 1) <= pattern_out(0);
24     end process;
25
26     process (clk, load, pattern_in)
27     begin
28         if load = '1' then
29             pattern_out <= pattern_in;
30         elsif clk'event and clk = '1' then
31             pattern_out <= pattern_shift;
32         end if;
33     end process;
34 end behave;
```

Listing 3: Code in VHDL for `lr_ring_reg`.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity clk_div_n is
    generic (
        CNT_WIDTH : integer := 32
    );
    port (
        clk_in, rst: in  std_logic;
        n: in  unsigned(CNT_WIDTH-1 downto 0);
        clk_out: out std_logic
    );
end clk_div_n;

architecture rtl of clk_div_n is
    signal cnt: unsigned(CNT_WIDTH-1 downto 0);
    signal pulse_reg: std_logic;
    signal n_prev: unsigned(CNT_WIDTH-1 downto 0);
begin

    process(clk_in, rst, n)
    begin
        if rst = '1' then
            cnt <= (others => '0');
            pulse_reg <= '0';
            --n_prev <= n;
        elsif clk_in'event and clk_in = '1' then
            if n /= n_prev then
                cnt <= to_unsigned(1, CNT_WIDTH);
                pulse_reg <= '0';
                n_prev <= n;
            elsif cnt = (n - 1) then
                cnt <= (others => '0');
                pulse_reg <= '1';
            else
                cnt <= cnt + 1;
                pulse_reg <= '0';
            end if;
        end if;
    end process;

    clk_out <= pulse_reg;

end rtl;
```

Listing 4: Code in VHDL for `clk_div_n`.

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

use work.const_types_pkg.all;
```

```
5
6  entity decimal_cnt is
7      port(inc, rst: in std_logic;
8           cnt_out: inout digit_array);
9  end decimal_cnt;
10
11 architecture behave of decimal_cnt is
12     component digit_cnt
13         port (inc, rst: in std_logic;
14               inc_out: out std_logic;
15               cnt_out: inout digit);
16     end component;
17     signal inc_vec: std_logic_vector(cnt_out'high + 1 downto 0)
           ;
18 begin
19     inc_vec(0) <= inc;
20     gen_uutx: for i in 0 to cnt_out'high generate
21     begin
22         uutx: digit_cnt port map (inc_vec(i), rst, inc_vec(i+1)
               , cnt_out(i));
23     end generate gen_uutx;
24 end behave;
```

Listing 5: Code in VHDL for `decimal_cnt`.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  use work.const_types_pkg.all;
5
6  entity digit_cnt is
7      port (inc, rst: in std_logic;
8          inc_out: out std_logic;
9          cnt_out: inout digit);
10 end digit_cnt;
11
12 architecture behave of digit_cnt is
13     signal cnt_tmp: digit;
14     signal inc_out_tmp: std_logic;
15 begin
16     process (cnt_out)
17     begin
18         if cnt_out = 9 then
19             cnt_tmp <= 0;
20             inc_out_tmp <= '1';
21         else
22             cnt_tmp <= cnt_out + 1;
23             inc_out_tmp <= '0';
24         end if;
25     end process;
26
27     process (inc, rst)
```

```
28      begin
29          if rst = '1' then
30              cnt_out <= 0;
31              inc_out <= '0';
32          elsif inc'event and inc = '1' then
33              cnt_out <= cnt_tmp;
34              inc_out <= inc_out_tmp;
35          end if;
36      end process;
37  end behave;
```

Listing 6: Code in VHDL for `digit_cnt`.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  use work.const_types_pkg.all;
5
6  entity seven_segments is
7    port (digit_in: in digit;
8          seven_seg_leds: out sev_seg_disp);
9  end seven_segments;
10
11 architecture behave of seven_segments is
12 begin
13     seven_seg_leds <= ss_disp_0 when digit_in = 0 else
14                       ss_disp_1 when digit_in = 1 else
15                       ss_disp_2 when digit_in = 2 else
16                       ss_disp_3 when digit_in = 3 else
17                       ss_disp_4 when digit_in = 4 else
18                       ss_disp_5 when digit_in = 5 else
19                       ss_disp_6 when digit_in = 6 else
20                       ss_disp_7 when digit_in = 7 else
21                       ss_disp_8 when digit_in = 8 else
22                       ss_disp_9 when digit_in = 9;
23 end behave;
```

Listing 7: Code in VHDL for `seven_segments`.

```
1  import numpy as np
2
3  run_min_freq = 0.1
4  run_max_freq = 5
5  num_adr_bits = 5
6
7  num_steps = 2 ** num_adr_bits
8  fpga_clk_freq = 50e6
9  num_leds = 18
10
11 min_led_clk_freq = run_min_freq * num_leds
12 max_led_clk_freq = run_max_freq * num_leds
13 # led_clk_freq_range = np.linspace(min_led_clk_freq,
       max_led_clk_freq, num_steps)
```

```python
led_clk_freq_range = np.geomspace(min_led_clk_freq, max_led_clk_freq, num_steps)

cnt_div_clk = (fpga_clk_freq // led_clk_freq_range).astype(np.int64)
num_cnt_bits = len(bin(cnt_div_clk.max()))
bin_format = f"#0{num_cnt_bits}b"

rom_def = f"\tconstant clk_leds_cnt_len: integer := {num_cnt_bits-2};\n"
rom_def += f"\tconstant adr_len: integer := {num_adr_bits};\n"
rom_def += f"\tconstant num_speeds: integer := {num_steps};\n"
rom_def += f"\ttype rom_speed is array (0 to num_speeds - 1) of unsigned(clk_leds_cnt_len - 1 downto 0);\n"
rom_def += "\tconstant speeds: rom_speed := (\n"
for i, cnt in enumerate(cnt_div_clk):
    rom_def += f'\t\t{i} => "{format(cnt, bin_format)[2:]}",\n'
rom_def = rom_def[:-2] + ");\n"


with open('running_light/speed_rom.vhd', 'w') as f:
    f.write("library ieee;\n")
    f.write("use ieee.std_logic_1164.all;\n")
    f.write("use ieee.numeric_std.all;\n\n")
    f.write("package speed_rom is\n")
    f.write(rom_def)
    f.write("end package;\n")
```

Listing 8: Code in python for creating the speed room.

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

package speed_rom is
  constant clk_leds_cnt_len: integer := 25;
  constant adr_len: integer := 5;
  constant num_speeds: integer := 32;
  type rom_speed is array (0 to num_speeds - 1) of unsigned(clk_leds_cnt_len - 1 downto 0);
  constant speeds: rom_speed := (
    0 => "1101001111101101011110001",
    1 => "1011101011001101010111111",
    2 => "1010010010100111110101111",
    3 => "1001000100100010011110001",
    4 => "0111111111101101100100011",
    5 => "0111000011000010111001011",
    6 => "0110001101100100100010101",
    7 => "0101011110011011111011011",
    8 => "0100110100111000111100111",
    9 => "0100010000010001001101110",
    10 => "0011101111111111010110000",
```

```
22      11 => "00110100111000100110010 10",
23      12 => "00101110100111010101010",
24      13 => "00101001000101101000111 10",
25      14 => "00100100001101111000010 00",
26      15 => "00011111111011000101000 11",
27      16 => "00011100001000110111000 00",
28      17 => "00011000110011010110110 00",
29      18 => "00010101110111001010100 01",
30      19 => "00010011010001010010001 11",
31      20 => "00010000111111000100100 01",
32      21 => "00001110111110001100010 00",
33      22 => "00001101001100100101111 00",
34      23 => "00001011101000011101011 10",
35      24 => "00001010010000001100110 00",
36      25 => "00001001000010011001110 01",
37      26 => "00000111111101110101000 10",
38      27 => "00000111000001011000101 10",
39      28 => "00000110001100000110111 01",
40      29 => "00000101011101001001011 01",
41      30 => "00000100110011111000000 111",
42      31 => "00000100001111010001000 11");
43  end package;
```

Listing 9: Code in VHDL for `speed_rom`.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   package const_types_pkg is
6
7       constant num_sev_seg: integer := 4;
8       constant num_dip_sws: integer := 18;
9       constant num_lights: integer := 18;
10      constant clk_cnt_len: integer := 5;     -- at least ceil(
            log2(num_lights))
11
12      subtype digit is integer range 0 to 9;
13      type digit_array is array (natural range <>) of digit;
14      subtype sev_seg_disp is std_logic_vector (0 to 6);
15      type sev_seg_disp_array is array (num_sev_seg - 1 downto 0)
            of sev_seg_disp;
16      constant ss_disp_0: sev_seg_disp := "0000001";
17      constant ss_disp_1: sev_seg_disp := "1001111";
18      constant ss_disp_2: sev_seg_disp := "0010010";
19      constant ss_disp_3: sev_seg_disp := "0000110";
20      constant ss_disp_4: sev_seg_disp := "1001100";
21      constant ss_disp_5: sev_seg_disp := "0100100";
22      constant ss_disp_6: sev_seg_disp := "0100000";
23      constant ss_disp_7: sev_seg_disp := "0001111";
24      constant ss_disp_8: sev_seg_disp := "0000000";
25      constant ss_disp_9: sev_seg_disp := "0000100";
```

```
26
27      constant num_lights_bit_vec: unsigned(clk_cnt_len - 1
           downto 0) := to_unsigned(num_lights, clk_cnt_len);
28      type direction is (left, right);
29  end package;
```

Listing 10: Code in VHDL for `const_types_pkg`.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  use work.const_types_pkg.all;
6  use work.speed_rom.all;
7
8  entity running_light is
9      port(clk, start_fpga, stop_sys_fpga, rst_fpga,
           load_pattern_fpga: in std_logic;
10         dip_sws: in std_logic_vector(num_dip_sws - 1 downto 0);
11         seven_segs: out sev_seg_disp_array;
12         leds: out std_logic_vector(num_lights - 1 downto 0);
13         leds_status: out std_logic_vector(adr_len + 2 downto 0)
              );
14 end running_light;
15
16 architecture behave of running_light is
17     component lr_ring_reg
18         generic (num_cells: integer);
19         port (clk, load: in std_logic;
20             pattern_in: in std_logic_vector(num_lights - 1
                  downto 0);
21             pattern_out_l, pattern_out_r: inout
                  std_logic_vector(num_lights - 1 downto 0));
22     end component;
23     component clk_div_n
24         generic (CNT_WIDTH: integer);
25         port (clk_in, rst: in std_logic;
26             n: in unsigned;
27             clk_out: out std_logic);
28     end component;
29     component seven_segments
30         port (digit_in: in digit;
31             seven_seg_leds: out sev_seg_disp);
32     end component;
33     component decimal_cnt
34         port(inc, rst: in std_logic;
35             cnt_out: inout digit_array);
36     end component;
37
38     signal rst, start, stop_sys, load_pattern: std_logic;
39     signal rst_dec_cnt, rst_clk_leds, rst_clk_cnt: std_logic;
40     signal inc_cnt, leds_clk, load: std_logic;
```

```
41      signal cnt_runs: digit_array (num_sev_seg - 1 downto 0);
42      signal speed: unsigned(clk_leds_cnt_len - 1 downto 0);
43      signal pattern_in, pattern_out_l, pattern_out_r:
           std_logic_vector(num_lights - 1 downto 0);
44      type state is (ss_reset, ss_stop_sys, ss_run_light,
           ss_load_pattern);
45      signal curr_state, next_state: state;
46      signal dir: direction := left;
47      signal dir_mode: std_logic_vector(2 downto 0);
48
49  begin
50      dec_cnt: decimal_cnt
51          port map(inc_cnt, rst_dec_cnt, cnt_runs);
52      gen_ss : for i in 0 to num_sev_seg - 1 generate
53          ssx: seven_segments port map(cnt_runs(i), seven_segs(i)
               );
54      end generate;
55      clk_leds: clk_div_n
56          generic map(CNT_WIDTH => clk_leds_cnt_len)
57          port map(clk, rst_clk_leds, speed, leds_clk);
58      clk_cnt: clk_div_n
59          generic map(CNT_WIDTH => clk_cnt_len)
60          port map(leds_clk, rst_clk_cnt, num_lights_bit_vec,
               inc_cnt);
61      lr_rr: lr_ring_reg
62          generic map(num_cells => num_lights)
63          port map(leds_clk, load, pattern_in, pattern_out_l,
               pattern_out_r);
64
65      speed <= speeds(to_integer(unsigned(dip_sws(adr_len - 1
           downto 0))));
66      dir_mode <= dip_sws(adr_len + 2 downto adr_len);
67      leds_status(adr_len - 1 downto 0) <= dip_sws(adr_len - 1
           downto 0);
68      leds <= pattern_out_l when dir = left else
69              pattern_out_r when dir = right;
70      rst <= not rst_fpga;
71      start <= not start_fpga;
72      stop_sys <= not stop_sys_fpga;
73      load_pattern <= not load_pattern_fpga;
74
75      process(inc_cnt, dir_mode, dir)
76      begin
77          if dir_mode = "000" then
78              leds_status(adr_len + 2 downto adr_len) <= "001";
79              dir <= left;
80          elsif dir_mode(0) = '1' then
81              dir <= left;
82              leds_status(adr_len + 2 downto adr_len) <= "001";
83          elsif dir_mode(1) = '1' then
84              dir <= right;
```

```
85              leds_status(adr_len + 2 downto adr_len) <= "010";
86          else
87              leds_status(adr_len + 2 downto adr_len) <= "100";
88              if inc_cnt'event and inc_cnt = '1' then
89                  if dir = left then
90                      dir <= right;
91                  else
92                      dir <= left;
93                  end if;
94              end if;
95          end if;
96      end process;
97
98      process(curr_state, start, load_pattern, stop_sys, dip_sws)
99      begin
100         case curr_state is
101             when ss_reset =>
102                 pattern_in <= (num_lights - 1 => '1', others =>
                        '0');
103                 load <= '1';
104                 rst_dec_cnt <= '1';
105                 rst_clk_leds <= '1';
106                 rst_clk_cnt <= '1';
107                 if start = '1' then
108                     next_state <= ss_run_light;
109                 elsif load_pattern = '1' then
110                     next_state <= ss_load_pattern;
111                 else
112                     next_state <= ss_reset;
113                 end if;
114             when ss_stop_sys =>
115                 pattern_in <= (others => '0');
116                 load <= '0';
117                 rst_dec_cnt <= '0';
118                 rst_clk_leds <= '1';
119                 rst_clk_cnt <= '1';
120                 if start = '1' then
121                     next_state <= ss_run_light;
122                 elsif load_pattern = '1' then
123                     next_state <= ss_load_pattern;
124                 else
125                     next_state <= ss_stop_sys;
126                 end if;
127             when ss_run_light =>
128                 pattern_in <= (others => '0');
129                 load <= '0';
130                 rst_dec_cnt <= '0';
131                 rst_clk_leds <= '0';
132                 rst_clk_cnt <= '0';
133                 if stop_sys = '1' then
134                     next_state <= ss_stop_sys;
```

```
135                 elsif load_pattern = '1' then
136                     next_state <= ss_load_pattern;
137                 else
138                     next_state <= ss_run_light;
139                 end if;
140             when ss_load_pattern =>
141                 pattern_in <= dip_sws;
142                 load <= '1';
143                 rst_dec_cnt <= '1';
144                 rst_clk_leds <= '1';
145                 rst_clk_cnt <= '1';
146                 if start = '1' then
147                     next_state <= ss_run_light;
148                 else
149                     next_state <= ss_load_pattern;
150                 end if;
151         end case;
152     end process;
153 
154     process (clk, rst)
155     begin
156         if rst = '1' then
157             curr_state <= ss_reset;
158         elsif clk'event and clk = '1' then
159             curr_state <= next_state;
160         end if;
161     end process;
162 end behave;
```

Listing 11: Code in VHDL for `running_light`.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  
5  package const_types_pkg is
6  
7      constant fir_order: integer := 2;
8      constant adc_width: integer := 14;
9      subtype adc_num is signed (adc_width - 1 downto 0);
10     type tap_reg is array (0 to fir_order) of adc_num;
11 
12     subtype sev_seg_disps is std_logic_vector (20 downto 0);
13     constant ss_disp_low_pass : sev_seg_disps := "
          00011111101111110111";
14     constant ss_disp_high_pass: sev_seg_disps := "
          11101111101110111001";
15     constant ss_disp_band_pass: sev_seg_disps := "
          11101110001001110111";
16     constant ss_disp_band_stop: sev_seg_disps := "
          00011111101110111001";
```

```
17 end package;
```

Listing 12: Code in VHDL for the package used in the fir filter.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  use work.const_types_pkg.all;
6
7  entity band_pass_direct_vhdl is
8      port (
9          clk : in std_logic;
10         rst : in std_logic;
11         sig_in: in adc_num;
12         sig_out: out adc_num
13     );
14 end band_pass_direct_vhdl;
15
16 architecture behave of band_pass_direct_vhdl is
17
18     signal taps: tap_reg;
19
20 begin
21     process(clk, rst)
22     begin
23         if rst = '1' then
24             for i in 0 to fir_order loop
25                 taps(i) <= to_signed(0, adc_width);
26             end loop;
27         elsif clk'event and clk = '1' then
28             sig_out <= taps(2) / 2 - taps(0) / 2;
29             for i in 1 to fir_order loop
30                 taps(i) <= taps(i - 1);
31             end loop;
32             taps(0) <= sig_in;
33         end if;
34     end process;
35 end architecture;
```

Listing 13: Code in VHDL for the band pass filter.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  use work.const_types_pkg.all;
6
7  entity band_stop_transposed_vhdl is
8      port (
9          clk : in std_logic;
10         rst : in std_logic;
11         sig_in: in adc_num;
```

```
12          sig_out: out adc_num
13      );
14  end band_stop_transposed_vhdl;
15
16  architecture behave of band_stop_transposed_vhdl is
17
18      signal taps: tap_reg;
19
20  begin
21      process(clk, rst)
22      begin
23          if rst = '1' then
24              for i in 0 to fir_order loop
25                  taps(i) <= to_signed(0, adc_width);
26              end loop;
27          elsif clk'event and clk = '1' then
28              taps(2) <= sig_in / 2;
29              taps(1) <= taps(2);
30              taps(0) <= taps(1) + sig_in / 2;
31          end if;
32      end process;
33      sig_out <= taps(0);
34  end architecture;
```

Listing 14: Code in VHDL for the band stop filter.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   use work.const_types_pkg.all;
6
7   entity fir_filters_mux is
8       port (
9           clk : in std_logic;
10          rst : in std_logic;
11          sws : in std_logic_vector (3 downto 0);
12          sig_in_bp, sig_in_bs, sig_in_hp, sig_in_lp: in adc_num;
13          sig_out: out adc_num;
14          clk_adc: out std_logic;
15          ss_disps: out sev_seg_disps
16      );
17  end fir_filters_mux;
18
19  architecture behave of fir_filters_mux is
20
21      type fir_options is (band_pass, band_stop, high_pass,
            low_pass);
22      signal fir_select: fir_options;
23
24      -- f_clk_adc = f_clk_fpga / div_n
25      constant div_n: integer := 25;       -- fs/2 = 500 kHz
```

```
26      signal cnt: unsigned(4 downto 0);
27      signal pulse_reg: std_logic;
28
29  begin
30
31      sig_out <= sig_in_bp when fir_select = band_pass else
32                 sig_in_bs when fir_select = band_stop else
33                 sig_in_hp when fir_select = high_pass else
34                 sig_in_lp when fir_select = low_pass;
35
36      ss_disps <= ss_disp_band_pass when fir_select = band_pass
           else
37                  ss_disp_band_stop when fir_select = band_stop
                       else
38                  ss_disp_high_pass when fir_select = high_pass
                       else
39                  ss_disp_low_pass when fir_select = low_pass;
40
41      process(rst, clk)
42      begin
43          if rst = '1' then
44              fir_select <= band_pass;
45        elsif clk'event and clk = '1' then
46          if sws(0) = '0' then
47            fir_select <= band_pass;
48          elsif sws(1) = '0' then
49            fir_select <= band_stop;
50          elsif sws(2) = '0' then
51            fir_select <= high_pass;
52          elsif sws(3) = '0' then
53            fir_select <= low_pass;
54          end if;
55        end if;
56      end process;
57
58
59      process(clk, rst)
60      begin
61          if rst = '1' then
62              cnt <= (others => '0');
63              pulse_reg <= '0';
64              --n_prev <= n;
65          elsif clk'event and clk = '1' then
66              if cnt = to_unsigned(div_n - 1, cnt'length) then
67                  cnt <= (others => '0');
68                  pulse_reg <= '1';
69              else
70                  cnt <= cnt + 1;
71                  pulse_reg <= '0';
72              end if;
73          end if;
```

```
74      end process;
75
76      clk_adc <= pulse_reg;
77
78  end architecture;
```

Listing 15: Code in VHDL for the multiplexer uesd to select the filter.