Design and implementation of a reconfigurable FIR filter in FPGA

Gianmarco Nagaro Quiroz[1], Daniele Ninni[2], Lorenzo Valentini[3], and Emerson Rodriguez Vero Filho[4],
Department of Physics and Astronomy "Galileo Galilei", University of Padua, Italy
[1]gianmarco.nagaroquiroz@studenti.unipd.it, [2]daniele.ninni@studenti.unipd.it,
[3]lorenzo.valentini.2@studenti.unipd.it, [4]emerson.rodriguesverofilho@studenti.unipd.it.

FIR filters are among the most commonly used filters in signal processing. In this report the design and implementation of a FIR filter in FPGA is discussed. The performance of the implementation is evaluated by analyzing the response of the filter to various input sample signals generated in a Python environment. A USB serial interface, managed by a UART, carries the communication between computer and FPGA. The implementation in question allows updating the values of the FIR coefficients without having to rewrite the bitstream and, consequently, without having to reprogram the FPGA. The output of the FPGA is in agreement with the output of the Python implementation of the same filter, thus proving the goodness of the FPGA implementation. Furthermore, both outputs resemble the noiseless input signal, which proves that the filtering process respects all the characteristics established in the design phase.

***Index Terms*—FIR filter, FPGA, VHDL.**

I. Introduction

In this report the design and implementation of a FIR filter in FPGA is discussed. The corresponding code is written in VHDL while the synthesis and implementation is performed using "Vivado Design Suite", a software suite produced by Xilinx for synthesis and analysis of HDL (Hardware Description Language) designs.

The input sample signals are generated in a Python environment and then sent to the FPGA in order to be filtered. Afterwards, the FPGA output is imported and displayed in the Python environment itself. The communication between the Python client and the FPGA is achieved through a serial USB interface managed by a UART (Universal Asynchronous Receiver-Transmitter) device implemented in the FPGA itself. Data exchange and processing is managed by Python scripts implemented in a Jupyter notebook.

The implementation in question allows updating the values of the FIR coefficients used by the FIR filter entity of the FPGA in real time, i.e. without having to rewrite the bitstream and, consequently, without having to reprogram the FPGA via Vivado. This means that it is possible to update the key characteristics of the filter (i.e. sampling frequency, cutoff frequency, transition bandwidth and filter type) in a relatively simple and fast way. The FIR coefficients are generated through a Python script and then, after putting the FPGA in *setting mode*, they are sent to the FPGA itself through the UART device. It is possible to switch from *filtering mode* to *setting mode* and vice versa by moving the switch *SW0* of the FPGA: in other words, the position of *SW0* determines whether the input data must be treated as signals to be filtered or as FIR coefficients to be fixed for the filtering procedure.

Moreover, it is possible to update the number of taps of the FIR filter simply by updating the value of the constant `taps` defined in the VHDL code. However, in this case it is necessary to rewrite the bitstream and consequently reprogram the FPGA.

The performance of the implementation is evaluated by analyzing the response of the filter to various input sample signals generated in the Python environment. Furthermore, the goodness of the FPGA implementation is evaluated by comparing the output of the FPGA with the output of the Python implementation of the same filter, obtained through the function `lfilter` of the SciPy library.

II. FIR filter

[1] A FIR (Finite Impulse Response) filter is characterized by an impulse response (or response to any finite length input) of finite duration. In particular, the impulse response of an N-th order discrete-time FIR filter lasts exactly $N+1$ samples (from first nonzero element through last nonzero element) before it then settles to zero. This is in contrast to IIR (Infinite Impulse Response) filters, which may have internal feedback and may continue to respond indefinitely (usually decaying).

For a direct form discrete-time FIR filter of order N, each value of the output sequence is given by a computation known as discrete convolution, i.e. by a weighted sum of the most recent input values:

$$y[n] = \sum_{i=0}^{N} b_i \cdot x[n-i] \tag{1}$$

where:

- $x[n]$ is the input signal;
- $y[n]$ is the output signal;
- N is the filter order ($N+1$ terms on the right-hand side);
- b_i is the i-th coefficient of the filter.

The $x[n-i]$ in these terms are commonly referred to as *taps*, based on the structure of a tapped delay line that in many implementations or block diagrams provides the delayed inputs to the multiplication operations. One may speak of a 34-th order $\Leftrightarrow$ 35-tap filter, for instance.

III. Design

The response of a FIR filter is determined by both the number of FIR coefficients and the value of each of them. In the following, a FIR filter with the key characteristics reported in Table I is implemented.

TABLE I
KEY CHARACTERISTICS OF THE DESIGNED FIR FILTER.

Sampling frequency	48 kHz
Cutoff frequency	6 kHz
Transition bandwidth	6.5 kHz
Filter type	low-pass

The choice of the number of taps is a fundamental step of the FIR filter design. There are several online tools [2] that can guide the designer in making this choice. Compatibly with the key characteristic reported in Table I, the suggested number of taps is equal to

$$N + 1 = 35 \tag{2}$$

The values of the FIR coefficients are calculated in the Python environment using the function `firwin` provided by the SciPy library.

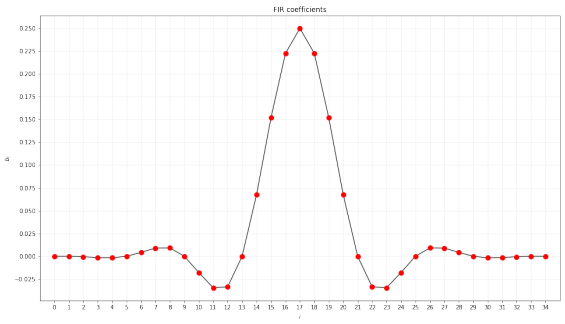

Fig. 1. Values of the FIR coefficients.

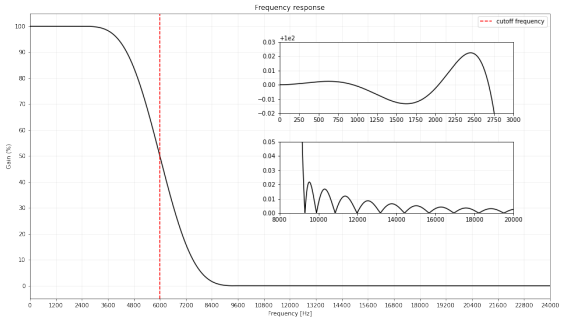

Fig. 2. Frequency response of the designed FIR filter.

IV. Implementation

The VHDL design sources and constraints are reported, respectively, in Appendix A and B.

TABLE II
VHDL ENTITIES.

Entity	Purpose
UART	Communication Python/FPGA
FIR filter	Signal filtering
TOP	Loopback and mode selection

A. UART entity

A UART (Universal Asynchronous Receiver-Transmitter) is a device for asynchronous serial communication in which the data format and transmission speeds are configurable. It sends data bits one by one, from the least significant to the most significant, framed by start and stop bits so that precise timing is handled by the communication channel.

In order for the data transmission to take place correctly, the so-called *baud-rate*, that is the transmission speed, must be established in advance. In this context, a baud-rate equal to $115200 \frac{bits}{s}$ is adopted.

The UART implemented in the FPGA allows the exchange of 8-bit integers between the Python client and the FPGA itself. Thus, the input sample signals are sent to the FPGA value-by-value as 8-bit signed integers. It follows that the input data range is equal to

$$[-2^{data\_width-1}, 2^{data\_width-1} + 1] = [-128, 127] \tag{3}$$

where $data\_width = 8$.
However, both the FIR coefficients and the sample input signals are generated in the Python environment as floating point numbers. On the other hand, FPGAs are known to be less efficient with floating point tasks, while they are better optimized for integer and binary operations. Therefore, all floating point values of the input data are rounded to integers before being sent to the FPGA. This is certainly an approximation and inevitably introduces an error in the filtering procedure. That said, given that the results turn out to be satisfactory, this error is unlikely to have a significant impact on the performance of the filter.

Regarding the FIR coefficients, a further step is necessary. In fact, they generally have a small value compared to 1, therefore rounding them to the nearest integer would almost always give a result equal to 0. This would obviously lead to a malfunction of the filter. Therefore, this problem is solved by normalizing the FIR coefficients to the input data range before sending them to the FPGA:

$$b'_i = input\_th \cdot b_i \tag{4}$$

where $input\_th = 2^{data\_width-1}$.
From the linearity of the discrete convolution performed by the FIR filter, it follows that in this way also the values of the output data are multiplied by $input\_th$:

$$y'[n] = \sum_{i=0}^{34} b'_i \cdot x[n-i] = input\_th \cdot \sum_{i=0}^{34} b_i \cdot x[n-i] \quad (5)$$

Therefore, given the constraint of 8-bit data transmissions, the output values of the FIR filter are divided by $input\_th$ before being sent to the Python environment:

$$y[n] = \frac{1}{input\_th} y'[n] = \sum_{i=0}^{34} b_i \cdot x[n-i] \quad (6)$$

This division by a power of 2 is performed using the VHDL function `shift_right`. Again, this division certainly introduces an error but, for the same reason as before, this error turns out to be negligible.

B. FIR filter entity

A VHDL package named `TYPES` is created. It contains the declaration of several user-defined constants and array-types which simplify the implementation of the FIR filter entity. This package is subsequently used both in the FIR filter entity and in the TOP entity: in this way, any change of the user-defined items made within the package `TYPES` (e.g. the constant `taps` representing the number of taps) is automatically applied to both entities without the need for further changes.

TABLE III
VHDL USER-DEFINED CONSTANTS.

Name	Type	Value
taps	integer	35
coeff_width	integer	8
data_width	integer	8
result_width	integer	data_width + coeff_width + int(ceil(log2(taps)))

TABLE IV
VHDL USER-DEFINED ARRAY TYPES.

Name	Type	Size
coeff_array	std_logic_vector[coeff_width]	taps
data_array	signed[data_width]	taps
product_array	signed[data_width + coeff_width]	taps

The FIR filter entity is implemented according to the following flow of operations. At each rising edge of the 100 MHz FPGA's clock:

1) the values of the 8-bit input sample data are stored progressively as signed values in `data_pipeline`, an array of type `data_array`: as soon as a new input is supplied to the FIR filter entity, the previous inputs shift one position to the right, the new input is stored in the position thus freed and the oldest input is discarded;
2) `data_pipeline` is multiplied element-wise by `coeffs`, an array of type `coeff_array`, and the result is stored in `products`, an array of type `product_array`;
3) each element of `products` is progressively added to `result`, a signed variable of size equal to `result_width`;
4) `result` is divided by `input_th` through the function `shift_right`, then it is resized to 8 bits through the function `resize` and finally it is assigned as `std_logic_vector` to `data_out`, the output of the FIR filter entity.

Note that the size of the VHDL user-defined constants and array types are set according to the following rules for binary operations:

- size(a + b) = max(size(a), size(b)) + 1
- size($a \cdot b$) = size(a) + size(b)

C. TOP entity

The top entity takes care of looping back the UART entity and the FIR filter entity according to the following flow of operations:

1) the UART receiver receives 8-bit input sample data from the Python client via the serial USB interface;
2) the UART receiver provides such input data to the FIR filter, together with a validation signal that gives the green light for the filter itself to start the filtering process;
3) the output of the filtering process is sent to the UART transmitter, together with a validation signal that gives the green light for the UART transmitter to send the output itself to the Python client;
4) the UART transmitter sends the output of the FIR filter to the Python client via the serial USB interface.

The implementation in question allows updating the values of the FIR coefficients used by the FIR filter entity in real time, i.e. without having to rewrite the bitstream and, consequently, without having to reprogram the FPGA via Vivado. The FIR coefficients are generated through a Python script and then, after putting the FPGA in *setting mode*, they are sent to the FPGA itself through the UART. It is possible to switch from *filtering mode* to *setting mode* and vice versa by moving the switch *SW0* of the FPGA. In particular:

- *setting mode* (switch $SW0$ off $\rightarrow$ led $LD0$ red):
  the input data are interpreted as FIR coefficients, therefore they are stored in `coeffs` and then used by the FIR filter entity for discrete convolution operations: if the acquisition of the coefficients is completed successfully, the led $LD0$ turns green;
- *filtering mode* (switch $SW0$ on $\rightarrow$ led $LD0$ blue):
  the input data are interpreted as values of the input sample signals, therefore they are stored in `data_pipeline` and then processed by the FIR filter entity in order to generate the filtered output signal.

Note that, when switching from one mode to another, `data_pipeline`, `products` and `results` are reset.

V. Results

Here are reported the results concerning the filtering of various input sample signals generated in the Python environment. A sinusoidal noise, characterized by a frequency higher than the FIR filter cutoff frequency, is added to each of the input sample signals in order to evaluate the performance of the implementation. Furthermore, the goodness of the FPGA implementation is evaluated by comparing the output of the FPGA with the output of the Python implementation of the same filter, obtained through the function `lfilter` of the SciPy library.

TABLE V
KEY CHARACTERISTICS OF THE INPUT SAMPLE SIGNALS.

Frequency	1 kHz
Number of sampled complete oscillations	5
Number of samples	240

TABLE VI
KEY CHARACTERISTICS OF THE SINUSOIDAL NOISE.

Amplitude	0.5
Frequency	15 kHz
Initial phase	0.8

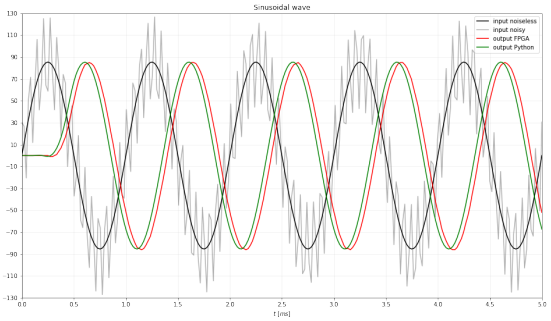

Fig. 3. Sinusoidal wave.

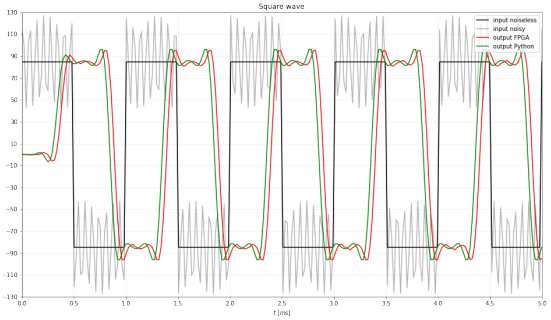

Fig. 4. Square wave.

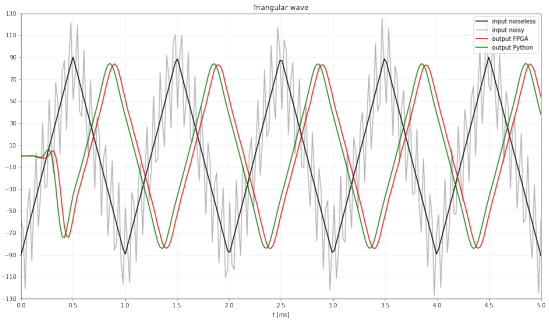

Fig. 5. Triangular wave.

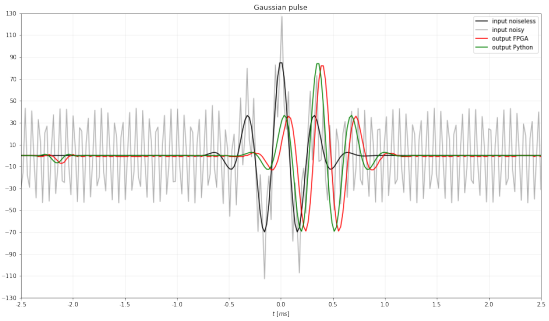

Fig. 6. Gaussian pulse.

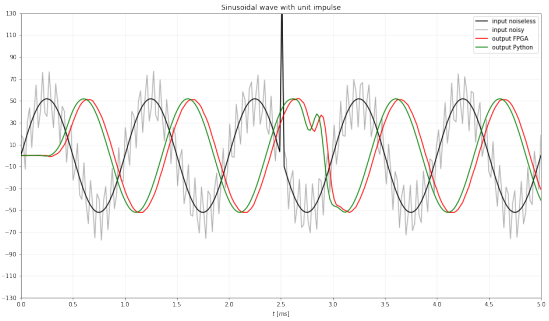

Fig. 7. Sinusoidal wave with unit impulse.

In all cases the Python output signal is delayed with respect to the input sample signal. This could be due to the so-called group delay of the FIR filter, due to the delay line of the filter and therefore dependent on the number of taps of the filter itself.

Moreover, the FPGA output signal is systematically delayed with respect to the Python output signal. This could be due to the serial USB interface managed by the UART.

Aside from that, the FPGA output is in agreement with the Python output, thus proving the goodness of the FPGA implementation of the FIR filter. Furthermore, both outputs resemble the noiseless input signal, which proves that the

filtering process respects all the characteristics established in the design phase. In other words, the FIR filter thus implemented is able to effectively attenuate the high frequency sinusoidal noise component and returns an output signal which is sufficiently compatible with the corresponding noiseless input signal.

Any small discrepancies could be due to the fact that:

- the floating point values of the input data are rounded to integers before being sent to the FPGA;
- FIR coefficients are normalized to the input data range before being sent to the FPGA.

Both sources of error are discussed in IV-A.

VI. Conclusions

Given the results obtained in terms of signals effectively filtered by the FPGA and given the evident compatibility between the FPGA output and the Python output, it is reasonable to argue that the implementation of the FIR filter in FPGA has been performed successfully.

Furthermore, the possibility of being able to update, on the one hand, the values of the FIR coefficients in real time and, on the other hand, the number of taps simply by updating the value of the constant `taps` defined in the VHDL code, undoubtedly constitute an added value and give the implementation versatility and practicality.

Possible improvements could consist of:

- developing an implementation that allows the use of N-bit integers with $N > 8$ in order to reduce the error introduced by the rounding of the floating point input values to integers;
- developing an implementation that involves the use of multiple FIR filters in cascade in order to increase the precision of the filtering procedure and, at the same time, simplify the architecture of the single filter.

References

[1] Finite impulse response. [Online]. Available: https://en.wikipedia.org/wiki/Finite_impulse_response

[2] FIIIR! - Design FIR and IIR Filters. [Online]. Available: https://fiiir.com

APPENDIX A
VHDL DESIGN SOURCES

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

package TYPES is

  constant taps         : integer := 35;                                                    -- number of FIR coefficients
  constant coeff_width  : integer := 8;                                                     -- width of each FIR coefficient
  constant data_width   : integer := 8;                                                     -- width of input/output data
  constant result_width : integer := data_width + coeff_width + integer(ceil(log2(real(taps)))); -- width of FIR filter result

  type coeff_array   is array (0 to taps-1) of std_logic_vector(coeff_width-1 downto 0);    -- array of FIR coefficients
  type data_array    is array (0 to taps-1) of signed(data_width-1 downto 0);               -- array of data
  type product_array is array (0 to taps-1) of signed((data_width + coeff_width)-1 downto 0); -- array of (coefficient * data) products

end package TYPES;

-------------------------------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.MATH_REAL.ALL;
use WORK.TYPES.ALL;

entity fir_filter is
  port (clock     : in  std_logic;
        reset     : in  std_logic;
        coeffs    : in  coeff_array;
        data_in   : in  std_logic_vector(data_width-1 downto 0);
        valid_in  : in  std_logic;
        data_out  : out std_logic_vector(data_width-1 downto 0);
        valid_out : out std_logic);
end fir_filter;

architecture Behavioral of fir_filter is

  signal data_pipeline : data_array    := (others => (others => '0'));
  signal products      : product_array := (others => (others => '0'));

begin

  main : process (clock, reset) is

  variable result : signed(result_width-1 downto 0);

  begin -- process main
    if reset = '0' then

      data_pipeline <= (others => (others => '0'));
      products      <= (others => (others => '0'));
      result        := (others => '0');

    elsif rising_edge(clock) then

      if valid_in = '1' then

        data_pipeline <= signed(data_in) & data_pipeline(0 to taps-2); -- shift old data inside data_pipeline to insert data_in

        result := (others => '0');                                  -- initialize result to 0
        for i in 0 to taps-1 loop                                   -- for each FIR coefficient
          products(i) <= signed(coeffs(i)) * data_pipeline(i);      -- compute product
          result      := result + products(i);                      -- add product to result
        end loop;

        data_out  <= std_logic_vector(resize(shift_right(result, data_width-1), data_width));
        valid_out <= '1';

      else

        valid_out <= '0';

      end if;
    end if;
```

```
  end process main;

end Behavioral;
```

Listing 1. FIR filter entity.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use WORK.TYPES.ALL;

entity top is
  port (clock              : in  std_logic;
        mode_sel_in        : in  std_logic;
        uart_txd_in        : in  std_logic;
        led_setting_coeffs : out std_logic;
        led_ready          : out std_logic;
        led_filtering      : out std_logic;
        uart_rxd_out       : out std_logic);
end entity top;

architecture str of top is

  component uart_receiver is
    port (clock         : in  std_logic;
          uart_rx       : in  std_logic;
          valid         : out std_logic;
          received_data : out std_logic_vector(7 downto 0));
  end component uart_receiver;

  component fir_filter is
    port (clock     : in  std_logic;
          reset     : in  std_logic;
          coeffs    : in  coeff_array;
          data_in   : in  std_logic_vector(data_width-1 downto 0);
          valid_in  : in  std_logic;
          data_out  : out std_logic_vector(data_width-1 downto 0);
          valid_out : out std_logic);
  end component fir_filter;

  component uart_transmitter is
    port (clock        : in  std_logic;
          data_to_send : in  std_logic_vector(7 downto 0);
          data_valid   : in  std_logic;
          busy         : out std_logic;
          uart_tx      : out std_logic);
  end component uart_transmitter;

  signal reset        : std_logic;
  signal valid_in     : std_logic;
  signal data_in      : std_logic_vector(data_width-1 downto 0);
  signal coeffs       : coeff_array;
  signal data_out     : std_logic_vector(data_width-1 downto 0);
  signal valid_out    : std_logic;
  signal busy         : std_logic;
  signal n_coeffs_set : integer := 0; -- number of FIR coefficients already set

begin -- architecture str

  uart_receiver_1 : uart_receiver port map (clock         => clock,
                                            uart_rx       => uart_txd_in,
                                            valid         => valid_in,
                                            received_data => data_in);

  fir_filter_1 : fir_filter port map (clock     => clock,
                                      reset     => reset,
                                      coeffs    => coeffs,
                                      data_in   => data_in,
                                      valid_in  => valid_in,
                                      data_out  => data_out,
                                      valid_out => valid_out);

  uart_transmitter_1 : uart_transmitter port map (clock        => clock,
                                                  data_to_send => data_out,
                                                  data_valid   => valid_out,
                                                  busy         => busy,
                                                  uart_tx      => uart_rxd_out);

  main : process (clock) is
  begin -- process main
    if rising_edge(clock) then
```

```
76
77        -- filtering mode
78        if mode_sel_in = '1' then
79
80          led_setting_coeffs <= '0';
81          led_ready          <= '0';
82          led_filtering      <= '1';
83          reset              <= '1';
84          n_coeffs_set       <=  0;
85
86        -- setting mode
87        else
88
89          led_setting_coeffs <= '1';
90          led_filtering      <= '0';
91          reset              <= '0';
92
93          if n_coeffs_set = taps then
94
95            led_ready <= '1';
96
97          elsif valid_in = '1' then
98
99            coeffs(n_coeffs_set) <= data_in;
100           n_coeffs_set         <= n_coeffs_set + 1;
101           led_ready            <= '0';
102
103         end if;
104       end if;
105     end if;
106   end process main;
107
108 end architecture str;
```

Listing 2. TOP entity.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity uart_receiver is
5    port (clock         : in  std_logic;
6          uart_rx       : in  std_logic;
7          valid         : out std_logic;
8          received_data : out std_logic_vector(7 downto 0));
9  end entity uart_receiver;
10
11 architecture str of uart_receiver is
12
13   type state_t is (idle_s, start_s, bit0_s, bit1_s, bit2_s, bit3_s, bit4_s, bit5_s, bit6_s, bit7_s, bit8_s, stop_s);
14   signal state : state_t := idle_s;
15
16   signal baudrate_out    : std_logic;
17   signal received_data_s : std_logic_vector(7 downto 0);
18
19   component sampler_generator is
20     port (clock        : in  std_logic;
21           uart_rx      : in  std_logic;
22           baudrate_out : out std_logic);
23   end component sampler_generator;
24
25 begin -- architecture str
26
27   sampler_generator_1 : sampler_generator port map (clock        => clock,
28                                                     uart_rx      => uart_rx,
29                                                     baudrate_out => baudrate_out);
30
31   main : process (clock) is
32   begin -- process main
33     if rising_edge(clock) then
34       case state is
35
36         when idle_s =>
37           received_data_s <= (others => '0');
38           valid           <= '0';
39           if uart_rx = '0' then
40             state <= start_s;
41           end if;
42
43         when start_s =>
44           if baudrate_out = '1' then
45             received_data_s(0) <= uart_rx;
```

```
            state              <= bit0_s;
          end if;

        when bit0_s =>
          if baudrate_out = '1' then
            received_data_s(1) <= uart_rx;
            state              <= bit1_s;
          end if;

        when bit1_s =>
          if baudrate_out = '1' then
            received_data_s(2) <= uart_rx;
            state              <= bit2_s;
          end if;

        when bit2_s =>
          if baudrate_out = '1' then
            received_data_s(3) <= uart_rx;
            state              <= bit3_s;
          end if;

        when bit3_s =>
          if baudrate_out = '1' then
            received_data_s(4) <= uart_rx;
            state              <= bit4_s;
          end if;

        when bit4_s =>
          if baudrate_out = '1' then
            received_data_s(5) <= uart_rx;
            state              <= bit5_s;
          end if;

        when bit5_s =>
          if baudrate_out = '1' then
            received_data_s(6) <= uart_rx;
            state              <= bit6_s;
          end if;

        when bit6_s =>
          if baudrate_out = '1' then
            received_data_s(7) <= uart_rx;
            state              <= bit7_s;
          end if;

        when bit7_s =>
          if baudrate_out = '1' then
            valid         <= '1';
            received_data <= received_data_s;
            state         <= idle_s;
          end if;

        when others => null;

      end case;
    end if;
  end process main;

end architecture str;
```

Listing 3. UART receiver entity.

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

entity uart_transmitter is
  port (clock        : in  std_logic;
        data_to_send : in  std_logic_vector(7 downto 0);
        data_valid   : in  std_logic;
        busy         : out std_logic;
        uart_tx      : out std_logic);
end entity uart_transmitter;

architecture rtl of uart_transmitter is

  component baudrate_generator is
    port (clock        : in  std_logic;
          baudrate_out : out std_logic);
  end component baudrate_generator;

```

```
19    type state_t is (idle_s, data_valid_s, start_s, bit0_s, bit1_s, bit2_s, bit3_s, bit4_s, bit5_s, bit6_s, bit7_s, bit8_s,
         stop_s);
20    signal state : state_t := idle_s;
21
22    signal baudrate_out : std_logic;
23
24  begin -- architecture rtl
25
26    baudrate_generator_1 : baudrate_generator port map (clock        => clock,
27                                                        baudrate_out => baudrate_out);
28
29    main_state_machine : process (clock) is
30    begin -- process main_state_machine
31      if rising_edge(clock) then
32        case state is
33
34          when idle_s =>
35            busy    <= '0';
36            uart_tx <= '1';
37            if data_valid = '1' then
38              state <= data_valid_s;
39            end if;
40
41          when data_valid_s =>
42            busy <= '1';
43            if baudrate_out = '1' then
44              state <= start_s;
45            end if;
46
47          when start_s =>
48            uart_tx <= '0';
49            if baudrate_out = '1' then
50              state <= bit0_s;
51            end if;
52
53          when bit0_s =>
54            uart_tx <= data_to_send(0);
55            if baudrate_out = '1' then
56              state <= bit1_s;
57            end if;
58
59          when bit1_s =>
60            uart_tx <= data_to_send(1);
61            if baudrate_out = '1' then
62              state <= bit2_s;
63            end if;
64
65          when bit2_s =>
66            uart_tx <= data_to_send(2);
67            if baudrate_out = '1' then
68              state <= bit3_s;
69            end if;
70
71          when bit3_s =>
72            uart_tx <= data_to_send(3);
73            if baudrate_out = '1' then
74              state <= bit4_s;
75            end if;
76
77          when bit4_s =>
78            uart_tx <= data_to_send(4);
79            if baudrate_out = '1' then
80              state <= bit5_s;
81            end if;
82
83          when bit5_s =>
84            uart_tx <= data_to_send(5);
85            if baudrate_out = '1' then
86              state <= bit6_s;
87            end if;
88
89          when bit6_s =>
90            uart_tx <= data_to_send(6);
91            if baudrate_out = '1' then
92              state <= bit7_s;
93            end if;
94
95          when bit7_s =>
96            uart_tx <= data_to_send(7);
97            if baudrate_out = '1' then
98              state <= stop_s;
99            end if;
100
101         when stop_s =>
102           uart_tx <= '1';
103           if baudrate_out = '1' then
```

```
            state <= idle_s;
          end if;

        when others => null;

      end case;
    end if;
  end process main_state_machine;

end architecture rtl;
```

Listing 4. UART transmitter entity.

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

entity baudrate_generator is
  port (clock        : in  std_logic;
        baudrate_out : out std_logic);
end entity baudrate_generator;

architecture rtl of baudrate_generator is

  signal counter   : unsigned(9 downto 0) := (others => '0');
  constant divisor : unsigned(9 downto 0) := to_unsigned(867, 10);

begin -- architecture rtl

  main : process (clock) is
  begin -- process main
    if rising_edge(clock) then
      counter <= counter + 1;
      if counter = divisor then
        baudrate_out <= '1';
        counter      <= (others => '0');
      else
        baudrate_out <= '0';
      end if;
    end if;
  end process main;

end architecture rtl;
```

Listing 5. Baud-rate generator entity.

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

entity sampler_generator is
  port (clock        : in  std_logic;
        uart_rx      : in  std_logic;
        baudrate_out : out std_logic);
end entity sampler_generator;

architecture rtl of sampler_generator is

  type state_t is (idle_s, start_s, bit0_s, bit1_s, bit2_s, bit3_s, bit4_s, bit5_s, bit6_s, bit7_s, bit8_s, stop_s);
  signal state : state_t := idle_s;

  signal counter        : unsigned(10 downto 0) := (others => '0');
  signal delay_counter  : unsigned(10 downto 0) := (others => '0');
  constant divisor      : unsigned(10 downto 0) := to_unsigned(867, 11);
  constant half_divisor : unsigned(10 downto 0) := to_unsigned(433, 11);
  signal busy           : std_logic             := '0';
  signal pulse_out      : std_logic;
  signal enable_counter : std_logic             := '0';
  signal enable_delay   : std_logic             := '0';

begin

  pulse_generator : process (clock) is
  begin -- process pulse_generator
    if rising_edge(clock) then
      if enable_counter = '1' then
```

```
31        counter <= counter + 1;
32        if counter = divisor then
33          pulse_out <= '1';
34          counter   <= (others => '0');
35        else
36          pulse_out <= '0';
37        end if;
38      else
39        counter <= (others => '0');
40      end if;
41    end if;
42  end process pulse_generator;
43
44  state_machine : process (clock) is
45  begin -- process state_machine
46    if rising_edge(clock) then
47      case state is
48
49        when idle_s =>
50          enable_counter <= '0';
51          if uart_rx = '0' then
52            state <= start_s;
53          end if;
54
55        when start_s =>
56          -- enable baudrate_generator
57          enable_counter <= '1';
58          if pulse_out = '1' then
59            state <= bit0_s;
60          end if;
61
62        when bit0_s =>
63          if pulse_out = '1' then
64            state <= bit1_s;
65          end if;
66
67        when bit1_s =>
68          if pulse_out = '1' then
69            state <= bit2_s;
70          end if;
71
72        when bit2_s =>
73          if pulse_out = '1' then
74            state <= bit3_s;
75          end if;
76
77        when bit3_s =>
78          if pulse_out = '1' then
79            state <= bit4_s;
80          end if;
81
82        when bit4_s =>
83          if pulse_out = '1' then
84            state <= bit5_s;
85          end if;
86
87        when bit5_s =>
88          if pulse_out = '1' then
89            state <= bit6_s;
90          end if;
91
92        when bit6_s =>
93          if pulse_out = '1' then
94            state <= bit7_s;
95          end if;
96
97        when bit7_s =>
98          if pulse_out = '1' then
99            state <= idle_s;
100         end if;
101
102       when others => null;
103
104     end case;
105   end if;
106 end process state_machine;
107
108 delay_line : process (clock) is
109 begin -- process delay_line
110   if rising_edge(clock) then
111     if pulse_out = '1' then
112       -- start count
113       enable_delay <= '1';
114     end if;
115     if delay_counter = half_divisor then
116       enable_delay <= '0';
```

```
117         baudrate_out <= '1';
118       -- end count
119       else
120         baudrate_out <= '0';
121       end if;
122       if enable_delay = '1' then
123         delay_counter <= delay_counter + 1;
124       else
125         delay_counter <= (others => '0');
126       end if;
127     end if;
128   end process delay_line;
129
130 end architecture rtl;
```

Listing 6. Sampler generator entity.

APPENDIX B
VHDL CONSTRAINTS

```
1  ## Clock signal
2  set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 }      [get_ports { clock }]; #IO_L12P_T1_MRCC_35 Sch=gclk[100]
3  create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports { clock }];
4
5  ## Switches
6  set_property -dict { PACKAGE_PIN A8    IOSTANDARD LVCMOS33 }      [get_ports { mode_sel_in }]; #IO_L12N_T1_MRCC_16 Sch=sw
       [0]
7
8  ## RGB LEDs
9  set_property -dict { PACKAGE_PIN E1    IOSTANDARD LVCMOS33 }      [get_ports { led_filtering }]; #IO_L18N_T2_35 Sch=led0_b
10 set_property -dict { PACKAGE_PIN F6    IOSTANDARD LVCMOS33 }      [get_ports { led_ready }]; #IO_L19N_T3_VREF_35 Sch=
       led0_g
11 set_property -dict { PACKAGE_PIN G6    IOSTANDARD LVCMOS33 }      [get_ports { led_setting_coeffs }]; #IO_L19P_T3_35 Sch=
       led0_r
12
13 ## USB-UART Interface
14 set_property -dict { PACKAGE_PIN D10   IOSTANDARD LVCMOS33 }      [get_ports { uart_rxd_out }]; #IO_L19N_T3_VREF_16 Sch=
       uart_rxd_out
15 set_property -dict { PACKAGE_PIN A9    IOSTANDARD LVCMOS33 }      [get_ports { uart_txd_in }]; #IO_L14N_T2_SRCC_16 Sch=
       uart_txd_in
```