Report on Management and Analysis of Physics Dataset project

Project: Registers

Exam date: February 25, 2022

Peri Andrea
andrea.peri.1@studenti.unipd.it

Choubabi Salah
salaheldine.choubabi@studenti.unipd.it

Pedrazzi Matteo
matteo.pedrazzi@studenti.unipd.it

Seyedeh Maryam
seyedehmaryam.hashemitaklimi@studenti.unipd.it

Table of contents

1. Scope and objectives

2. UART module
   - 2.1 Receiver
   - 2.2 Transmitter

3. Overall look at the project
   - 3.1 Main register: parallel to parallel register
   - 3.2 Second register: two’s complement register
   - 3.3 Third register: parallel-to-serial shift register
   - 3.4 Multiplexer 3-to-1

4. Constraint

5. Python script

6. Testing

1. Scope and objectives

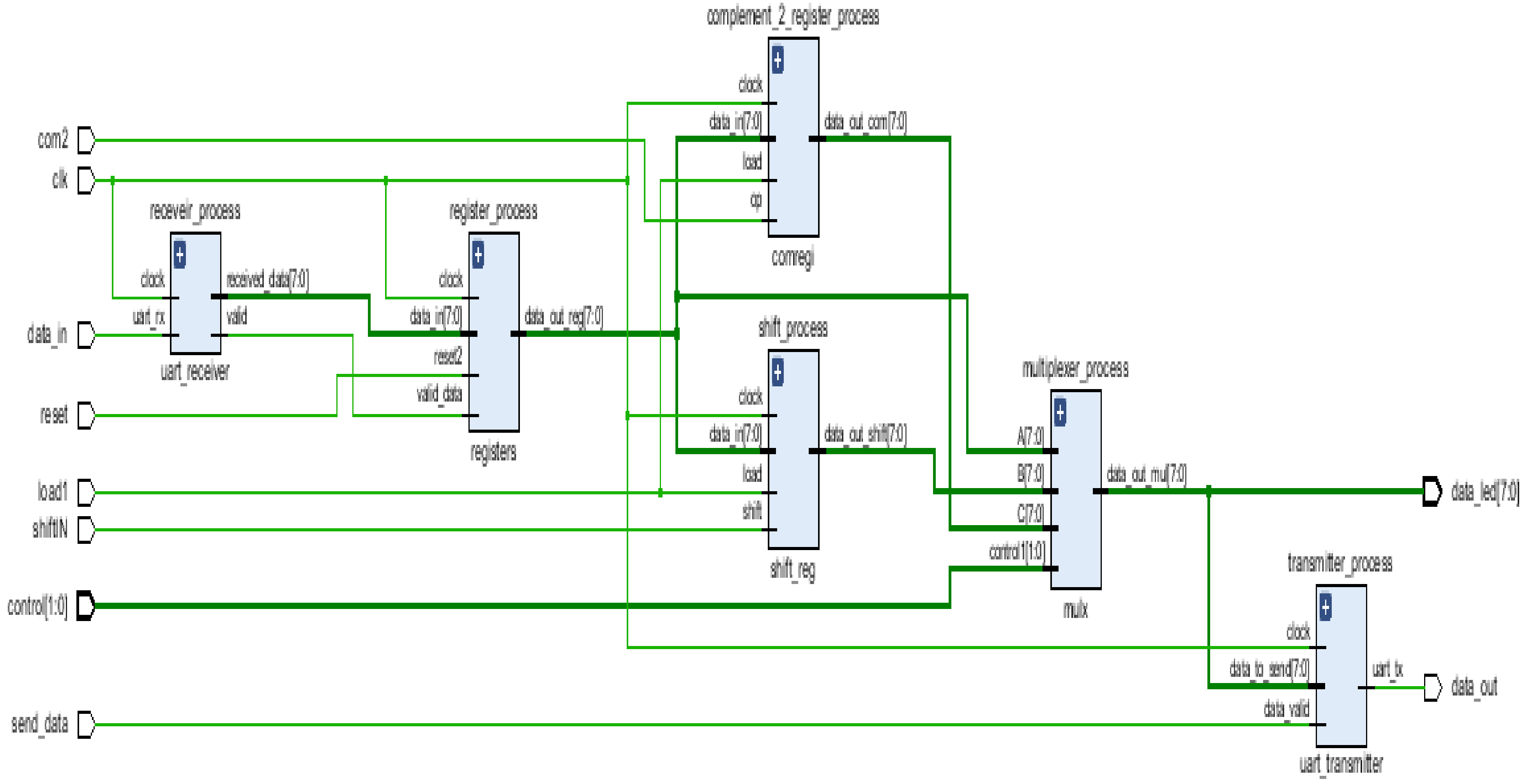
We discuss here the results of the project developed, which consists in an implementation of three different registers used to accomplish different tasks. Each register takes in input a set of eight bit sent from a PC exploiting a python script. Data sent from the computer are firstly imported as an array of bits from the UART module, which is used in order to save incoming bits and put them in an array.

We implemented three different registers:

- The first one consists of a register parallel-to-parallel that allows us to record the data sent to us by the computer. This register also has a reset input that allows the register to be initialised with a predefined value. This register is directly connected to the receiver and this allows us to save the byte that was sent to us.
- The second one is a register which allows a shift of the byte supplied to it. This register is connected to the first register and through a variable, we can load the value of the first register into this one.
- The third register one is used to perform a 2's complement on the incoming byte. Like the shift register, this one is also connected to the first register, and it is up to the user to load the byte of the first register into this one using an input signal load

In addition to these three registers, we have also used a multiplexer, controlled by a control signal operated through the use of two switches. The latter allows two actions to be carried out:

- It allows the output value of a register to be observed on the LEDs.
- to select the byte in one of the three registers that we wish to transmit to the PC.



complement_2_register_process
clock
data_in[7:0]
load
cp
data_out_com[7:0]
comregi
com2
clk
receveir_process
register_process
clock
received_data[7:0]
uart_rx
valid
uart_receiver
clock
data_in[7:0]
reset2
valid_data
data_out_reg[7:0]
registers
data_in
reset
shift_process
clock
data_in[7:0]
load
shift
data_out_shift[7:0]
shift_reg
multiplexer_process
A[7:0]
B[7:0]
C[7:0]
control1[1:0]
data_out_mu[7:0]
mux
data_led[7:0]
load1
shiftIN
control[1:0]
transmitter_process
clock
data_to_send[7:0]
data_valid
uart_tx
uart_transmitter
data_out
send_data

2. UART module

Communication between the PC and the FPGA is ensured through the use of the UART protocol. This protocol allows data in parallel to be converted into data in series and transferred via the USB cable. In our project, we have used two entities: a receiver which receives the data sent by the computer, and a transmitter which sends the data to the computer.

2.1 Receiver

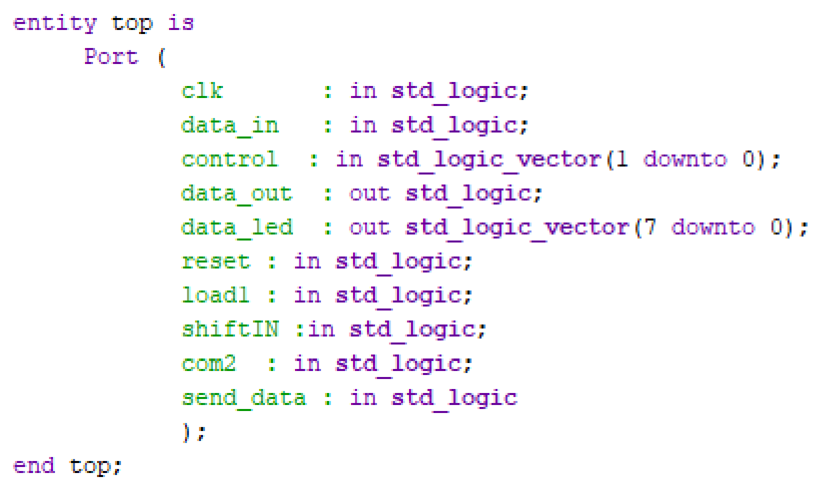
The receiver consists in turn of two submodules: the first is the state machine used to import data from the PC: the results is the conversion of a sequence of bits in a parallel register. The second submodule is a sampler generator: generating an internal baudrate and shifting the signal of half of the baudrate period, it allows to read data in the best moment possible, which is at the half time of the signal coming from the input port.
We wrote some little modifications to the UART receiver state machine. Firstly, we use two control signals to regulate the reception phase: the receiver only starts saving data when the busy signal is 0, and when the eight-bit array is imported it emits a short pulse in the valid channel. This valid signal acts like a set for the buffer implemented, and it allows to set the busy channel to 1 until the transmitter has finished to transmit data.
We also added another state at the top of the state machine; this state does anything but allowing the valid pulse to stay high for two clock cycles, in order to guarantee a solid set option to the various registers

2.2 Transmitter

The transmitter also consists in two different submodules, which are a state machine which convert data from parallel to serial and sends them to the output port, and a baudrate generator, needed to synchronize the FPGA and the PC. Also, in this case we use two control signals, which are a valid signal needed to let the transmission start, and a busy signal which consists in a short pulse (two clock periods) activated when the transmitter finishes to send data and is used to let the receiver know that the input wire is free and to start the receiving process. Note that also here was added another state on the top of the state machine to delay the busy pulse.

3. Overall look at the project

In this section we will discuss in detail the implementation of the three different registers used. Let's firstly have a look at the entity created for the purpose:



```
entity top is
     Port (
            clk        : in std_logic;
            data_in    : in std_logic;
            control   : in std_logic_vector(1 downto 0);
            data_out   : out std_logic;
            data_led   : out std_logic_vector(7 downto 0);
            reset : in std_logic;
            load1 : in std_logic;
            shiftIN :in std_logic;
            com2   : in std_logic;
            send_data : in std_logic
            );
end top;
```

Our top entity has many inputs and outputs that allow it to carry out a series of actions.

-clk : corresponds to the clock signal of the entire entity

-data_in : the data that is sent from the PC via the USB cable.

-control : a two bit signal that allows you to select the register whose output you want to observe on the leds and the value of the register that you want to send to the PC.

-reset: a control bit that allows to initialise the value of the first register to "00000000".

-load1: a signal that is activated through the use of a button and which allows the value of the first register to be loaded into the other two registers

-shiftN: a control bit that allows a shift operation to be carried out on the byte present on the input of the shift register.

-com2: a control bit that allows a mathematical 2's complement operation to be performed on the byte present at the input of the third register

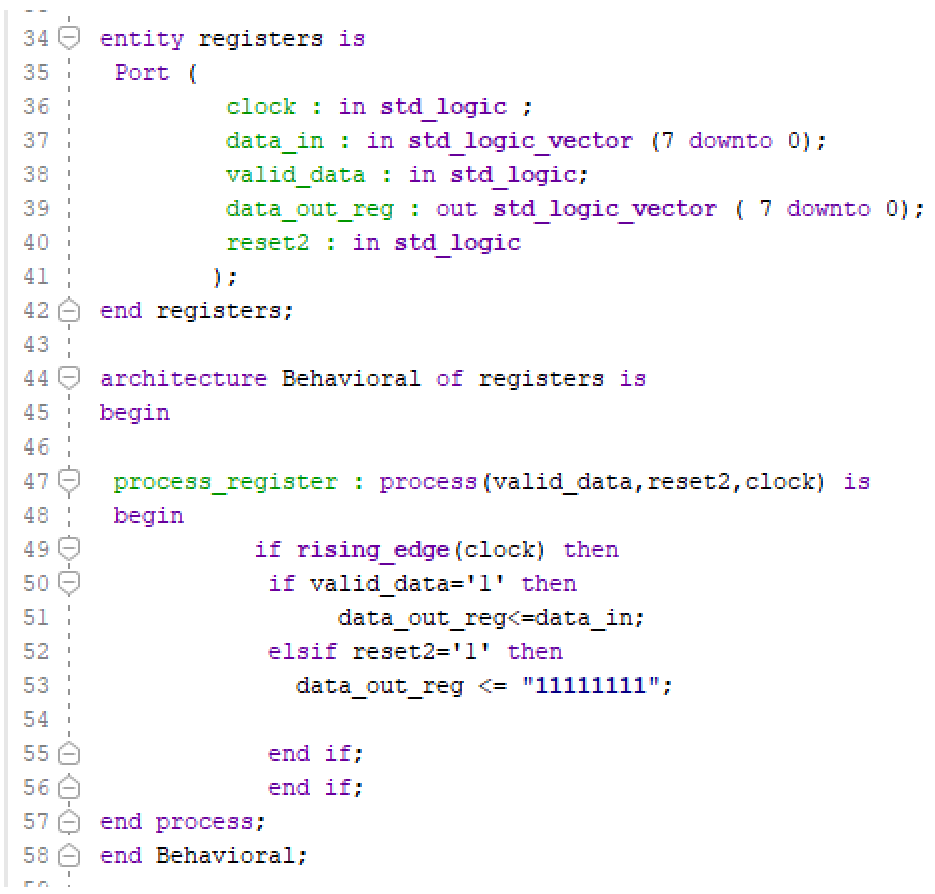
-Send_data: signal that allows the byte present in one of the three registers to be sent to the PC through the use of the UART transmitter.

-Data_out: output signal that corresponds to the data sent from the FPGA to the PC.

-Data_led: a vector of 8 output bits connected to eight leds that allows to observe the value of one of the three registers.

3.1 Main register: parallel to parallel register

This register is a simple parallel-to-parallel register provided with a set signal which is again the valid_out signal provided by the receiver.



```
entity registers is
 Port (
        clock : in std_logic ;
        data_in : in std_logic_vector (7 downto 0);
        valid_data : in std_logic;
        data_out_reg : out std_logic_vector ( 7 downto 0);
        reset2 : in std_logic
       );
end registers;

architecture Behavioral of registers is
begin

 process_register : process(valid_data,reset2,clock) is
 begin
           if rising_edge(clock) then
            if valid_data='1' then
                 data_out_reg<=data_in;
            elsif reset2='1' then
              data_out_reg <= "11111111";

            end if;
            end if;
end process;
end Behavioral;
```

The first and perhaps most important register is the classic parallel-to-parallel synchronous register. This register is directly connected to the UART receiver and is used to store data sent from the computer. In addition, this register has a synchronous reset that allows the value of the register to be initialised to a predefined value.

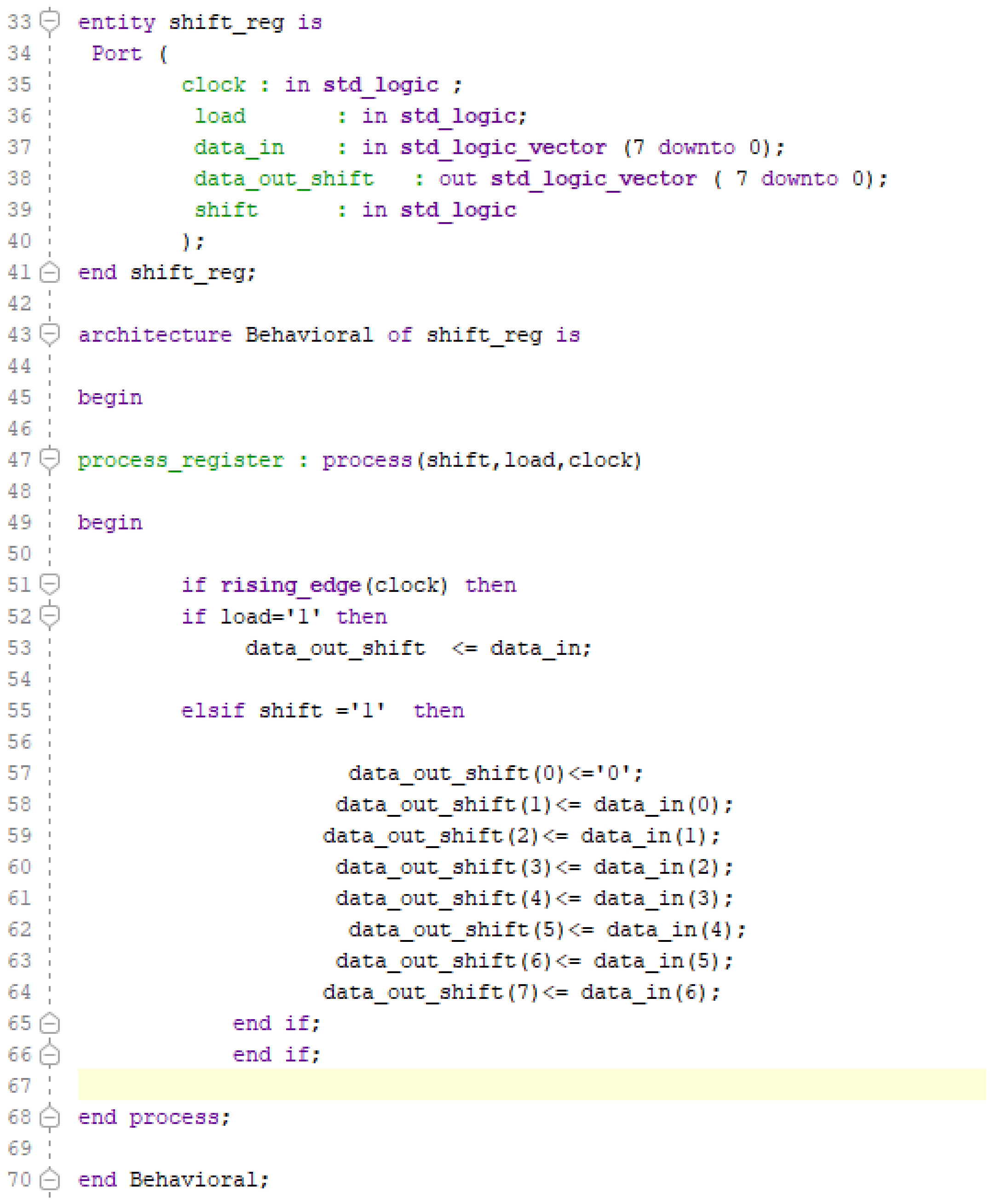
3.2 Second register: two’s complement register

```vhdl
entity comregi is
 Port (
        clock : in std_logic;
        load       : in std_logic;
         data_in     : in std_logic_vector (7 downto 0);
         data_out_com    : out std_logic_vector ( 7 downto 0);
         op       : in std_logic

        );
end comregi;

architecture Behavioral of comregi is
--signal data : std_logic_vector (7 downto 0);


begin

process_register : process(clock,op,load) is
begin
    if rising_edge(clock) then
           if load='1' then
            data_out_com <= data_in ;
            elsif op='1' then
                data_out_com <= std_logic_vector(unsigned(not(data_in))+1);

        end if;
     end if;
end process;

end Behavioral;
```

The second register that is used in our entity is a register that allows a two-way complement to be made on the data contained in the register. The input of this register is linked to the first register. Thanks to the use of the input load, we can load the value of the main register into this register. The input op, on the other hand, allows us to perform a two-tuple operation on the bytes we have in the input of the register.

3.3 Third register: parallel-to-serial shift register

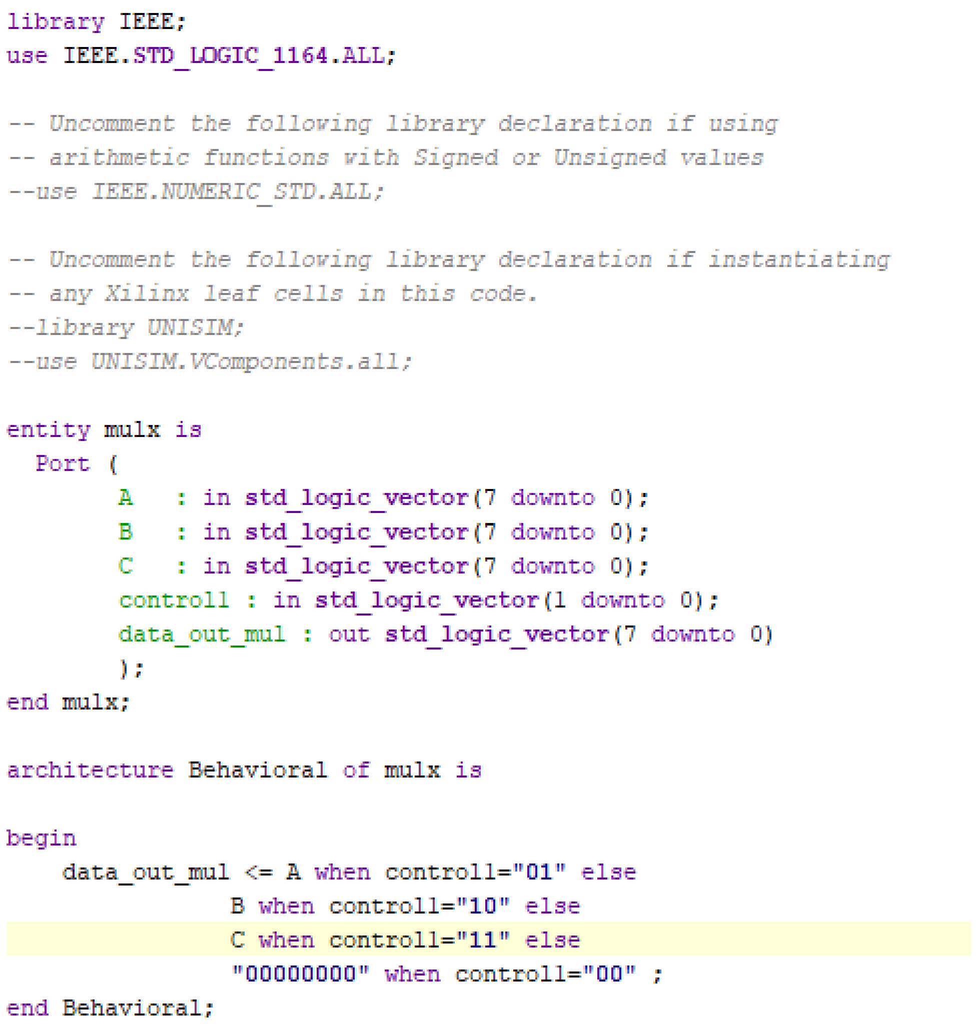
The last register implemented consists in a parallel-to-parallel shift register:



```
entity shift_reg is
 Port (
        clock : in std_logic ;
         load        : in std_logic;
         data_in     : in std_logic_vector (7 downto 0);
         data_out_shift    : out std_logic_vector ( 7 downto 0);
         shift       : in std_logic
        );
end shift_reg;

architecture Behavioral of shift_reg is

begin

process_register : process(shift,load,clock)

begin

        if rising_edge(clock) then
        if load='1' then
             data_out_shift   <= data_in;

        elsif shift ='1'  then

                    data_out_shift(0)<='0';
                   data_out_shift(1)<= data_in(0);
                  data_out_shift(2)<= data_in(1);
                   data_out_shift(3)<= data_in(2);
                   data_out_shift(4)<= data_in(3);
                    data_out_shift(5)<= data_in(4);
                   data_out_shift(6)<= data_in(5);
                  data_out_shift(7)<= data_in(6);
            end if;
            end if;

end process;

end Behavioral;
```

The shift_register is a synchronous register with an entry corresponding to the byte to be placed in the register and two control signals. The first, load, allows the value of the first register to be copied into this one. Shift, on the other hand, is a signal that is activated and allows a shift operation to be carried out on the input byte of this register. This register therefore allows a one-bit shift to be performed on the byte sent by the PC.

3.4 Multiplexer 3-to-1



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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity mulx is
  Port (
        A   : in std_logic_vector(7 downto 0);
        B   : in std_logic_vector(7 downto 0);
        C   : in std_logic_vector(7 downto 0);
        controll : in std_logic_vector(1 downto 0);
        data_out_mul : out std_logic_vector(7 downto 0)
        );
end mulx;

architecture Behavioral of mulx is

begin
    data_out_mul <= A when controll="01" else
                    B when controll="10" else
                    C when controll="11" else
                    "00000000" when controll="00" ;
end Behavioral;
```

As explained above, it is up to the user to choose which register he wishes to observe on the LEDs or to select the value of the register which he wishes to send to the PC. To achieve this, we use a multiplexer which allows one of the 3 registers to be selected by means of a signal called control. This signal, consisting of two bits, is used to select the output of the multiplexer, which is connected to the data_led output and the input of the UART transmitter.

4. Constraint

We have used leds, switches and buttons for the constraints of our entity.

```
set_property -dict { PACKAGE_PIN D9    IOSTANDARD LVCMOS33 } [get_ports { loadl }]; #IO_L6N_T0_VREF_16 Sch=btn[0]
set_property -dict { PACKAGE_PIN C9    IOSTANDARD LVCMOS33 } [get_ports { com2  }]; #IO_L11P_T1_SRCC_16 Sch=btn[1]
set_property -dict { PACKAGE_PIN B9    IOSTANDARD LVCMOS33 } [get_ports { shiftIN }]; #IO_L11N_T1_SRCC_16 Sch=btn[2]
set_property -dict { PACKAGE_PIN B8    IOSTANDARD LVCMOS33 } [get_ports { send_data }]; #IO_L12P_T1_MRCC_16 Sch=btn[3]
```

The buttons are used to control the signals used to load the values of the registers, the signals used to perform the shift and 2-complement operations and finally the signal used to send the value of one of the registers to the PC.

```
#Switches

set_property -dict { PACKAGE_PIN A8    IOSTANDARD LVCMOS33 } [get_ports { control[0] }]; #IO_L12N_T1_MRCC_16 Sch=sw[0]
set_property -dict { PACKAGE_PIN C11   IOSTANDARD LVCMOS33 } [get_ports { control[1] }]; #IO_L13P_T2_MRCC_16 Sch=sw[1]
set_property -dict { PACKAGE_PIN C10   IOSTANDARD LVCMOS33 } [get_ports { reset }]; #IO_L13N_T2_MRCC_16 Sch=sw[2]
#set_property -dict { PACKAGE_PIN A10   IOSTANDARD LVCMOS33 } [get_ports { SW[3] }]; #IO_L14P_T2_SRCC_16 Sch=sw[3]
```

The first two switches correspond to the control signal of our entity, which allows the multiplexer to be controlled. The third switch is used to reset the main register.

```
## RGB LEDs
set_property -dict { PACKAGE_PIN E1    IOSTANDARD LVCMOS33 } [get_ports { data_led[0] }]; #IO_L18N_T2_35 Sch=led0_b
set_property -dict { PACKAGE_PIN G4    IOSTANDARD LVCMOS33 } [get_ports { data_led[1] }]; #IO_L20P_T3_35 Sch=led1_b
set_property -dict { PACKAGE_PIN H4    IOSTANDARD LVCMOS33 } [get_ports { data_led[2] }]; #IO_L21N_T3_DQS_35 Sch=led2_b
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { data_led[3] }]; #IO_L23P_T3_35 Sch=led3_b

## LEDs
set_property -dict { PACKAGE_PIN H5    IOSTANDARD LVCMOS33 } [get_ports { data_led[4]}]; #IO_L24N_T3_35 Sch=led[4]
set_property -dict { PACKAGE_PIN J5    IOSTANDARD LVCMOS33 } [get_ports { data_led[5] }]; #IO_25_35 Sch=led[5]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { data_led[6] }]; #IO_L24P_T3_A01_D17_14 Sch=led[6]
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { data_led[7]}]; #IO_L24N_T3_A00_D16_14 Sch=led[7]
```

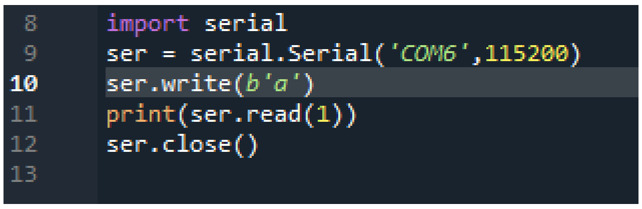
The LEDs, on the other hand, allow us to observe in real time the value of one of the three registers present in our entity

```
## USB-UART Interface
set_property -dict { PACKAGE_PIN D10   IOSTANDARD LVCMOS33 } [get_ports { data_out  }]; #IO_L19N_T3_VREF_16 Sch=uart_rxd_out
set_property -dict { PACKAGE_PIN A9    IOSTANDARD LVCMOS33 } [get_ports {  data_in }]; #IO_L14N_T2_SRCC_16 Sch=uart_txd_in
```

Finally, we have the FPGA's USB-UART interfaces, which are connected to the input and output of our unit and which allow us to receive and send data.Finally, we have the FPGA's USB-UART interfaces, which are connected to the input and output of our unit and which allow us to receive and send data.

5. Python script

In order to enable communication between the FPGA and the computer, we have used a USB cable, which not only serves to power the FPGA, but also to receive and send data. A Python script was used to implement this communication.

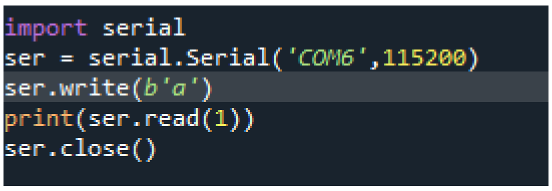


```python
import serial
ser = serial.Serial('COM6',115200)
ser.write(b'a')
print(ser.read(1))
ser.close()
```

First of all, we open a serial communication with port COM6, which corresponds to the USB output, defining at the same time the bauderate, which must be the same as that used in the FPGA. Then we send a character with the write() function to the FPGA and wait until we receive something back from the FPGA (with the read() function) and then close the communication.

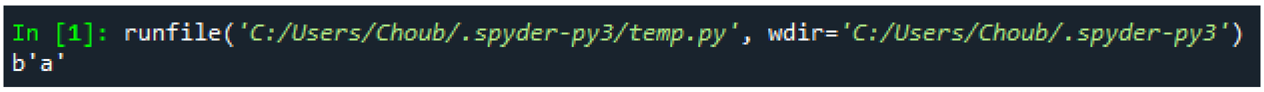
It is important to understand the format of the data we send with the write function. In the example above we send the string b'a' where b stands for binary. This means that the FPGA receives a byte that corresponds to the ASCII character of the letter 'a', i.e. hex 61 =>01100001.

6. Testing



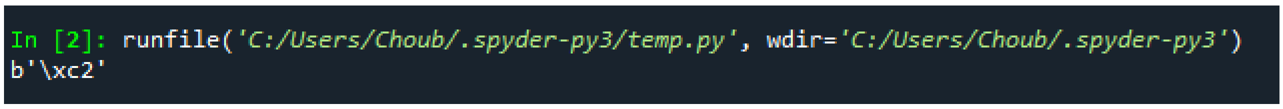
```python
import serial
ser = serial.Serial('COM6',115200)
ser.write(b'a')
print(ser.read(1))
ser.close()
```

We send to the FPGA the character 'a' which corresponds to the byte "01100001". Now let's look at the byte that is sent back to us by the FPGA according to the value of the register we select



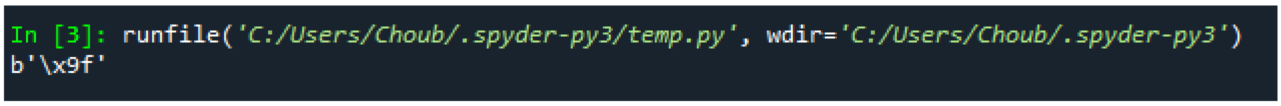
```
In [1]: runfile('C:/Users/Choub/.spyder-py3/temp.py', wdir='C:/Users/Choub/.spyder-py3')
b'a'
```

If we look at the first register, the byte sent to us by the FPGA is 'a' which corresponds to the character we sent it



```
In [2]: runfile('C:/Users/Choub/.spyder-py3/temp.py', wdir='C:/Users/Choub/.spyder-py3')
b'\xc2'
```

If we observe the shift register, the byte that is returned to us is C2 = "11000010" which corresponds to a shift of one bit towards the left of the byte "01100001".



```
In [3]: runfile('C:/Users/Choub/.spyder-py3/temp.py', wdir='C:/Users/Choub/.spyder-py3')
b'\x9f'
```

Finally, if we observe the 2's complement register, we obtain the value 9f = "10011111", the latter corresponds to not (01100001) +1.