Group 8 – Simon Game with FPGA

Filippo Festa, Marija Mojsovska, Filippo Orlando, Roya Julaei, and Shabnaz Ghaffari

This paper presents a work in the Management and Analysis of Physics Dataset consisting of the construction of a slightly modified version of the Simon Game in which the player must memorize a sequence that becomes more difficult as the levels progress. The work is implemented on FPGA: Digilent Arty-A7 100T board.

INTRODUCTION

Simon is a game designed to test your memory. In this game, there are four distinct light sources (LEDs) and four corresponding inputs from the user (buttons). The game displays a LEDs pattern, which the user has to memorize and repeat using the corresponding buttons. The game starts at level 1, where the velocity with which the LEDs are being turned on is the lowest. In each subsequent level the velocity increases. There are four levels. A level is passed if the user enters the right pattern as displayed by the game. If the user fails to do so, the game ends.

UART

For the developing of this project we use the UART (Universal asynchronous receiver-transmitter) hardware communication protocol.

UART Receiver

The receiver receives six random sequences of eight bits randomly generated using the dedicated function **rand_pattern** in the python script for each level.

The sequences are created to have only one bit equal to '1', which correspond to the LED on, and all the others to '0', the remaining LEDs off, for instance:

- $1^{st} LED$ on $\Rightarrow$ 1 integer or ~~0000~~ 0001 sequence
- $2^{nd} LED$ on $\Rightarrow$ 2 integer or ~~0000~~ 0010 sequence
- $3^{rd} LED$ on $\Rightarrow$ 4 integer or ~~0000~~ 0100 sequence
- $4^{th} LED$ on $\Rightarrow$ 8 integer or ~~0000~~ 1000 sequence

The **rand_pattern** function (Fig. 1) creates a list of six random integer elements extracted with no neighbouring repetition from the $\{1, 2, 4, 8\}$ set, each matching the respective sequence, which altogether will form the pattern of the n^{th} level.

So then six sequences are generated by the script, each of them collected in the **receive_data_s** eight bits standard logic vector and at the end sequentially stored in

```python
import numpy.random as npr

def rand_pattern():
    set = [1,2,4,8]
    l = [npr.choice(set)]

    for i in range(5):
        set = [1,2,4,8]
        set.remove(l[i])
        l.append(npr.choice(set))
    return l
```

FIG. 1. Python function that generates the random sequences

a two dimensional 6x4 array by the receiver, the **input_sequence** matrix. For this purpose we define a new type using the following syntax:

type *type_name* **is array** *(range)* **of** *element_type;*

where the *(range)* defines the number of rows (length of the sequence) and *element_type* is the number of columns (LEDs). The command **use work.newtype.all**, in the following code (Fig. 2), allows to import the package containing the type **matrix**.

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


package newtype is
    type matrix is array(0 to 5) of std_logic_vector(3 downto 0);
end newtype;
```

FIG. 2. Type matrix

Thereby, the output of the receiver (Fig. 3), named **input_sequence**, is the matrix containing the sequences which is then send as input to the game.

Notice that the first four bits of each sequence act as padding, in fact they are always zero, and the only bits that matters for our purpose are the last four, each associated to its LED. Therefore we modify the final case of the UART state machine, that is *bit7_s*, in order to store in the rows of the **input_sequence** matrix only the bits of interest. The other main modification on this very spot is the deployment and the usage of the variable **i** (Fig. 4), which has been exploited to sequentially save in **input_sequence** all the six sequence, one in each row of

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


entity uart_receiver is
  port (clock          : in  std_logic;
        uart_rx        : in  std_logic;
        valid          : out std_logic;
        input_sequence: out matrix);
end entity uart_receiver;
```

FIG. 3. Changes in UART receiver code

the matrix.

```
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
    input_sequence(i) <= received_data_s(3 downto 0);
    i := i + 1;
    if i = 6 then
        valid <= '1';
        i := 0;
    end if;

    state <= idle_s;
   end if;
```

FIG. 4. Changes in UART receiver state machine code

In particular we have to underline the choice of variable instead of a signal, recall that:

- Variables can only be used inside processes, while signals can be used inside or outside them;
- Variables need to be defined after the keyword process but before the keyword begin;
- Any variable that is created in one process cannot be used in another process. Signals can be used in multiple processes, though they can only be assigned in a single process;
- Variables that are assigned in a spot immediately take the value of the assignment.

After this bits gathering procedure is done, hence when $i = 6$, the receiver modifies the valid standard logic signal from zero to one for a whole clock cycle, allowing the game to start, and at this point it waits for a new pattern.

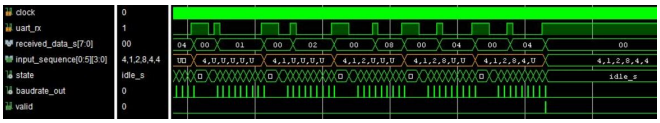

FIG. 5. UART receiver's testbench

UART Transmitter

The UART transmitter component design is the standard one seen during the lessons, no modifications has been done. It basically receives the 8 bits standard logic vector **filtered_data** from the game component and sends it back to the computer bit by bit. The **filtered_data** vector, as explained more in detail in the game section, can assume only two values because of the game's design:

- 00000000 binary ⇒ 0 integer
  It stops the game, so then the python script (Fig. 6) waits until the player wants to restart a new try by means of the **rst** button
- 11111111 binary ⇒ 255 integer
  The game requests for a new pattern, so then the python script (Fig. 6) generates it and sends it back to the receiver

```
import serial
import time

ser = serial.Serial("COM4", baudrate = 115200)
l = rand_pattern()

for i in l:
    value = chr(i)
    print()
    print(i)
    print(bin(i))

    ser.write(value.encode())
d = ser.read()
print("\n=>", ord(d))

while True:
    if ord(d) == 255:
        print("\n\n\n")
        l = rand_pattern()
        time.sleep(2)
        for i in l:
            value = chr(i)
            print()
            print(i)
            print(bin(i))
            ser.write(value.encode())
        d = ser.read()
        print("\n=>", ord(d))

    if ord(d) == 0:
        d = ser.read()
        print("\n\n\n=>", ord(d))
```

FIG. 6. Python script for the complete game

GAME

In this section, we analyse the entire game code.

entity *name* **is**
port(port_name: mode data_type);
generic(generic list);
end *name*;

Starting with the entity box, shown in Fig. 7, we define four input ports:

- **pattern_input** is the *matrix* of led sequences described in the previous section;
- **valid_in** is a *std_logic* that allows to start the game. It is '1' when the UART receiver stores all six led sequences;
- **rst** is a *std_logic* which restarts the game, it is the A8 switch on the FPGA;
- **clock** is the 100 MHz clock implemented in the FPGA;
- **button_in** is a *std_logic_vector(3 downto 0)*. The four components button_in[i] are mapped with the four physical buttons in the FPGA (D9, C9, B9, B8).

and six output ports:

- **led** is a *std_logic_vector(3 downto 0)*. These LEDs show the pattern to be memorized. Each component is mapped with the FPGA blue LEDs (E1, G4, H4, K2);
- **led_red** is a *std_logic_vector(3 downto 0)*. These LEDs light up simultaneously when the player fails. Each component is mapped with the FPGA red LEDs (G6, G3, J3, K1);
- **led_green** is a *std_logic_vector(3 downto 0)*. These LEDs light up simultaneously when the player inserts the correct sequence. Each component is mapped with the FPGA green LEDs (F6, J4, J2, H6);
- **led_level** is a *std_logic_vector(3 downto 0)*. These LEDs show the level. Each component is mapped with the FPGA LEDs (H5, J5, T9, T10). For instance, led_level[0] turns on when the first level is passed; led_level[0] and led_level[1] correspond to the second level and so on;
- **valid_out** is a *std_logic* that is send as an input to the transmitter when we have a result from the game (success or fail of the player) to transmit;
- **to_trasmit** is a *std_logic_vector(7 downto 0)* that returns to the UART transmitter '11111111' (decimal 255), if the sequence inserted by the player is correct, or '00000000' (decimal 0), if the inserted sequence is wrong.

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.NUMERIC_STD.ALL;
use work.newtype.all;


entity game is
    port(pattern_input : in matrix;
         valid_in      : in std_logic;          -- from receiver

         rst           : in std_logic;
         clock         : in std_logic;
         button_in     : in std_logic_vector(3 downto 0);

         led           : out std_logic_vector(3 downto 0) := (others => '0');
         led_red       : out std_logic_vector(3 downto 0) := (others => '0');
         led_green     : out std_logic_vector(3 downto 0) := (others => '0');
         led_level     : out std_logic_vector(3 downto 0) := (others => '0');

         valid_out   : out std_logic := '0';              -- to transmitter
         to_transmit    : out std_logic_vector(7 downto 0));
end entity game;
```

FIG. 7. The ports in the entity *game*

The following image Fig. 8 shows all the signals used in the game, the edge detector component and the cases of the state machine.

```
component edge_detector is
        Port (sig   : in std_logic;
              clk   : in std_logic;
              edge  : out std_logic);
end component;

type state_t is (waiting, S0, Detect);
signal state : state_t := waiting;

signal slow_clk       : std_logic;
signal counter        : unsigned(27 downto 0) := (others => '0');
signal enable_counter : std_logic := '0';

signal level          : integer := 0;

signal valid_pattern  : std_logic := '0';


signal slow_clk_detect : std_logic;      -- edge detection
signal rst_detect      : std_logic;

signal button_in0_detect   : std_logic;  -- button detection variables
signal button_in1_detect   : std_logic;
signal button_in2_detect   : std_logic;
signal button_in3_detect   : std_logic;
signal edge_detect_buttons : std_logic;
```

FIG. 8. Signals and state machine declaration

signal sig_name: data_type [:=initial_value];

Initially, we define the cases of the state machine:

- **waiting**: this is the state in which the machine waits until the whole sequence is shown to the player;
- **S0**: is the case in which the input from the buttons is compared with the sequence to be guessed;

- **Detect**: is the last step in which the led corresponding to the current level is turned on (in the case of a correctly guessed sequence) and the game goes back to the initial state waiting for a new sequence.

The *state_t* type is initialized in the **waiting** state.

Then we define the *slow_clk* as *std_logic*. This *slow_clk* takes the value of the last bit of the counter. The latter is obviously defined as an *unsigned* of twenty-seven bits initialized to '0'. In fact, if the counter was of type signed we would not be able to count correctly from zero to the maximum number.

The *enable_counter* signal allows to start and stop the counter when certain conditions are satisfied.

Now, *level* signal is defined as *integer*. Integer data type can be used to define objects whose value can be a whole number. In our case, the level signal assumes the 0,1,2,3,4 values.

valid_pattern is a *std_logic* that turns to '1' when the entire led sequence is shown.

For the detection of the rising edges of the slow clock, the button's input, and the restart switch we use the following code for edge detection (Fig. 9):

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity edge_detector is
        Port (sig   : in std_logic;
              clk   : in std_logic;
              edge  : out std_logic);
end edge_detector;

architecture rtl of edge_detector is

signal ff1 :std_logic;
signal ff2 :std_logic;

begin

p_edge : process (clk) is
        begin
                if rising_edge(clk) then
                        ff1 <= sig;
                        ff2 <= ff1;
                        edge <= ff1 and not(ff2);
                end if;
        end process;

end rtl;
```

FIG. 9. Edge detector code

All the processes in the *game* are synchronized with the rising edge of the *clock*.

p_clk process

In the *p_clk* process (Fig. 10) we use a counter, standard logic vector of twenty seven elements, in order to create a slow clock (*slow_clk*) that is used for defining the time intervals in which the LEDs turn on to show the sequence. The counter starts when it receives *valid_in = '1'* from the receiver, meaning that a pattern is read and it's ready to be displayed for the player. It continues counting until it gets *valid_pattern = '1'*, which happens right after the pattern is shown, then it stops.

The slow clock is assigned the value of *counter(27-level)* which depends on the current level, this way the *slow_clk* becomes faster as the levels progress. For example for the first level the period of the *slow_clk* is *1,34s*, for the second level is *0,67s*.

```
begin

    p_clk: process (clock) is
        begin
        if rising_edge(clock) then

            if valid_in = '1' then
                enable_counter <= '1';
            end if;

            if enable_counter = '1' then
                counter <= counter + 1;
            end if;

            if valid_pattern = '1' then
                enable_counter <= '0';
                counter <= (others => '0');
            end if;

        end if;
    end process;

    slow_clk <= counter(26-level);
```

FIG. 10. *p_clk* process for creating the slow clock

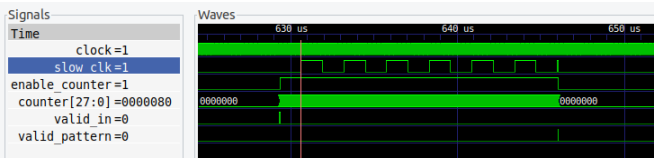

FIG. 11. Testbench *p_clk* process

p_led process

The *p_led* process(Fig. 12) is used for assigning to each LED the values corresponding to the sequences of the pattern that needs to be displayed. This procedure is synchronized with the *slow_clk*, for which we use an edge detector. Hence at each rising edge of the *slow_clk* the values of the four LEDs change. This is repeated for six cycles, given the fact that we have a pattern of 6x4, at the end of which *valid_pattern* becomes *'1'*.

```
p_led : process (clock) is
variable i : integer := 0;

begin
    if rising_edge(clock) then
        valid_pattern <= '0';

        if slow_clk_detect = '1' then
            if i < 6 then
                led <= pattern_input(i);
                i := i+1;
            elsif i = 6 then
                led <= (others => '0');
                valid_pattern <= '1';
                i := 0;
            end if;
        end if;
    end if;
end process;
```

FIG. 12. *p_led* process - displays the pattern

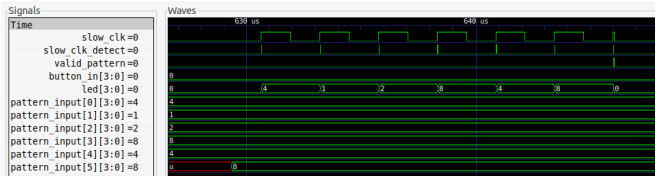

FIG. 13. Testbench *p_led* process

main process

This is the main process in which the input form the buttons is read and confronted with the shown pattern. It consists of a state machine with three states: *waiting*, *S0* and *detect*.

```
case state is

   when waiting =>
        if valid_pattern = '1' then
            i := 0;
            state <= S0;
        end if;

        if valid_in = '1' then
            led_green <= (others => '0');
            led_red   <= (others => '0');
        end if;
```

FIG. 14. State *waiting*

At the beginning the state machine is in *waiting* (Fig. 14) and it remains there until the pattern is shown to the player, that is until *valid_pattern* becomes *'1'*, then it goes to *S0* (Fig. 15). In this state we are using an edge detector for the buttons, every time the player presses a button this input is confronted with the given pattern. If the sequence inserted by the player is correct the green LEDs turn on, *'11111111'* is sent to the transmitter (*valid_out = '1'*), and the level is increased, unless we are in the last level, in that case *'00000000'* is sent to the transmitter because the game is won and there is no need for a new pattern to be created. On the contrary if the player presses a wrong button the red LEDs turn on, *'00000000'* (*valid_out = '1'*) is sent to the transmitter and the game is over.

```
when S0 =>
   if edge_detect_buttons = '1' then                  -- button push detection
              if button_in = pattern_input(i) then
                 if i = 5 then
                      if level = 3 then
                          to_transmit <= "00000000"; -- end of game
                      else
                          to_transmit <= "11111111"; -- new sequence
                      end if;
                      valid_out <= '1';
                      led_green <= (others => '1');
                      level <= level + 1;
                      state     <= Detect;
                 else
                      i := i+1;
                 end if;
              else
                  to_transmit <= "00000000";          -- end of game
                  valid_out   <= '1';
                  led_red <= (others => '1');
                  state       <= Detect;
              end if;
      end if;
```

FIG. 15. State *S0*

From here we enter in the state *detect* (Fig. 16) where depending in which level the game is, the corresponding *level* LED turns on, *valid_out* goes back to *'0'* and the state machine returns to the *waiting* state where the red/green LEDs are turned off when a new sequence is read by the receiver, that is when *valid_in* becomes *'1'*.

```
when Detect =>

    if level = 1 then
        led_level(0) <= '1';
    elsif level = 2 then
        led_level(1) <= '1';
    elsif level = 3 then
        led_level(2) <= '1';
    elsif level = 4 then
        led_level(3) <= '1';
    end if;

    valid_out <= '0';
    state <= waiting;

    when others => null;

end case;
```

FIG. 16. State *Detect*

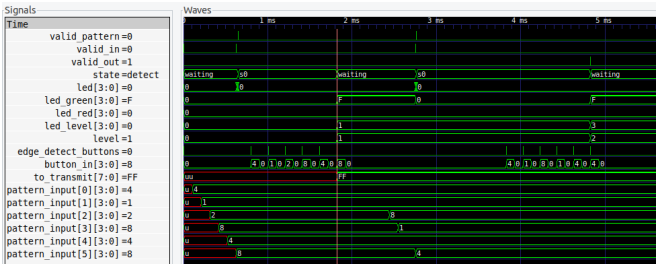

FIG. 17. Testbench *main* process

In this process we also have a *restart* switch used to start a new game (Fig. 18). When this is detected by the edge detector, all the LEDs (green/red, level) turn off, the *level* goes back to *0* and *'11111111'* is send to the transmitter (*valid_out = '1'*) in order to get a new pattern. From there the state machine goes in *detect*.

```
if rising_edge(clock) then

    if rst_detect = '1' then  -- restart game => new try
        state <= Detect;
        level <= 0;
        i := 0;

        to_transmit <= "11111111";    -- get new sequence
        valid_out   <= '1';
        led_green <= (others => '0');
        led_red   <= (others => '0');
        led_level <= (others => '0');

    else
        case state is
```

FIG. 18. *rst* switch - start new game

CONSTRAIN

```
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { CLK100MHZ }]; #IO_L12P_T1_MRCC_35 Sch=gclk[100]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports { CLK100MHZ }];

set_property -dict { PACKAGE_PIN D10   IOSTANDARD LVCMOS33 } [get_ports { uart_rxd_out }]; #IO_L19N_T3_VREF_16 Sch=uart_rxd_out
set_property -dict { PACKAGE_PIN A9    IOSTANDARD LVCMOS33 } [get_ports { uart_txd_in }]; #IO_L14N_T2_SRCC_16 Sch=uart_txd_in

set_property -dict { PACKAGE_PIN E1    IOSTANDARD LVCMOS33 } [get_ports { led[0] }]; #IO_L18N_T2_35 Sch=led0_b
set_property -dict { PACKAGE_PIN F6    IOSTANDARD LVCMOS33 } [get_ports { led_green[0] }]; #IO_L19N_T3_VREF_35 Sch=led0_g
set_property -dict { PACKAGE_PIN G6    IOSTANDARD LVCMOS33 } [get_ports { led_red[0] }]; #IO_L19P_T3_35 Sch=led0_r
set_property -dict { PACKAGE_PIN G4    IOSTANDARD LVCMOS33 } [get_ports { led[1] }]; #IO_L20P_T3_35 Sch=led1_b
set_property -dict { PACKAGE_PIN J4    IOSTANDARD LVCMOS33 } [get_ports { led_green[1] }]; #IO_L21P_T3_DQS_35 Sch=led1_g
set_property -dict { PACKAGE_PIN G3    IOSTANDARD LVCMOS33 } [get_ports { led_red[1] }]; #IO_L20N_T3_35 Sch=led1_r
set_property -dict { PACKAGE_PIN H4    IOSTANDARD LVCMOS33 } [get_ports { led[2] }]; #IO_L21N_T3_DQS_35 Sch=led2_b
set_property -dict { PACKAGE_PIN J2    IOSTANDARD LVCMOS33 } [get_ports { led_green[2] }]; #IO_L22N_T3_35 Sch=led2_g
set_property -dict { PACKAGE_PIN J3    IOSTANDARD LVCMOS33 } [get_ports { led_red[2] }]; #IO_L22P_T3_35 Sch=led2_r
set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 } [get_ports { led[3] }]; #IO_L23P_T3_35 Sch=led3_b
set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 } [get_ports { led_green[3] }]; #IO_L24P_T3_35 Sch=led3_g
set_property -dict { PACKAGE_PIN K1    IOSTANDARD LVCMOS33 } [get_ports { led_red[3] }]; #IO_L23N_T3_35 Sch=led3_r

set_property -dict { PACKAGE_PIN H5    IOSTANDARD LVCMOS33 } [get_ports { led_level[0] }]; #IO_L24N_T3_35 Sch=led[4]
set_property -dict { PACKAGE_PIN J5    IOSTANDARD LVCMOS33 } [get_ports { led_level[1] }]; #IO_25_35 Sch=led[5]
set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 } [get_ports { led_level[2] }]; #IO_L24P_T3_A01_D17_14 Sch=led[6]
set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { led_level[3] }]; #IO_L24N_T3_A00_D16_14 Sch=led[7]
```

FIG. 19. First part of the constrain file

```
set_property -dict { PACKAGE_PIN D9    IOSTANDARD LVCMOS33 } [get_ports { button_in[0] }]; #IO_L4N_T0_VREF_16 Sch=btn[0]
set_property -dict { PACKAGE_PIN C9    IOSTANDARD LVCMOS33 } [get_ports { button_in[1] }]; #IO_L11P_T1_SRCC_16 Sch=btn[1]
set_property -dict { PACKAGE_PIN B9    IOSTANDARD LVCMOS33 } [get_ports { button_in[2] }]; #IO_L11N_T1_SRCC_16 Sch=btn[2]
set_property -dict { PACKAGE_PIN B8    IOSTANDARD LVCMOS33 } [get_ports { button_in[3] }]; #IO_L12P_T1_MRCC_16 Sch=btn[3]


set_property -dict { PACKAGE_PIN A8    IOSTANDARD LVCMOS33 } [get_ports { rst }]; #IO_L12N_T1_MRCC_16 Sch=sw[0]
```

FIG. 20. Second part of the constrain file

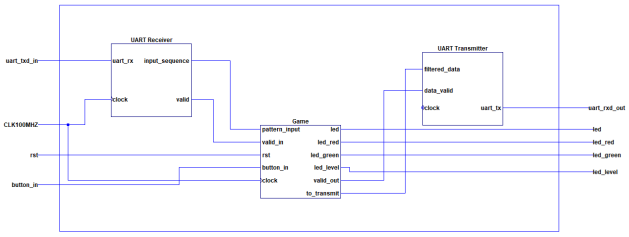

FIG. 21. Diagram of the game

POSSIBLE IMPROVEMENT OF THE CODE

Button Debouncers

A possible improvement on the shown code can be done introducing button debouncers. The problem with mechanical buttons and switches is that they might bounce one or more times before finally settling in the proper position, such bounces might last up to several milliseconds.

FIG. 22. Expected output from the button VS reality

In order to be sure that this won't happen a debounce circuit can be included. The circuit is shown in Fig. 23. At each rising edge of the clock the button's value is stored in FF1 and subsequently in FF2. When this values remain unchanged for a certain time FF3 is enabled and the stable value is saved as a result.

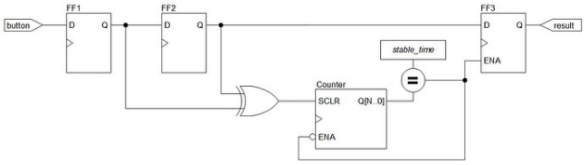

FIG. 23. Debounce circuit

CONCLUSIONS

The code described previously was successfully implemented on FPGA: Digilent Arty-A7 100T board and the expected behavior was observed.

Note: In order to make the testbench *slow_clk* was set to *counter(7-level)*.

[1] Volnei A. Pedroni (2020), *Circuit Design with VHDL*, The MIT Press

[2] Mark Zwolinski (2004), *Digital system design with VHDL*, Pearson Education

[3] Bryan Mealy, Fabrizio Tappero (2018), *Free range VHDL*, http://www.freerangefactory.org