ENGINE

Teaching online electronics, microcontrollers and programming in Higher Education

Hardware Implementation of Algorithms

6. Pseudorandom number generation using LFSR.

Lead Partner: Warsaw University of Technology

Autor: Lukasz Mik

University of Applied Sciences in Tarnow

Erasmus+

GRANT NUMBER: 2020-1-PL01-KA226-HE-095653

Declaration

This laboratory instruction has been prepared in the context of the ENGINE project. Where other published and unpublished source materials have been used, these have been acknowledged.

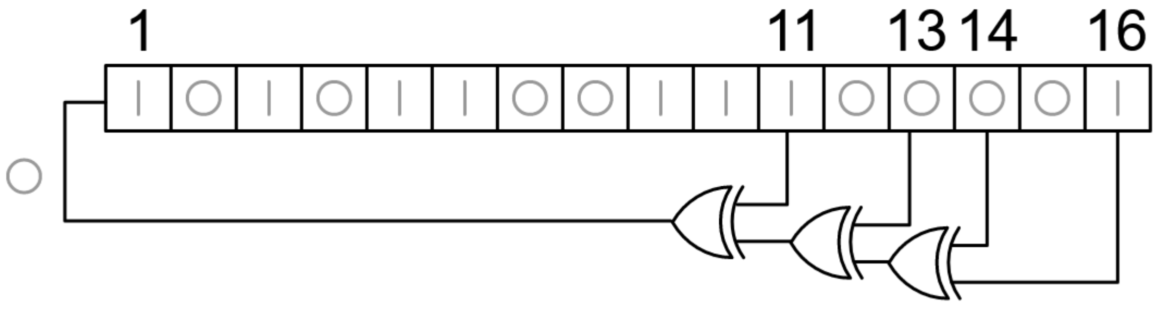
Copyright

Funding Disclaimer

This project has been funded with support from the European Commission. This report reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

I. LFSR – theoretical background.

The LFSR is a linear feedback shift register whose input bit is a linear function of the previous state. To implement such feedback, an XOR gate is most often used, whose inputs are connected to selected register outputs, and the gate output to its input. The initial value of the LFSR is called the seed, and since the operation of the register is deterministic, the stream of values generated by the register is completely determined by its current (or previous) state. Due to the fact that the LFSR register has a finite number of states, after passing through all of them, it returns to its initial state and the cycle repeats again. However, an LFSR with a well-chosen feedback function can produce a sequence of bits that appears random and that has a very long cycle without repetition. Typical applications of LFSR counters include generators of: pseudo-random numbers, white noise, bit pseudo-random sequences. The hardware and software implementations of the LFSR are the same. An example of a 16-bit LFSR is shown in the figure below.



The bits that affect the next state are called taps. In this case, the taps are derived from bits 16, 14, 13, and 11. The rightmost LFSR bit is the output bit. The bits from the shift register taps are XORed with the output bit, and then fed back to the leftmost bit. For a register of length **n** bits, the number of all possible sequences of bits, and thus pseudo-random numbers, is **2^n-1**. The feedback tap system in the LFSR can be expressed in finite field arithmetic as a mod 2 polynomial. This means that the coefficients of the polynomial must be either 1 or 0. This is called the feedback polynomial or inverse characteristic polynomial. For example, if the taps are on 16, 14, 13, and 11 bits (as shown in the figure), the feedback polynomial is:

$$x^{16} + x^{14} + x^{13} + x^{11} + 1$$

The "one" in the polynomial does not correspond to The powers of the terms represent the selected bits, counting from the left. The first and last bits are always connected as input and output taps respectively. The LFSR can reach its maximum length if and only if the corresponding feedback polynomial is a primary polynomial the tap - it corresponds to the input to the first bit (i.e. x^0, which corresponds to 1). The powers of the **x** represent the selected bits, counting from the left. The first and last bits are always connected as input and output taps respectively. The LFSR can reach its maximum length if and only if the corresponding feedback polynomial is a primary polynomial. An example code for implementing a 16-bit LFSR in C is shown in the listing below.

```c
#include <stdint.h>
unsigned lfsr_fib(void)
{
  uint16_t start_state = 0xACE1u;  //Any nonzero start state will work
  uint16_t bit;    //Must be 16-bit to allow bit<<15 later in the code
  unsigned period = 0;

  do
  {   // taps: 16 14 13 11;
      //feedback polynomial: x^16 + x^14 + x^13 + x^11 + 1
     bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5)) & 1u;
     lfsr = (lfsr >> 1) | (bit << 15);
     ++period;
  }
  while (lfsr != start_state);

  return period;
}
```

LFSR counters have simpler feedback logic than natural binary or gray code counters and can therefore operate at higher clock rates. However, you must ensure that the LFSR never goes to zero, for example by setting it at startup to any other state in the sequence. The primitive polynomial table (Xilinx XAPP 052) shows how the LFSR can be arranged in Fibonacci or Galois form to obtain maximum sequence generation periods without repetition. Any other period can be obtained by adding to the LFSR, which has a longer period, some logic that shortens the sequence by omitting some states.

LFSR output streams are deterministic. If the current state and positions of the XOR gates in the LFSR are known, the next state can be predicted. This is not possible for truly random events. For a maximum-length LFSR, it is much easier to compute the next state because there are only a limited number of them for each length.

II. Pseudorandom number generation using LFSR.

As written earlier, a register of length n can generate a pseudo-random string with a maximum length of 2n-1. The binary codes at the output of the counter constructed in this way, after D/A conversion, generate noise with a uniform distribution.

During the course, we will first implement a pseudorandom number generator, built of a 4-bit shift register and a very low-order polynomial. Such a generator will be able to generate 15 different sequences of bits (numbers). When the generator register reaches state 15, the sequence repeats. Note that the LFSR is a one-bit random generator - the binary string is taken from its output. If we want to check the states on each of the taps, then each tap must be derived in the form of a vector of bits. In VHDL, this can be done by setting the port to **inout** mode. For a 4-bit register, the primary polynomial has the following form: $\mathbf{x^3 + x^2 + 1}$

A practical implementation of a pseudo-random number generator in VHDL

STEP 1: Create a project called **lfsr4bit** in ISE Webpack. Remember about the appropriate parameters of the target FPGA

STEP 2: Inside the project, create a new VHDL source file called **lfsr4bit**. In the header part of this file, import the necessary libraries and define the I/O ports of the project unit.

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

entity lfsr4bit is
  port (
        rst : in std_logic;
        clk : in  std_logic;
        rand : out std_logic_vector(3 downto 0)  -- LFSR
  );
end entity;
```

STEP 3: Then, in the description of the architecture, add the process in which the linear feedback shift register will be implemented.

```
architecture Behavioral of lfsr4bit is

  signal lfsr       : std_logic_vector (3 downto 0); -- LFSR register
  signal feedback      : std_logic;                   -- LFSR feedback

begin

  feedback <= not(lfsr(3) xor lfsr(2));-- feedback by polynomial x^3+x^2+1

  lfsr_pr : process (clk)
   begin
    if (rising_edge(clk)) then
      if (rst = '0') then
        lfsr <= (others=>'0');
      else
        lfsr <= lfsr(2 downto 0) & feedback;
      end if;
    end if;
  end process lfsr_pr;
  rand <= lfsr;                          -- parallel output of LFSR

end architecture;
```

STEP 4: Now you need to bind the I/O ports from the unit declaration to the FPGA pins. For this purpose, create a UCF file (*Implementation Constraints File* option) and add the code binding the ports with the pins of the system in it.

```
#clock
NET "clk"   LOC = P129 | IOSTANDARD = LVCMOS33 | PERIOD = 12MHz;
#reset
NET "rst"   LOC = P80  | PULLUP   | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 12;
#LFSR parallel output on 4 LEDs
NET "rand[0]"  LOC = P46 | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 12;
NET "rand[1]"  LOC = P47 | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 12;
NET "rand[2]"  LOC = P48 | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 12;
NET "rand[3]"  LOC = P49 | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 12;
```

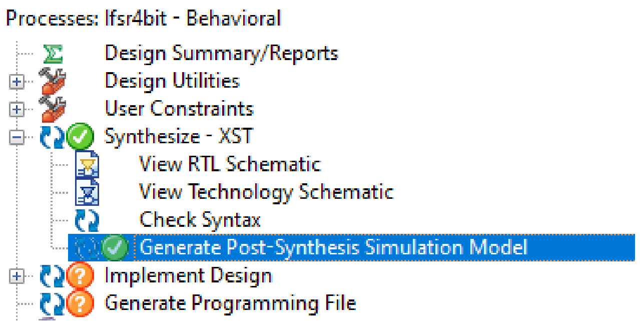
After completing the source files, you can perform a synthesis of the project to check for any syntactic or formal errors in the VHDL language. The tool will also check whether the description prepared by us is synthesizable to digital form.

STEP 5: In the next step, we will simulate the project. So we create a new file called **lfsr4bit_tb** selecting the VHDL Test Bench option when creating. In the editing window of the test program file, paste the following code.

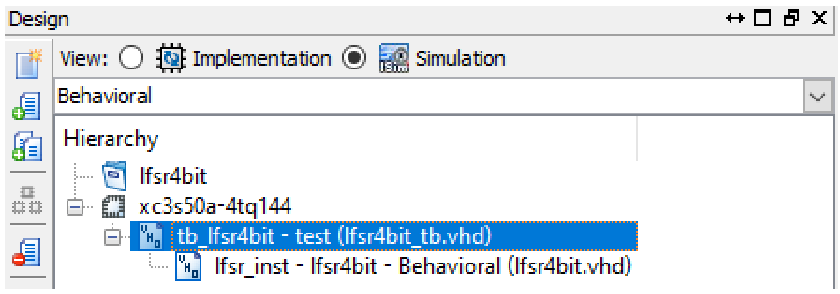
```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.ALL;

entity tb_lfsr_v1 is
end entity;
```

```
architecture test of tb_lfsr_v1 is

  constant PERIOD   : time     := 83 ns;--frequency ~12MHz
  signal clk        : std_logic := '0';
  signal rst        : std_logic := '0';
  signal rand       : std_logic_vector(3 downto 0);
  signal endSim     : boolean   := false;

  component lfsr_v1 is
    port (
      rst       : in  std_logic;
      clk       : in  std_logic;
      rand      : out std_logic_vector(3 downto 0)
    );
  end component;

begin
  clk      <= not clk after PERIOD/2;
  rst     <= '1' after  PERIOD*2;
  endSim  <= true after PERIOD*60;

 -- End the simulation
 process
 begin
   if (endSim) then
     assert false
       report "End of simulation."
       severity failure;
   end if;
   wait until (clk = '1');
 end process;

 lfsr_inst : lfsr_v1
 port map (
   clk       => clk,
   rst       => rst,
   rand      => rand
 );
end architecture;
```

STEP 6: Generate a behavioral simulation model (Post-Synthesis).


Processes: lfsr4bit - Behavioral

STEP 7: In the project window, change the view to the simulation view, and select the *Behavioral* option from the drop-down menu.



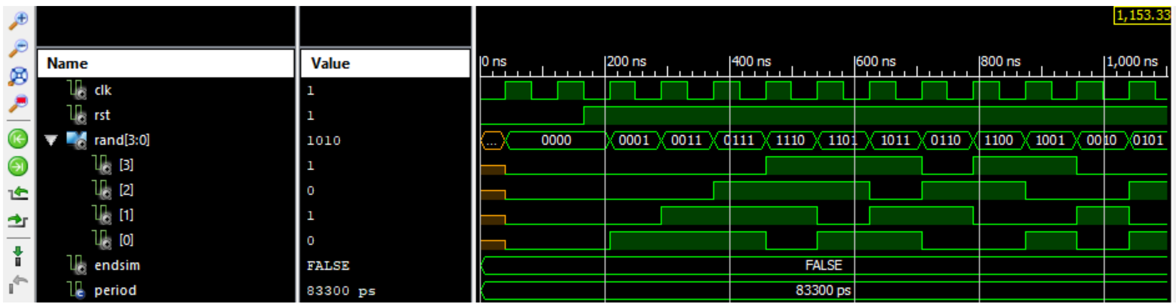
Design

View: ○ Implementation ● Simulation

Behavioral

Hierarchy
- lfsr4bit
- xc3s50a-4tq144
  - tb_lfsr4bit - test (lfsr4bit_tb.vhd)
    - lfsr_inst - lfsr4bit - Behavioral (lfsr4bit.vhd)

STEP 8: In the processes window, double-click on the *Simulate Behavioral Mode* option, after which the **iSim** simulator window will open. In the horizontal toolbar, in the place where we specify the end time of the simulation, enter 5us. Adjust the contents of the simulator window with the Zoom options to display the fragment of time waveforms we are interested in.



Name	Value
clk	1
rst	1
rand[3:0]	1010
[3]	1
[2]	0
[1]	1
[0]	0
endsim	FALSE
period	83300 ps

STEP 9: In order to be able to track changes of states on the output of the LFSR using LEDs, it is necessary to reduce the clock frequency of the register. For this purpose, we introduce an additional signal inside the architecture called **clk_1Hz**, which will be responsible for generating a clock signal with a frequency of 1Hz. After introducing corrections in the code, the description of the architecture should take the form as in the code fragment presented below.

```vhdl
architecture Behavioral of lfsr4bit is
  signal lfsr          : std_logic_vector (3 downto 0); -- LFSR register
  signal feedback  : std_logic;                      -- LFSR feedback
  signal clk_1Hz     : std_logic;

begin

  feedback <= not(lfsr(3) xor lfsr(2));-- feedback by polynomial x^3+x^2+1
  process(clk)
        variable counter : integer:=0;
  begin
        if (rising_edge(clk)) then
      if (counter>6000000) then
        counter:=0;
       clk_1Hz <= not clk_1Hz;
      else
        counter:=counter+1;
      end if;
    end if;
  end process;

  lfsr_pr : process (clk_1Hz)
       begin
    if (rising_edge(clk_1Hz)) then
      if (rst = '0') then
        lfsr <= (others=>'0');
      else
        lfsr <= lfsr(2 downto 0) & feedback;
      end if;
    end if;
  end process lfsr_pr;
  rand <= lfsr;     -- parallel output of LFSR

end architecture;
```

STEP 10: After compiling the project and generating the configuration file, the target system should be programmed and the correct operation of the generator should be checked.

ZADANIE: Using the example discussed in the lesson, design a pseudo-random number generator with an 8-bit LFSR register. Feedback is to be implemented using a polynomial: $\mathbf{x^8 + x^6 + x^5 + x^4 +1}$

References

- *User manual for Elbert V2 - Spartan 3A FPGA Development Board.* https://numato.com/docs/elbert-v2-spartan-3a-fpga-development-board/

- Pseudo random generation Tutorial - https://fpgaer.tech/?p=51

- Xilinx iSim User Guide - UG660 (v14.1) https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx14_1/plugin_ism.pdf