Embedded Computing Systems

Linear Feedback Shift Register

Design of Digital System

2018 / 2019

Author:
Edoardo Cittadini

Contents

1 Introduction

Nowadays the ongoing debate on the future of the semiconductor industry has turned into a discussion about the growing choice of technologies that, instead of being obsessed by the further process of geometry reduction, focus on system architectures and better use of the silicon available through new circuit, device and packaging design concepts.

FPGAs represent a solution that aims to be much more than a simple innovation, but thanks to their versatility they allow more efficient, faster and more economical development processes that have quickly led them to be widely used.

In some cases it is possible to observe that some components have been marketed as hardware solutions directly implemented on the FPGA whose performance, power consumption have been evaluated such as not to consider convenient the transition to the corresponding ASIC (Application Specific Integrated Circuit).

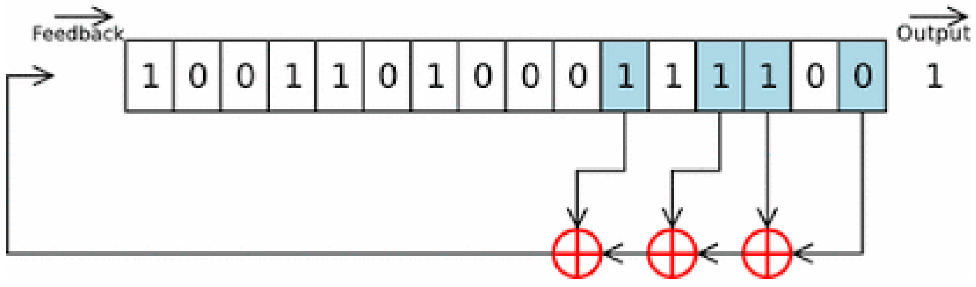
The FPGAs due to the almost total control that they leave to the user more and more frequently are combined with microcontrollers in embedded systems designed boards where the programmer is free to use it to implement additional hardware logic, specific purpose circuits and custom peripherals.

The proof of the great success of this technology is also given by the fact that Intel, one of the largest processor manufacturers in the world, has decided to buy Altera to develop a hybrid processor.

2 LFSR

A Linear Feedback Shift Register is an electronic component, in particular is a particular kind of shift register whose input bit is a linear function of the previous internal state.

Typically the linear functions used in the feedback chain are XOR (exclusive OR) and more rarely XNOR (negated exclusive OR).
Feedback is organized in such a way that the input bit is the result of an XOR operation between some bits of the previous state of the shift register in case of Fibonacci architecture or simply the last bit in case of Galois one.
Architectures are explained in more detailed way in the next paragraph.

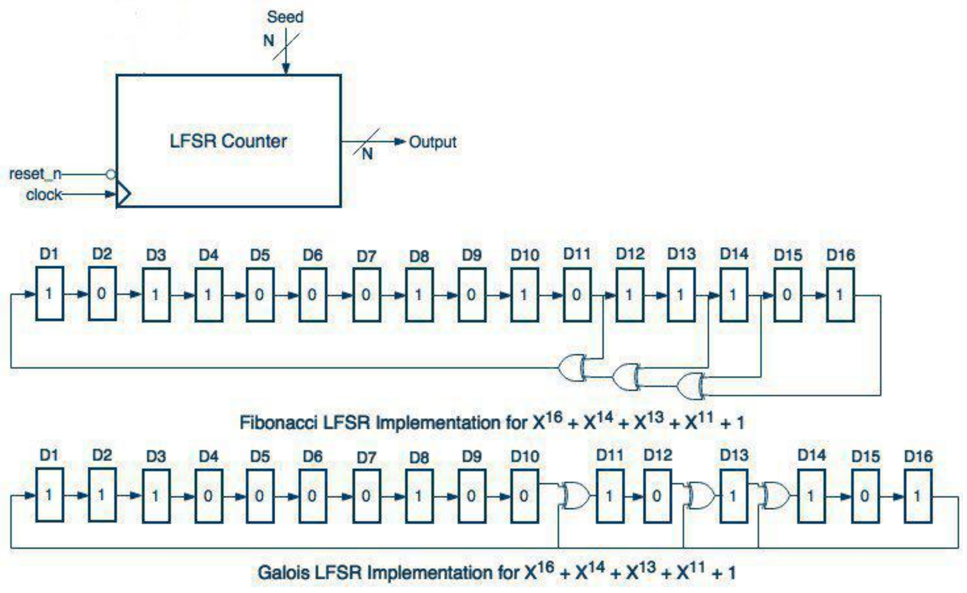


The initial value assigned to the register is called seed , and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state. Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle.

However, an LFSR with a well-chosen feedback function can produce a sequence of bits that appears random and has a very long cycle.

2.1 Architectures

There are two main different architectures of this component which are respectively the Fibonacci and the Galois shown below.

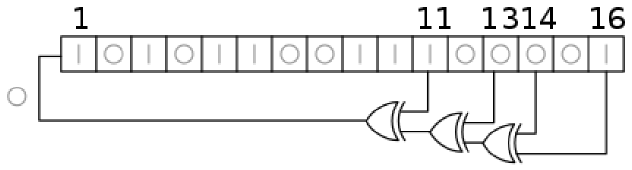

Seed
N
LFSR Counter
N
Output
reset_n
clock
D1 D2 D3 D4 D5 D6 D7 D8 D9 D10 D11 D12 D13 D14 D15 D16
Fibonacci LFSR Implementation for $X^{16} + X^{14} + X^{13} + X^{11} + 1$
D1 D2 D3 D4 D5 D6 D7 D8 D9 D10 D11 D12 D13 D14 D15 D16
Galois LFSR Implementation for $X^{16} + X^{14} + X^{13} + X^{11} + 1$

Galois LFSR take his name from the french mathematician Évariste Galois and it is also known as Modular LFSR or Internal-Xor-LFSR.

In general Galois implementation offers higher throughput than its Fibonacci counterpart and this due to the fact that the maximum clocking frequency is determined strongly by the maximum propagation delay of the circuit. Fibonacci configuration have to XOR several bits together requiring either cascaded 2 input XOR gates or multiple-input XOR gates, whereas Galois configuration use 2 input XOR gates, so their propagation delay is minimized and can run at higher frequencies.

Fibonacci LFSR is also known as Standard LFSR or External-Xor-LFSR.

2.2 How it works



Of all the bits of the shift register, in particular, those involved in the feedback function are called "taps".

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

The arrangement of taps for feedback in an LFSR can be expressed in finite field arithmetic as a polynomial mod 2. This means that the coefficients of the polynomial must be 1s or 0s. This is called feedback polynomial or reciprocal characteristic polynomial. In this implementation the feedback polynomial is the one shown above which means that the tap list is formed by the bits [16, 14, 13, 11] and the constant term "one" specifies, as shown in the figure, the input for the first bit at the very first iteration.

The LFSR is said maximal if and only if the corresponding feedback polynomial is primitive, that means it's necessary but not sufficient that the number of taps must be even and the positional indexes of the taps must be setwise co-prime; it is equivalet to say that the GCD of all the taps is equal to 1.

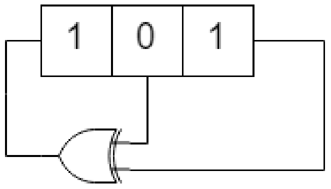
We can directly verify that our polynomial is primitive because the tap list [16, 14, 13, 11] is composed by an even number of elements and at the same time GCD(16, 14, 13, 11) = 1 .
When a polynomial is maximal an LFSR of n bits passes through all the possible $2^n - 1$ combinations before ending its period.

There is the -1 term because the impossible transaction is subtracted; in particular for Fibonacci architecture is the sequence of only 0.

2.3 Component working example

Can be immediatly observed, since we are talking of a maximal 3 bit LFSR, that the period is composed of $2^3 - 1 = 7$ different states if the characteristic polynomyal is maximal.

Feedback polynomial for a 3 bit LFSR is $1 + x^2 + x^3$ and its maximality is guaranteed because the two necessary but not sufficient conditions are satisfied and it is included in the maximal feedback polynomial table



1.	[1 0 1]	Feedback bit $= \text{bit}[2] \oplus bit[3] = 1$
2.	[1 1 0]	Feedback bit $= \text{bit}[2] \oplus bit[3] = 1$
3.	[1 1 1]	Feedback bit $= \text{bit}[2] \oplus bit[3] = 0$
4.	[0 1 1]	Feedback bit $= \text{bit}[2] \oplus bit[3] = 0$
5.	[0 0 1]	Feedback bit $= \text{bit}[2] \oplus bit[3] = 1$
6.	[1 0 0]	Feedback bit $= \text{bit}[2] \oplus bit[3] = 0$
7.	[0 1 0]	Feedback bit $= \text{bit}[2] \oplus bit[3] = 1$
8.	[1 0 1]	Feedback bit $= \text{bit}[2] \oplus bit[3] = 1$

As expected values are different for seven iterations after which LFSR starts a new period identical to the previous one; this situation was known a priori due to the features of the feedback polynomial and this is just its proof.

2.4 Mathematical representation

Both Fibonacci and Galois configuration can be represented mathematically by the mean of linear algebra equations system in matrix form as reported below

$$\begin{pmatrix} a_k \\ a_{k+1} \\ a_{k+2} \\ \vdots \\ a_{k+n-1} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_0 & c_1 & c_2 & \cdots & c_{n-1} \end{pmatrix} \begin{pmatrix} a_{k-1} \\ a_k \\ a_{k+1} \\ \vdots \\ a_{k+n-2} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_0 & c_1 & c_2 & \cdots & c_{n-1} \end{pmatrix}^k \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

For the Galois form, we have

$$\begin{pmatrix} a_k \\ a_{k+1} \\ a_{k+2} \\ \vdots \\ a_{k+n-1} \end{pmatrix} = \begin{pmatrix} c_0 & 1 & 0 & \cdots & 0 \\ c_1 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{n-1} & 0 & 0 & \cdots & 0 \end{pmatrix} \begin{pmatrix} a_{k-1} \\ a_k \\ a_{k+1} \\ \vdots \\ a_{k+n-2} \end{pmatrix} = \begin{pmatrix} c_0 & 1 & 0 & \cdots & 0 \\ c_1 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{n-1} & 0 & 0 & \cdots & 0 \end{pmatrix}^k \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Mathematical model describes the algorithm used to obtain a new register state by the product of the companion matrix of the feedback polynomial with the seed, or initialization vector.

The output for a given iteration is obtained simply by changing the value of the exponent k of the companion matrix.

The only difference in the two linear systems is where polynomial coefficients are placed as matrix component; for Fibonacci implementation it is the last row of the companion matrix while for Galois is the first column.

One important implication of that mathematical model is that it is easy to prove that LFSR sequences are reversible.

2.5 Possible applications

Linear feedback shift registers have a lot of different applications in nowadays technologies.

They are useful in applications that require very fast pseudo-random sequences generation, such as direct-sequence spread spectrum radio or approximation of white noise in programmable sound generators.

2.5.1 Counter

The repeating sequence of states of an LFSR allows it to be used as a clock divider or as a counter. LFSR counters have simpler feedback logic than natural binary counters or Gray-code counters, and therefore can operate at higher clock rates.
Gray code is an ordering of the binary numeral system such that two successive values differ in only one bit (binary digit). Nowadays Gray codes are widely used to facilitate error correction in digital communications such as digital terrestrial television and some cable TV systems.

Going back to counters, it is necessary to ensure that the LFSR never enters an all-zero state, for example by presetting the seed to any other state in the sequence.

The table of primitive polynomials shows how LFSR can be arranged in Fibonacci or Galois form to give maximal periods. One can obtain any other period by adding to an LFSR that has a longer period some logic that shortens the sequence by skipping some states.

2.5.2 Cryptography

In the past they have been used as pseudo-random number generators for use in stream ciphers, especially in military cryptography, due to the ease of construction from simple electromechanical and electronic circuits. They have long periods and very uniformly distributed output streams that is a crucial requirement in crittanalysis.

The greatest problem of this component with respect to security analysis is the linearity of the system, leading to fairly easy attack that tries to discover the feedback polynomial.

A practical example of cyber-attack on this component is as follow: given a set of known plaintext and the corresponding set of ciphertext, an attacker can intercept and recover a subset of LFSR output stream, and from that set can construct an LFSR of minimal size that simulates the intended one.
This particular LFSR can then be use the intercepted set of ciphertext to recover the remaining plaintext.

Considering this problem this kind of components are mostly used in pair with other hardware components that provides non linearity to the system; but the ease of the design make this configuration particularly efficient to generate symmetric keys in stream ciphers and the efficiency grow up with the increasing size of the key. Important LFSR-based stream ciphers include A5/1 and A5/2, used in GSM mobile phones, E0, used in Bluetooth, and the shrinking generator. The A5/2 cipher has been broken and both A5/1 and E0 have serious weaknesses.

The linear feedback shift register has a strong relationship to Linear Congruential Generators (LCG) of which the most classic example is the pseudo-random generation of numbers offered by std libraries of programming languages. like C using the rand() mod n function.

2.5.3 BIST - Built In Self Test

Sometimes LFSR are used to get a fully and complete automated test vector generation applying them to the Circuit Under Test (CUT) or Device Under Test (DUT) and then verifying its response.

LFSR with reseeding scheme is used to generate test vectors for circuits.
Also in this case LFSR is used as a pseudo-random sequence generator.
These patterns are applied to a certain circuit that use them as input.

For 16bit inputs there are at most 65534 different digital combinations if and only if feedback polynomial is maximal.
One of the property of a maximal LFSR is that it reaches all the possible different values for a given number of bit before restart its period; that property makes it very useful to do full input coverage test for a software that accepts a big set of input values hardly manageable by hand.

2.5.4 LCG - Linear Congruential generator

A Linear Congruential Generator (LCG) is an algorithm that yields a sequence of pseudo-randomized numbers calculated with a discontinuous piecewise linear equation.

The method represents one of the oldest, tested and best-known pseudo-random number generator algorithms. Theory behind them is relatively easy to understand, and they can be easily and fastly implemented, especially on computers where hardware can natively provide modular arithmetic.

Linear Congruential Generators have the problem that all of the bits in each number are usually not equally randomized (not equally distributed).
Either the modulus is a power of 2 and the least significant bits (LSB) are less random than the most-significant bits (MSB), or the modulus is not a power of 2 and the outputs are biased.

A linear feedback shift register as PRNG (Pseudo-Random Number Generator) produces a stream of pseudo-random bits, each of which is truly pseudo-random and can be implemented with essentially the same amount of memory as an equivalent LCG but with a bit more computational requirements.

Relation between LFSR and LCG is really close. Given a few values in the sequence, some techniques can predict the following values in the sequence for not only Linear Congruential Generators but any other polynomial congruent generator as the LFSR.

2.5.5 Scrambling

LFSR are also used in broadcast communications to prevent short repeating sequences like sequences of zero or one from forming spectral lines that may complicate symbol tracking at the receiver or interfere with other transmissions.

Data bit sequence is combined with the output of an LFSR before modulation and transmission. This operation is called scrambling and is reversed at the receiver after demodulation.

3 VHDL

VHDL is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as Field Programmable Gate Arrays (FPGA) and Integrated Circuits (IC).
VHDL can also be used as a general purpose parallel programming language.

The IEEE Standard 1076 defines the VHSIC Hardware Description Language or VHDL.

It was originally developed by the USAF (United States Air Force) in 1983 with Texas Instruments and IBM. The language has undergone numerous revisions and has a variety of sub-standards associated with it that augment or extend it in important ways.

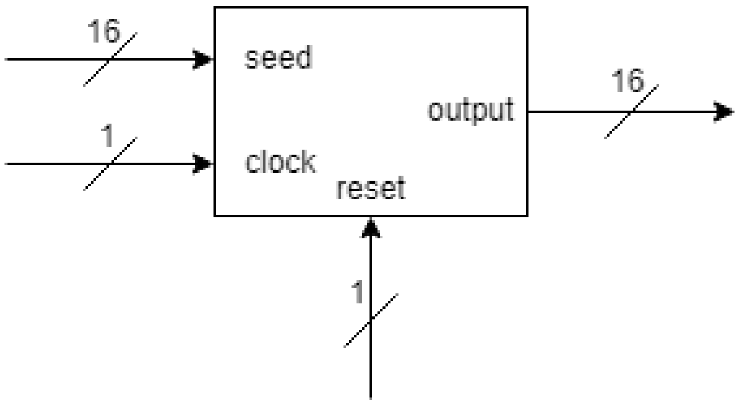
One can design hardware in a VHDL IDE, for FPGA implementation such as Xilinx Vivado, Intel Quartus (ex Altera) or Mentor Graphics MODELSIM HDL Designer, to produce the RTL schematic of the desired circuit. After that, the generated schematic can be verified using simulation software which shows the waveforms of inputs and outputs of the circuit after generating the appropriate testbench.

To generate an appropriate testbench for a particular circuit or VHDL code, the inputs have to be defined correctly. For example, for clock signal, a loop process or an iterative statement is required.

When a VHDL model is translated into the "gates and wires" with the process called "Place and Route", it is mapped onto a programmable logic device such as a CPLD (Complex Programmable Logic Device) or FPGA, then it is the actual hardware being configured.

VHDL code is executed as a form of physical hardware component on the chip.

3.1 Implemented design



High-level block diagram of the implemented LFSR in Fibonacci configuration is reported above.

The component is characterized by an initial value, the seed, that is a 16 bit signal represented in VHDL as STD_LOGIC_VECTOR.
Seed is the initial value from which the linear feedback shift register starts its period after every reset.

Clock is a STD_LOGIC signal and it is used to drive the component state transition. In particular register state changes at each rising edge of the clock signal if the reset signal is in the logic state zero.

Reset is a one bit STD_LOGIC signal that is active at logic state one and brings always back the register state to the seed value.

Output is a STD_LOGIC_VECTOR signal of 16 bit that is the register internal state that changes for each clock rising edge if and only if the reset signal is low.

3.2 Code

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY LFSR_16_BIT IS
              PORT(
                    clock        : IN  STD_LOGIC;
                    reset        : IN  STD_LOGIC;
                    input_value  : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
                    output_value : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
                  );
END LFSR_16_BIT;


ARCHITECTURE BHV OF LFSR_16_BIT IS
    SIGNAL      feedback : STD_LOGIC;
    SIGNAL     act_value : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL    next_value : STD_LOGIC_VECTOR(15 DOWNTO 0);

BEGIN
--------------------------------------------------------------------------------------
--                STATEMENTS INSIDE PROCESS ARE SEQUENTIALLY EXECUTED               --
--                OUTSIDE EVERYYHING IS PARALLEL... PROCESS ITSELF IS               --
--                CONCURRENTLY EXECUTED WITH THE REST OF  THE  ENTITY               --
--------------------------------------------------------------------------------------
ragister_state : PROCESS(clock, reset)
     BEGIN
            IF(reset = '1') THEN
                act_value <= input_value;
            ELSIF(rising_edge(clock)) THEN
                act_value <= next_value;
            END IF;
     END PROCESS;
--------------------------------------------------------------------------------------
--               HERE FEEDBACK IS EXECUTED AND STATE AND OUTPUT ARE UPDATED         --
--------------------------------------------------------------------------------------
   feedback     <= act_value(0) XOR act_value(2) XOR act_value(3) XOR act_value(5); --
   next_value   <= feedback & act_value(15 DOWNTO 1);                               --
   output_value <= act_value;                                                       --
--------------------------------------------------------------------------------------

END BHV;
```

The first block, ENTITY LFSR_16_BIT, describes the block diagram specifying inputs and outputs of the component by the meaning of the PORT() function.

In this implementation we have three inputs:

1. clock
2. reset
3. input_value

and one output value:

1. output_value

The second block describes the architecture of the component that uses three internal signal:

1. feedback
2. act_value
3. next_value

The first of the three signals represents the output of the feedback

The second represents the actual state of the register as a STD_LOGIC_VECTOR

The third one represents the new state as a STD_LOGIC_VECTOR

The second part is the PROCESS that is a particular construct of VHDL inside which everything is executed sequentially. All the other code sections are concurrently executed and the process itself is executed in parallel with all the rest.

A process is declared with a list of elements that is called sensitivity list and the process itself is triggered and begin its execution whenever an element in such list changes its state.

In this implementation sensitivity list is composed by the two relevant signals [clock, reset] that allow state register transition; in fact if the logic value of reset is '1' then the register is brought back to the seed value, in this case the signal named initial_value.

If reset is in the logic state '0' (or low) and the clock change its logic state from low to high (or from '0' to '1') then the internal value of the register is updated with the new value. This not happens when the clock signal changes from high to low because this design, the design of the whole component, is rising-edge triggered defined using the macro rising_edge() in the if condition on the clock state in the process.

In the last part of the entity it is executed the internal logic that leads to the new register state.

Feedback signal takes the result of the feedback function described by the characteristic polynomial while next_value, the signal containing the new register state, is obtained by concatenating feedback signal with the previous register value right shifted by one bit and so from 15 to 1; bit 0 is not more in the register.

Vector had been declared using the keyword DOWNTO so the most significant bit (MSB) is the 15^{th} and bit 0 is the least significant one (LSB) and so bit 0 is the one that disappears at each transition.

3.3 Modelsim IDE

ModelSim is a multi-language HDL simulation environment developed by Mentor Graphics for simulation of hardware description languages like VHDL, Verilog and SystemC.
The company was founded in 1981 and sold to Siemens in 2017.

ModelSim can be used independently, or in conjunction with Intel Quartus Prime, Xilinx ISE or Xilinx Vivado.
Simulation is performed using the graphical user interface (GUI) or automatically using scripts.

3.4 Testbench

Testbench is a particular VHDL file used to test and make experiments on the implemented design.

The design to be tested is commonly called DUT, as Design Under Test, and must be placed as component inside the architecture implementation of the testbench entity.

Another important charcteristic of the testbench is that its entity is always empty but it has to be declared because it is needed to have one entity in each file and because it works as container for the instance of the DUT.

It is a very useful step in the digital component development process because it allows engineers to know if everything in the previous step (VHDL code implementation) went good and allows to proceed to more advanced steps closer to the physical implementation on FPGA.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
use STD.textio.all;
use ieee.std_logic_textio.all;

entity LFSR_16_BIT_TB is                        -- TESTBENCH HAS NO INTERFACE IT IS AN EMPTY ENTITY
end LFSR_16_BIT_TB;


ARCHITECTURE BHV OF LFSR_16_BIT_TB IS
        constant T_CLK   : time    := 10 ns;    -- Clock period
        constant T_RESET : time    := 25 ns;    -- Period before the reset deassertion
    ------------------------------------------------------------------------------------------------------------------
    --                                              Testbench signals                                              --
    ------------------------------------------------------------------------------------------------------------------
        signal clock_tb        : STD_LOGIC := '0';
        signal reset_tb        : STD_LOGIC := '1';
        signal input_value_tb  : STD_LOGIC_VECTOR(15 DOWNTO 0) := "1010110011100001";
        signal output_value_tb : STD_LOGIC_VECTOR(15 DOWNTO 0);
        signal en_tb           : std_logic := '1';
        signal end_sim         : std_logic := '1';
    ------------------------------------------------------------------------------------------------------------------
    --                         Component to test (DUT) declaration (DUT = Design Under Test)                       --
    ------------------------------------------------------------------------------------------------------------------
        COMPONENT LFSR_16_BIT IS
              PORT(
                    clock        : IN  STD_LOGIC;
                    reset        : IN  STD_LOGIC;
                    input_value  : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
                    output_value : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
        END COMPONENT;

        begin
          clock_tb <= (not(clock_tb) and end_sim) after T_CLK / 2;  -- The clock toggles after T_CLK / 2
          reset_tb <= '0' after T_RESET;                            -- Deasserting the reset after T_RESET nanoseconds

          LFSR : LFSR_16_BIT
                    port map(
                              clock        => clock_tb,
                              reset        => reset_tb,
                              input_value  => input_value_tb,
                              output_value => output_value_tb
                            );
          tb_process: PROCESS(clock_tb, reset_tb)
                variable t : integer := 0;                          -- variable to count clock cycle after the reset
                file file_handler     : text open write_mode is "output_to_be_tested.txt";
                Variable row          : line;
                Variable data_write   : std_logic_vector(15 DOWNTO 0);

          begin
            if(reset_tb = '1') then
                  t := 0;
                ELSIF(rising_edge(clock_tb)) THEN
                  CASE(t) IS
                     when 65535 => end_sim <= '0';
                     WHEN OTHERS => data_write := output_value_tb; write(row, data_write); writeline(file_handler, row);
                  END CASE;
                  t := t + 1; -- variable is updated exactly here
                END IF;
          END PROCESS;
END BHV;
```

In the first part, as usual, internal signals are declared.
In particular there are two signals, T_CLK and T_RESET, used for the temporization of the DUT.

Then the component declaration is added as requested by the syntax of VHDL.

After this declarative part there is the logic part composed by the wiring procedure and a process that regulates the component behaviour acting on clock and reset signals.

The function PORT_MAP() is used to bind testbench signals to the ones of the desired component; it is associative left to right so the signal on the left side, the one of the component, is binded to the corresponding one of the tesbench on the right side.

tb_process has some internal variables:

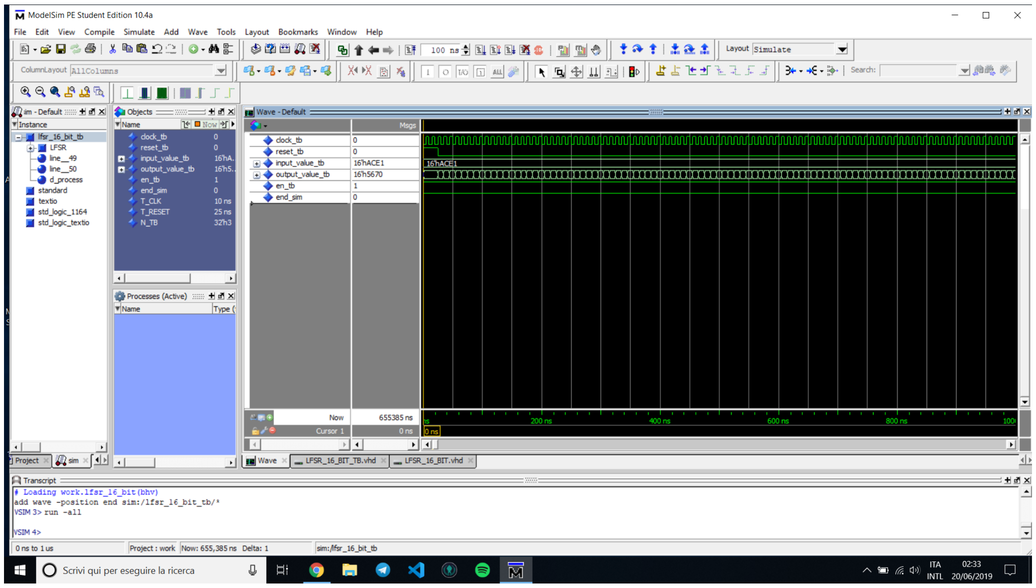
1. t
2. file_handler
3. row
4. data_write

t is the time variable used to manage the time in terms of clock cycles while the other three are used to save the output value of the register in a text file that is going to be used in the software simulation described in the next chapter.

It was chosen to end the simulation after 65535 to get enough value to prove that the period is going to restart with the same values (according to the feedback polynomial law previously shown) and all the values within the period are one different from all the other and different from the value 0x0000

3.5 Simulation output

Turning into Modelsim IDE the following output has been retrieved

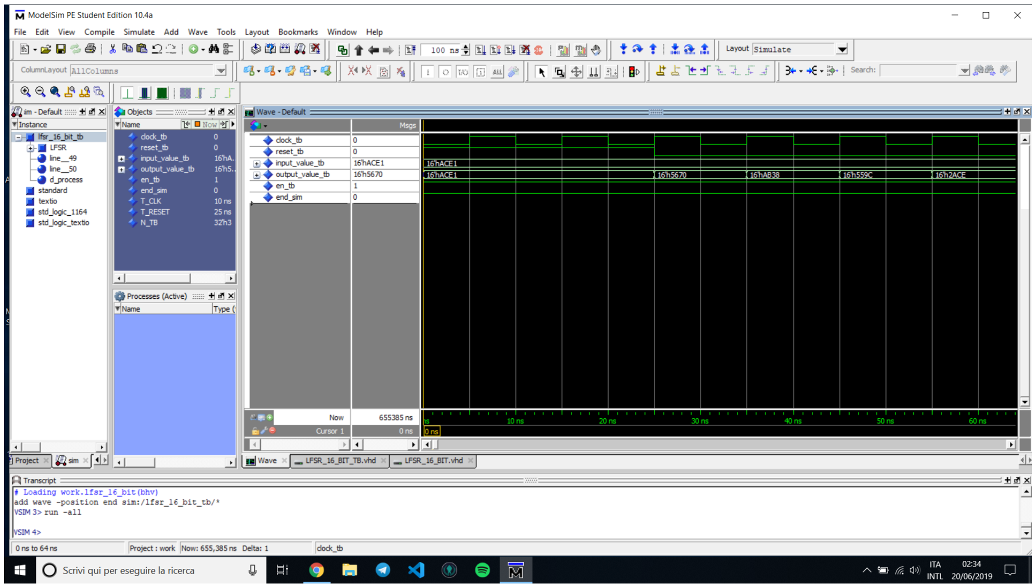


```
clock_tb        0
reset_tb        0
input_value_tb  16'hACE1
output_value_tb 16'h5670
en_tb           1
end_sim         0
```

```
# Loading work.lfsr_16_bit(bhv)
add wave -position end sim:/lfsr_16_bit_tb/*
VSIM 3> run -all

VSIM 4>
```

Zoomming in we can observe the following signals and values



```
clock_tb        0
reset_tb        0
input_value_tb  16'hACE1
output_value_tb 16'h5670
en_tb           1
end_sim         0
```

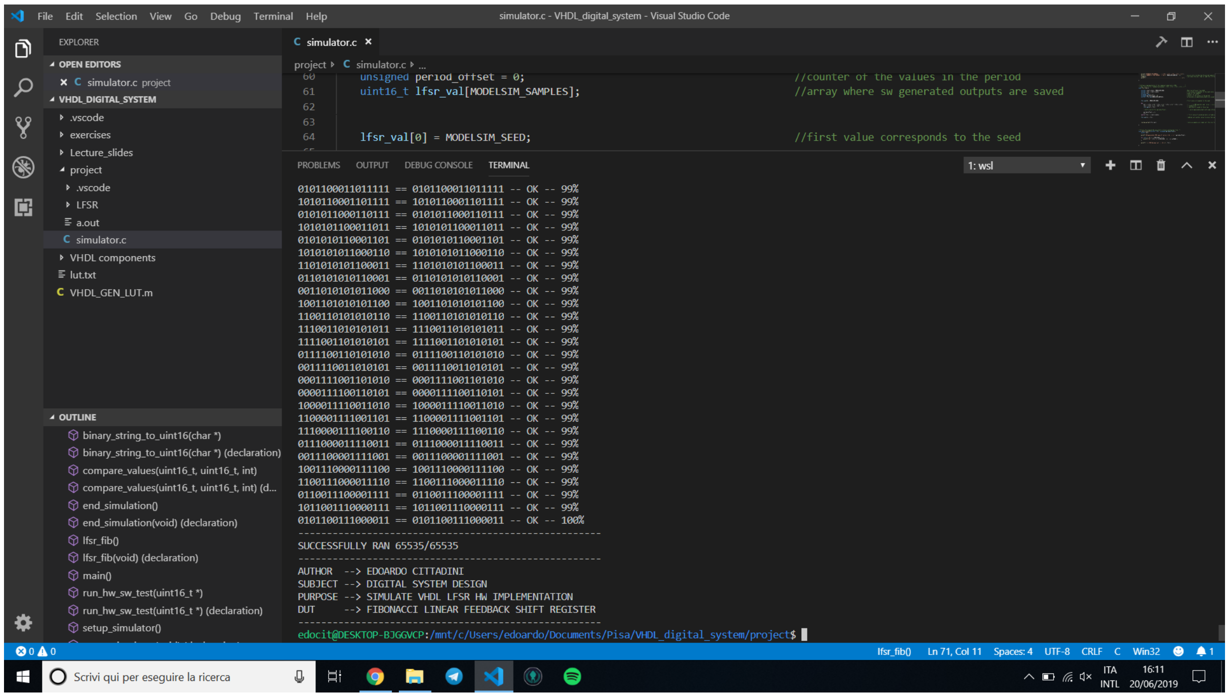
```
# Loading work.lfsr_16_bit(bhv)
add wave -position end sim:/lfsr_16_bit_tb/*
VSIM 3> run -all

VSIM 4>
```

Value is maintained or reported to the seed value while the reset signal is high and after that we can see some of the iteration of the first period.

4 Simulator

Since the test of the component involves in a lot of different values a simulator was developed both to check the correct behaviour of the component at each state transition (logic implementation checking) and validate the previous outputs given by the the testbench simulation (system checking).



```c
unsigned period_offset = 0;                                         //counter of the values in the period
uint16_t lfsr_val[MODELSIM_SAMPLES];                                //array where sw generated outputs are saved


lfsr_val[0] = MODELSIM_SEED;                                        //first value corresponds to the seed
```

```
0101100011011111 == 0101100011011111 -- OK -- 99%
1010110001101111 == 1010110001101111 -- OK -- 99%
0101011000110111 == 0101011000110111 -- OK -- 99%
1010101100011011 == 1010101100011011 -- OK -- 99%
0101010110001101 == 0101010110001101 -- OK -- 99%
1010101011000110 == 1010101011000110 -- OK -- 99%
1101010101100011 == 1101010101100011 -- OK -- 99%
0110101010110001 == 0110101010110001 -- OK -- 99%
0011010101011000 == 0011010101011000 -- OK -- 99%
1001101010101100 == 1001101010101100 -- OK -- 99%
1100110101010110 == 1100110101010110 -- OK -- 99%
1110011010101011 == 1110011010101011 -- OK -- 99%
1111001101010101 == 1111001101010101 -- OK -- 99%
0111100110101010 == 0111100110101010 -- OK -- 99%
0011110011010101 == 0011110011010101 -- OK -- 99%
0001111001101010 == 0001111001101010 -- OK -- 99%
0000111100110101 == 0000111100110101 -- OK -- 99%
1000011110011010 == 1000011110011010 -- OK -- 99%
1100001111001101 == 1100001111001101 -- OK -- 99%
1110000111100110 == 1110000111100110 -- OK -- 99%
0111000011110011 == 0111000011110011 -- OK -- 99%
0011100001111001 == 0011100001111001 -- OK -- 99%
1001110000111100 == 1001110000111100 -- OK -- 99%
1100111000011110 == 1100111000011110 -- OK -- 99%
0110011100001111 == 0110011100001111 -- OK -- 99%
1011001110000111 == 1011001110000111 -- OK -- 99%
0101100111000011 == 0101100111000011 -- OK -- 100%
-----------------------------------------------------
SUCCESSFULLY RAN 65535/65535
-----------------------------------------------------
AUTHOR  --> EDOARDO CITTADINI
SUBJECT --> DIGITAL SYSTEM DESIGN
PURPOSE --> SIMULATE VHDL LFSR HW IMPLEMENTATION
DUT     --> FIBONACCI LINEAR FEEDBACK SHIFT REGISTER
-----------------------------------------------------
edocit@DESKTOP-BJGGVCP:/mnt/c/Users/edoardo/Documents/Pisa/VHDL_digital_system/project$
```

Graphic interface shows respectively from left to right the value generated by the software implementation of the LFSR, the value generated by the MODELSIM testbench, the state of the test and the accomplished percentage with respect to the overall number of samples taken by the hardware simulation.

At the end of the program a report about the number of tests successfully ran and some general data about the project such as DUT and purpose had been printed.

4.1 C code implementation

```c
/* Software implementation of the component modelled in VHDL              *
 * taps: 16 14 13 11; feedback polynomial: x^16 + x^14 + x^13 + x^11 + 1 */

void lfsr_fib(){

    uint16_t start_state = MODELSIM_SEED; //Must be different from 0x00
    uint16_t lfsr = start_state;          //first value corresponds to the seed value
    uint16_t bit;                         //used for the feedback
    unsigned long i = 1;                  //counter for cycles
    unsigned period_offset = 0;           //counter of the values in the period
    uint16_t lfsr_val[MODELSIM_SAMPLES];  //array where sw generated outputs are saved


    lfsr_val[0] = MODELSIM_SEED;          //first value corresponds to the seed



    do{
        bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5));  // ^ is XOR operator
        lfsr = (lfsr >> 1) | (bit << 15); //feedback bit
        lfsr_val[i] = lfsr;               // new value is saved in the array

        period_offset += 1;
        i++;
    }while (lfsr != start_state);         //first value of second period is already added here


    lfsr_val[i] = lfsr;                   //another value to prove that period has begun again


    //once sw samples are ready call the core function to execute SW and VHDL output comparison
    run_hw_sw_test(lfsr_val);

}
```

Function above implement in software exactly what had been designed in VHDL, so it is a C programming language abstraction of the design under test which states are saved and compared with respect to the specific sample times used as parameters in MODELSIM simulation.

5 Synthesis process

Logic synthesis is an electronic process by which an abstract specification of desired circuit behavior, typically at Register Transfer Level (RTL), is turned into a design implementation in terms of logic gates, typically by a computer program called synthesis tool.

Common examples of this process are strictly related to the synthesis of hardware designs written in hardware description languages such VHDL.

Some synthesis tools generate bitstreams for programmable logic devices such as PALs (Programmable Array Logic) or FPGAs (Field Programmable Gate Array), while others target is the creation of ASICs (Application Specific Integrated Circuit). Logic synthesis is one aspect of electronic design automation.

5.1 FPGA vs ASIC

FPGA stands for Field Programmable Gate Array; it is an integrated circuit which can be “field” programmed to work as the intended design. It means it can work as a microprocessor, or as an encryption unit, or graphics card, or everything whose logic is implementable using a lesser or equal number of CLBs of the unit.

As implied by the name itself, the FPGA is field programmable. So, an FPGA working as a microprocessor can be reprogrammed to function as a graphics card. FPGA is made up of thousands of Configurable Logic Blocks (CLBs) embedded in an ocean of programmable interconnections. The CLBs are primarily made of Look-Up Tables (LUTs), Multiplexers and Flip-Flops. They can implement complex logic functions. Apart from CLBs, and routing interconnections, many FPGAs also contain dedicated hard-silicon blocks for various functions such as Block RAM, DSP (Digital Signal Processor) Blocks, External Memory Controllers, PLLs (Phase Locked Loop), Multi-Gigabit Transceivers.

A recent trend is providing a hard-silicon processor core (such as ARM Cortex A9 in case of Xilinx Zynq) inside the FPGA itself so that the processor can take care of non-critical tasks whereas FPGA can take care of high-speed acceleration which cannot be done using processors. These dedicated hardware blocks are critical in competing with ASICs.

ASIC stands for Application Specific Integrated Circuit. As the name implies, ASICs are designed for one only purpose and they will work the same for their whole operating life.

For example, the CPU inside a smartphone is an ASIC. It is meant to work as a CPU for its whole life. Its logic function cannot be changed to anything else because its digital circuitry is made up of permanently connected gates and flip-flops in silicon.

The logic function of ASIC is specified in a similar way as in the case of FPGAs, using hardware description languages such as Verilog or VHDL.

The difference in case of ASIC is that the resultant circuit is permanently drawn into silicon whereas in FPGAs the circuit is made by connecting a number of configurable blocks.

5.2 Vivado IDE

Vivado Design Suite is an Integrated Development Environment (IDE) produced by Xilinx for synthesis and analysis of HDL designs. Vivado has features for system on a chip development and high-level synthesis.

Unlike ISE which relied on MODELSIM for simulation, Vivado System Edition includes an in-built logic simulator and it also introduces high-level synthesis, with a toolchain that allows to design component also in C language code that will be translated into programmable logic.

Replacing the 15 year old ISE with Vivado Design Suite took 1000 person-years and cost US $200 million.

The latters are just numbers to make the reader an idea of the complexity of the software and its environment.

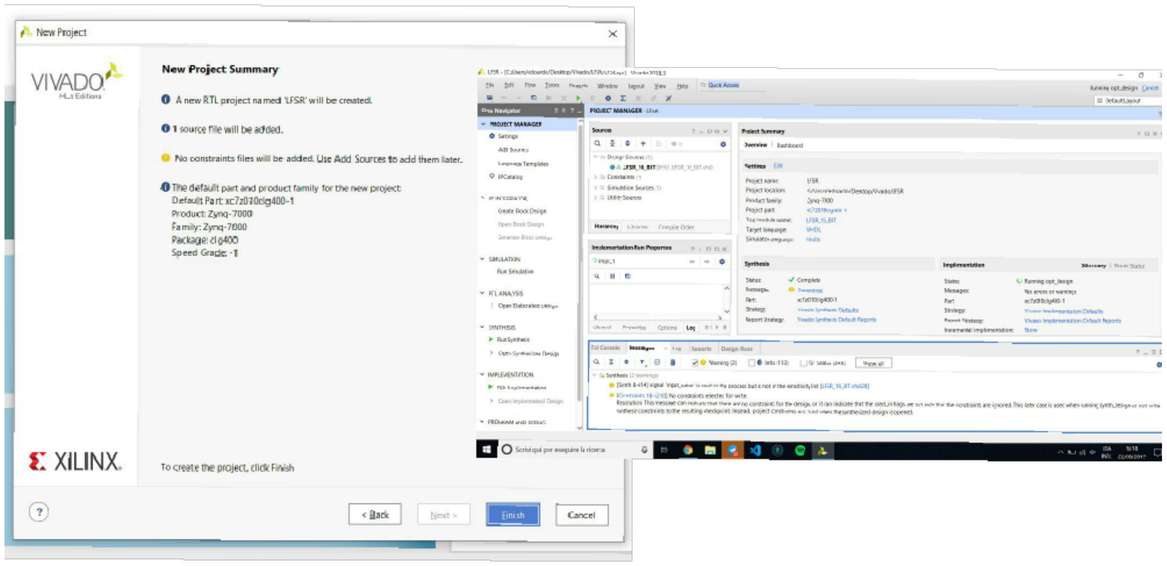
Vivado is able to provide full analysis of the component so it is possible to get the schematic, power analysis, timing analysis, temperature and hardware constraints report.

The developed design needs first to be converted in a particular file format called bitstream, that is a sort of FPGA executable file, and then Vivado loader will flash this file into the FPGA EEPROM.

5.3 Synthesis

First a new Vivado project was created and can be observed that since neither constraints or constraint file were applied or specified, after synthesis is expected to have at least a warning that will specify that no constraints were specified or used for such design.

This situation is reported in figures below which are respectively on the left the dialog window without constraints at creation time and on the right the reported warning after synthesis ran.



New Project Summary

A new RTL project named 'LFSR' will be created.

1 source file will be added.

No constraints files will be added. Use Add Sources to add them later.

The default part and product family for the new project:
Default Part: xc7z010clg400-1
Product: Zynq-7000
Family: Zynq-7000
Package: clg400
Speed Grade: -1

To create the project, click Finish

Synthesis
Status: Complete
Messages: 2 warnings
Part: xc7z010clg400-1
Strategy: Vivado Synthesis Defaults
Report Strategy: Vivado Synthesis Default Reports

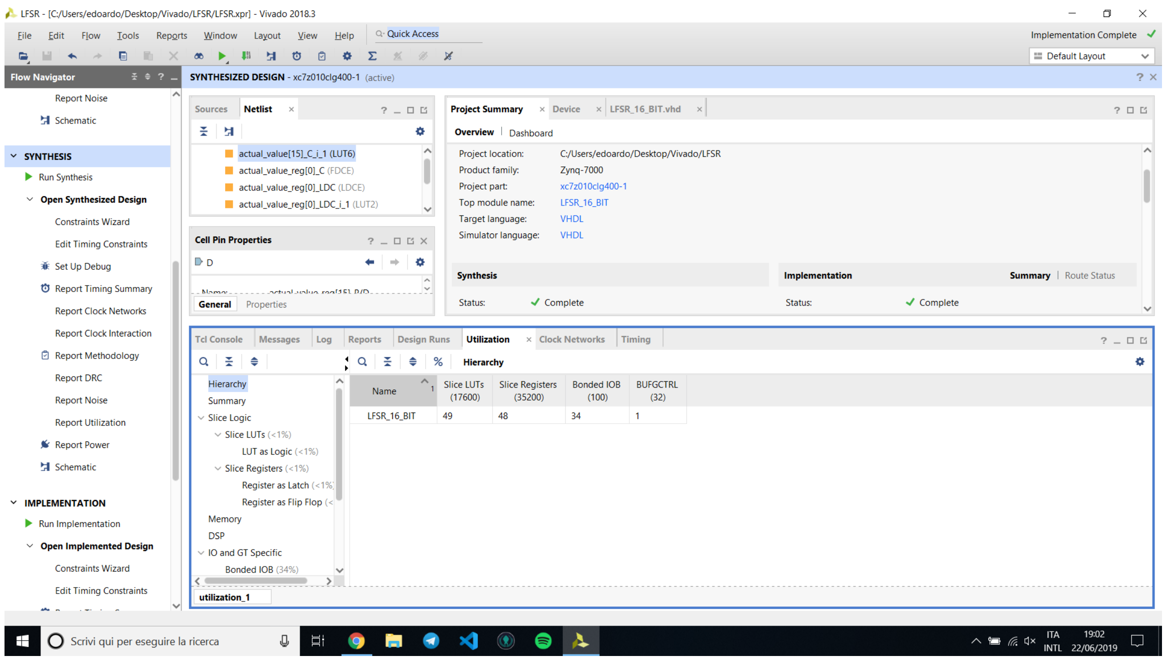
Implementation
Status: Running opt_design
Messages: No errors or warnings
Part: xc7z010clg400-1
Strategy: Vivado Implementation Defaults
Report Strategy: Vivado Implementation Default Reports
Incremental implementation: None

Warning (2) Info (112) Status (246)

Synthesis (2 warnings)
[Synth 8-614] signal 'input_value' is read in the process but is not in the sensitivity list [LFSR_16_BIT.vhd:28]
[Constraints 18-5210] No constraints selected for write.
Resolution: This message can indicate that there are no constraints for the design, or it can indicate that the used_in flags are set such that the constraints are ignored. This later case is used when running synth_design to not write synthesis constraints to the resulting checkpoint. Instead, project constraints are read when the synthesized design is opened.

Another warning is shown about the fact that signal input_value is used inside the process but it is not present in its sensitivity list.
This is not an important warning because input_value is just a reference signal and it is used whenever a reset is applied to rollback LFSR state to its value, so it is not important to trigger the process with respect to this change.

5.4 Utilized resources



LFSR - [C:/Users/edoardo/Desktop/Vivado/LFSR/LFSR.xpr] - Vivado 2018.3

SYNTHESIZED DESIGN - xc7z010clg400-1 (active)

Project location: C:/Users/edoardo/Desktop/Vivado/LFSR
Product family: Zynq-7000
Project part: xc7z010clg400-1
Top module name: LFSR_16_BIT
Target language: VHDL
Simulator language: VHDL

Synthesis Status: Complete
Implementation Status: Complete

Name	Slice LUTs (17600)	Slice Registers (35200)	Bonded IOB (100)	BUFGCTRL (32)
LFSR_16_BIT	49	48	34	1

FPGA implements logic designs using Look-up Tables (LUTs) so it is important to know how many of those elements are involved in order to physically implement the component.

For this project as shown 49 LUTs and 48 slice registers are involved.

The parameter IOB is the number of Input/Output signals bonded in the design; so in this project is 34 because we have 16 bit input_value, 16 bit output_value, 1 bit signal for clock and 1 bit signal for reset.

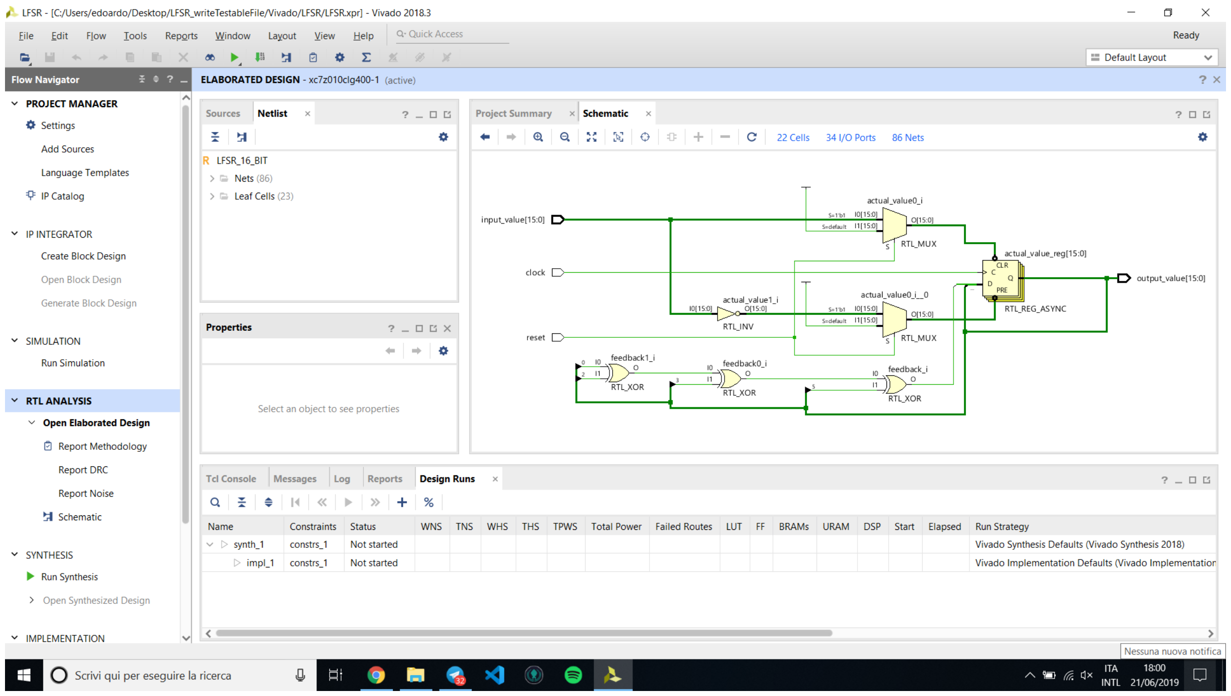
Switching those data to percentages, resulting occupancy values become:

1. Slice LUTs = 0.28 %
2. Slice registers = 0.14 %
3. IOB = 34.00 %

5.5 Technical analysis

After synthesis procedure is possible to get a lot of information about the behaviours, limitations and bounds of the physical component that will be implemented for that specific design with that specific constraints.

5.5.1 Schematic

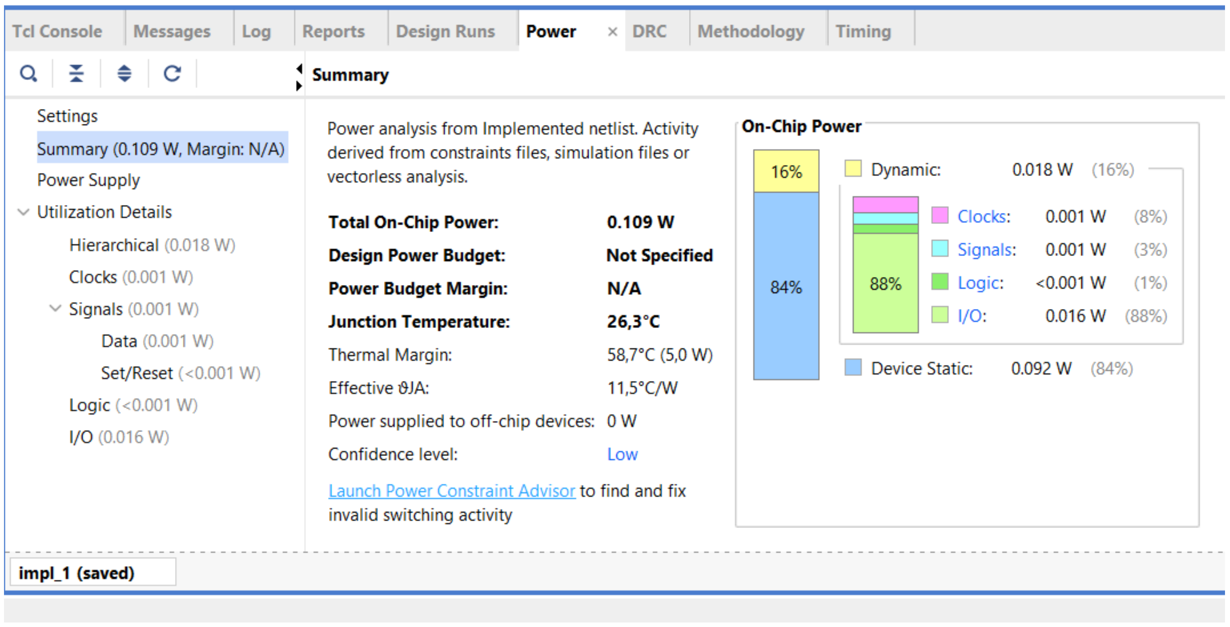


Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP	Start	Elapsed	Run Strategy
synth_1	constrs_1	Not started															Vivado Synthesis Defaults (Vivado Synthesis 2018)
impl_1	constrs_1	Not started															Vivado Implementation Defaults (Vivado Implementation

Schematic represented is not the same one that would have been designed from a human engineer by hand.

This because Vivado IDE represents circuit schematic as it has been optimized and converted by the synthesis tool in order to have as many advantages as possible with respect to the specific chosen FPGA target board.

5.5.2 Power analysis



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.109 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26,3°C**
Thermal Margin:	58,7°C (5,0 W)
Effective ϑJA:	11,5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:		0.018 W	(16%)
	Clocks:	0.001 W	(8%)
	Signals:	0.001 W	(3%)
	Logic:	<0.001 W	(1%)
	I/O:	0.016 W	(88%)
Device Static:		0.092 W	(84%)

Power analysis is divided in two sections: Static power analysis and Dynamic power analysis.

Static power is the power consumed by the component while there is no circuit activity. For example, the power consumed by a D Flip-Flop when neither the clock nor the D input have active inputs in fact all inputs are ”static” because they are fixed at a certain DC level.

Dynamic power is power consumed while the inputs are active. When inputs have AC activity, capacitances are charging and discharging and the power increases as result. The dynamic power includes both the AC component as well as the static component.

In this LFSR implementation the greatest power consumption is given by I/O operations.

Leakage current increases exponentially with Junction Temperature which results in higher Static Power.

Junction Temperature depends on various factors:

1. Device total power
2. Cooling system
3. Board
4. Ambience

By default Junction Temperature is computed using other thermal inputs:

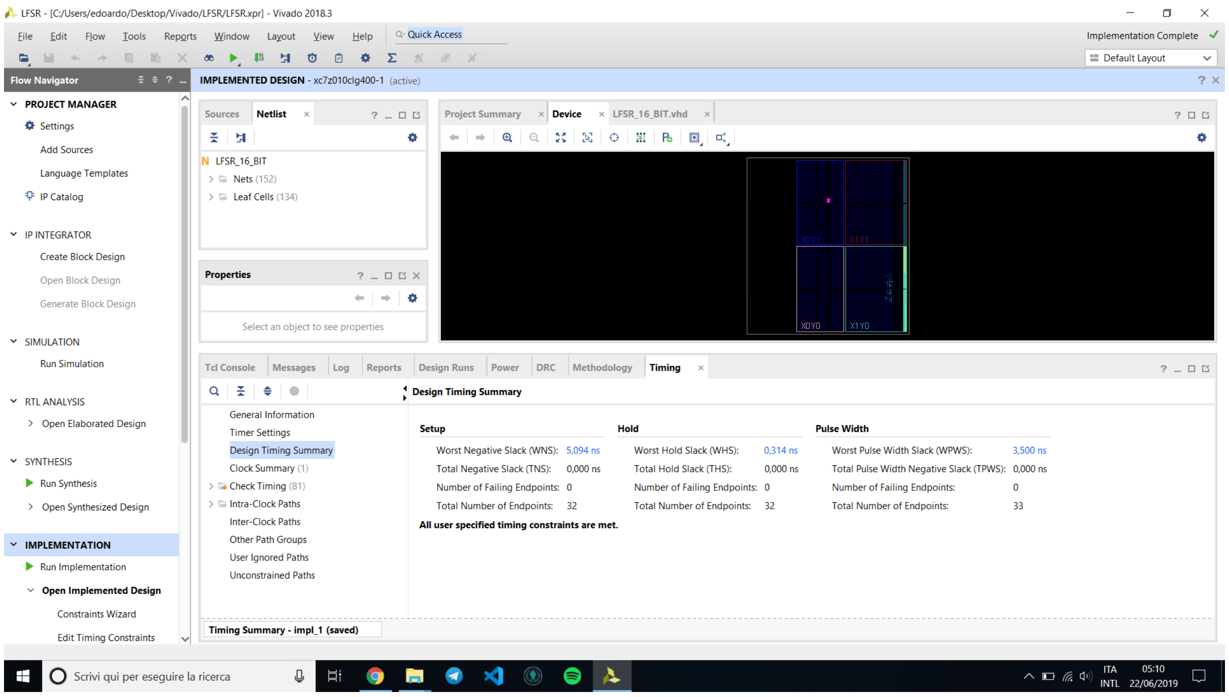
1. Ambient Temperature
2. Heat Sink
3. Specific board

It is directly proportional to Total Power, so it changes whenever dynamic Power value increases.

It is very important to specify the correct Junction Temperature to estimate accurate Static Power.

However in this implementation junction temperature was not specified so the value shown in figure is the one calculated by default from Vivado for such implementation on Zynq-7000 board.

5.5.3 Timing analysis



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5,094 ns	Worst Hold Slack (WHS):	0,314 ns	Worst Pulse Width Slack (WPWS):	3,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	32	Total Number of Endpoints:	32	Total Number of Endpoints:	33

All user specified timing constraints are met.

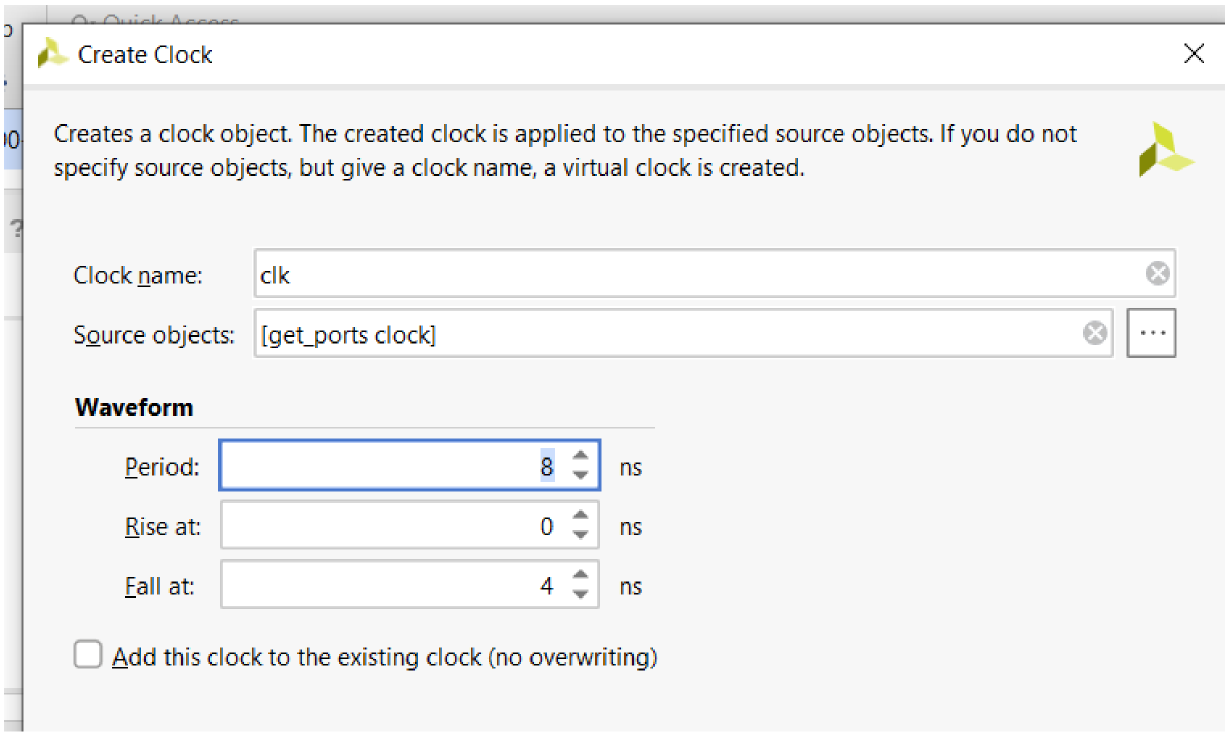
From timing analysis can be retrieved some important values that allow to see if given timing constraints such the clock fits or not for the component.

$$t_{CLK} = t_{setup} + t_{plogic} + t_{cq} + slack$$

Since no one of time to setup, time to logic propagation and time clock to queue can be modified the only parameter on which it is possible to work is the slack that is the ”free” space between two clock rising edges.

Maximum clock frequency can be calculated starting from the above formula as follow

$$t_{MAX\_CLK} = t_{CLK} - slack$$



Create Clock

Creates a clock object. The created clock is applied to the specified source objects. If you do not specify source objects, but give a clock name, a virtual clock is created.

Clock name: clk

Source objects: [get_ports clock]

Waveform

Period: 8 ns

Rise at: 0 ns

Fall at: 4 ns

☐ Add this clock to the existing clock (no overwriting)

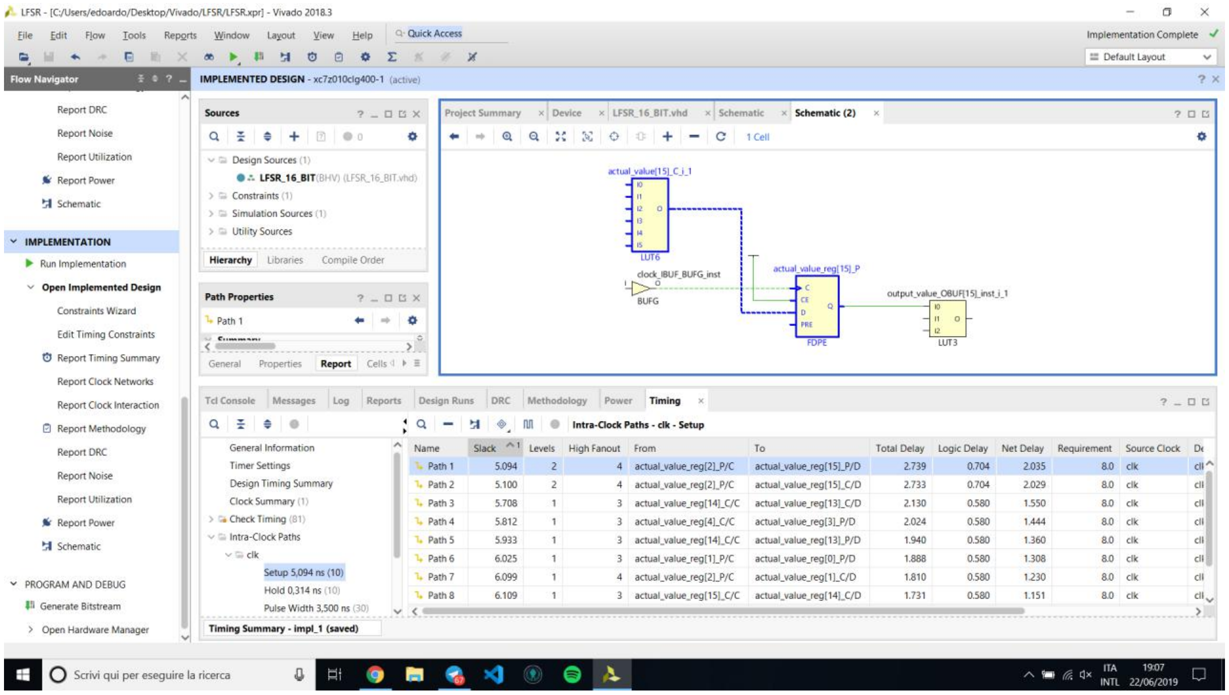
Clock was created as constraint with a period of 8ns and so 125MHz, slack is retrieved from Vivado timing analysis and precisely is $slack = 5.094$.

All the needed elements to find the maximum clock frequency are known so

$$t_{MAX\_CLK} = 8ns - 5.094ns = 2.906ns \simeq 344\ MHz$$

From the latter the maximum period value was found but since it is a floating value and Vivado clock constraint allows only to put integer values as period the conclusion is that smallest period for this LFSR is 3ns and so $clock = 333MHz$.

5.5.4 Critical path


Intra-Clock Paths - clk - Setup

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	De
Path 1	5.094	2	4	actual_value_reg[2]_P/C	actual_value_reg[15]_P/D	2.739	0.704	2.035	8.0	clk	cl
Path 2	5.100	2	4	actual_value_reg[2]_P/C	actual_value_reg[15]_C/D	2.733	0.704	2.029	8.0	clk	cl
Path 3	5.708	1	3	actual_value_reg[14]_C/C	actual_value_reg[13]_C/D	2.130	0.580	1.550	8.0	clk	cl
Path 4	5.812	1	3	actual_value_reg[4]_C/C	actual_value_reg[3]_P/D	2.024	0.580	1.444	8.0	clk	cl
Path 5	5.933	1	3	actual_value_reg[14]_C/C	actual_value_reg[13]_P/D	1.940	0.580	1.360	8.0	clk	cl
Path 6	6.025	1	3	actual_value_reg[1]_P/C	actual_value_reg[0]_P/D	1.888	0.580	1.308	8.0	clk	cl
Path 7	6.099	1	4	actual_value_reg[2]_P/C	actual_value_reg[1]_C/D	1.810	0.580	1.230	8.0	clk	cl
Path 8	6.109	1	3	actual_value_reg[15]_C/C	actual_value_reg[14]_C/D	1.731	0.580	1.151	8.0	clk	cl

By clicking on the Worst Negative Slack in Timing analysis panel Vivado automatically shows as schematic the critical path, that is the path over all the other that takes the greatest time to be completed.

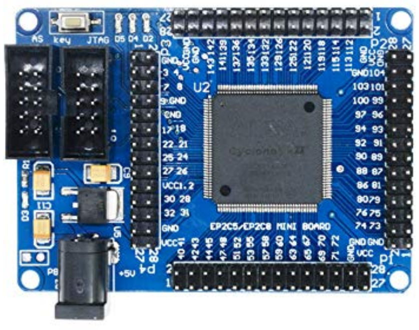
Critical path is very important because it represent a bound for the clocking frequency of the component.

What calculated in the previous section was done with respect to the critical path, so it means that other paths that are not critical may accept higher clock frequencies but this would lead to a time violation by the critical path, negatively affecting the value of the Worst Negative Slack (WNS).

Finally in order to have all the path in the circuit to have legal timing constraints the only path for which the analysis will be done is the critical one because if it respects the given clock constraint also the other for sure will.

6 Altera Cyclone II Board

The physical implementation of the component explained and simulated in chapter 3 is done using a cheap entry-level FPGA board that is Altera Cyclone II EP2C5ST144C8N.



This board has no complex port but has enough GPIO pins to drive 18 bit signals, one for each bit of the LFSR (16 pin for register state, 1 pin for clock and 1 pin for reset).

6.1 Quartus IDE

Intel (ex Altera) Quartus Prime is programmable logic device design software produced by Intel; prior to Intel's acquisition of Altera the tool was called Altera Quartus II.

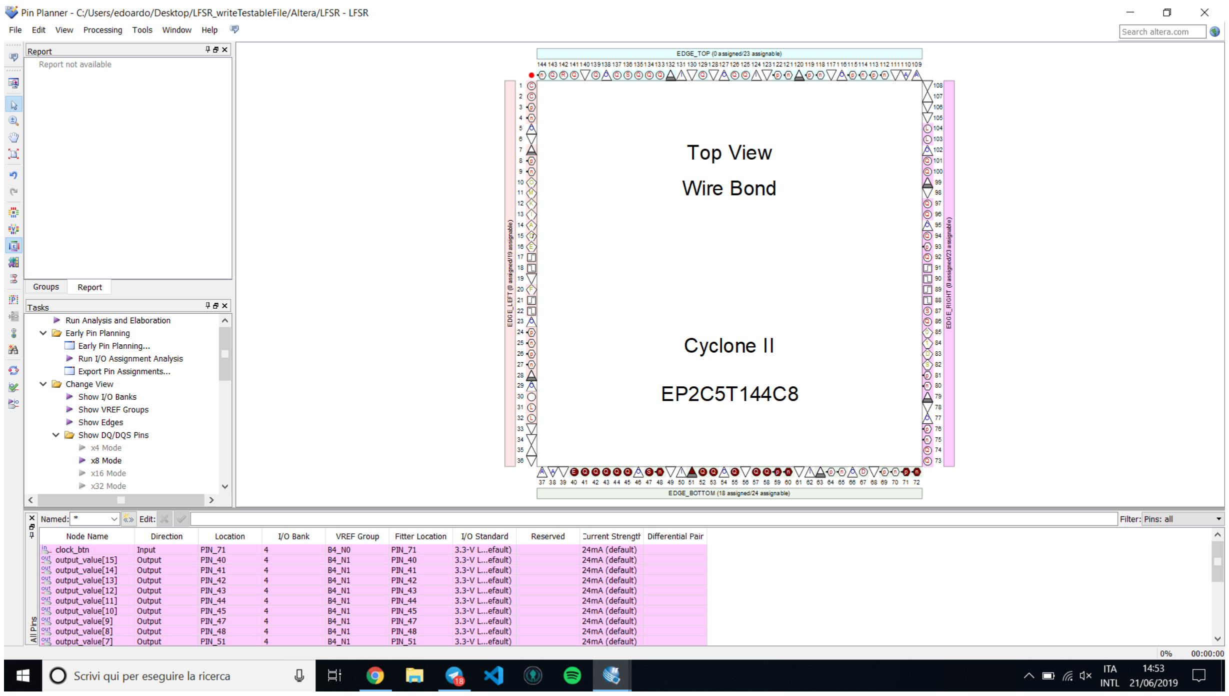
Quartus Prime enables analysis and synthesis of HDL designs, which enables the developer to compile their designs, perform timing analysis, examine RTL diagrams, simulate reactions of the DUT to different stimuli and configure the target device with the programmer.

Quartus Prime includes an implementation of VHDL and Verilog for hardware description, visual editing of logic circuits, and vector waveform simulation.

6.2 Pin assignment

After recompiling the files for the cyclone II FPGA we have to map physical pins with the desired signals.

The operation is the same that has been done for wiring high level entity (such testbench entity) with input and outputs of lower level component but with some additional limitations due to hard-wiring of pins with the real on-board FPGA.


Pin Planner - C:/Users/edoardo/Desktop/LFSR_writeTestableFile/Altera/LFSR - LFSR
Top View
Wire Bond
Cyclone II
EP2C5T144C8
EDGE_TOP
EDGE_BOTTOM (18 assigned/24 assignable)
Run Analysis and Elaboration
Early Pin Planning
Run I/O Assignment Analysis
Export Pin Assignments...
Change View
Show I/O Banks
Show VREF Groups
Show Edges
Show DQ/DQS Pins

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Differential Pair
clock_btn	Input	PIN_71	4	B4_N0	PIN_71	3.3-V L...efault)		24mA (default)	
output_value[15]	Output	PIN_40	4	B4_N1	PIN_40	3.3-V L...efault)		24mA (default)	
output_value[14]	Output	PIN_41	4	B4_N1	PIN_41	3.3-V L...efault)		24mA (default)	
output_value[13]	Output	PIN_42	4	B4_N1	PIN_42	3.3-V L...efault)		24mA (default)	
output_value[12]	Output	PIN_43	4	B4_N1	PIN_43	3.3-V L...efault)		24mA (default)	
output_value[11]	Output	PIN_44	4	B4_N1	PIN_44	3.3-V L...efault)		24mA (default)	
output_value[10]	Output	PIN_45	4	B4_N1	PIN_45	3.3-V L...efault)		24mA (default)	
output_value[9]	Output	PIN_47	4	B4_N1	PIN_47	3.3-V L...efault)		24mA (default)	
output_value[8]	Output	PIN_48	4	B4_N1	PIN_48	3.3-V L...efault)		24mA (default)	
output_value[7]	Output	PIN_51	4	B4_N1	PIN_51	3.3-V L...efault)		24mA (default)	

As shown in GUI pin were assigned, as reported in legend, on the EDGE_BOTTOM side of the chip so 18/24 pins on that edge were assigned.

Both reset and clock are connected to two push buttons that will drive signals from ”off-the-chip” user inputs.

6.3 Signals handling

Since as experiment is desirable to see, from the human point of view, the output of the register for each cycle, it was decided to drive clock signal via push button in order to make state transition a user driven decision that simulates the rising edge of the clock each time the button is pressed.

Seed was hard-coded in the design with hexadecimal value 0xACE1 that is the corresponding of 1010110011100001 in binary form.

Reset is driven as the clock with another push button, precisely the only built-in push button connected to pin 144 and assigned via the macro pin_144, and since it it active-low, every time the button is pressed the state of the reset signal changes from logic 1 to logic 0 and the output is brought back to the just told seed value.

The new entity is just like the other but with two input signals instead of three as reported below

```
ENTITY LFSR_16_BIT IS
             PORT(
                  clock_btn    : IN  STD_LOGIC;
                  reset        : IN  STD_LOGIC;
                  output_value : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
                );
END LFSR_16_BIT;
```

While in the architecture is added the constant

```
CONSTANT seed : STD_LOGIC_VECTOR(15 DOWNTO 0) := "1010110011100001";
```

7 Conclusion

From what has been written in this report it can be understood that the process of developing an electronic component, whatever it is, is quite complex and as such requires auxiliary development tools that simplify it and at the same time reduce the possibility of error.

Synthesized VHDL code can be considered to all intents and purposes as a hardware component that runs on the FPGA of which the behavior has been described.

Furthermore as hardware the synthesized design can be sent to any electronic component manufacturer to make the component in a chip as ASIC.

From this project it is possible to conclude saying that depending on the specific application for what a particular LFSR is meant, would be desirable to improve some different parameters analyzed during the synthesis process; for example if the application target is a cryptographic pseudo-random key generator (PRNG) it is important to have a high clock frequency, so the designer will try to reduce the time on the critical path in order to have higher frequencies.

There will never be a strictly better implementation than another but there will always be an optimal implementation in relation to the specific application.