
# UNIVERSITÀ DI PISA

Computer Engineering

Electronic Systems

## Linear Feedback Shift Register

Project Report

---

Biagio Cornacchia

Academic Year: 2021/2022

# Contents

# 1 Introduction

A **linear feedback shift register** (LFSR) is a type of shift register whose input bit is a linear function of its previous state. In particular, the input bit is driven by the XOR of some bits of the overall shift register value. The initial value is called the **seed** and it will affect the stream of bits generated by the LFSR. Since the operation of the register is **deterministic**, the stream of values produced is completely determined by its current state or previous state. Moreover, 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 which **appears random** and which has a very long cycle.

## 1.1 Algorithm Description

The general form of an LFSR of degree $m$ (an LFSR with m flip-flops) is shown in the figure below. It shows $m$ flip-flops and $m$ possible feedback location, all combined by the XOR operation. Whether a feedback path is active or not, is defined by the feedback coefficient $p_0, p_1, ..., p_{m-1}$. The list of bit positions that affects the next state is called the **tap sequence**. Thus, the taps are XOR'd sequentially and then fed back into the leftmost bit. The sequence of bits in the rightmost position is called the **output stream**. The values of the feedback coefficients are crucial for the output sequence produced by the LFSR.

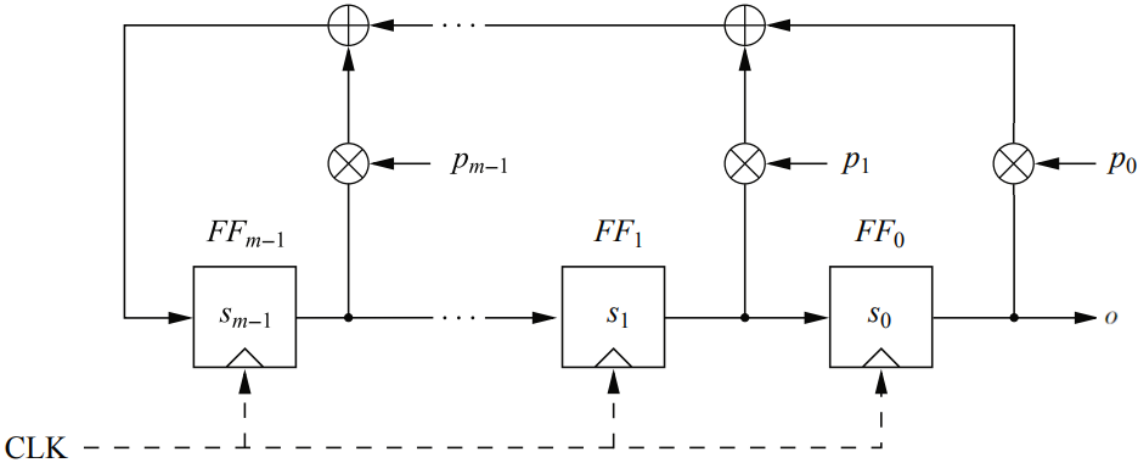



Figure 1: General LFSR with feedback coefficients $p_i$ and initial values $s_0, ..., s_{m-1}$

Let's assume the LFSR is initially loaded with the values $s_0, ..., s_{m-1}$. The next feedback bit of the LFSR $s_m$, which is also the input to the leftmost flip-flop, can be computed by the XOR-sum of the products of flip-flop **outputs** and the corresponding **feedback coefficients**:

$$s_m \equiv s_{m-1}p_{m-1} + \cdots + s_1p_1 + s_0p_0 \bmod 2$$

The next LFSR feedback bit can be computed as:

$$s_{m+1} \equiv s_mp_{m-1} + \cdots + s_2p_1 + s_1p_0 \bmod 2$$

In general, the output sequence can be described as:

$$s_{i+m} \equiv \sum_{j=0}^{m-1} p_j s_{i+j} \bmod 2 \qquad s_i, p_j \in \{0,1\} \quad i = 0,1,2,..$$

As said before, due to the finite number of recurring states, the output sequence of an LFSR will be repeated **periodically**. Moreover, an LFSR can produce output sequences of different length, depending on the feedback coefficients. The LFSR is **maximal-length** if and only if the corresponding feedback polynomial is **primitive**. This means that the following conditions are necessary (but not sufficient):

- The number of taps is **even**.
- The set of taps is **set-wise co-prime**; i.e. there must be no divisor other than 1 common to all taps.

The property of a maximum-length LFSR ensures that the maximum sequence length generated by an LFSR of degree $m$ is $2^m - 1$. Note that the all zero state must be excluded otherwise it will never be able to leave it again.

## 1.2 Possible Applications

LFSR is an elegant way of realizing long **pseudo-random sequences**. LFSRs are easily implemented in hardware which makes them useful in applications that require very **fast generation** of a pseudo-random sequence. For example they are adopted in:

- Electronics as **pattern generators** for exhaustive testing, since they cover all possible inputs for an n-input circuit.
- Cryptography as **pseudo-random number generators** because they can be easily and inexpensively implemented in hardware and can be mathematically analyzed if necessary. The use of LFSRs alone, however, is insufficient to provide good security due to their linearity. However, using a technique to remove **linearity** can improve the security of the resulting scheme.
- Communication systems such as **the Global Positioning System** (GPS) which uses LFSRs to rapidly transmit a sequence indicating relative instants with high precision, exploiting its determinism: in fact, it is enough to transmit the seed used in the transmitter and the generated sequence will also be identical at the receiver. Moreover, LFSRs are also used in radio jamming systems to generate pseudo-random noise to to disrupt a target communication system.

## 1.3 Possible Architectures

There are two type of possible LFSR's architectures:

- **Fibonacci LFSRs** also known as many-to-one architecture
- **Galois LFSRs** also known as modular, internal XORs, or one-to-many LFSR

### 1.3.1 Fibonacci LFSR

**Fibonacci LFSR** is the standard architecture used for LFSR. It works as the same described above. An example of Fibonacci LFSR architecture is the following:

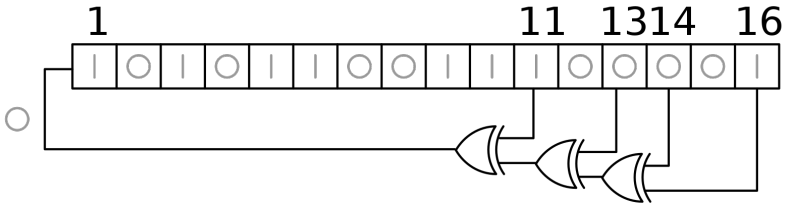



Figure 2: Fibonacci LFSR.

### 1.3.2 Galois LFSR

**Galois LFSR** is an alternative structure that can generate the same output stream as a conventional LFSR. In this configuration bits that are not taps are shifted one position to the right unchanged. The taps are XOR'd with the output bit before they are stored in the next position.

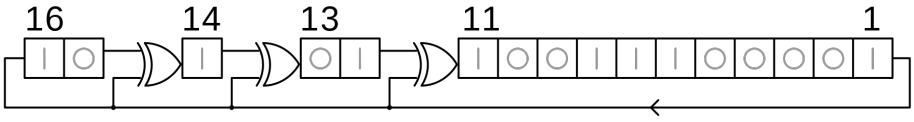



Figure 3: Galois LFSR.

## 1.4 Design Specifications

In the project assignment the LFSR has to implement the following feedback polynomial:

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

The architecture chosen for such circuit was the **Fibonacci** one. The circuit is characterized by:

- An input represented on 16 bits and called *INIT_VALUE* in order to initialize the digital circuit with an initial seed.
- An input bit called *SEED_LOAD* used to load the initial seed into the LFSR's starting state.
- An output bit that is the result produced by the LFSR algorithm.

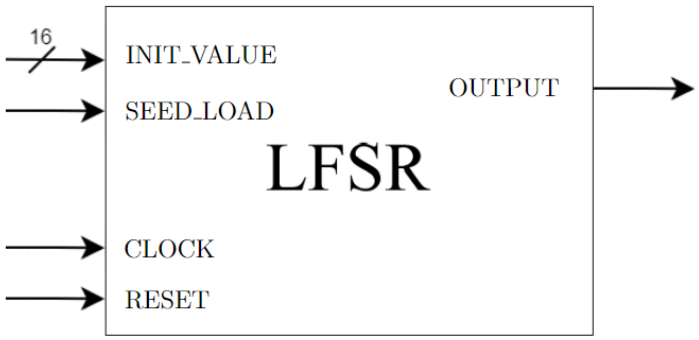



Figure 4: I/O Ports

# 2 Architecture Description

## 2.1 Block Diagram

The general structure of the LFSR can be summarized by the following block diagram:

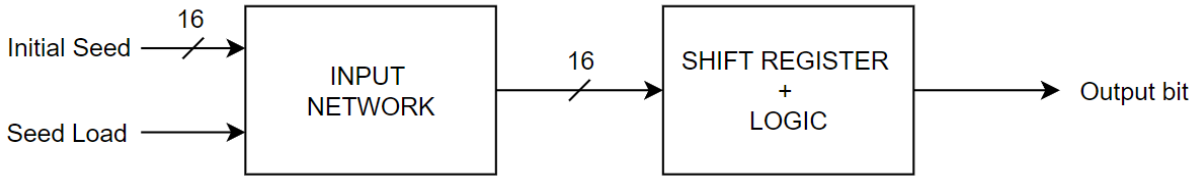



Figure 5: General Block Diagram.

## 2.2 Shift Register with Additional Logic

The rightmost part of the network is the core of the circuit. It is basically composed by a shift register with the addition of some logic. In particular, in order to allow the initialization of the shift register using an input initial value, a series of multiplexers were used. Each multiplexer is placed before each shift register's flip-flop and allows to choose the input to be given to each of them. A multiplexer is controlled by the **SEED_LOAD** bit:

- if $SEED\_LOAD = 1$ then each D flip-flop that composes the shift register is initialized using the $i\text{-}th$ bit of the inital value $s_i$.
- if $SEED\_LOAD = 0$ each D flip-flop uses as input data the output $q_{i-1}$ provided by the previous flip-flop whereas the first flip-flop will use the feedback bit generated by the network.

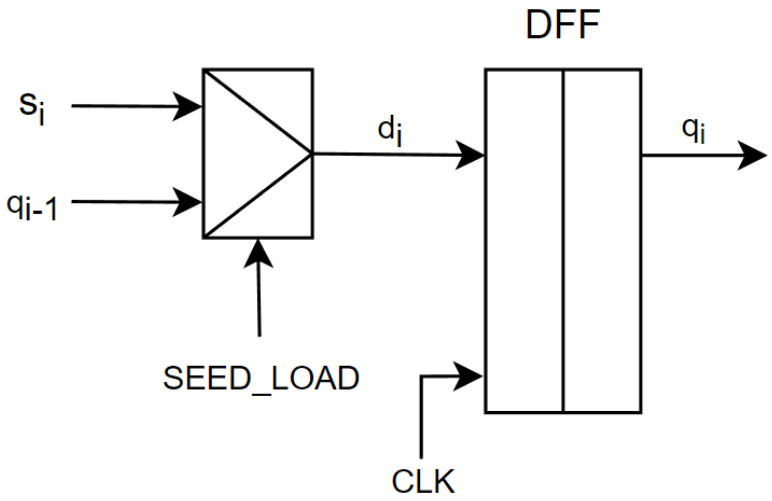



Figure 6: Example of Shift Register's flip-flop.

Thus, in order to implement the feedback polynomial required by the specifications, it

is necessary to implement a 16-bit LFSR using a 16-bit shift register and 16 one-bit controlled multiplexers linked together as described above.

Finally, an additional logic is needed for the generation of the feedback bit. As said before the feedback bit is obtained by XORing sequentially the taps and then fed back into the leftmost bit. In this case the taps are the $16^{th}$, $14^{th}$, $13^{th}$ and $11^{th}$ bit, so 3 XOR port are needed.

An example of the resulting network is shown below (for simplicity the clock and reset wire have been neglected and length of the shift register is 4-bit).

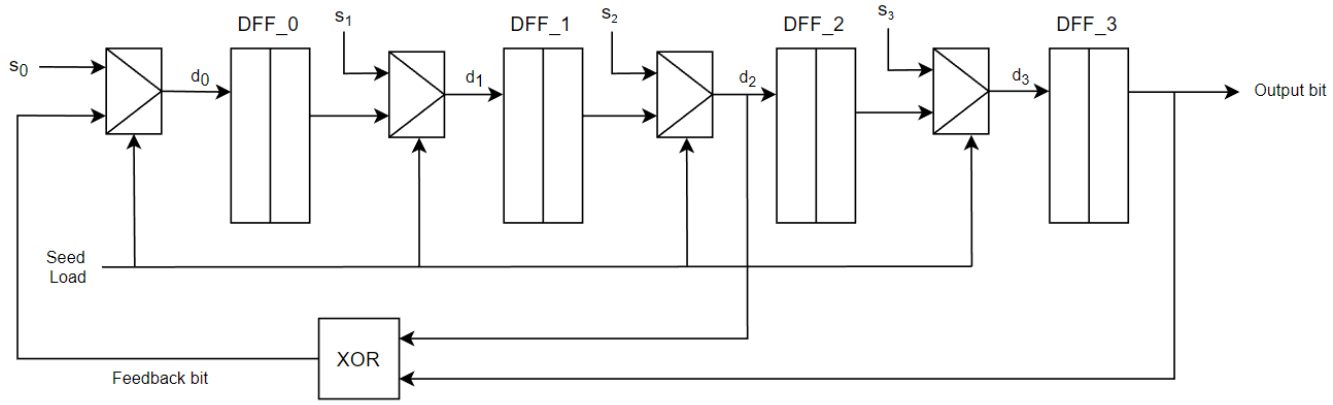



Figure 7: Example of LFSR with $1 + x + x^3$ as feedback polynomial and $s_0, s_1, s_2, s_3$ as initial seed.

## 2.3 Input Network

The **input network** is used to receive and store the initial seed and to receive the signal to actually load the seed into the shift register. In order to store the inputs, two registers have been used: a 16-bit register and a 1-bit register. The first register, called **INIT_VALUE**, takes as input the initial seed whereas the second register, called **SEED_LOAD**, is used to store the seed load bit. The network described above is the following:

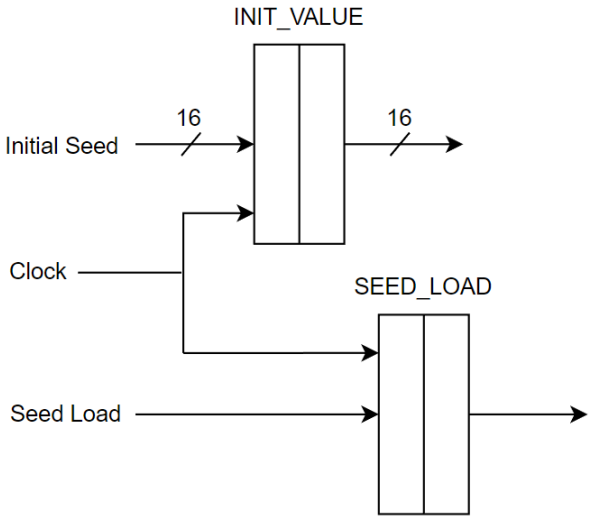



Figure 8: Input Network.

Each bit of the INIT_VALUE output will be connected to a multiplexer as shown before (i.e $s_0$ will be connected to the first one), whereas the SEED_LOAD output will be used as control line for each of them.

# 3 VHDL Code Analysis

In this section will be described the VHDL code for all the components described above. The analysis will be done using a **bottom-up** approach.

## 3.1 D Flip-Flop

The basic device needed for the implementation of the LFSR is a D flip-flop. It is characterized by an active low asynchronous reset.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  ----------------------
5  -- Entity declaration
6  ----------------------
7  entity DFF is
8      port(
9          clk     : in std_logic;
10         reset_n : in std_logic; -- async reset
11         d       : in std_logic;
12         q       : out std_logic
13     );
14
15 end DFF;
16
17 -----------------------------
18 -- Architecture declaration
19 -----------------------------
20 architecture beh of DFF is
21   begin
22     dff_proc: process(clk, reset_n)
23         begin
24             if(reset_n = '0') then
25                 q <= '0';
26             elsif(rising_edge(clk)) then
27                 q <= d;
28             end if;
29         end process;
30 end beh;
```

## 3.2 N-bit D Flip-Flop

Since it is necessary to store LFSR with an initial value on 16 bits, a **register** is needed. In particular, it can be implemented using a fixed number N of flip-flops such as those described above, with a common *clock* and *reset* terminal.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
```

```
4  ----------------------
5  -- Entity declaration
6  ----------------------
7  entity DFF_N is
8      generic(Nbit : positive := 16);
9      port(
10         clk         : in std_logic;
11         reset_n     : in std_logic; -- async reset
12         d_in        : in std_logic_vector(0 to Nbit-1);
13         q_out       : out std_logic_vector(0 to Nbit-1)
14     );
15 end DFF_N;
16
17 -----------------------------
18 -- Architecture declaration
19 -----------------------------
20 architecture beh of DFF_N is
21
22     component DFF
23         port (
24             clk     : in    std_logic;
25             reset_n : in    std_logic;
26             d       : in    std_logic;
27             q       : out   std_logic
28         );
29     end component DFF;
30
31 begin
32     GEN: for i in 0 to Nbit-1 generate
33         i_DFF: DFF port map(
34             clk         => clk,
35             reset_n     => reset_n,
36             d           => d_in(i),
37             q           => q_out(i)
38         );
39     end generate GEN;
40 end beh;
```

## 3.3 Linear Feedback Shift Register

This module will effectively realize the LFSR. As said before, the LFSR consists of a 16-bit shift register with additional logic (as described in Chapter 2) and an input network composed by the two registers mentioned before.

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  -------------------------
5  -- Entity declaration
```

```
6  -------------------------
7  entity LFSR is
8      generic (
9          Nbit    : positive  := 16
10     );
11     port (
12         clk         : in std_logic;
13         reset_n     : in std_logic;
14         seed        : in std_logic_vector(0 to Nbit-1);
15         seed_load   : in std_logic;
16         state       : out std_logic_vector(0 to Nbit-1); -- Used for testing
17         output_bit  : out std_logic
18     );
19 end LFSR;
```

The architecture contains the declaration of the D flip-flop used to realize the *N-bit shift register* and the *SEED_LOAD* register whereas the N-bit D flip-flop is used to implement the *INITIAL_VALUE* register.

```
1  -------------------------------
2  --  Architecture Declaration
3  -------------------------------
4  architecture rtl of LFSR is
5
6      component DFF
7          port (
8              clk     : in    std_logic;
9              reset_n : in    std_logic;
10             d       : in    std_logic;
11             q       : out   std_logic
12         );
13     end component DFF;
14
15     component DFF_N is
16         generic(Nbit : positive := 16);
17         port(
18             clk     : in std_logic;
19             reset_n : in std_logic;
20             d_in    : in std_logic_vector(0 to Nbit-1);
21             q_out   : out std_logic_vector(0 to Nbit-1)
22         );
23     end component DFF_N;
24
25     signal seed_load_s  : std_logic; -- Signal to control the MUXs
26     signal loaded_seed  : std_logic_vector (0 to Nbit-1);
27
28     signal feedback_bit : std_logic;
```

In the following part of the architecture it is possible to see the implementation of the **shift register**. In particular, it is formed by N properly connected flip-flops.

Moreover, in order to select the seed value as starting state for the shift register during the **initialization phase**, a multiplexer *MUX_1* controlled by the seed_load signal is used. The initialization phase starts when the **seed load** signal is **set to 1**.

```
signal d_in : std_logic_vector (0 to Nbit-1);
signal q_s  : std_logic_vector (0 to Nbit-2);
signal output_bit_s : std_logic;

begin

    init_reg : DFF_N
    generic map(Nbit => 16)
    port map(
        clk     =>  clk,
        reset_n =>  reset_n,
        d_in    =>  seed,
        q_out   =>  loaded_seed
    );

    seed_load_reg : DFF
    port map(
        clk     =>  clk,
        reset_n =>  reset_n,
        d       =>  seed_load,
        q       =>  seed_load_s
    );

    GEN: for i in 0 to Nbit-1 generate
        -- FIRST STAGE
        FIRST: if i = 0 generate
            FF0: DFF port map(
                d       => d_in(i),
                q       => q_s(i),
                reset_n => reset_n,
                clk     => clk
            );
        end generate FIRST;
        -- INTERNAL STAGES
        INTERNAL: if i > 0 and i < Nbit-1 generate
            FFI: DFF port map (
                d       => d_in(i),
                q       => q_s(i),
                reset_n => reset_n,
                clk     => clk
            );
        end generate INTERNAL;
        -- LAST STAGE
        LAST: if i = Nbit-1 generate
            FFN: DFF port map(
```

```
46                  d       => d_in(i),
47                  q       => output_bit_s,
48                  reset_n => reset_n,
49                  clk     => clk
50              );
51          end generate LAST;
52      end generate GEN;
53
54      MUX_1: process(seed_load_s, loaded_seed, q_s, feedback_bit, output_bit_s)
55      begin
56          case seed_load_s is
57              when '1' => d_in <= loaded_seed(0 to 15);
58              when '0' => d_in <= feedback_bit & q_s(0 to Nbit-2);
59              when others => d_in <= (others => '0');
60          end case;
61      end process;
62
63      -- the feedback bit is obtained XORing the taps
64      feedback_bit <= (output_bit_s xor q_s(13)) xor q_s(12) xor q_s(10);
65
66      state <= q_s & output_bit_s; -- Updating the current status
67      output_bit <= output_bit_s;
68
69  end rtl;
```

# 4 Test Plan

In this chapter will be tested the digital circuit in order to be sure that everything works properly. The tests performed are:

- Test of a **D Flip-Flop**.
- Test of an **N-bit D Flip-Flop**.
- Test of the **LFSR** with the help of a script written in *Python*.

## 4.1 D Flip-Flop Test

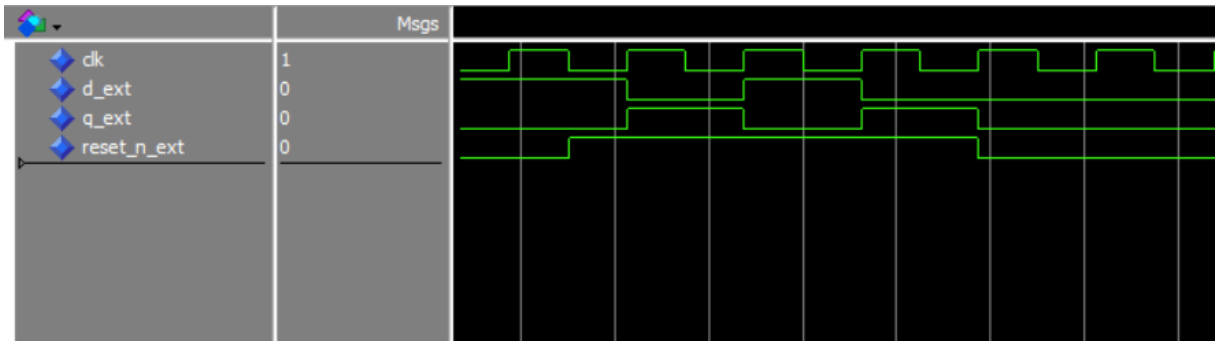



Figure 9: D Flip-Flop Test.

The test consists in testing the output value at the reset and the output changes after each clock cycle. As it possible to see in the figure, it works as expected:

- At the reset the output *q_ext* is equal to 0
- Then, after the reset, the output *q_ext* follows the input *d_ext* at every rising edge of the clock.

## 4.2 N-bit D Flip-Flop Test

As before, the test has to show that the n-bit output *q_out* follows input *d_in* at each clock cycle *clk*.

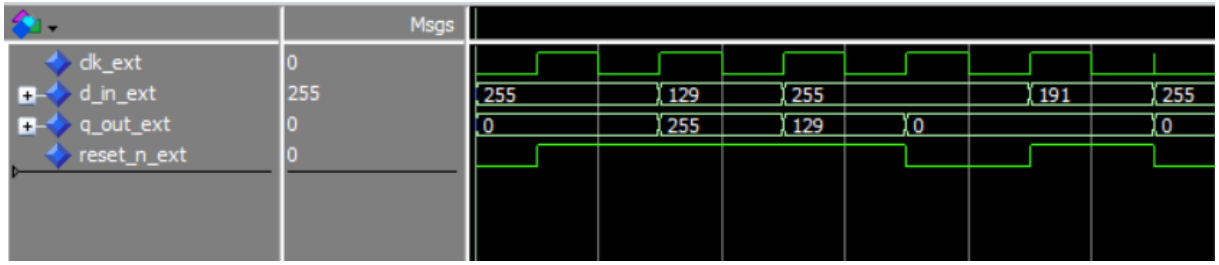



Figure 10: N-bit D Flip-Flop Test.

The circuit works as expected.

## 4.3 LFSR Test

The correctness of the LFSR has been verified with the help of a script written in Python that performs the algorithm in software. In particular, it has been used a standard VHDL package named *std_logic_textio* that provides the *readline* function and the *writeline* function. These functions have been exploited to create an output file called *LFSR_OUTPUT_STREAM.txt* containing the output stream generated by the circuit. Likewise, the Python script creates an output files used to perform a comparison between the two results.

### 4.3.1 LFSR Test-bench

The process code used to write the output file is the following:

```
file LFSR_OUTPUT : text is out "LFSR_OUTPUT_STREAM.txt";

stimuli: process(clk_tb, reset_n_tb)

variable bit_to_write : line;
variable t : integer := 0;

begin
    if(reset_n_tb = '0') then
        seed_tb <= "1000000000000001";
        seed_load_tb <= '1'; -- Start the initialization phase
        t := 0;
    elsif(rising_edge(clk_tb)) then
        seed_load_tb <= '0'; -- End the initialization phase

        -- When the current state is equal to the initial status
        -- it means that the LFSR will start generating
        -- the same output bits
        if(state_tb = seed_tb and t > 2) then
            end_sim <= '0';
        elsif(t >= 2) then
            -- After 2 clock cycles all the registers are ready
            -- then start collecting the output bits
            WRITE(bit_to_write, output_bit_tb);
            WRITEline(LFSR_OUTPUT, bit_to_write);
        end if;

        t := t + 1;
    end if;
end process;
```

A part of the circuit test is shown below:

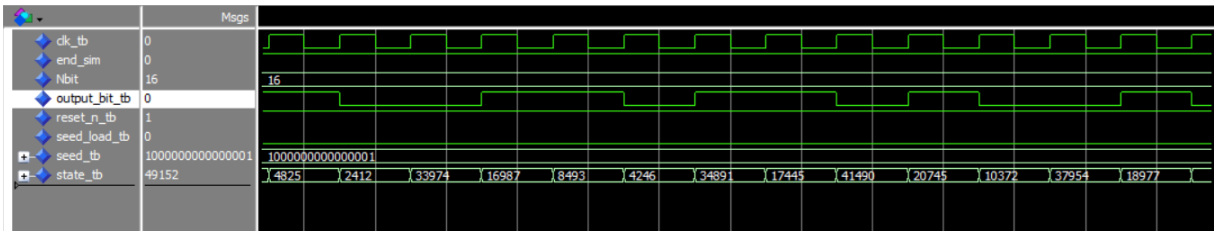



Figure 11: LFSR Test.

In the figure it is possible to see:

- the starting seed called *seed_tb*
- the output bit generated by the LFSR
- the current state called *state_tb*

It is possible to check the test using pen and paper but it is not very practical. For this reason, a script in python was used.

### 4.3.2 Python Test Program

As last step, a python script named **LFSR_python** has been created. This script emulates the LFSR algorithm in software. It can be configured with a **custom seed**. To do that, it is necessary to change the line *seed* in the python file:

```
1 seed = 0b1000000000000001
2 lfsr = seed
3 period = 0
4 outputstream = 0
```

In this case the seed used is 1000000000000001. Below there is an example of the output generated by the script:

```
+------------------------------------------------------+
|                    LFSR PYTHON                       |
+------------------------------------------------------+

[!] INITIAL SEED: 1000000000000001
[!] OUTPUT PERIOD: 65535

First 512 bits of the output stream:
00000000000000011000000000111011100001100111100101100011011110111001101000111010011001000100110001011110100101000000000010
11100100010000101101111100000100101110111010010011111100100011110111101100100110001010100100101010011000011001110011011 0
00100100100111011000000010100111100100111111011100111101011010110001101101111100110000101111000110000000100010111001011 1
10001011100001001000100100100011110000001100101011011100111011001101010110100011100110100100001001101001000101011010011 1
01011010111010111011101101101111

[+] OUTPUT STREAM IS CORRECT
```

The figure shows the initial seed and the output period of the LFSR. Then it shows the first 512 bits of the output stream in order to do a check through a visual analysis. However, the script performs a comparison between the entire output stream generated from the LFSR circuit and from the script, printing the result of the test. The output stream generated by the script is collected in a file called $LFSR\_OUTPUT\_STREAM\_PYTHON$.

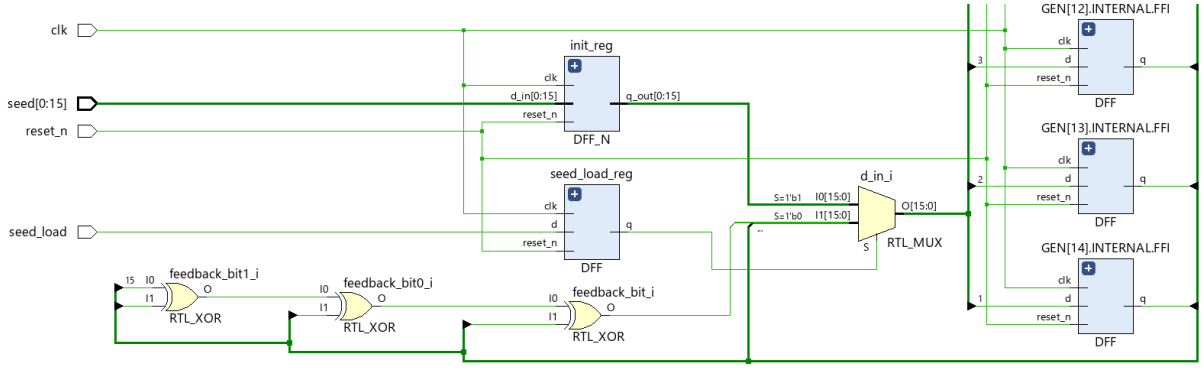
The circuit was tested using various combinations of seeds which result verified the correctness of the circuit.

# 5 Xilinx Vivado Report

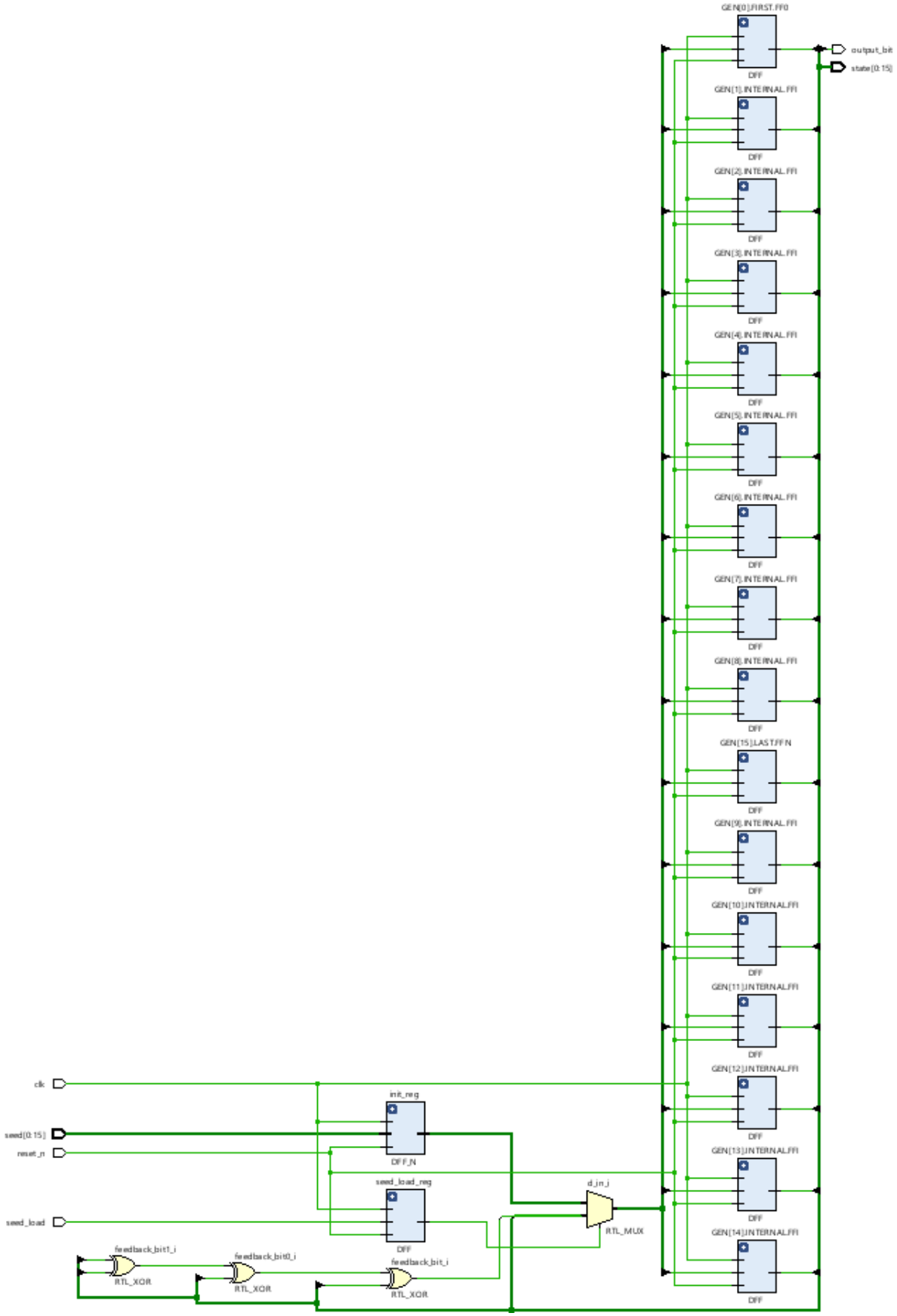
In this last chapter, will be presented the report obtained by creating a project with **Xilinx VIVADO** by selecting the **Zybo Zynq-7000** (*xc7z010clg400-1*) as working device.

## 5.1 RTL Analysis

The **elaborated design** that VIVADO uses to represent the circuit at the **Register Transfer Level** is the following:




The scheme above represents the first part of the network composed by the multiplexer, the SEED_LOAD register and the SEED_VALUE register. A complete overview is shown below.


GEN[0].FIRST_FF0
DFF
output_bit
state[0:15]
GEN[1].INTERNAL_FF1
GEN[2].INTERNAL_FF2
GEN[3].INTERNAL_FF3
GEN[4].INTERNAL_FF4
GEN[5].INTERNAL_FF5
GEN[6].INTERNAL_FF6
GEN[7].INTERNAL_FF7
GEN[8].INTERNAL_FF8
GEN[15].LAST_FF15
GEN[9].INTERNAL_FF9
GEN[10].INTERNAL_FF10
GEN[11].INTERNAL_FF11
GEN[12].INTERNAL_FF12
GEN[13].INTERNAL_FF13
GEN[14].INTERNAL_FF14
clk
seed[0:15]
reset_n
seed_load
init_reg
DFF_N
seed_load_reg
d_in_i
RTL_MUX
feedback_bit1_i
feedback_bit0_i
feedback_bit_i
RTL_XOR

## 5.2 Synthesis Analysis

Now it is possible to proceed with the **synthesis** of the circuit. This will be done adding a timing constraint. In particular, the clock period has been set to $8ns$. The synthesis result are the following:

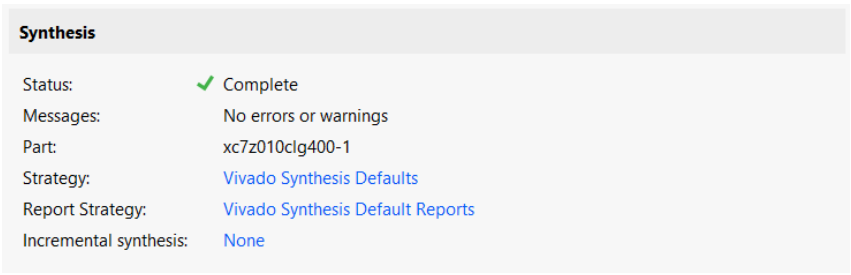



Figure 12: Synthesis result.

No errors or warnings found. So it is possible to analyze the **timing report summary**:

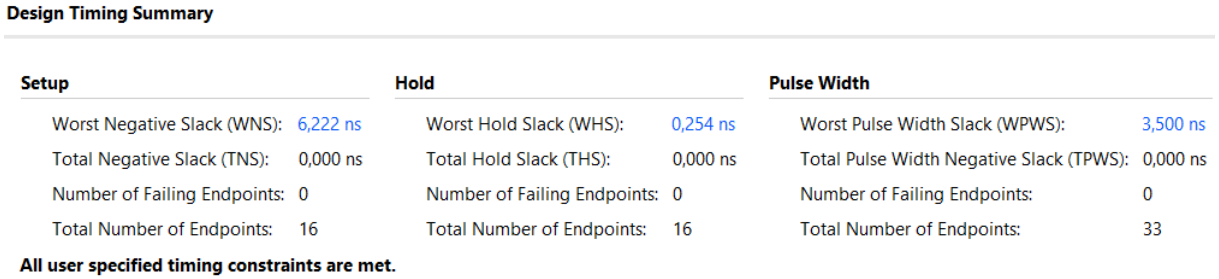



Figure 13: Timing Report Summary.

As we can see the **Worst Negative Slack** (WNS) is **positive**, so it is possible to drive the board at an higher frequency than 125MHz. However, the constraints are met because the Zybo Board operates with 125 Mhz.

It is possible to calculate the **maximum frequency** as:

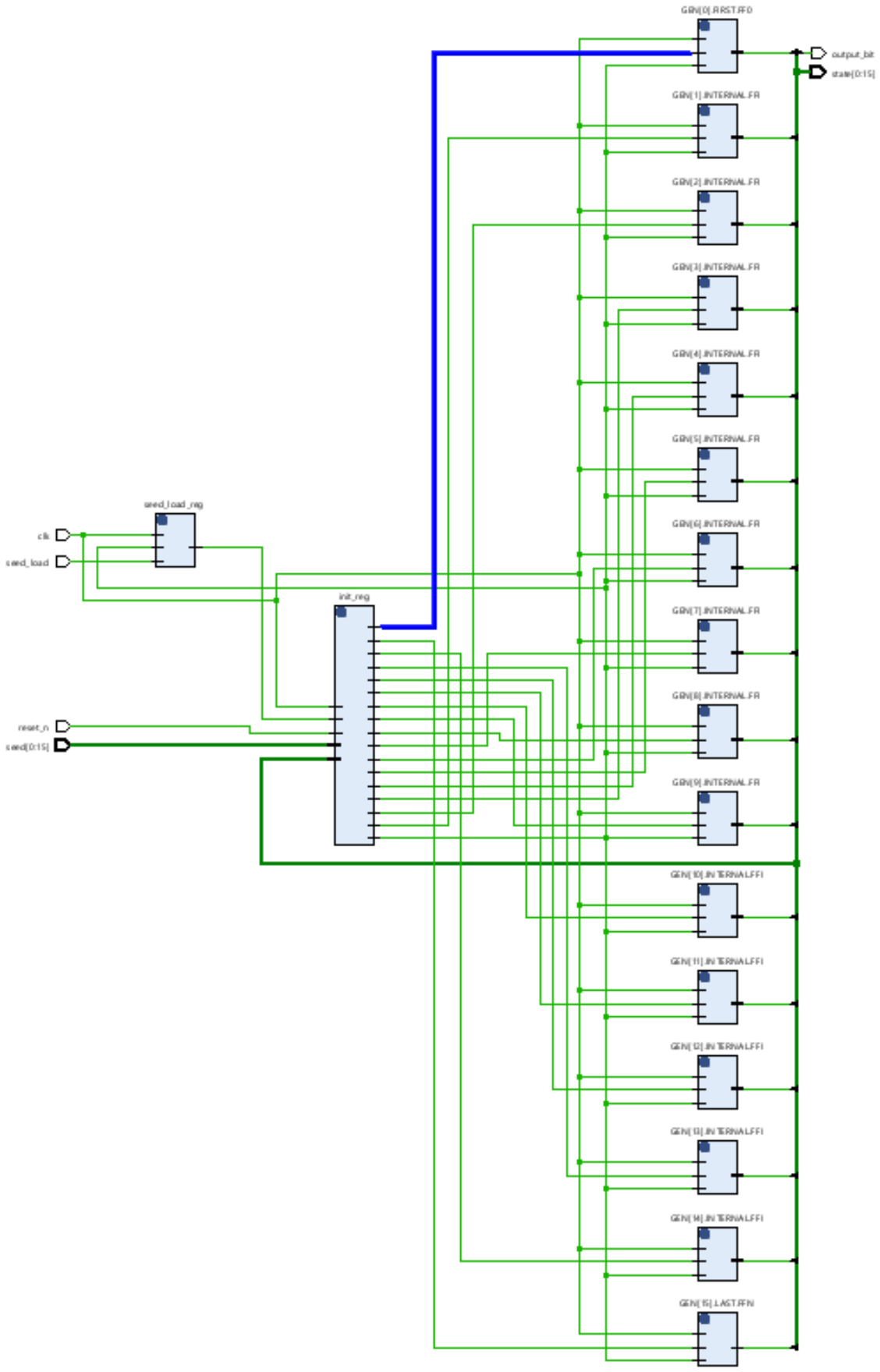
$$f_{clk} = \frac{1}{T_{clk} - WNS} = 562Mhz$$

where $T_{clk} = 1/125Mhz = 8ns$.

The WSN is determined by the **Critical Path** of the architecture which is shown in the following figure:

| Name | Slack | Levels | Routes | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | 6.222 | 1 | 2 | 1 | init_reg/GEN...DFF/q_reg/C | GEN[0].FIRST.FF0/q_reg/D | 1.756 | 0.791 | 0.965 | 8.0 | LFSR_clk |
| Path 2 | 6.446 | 1 | 2 | 1 | init_reg/GE...DFF/q_reg/C | GEN[10].INTE....FFI/q_reg/D | 1.532 | 0.791 | 0.741 | 8.0 | LFSR_clk |
| Path 3 | 6.446 | 1 | 2 | 1 | init_reg/GE...DFF/q_reg/C | GEN[11].INTE....FFI/q_reg/D | 1.532 | 0.791 | 0.741 | 8.0 | LFSR_clk |
| Path 4 | 6.446 | 1 | 2 | 1 | init_reg/GE...DFF/q_reg/C | GEN[12].INTE....FFI/q_reg/D | 1.532 | 0.791 | 0.741 | 8.0 | LFSR_clk |
| Path 5 | 6.446 | 1 | 2 | 1 | init_reg/GE...DFF/q_reg/C | GEN[13].INTE....FFI/q_reg/D | 1.532 | 0.791 | 0.741 | 8.0 | LFSR_clk |
| Path 6 | 6.446 | 1 | 2 | 1 | init_reg/GE...DFF/q_reg/C | GEN[14].INTE....FFI/q_reg/D | 1.532 | 0.791 | 0.741 | 8.0 | LFSR_clk |
| Path 7 | 6.446 | 1 | 2 | 1 | init_reg/GE...DFF/q_reg/C | GEN[15].LAST.FFN/q_reg/D | 1.532 | 0.791 | 0.741 | 8.0 | LFSR_clk |
| Path 8 | 6.446 | 1 | 2 | 1 | init_reg/GEN...DFF/q_reg/C | GEN[1].INTER...FFI/q_reg/D | 1.532 | 0.791 | 0.741 | 8.0 | LFSR_clk |
| Path 9 | 6.446 | 1 | 2 | 1 | init_reg/GEN...DFF/q_reg/C | GEN[2].INTER...FFI/q_reg/D | 1.532 | 0.791 | 0.741 | 8.0 | LFSR_clk |
| Path 10 | 6.446 | 1 | 2 | 1 | init_reg/GEN...DFF/q_reg/C | GEN[3].INTER...FFI/q_reg/D | 1.532 | 0.791 | 0.741 | 8.0 | LFSR_clk |

Figure 14: Timing List.


GEN[0].FIRST_FF
output_bit
state[0:15]
GEN[1].INTERNAL_FF
GEN[2].INTERNAL_FF
GEN[3].INTERNAL_FF
GEN[4].INTERNAL_FF
GEN[5].INTERNAL_FF
GEN[6].INTERNAL_FF
GEN[7].INTERNAL_FF
GEN[8].INTERNAL_FF
GEN[9].INTERNAL_FF
GEN[10].INTERNAL_FF
GEN[11].INTERNAL_FF
GEN[12].INTERNAL_FF
GEN[13].INTERNAL_FF
GEN[14].INTERNAL_FF
GEN[15].LAST_FF
seed_load_reg
clk
seed_load
init_reg
reset_n
seed[0:15]

### 5.2.1 Utilization Analysis

The utilization report obtained are as follows:

**Utilization** **Post-Synthesis** | Post-Implementation

Graph | **Table**

| Resource | Estimation | Available | Utilization % |
|---|---|---|---|
| LUT | 17 | 17600 | 0.10 |
| FF | 33 | 35200 | 0.09 |

Figure 15: Utilization Report.

The utilization for FPGA resources is lower than the 1%, so there will be no problem in implementing this circuit.

### 5.2.2 Power Consumption Analysis

The last report is about the power consumption which gives a rough idea of the circuit's power requirements.

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

| | |
|---|---|
| **Total On-Chip Power:** | **0.09 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26,0°C** |
| Thermal Margin: | 59,0°C (5,0 W) |
| Effective ϑJA: | 11,5°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | High |

Launch Power Constraint Advisor to find and fix invalid switching activity

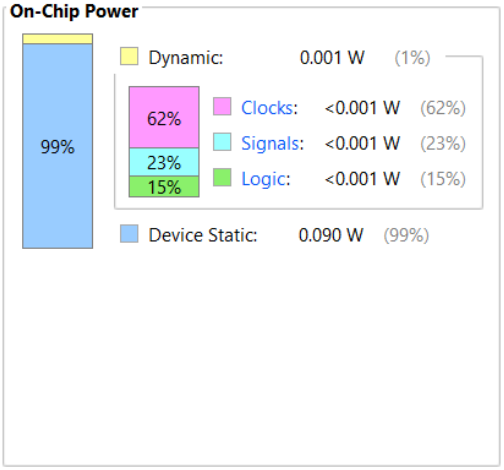



Figure 16: Power Consumption Report.

As we can see, a total of 0.09W of power is needed, which is divided by 1% into dynamic power and 99% into static power.

## 5.3 Implementation Analysis

In this phase the **place and route** will be performed by VIVADO, with some optimizations. The implementation will be performed in **Out of Context Mode** which allows to do the implementation without **I/O Planning**. The implementation result is the following:

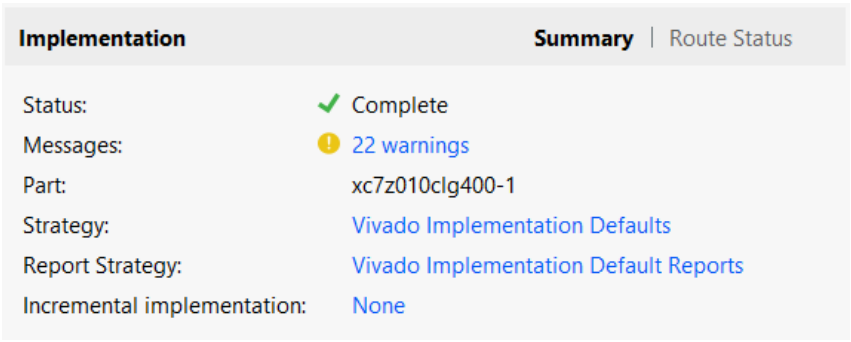



Figure 17: Implementation Summary.

In this case there are no errors but warnings. They will be analyzed later.

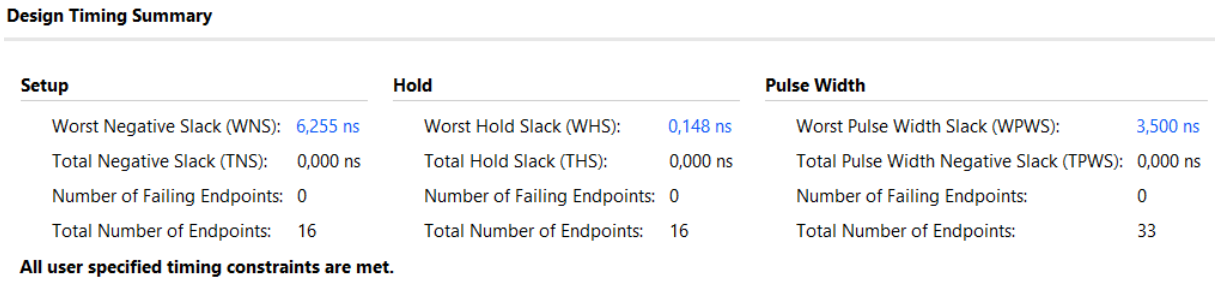
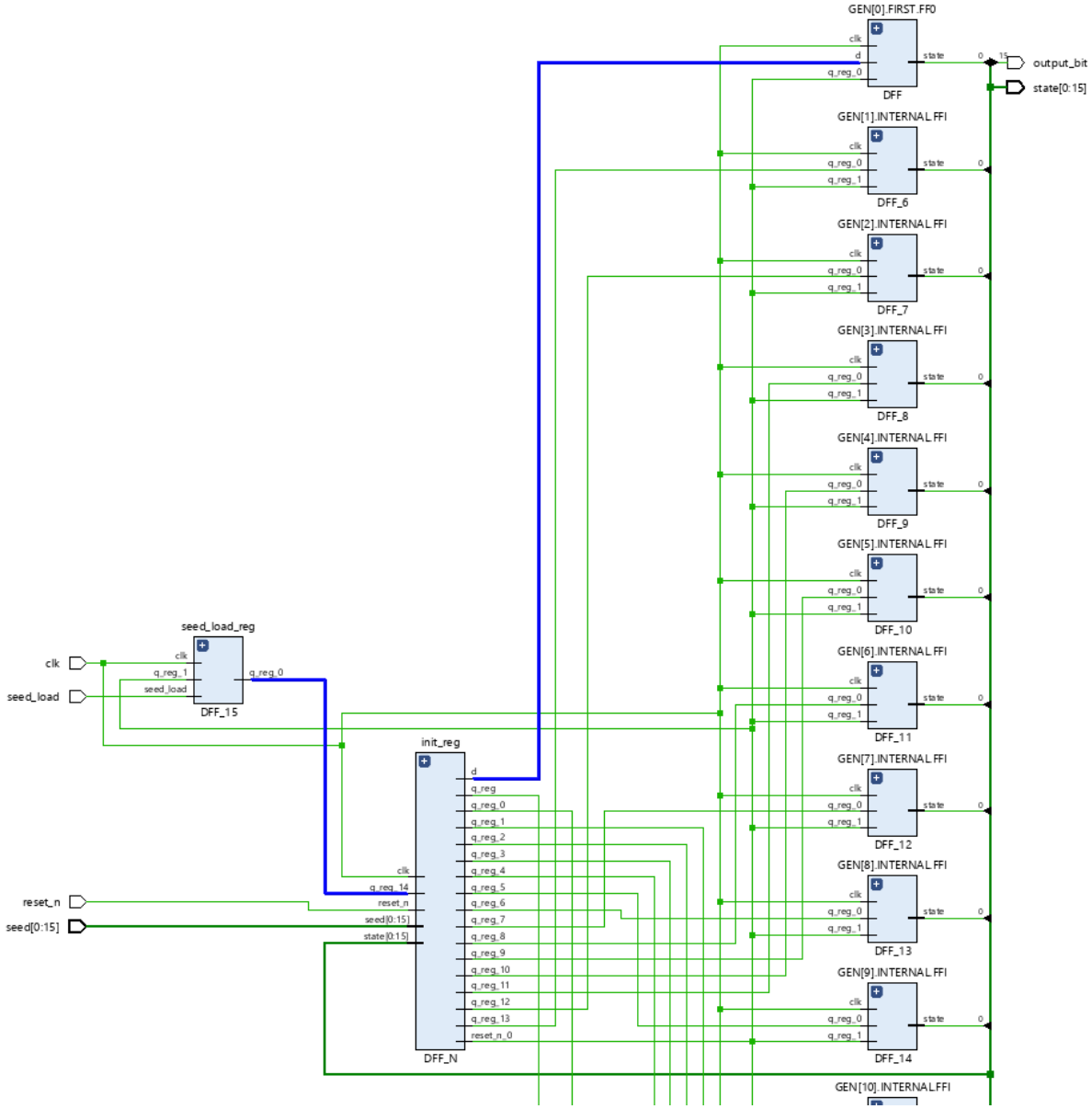



Figure 18: Implementation Timing Summary.

It is possible to see how the negative slack has changed slightly but the constraints are still met. This change is due to the fact that the implementation give a better estimation for the timing.

The critical path is the following:




### 5.3.1 Utilization Analysis

The utilization report is the following:

**Utilization** Post-Synthesis | **Post-Implementation**

Graph | **Table**

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 17 | 17600 | 0.10 |
| FF | 33 | 35200 | 0.09 |

Figure 19: Utilization Report.

There are no changes from the synthesis report.

### 5.3.2 Power Consumption Analysis

The power consumption report is the following:

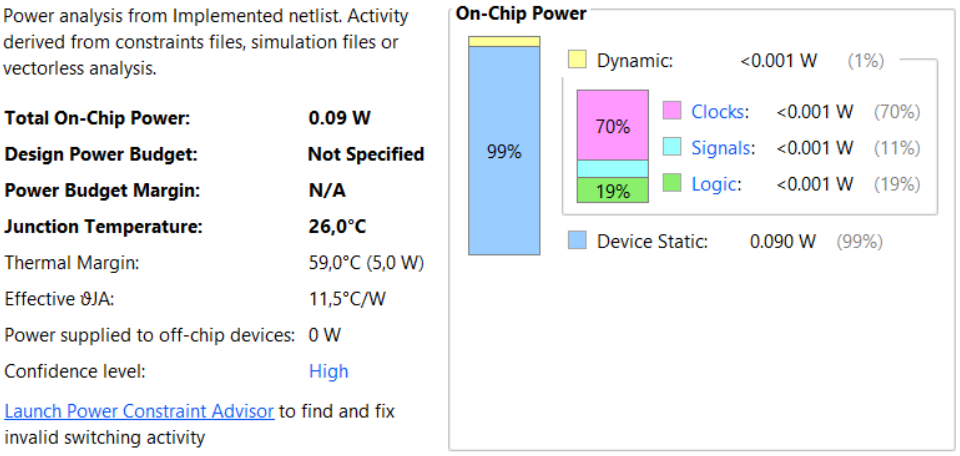
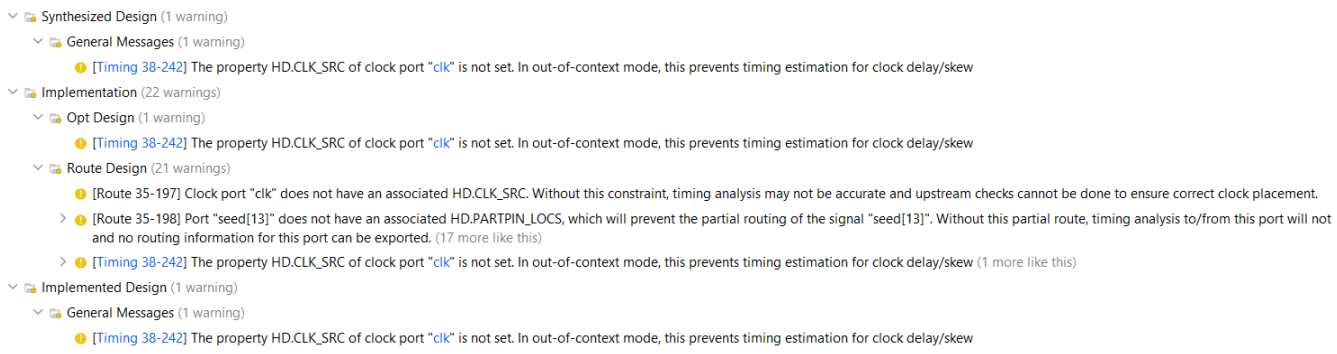



Figure 20: Power Consumption Report.

There are no significant changes from the synthesis report.

### 5.3.3 Warning Analysis

As said before, during the implementation phase some warnings came out:




These warnings are related to the fact that the *clk signal* is not linked to anything. In this way Vivado cannot provide a complete analysis of the circuit. They can be labeled as **internal vivado warnings** and they will disappear in a real scenario.

# 6 Conclusions

The result of this project is a Linear Feedback Shift Register implemented programming the ZyBo Board. The LFSR is characterized by the following feedback polynomial:

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

The architecture chosen is Fibonacci ones. From the viewpoint of the performance, the device implemented on the ZyBo meets the clock constraint (8ns) and can work with an higher clock frequency. The device has a power equal to 0,09 W. The utilized resources are very low and they reach a great value only for the I/O resources.