University of Pisa

M.Sc IN COMPUTER ENGINEERING

"LINEAR FEEDBACK SHIFT REGISTER"

AUTHOR

Francesco Iemma

Academic Year: 2020/2021

Contents

Chapter 1

Introduction

The Linear Feedback Shift Register (LFSR) is a shift register [1] in which some bits are manipulated and fed back to the input in order to generate, thanks to the shifting, a sequence of bits. So we can say that its input is the output of a linear function of its previous state.
The initial value of the LFSR is the so-called seed and it affects the stream of bits generated by the LFSR. However, this stream is not (obviusly) really random because the number of possible states is finite and the stream of bits is determined by its current or previous state and, moreover, it is possible to compute the bit generated knowing the seed and the feedback function. Anyway if the feeback function is well-choosen then the LFSR can produce a sequence of bits that appears random and which has a very long cycle, so this means that the output starts to repeat the generated numbers after a very long time. So we can assume, at least within a cycle, that it generates random numbers. For this reason the sequences generated from an LFSR are called pseudo-random.

1.1 Algorythm Description

In order to understand the LFSR algorythm we need to discuss about the *feedback function*: it is the function which describes the way in which some spefic bits of the shift register are manipulated and then fed back to the input. Usually the arrangment of the inputs of the feedback function is represented as polynomial mod 2. In our case the feedback polynomial is:

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

Analysing the polynomial we can understand how LFSR works. In particular the feedback polynomial indicates which bits of the shift register affects the next state, these are called *taps*. In most cases the way in which the taps affect the next state is using XOR gates (but also XNOR gates can be used).
Now let's assume that we have an LFSR with a length of N and see how it works. At the starting state in the LFSR we have the seed. Then the feedback logic starts to work and the bit released from this logic is the new 0^{th} bit and the others bits are shifted by one to the right and due to the shift the previous last bit, i.e. the $(N-1)^{th}$ which has become[2] the N^{th}, will be the output of the LFSR.
In a more formal way (let's assume b'_i the i-th old bit, b_i the i-th new bit, N the length of LFSR with $N > 0$, T the set which contains the indexes of the taps and y the function which represents the feedback logic) I can say:

- $b_0 = y(b'_j, ..., b'_k)$ where $j, ..., k \in T$
- $b_i = b'_{i-1}$ for $1 \leq i < N$
- $output = b'_{N-1}$

[1] A shift register is a sequential logic circuit made up of chain of flip flops, for each rising edge of the clock the value of each flip-flop is shifted to the next flip-flop

[2] with N^{th} we indicate the bit which is out of the LFSR. A shift register with a length of N has N bits and so its bits are the ones in the interval [0, N-1], the N^{th} doesn't exist but it's a way to indicate the "overflow" bit.

The taps in our case are the bits 11, 13, 14, 16 (we can recognize them gazing to the exponents of x in the feedback polynomial) so this means that the inputs of the feedback logic are these. And so in our previous notation (considering that we start to count from 0) we will have $T = \{10, 12, 13, 15\}$.
In the following image it's possible to see a logic representation of a generic LFSR.

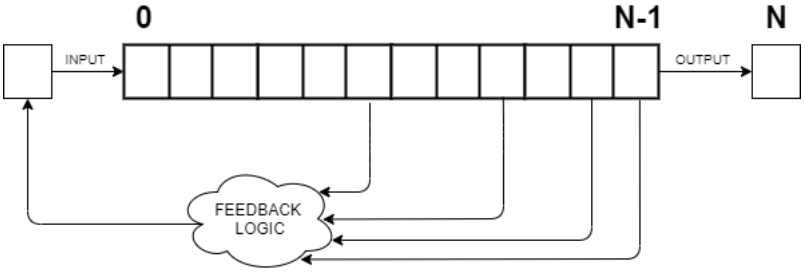

Figure 1.1: Logic representation of a LFSR

1.2 LFSR's Properties

Now let focus on a particular type of LFSR which has an interesting property, this is the so-called maximum-length LFSR. An LFSR is a maximum-length one, if and only if, the feedback polynomial is primitive and so it's necessary that:

- The number of taps is even
- The taps values are coprime

The property of a maximum-length LFSR is the following: if its length is N then it will assume all the possible $2^N - 1$ states (the state with all 0 is excluded because otherwise it will stop). Others properties for generic LFSRs are:

- As we can saw previously, the output is deterministic, in fact it's possible to compute it starting to the current state
- The state with all zeroes is not allowed, for this reason even a maximum-length LFSR cannot generate a sequence of 2^N states

It's important to point out that even if the output is deterministic the LFSRs are spreadly used, this happens because some techniques can be adopted to resolve this issue, for example giving an irregular clock to the device or manipulating the output stream with some non-linear combination of two or more bits extracted from this stream.

1.3 Possible Applications

LFSRs are spreadly used in a lot of fields. Let's see the most important ones:

- They are used in Criptograhpy as pseudo-random number generators for stream-chipers (especially in military applications). In this field it's very important to resolve the issue about the deterministic behaviour of the LFSR in order to avoid possible attacks, some possible strategies are the ones seen in the previous section.
- They are used in Communication, for example in the Global Position System (GPS) and also in radio-jamming in order to generate a pseudo-random noise. Then another application in this field is for scrabbling, the bits produced from an LFSR are combined with the data bit in order to have a convenient stream of data that ensures good properties and better performance from the viewpoint of the modulator/demodulator. Then it is also used in CRC (Cyclic Redundancy Check).
- They are used in Electronics in order to do a pseudo-random testing of a circuit in order to test "all" possibile inputs of the circuit (in fact this technique is also called pseudo-exhaustive testing)

1.4 Possible Architectures

There are two type of possible LFSR's architectures:

- Fibonacci LFSR (also called "Many-To-One architecture", because from "many" bits you obtain "one" bit that is the new input bit)
- Galois LFSR (also called "One-To-Many architecture", because from "one" bit - that is the last one and also the output - you obtain "many" bits that are the ones which are in positions next to the taps)

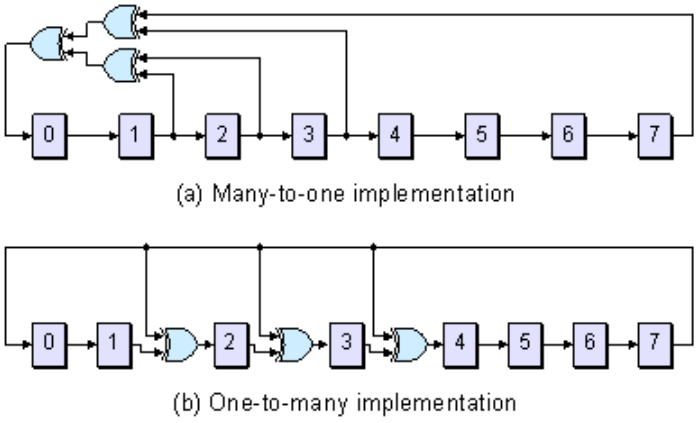

Figure 1.2: LFSR Architectures - Source: www.eetimes.com

1.4.1 Fibonacci LFSR

The Fibonacci LFSR is the simplier possible architecture for an LFSR and it is also the standard one, basically the generical things said so far about LFSRs describe the Fibonacci LFSR. In fact, for instance, we can observe that the figure 1.2.(a) and the figure 1.1 are indeed the same figure with the difference that the 1.1 is a logic representation and so the gates have not been drawn (they are in the so-called "Feedback Logic").

1.4.2 Galois LFSR

The Galois LFSR is an alternative architecture for LFSR, from the point of view of the output it's, more or less[3], equivalent to Fibonacci one (the output of a Galois it's the same of a Fibonacci) but indeed they are internally different. On the rising edge of the clock, all bits that are not taps are shifted unchanged to the right by one position. Instead the taps are XORed with the previous last bit and the output of this XOR operation goes to the next bit on the right. In a more formal way we can write (let's assume b'_i the i-th old bit, b_i the new i-th bit, N the length of LFSR with $N > 0$, T the set which contains the indexes of the taps):

- $b_0 = b'_{N-1}$
- $b_{i+1} = b'_i$ with $i \notin T$, $0 \leq i < N-1$
- $b_{j+1} = XOR(b'_{N-1}, b_j)$ with $j \in T$, $0 \leq j < N-1$
- $output = b'_{N-1}$

[3] A time offset is present and if some techniques are not adopted (the bit positions must be counted in the opposite direction of the shift) then one output is the reverse of the other one. In our case we neglect this difference because in any case the Galois is indeed a LFSR

1.4.3 Comparison

The Galois architecture has a big advantage with respect to Fibonacci one. In fact in the latter there is a chain of XOR gates and the XOR operations must be done in serial and it cannot be done in parallel. Furthermore, the propagation time of this chain is not negligible. On the other hand, in the Galois implementation, the XOR operations can be done in parallel and the propagation time regard a single gate (not a chain as in the Fibonacci's case). So the propagation time of Galois implementation is lesser than the Fibonacci one, this means that with this architecture we can achieve a faster clock frequency. In our case we have four taps, so if we assume that we have only XOR gates with two inputs and each one has a $t_{propagation_i} = x$ we are in the following situation:

- Fibonacci: we have to use three XOR gates (installed as we can see in the following image) and so the minimum clock period T is equal to $T = t_{propagation_1} + t_{propagation_2} = 2x$. Then the maximum frequency achievable is $f = 1/T = 1/2x$. If $x = 4ns$ then $f = 125MHz$

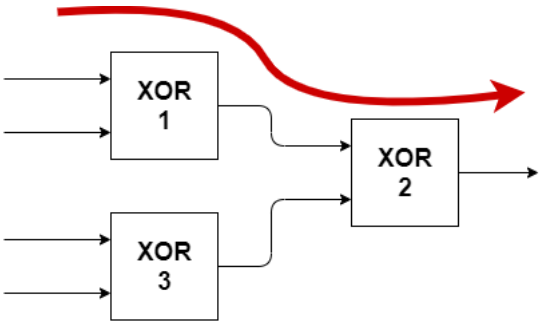

Figure 1.3: Critical path (in red) related to XOR gate in Fibonacci with 4 taps

- Galois: also in this case we use three XOR gates, but in this case each one is installed between two flip-flop so the critical path is made up of one gate. Then $T = t_{propagation_1} = x$ and so $f = 1/T = 1/x$. If $x = 4ns$ then $f = 250MHz$.

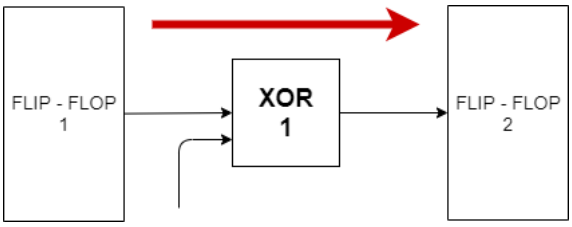

Figure 1.4: Critical path (in red) related to XOR gate in Galois

We can see that in this case $f_{Galois} = 2f_{Fibonacci}$ but if the number of taps increase we have a larger difference, for example with $NumTaps = 8$ I have $f_{Galois} = 3f_{Fibonacci}$, with $NumTaps = 16$ then $f_{Galois} = 4f_{Fibonacci}$. So if $NumTaps$ is 2^a then $f_{Galois} = af_{Fibonacci}$.
This results shown as the Galois implementation is better from the viewpoint of the maximum clock frequency.
This difference is bigger when the number of taps grows, but in most applications this difference is not so evident as in the example shown and, at the end of the day, the performance are more or less the same. For example if we use XOR gate with more inputs than two, the $t_{propagation}$ of Fibonacci is less than the one in the case of a cascade of XOR with two inputs. So the relation with Galois is not so unbalanced. Then in some FPGA are used LUTs that support 6 input logic function, so using a XOR with two inputs or a XOR with six is basically the same.
Indeed, theoretically speaking, the difference in performance between the two implementations exists, and so in the following report it will be treaten the implementation in VHDL of an LFSR with Galois architecture and then its testing and its practical realization obtained by programming the ZyBo Board with Xilinx Vivado.

Chapter 2

Architecture Description

As we discussed in the previous chapter, we will implement a Galois LFSR. Let's see the necessary devices to implement it. In order to implement a memory element I need a D Flip-Flop with set and reset. In fact we need to initialize each memory element to the seed value, so the i-th flip flop will be initialize with the i-th bit of the seed. The aim of the implementation is to have a functional LFSR, but another aim is also having a customizable LFSR. In order to do this we have to add a circuit to change (or not) the input of a simple flip-flop, we call this circuit *tap circuit*. It will allow to set the taps, changing a proper input of the LFSR.

2.1 D Flip-Flop with tap circuit

Assume that we have the i-th flip-flop and the j-th (with $j = i+1$). If i is a tap then its output must be XORed with the $N-1$ bit, otherwise the output of i goes unchanged to the input of j. So we have to implement the following logic function (described using a pseudo-code and assuming that $feedback$ is the $N-1$ bit):

```
1  void logicFunctionOmega(isTap_i, feedback)
2  {
3    if(isTap_i==true)
4      input_j = xor(output_i,feedback);
5    else
6      input_j = output_i;
7  }
```

Now we have to translate this logic function from this pseudo-code to logic gates. But, before doing this, we need to point out some concepts:

- $isTap_i$, $feedback$ and $output_i$ are bits. We will use the following convention: $isTap_i$ is true if it is 1, it is false when it is 0.
- We know that the neutral element for the XOR is the 0 (A and B are the inputs of XOR: if B is 0 then A passes the gate unchanged), in fact the truth table for the XOR is:

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2.1: XOR Truth Table

A	B	AΩB
0	0	0
0	1	0
1	0	0
1	1	1

Figure 2.2: Ω Truth Table. A is $isTap_i$ and B is $feedback$

After fixed these points we can implement our function, let's call it Ω and let see what is the expected truth table (see figure 2.2).
Remember that the output must be equal to B ($feedback$) if A(tap_i) is equal to 1, otherwise it must be equal to the neutral element of the xor, so 0.
We can easily recognize that the "mysterious" logic function Ω is indeed the AND gate. So the logic block diagram of a flip-flop with the tap circuit is the following:

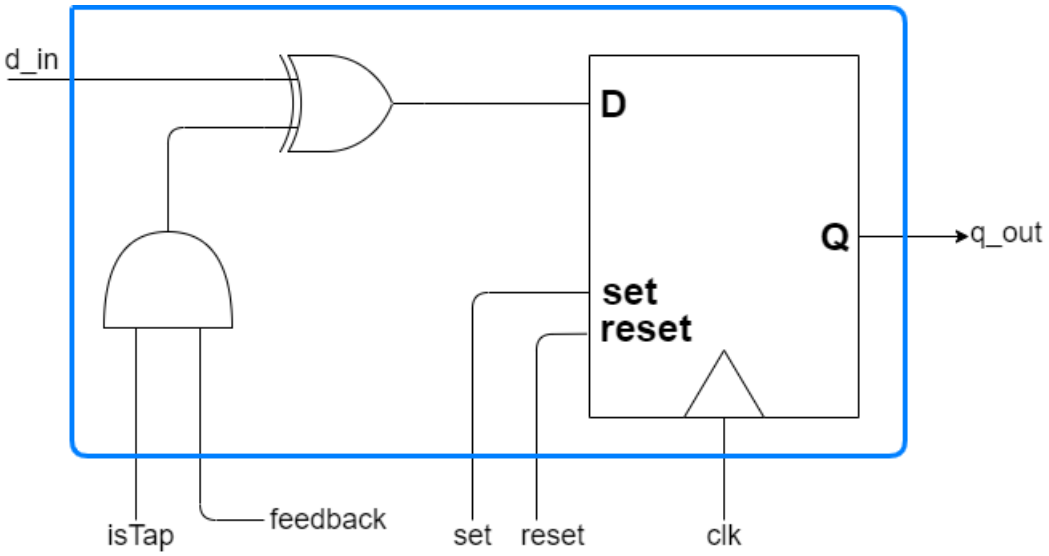

Figure 2.3: Logic block diagram of a D Flip-Flop with set/reset and tap circuit.

2.2 LFSR Implementation

Once we have the D Flip-Flop with the tap circuit if we want to implement a LFSR with a length of N, we need N D Flip-Flop with tap circuit.

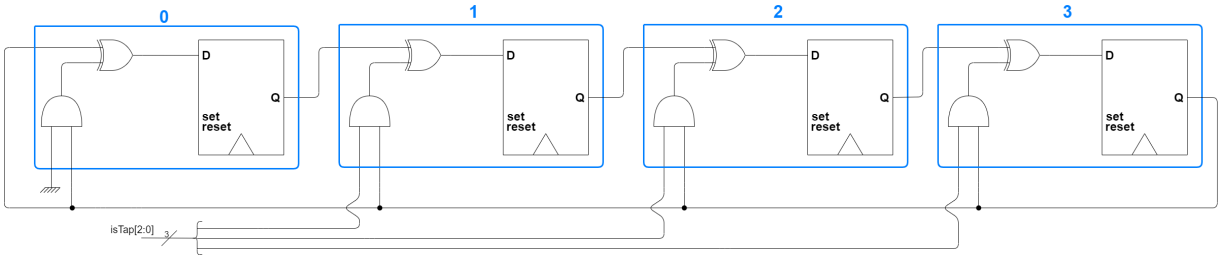

Figure 2.4: Implementation of a LFSR (length of 4) with D Flip-Flop with tap circuit. (For simplicity we neglect the links for clock and set/reset)

It's important to do some observations about this implementation.

- The first flip flop (the 0^{th}) is a particular one. Its input port "isTap" must be linked to the ground (it must be always 0) because doesn't exist a flip flop before the zero one.
- The initialization of the flip-flops is ensured by the basic D Flip-Flop linking the set port of the i^{th} flip-flop to $seed[i]$ (this is not shown in figure 2.4)

2.3 Input/Output and Overall

Abstracting from the internal details it's possible to describe the LFSR as in the image 2.5.

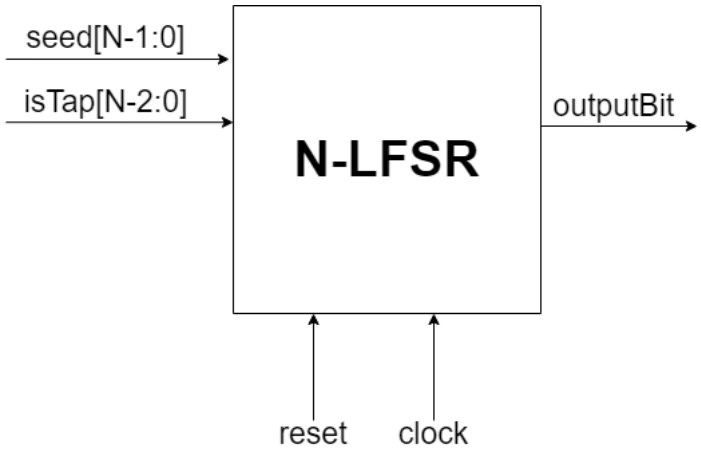

Figure 2.5: Logic block diagram of a LFSR with a length of N.

Let's analyse the input/output ports:

- *seed[N-1:0]*: it contains the initialization value of the LFSR (the so-called seed). As we said previously the $i-th$ set port of the flip-flop will be connected to $seed[i]$.
- *isTap[N-2:0]*: it indicates which are the taps. Its length is N-1 because there is no flip-flop before the 0^{th} flip-flop and so its isTap port will be connected to the ground (zero).
- *reset*: the reset in common for all flip-flops.
- *clock*: the clock in common for all flip-flops.
- *outputBit*: the bit that is the result of the LFSR's algorithm.

Let's sum up the operations done by the LFSR:
The LFSR is programmable, because once we have fixed the length N it can works with any polynomial feedback because of *isTap[N-2:0]*. Then it's also possible to initialize, at the reset, the shift register using *seed[N-1:0]*. Then, when all these signal are given to the device, it start to work and to produce a pseudo-random sequence of bits.

Chapter 3

VHDL Code

In this section we will see the VHDL code for all the devices seen previously: starting from the basic ones, that are necessary to implement an LFSR, to arrive at the LFSR itself.

3.1 D Flip-Flop with Set/Reset

The basic device needed for the LFSR is a D Flip-Flop wit Set/Reset.

```
1  -- D Flip-Flop with Set/Reset
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  entity dff is
6    port(
7      clk   : in  std_logic;
8      reset : in  std_logic;
9      set   : in  std_logic;
10     d     : in  std_logic;
11     q     : out std_logic
12   );
13 end dff;
14
15 architecture rtl of dff is
16 begin
17   dff_p:process(reset,clk)
18   begin
19     if reset='0' then
20       q <= set;
21     elsif(rising_edge(clk)) then
22       q <= d;
23     end if;
24   end process;
25 end rtl;
```

The behaviour of this device is very simple, at the initialization phase (when reset is 0) the output is initialized with the set value. Then in the normal case the output follow the input and it is update at each rising edge of the clock. An important note is that the reset polarity will be changed afterwards for synthesis reasons that will be explained afterwards in chapter 5.

3.2 D Flip-Flop with Set/Reset and Tap Circuit

The memory cell which maintains a bit within the LFSR is the D-Flip-Flop with Set/Reset and tap circuit that implements the characteristic features of the LFSR which differentiates it from a simple shift register.

```
1  -- D Flip-Flop with Set/Reset and tap circuit
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  entity dff_tap_circuit is
6    port(
```

```
7       clk        : in  std_logic;
8       reset      : in  std_logic;
9       set        : in  std_logic;  -- initialization input
10      d_in       : in  std_logic;
11      isTap      : in  std_logic;  -- it indicates if the input must be changed
12                                   -- (it does if the previous ff is a tap)
13      feedback   : in  std_logic;  -- it is the feedback bit (the N-1)
14      q_out      : out std_logic
15    );
16  end dff_tap_circuit;
17
18
19  architecture rtl of dff_tap_circuit is
20  component dff is
21    port(
22      clk    : in  std_logic;
23      reset  : in  std_logic;
24      set    : in  std_logic;
25      d      : in  std_logic;
26      q      : out std_logic
27    );
28  end component;
29
30  signal new_d : std_logic;
31  begin
32    internal_dff : dff
33    port map(
34      clk    => clk,
35      reset  => reset,
36      set    => set,
37      d      => new_d,
38      q      => q_out
39    );
40
41    new_d <= d_in xor (isTap and feedback);
42
43  end rtl;
```

This device acts as the previous, the only difference is the addition of the tap circuit which is described in the row 41.

3.3 LFSR

Now we see the way in which we have to aggregate the previous devices in order to implement an LFSR.
We use (0 to N) instead of (N downto 0) in the declaration of the standard logic vectors for isTap and seed, because they are not numbers but they indicate positions, so the left-most bit is the one for the 0^{th} flip-flop and so in order to access it I would like to use the following notation: *vector(0)* and not this *vector(N-1)*. To achieve this aim I have to use the keyword *to* instead of *downto*.

```
1   -- LinearFeedbackShiftRegister
2   library ieee;
3   use ieee.std_logic_1164.all;
4
5   entity lfsr is
6     generic (Nbit : positive := 16);
7     port(
8       clock      : in  std_logic;
9       reset      : in  std_logic;
10      isTap      : in  std_logic_vector(0 to Nbit-2);
11      seed       : in  std_logic_vector(0 to Nbit-1);
12      outputBit  : out std_logic;
13      -- debugging
14      state      : out std_logic_vector(Nbit-1 downto 0)
15    );
16  end lfsr;
17
18
19  architecture rtl of lfsr is
```

```
20      component dff_tap_circuit is
21        port(
22          clk       : in  std_logic;
23          reset     : in  std_logic;
24          set       : in  std_logic;
25          d_in      : in  std_logic;
26          isTap     : in  std_logic;
27          feedback  : in  std_logic;
28          q_out     : out std_logic
29        );
30      end component;
31
32      signal lastBit : std_logic := '0';
33      signal intercon : std_logic_vector(0 to Nbit-2);
34    begin
35      GEN:for i in 0 to Nbit-1 generate
36        FIRST:  if i = 0 generate
37        FF1 : dff_tap_circuit port map(clock, reset, seed(i), lastBit,'0', lastBit
              ,intercon(i));
38        end generate FIRST;
39
40        INTERNAL: if (i>=1) and (i<Nbit-1) generate
41        FFI : dff_tap_circuit port map(clock, reset, seed(i), intercon(i-1), isTap
              (i-1), lastBit, intercon(i));
42        end generate INTERNAL;
43
44        LAST: if i=Nbit-1 generate
45        FFL : dff_tap_circuit port map(clock, reset, seed(i), intercon(i-1), isTap
              (i-1), lastBit, lastBit);
46        end generate LAST;
47      end generate GEN;
48
49      state <= intercon & lastBit;
50      outputBit <= lastBit;
51    end rtl;
```

In the next chapter we will see the strategies to test this device and to ensure that it works as a Linear Feedback Shift Register.

Chapter 4

Test-Plan

In this chapter we will test the LFSR implemented in VHDL in the previous chapter. In order to accomplish this aim we will start testing the basic elements of the implementation up to the final LFSR. The tests performed and explained will be:

- Test of a simple D-Flip-Flop
- Test of a D-Flip-Flop with tap circuit
- Test of the LFSR with isTap inputs to 0, so it's a simple shift register
- Final test of the LFSR using a C++ program which does in software what we implemented in hardware using VHDL

In some cases only a portion of the code will be shown, for the full code see files in LFSR/hdl/tb/ When we indicate the number of clock cycles we don't consider the clock cycles of the reset state. So "after x clock cycles" means "after x clock cycles counted from when reset is not active.

4.1 D-Flip-Flop Test

In order to test the simpler device (dff) we give as input 0, so the expected result is that until we change the input the output remains 0.

```
1  dff_process : process(clk_tb,reset_tb)
2  variable t : integer := 0;
3  begin
4    if(reset_tb='0') then
5      t:=0;
6      d_tb <= '0';
7    elsif (rising_edge(clk_tb)) then
8      case(t) is
9        when 5 => d_tb <= '1';
10       when 10 => end_sim <= '0';
11       when others => null;
12     end case ;
13     t:=t+1;
14   end if;
15 end process;
```

The test consist in given 0 at the reset and then after 5 clock cycles change the input to 1. As we can see in the image flip flop works as we expected:

- At the reset the output is equal to the value of the set
- The ouput remains unchanged until the input is 0, when the input change to 1 (5 clocks after the end of reset) also the output follows it after a clock cycle.

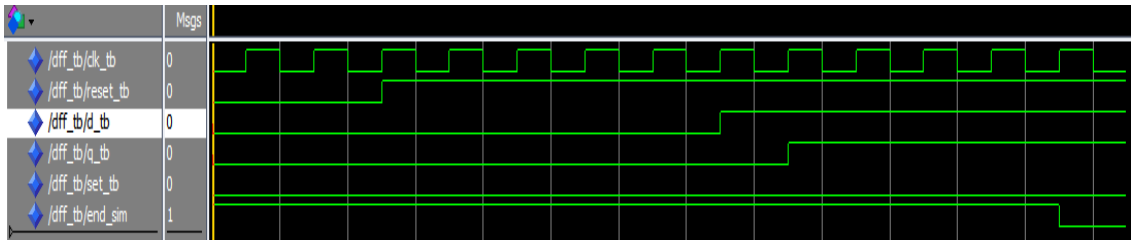

Figure 4.1: Simulation of dff test-bench using ModelSim

4.2 D-Flip-Flop with Tap Circuit Test

Now we want to test also the flip-flop with tap circuit, if the input isTap is 0 the flip-flop works exactly as the previous one. Instead if isTap is 1 and the feedback bit is 1 too then it's like an inverter (because the xor at the input has always an input at 1).

```
1  dff_process : process(clk_tb,reset_tb)
2  variable t : integer := 0;
3  begin
4    if(reset_tb='0') then
5      t:=0;
6      isTap_tb <= '0';  -- It's a normal dff
7      feedback_tb <= '1';
8      d_tb <= '0';
9    elsif (rising_edge(clk_tb)) then
10     case(t) is
11       when 5  => d_tb <= '1';
12       when 10 => isTap_tb <= '1'; d_tb <='0';
13       when 15 => d_tb <='1';
14       when 20 => end_sim <= '0';
15       when others => null;
16     end case ;
17     t:=t+1;
18   end if;
19 end process;
```

So we expect that for the first 10 clock cycles after reset the waves are the same of the previous case and then they present a inverter-like behaviour.

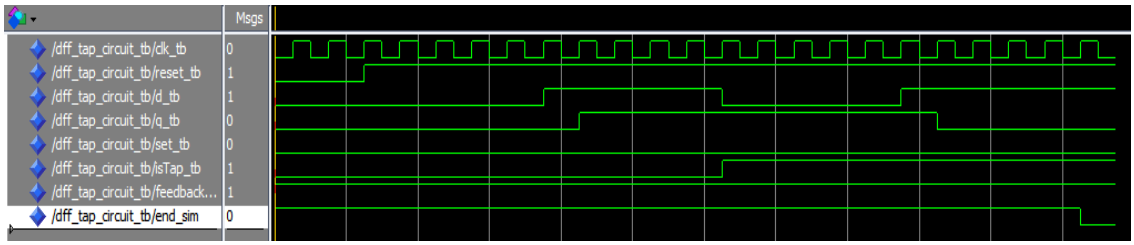

Figure 4.2: Simulation of test-bench of dff with tap circuit using ModelSim

From the figure 4.2 we can say that the behaviour is the one that we expect:

- In the first 10 clock cycles (after the reset) the waves are the same of the dff case. This is right because in this period we have isTap to 0 and so the tap circuit is disabled
- When isTap goes to 1 the dff with tap circuit works as an inverter (remember that feedback is 1). In fact in this moment we can see that the output remains one even if the input goes from 1 to 0. This happens because the input goes to 0 so the output at the next clock cycle will became one but it is already one, then it remains 1.

  When the input goes to 1 (after 15 clock cycles), the output, after a clock cycle, goes to 0 as we expected.

4.3 LFSR as Shift Register

If all bits of isTap are setted to 0 then we have a simple shift register because there are no taps. So this test consists in doing this and in controlling the proper work of this shift register.

```
1  lfsr_test_process : process(clk_tb,reset_tb)
2  variable t : integer := 0;
3  begin
4    if(reset_tb='0') then
5      t:=0;
6      isTap_tb <= "0000000000000000";
7      seed_tb <= "0000101011000110";
8    elsif (rising_edge(clk_tb)) then
9      if actual_state=seed_tb and t=0 then
10       t:=t+1;
11     elsif actual_state=seed_tb and t=1 then
12       end_sim <='0';
13     end if;
14   end if;
15 end process;
```

We stop the simulation when the actual state of the shift register is equal to the initial one, hence we expect that the simulation ends after 16 clock cycles because the register has a length of 16 and so after this period each bit are shifted 16 times and so it has returned in the initial position.
The reason why we neglect the first time that th actual state is equal to the seed is because this happens at the initialization and if we consider this the first time then the simulation ends early.

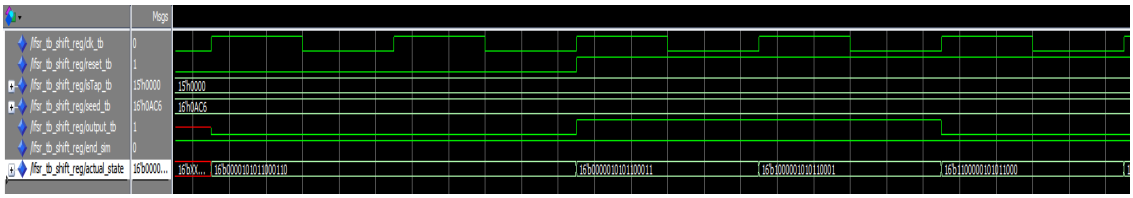

Figure 4.3: Part of the simulation of the test-bench of a lfsr setted as shift register

In the figure 4.3 we can see the initial part of the simulation, as we can see the first states of the device are the ones that we expect from a shift register.

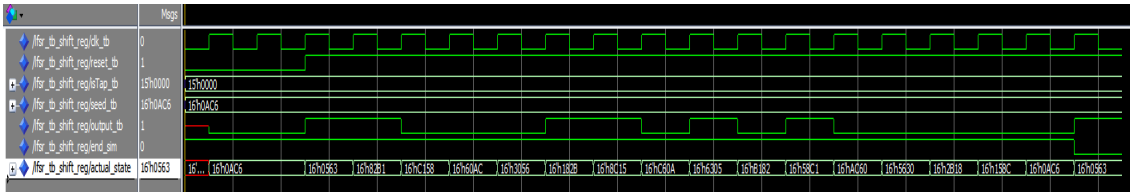

Figure 4.4: Complete simulation of the test-bench of a lfsr setted as shift register

In the figure 4.4 we can see the complete configuration (in order to have a more compact image the format of the state is hexadecimal). Here we can see that after the reset the shift register spends 16 clock cycles to reach its initial state as we expected.

4.4 LFSR

In order to prove the proper work of the LFSR we do a final test using a test-bench in VHDL and a C++ program. The program does in software the operations that are done in hardware by an LFSR. Both the test-bench and the program write their output on a file, the two file must be equal to ensure the correctness of the LFSR. Two script helps in doing this operations, one compiles and executes the C++ program, instead the other one compares the two output file using the UNIX tool *diff*. The scripts work only in a UNIX-like environment.

4.4.1 LFSR Test-bench

The test-bench is like the previous one, but in this case the tap are setted at the following feedback polynomial:

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

Then in terms of bit position the taps are: 10, 12, 13. Because of the fact that we start to count from 0. The bits 0 and 15 (i.e. the first one and the last one are always taps in a LFSR because the output is fed back to the input, but they are not taps in the sense that the must be XORed).

```
1  lfsr_test_process : process(clk_tb,reset_tb)
2  variable t : integer := 0;
3  begin
4    if(reset_tb='0') then
5      t:=0;
6      isTap_tb <= "000000000010110";
7      seed_tb <= "0000101011000110";
8    elsif (rising_edge(clk_tb)) then
9      if actual_state=seed_tb and t=0 then
10       t:=t+1;
11     elsif actual_state=seed_tb and t=1 then
12       end_sim <='0';
13     end if;
14   end if;
15 end process;
16
17 write_file : process
18 variable BUF : line;
19 begin
20   loop
21     wait on clk_tb until clk_tb='1';
22     if reset_tb='1' then
23       WRITE(BUF,output_tb);
24       WRITEline(OUT_LFSR,BUF);
25     end if;
26   end loop;
27 end process;
```

The last process is in charge of writing the output on a file in order to compare it afterwards with the output of the C++ program.
The statement at row 21 means that the output is written in the file on the rising edge of the clock (it waits until the clock changes from 0 to 1). Then at row 22 the if statement is needed in order to avoid the print of the output during the reset state.

4.4.2 C++ Test Program

The C++ program follows the same approach of the VHDL implementation. It is a Galois LFSR implemented in software. Anyway the implementation is not very performing because the target of the program is to test an already implemented hardware LFSR and not to implement in the most efficient way an LFSR.

```
1  int main()
2  {
3    // Initialization value
4    int seed[N] = {0,0,0,0,1,0,1,0,1,1,0,0,0,1,1,0};
5    // Actual state of the LFSR
6    int actual_state[N];
7    // Reset Phase: actual state is initialized to the seed value
8    for(int i = 0; i<N; i++)
9      actual_state[i] = seed[i];
10   do {
11     // Feedback bit
12     int lastBit = actual_state[N-1];
13     // Next LFSR state
14     int next_state[N];
15     // Shift operation
16     for(int i = 0; i<N; i++)
17     {
18       if(i!=0 && isTap(i-1))
```

```
19            next_state[i] = xorFunction(actual_state[i-1],lastBit);
20          else if(i==0)
21            next_state[i] = lastBit;
22          else
23            next_state[i] = actual_state[i-1];
24        }
25
26        // The next state became the actual state
27        for(int i = 0; i<N; i++)
28          actual_state[i] = next_state[i];
29
30        // Print the output
31        cout << lastBit << endl;
32      }while(!compare(actual_state,seed));
33
34      /* Print the output of the last state (i.e. the first of the new cycle)
35       * This is necessary to emulate the effect of the end_sim
36       * in VHDL. When the condition in the while is true then
37       * the simulation ends but in the ModelSim simulation
38       * it ends after a clock (because end_sim goes to 0 after
39       * a clock cycle). So the output of the first state of the
40       * new cycle must be considered*/
41      cout << actual_state[N-1] << endl;
42      return 0;
43    }
```

Only the main is shown, for the complete code see

`/lfsr_cpp/lfsr_software.cpp`

It's important to underline the statement at row 41. This statement is necessary in order to take into account the fact that the ModelSim simulation ends after a clock cycle, that is the period that the signal that is in charge of ending simulation needed to change from 0 to 1. More in the comment within the code.

4.4.3 Helper Scripts

The first script is in charge of compiling the code and redirecting the output in the file output.txt.

```
1  #!/bin/bash
2  g++ lfsr_software.cpp -o lfsr
3  ./lfsr > output.txt
```

The second one is in charge of use the tool *diff* to compare the output of the program and the output of the ModelSim simulation.

```
1  #!/bin/bash
2  # Check if there are differences between two file
3
4  VAR=`diff output.txt ../LFSR/ModelSim/LFSR_test/fileout.tv`
5
6  if [ -z "$VAR" ]
7  then
8    echo "OK - NO DIFFERENCES"
9  else
10   echo $VAR
11 fi
12
13 echo "Press a key to exit"
14 read input
```

At row 6 the output of diff is controlled and the condition in the if statement return true if and only if the command return null (so it means that there are no differences between the two files). Otherwise it means that there are some differences and so these are shown.

4.4.4 ModelSim Simulation

The first step of the complete test is the ModelSim simulation performed with the test-bench seen at paragraph 4.4.1.

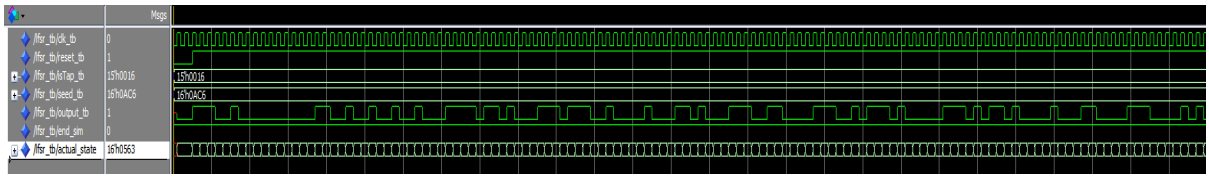

Figure 4.5: Part of the simulation of the lfsr test-bench

In the figure 4.5 is shown the output of the lfsr simulation as in the wave form. From this we can extract very poor information, in fact the only thing that we can say is that at the first view the output doesn't show a pattern and so maybe it can considered pseudo-random, but in order to be sure of this we have to check the output of this simulation with the C++ program output.

4.4.5 Check The Simulation With C++ Program

After the ModelSim simulation we have the output in the file at the following path:

```
/LFSR/ModelSim/LFSR_test/fileout.tv
```

Now we have to use the helper scripts.
The first thing to do is to go in the directory in which there are C++ program and scripts.

```
cd /lfsr_cpp/
```

Then we can execute the script which compiles and redirect the output of the program to the file output.txt

```
./run_lfsr.sh
```

If the console gives no error then the compilation and the execution are done successfully. So now we can check the two output using the proper script:

```
./check.sh
```

If the console returns

```
OK - NO DIFFERENCES
```

Then it means that the test is ended successfully and that the LFSR implemented in VHDL works in a proper way. In the figure 4.6 we can see the output of the last steps in Linux console.

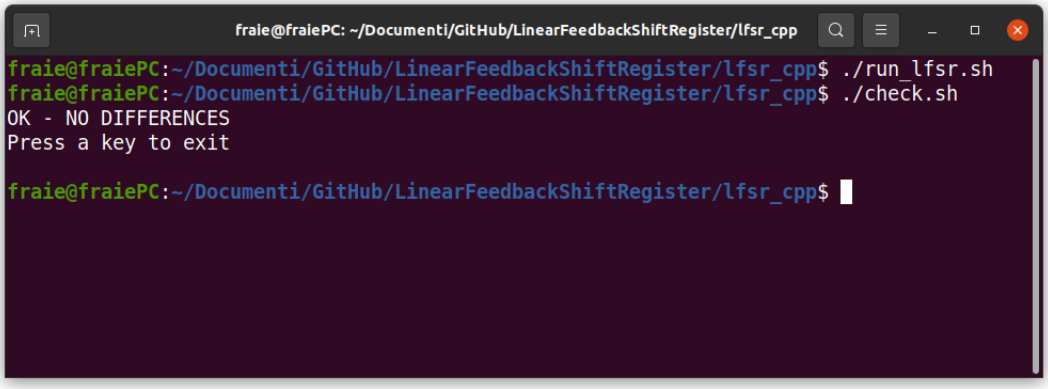

Figure 4.6: Execution of the scripts in Linux console

4.4.6 LFSR Test Conclusion

For a complete test plan we have to test all the $2^{16} - 1$ (Remember that the state with all 0 is not allowed) possible seeds and so we have to do 65535 tests and comparisons. Only in this way we can be sure, without doubts, that the LFSR works properly. Anyway we do only some tests and we can say with a reasonable confidence that the LFSR works in a proper way.
In particular three tests are performed with the following seeds:

- 0000101011000110

  ```
  /lfsr_cpp/output.txt
  /LFSR/ModelSim/LFSR_test/fileout.tv
  ```

- 1100101010100111

  ```
  /lfsr_cpp/output1.txt
  /LFSR/ModelSim/LFSR_test/fileout1.tv
  ```

- 0000000000000110

  ```
  /lfsr_cpp/output2.txt
  /LFSR/ModelSim/LFSR_test/fileout2.tv
  ```

Each test has been performed as the one previously explained and each one has given a successfully output.

Chapter 5

Logic Synthesis

In this chapter we discuss the Logic Synthesis of the previously implemented LFSR, with Xilinx Vivado, for ZyBo Board (ZYNQ XC7Z010-1CLG400C).

5.1 LFSR Wrapper

In order to synthesize the LFSR on the FPGA we have to do some trade-offs. This because the ZyBo Board has only 4 push buttons and 4 slide switches as input. So we have to implement a wrapper that reduces the number of inputs of our LFSR.
Before the synthesis we have the following inputs[1]:

- isTap: 15 bits
- seed: 16 bits
- reset: 1 bit

So we would need 32 input ports and we have not them. To resolve this issue we do the following trade-offs:

- The input for decide which bits are taps or not, is given by the wrapper and it is a pre-determined value (the same seen in test-plan, that is the one of the requirements).
- Only a part of the seed value is editable, this part will be connected with the 4 slides switches and a feedback of which slides are active, is given to the user using 4 leds provided by the board. The rest of the seed is pre-determined with a non-zero value, so even if the user doesn't change the slides, the LFSR works properly because in any case the seed is a non-zero value.

  So the seed will be *userInput* + *101011000110* (where + means concatenation).
- The reset value is connected with one of the push buttons. This means that we have to invert the polarity of the reset, because when a user pushes the button this means that a V_{cc} to the reset is given, so the reset is asserted when the push buttons is pressed only if the polarity of the reset is inverted. In order to resolve this issue, the polarity of the dff must be inverted.

Then we have to resolve also the issue about the output. In this case we have as output only a bit, so if we want to show the result on a screen we have to use the VGA output. In order to do this we can use 6 bits of DAC on the GRN channel. Thus, we assign the output bit to the less significant bit of these 6 and the others are setted to 0.
To see the wrapper VHDL code see the correspondent VHDL file:

`/LFSR/hdl/src/lfsr_wrapper.vhd`

[1] The clock is not considered because it is not an input given by the user

5.1.1 Wrapper Test

Also the wrapper has been tested using the test-bench and the comparison with the C++ program. The wrapper test bench is in the path:

`/LFSR/hdl/tb/lfsr_wrapper_tb.vhd`

Some observations about it are needed:

- The reset polarity is inverted in order to emulate the behaviour of the push buttons (as we can see in the figure 5.1).

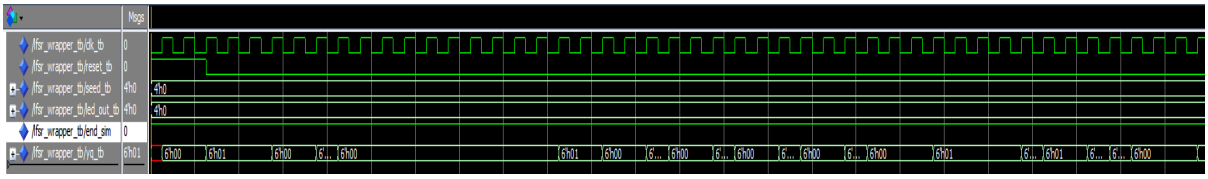

Figure 5.1: Part of the simulation of the lfsr wrapper

- The output (only the LSB) is saved in a file in order to be compared with the C++ program output.
- Due to the fact that now there is no way to know the internal state of the LFSR we stop the simulation after 65536 clock cycles.

Then, after ModelSim simulation, we can use the scripts (changing the paths of the file) to check the simulation output file. The correctness of the wrapper implementation is ensured by the fact that the wrapper doesn't modify the job of the standard LFSR (that is emulated by the C++ program) because the simulation output file and the C++ output file have no differences.

5.2 RTL Analysis

The first step of Vivado workflow is to perform the RTL analysis of our device. The output of this step is a schematic of the LFSR.

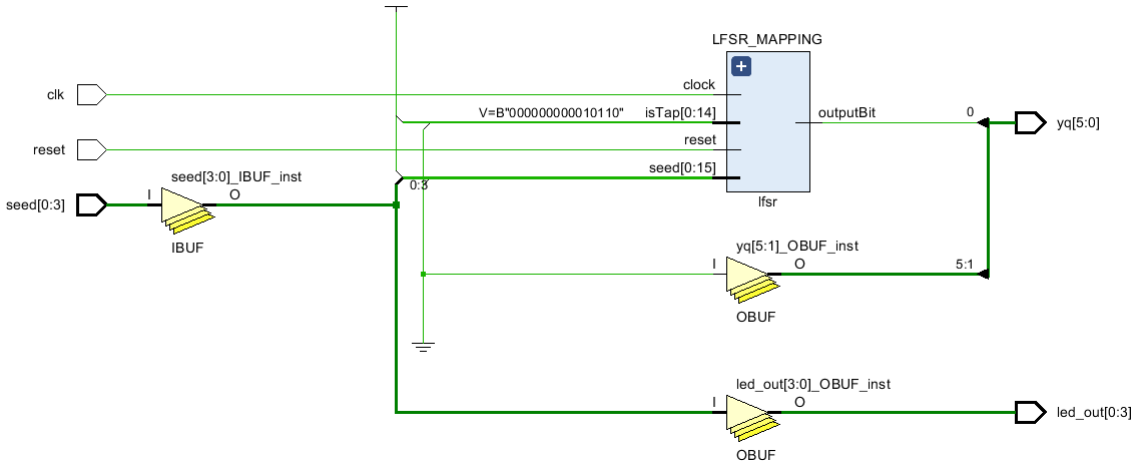

Figure 5.2: RTL Analysis Schematic

In the figure 5.2 we can see the schematic, we can recognize the inputs and the output and the fact that some of them are initialized to constant values (the connections to ground and to supply).

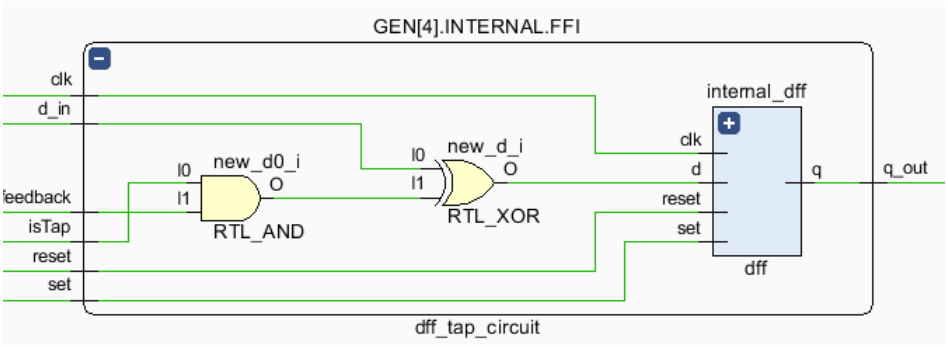

Figure 5.3: Detail of a D-Flip-Flop of RTL Analysis Schematic

5.3 Synthesis

With this second step we map our structure on the target technology.

5.3.1 Clock Period Setting

In this step we have to set the clock timing. We try to get a clock frequency of 125 MHz (8 ns). If we run the synthesis with this clock frequency we obtain the timing summary in figure 5.4.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	6,357 ns	Worst Hold Slack (WHS):	0,134 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:	20	Total Number of Endpoints:	20	Total Number of Endpoints:	21

All user specified timing constraints are met.

Figure 5.4: Design Timing Summary with period clock equal to 8 ns

We can see that the slack is 6,357 ns, so this means that our device can work at an higher frequency. So we edit the clock period to 5 ns. In this case we have the timing summary in figure 5.5.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3,357 ns	Worst Hold Slack (WHS):	0,134 ns	Worst Pulse Width Slack (WPWS):	2,000 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:	20	Total Number of Endpoints:	20	Total Number of Endpoints:	21

All user specified timing constraints are met.

Figure 5.5: Design Timing Summary with period clock equal to 5 ns

So due to the fact that the slack is positive also now, we reduce the clock period to 2.7 ns so 370 Mhz. Also in this case the Worst Negative Slack is positive, then we set a clock period equal to 2.5 ns and so we have a clock frequency equal to 400 MHz. The resulting Design Timing Summary is shown in figure 5.6.
We can reduce also now the clock period but, in order to have a bit margin of error, we maintain this clock period. Repeating this procedure we arrive at the point that the minimum clock period is 2.1 ns and so the maximum clock frequency achievable is 476 MHz.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,857 ns	Worst Hold Slack (WHS):	0,134 ns	Worst Pulse Width Slack (WPWS):	0,345 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:	20	Total Number of Endpoints:	20	Total Number of Endpoints:	21

All user specified timing constraints are met.

Figure 5.6: Design Timing Summary with period clock equal to 2.5 ns

5.3.2 I/O Planning

The next step is to do the I/O planning: this means that we have to connect our inputs and our outputs to the pins of the board. We can see it in the figure 5.7.

Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type	Off-Chip Termination	IN_TERM
All ports (16)													
led_out (4)	OUT			☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
led_out[0]	OUT		M14	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
led_out[1]	OUT		M15	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
led_out[2]	OUT		G14	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
led_out[3]	OUT		D18	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
seed (4)	IN			☑	(Mu...	LVCMOS33*	3.300				NONE	NONE	
seed[0]	IN		G15	☑	35	LVCMOS33*	3.300				NONE	NONE	
seed[1]	IN		P15	☑	34	LVCMOS33*	3.300				NONE	NONE	
seed[2]	IN		W13	☑	34	LVCMOS33*	3.300				NONE	NONE	
seed[3]	IN		T16	☑	34	LVCMOS33*	3.300				NONE	NONE	
yq (6)	OUT			☑	(Mu...	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
yq[5]	OUT		F20	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
yq[4]	OUT		H20	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
yq[3]	OUT		J19	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
yq[2]	OUT		L19	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
yq[1]	OUT		N20	☑	34	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
yq[0]	OUT		H18	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
Scalar ports (2)													
clk	IN		L16	☑	35	LVCMOS33*	3.300				NONE	NONE	
reset	IN		R18	☑	34	LVCMOS33*	3.300				NONE	NONE	

Figure 5.7: I/O Planning

5.3.3 Warning Message Analysis

At the end of the synthesis we have 11 warning messages. We can see them in the figure 5.8.

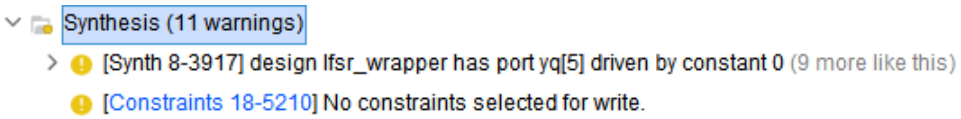

Figure 5.8: Warning messages after synthesis

The warning messages can be easily explained: they are related to the fact that we connect part of the output to the constant 0. But this is what we want, because we have only a bit as output and if we want to use the VGA then the other bits must be setted manually to 0, because only the LSB of yq is the real output.

5.3.4 Estimated Resources Utilization

The estimated resources utilized are shown in the figure 5.9 and 5.10.

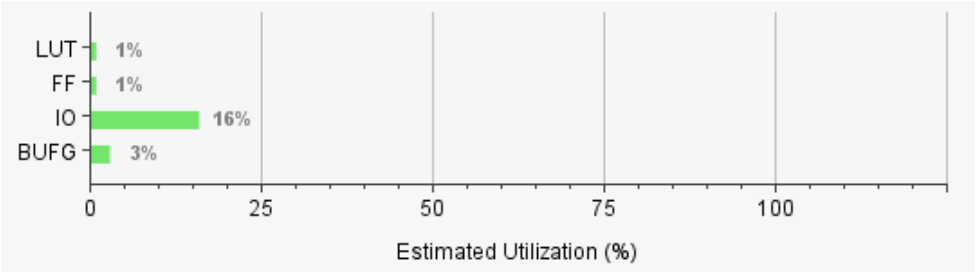

Figure 5.9: Graph representation of the estimated resource utilization

In the figure 5.9 we can see that the most estimated utilization is about the IO resource, instead we use a very low number of the available flip-flops and LUTs.

Resource	Estimation	Available	Utilization %
LUT	14	17600	0.08
FF	20	35200	0.06
IO	16	100	16.00
BUFG	1	32	3.13

Figure 5.10: Table representation of the estimated resource utilization

In particular we can see in the figure 5.10 that we use 32 flip-flops and 14 LUT with an utilization that is less than 0.1%

5.3.5 Schematic after synthesis

After the synthesis we can see a schematic of the LFSR mapped on the target technology, so we can see how our logic is translated in terms of flip-flops and LUTs. In the figure 5.11 we can see the schematic after synthesis and in the figure 5.12 the detail of a flip flop.

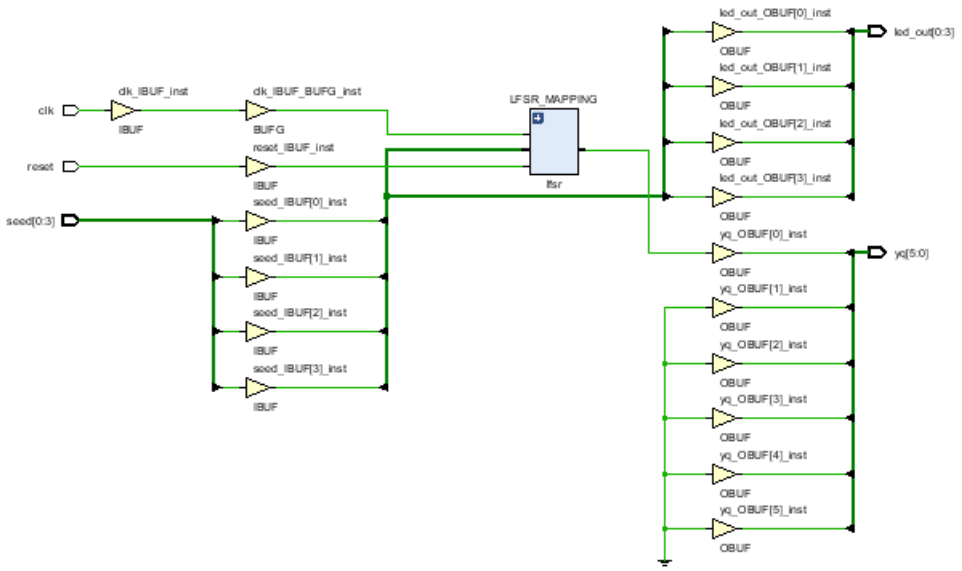

Figure 5.11: Schematic after synthesis

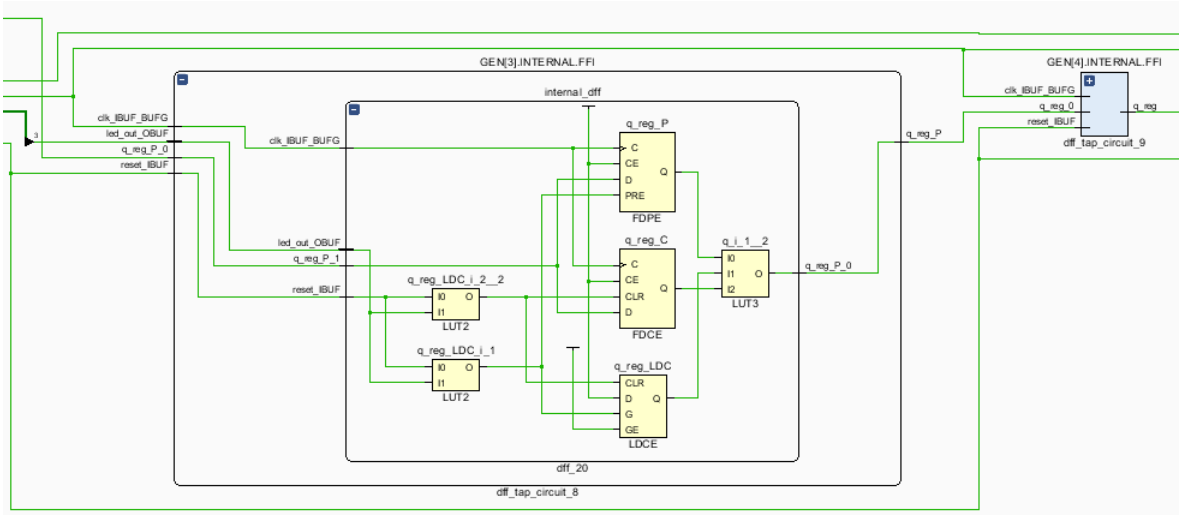

Figure 5.12: Detail of a D-Flip-Flop with tap circuit in the schematic after synthesis

5.4 Implementation

Scrivere che anche dopo l'implementazione lo slack è ancora positivo e mostrare il power report e commentarlo. Dire inoltre che le estimated resource sono correte e sono le stess dopo l'implementazione

Chapter 6

Conclusion