UNIVERSITÀ DI PISA

MSc. Computer Engineering

Electronics and Communications Systems

Pseudo Noise Sequence Generator

Project Report

Veronica Torraca

A. Y. 2023-24

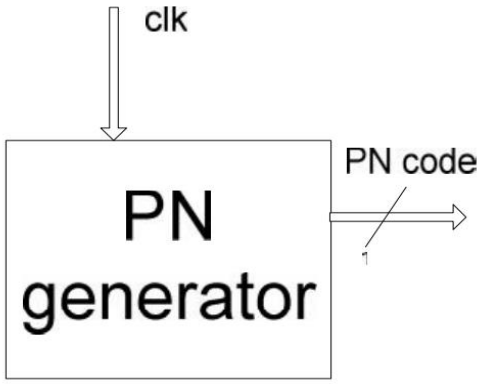
Contents

Introduction and specifications 3
Pseudo Noise Sequence Generator 3
PNSG for CDMA transmissions 4
Algorithm description and possible structure 4
Architecture description 5
Components and blocks 7
Combinational Logic 7
VHDL Code Analysis 8
Mutex 8
D Flip-Flop 9
LFSR element 10
PNSG 11
Test-plan and Testbench 12
Testbench for the PNSG 12
Test on Modelsim 13
Python Test Script 14
Synthesis on Xilinx FPGA Zynq 14
Analysis 15
Critical path 16
Resource utilization and power consumption 19
Conclusions 20

Introduction and specifications

The aim of this project is to design a **Pseudo Noise Sequence Generator**, for CDMA transmissions (IS-95 phase generator) with the following requirements:

1. 15 stage PN generator
2. Feedback polynomial: $\boldsymbol{x^{15} + x^{13} + x^9 + x^8 + x^7 + x^5 + 1}$



Pseudo Noise Sequence Generator

A **Pseudo-random noise generator** (also known as Pseudo Random Number Generator, PRNG) is an electronic device for generating a sequence of numbers (bits) that appear to be random, but actually they are generated in a deterministic way. A good **PRNG** is a generator whose output is much closer to a **truly random** sequence and it can be useful for several applications, such as:

- **Simulations**, to simulate random events
- **Neural Networks**, to introduce randomness during the network training
- **Cryptography**, to generate encryption keys (but with the addition of a more elaborate algorithm to make the output unpredictable)
- **Electronic games**, for example for procedural generation (method to create data algorithmically)

and many other scenarios requiring random elements.

The PNSG is fed by a *sequence of bits* called **seed** and uses an algorithm to expand the seed into a longer sequence. Depending on the algorithm chosen (and the feedback polynomial), the sequence starts to repeat after a certain period of time.

PNSG for CDMA transmissions

As required, the generator to design for this project has to be an **IS-95 phase generator** for **CDMA** transmissions.

IS-95 was the first CDMA digital cellular technology, a multiple access scheme for digital radio transmissions. CDMA (Code Division Multiple Access) transmits streams of bits, called Pseudo Noise code (**PN codes**), and uses the same channel for all the users. To distinguish them from each other, each user has its own PN code, which is orthogonal to each other. Moreover these PN codes have zero (or minimum) cross-correlation and maximum autocorrelation. Another characteristic of the PN codes is that they repeat after a very long time (period), making them appear random. The sequence of the bit stream is usually generated by an **LFSR** (Linear Feedback Shift Register), starting from a keystream (seed), different for each user and shared between the mobile station and the base station, so both parts can generate the same PN sequence.

Algorithm description and possible structure

As already mentioned, a PNSG uses a LFSR to produce the sequence of bits, so the algorithm for generating the PN code, basically, is that of a Linear Feedback Shift Register.

An LFSR takes as input a sequence of **n bits**, called **seed**, and at each step produces an output of the same length, elaborating the seed in some way through a specific logic function. After a certain number of steps, known as the **sequence period** and typically measured in clock cycles, the generated code repeats. The period length depends on the seed and the combinational logic implemented by the generator, i.e. the feedback polynomial of degree *n*. With a seed of **n bits**, the maximum output sequence length will be $(\boldsymbol{2^n - 1})\boldsymbol{n}$ bit, sequence obtained after $2^n - 1$ clock cycles, since in each clock the generator provide, in parallel, a sequence of *n bits*.

The coefficients of the feedback polynomial define whether the feedback path through the corresponding bit is active or not (the corresponding bit must be *xor'd* or not). The output bits that will influence the next input state are called ***taps***, so the list of positions of these bits in the feedback polynomial is the ***tap sequence***. So, the taps are xor'd sequentially and the resulting bit is fed back as the leftmost input bit. All other bits are shifted one position to the right, as a typical shift register.

In this particular case, we have a PNSG composed by **15 stages**, so we have a seed of 15 bits in input and therefore we need15 memory units to store these values.

A general block diagram is represented in Figure 1.

Figure 1 - Block diagram

As required in the specification, the feedback polynomial is:

$$x^{15} + x^{13} + x^9 + x^8 + x^7 + x^5 + 1$$

So the tap sequence is ***[15 13 9 8 7 5].***

A possible basic general structure is shown in Figure 2, with the taps in green.

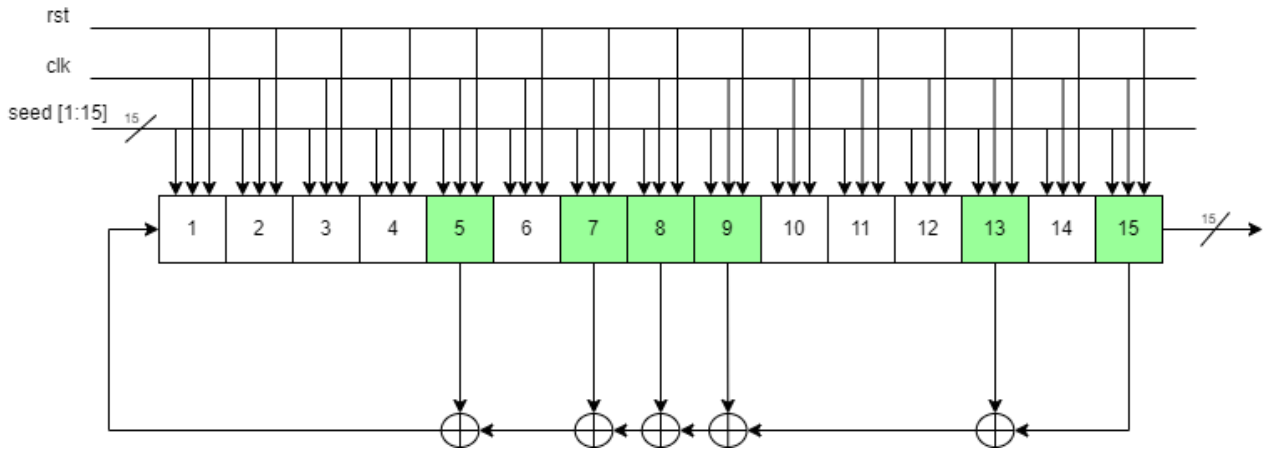


Figure 2 - Basic structure

Architecture description

The architecture chosen for the PNSG is a simple architecture with 3 basic elements:

- **Multiplexer**, to select the correct input bits
- **D Flip-Flop**, to store and keep stable the bits for a clock cycle, so used as registers
- **XOR** port, for the combinational logic

Then we have multiple input and output signals, in particular:

- Input
  - **`clk`** [1 bit], the clock signal of the system
  - **`rst_n`** [1 bit], an asynchronous active low reset signal
  - **`load_seed`** [1 bit], control signal to load the initial seed in the D-FF registers
  - **`seed`** [15 bits], initial seed of the generator
- Output
  - **`PN_code`** [15 bits], the sequence of bits generated at each step

In addition to these external signals, there are also a number of internal signals used to connect the various components, as described later in this report.

In Figure 3, the schematic of the system architecture generated through *Vivado* tool is shown.

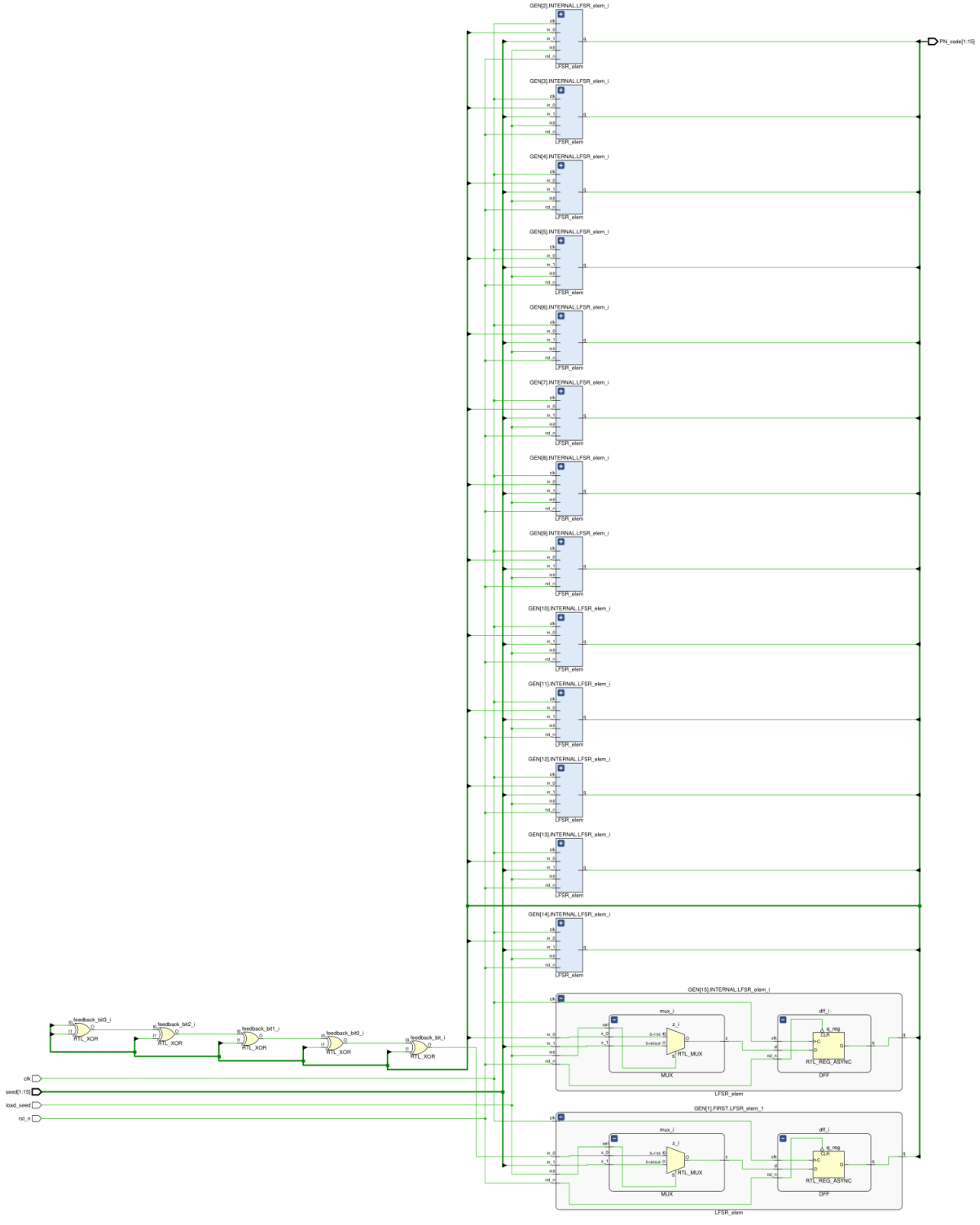


Figure 3 - System architecture

Components and blocks

As mentioned above, the designed PNSG consists mainly of **multiplexers** and **D Flip-Flops**, linked together to form the LFSR.

Figure 4 shows the basic element of the LFSR, a mutex linked to a D-FF register. A number of these types of elements cascaded to form the LFSR and hence the PNSG. In this particular case we have a 15 stage generator, so there are 15 of these *LFSR elements* connected in a chain.

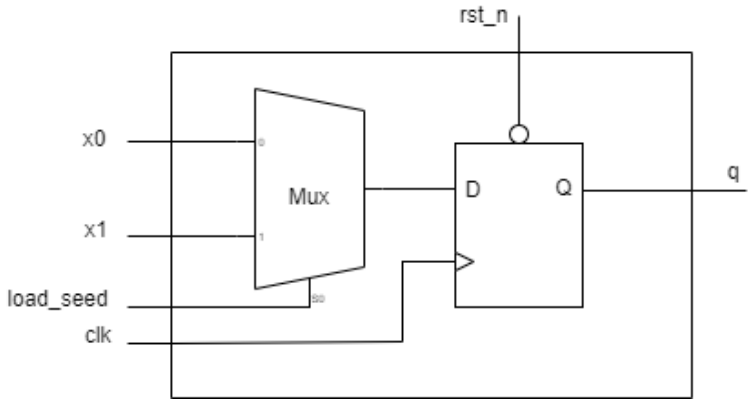


Figure 4 - LFSR element

The **mutex** is necessary to select the **correct input bit** for each **DFF**, the choice is between the corresponding bit of the **seed** (*input 1*) and the **feedback** bit, for the first DFF of the chain, or the output bit of the previous DFF, for a generic internal block (*input 0*).

The bits of the **seed** are loaded into the shift register during the initial phase, i.e. when the **load_seed** bit of the circuit is active (`load_seed='1'`). Otherwise, during the PN code generation phase, this bit is equal to '0', and a generic *dff(i)* takes as input the output *q(i-1)* provided by the previous *dff(i-1)*, while the *first dff* has as input the feedback bit, generated in the previous step.

Combinational Logic

The combinational logic of this system depends on the **taps** of the characteristic (feedback) polynomial. In particular, we have a 15 degree polynomial with the tap sequence as indicated above, *[15 13 9 8 7 5]*. Each tap corresponds to the bit to **XOR** in sequence with the result of the previous XOR port, as shown in Figure 5. More precisely, the function to implement is:

$$\boldsymbol{q_{15} \oplus q_{13} \oplus q_9 \oplus q_8 \oplus q_7 \oplus q_5}$$

where q_i is the output bit of the *dff(i)* and the symbol $\oplus$ is the XOR operator.

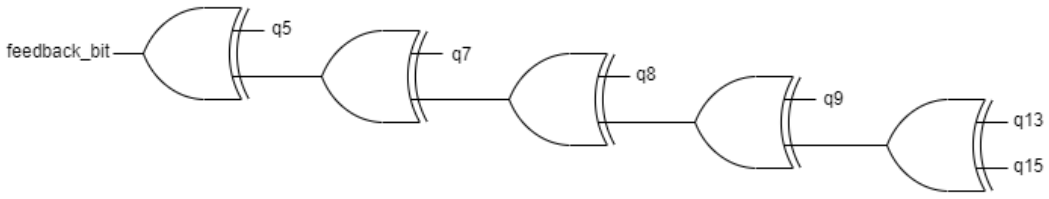


Figure 5 - Combinational logic

VHDL Code Analysis

In this section the VHDL code for each components of the PNSG described above, is reported.

Mutex

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

-- multiplexer 2-to-1 -> to select the initial seed or the feedback bit

entity MUX is
    port (
      sel   : in std_logic;     -- control signal to select the correct input, active high
      x_0   : in std_logic;     -- input if sel = 0 -> previous stage output bit
      x_1   : in std_logic;     -- input if sel = 1 -> in this particular case, the seed bit
      z     : out std_logic     -- output bit
    );
end MUX;

architecture beh of MUX is

begin
  mux_p: process(sel, x_0, x_1)
  begin
    if sel = '0' then
        z <= x_0;       -- previous stage bit in output
    else
        z <= x_1;       -- bit of the seed in output
    end if;
  end process mux_p;

end beh;
```

Figure 6 - VHDL code for mutex

As already mentioned, the mutex component is necessary to select the correct input bit for the DFF.

We have **4 input** ports and **1 output** port, all of a single bit:

- **`sel`** (in), control signal of the mutex, active high
- **`x_0`** (in), input bit if `sel` is '0', corresponding to the feedback bit or the output bit of the previous stage
- **`x_1`** (in), input bit if `sel` is '1', corresponding to the bit of the seed
- **`z`** (out), output bit

In the architecture declaration part, is described the behavior of the mutex: the sensitivity list includes the signals: `sel, x_0, x_1`. That is because a change in one of these inputs affects the output: if the control signal `sel` is '0', the output follows the `x_0` input bit, otherwise, in output we have the input bit `x_1` (the corresponding bit of the seed, practically).

D Flip-Flop

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

-- d flip-flop -> positive edge triggered ff - asynchronous reset -> to store value

entity DFF is
    port (
      clk   : in std_logic;      -- clock signal
      rst_n : in std_logic;      -- active low reset
      d     : in std_logic;      -- input bit
      q     : out std_logic      -- output bit
    );
end DFF;

architecture beh of DFF is

begin
  dff_p: process(rst_n, clk)    -- sensitivity list: (resetn, clk)
  begin
    if rst_n='0' then
      q <= '0';
    elsif (rising_edge(clk)) then
      q <= d;
    end if;
  end process dff_p;

end beh;

```

Figure 7 - VHDL code for D Flip-Flop

The D Flip-Flop is the classical memory unit, with 3 input and 1 output signals, all of 1 single bit.

Since we have an active low reset, if the **rst_n** signal is '0', also the output **q** is '0', otherwise it follows the input **d** at each rising edge of the **clock** signal.

LFSR element

```vhdl

entity LFSR_elem is

    port (
      clk   : in std_logic;     -- clock signal
      rst_n : in std_logic;     -- active low reset
      init  : in std_logic;     -- control signal for mutex to insert the initial value
      in_0  : in std_logic;     -- input bit (previous stage result)
      in_1  : in std_logic;     -- input bit (seed)
      q     : out std_logic     -- output bit
    );

end LFSR_elem;

architecture struct of LFSR_elem is

    signal z_d : std_logic; -- internal signal to connect mux output to dff input

    component MUX is
        port (
          sel   : in std_logic;
          x_0   : in std_logic;
          x_1   : in std_logic;
          z     : out std_logic
        );
    end component MUX;

    component DFF is
        port (
            clk     : in std_logic;
            rst_n   : in std_logic;
            d       : in std_logic;
            q       : out std_logic
        );
    end component DFF;

begin

    mux_i: MUX
    port map (
        sel => init,
        x_0 => in_0,
        x_1 => in_1,
        z => z_d
    );

    dff_i: DFF
    port map (
        clk => clk,
        rst_n => rst_n,
        d => z_d,
        q => q
    );

end struct;

```

Figure 8 - VHDL code for LFSR element

The **LFSR_elem** is a structural element composed of a **mux** and a **dff**.

So, in the architecture declaration, we introduce an internal signal, **z_d**, to connect the output of the `mux` with the input of the `dff`. In addition, the input `in_0` of the LFSR_elem corresponds to the input `x_0` of the mux, so for the bit of previous stage, while the input `in_1` is for load the initial seed, so connected to the input `x_1` of the mutex.

PNSG

```vhdl
entity PNSG is

    generic (
        N_stage : positive := 15
        );

    port (
        clk         : in std_logic;      -- clock signal
        rst_n       : in std_logic;      -- reset signal, active low
        load_seed   : in std_logic;      -- control signal to load the initial seed in the registers
        seed        : in std_logic_vector(1 to N_stage);    -- initial seed of the generator
        PN_code     : out std_logic_vector(1 to N_stage)    -- generated output sequence
    );

end PNSG;

architecture beh of PNSG is

    signal q_out    : std_logic_vector (1 to N_stage);      -- output bits from each stage
    signal feedback_bit : std_logic;

    component LFSR_elem is
        port (
            clk     : in std_logic;
            rst_n   : in std_logic;
            init    : in std_logic;
            in_0    : in std_logic;
            in_1    : in std_logic;
            q       : out std_logic
        );
    end component LFSR_elem;

begin

    GEN: for i in 1 to N_stage generate
        -- first stage element
        FIRST: if i = 1 generate
            LFSR_elem_1: LFSR_elem
            port map (
                clk     => clk,
                rst_n   => rst_n,
                init    => load_seed,
                in_0    => feedback_bit,
                in_1    => seed(i),
                q       => q_out(i)
            );
        end generate FIRST;

        -- all other internal stages, including the last one
        INTERNAL: if i > 1 and i < N_stage+1 generate
            LFSR_elem_i: LFSR_elem
            port map (
                clk     => clk,
                rst_n   => rst_n,
                init    => load_seed,
                in_0    => q_out(i-1),
                in_1    => seed(i),
                q       => q_out(i)
            );
        end generate INTERNAL;
    end generate GEN;

    -- combinational logic
    -- the feedback bit is generated through the xor operations of the taps
    feedback_bit <= ((((( q_out(15) xor q_out(13) )
                                    xor q_out(9) )
                                        xor q_out(8) )
                                            xor q_out(7) )
                                                xor q_out(5) );
    PN_code <= q_out;

end beh;
```

Figure 9 - VHDL code for PNSG

The **PNSG** module is composed of 15 LFSR_elem in a chain, and additional logic.

Hence, we introduce 2 internal signals: a single bit **feedback_bit** signal and a **q_out** vector signal of 15 bit. The former is a support signal for the result of the combinational logic (the XOR operation of the taps), the latter is a support signal vector for the output bits of the PN_code, signals also taken as input of the XORs operator (the bits corresponding to the taps).

More in details, the **first** LFSR_elem takes as input **in_0** the **feedback_bit**, because corresponding to the first stage of the PNSG, while for the other stages **i**, the input **in_0** is connected to the output **q_out(i-1)** of the previous stage. Instead, for all stages, the input **in_1** is connected to the corresponding bit **seed(i)**.

As described before the **feedback_bit** is computed through a sequence of xor operations.

In the last part we can saw that the vector **q_out** is connected to the output vector **PN_code**.

Test-plan and Testbench

In order to check the correctness of the designed Pseudo Noise Sequence Generator system, some tests were performed via the *ModelSim* tool to simulate the circuit, and a Python script to verify that the sequence of PN_codes generated by the simulation in ModelSim was correct.

Testbench for the PNSG

To validate the PNSG circuit a simple tb code was implemented, as shown in the following Figures 10-11.

```
5
6   entity PNGS_tb is
7   end PNGS_tb;
8
9   architecture beh of PNGS_tb is
10
11      -- file for output sequence -> for checking correctness
12      file PN_CODE_OUT_FILE: text is out "PN_code_out_file2.txt";
13
14      -- constants
15      constant T_CLK: time := 10 ns;      -- clock period
16      constant T_RST: time := 5 ns;       -- period before the reset deassertion
17      constant N_STAGE: positive := 15;   -- number of PNSG stages
18      constant MAX_PERIOD: positive := 32767; -- ma period before the pn_code repeats: 2^15 - 1
19
20      -- signals
21      signal clk_tb: std_logic := '0';    -- clock signal, initialized to '0'
22      signal rst_n_tb: std_logic := '0';  -- reset signal, active low
23      signal load_tb: std_logic := '1';   -- initialization signal, active high, to load the seed
24      signal seed_tb: std_logic_vector(1 to N_STAGE) := "101010101010101";    -- initial seed for testing
25      signal PN_code_tb: std_logic_vector(1 to N_STAGE);  -- output signal vector
26      signal testing: boolean := true;
27      signal t: integer := 0;
28
29
30      -- dut
31      component PNSG is
32
33          generic (
34              N_stage : positive := 15
35              );
36
37          port (
38              clk         : in std_logic;     -- clock signal
39              rst_n       : in std_logic;     -- reset signal, active low
40              load_seed   : in std_logic;     -- control signal to load the initial seed in the registers
41              seed        : in std_logic_vector(1 to N_stage);    -- initial seed of the generator
42              PN_code     : out std_logic_vector(1 to N_stage)    -- generated output sequence
43          );
44
45      end component PNSG;
```

Figure 10 - PNSG_tb architecture description

```
    rst_n_tb <= '1' after T_RST;    -- deasserting the reset after T_RST ns

    PNSG_dut: PNSG

    generic map (
        N_stage => N_STAGE
    )

    port map (
        clk => clk_tb,
        rst_n => rst_n_tb,
        load_seed => load_tb,
        seed => seed_tb,
        PN_code => PN_code_tb
    );

    stimuli: process (clk_tb, rst_n_tb)

        -- variable used to count the number of clock cycles after the reset,
        -- to check when the pn_code starts to repeat
        variable clk_counter: integer := 0;

        variable out_bit_code: line;
        variable out_stream_code: std_logic_vector(1 to N_STAGE);

    begin

        if (rst_n_tb = '0') then
            clk_counter := 0;       -- reset the clock counter
            seed_tb <= "101010101010101";
            load_tb <= '1';     -- to set the initial value of the seed
        elsif (rising_edge(clk_tb)) then
            load_tb <= '0';     -- stop initialization phase
            t <= clk_counter;

            if (PN_code_tb = seed_tb and clk_counter > 1) then
                -- when the PN_code in output is equal to the initial seed
                -- it means a new period starts (periodicity of the sequence)
                testing <= false;
            elsif (clk_counter >= 1) then
                -- write the pn_code sequence of each step into an output file
                -- also the initial seed is included in the file
                WRITE(out_bit_code, PN_code_tb, right, N_STAGE);
                WRITELINE(PN_CODE_OUT_FILE, out_bit_code);

            end if;

            -- update the clk_counter
            clk_counter := clk_counter + 1;

        end if;

    end process;
```

Figure 11 - PNSG_tb architecture-process

Test on Modelsim

To simulate the behavior of the designed PN generator, the previously developed VHDL code was added and compiled in the ModelSim tool. Figures 12 and 13 below show the compilation results and an example of the simulation results as appear in ModelSim.

```
# Loading project PNSG
# Compile of DFF.vhd was successful.
# Compile of LFSR_elem.vhd was successful.
# Compile of MUX.vhd was successful.
# Compile of PNSG.vhd was successful.
# Compile of PNSG_tb.vhd was successful with warnings.
# 5 compiles, 0 failed with no errors.
```

Figure 12 - ModelSim compilation results

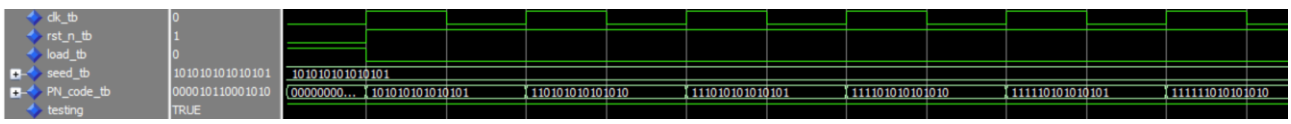


Figure 13 - Simulation results (example)

Python Test Script

To check if the PN_code sequence generated by the developed PNSG through the testbench was correct, a basic Python script was employed (the seed and other characteristics were hardcoded, just for testing).

```python
from pylfsr import LFSR
import difflib


seed = [1,0,0,0,0,0,0,0,0,0,0,0,0,0,1]
fpoly = [15, 13, 9, 8, 7, 5]
L = LFSR(fpoly=fpoly, initstate=seed, verbose=True)
L.info()

with open("pnsg_output_py.txt", "w") as f:

    for _ in range(2**15 - 1):
        f.write(L.getState() + '\n')
        L.next()

with open("pnsg_output_py.txt", "r") as f1, open("../modelsim/PN_code_out_file.txt", "r") as f2:
    diff = difflib.unified_diff(f1.readlines(), f2.readlines())
    for line in diff:
        print(line)
    if len(list(diff)) == 0:
        print("\n output sequences are equals -> pnsg correct")
    else:
        print("output sequences are different -> pnsg wrong")


with open("pnsg_output_py.txt",...
Run:  pnsg_python_test
   output sequences are equals -> pnsg correct
```

Figure 14 - Python testing code and result

As shown in the previous Figure 14, the *LFRS* library in the `pylfsr` package was used to generate the PN_code sequence from Python. All generated codes were written in a text file, then the `difflib` library was used to compare this file with the output file generated during the simulation on ModelSim. The `difflib.unified_diff` method produces an object containing a list of all differences found. As we can see at the bottom of the figure, the outputs match because no different lines were printed.

Synthesis on Xilinx FPGA Zynq

The final step was to synthesize the designed PNSG on the Zynq FPGA family device, using the *Vivado* tool. The complete overview produced by Vivado was previously shown in Figure 3 and reported in a snippet in Figure 15 below.

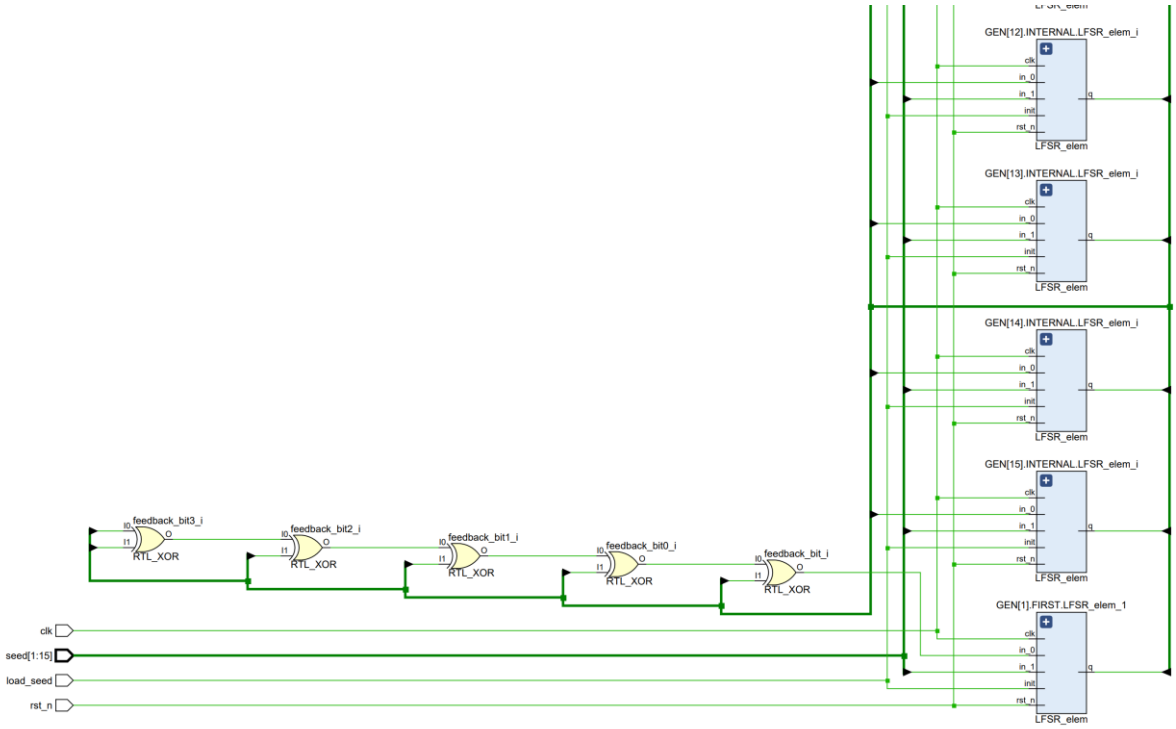


Figure 15 - Snippet of the RTL schematic

As already mentioned, the *LFSR element* shown in the figure is composed of a *Mutex* and a *D-FlipFlop*. The five XORs are connected in cascade, with one input corresponding to the output of the previous port and the second input the corresponding tap, except for the first port, which has the first two taps as inputs.

Analysis

After an initial RTL analysis, which produced the expected output circuit, the synthesis of the system was performed, obtaining the results shown in the following Figure 16.

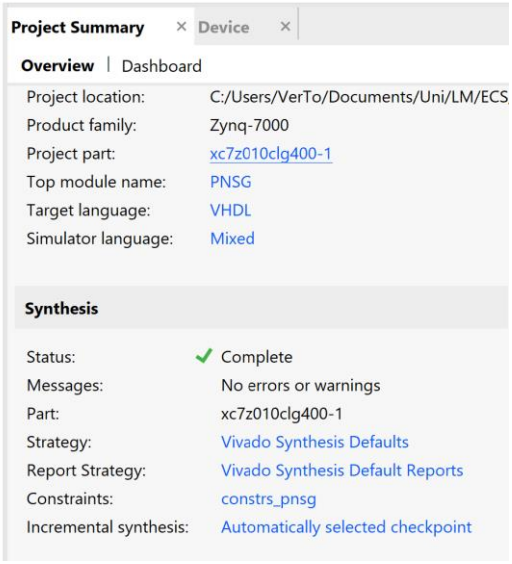
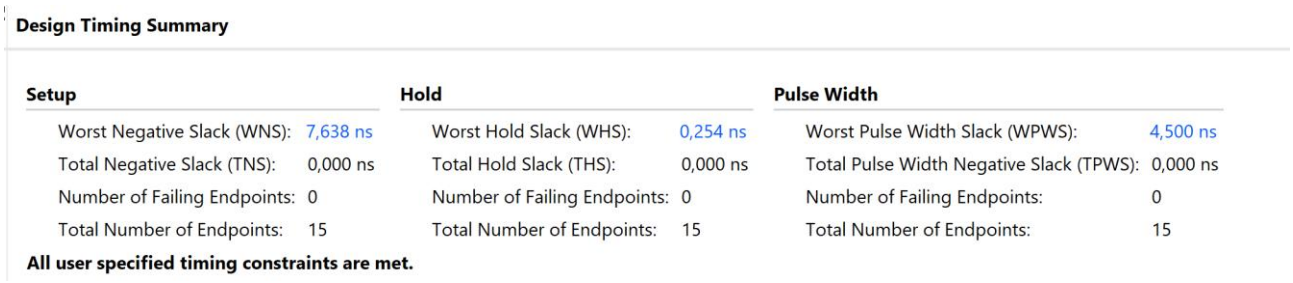


Figure 16 - Synthesis results

Since no errors or warnings were found, the next step was to run and analyze the synthesis introducing a timing constraint, more specifically the following:

```
create_clock -period 10.000 -name PNSG_clk -waveform {0.000 5.000} -add [get_ports clk]
```

This constraint specifies a period of 10ns, thus a clock frequency $f_{clk} = \frac{1}{10ns} = 100MHz$. With these values, the resulting WNS (Worst Negative Slack) obtained was $7.683\ ns$, as shown in Figure 17.



Design Timing Summary

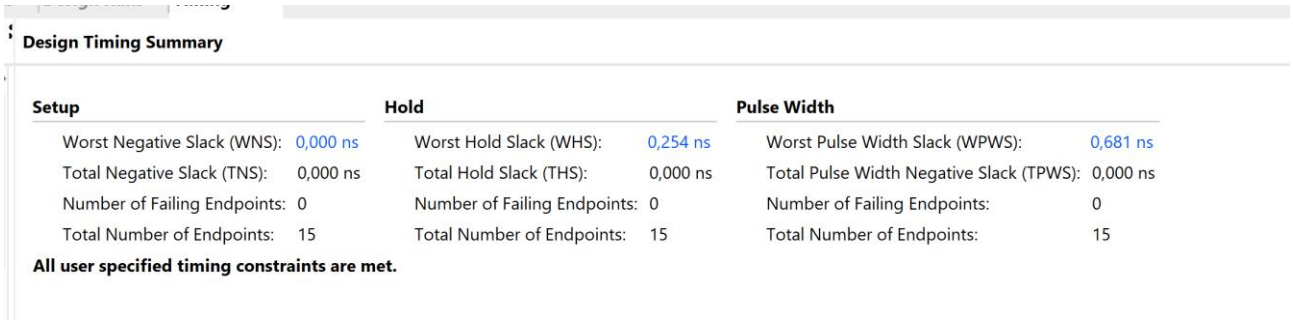
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	7,638 ns	Worst Hold Slack (WHS):	0,254 ns	Worst Pulse Width Slack (WPWS):	4,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	15	Total Number of Endpoints:	15	Total Number of Endpoints:	15

All user specified timing constraints are met.

Figure 17 - Timing summary

As it is a **positive** value, this means that the *optimal* clock period and the corresponding *maximum operating frequency* are:

$$T_{clk} = (10 - 7.683)ns = \mathbf{2.362ns} \text{ and } f_{clk} = \frac{1}{2.362} = \mathbf{423.37\ MHz.}$$



Design Timing Summary

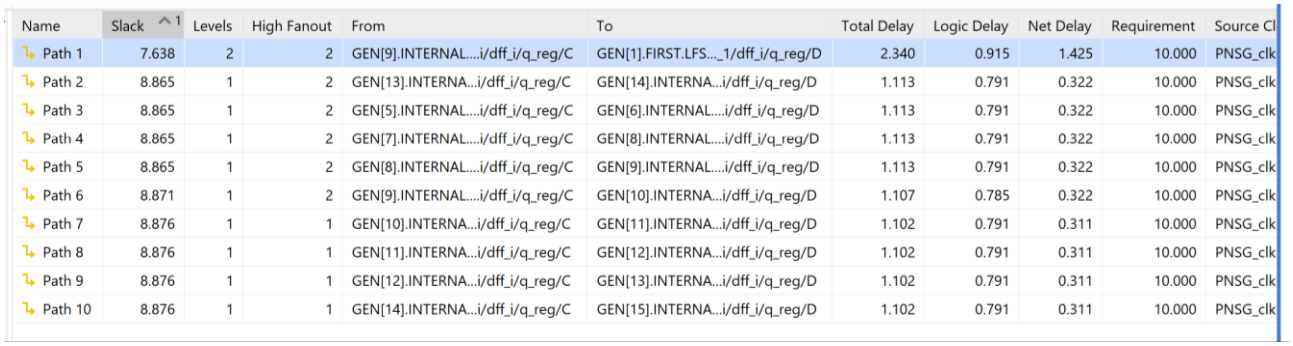
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,000 ns	Worst Hold Slack (WHS):	0,254 ns	Worst Pulse Width Slack (WPWS):	0,681 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	15	Total Number of Endpoints:	15	Total Number of Endpoints:	15

All user specified timing constraints are met.

Figure 18 - Timing summary with optimal clock

Critical path

Using a clock of $T_{clk} = 10ns$, the critical path results to be the path between the one shown in Figures 19-20-21 below.



Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Cl
Path 1	7.638	2	2	GEN[9].INTERNAL...i/dff_i/q_reg/C	GEN[1].FIRST.LFS..._1/dff_i/q_reg/D	2.340	0.915	1.425	10.000	PNSG_clk
Path 2	8.865	1	2	GEN[13].INTERNA...i/dff_i/q_reg/C	GEN[14].INTERNA...i/dff_i/q_reg/D	1.113	0.791	0.322	10.000	PNSG_clk
Path 3	8.865	1	2	GEN[5].INTERNAL...i/dff_i/q_reg/C	GEN[6].INTERNAL...i/dff_i/q_reg/D	1.113	0.791	0.322	10.000	PNSG_clk
Path 4	8.865	1	2	GEN[7].INTERNAL...i/dff_i/q_reg/C	GEN[8].INTERNAL...i/dff_i/q_reg/D	1.113	0.791	0.322	10.000	PNSG_clk
Path 5	8.865	1	2	GEN[8].INTERNAL...i/dff_i/q_reg/C	GEN[9].INTERNAL...i/dff_i/q_reg/D	1.113	0.791	0.322	10.000	PNSG_clk
Path 6	8.871	1	2	GEN[9].INTERNAL...i/dff_i/q_reg/C	GEN[10].INTERNA...i/dff_i/q_reg/D	1.107	0.785	0.322	10.000	PNSG_clk
Path 7	8.876	1	1	GEN[10].INTERNA...i/dff_i/q_reg/C	GEN[11].INTERNA...i/dff_i/q_reg/D	1.102	0.791	0.311	10.000	PNSG_clk
Path 8	8.876	1	1	GEN[11].INTERNA...i/dff_i/q_reg/C	GEN[12].INTERNA...i/dff_i/q_reg/D	1.102	0.791	0.311	10.000	PNSG_clk
Path 9	8.876	1	1	GEN[12].INTERNA...i/dff_i/q_reg/C	GEN[13].INTERNA...i/dff_i/q_reg/D	1.102	0.791	0.311	10.000	PNSG_clk
Path 10	8.876	1	1	GEN[14].INTERNA...i/dff_i/q_reg/C	GEN[15].INTERNA...i/dff_i/q_reg/D	1.102	0.791	0.311	10.000	PNSG_clk

Figure 19 - Timing paths report

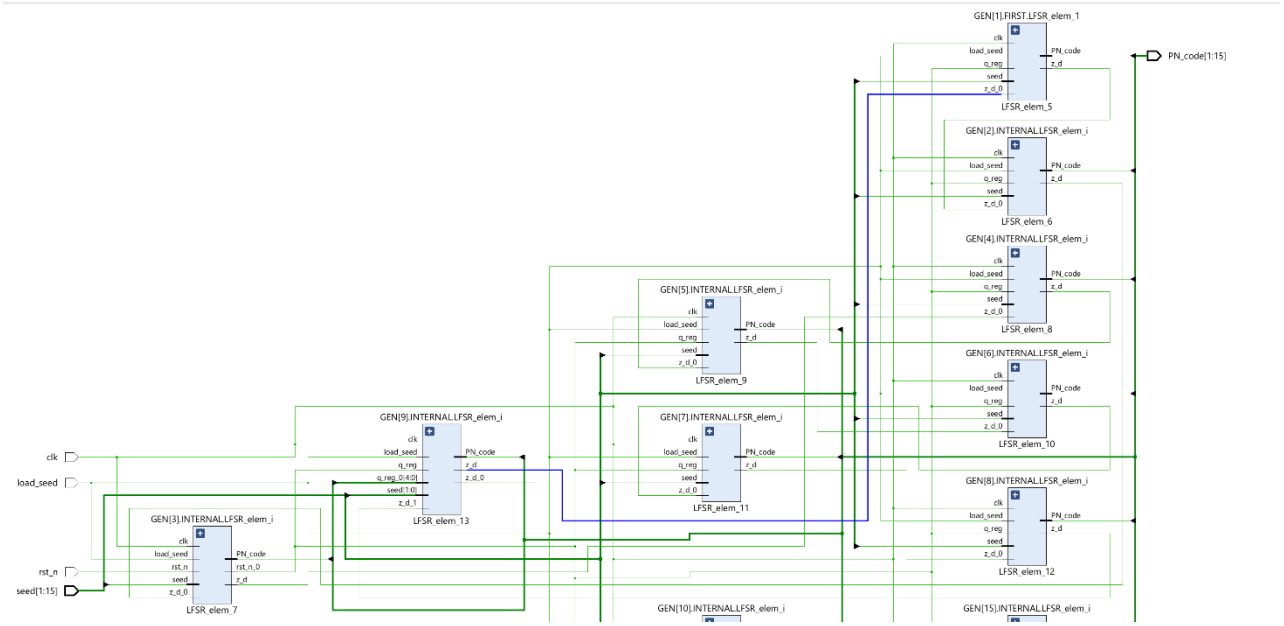


Figure 20 - Critical path (in blue) schematic general

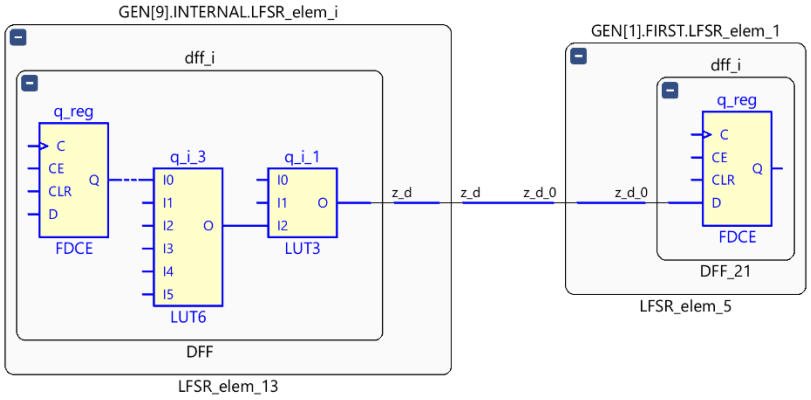


Figure 21 - Critical path schematic details

The same critical path is obtained by setting the clock to $T_{clk} = 2.36$, (below the optimum), as shown in Figure .

Design Runs | Timing

Intra-Clock Paths - PNSG_clk - Setup

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source
Path 1	-0.002	2	2	GEN[9].INTERNAL...i/dff_i/q_reg/C	GEN[1].FIRST.LFS..._1/dff_i/q_reg/D	2.340	0.915	1.425	2.360	PNSG_
Path 2	1.225	1	2	GEN[9].INTERNAL...i/dff_i/q_reg/C	GEN[10].INTERNA...i/dff_i/q_reg/D	1.113	0.791	0.322	2.360	PNSG_
Path 3	1.225	1	2	GEN[13].INTERNA...i/dff_i/q_reg/C	GEN[14].INTERNA...i/dff_i/q_reg/D	1.113	0.791	0.322	2.360	PNSG_
Path 4	1.225	1	2	GEN[5].INTERNAL...i/dff_i/q_reg/C	GEN[6].INTERNAL...i/dff_i/q_reg/D	1.113	0.791	0.322	2.360	PNSG_
Path 5	1.225	1	2	GEN[7].INTERNAL...i/dff_i/q_reg/C	GEN[8].INTERNAL...i/dff_i/q_reg/D	1.113	0.791	0.322	2.360	PNSG_
Path 6	1.225	1	2	GEN[8].INTERNAL...i/dff_i/q_reg/C	GEN[9].INTERNAL...i/dff_i/q_reg/D	1.113	0.791	0.322	2.360	PNSG_
Path 7	1.236	1	1	GEN[10].INTERNA...i/dff_i/q_reg/C	GEN[11].INTERNA...i/dff_i/q_reg/D	1.102	0.791	0.311	2.360	PNSG_

Figure 22 - Timing paths report, clock lower than optimal

Instead, the critical path after implementation is the path from element 13 to the first element with the optimal clock $T_{clk} = 2.362ns$ (as shown in Figure 23), and the path from element 7 to element 1 with the clock $T_{clk} = 10ns$ (as shown in Figure 24).

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requ
Path 1	0.649	2	2	GEN[13].INTERNA...i/dff_i/q_reg/C	GEN[1].FIRST.LFS..._1/dff_i/q_reg/D	1.660	0.704	0.956	
Path 2	0.859	1	1	GEN[4].INTERNAL....i/dff_i/q_reg/C	GEN[5].INTERNAL....i/dff_i/q_reg/D	1.448	0.642	0.806	
Path 3	0.869	1	2	GEN[13].INTERNA...i/dff_i/q_reg/C	GEN[14].INTERNA...i/dff_i/q_reg/D	1.438	0.580	0.858	
Path 4	0.899	1	2	GEN[8].INTERNAL....i/dff_i/q_reg/C	GEN[9].INTERNAL....i/dff_i/q_reg/D	1.456	0.642	0.814	
Path 5	0.911	1	1	GEN[3].INTERNAL....i/dff_i/q_reg/C	GEN[4].INTERNAL....i/dff_i/q_reg/D	1.444	0.642	0.802	
Path 6	0.954	1	1	GEN[10].INTERNA...i/dff_i/q_reg/C	GEN[11].INTERNA...i/dff_i/q_reg/D	1.401	0.580	0.821	
Path 7	0.986	1	2	GEN[9].INTERNAL....i/dff_i/q_reg/C	GEN[10].INTERNA...i/dff_i/q_reg/D	1.324	0.642	0.682	
Path 8	1.009	1	1	GEN[6].INTERNAL....i/dff_i/q_reg/C	GEN[7].INTERNAL....i/dff_i/q_reg/D	1.348	0.642	0.706	
Path 9	1.035	1	2	GEN[7].INTERNAL....i/dff_i/q_reg/C	GEN[8].INTERNAL....i/dff_i/q_reg/D	1.322	0.642	0.680	
Path 10	1.198	1	1	GEN[11].INTERNA...i/dff_i/q_reg/C	GEN[12].INTERNA...i/dff_i/q_reg/D	1.161	0.642	0.519	

Figure 23 - Timing paths report after implementation with optimal clock

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requ
Path 1	8.217	2	2	GEN[7].INTERNAL....i/dff_i/q_reg/C	GEN[1].FIRST.LFS..._1/dff_i/q_reg/D	1.774	0.700	1.074	
Path 2	8.372	1	2	GEN[9].INTERNAL....i/dff_i/q_reg/C	GEN[10].INTERNA...i/dff_i/q_reg/D	1.575	0.580	0.995	
Path 3	8.417	1	1	GEN[1].FIRST.LFS..._1/dff_i/q_reg/C	GEN[2].INTERNAL....i/dff_i/q_reg/D	1.530	0.716	0.814	
Path 4	8.506	1	1	GEN[14].INTERNA...i/dff_i/q_reg/C	GEN[15].INTERNA...i/dff_i/q_reg/D	1.441	0.580	0.861	
Path 5	8.547	1	2	GEN[7].INTERNAL....i/dff_i/q_reg/C	GEN[8].INTERNAL....i/dff_i/q_reg/D	1.398	0.580	0.818	
Path 6	8.665	1	2	GEN[5].INTERNAL....i/dff_i/q_reg/C	GEN[6].INTERNAL....i/dff_i/q_reg/D	1.282	0.580	0.702	
Path 7	8.697	1	1	GEN[3].INTERNAL....i/dff_i/q_reg/C	GEN[4].INTERNAL....i/dff_i/q_reg/D	1.248	0.580	0.668	
Path 8	8.707	1	1	GEN[2].INTERNAL....i/dff_i/q_reg/C	GEN[3].INTERNAL....i/dff_i/q_reg/D	1.241	0.580	0.661	
Path 9	8.707	1	1	GEN[11].INTERNA...i/dff_i/q_reg/C	GEN[12].INTERNA...i/dff_i/q_reg/D	1.238	0.580	0.658	
Path 10	8.710	1	1	GEN[6].INTERNAL....i/dff_i/q_reg/C	GEN[7].INTERNAL....i/dff_i/q_reg/D	1.238	0.580	0.658	

Figure 24 - Timing paths report (after implementation, clock 10 ns)

However, in both cases the critical path passes through element 9, as we can see in Figures 25 and 26 below.

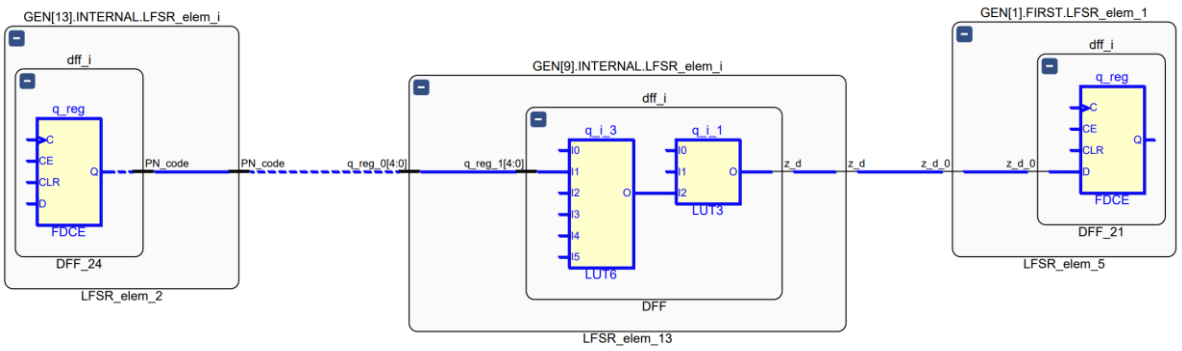


Figure 25 - Critical path schematic details (after implementation, optimal clock)

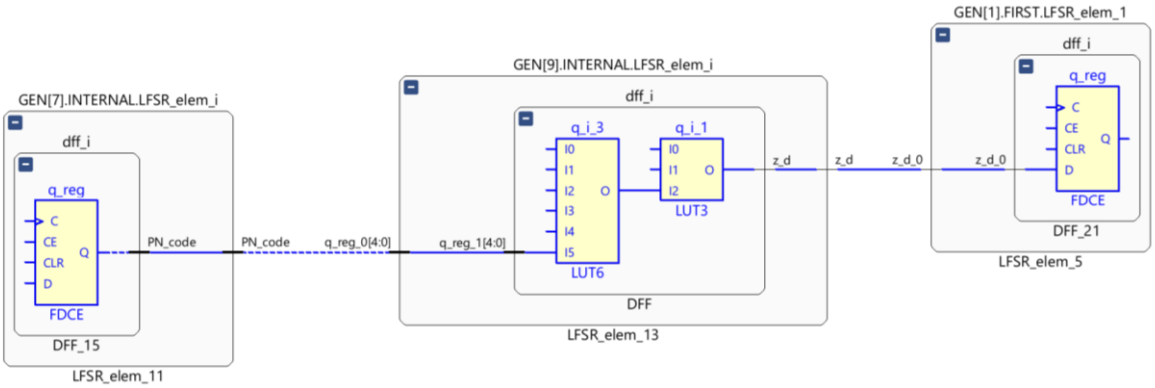


Figure 26 - Critical path schematic details (after implementation, clock 10 ns)

Resource utilization and power consumption

The resource utilization (equal post-synthesis and post-implementation) and power consumption of this system architecture is shown in the following Figures 27 and 28.

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

Graph | **Table**

Resource	Estimation	Available	Utilization %
LUT	16	17600	0.09
FF	15	35200	0.04

Power **Summary** | On-Chip

Total On-Chip Power:	**0.09 W**
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

Implemented Power Report

Figure 27 - Resource utilization and power consumption summary

Summary

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

Total On-Chip Power:	**0.09 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**26,0°C**
Thermal Margin:	59,0°C (5,0 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	11,5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	High

On-Chip Power

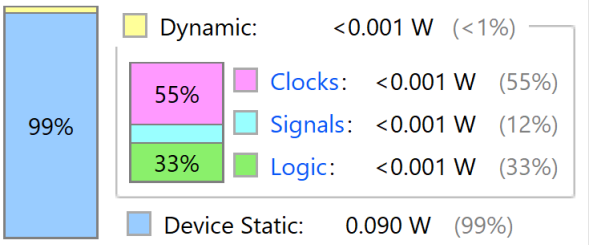


Figure 28 - On-Chip Power

As expected, the use of resources is quite low compared to those available (less than 1%), due to the simplicity of the operations to be performed and the simplicity of the circuit designed.

Similarly, the total on-chip power consumption is **0.09 W**, with a high imbalance between static and dynamic power: **99% static** and only **1% dynamic**. We can also observe that the main contribution to the dynamic power comes from the clocks (55%), followed by 33% from the logic.

Conclusions

The aim of this project was to design a Pseudo Noise Sequence Generator architecture on a Zybo-Zinq 7000, and all the results obtained are presented in this report.

In summary, we can state that with this simple implementation, we achieved the results expected. There are certainly other implementations, but quite similar to this one, e.g. with additional logic for a more secure use in cryptographic key generation, as mentioned before in the first section of the report.