



UNIVERSITÀ DI PISA

MASTER DEGREE IN EMBEDDED COMPUTING SYSTEMS

DIGITAL SYSTEMS DESIGN

PseudoNoise Sequence Generator project report

Professor

Luca Fanucci

Student

Arianna Gavioli

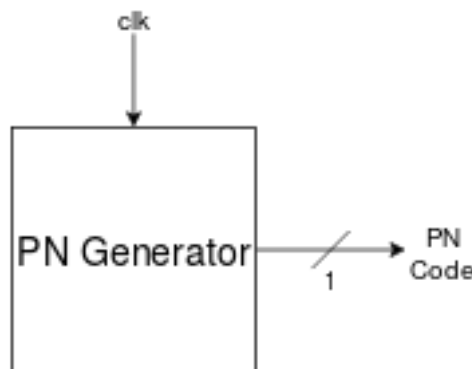
A.Y. 2018/2019

Abstract

The aim of this project is to implement a PseudoNoise Generator for CDMA transmissions using the VHDL language and synthesize it on the Zync FPGA. The generator must comply with the following specifications:

1. The generator must be a 15-stage PN generator;
2. The generator must follow the characteristic polynomial

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



In particular, a brief introduction to the use case of this generator is presented alongside to the correspondent circuit. After the discussion of the generator implementation in VHDL and its synthesis on the Zync FPGA, some tests are presented in order to verify the correct behavior of the designed circuit.

Pseudo-Random Noise Generator

A pseudo-random noise (**PRN**) is a signal that is similar to noise: its output must look like a random bitstream, so it has to satisfy one or more standard tests for statistical randomness. In reality, even though the PRN output looks like it hasn't any particular pattern, the pseudorandom noise actually follows a deterministic sequence of outputs. The quality of a generator is strongly linked to the randomness of its output.

A pseudo-random noise generator is an electronic device that is able to output a bit-stream, also referred as *key-stream*, that is a pseudo-random noise signal. The stream deterministically depends on a particular value, called *seed*, which is the actual random element. The generator uses an algorithm to expand this value in a longer sequence of bits. Because of this, not only the bit-stream is deterministically produced, but it also starts replicating itself after a certain number of steps, called period of the pseudo-random sequence. The latter depends on the algorithm used to implement the generator.

There are many possible applications for a pseudo-random generator. Some of them can be:

- **Cryptography:** an evident application of a pseudo-random generator in the field is generating keys used to encrypt and decrypt messages;
- **Simulation:** simulating a random event;
- **Neural Networks:** introduce randomness in the training of neural networks;

and many other scenarios in which an introduction of random elements is required.

PN Generator for CDMA transmissions

The particular PRN generator implemented for this project could be used for the IS-95 digital cellular technology. The latter was the first technology based on CDMA, which stands for *code division multiple access*, that is a digital radio system that allows the simultaneous transmission of different streams of bits from different mobile stations to a base station using the same channel.

In order to properly separate the streams of different users, a numerical mechanism involving the pseudo-random noise sequences is used. In particular, each mobile station shares a different key (seed), which is only a 15-bit word, with the base station. Both of the terminals are equipped with the same pseudo-random noise generator, so that once the key is shared, both the mobile and base stations are able to produce the same pseudo-random sequence. Once the mobile station starts sending a data-stream to the base station, the signal is sent in convolution with the pseudo-random sequence produced by the generator. The produced signals respect some particular properties that allow the base station to properly separate the different data stream.

Structure of the PN Generator

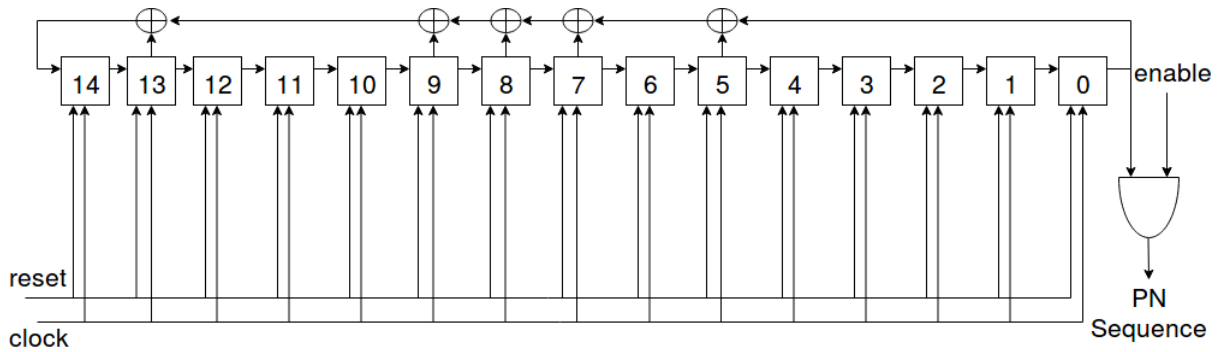
In this section, the structure of the pseudo-random generator used for the project is presented. The general block diagram is the following:



The reference architecture for the generator consists of a 15 positions linear-feedback shift register: the characteristic polynomial is

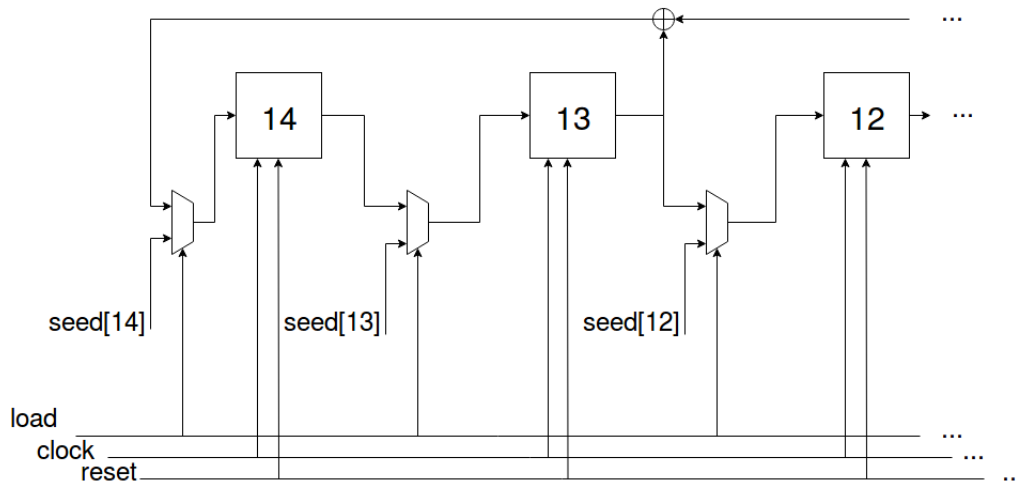
$$x^{15} + x^{13} + x^9 + x^8 + x^7 + x^5 + 1$$

and it represents the structure of the feedback. In a linear-feedback shift register, the input bit is the result of an xor function involving the previous state: not all the bits are considered in this operation and the characteristic polynomial identifies which bits to feedback to produce the new input bit. In a more specific way, the monomials of the polynomial determine which taps to open, where when a tap is opened the corresponding bit is put in the feedback loop.



As showed in the picture, the output stream of the generator is provided by the value of the stage 0.

In the previous scheme, two signals were excluded to make the image more readable altogether. A more specific and clearer overview of a portion of the generator, including the additional signals load and seed, can be seen in the following image:



The seed of the generator consists in the initial value of the register, from which the output stream starts to evolve: the seed is not embedded in the code, but the possibility to dynamically update it is provided by the signal inputs load and seed. If the signal load is set to '1', then the i -th flip flop of the register is overwritten with the i -th bit value of the signal seed, which is a 15-bit word that has to be opportunely set. This is done through a binary multiplexer that precedes each flip flop of the shift register. If load is low ('0'), $\forall 0 \leq i \leq 13$ each multiplexer shifts the value of the $(i+1)$ -th flip flop into the i -th. The multiplexer preceding the 14-th flip flop conveys the result of the xor operation into it. If the signal load is high ('1'), then the i -th

binary multiplexer provides the i -th value of the seed word as input of the i -th flip flop of the register.

The output signal is flattened to 0 whenever the signal enable is set to '0', and is reset to the actual value of the output when enable is set to '1'.

Periodicity

The possible values storable in the register are exactly 2^{15} , hence we expect that at a certain point there will be a repetition in the output stream. Since each state is a function of the previous one, from that step the sequence will start repeating periodically. Note that the state in which all the stages are set to 0 can be obtained only if in the previous state all of the stages are 0. Hence, if the seed contains some non-zero bit, it is not possible to reach a state in which all the stages are 0. This implies that the maximum possible period is $2^{15} - 1$, since we have to exclude the zero vector.

It is well known that if the characteristic polynomial is irreducible of degree n , then the period is a divisor of $2^n - 1$ and it is exactly $2^n - 1$ when the polynomial is *primitive* (primitive polynomials are well known and are listed). In our case, the characteristic polynomial is irreducible and primitive, so that we expect the period to be $2^{15} - 1 = 32767$. A test of this property is provided below.

Implementation of the PN Generator

In this section, the VHDL implementation of the previously described generator is showed.

The first basic elements needed are the flip flop, which will store the value of the i -th stage of the generator, and the binary multiplexer, that will filter the proper input to its correspondent flip flop based on the value of the signal 'load', as discussed in the previous section.

dff.vhd:

```
entity dff is
  port (
    clk      : in std_logic;
    rstn     : in std_logic;
    din      : in std_logic;
    qout     : out std_logic
  );
end dff;
architecture rtl of dff is
begin
  dff_proc : process(clk, rstn)
  begin
    -- If reset then the output is
    -- reset to zero
    if (rstn = '0') then
```

```

                                qout <= '0';
                                -- Otherwise, if the clock strikes,
                                -- the output is updated
                                elsif(rising_edge(clk)) then
                                    qout <= din;
                                end if;
                            end process dff_proc;
end rtl;

```

binMux.vhd:

```

entity binMux is
    port (
        in1      : in std_logic;
        in2      : in std_logic;
        sel      : in std_logic;
        output   : out std_logic
    );
end binMux;
architecture bhv of binMux is
begin
    mux_process: process(in1,in2,sel)
    begin
        -- In the first case, if sel is 0 the mux
        -- conveys the first input signal
        if(sel = '0') then
            output <= in1;

        -- If sel is 1 the mux conveys the second
        -- input signal
        else
            output <= in2;
        end if;
    end process;
end bhv;

```

Here is the implementation of the generator: it links the flip flops and the multiplexers respecting the architecture that has been previously described.

PNgen.vhd:

```

entity PNgen is
    port (
        clk      : in std_logic;
        resetn   : in std_logic;
        load     : in std_logic;
        enable   : in std_logic;
        seed     : in std_logic_vector(14 downto 0);
        stream   : out std_logic
    );
end PNgen;

```

```

architecture rtl of PNgen is
component dff
    port (
        clk          : in std_logic;
        rstn         : in std_logic;
        din          : in std_logic;
        qout         : out std_logic
    );
end component;
component binMux
    port (
        in1          : in std_logic;
        in2          : in std_logic;
        sel          : in std_logic;
        output       : out std_logic
    );
end component;
-- Each stage output
signal qouts       : std_logic_vector(14 downto 0);
-- Each stage input
signal dins        : std_logic_vector(14 downto 0);
-- Vector used to compute the feedback value
signal xors        : std_logic_vector(4 downto 0);
begin
    GEN: for i in 0 to 14 generate
        -- For each stage of the generator we need
        -- a multiplexer that conveys the right
        -- input signal and a flip flop that stores
        -- the value of the correspondent stage

        -- First stage receives in input either
        -- the seed bit or the feedback
        FIRST: if i = 0 generate
            MUX1: binMux port map(seed(i),load,dins(i));
            FF1: dff port map (clk,resetn,dins(i),qouts(i));
        end generate FIRST;

        -- All of the other stages receive in input
        -- either the seed bit or the previous stage output
        INTERNAL: if i > 0 and i < 14 generate
            MUXI: binMux port map(qouts(i-1),seed(i),load,dins(i));
            FFI: dff port map (clk,resetn,dins(i),qouts(i));
        end generate INTERNAL;

        LAST: if i = 14 generate
            MUXL: binMux port map(qouts(i-1),seed(i),load,dins(i));
            FFL: dff port map (clk,resetn,dins(i),qouts(i));
        end generate LAST;
    end generate GEN;

    stream <= (enable and qouts(14));

    -- The xor vector is used to build the feedback signal
    -- according to the carachteristic polynomial
    xors(0) <= qouts(14) xor qouts(9);
end architecture rtl;

```

```

xors(1) <= xors(0) xor qouts(7);
xors(2) <= xors(1) xor qouts(6);
xors(3) <= xors(2) xor qouts(5);
xors(4) <= xors(3) xor qouts(1);

end rtl;

```

As we can see, we generate 15 stages of the generator through a "for - generate" statement, and we link each of them to the output of their correspondent multiplexer (dins[i]). The multiplexers are linked accordingly to the structure previously described.

The value of the feedback signal is built through a series of xors: let's notice that the indices in the code are perfectly specular to the ones showed in the previous pictures, so the indexes of the feedback stages are also specular with respect to the characteristic polynomial. The output stream is the 14-th flip flop output, because of the specularity.

Tests and Results

To verify the correctness of the implemented PN generator, some tests were executed. Note that all the reported files and programs are present in the project folder.

First, a VHDL test-benchmark file was created:

PNgenTB.vhd:

```

entity PNgenTB is
end PNgenTB;

architecture test of PNgenTB is

    -----
    -- Testbench constants
    -----

    -- Clock period
    constant T_CLK : time := 10 ns;
    -- Period before the reset deassertion
    constant T_RESET : time := 10 ns;

    -----
    -- Testbench signals
    -----

    signal clk_tb : std_logic := '0';
    signal resetn_tb : std_logic := '0';
    signal load_tb : std_logic := '0';
    signal enable_tb : std_logic := '1';
    signal q_tb : std_logic;
    signal seed_tb : std_logic_vector(14 downto 0);
    signal end_sim : std_logic := '1';

    component PNgen is
        port (clk : in std_logic;
              resetn : in std_logic;
              load : in std_logic;
              enable : in std_logic;
              seed : in std_logic_vector(14 downto 0);
              stream : out std_logic);
    end component;

```



```

end component;

begin

clk_tb <= (not(clk_tb) and end_sim) after T_CLK / 2;
resetsn_tb <= '1' after T_RESET;

PNgen_test : PNgen
    port map(clk      => clk_tb,
             resetsn  => resetsn_tb,
             load     => load_tb,
             enable   => enable_tb,
             seed     => seed_tb,
             stream   => q_tb);

tb_proc: process(clk_tb, resetsn_tb)
    variable t : integer := 0;
    begin
        if(resetsn_tb = '0') then
            t := 0;
        elsif(rising_edge(clk_tb)) then
            case(t) is
                -- Initially setting up the seed
                when 1 => enable_tb <= '1';
                        load_tb <= '1';
                        seed_tb <= "0000000000000001";

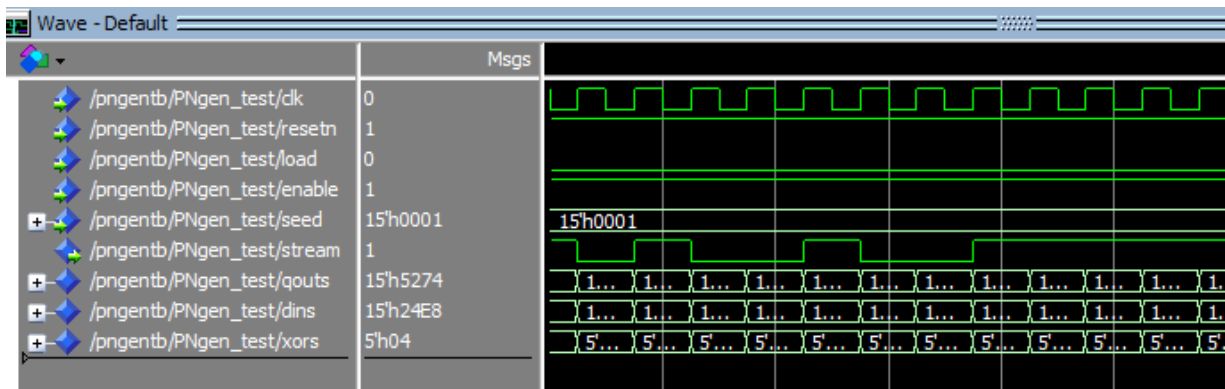
                -- Setting load to 0 since the
                -- seed has been loaded
                when 2 => load_tb <= '0';

                -- After 2^15-1 steps periodicity
                -- is reached
                when 32767 => end_sim <= '0';
                when others => null;
            end case;
            t := t+1;
        end if;
    end process;

end test;

```

This test-benchmark initially sets the seed inside the generator to a string '1000000000000000', and records the value of the output stream and the value of the internal stages of the register for exactly $2^{15} - 1$ steps.



We can't say much just looking at the waves: the output stream may appear random, but we can't know if it respects the properties of the PN generator and if it actually follows the right pattern.

To test it in more detail, the bit-stream has been recorded in a file called `bit_stream.txt`, included in the project directory, and inspected by a Python program. The latter simulates, through an algorithm, the output of the same PN generator with the same seed for the exactly same number of steps. The program is reported below:

`simulation.py`:

```
v0 = 15*[0]                # first vector memorized in the
v0[0] = 1                  # register (seed) = 0000000000000001

v_current = v0
step = 0

simulation = open("simulation.txt", 'w')

# We will write on the simulation file
# the output bit (v_current[14]) and number of step

simulation.write(str(step)+" "+str(v_current[14])+"\n")

'''
while the value in the register doesn't go
back to the initial value we can output a
pseudorandom bitstream: when we get back the
initial value, the bitstream will repeat itself
and we will have reached the number of
steps of periodicity
'''

while ( (v0 != v_current) or (step == 0) ) :
    step = step + 1

    # Bits from 0 to 13 are to be shifted to the
    # right, so the sequence is untouched
    slc = v_current[0:14]

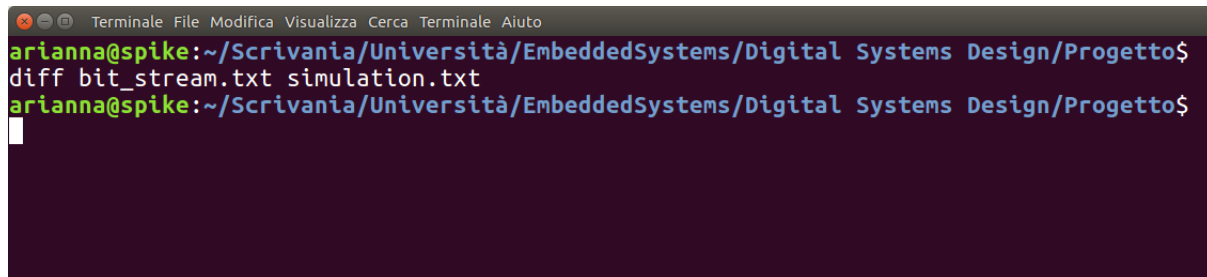
    # Bit 0 will be the result of the xor operation
    # done accordingly to the polynomial generator
    feed = v_current[1] + v_current[5] + v_current[6]
    feed = feed + v_current[7] + v_current[9] + v_current[14]
    feed = (feed%2)

    # Update of the register
    v_current = [feed] + slc

    # Append the output bit to the simulation file
    simulation.write(str(step)+" "+str(v_current[14])+"\n")

print (v_current, step)
```

A first test consists in verifying that the output of this program, saved in a file called `simulation.txt`, is the exact same output as the one produced by the VHDL implementation. This can be done by an efficient Linux command called `diff`: it compares two files and gives in output the different lines among them. Here's the output of the command `diff` run on the 2 produced files:

A terminal window with a dark background and light green text. The window title is "Terminale File Modifica Visualizza Cerca Terminale Aiuto". The prompt is "arianna@spike:~/Scrivania/Università/EmbeddedSystems/Digital Systems Design/Progetto\$". The command "diff bit_stream.txt simulation.txt" has been executed, and the output shows that the two files are identical, with no differences reported.

```
arianna@spike:~/Scrivania/Università/EmbeddedSystems/Digital Systems Design/Progetto$ diff bit_stream.txt simulation.txt
arianna@spike:~/Scrivania/Università/EmbeddedSystems/Digital Systems Design/Progetto$
```

As showed in the image, the two files coincide, so the implementation has succesfully passed the test.

Since the periodicity of the generator is at most $2^{15} - 1$, as discussed previously, and the VHDL and the Python implementations coincide, we can conclude that the Python program simulates exactly the same generator previously implemented in VHDL, because it reproduces exactly $2^{15} - 1$ output bits that coincide with the ones of the VHDL generator. Another test we could run consists in seeing if and when the periodicity is verified and it has been done by inspecting the output of the Python generator, since it is equivalent to the VHDL one. This test has been run through the following program:

periodicity.py:

```
v0 = 15*[0]                # first vector memorized in the
v0[0] = 1                  # register (seed) = 0000000000000001

v_current = v0
step = 0

'''
while the value in the register doesn't go back
to the initial value we can output a pseudorandom
bit-stream: when we get back the initial value,
the bitstream will repeat itself and we will
have reached the number of steps of periodicity
'''

while ( (v0 != v_current) or (step == 0) ) :
    step = step + 1

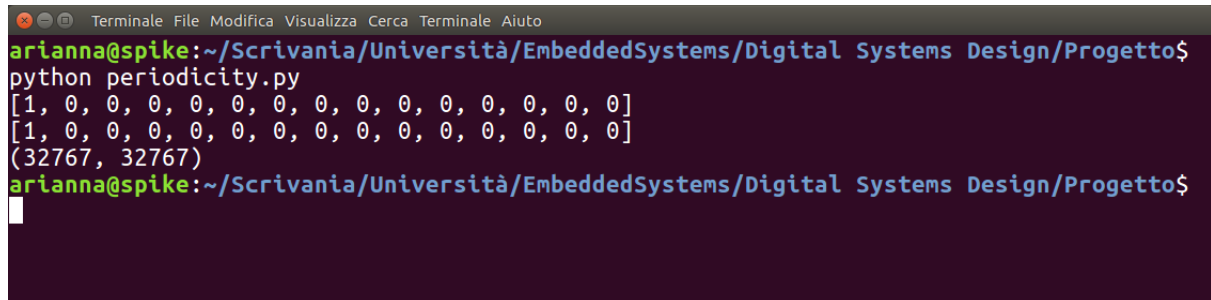
    # Bits from 0 to 13 are to be shifted to
    # the right, so the sequence is untouched
    slc = v_current[0:14]

    # Bit 0 will be the result of the xor operation
    # done accordingly to the polynomial generator
    feed = v_current[1] + v_current[5] + v_current[6]
    feed = feed + v_current[7] + v_current[9] + v_current[14]
    feed = (feed%2)

    # Update of the register
    v_current = [feed] + slc

print(v_current)
print(v0)
print(step, 2**15-1)
```

The algorithm that simulates the PN generator is exactly the same that was previously showed, but the generation of the bit-stream stops when the value stored in the register coincides with the initial value (seed), which is when the periodicity interval is reached. The output of the program provides the initial value of the register, the final value of the register and the number of steps that it took to reach periodicity, which is exactly $2^{15} - 1 = 32767$. The output of this program is:



```

arianna@spike:~/Scrivania/Università/EmbeddedSystems/Digital Systems Design/Progetto$
python periodicity.py
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
(32767, 32767)
arianna@spike:~/Scrivania/Università/EmbeddedSystems/Digital Systems Design/Progetto$

```

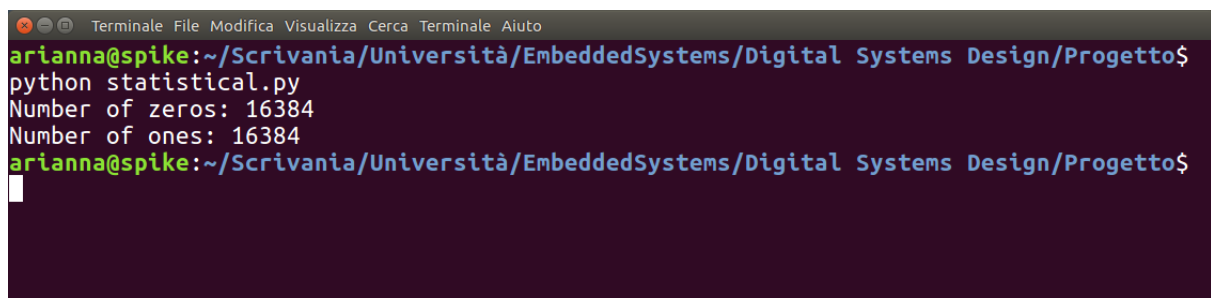
Lastly, we could see if the bit-stream respects some statistical tests; for example, we could count the number of ones and zeros inside the sequence and verify that they are more or less the same amount. This is done by the following program: `statistical.py`:

```

file = open("bit_stream.txt", 'r')
ones = 0
zeros = 0
for line in file:
    values = line.split(" ")
    if(int(values[1]) == 1):
        ones = ones + 1
    else:
        zeros = zeros + 1
print("Number of zeros: "+str(zeros))
print("Number of ones: "+str(ones))

```

The output of this program is:



```

arianna@spike:~/Scrivania/Università/EmbeddedSystems/Digital Systems Design/Progetto$
python statistical.py
Number of zeros: 16384
Number of ones: 16384
arianna@spike:~/Scrivania/Università/EmbeddedSystems/Digital Systems Design/Progetto$

```

Another test could consist in looking at patterns in pairs of bits, which means counting recurrences of the pairs (0,0), (0,1), (1,0), (1,1). They should be more or less the same.

The program that runs this test is the following:
statistical.py:

```
file = open("bit_stream.txt", 'r')

oneone = 0
zerozero = 0
zeroone = 0
onezero = 0

step = 0

for line in file:
    values = line.split(" ")

    if(step == 0):
        previous = int(values[1])
        step = step + 1
        continue

    if(int(values[1]) == 1):          #current bit value
        if(previous == 1):          #we have a couple 11
            oneone = oneone + 1
        else:                        #we have a couple 10
            onezero = onezero + 1

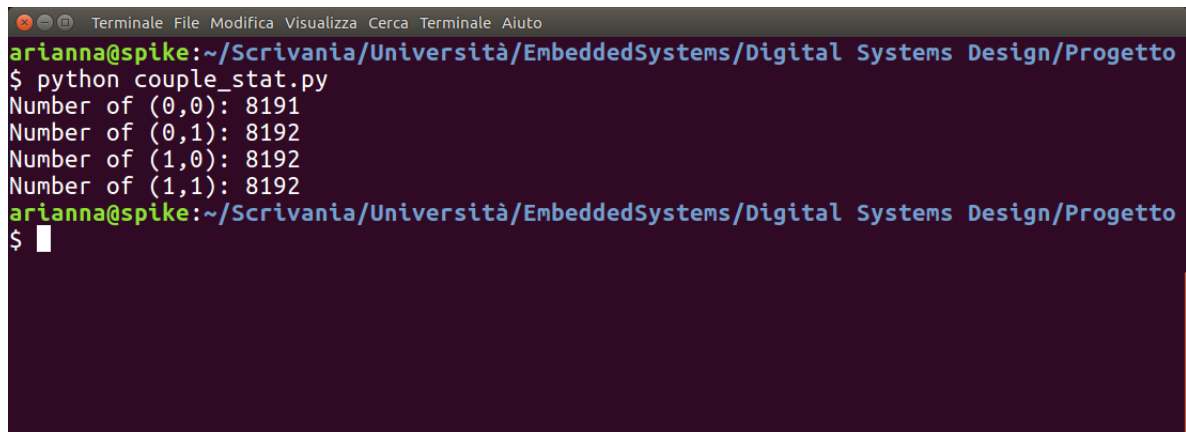
    if(int(values[1]) == 0):
        if(previous == 0):          #we have a couple 00
            zerozero = zerozero + 1
        else:                        #we have a couple 01
            zeroone = zeroone + 1

    step = step + 1

    #set previous for next iteration
    previous = int(values[1])

print("Number of (0,0): "+str(zerozero))
print("Number of (0,1): "+str(zeroone))
print("Number of (1,0): "+str(onezero))
print("Number of (1,1): "+str(oneone))
```

The output of this program is:



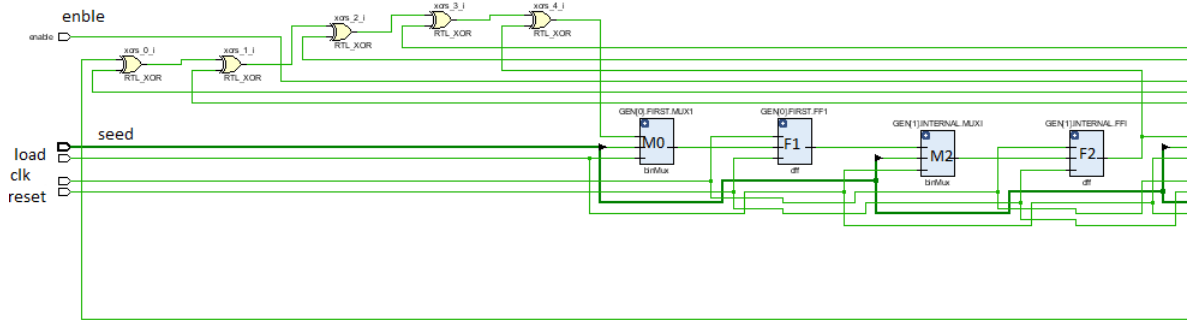
```
arianna@spike:~/Scrivania/Università/EmbeddedSystems/Digital Systems Design/Progetto
$ python couple_stat.py
Number of (0,0): 8191
Number of (0,1): 8192
Number of (1,0): 8192
Number of (1,1): 8192
arianna@spike:~/Scrivania/Università/EmbeddedSystems/Digital Systems Design/Progetto
$
```

We could keep making this kind of tests also for tuples of n elements and so on.

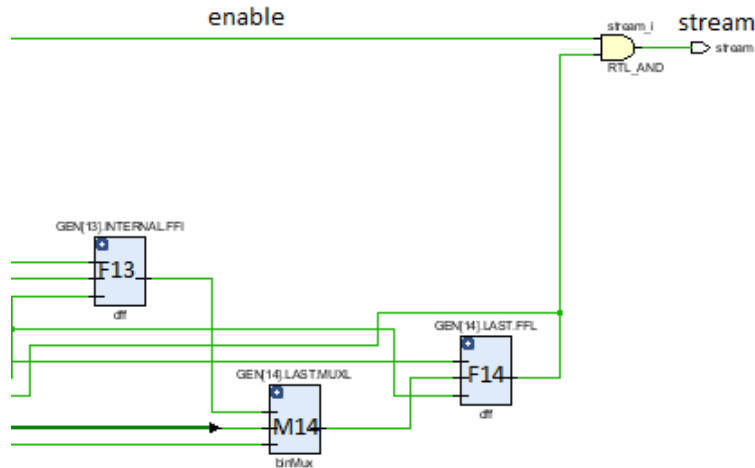
RTL Analysis and Synthesis

To synthesize the designed PN generator on the Zynq FPGAs family, Vivado tool was used.

An RTL (Register-Transfer-Level) analysis is automatically done by Vivado, the obtained schematic is correct, but difficult to report in the documentation due to its size and amount of signals to represent. A small portion of the schematic is presented: here we can see a portion of the feedback that is constituted by the xor elements put in series, and the first sequences of multiplexer-flip flop.



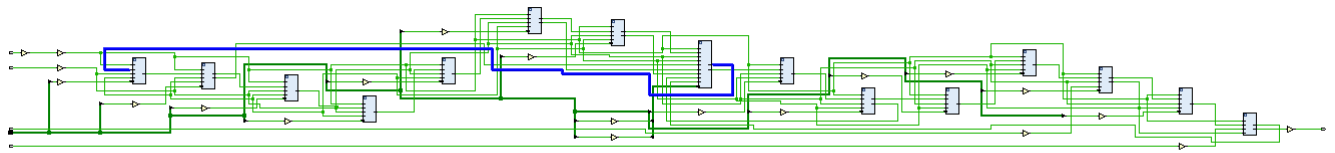
The output of the circuit is correctly set to the result of the enable signal and the value of the last flip flop.



After a first RTL analysis, the synthesis of the board has been run: the selected target frequency of the clock has been set up to $f^* = 100 MHz$. After the synthesis, the resulting WNS (Worst Negative Slack), computed by Vivado considering the critical path, is $7.549 ns$. Once the WNS is obtained, the maximum operating frequency can be computed as:

$$f_{clk}^{max} = \frac{1}{T_{clk}^* - WNS} \simeq 408 MHz$$

The WNS is computed taking into consideration the critical path, which bounds the maximum slack. Vivado identifies this path, and the critical path of this project has resulted in the path that links the output of the flip flop 7 to the input of the flip flop 0, which is one of the feedback signals. Here is the critical path highlighted in the synthesized schematic:



Timing x Package Pins I/O Ports							
Intra-Clock Paths - clk - Setup							
	Name	Slack	Levels	Routes	High Fanout	From	To
Path 1	GEN[7].INTERNAL.FFI/qout_reg/C	7.549	2	3	2	GEN[7].INTERNAL.FFI/qout_reg/C	GEN[0].FIRST.FF1/qout_reg/D
Path 2	GEN[9].INTERNAL.FFI/qout_reg/C	8.776	1	2	2	GEN[9].INTERNAL.FFI/qout_reg/C	GEN[10].INTERNAL.FFI/qout_reg/D
Path 3	GEN[1].INTERNAL.FFI/qout_reg/C	8.776	1	2	2	GEN[1].INTERNAL.FFI/qout_reg/C	GEN[2].INTERNAL.FFI/qout_reg/D
Path 4	GEN[5].INTERNAL.FFI/qout_reg/C	8.776	1	2	2	GEN[5].INTERNAL.FFI/qout_reg/C	GEN[6].INTERNAL.FFI/qout_reg/D
Path 5	GEN[6].INTERNAL.FFI/qout_reg/C	8.776	1	2	2	GEN[6].INTERNAL.FFI/qout_reg/C	GEN[7].INTERNAL.FFI/qout_reg/D
Path 6	GEN[7].INTERNAL.FFI/qout_reg/C	8.782	1	2	2	GEN[7].INTERNAL.FFI/qout_reg/C	GEN[8].INTERNAL.FFI/qout_reg/D
Path 7	GEN[10].INTERNAL.FFI/qout_reg/C	8.787	1	2	1	GEN[10].INTERNAL.FFI/qout_reg/C	GEN[11].INTERNAL.FFI/qout_reg/D
Path 8	GEN[11].INTERNAL.FFI/qout_reg/C	8.787	1	2	1	GEN[11].INTERNAL.FFI/qout_reg/C	GEN[12].INTERNAL.FFI/qout_reg/D
Path 9	GEN[12].INTERNAL.FFI/qout_reg/C	8.787	1	2	1	GEN[12].INTERNAL.FFI/qout_reg/C	GEN[13].INTERNAL.FFI/qout_reg/D
Path 10	GEN[13].INTERNAL.FFI/qout_reg/C	8.787	1	2	1	GEN[13].INTERNAL.FFI/qout_reg/C	GEN[14].LAST.FFL/qout_reg/D

And here's a small recall as a memo of the feedback scheme of the PNgen.vhd:

```

...
architecture rtl of PNgen is
...
begin
    ...
    -- The xor vector is used to build the feedback signal
    -- according to the carachteristic polynomial
    xors(0) <= qouts(14) xor qouts(9);
    -- HERE IS THE FEEDBACK OF THE CRITICAL PATH:
    xors(1) <= xors(0) xor qouts(7);

    xors(2) <= xors(1) xor qouts(6);
    xors(3) <= xors(2) xor qouts(5);
    xors(4) <= xors(3) xor qouts(1);

end rtl;

```

The Vivado tool also analyzed the used resources and the power consumption, that are reported below:

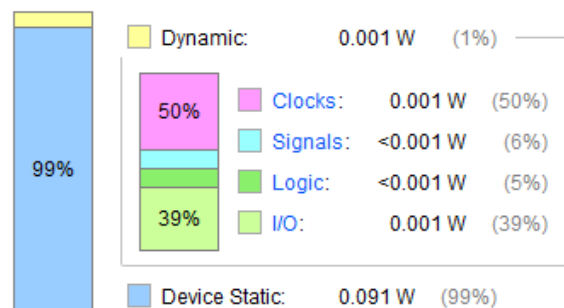
Name	Slice LUTs (17600)	Slice Registers (35200)	Bonded IOB (100)	BUFGCTRL (32)
▼ PNgen	17	15	20	1
GEN[9].INTERNAL.FFI (dff_13)	1	1	0	0
GEN[8].INTERNAL.FFI (dff_12)	1	1	0	0
GEN[7].INTERNAL.FFI (dff_11)	2	1	0	0
GEN[6].INTERNAL.FFI (dff_10)	1	1	0	0
GEN[5].INTERNAL.FFI (dff_9)	1	1	0	0
GEN[4].INTERNAL.FFI (dff_8)	1	1	0	0
GEN[3].INTERNAL.FFI (dff_7)	1	1	0	0
GEN[2].INTERNAL.FFI (dff_6)	1	1	0	0
GEN[1].INTERNAL.FFI (dff_5)	1	1	0	0
GEN[14].LAST.FFI (dff_4)	1	1	0	0
GEN[13].INTERNAL.FFI (dff_3)	1	1	0	0
GEN[12].INTERNAL.FFI (dff_2)	2	1	0	0
GEN[11].INTERNAL.FFI (dff_1)	1	1	0	0
GEN[10].INTERNAL.FFI (dff_0)	1	1	0	0
GEN[0].FIRST.FFI (dff)	1	1	0	0

Summary

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: 0.093 W
Design Power Budget: Not Specified
Power Budget Margin: N/A
Junction Temperature: 26,1°C
 Thermal Margin: 58,9°C (5,0 W)
 Effective θ_{JA} : 11,5°C/W
 Power supplied to off-chip devices: 0 W
 Confidence level: Low
[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

On-Chip Power



Conclusions

In conclusion, this project shows a possible implementation of a pseudo-random generator with respect to the telecommunication environment, but this exact implementation could be used also in very different fields, as discussed in the initial section of this report. This makes the implementation very versatile and exploitable in different contexts.

There exists many versions of pseudo-random generators that are similar to our study case, their implementation wouldn't be so different from the one presented.

References

- [1] Professor Luca Fanucci's and Assistant Gabriele Meoni's Digital Systems Design course slides.
- [2] Wikipedia
https://en.wikipedia.org/wiki/Linear-feedback_shift_register
<https://en.wikipedia.org/wiki/CdmaOne>
- [3] MathWorks
<https://it.mathworks.com/help/comm/ref/pnsequencegenerator.html>
- [4] Lawrence C. Washington
Introduction to Cryptography