University of Pisa

Convolutional code generator

Davide Rasla

Report for the project of digital system course

in

Embedded Computing System

June 21, 2019

Contents

List of Figures **ii**

1 Introduction **1**
1.1 Convolutional codes . . . 1
1.1.1 Possible field application . . . 2
1.2 Architecture with two shift register . . . 3
1.3 Architecture with one shift register . . . 3

2 Design and Implementation with two Shift Register **5**
2.1 Architecture Description and Data Flow . . . 5
2.2 VHDL Code . . . 7
2.3 Implementation under VIVADO . . . 10
2.3.1 Project Summary . . . 10
2.3.2 Critical path . . . 11
2.3.3 Max frequency . . . 12
2.3.4 Report LUT Utilization . . . 15
2.3.5 Report power utilization . . . 16

3 Design and Implementation with one Shift Register **17**
3.1 Architecture Description and Data Flow . . . 17
3.2 VHDL Code . . . 18
3.3 Synthesis under VIVADO . . . 19
3.3.1 Project summary . . . 20
3.3.2 Critical Path . . . 21
3.3.3 Max frequency . . . 22
3.3.4 Report LUT utilization . . . 23
3.3.5 Report power utilization . . . 24

4 Verification and Simulation of Architecture **25**
4.1 Simulation in C language . . . 26
4.2 Simulation with TestBench . . . 28
4.3 Wave on Model Sym . . . 29

Bibliography **31**

List of Figures

1.1 an overview of a convolutional code generator 2
1.2 General configuration taken from [1] 4

2.1 Logic with two shift register 6
2.2 VHDL description of entity generator with two shift register 7
2.3 PortMapping in GeneratorCode.vhd 8
2.4 Shift Register used for a_k 9
2.5 Shift Register used for C_k 9
2.6 Summary Report with two shift registers 10
2.7 Dashboard of the utilization 11
2.8 Summary of all paths 12
2.9 The critical path 12
2.10 Worst Negative Slack Value 12
2.11 Editing time constraint 14
2.12 New WNS with 2,271 as clock period 15
2.13 Summary of the utilization 15
2.14 Summary of power consumption 16

3.1 Logic with one shift register 17
3.2 VHDL description of entity generator with one register 18
3.3 PortMapping in GeneratorCode.vhd with one shift register 18
3.4 Shift Register 19
3.5 Summary Report with one shift register 20
3.6 DashBoard with one shift register 20
3.7 Summary of all paths 21
3.8 Critical path with one shift register 21
3.9 WNS with one shift register 22
3.10 New WNS with 1,575 as clock period 23
3.11 LUT utilization with one shift register 23
3.12 Power consumption with one shift register 24

4.1 Interface of the C simulator 26
4.2 A possible output with '1011 0110 0110 1101' 27
4.3 Compare the two outputs 27
4.4 VHDL description of TestBench 28

4.5 Reading from InputVhd.txt 29
4.6 Test with: '1011 0110 0110 1101' 29

Chapter 1

Introduction

The purpose of this document is to describe the realization of a generator of convolutional codes, implemented using VHDL and synthesized with Vivado Tool. Two architecture are described, taking into account the different performance among them.

1.1 Convolutional codes

Convolutional code generates a codeword of n symbols from some consecutive message blocks of k symbols. A convolutional encoder consists of k m-stage shift registers that perform some polynomial, like eq 1.2, to generate the codeword. The message symbols are fed into the shift registers k symbols at a time, as shown in figure 1.1. In this case the rate k/n is:

$$Rc = 1/2 \tag{1.1}$$

So, the linear polynomial generator generates an n-symbol block each time an input block is fed in. The number of message symbols from which an output block is generated is called the constraint length of the code, 11 in this case.

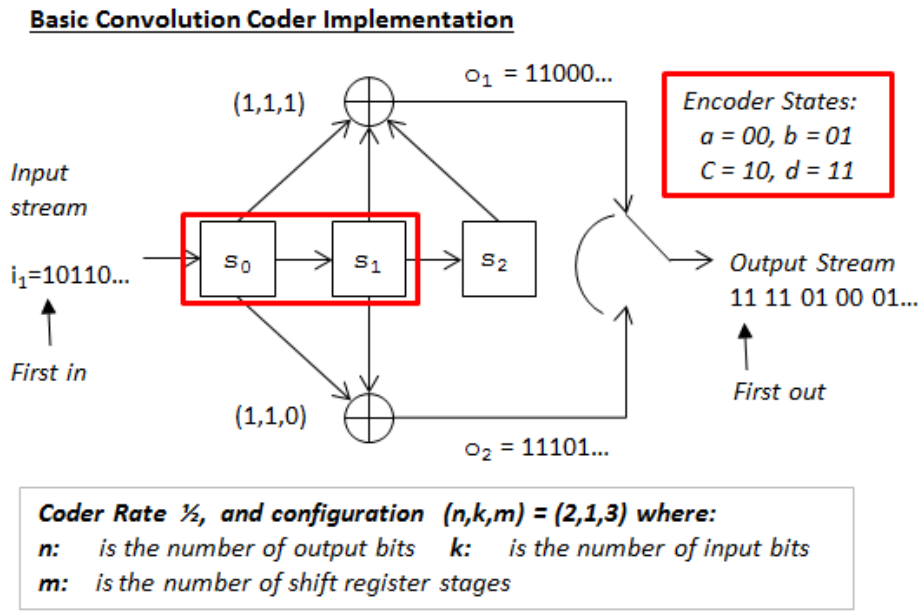

Figure 1.1: *an overview of a convolutional code generator*

1.1.1 Possible field application

Convolutional codes are used to obtain reliable data transfer in applications such as digital video transfer, radio, mobile telephony and satellite communications.

A field that includes them all is the spatial missions. For the NASA, just to say one spatial agency, convolutional code are essential in every mission. For example, Voyager 1 and 2 used a rate (2, 7) convolutional code, while Mars Exploration Rover and Pathfinder used a (6, 15) code.

The aim is to designing a protocol to communicate with a robot 12 billion miles away from earth. Of course must be accepted that the message will likely arrive garbled, noisy, and weak. One solution could be try to send the same message over-and-over again, hoping that at least one of your messages will make it through unscathed; but, how many times is enough? And how recognize the errors? Another example, back on earth, could be communicating with a satellite that goes across the globe. No matter how the antenna is designed , the messages will arrive garbled, noisy, and weak.
The solution is to implement robust encoding and decoding algorithms like, convolutional codes.

1.2 Architecture with two shift register

In this architecture two separate shift registers are used to save the states of the input ***ak*** and the feedback ***ck*** over time.

In particular, remembering the generating polynomial:

$$ck = ck_{-8} + ck_{-10} + ak + ak_{-3} + ak_{-4} \tag{1.2}$$

It's possible to use two separate shift registers to store respectively the input from the bitstream of the encoding, i.e. ak, and the recursive one that is ck. In addition to the shift registers, four xor ports are used to add the values and obtain the ck output.

Note also that, from the generator polynomial, it is necessary to take specific outputs states and therefore the number of logical gates can be reduced considering only the values required by the polynomial. For these reasons only two wires are used to take the values from each needed shift register.

More details will be provided in chapter 2

1.3 Architecture with one shift register

This implementation comes from the observation that it is not necessary to use two separate shift registers because, with reference to [1], the generator polynomial

$$ck = ck_{-8} + ck_{-10} + ak + ak_{-3} + ak_{-4} \tag{1.3}$$

can be divided into two parts.

The first part is called feedback polynomial, that makes the encoder recursive, and is defined as:

$$g(x) = \sum_{k=0}^{N} g_{\mathrm{i}} * x^{i} \tag{1.4}$$

while the second part is called the feed-forward polynomial and is defined as:

$$h_{\mathrm{k}}(x) = \sum_{k=0}^{N} h_{\mathrm{ki}} * x^{i} \tag{1.5}$$

So, the same shift register can be used in order to obtain feedback and feed forward simultaneously.
The figure below explain this concept. Once L is calculated, as the maximum degree between h(x) and g(x) (in this case 10), L flipflop are connected forming a shift register. When the coefficient g_i is equal to 1, the output of the flip-flop f_i, is taken in feedback. In the same way when h_i is equal to 1, the output of the flip-flop d_i is taken in feedforwarding.

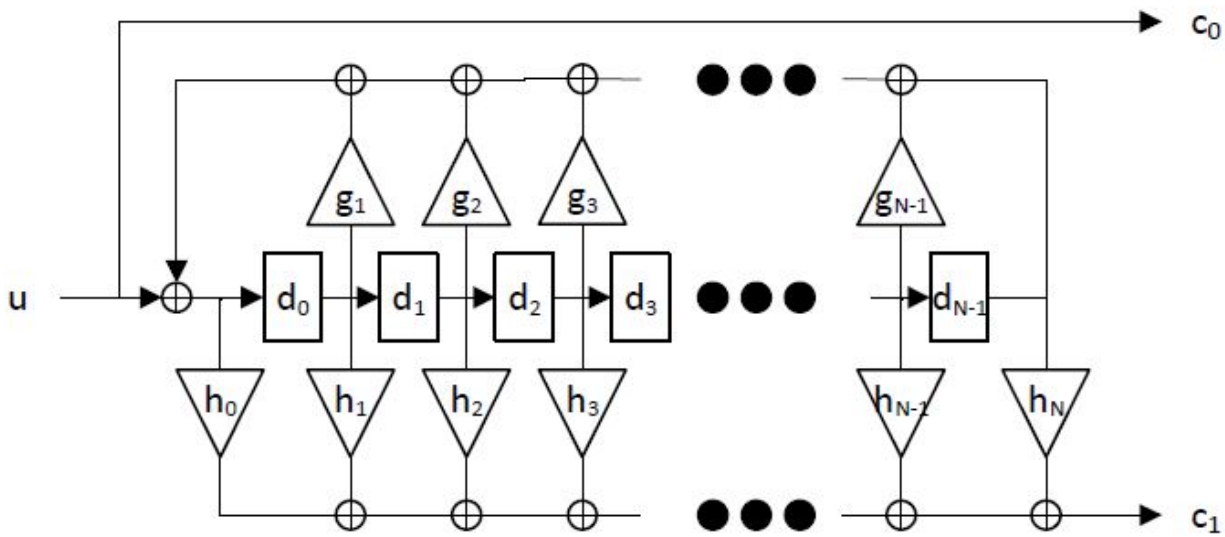

Figure 1.2: *General configuration taken from [1]*

More details will be provided in chapter 3 where will be explained how has been derived the correct implementation for this specific application from the general version shown in figure 1.2

Chapter 2

Design and Implementation with two Shift Register

A possible simple implementation of the generator polynomial has shown in this chapter. In particular are described the architecture, the vhdl modules, the relative tests with a test bench.

2.1 Architecture Description and Data Flow

Regarding this part, a graphical view is shown and described in order to present the architecture structure that, as mentioned before, is formed by two shift registers and four xor ports.

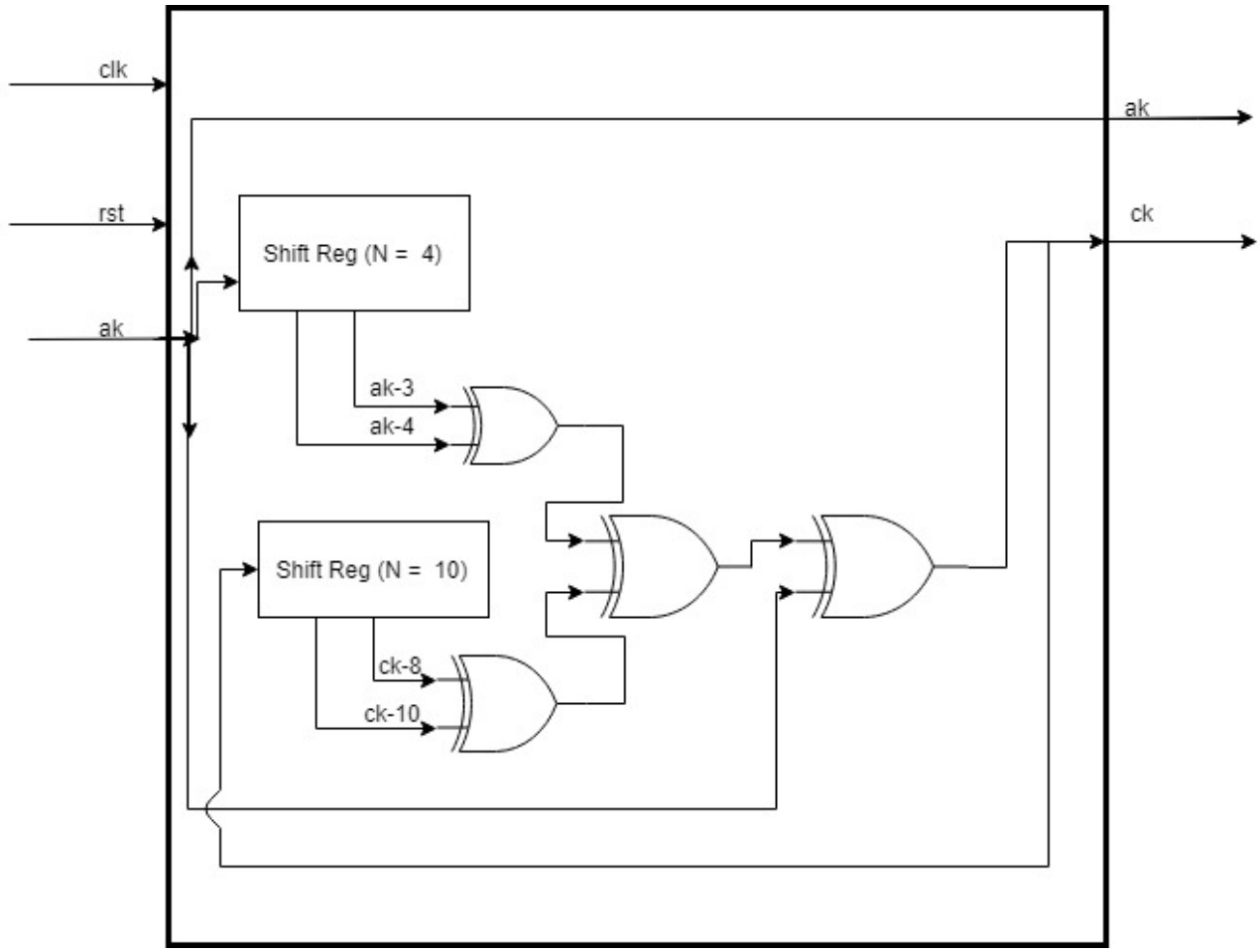

Figure 2.1: *Logic with two shift register*

In figure 2.1 there is shown that:

- The first shift stores the last four states of the a_k input
- The second shift stores the last ten states of the c_k input

from each shift register are taken in output two signals, in number equal to the number of terms of the sum of the polynomial, i.e a_{k-3} and a_{k-4} from the first one and c_{k-8} and c_{k-10} from the second one.

The signals are xored between them and with the input a_k in order to calculate c_k which will then be taken in feedback into the second shift register.

2.2 VHDL Code

The VHDL implementation of the generator is inside GeneratorCode.vhd.
More in detail has been implemented:

- GeneratorCode.vhd: that describes the entity generator.
- ShiftRegisterAK.vhd: that describes the shift register used to memorize the last 5 past state of a_k.
- ShiftRegisterCK.vhd: that describes the shift register used to memorize the last 10 past state of c_k.

```
entity ConvolutionalGenerator is
    port  (
        clk                 : in std_ulogic;                -- clock signal
        reset               : in std_ulogic;                -- reset signal
        ak_inputGen         : in std_ulogic;                -- the bit-stream is taken bit a bit
        ak_OutputGen        : out std_ulogic;               -- the ak output, the same in input
        ck_outGen           : out std_ulogic                -- the result of coding
    );
end ConvolutionalGenerator;

architecture struct of ConvolutionalGenerator is

    component ShiftRegAK
       generic(N_bit : integer := 5);

       port(
            ak      : in std_logic;                         -- input ak in the shift register, in feedforward
            q       : out std_logic_vector(1 downto 0);     -- output from two FF in order to take ak-3 and ak-4
            clk     : in std_logic;                         -- clock signal
            a_rst_n : in std_logic                          -- reset signal
    );
    end component;

    component ShiftRegCK is
   generic(N_bit : integer := 10);
       port(
            ck       : in std_logic;                        --  input ck in the shift register, in feedback
            q       : out std_logic_vector(1 downto 0);     --  output from two FF in order to take ck-8 and ck-10
            clk     : in std_logic;                         --  clock signal
            a_rst_n : in std_logic                          --  reset signal
        );
    end component;
```

Figure 2.2: *VHDL description of entity generator with two shift register*

In figure 2.2 the output q present in the 2 shift register is used to extract the outputs from the flip flops used to complete the polynomial.

In particular, for example, in the shiftRegAk are taken in output:

$$q(0) <= q_s(2) \tag{2.1}$$

$$q(1) <= q_s(3) \tag{2.2}$$

where $q_s(2)$ and $q_s(3)$ represent ak_{-3} and ak_{-4} respectively.

The same goes for the other shift register where now:

$$q(0) <= q_s(7) \tag{2.3}$$

$$q(1) <= q_s(9) \tag{2.4}$$

where $q_s(7)$ and $q_s(9)$ represent ck_{-8} and ck_{-10} respectively.

```
49    begin
50
51          i_ShiftAk: ShiftRegAK
52                  port map(
53                  ak                  =>ak_inputGen,
54                  clk                 => clk,
55                  a_rst_n             => reset,
56                  q                   => outShiftAK
57                          );
58
59          i_ShiftCk: ShiftRegCK
60                  port map(
61                  ck                  =>ck_out,
62                  clk                 => clk,
63                  a_rst_n             => reset,
64                  q                   => outShiftCK
65                                     );
66
67  XorAK <= outShiftAK(0) xor outShiftAK(1);
68  XorCK <= outShiftCK(0) xor outShiftCK(1);
69
70  ck_out          <=(XorAK xor XorCK) xor ak_inputGen;
71  ck_outGen       <= ck_out;
72  ak_OutpuTgen    <= ak_inputGen;
73
74  end struct;
```

Figure 2.3: *PortMapping in GeneratorCode.vhd*

Finally, as shown in figure 2.3 at the line 70, the polynomial (1.2) is calculated performing:

$$ck_{\text{out}} <= (XorAK\ xor\ XorCK)\ xor\ ak\_inputGen \tag{2.5}$$

where :

$$XorAK = q_s(2)\ xor\ q_s(3) \tag{2.6}$$

$$XorCK = q_s(7)\ xor\ q_s(9) \tag{2.7}$$

and ak_inputGen is the input of the bit-stream at the clock.

The code of the two shift register is:

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity ShiftRegAK is
5          generic(N_bit : integer:=5);
6              port(
7                   ak       : in std_logic;                                    --  Shift register input
8                   q        : out std_logic_vector(1 downto 0);                --  Shift register output
9                   clk      : in std_logic;                                    --  clk
10                  a_rst_n  : in std_logic                                     --  Asynchronous active low reset
11                 );
12 end ShiftRegAK;
13
14 architecture bhv of ShiftRegAK is                                            -- architectural declaration (behavioral description)
15
16 signal q_s : std_logic_vector(N_bit - 1 downto 0);                           -- internal signal used to map the internal registers
17
18 begin
19         shift_reg_proc: process(clk,a_rst_n)                                 -- Process realizing a sequential network with asynchronous reset
20                 begin
21
22                         if(a_rst_n = '1') then
23                            q_s <= (others => '0');
24                         elsif(rising_edge(clk)) then                         -- Positive edge trigger D-flip-flop register modelling
25                            q_s(0)                    <= ak;                   -- Sampling the input ak
26                            q_s(N_bit - 1 downto 1) <= q_s(N_bit - 2 downto 0);  -- Shifting the internal bits
27                         end if;
28         end process;
29     q(0) <= q_s(2);                                                          -- takes the output of ak-3
30     q(1) <= q_s(3);                                                          -- takes the output of ak-4
31 end bhv;
```

Figure 2.4: *Shift Register used for a_k*

Figure 2.4 shows the code for ShiftRegisterAk, where at the lines 29 and 30 are taken in output the values of the two shift register used to obtain ak_{-3} and ak_{-4} in order to perform the (2.6).

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  entity ShiftRegCK is
5          generic(N_bit : integer := 10);
6              port(
7                   ck       : in std_logic;                                    --  Shift register input
8                   q        : out std_logic_vector(1 downto 0);                --  Shift register output
9                   clk      : in std_logic;                                    --  clk
10                  a_rst_n  : in std_logic                                     --  Asynchronous active low reset
11                 );
12 end ShiftRegCK;
13
14 architecture bhv of ShiftRegCK is
15
16 signal q_s : std_logic_vector(N_bit - 1 downto 0);                           -- internal signal used to map the internal registers
17
18 begin
19
20         shift_reg_proc: process(clk,a_rst_n)                                 -- Process realizing a sequential network with asynchronous reset
21                 begin
22
23                         if(a_rst_n = '1') then
24                            q_s <= (others => '0');
25
26                         elsif(rising_edge(clk)) then                         -- Positive edge trigger D-flip-flop register modelling
27                            q_s(0)                    <= ck;                   -- Sampling the input d
28                            q_s(N_bit - 1 downto 1) <= q_s(N_bit - 2 downto 0);  -- Shifting the internal bits
29                         end if;
30         end process;
31
32     q(0) <= q_s(7);                                                          -- takes the output of ck-8
33     q(1) <= q_s(9);                                                          -- takes the output of ak-10
34 end bhv;
```

Figure 2.5: *Shift Register used for C_k*

Figure 2.14 shows the code for ShiftRegisterCk, where at the lines 32 and 33 are taken in output the values of the two shift register used to obtain ck_{-8} and ck_{-10} in order to perform the (2.7).

2.3 Implementation under VIVADO

In this section will be described the result of the implementation on VIVADO, taking into account parameters like the Worst Negativa Slack (from now on it will be called WNS), the Worst Hold Slack (from now on it will be called WHS), the LUT structure and the power utilization.

A description for each parameter is provided and, to clarify the schematic of the VIVADO implementation, in the same directory of this report are available two file .pdf with the schemes of this version.

2.3.1 Project Summary

The Summary available in VIVADO is a recap of the synthesis and the implementation where there are some information about the timing constraints, the power consumpion and the utilization of the resources.

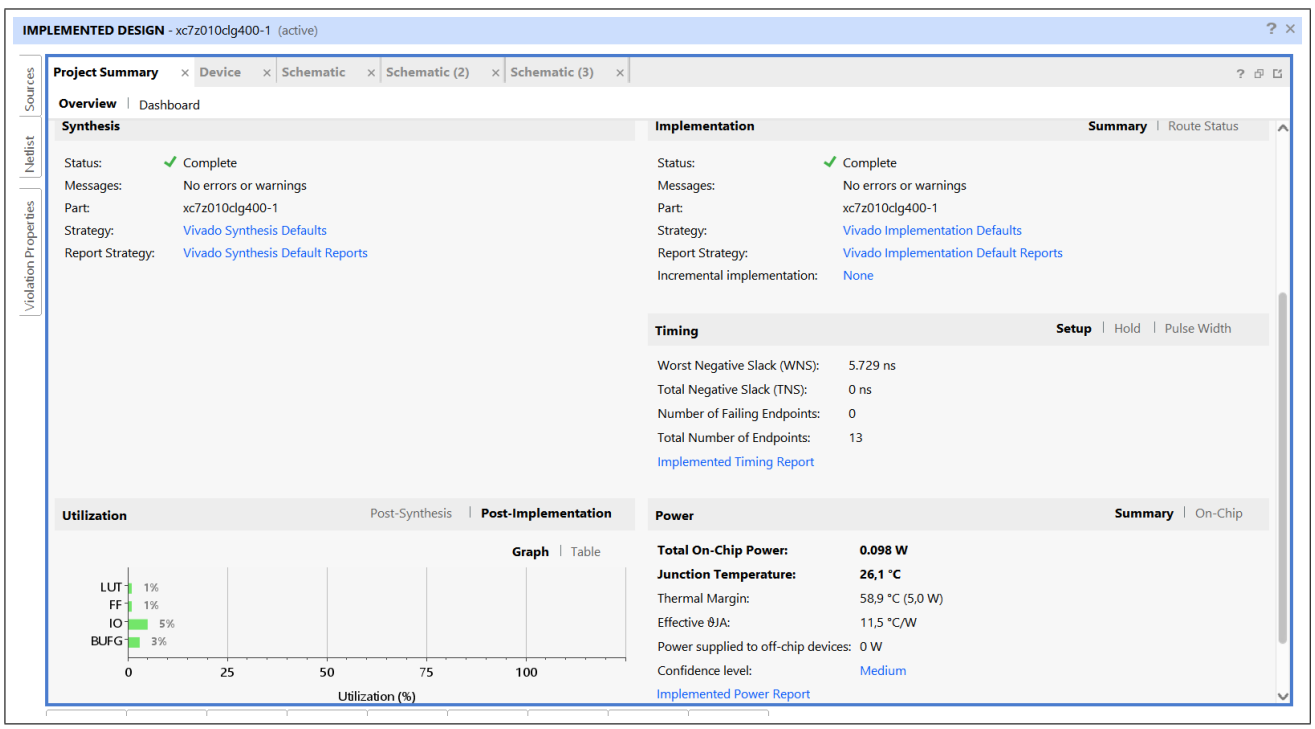

Figure 2.6: *Summary Report with two shift registers*

Everything shown in figure 2.6 is described in the following sections

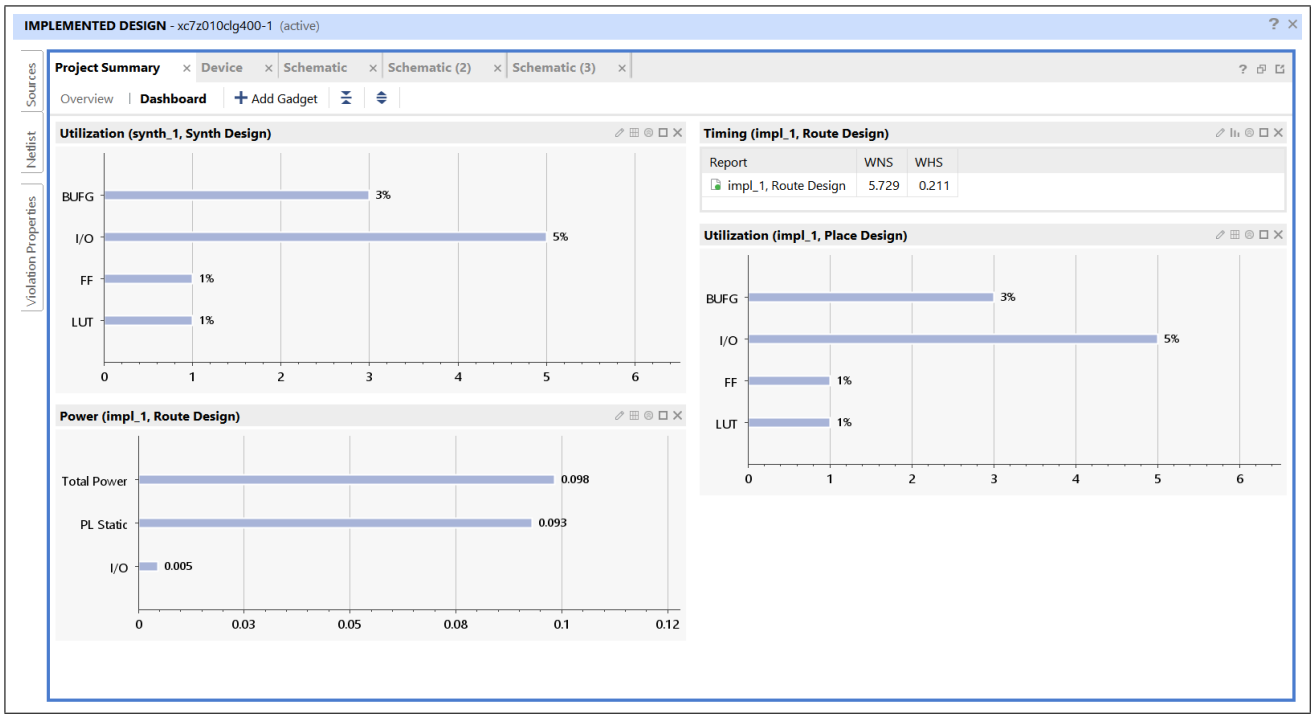

Figure 2.7: *Dashboard of the utilization*

In figure 2.7 there are diagrams with the percentage of utilization of the single resources.

2.3.2 Critical path

A critical path is a path with the largest delay, and for this reason is important to know it, in order to properly manage the system. With this version, as shown in figure 2.8, the WNS relative at the critical path is 5.729 ns.

A positive slack, like in this case, at some node implies that the arrival time at that node may be increased by the same value, without affecting the overall delay of the circuit.

On the contrary, negative slack means that the data signal is unable to traverse the combinational logic between the point A and the endpoint B of the timing path fast enough to ensure a correct circuit operation, so it indicates that the signal arrives at the endpoint later than the time it needs. In conclusion, it implies that the timing constraints given for the design, are not met.

In the other version with just one shift register will shown an improvment under this aspect, obtaining a WNS equal to 6.425ns.

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	So
Path 1	5.729	1	2	2	i_ShiftCk/q_s_reg[7]/C	i_ShiftCk/q_s_reg[0]/D	2.120	0.774	1.346	8.0	clk
Path 2	6.474	0	1	2	i_ShiftAk/q_s_reg[2]/C	i_ShiftAk/q_s_reg[3]/D	1.363	0.518	0.845	8.0	clk
Path 3	6.599	0	1	2	i_ShiftCk/q_s_reg[7]/C	i_ShiftCk/q_s_reg[8]/D	1.110	0.478	0.632	8.0	clk
Path 4	6.679	0	1	1	i_ShiftCk/q_s_reg[6]/C	i_ShiftCk/q_s_reg[7]/D	1.096	0.478	0.618	8.0	clk
Path 5	6.707	0	1	1	i_ShiftCk/q_s_reg[3]/C	i_ShiftCk/q_s_reg[4]/D	1.033	0.419	0.614	8.0	clk
Path 6	6.829	0	1	1	i_ShiftCk/q_s_reg[1]/C	i_ShiftCk/q_s_reg[2]/D	1.078	0.456	0.622	8.0	clk
Path 7	6.833	0	1	1	i_ShiftCk/q_s_reg[0]/C	i_ShiftCk/q_s_reg[1]/D	1.039	0.456	0.583	8.0	clk
Path 8	6.834	0	1	1	i_ShiftAk/q_s_reg[0]/C	i_ShiftAk/q_s_reg[1]/D	1.088	0.518	0.570	8.0	clk
Path 9	6.903	0	1	1	i_ShiftCk/q_s_reg[5]/C	i_ShiftCk/q_s_reg[6]/D	0.860	0.478	0.382	8.0	clk
Path 10	6.911	0	1	1	i_ShiftAk/q_s_reg[1]/C	i_ShiftAk/q_s_reg[2]/D	1.008	0.518	0.490	8.0	clk

Figure 2.8: *Summary of all paths*

Vivado allows also to see the path in the schematic, as shown in 2.9. Attached with this report there are two file .pdf with the schematic and the critical path of both the versions.

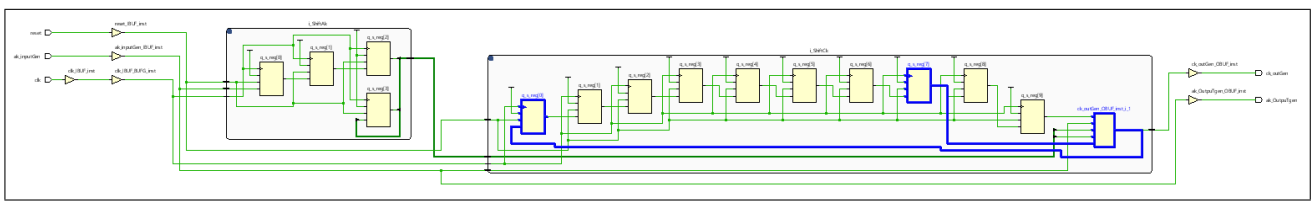

Figure 2.9: *The critical path*

2.3.3 Max frequency

Knowing the critical path and the relative WNS, is possible to calculate the Max possible frequency that can successfully pilot the circuit. With this implementation the circuit has a clock value of 8 ns, that means a frequency of 125Mhz, but as discussed in the previous section, can be increased.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5,729 ns	Worst Hold Slack (WHS):	0,211 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:	13	Total Number of Endpoints:	13	Total Number of Endpoints:	15

All user specified timing constraints are met.

Figure 2.10: *Worst Negative Slack Value*

Since with a clock of 8ns there is a WNS value equals to 5.729ns, can be calculated the difference:

$$NewPeriod = 8 - 5.729 \tag{2.8}$$

obtaining

$$NewPeriod = 2,271ns \tag{2.9}$$

The latter can be used to perform

$$\frac{1}{2,271 * 10^{-9}} \simeq 440Mhz \tag{2.10}$$

that is the max frequency at which the system still verifies the constraints.

At this point, editing the time constraint as shown in figure 2.11

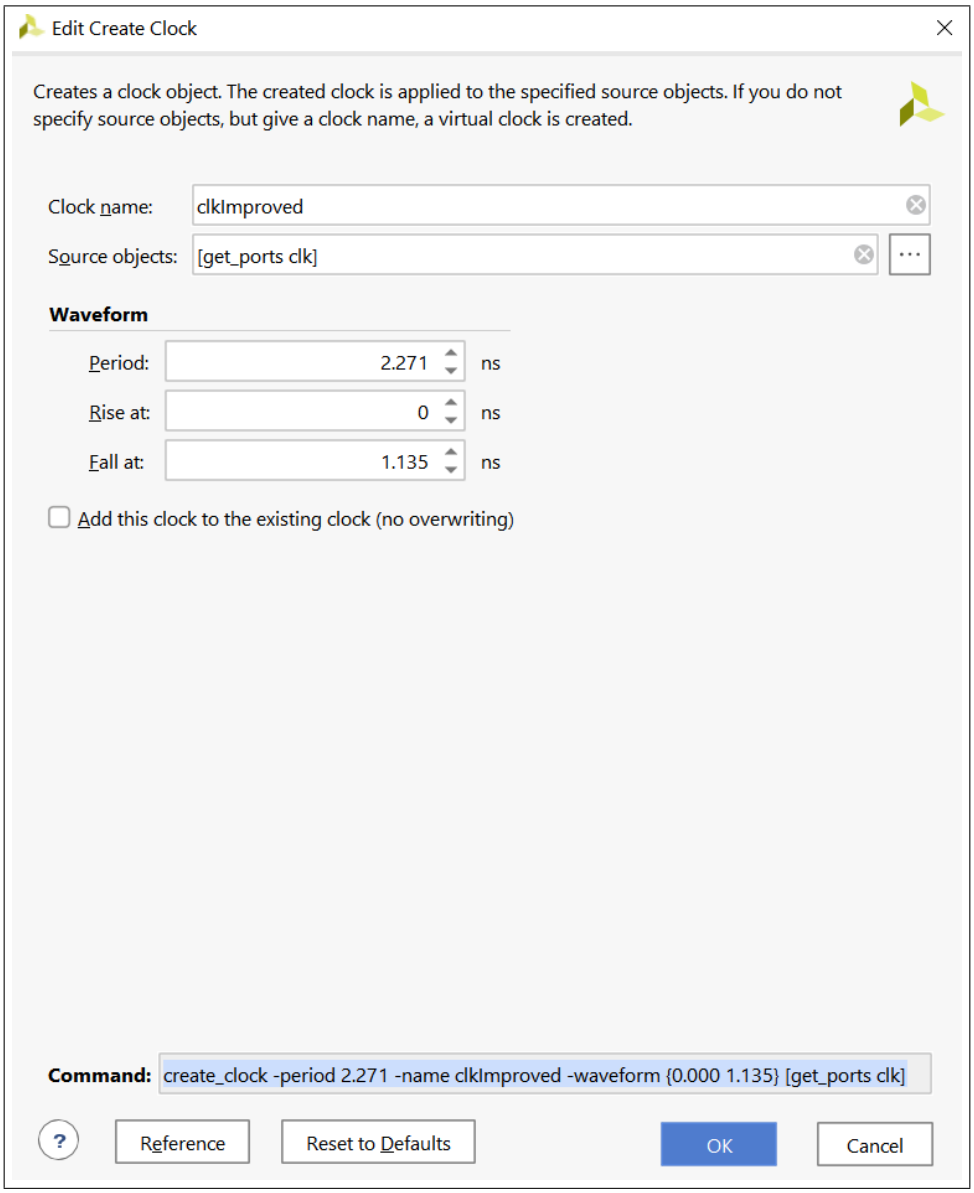

Figure 2.11: *Editing time constraint*

a new WNS equal to 0.864 ns is obtained, as shown in figure 3.10.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,864 ns	Worst Hold Slack (WHS):	0,106 ns	Worst Pulse Width Slack (WPWS):	0,116 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:	13	Total Number of Endpoints:	13	Total Number of Endpoints:	15

All user specified timing constraints are met.

Figure 2.12: *New WNS with 2,271 as clock period*

and no further improvement is possible since use a clock value less than 2.271 ns will provide a negative WNS.

2.3.4 Report LUT Utilization

From this report can be observed an estimation of the utilization.
For the Flip Flops 14/35200 are used. They are the sum of the two shift registers.
About the I/O, 5/100 pins are used. They are ak, ck, ak_out, clock and the reset.

Summary

Resource	Utilization	Available	Utilization %
LUT	1	17600	0.01
FF	14	35200	0.04
IO	5	100	5.00

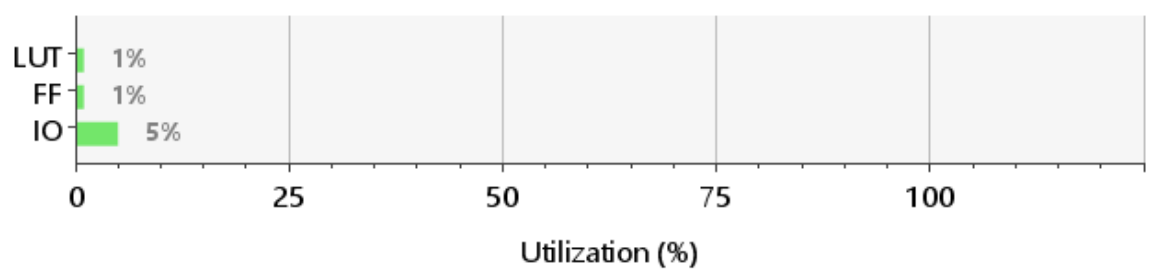

Figure 2.13: *Summary of the utilization*

2.3.5 Report power utilization

From this report it's possible to observe an estimation of the power consumption where 5% is for dynimic power and 95% for the static power consumption.

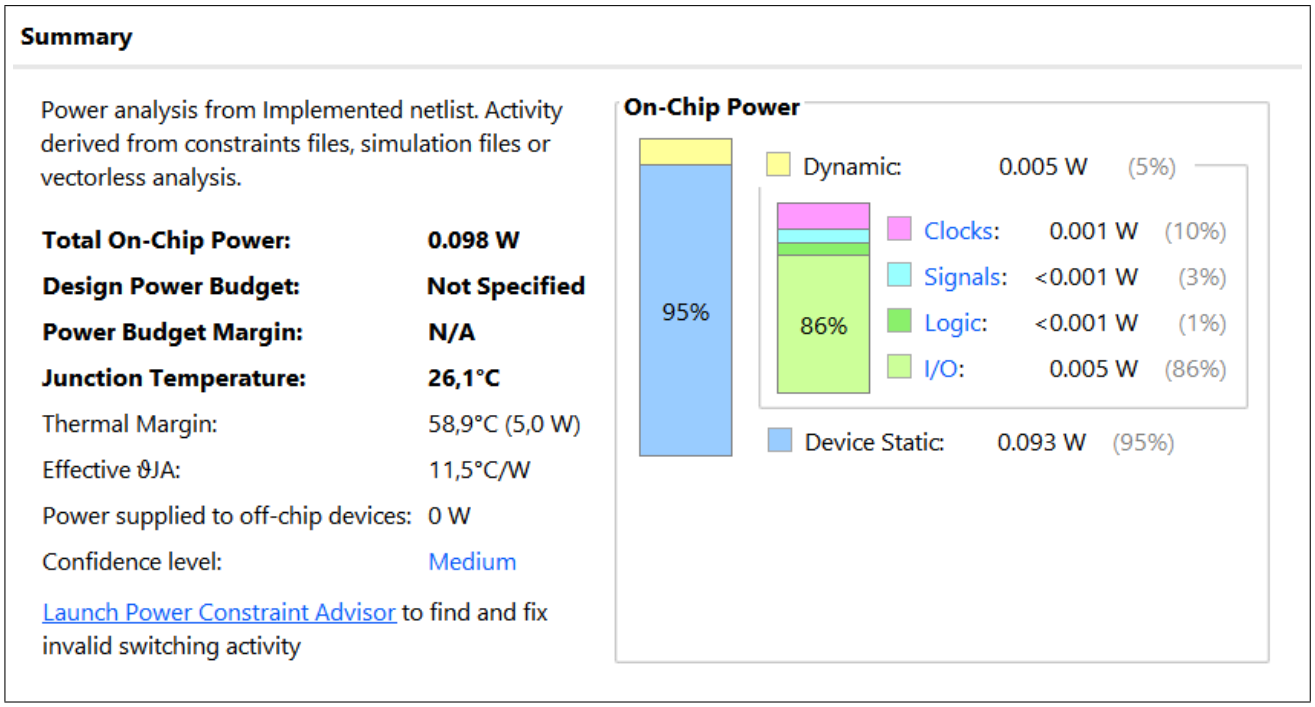

Figure 2.14: *Summary of power consumption*

Chapter 3

Design and Implementation with one Shift Register

Another possible implementation of the generator polynomial has shown in this chapter. In particular are described the architecture, the vhdl modules, the relative tests with a test bench. Particular attention is given to the differences between the two different versions, focusing on the WNS and the reports provided by VIVADO.

3.1 Architecture Description and Data Flow

Following [1], has been implemented a version that implements the same polynomial eq: 1.2. Using the theory and a single shift register with 10 FF the same result of the previous implementation can be achieved, obtaining also better performances.

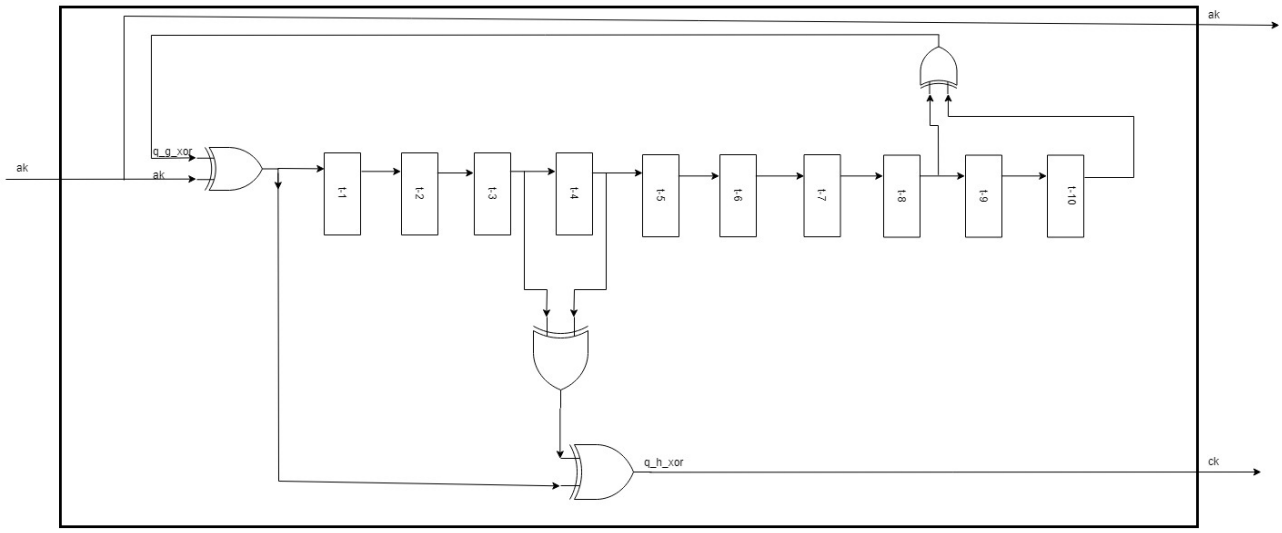

Figure 3.1: *Logic with one shift register*

3.2 VHDL Code

The VHDL implementation of the generator with one shift register instead two, is inside GeneratorCode.vhd.
More in detail has been implemented:

- GeneratorCode.vhd: that describes the entity generator.
- ShiftRegister.vhd: that describes the shift register used to memorize all the past values used to calculate the feed-back polynomial and the feed-forward polynomial.

```
7     entity ConvolutionalGenerator is
8         port  (
9             clk                 : in std_ulogic;        -- clock signal
10            reset               : in std_ulogic;        -- reset signal
11            ak_inputGen         : in std_ulogic;        -- the bit-stream is taken bit a bit
12            ak_OutpuTgen        : out std_ulogic;       -- the ak output, the same in input
13            ck_outGen           : out std_ulogic        -- the result of coding
14        );
15    end ConvolutionalGenerator;
16
17 architecture struct of ConvolutionalGenerator is
18
19        component ShiftReg
20           generic(N_bit : integer := 10);
21
22           port(
23                        ak      : in std_logic;
24                        clk     : in std_logic;
25                        a_rst_n : in std_logic;
26                        ck_out  : out std_logic
27        );
28        end component;
```

Figure 3.2: *VHDL description of entity generator with one register*

In figure 3.2 is shown the code of the generator, and his relative shift register.

```
34          signal ck_out : std_logic;                      -- Ck value in output
35
36     begin
37
38     i_Shift: ShiftReg
39                port map(
40                ak              =>ak_inputGen,
41                clk             => clk,
42                a_rst_n         => reset,
43                ck_out          => ck_outGen
44                        );
45
46  ak_OutpuTgen <= ak_inputGen;                            -- Ak value in output
47  end struct;
```

Figure 3.3: *PortMapping in GeneratorCode.vhd with one shift register*

The code of the shift register is:

```
4   entity ShiftReg is
5          generic(N_bit : integer);
6              port(
7                          ak      : in std_logic;                                  --  Shift register input
8                          clk     : in std_logic;                                  --  Clock
9                          a_rst_n : in std_logic;                                  --  Reset
10                         ck_out     : out std_logic                               --  Shift register output
11
12                                                    );
13  end ShiftReg;
14
15  architecture bhv of ShiftReg is
16
17  signal  q_s  : std_logic_vector(N_bit - 1 downto 0);
18  signal  q_g_xor  : std_logic;
19  signal  q_h_xor  : std_logic;
20  begin
21
22
23         shift_reg_proc: process(clk,a_rst_n)
24                 begin
25
26                         if(a_rst_n = '1') then
27                            q_s <= (others => '0');
28
29                         elsif(rising_edge(clk)) then                             -- Positive edge trigger D-flip-flop register modelling
30                            q_s(0)       <= ak xor q_g_xor;                        -- Sampling the input ak
31                            q_s(N_bit - 1 downto 1) <= q_s(N_bit - 2 downto 0);   -- Shifting the internal bits
32                         end if;
33         end process;
34
35  q_g_xor <=  (q_s(7)) xor  (q_s(9)) ;                                            -- Performing ck-8 + ck-10
36  q_h_xor <=  (ak xor q_g_xor) xor (q_s(2)  xor  q_s(3)) ;                        -- ak + ck-8 + ck-10 + ak-3 + ak-4
37  ck_out  <= q_h_xor;                                                             -- Result to Ck
38
39  end bhv;
```

Figure 3.4: *Shift Register*

As shown in figure 3.4 at the line 35, q_g_xor that is the feed-back polynomial presented in eq (1.4) is calculated performing:

$$q\_g\_xor <= q_s(7)\ xor\ q_s(9) \tag{3.1}$$

where q_s(7) and q_s(9) are ck_{-8} and ck_{-10}.

The latter can be used to perform

$$ck_{\text{out}} <= (ak\ xor\ q\_g\_xor)\ xor\ (q_s(2)\ xor\ q_s(3)) \tag{3.2}$$

where q_s(2) xor q_s(3) are ak_{-3} and ak_{-4} and togheter form the feed-forward polynomial as presented in eq (1.5).

So the two polynomials, associated with ak can perform the original polynomial (1.2).

3.3 Synthesis under VIVADO

In this section will be described the result of the implementation on VIVADO, taking into account parameters like the Worst Negativa Slack (from now on it will be called WNS),the Worst Hold Slack (from now on it will be called WHS), the LUT structure and the power utilization.

A description for each parameter is provided and, to clarify the schematic of the VIVADO implementation, in the same directory of this report are available two file .pdf with the schemes of this version.

3.3.1 Project summary

The Summary available in VIVADO is a recap of the synthesis and the implementation where there are some information about the timing constraints, the power consumpion and the utilization of the resources.

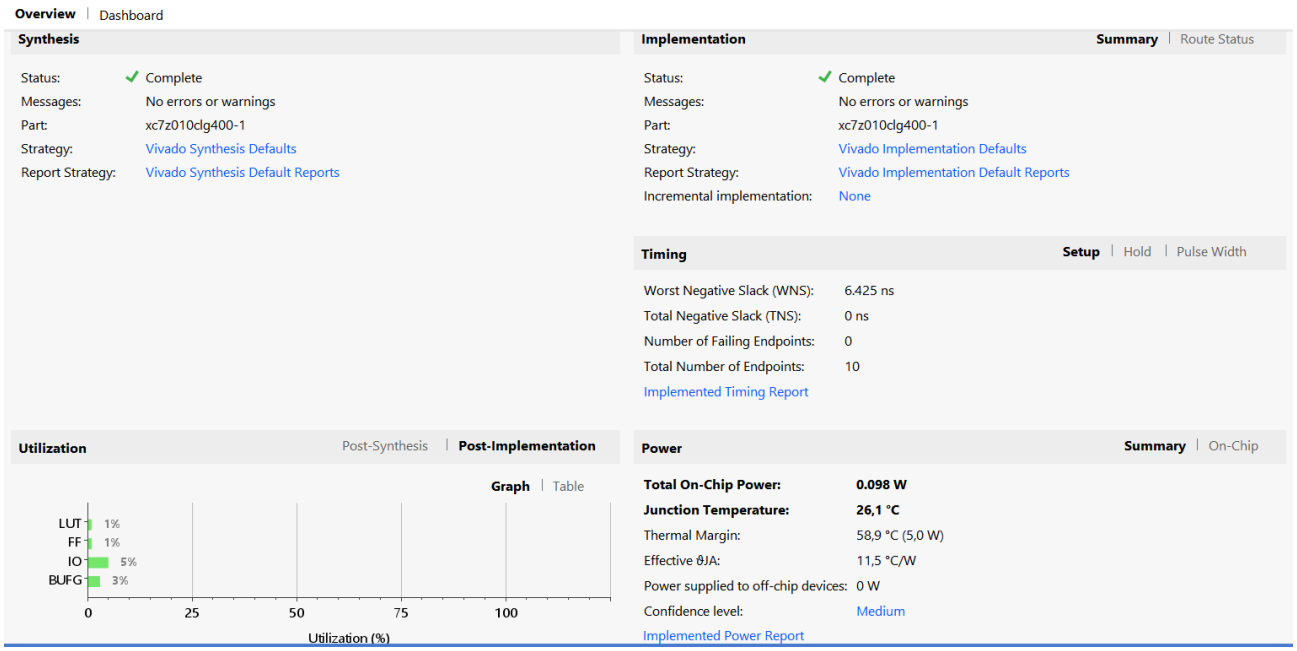

Figure 3.5: *Summary Report with one shift register*

Everything shown in figure 3.5 is described in the following sections

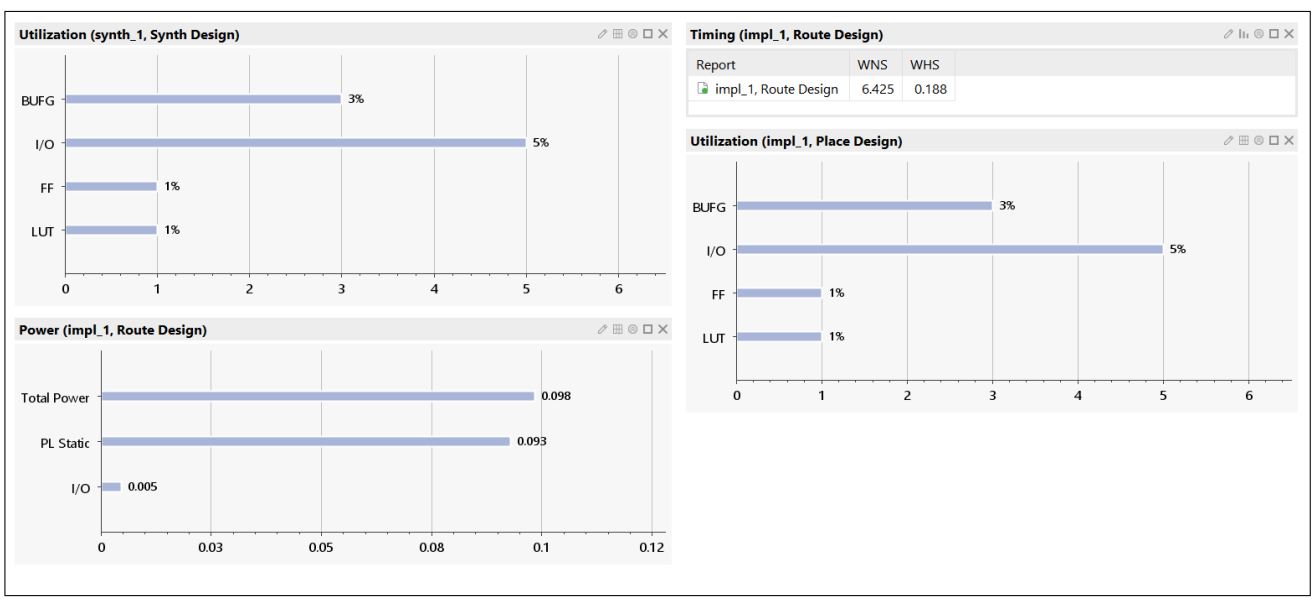

Figure 3.6: *DashBoard with one shift register*

In figure 3.6 there are diagrams with the percentage of utilization of the single resources.

3.3.2 Critical Path

Ad mentioned in 2.3.2 the critical path is a path with the largest delay, and in this version as shown in figure 3.7, the WNS relative at the critical path is 6.437 ns. In the previous version was 5.729 ns, so with this architecture can be reached a higher max frequency.

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Cl
Path 1	6.425	1	2	3	i_Shift/q_s_reg[7]/C	i_Shift/q_s_reg[0]/D	1.508	0.642	0.866	8.0	clc_125
Path 2	6.732	0	1	1	i_Shift/q_s_reg[0]/C	i_Shift/q_s_reg[1]/D	1.144	0.518	0.626	8.0	clc_125
Path 3	6.752	0	1	3	i_Shift/q_s_reg[7]/C	i_Shift/q_s_reg[8]/D	1.129	0.518	0.611	8.0	clc_125
Path 4	6.809	0	1	1	i_Shift/q_s_reg[6]/C	i_Shift/q_s_reg[7]/D	1.128	0.518	0.610	8.0	clc_125
Path 5	6.818	0	1	2	i_Shift/q_s_reg[3]/C	i_Shift/q_s_reg[4]/D	1.089	0.456	0.633	8.0	clc_125
Path 6	6.822	0	1	2	i_Shift/q_s_reg[2]/C	i_Shift/q_s_reg[3]/D	1.050	0.456	0.594	8.0	clc_125
Path 7	6.899	0	1	1	i_Shift/q_s_reg[5]/C	i_Shift/q_s_reg[6]/D	1.037	0.518	0.519	8.0	clc_125
Path 8	6.948	0	1	1	i_Shift/q_s_reg[8]/C	i_Shift/q_s_reg[9]/D	0.802	0.419	0.383	8.0	clc_125
Path 9	7.110	0	1	1	i_Shift/q_s_reg[4]/C	i_Shift/q_s_reg[5]/D	0.788	0.456	0.332	8.0	clc_125
Path 10	7.123	0	1	1	i_Shift/q_s_reg[1]/C	i_Shift/q_s_reg[2]/D	0.748	0.456	0.292	8.0	clc_125

Figure 3.7: *Summary of all paths*

Vivado allows also to see the path in the schematic, as shown in 3.8. Attached with this report there are two file .pdf with the schematic and the critical path of both the versions.

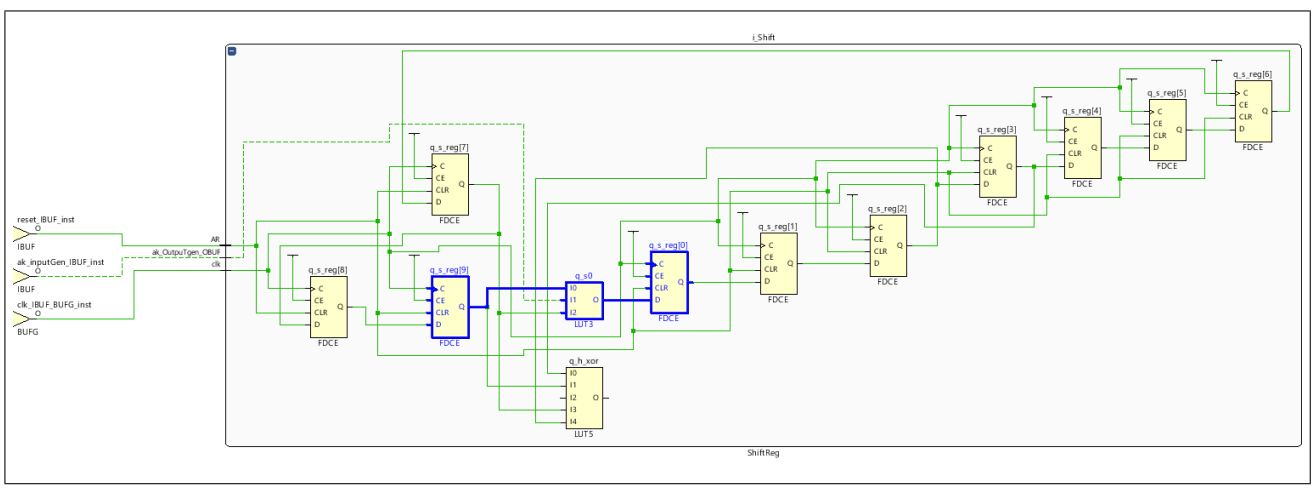

Figure 3.8: *Critical path with one shift register*

3.3.3 Max frequency

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	6,425 ns	Worst Hold Slack (WHS):	0,188 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:	10	Total Number of Endpoints:	10	Total Number of Endpoints:	11

All user specified timing constraints are met.

Figure 3.9: *WNS with one shift register*

With this architecture, since with a clock of 8ns there is a WNS value equals to 6.437ns, can be calculated the difference:

$$NewPeriod = 8 - 6.425 \tag{3.3}$$

obtaining

$$NewPeriod = 1,575ns \tag{3.4}$$

The latter can be used to perform

$$\frac{1}{1,575 * 10^{-9}} \simeq 634Mhz \tag{3.5}$$

that is the max frequency at which the system still verifies the constraints.

At this point, editing the time constraint as shown in figure 2.11
a new WNS equal to 0.389ns is obtained, as shown in figure 3.10.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,389 ns	Worst Hold Slack (WHS):	0,155 ns	Worst Pulse Width Slack (WPWS):	0,355 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:	10	Total Number of Endpoints:	10	Total Number of Endpoints:	11

Timing constraints are not met.

Figure 3.10: *New WNS with 1,575 as clock period*

and no further improvement is possible since use a clock value less than 1,575 ns will provide a negative WNS.

3.3.4 Report LUT utilization

From this report can be observed an estimation of the utilization.For the Flip Flops 10/35200 are used. They are the size of the only shift registers.

About the I/O, 5/100 pins are used. They are the same of the version with two shift register, i.e ak, ck, akout, clock and the reset.

So the utilization of the flip flops has decreased from 0.04% in the previous version, to 0.03%.

Summary

Resource	Utilization	Available	Utilization %
LUT	1	17600	0.01
FF	10	35200	0.03
IO	5	100	5.00

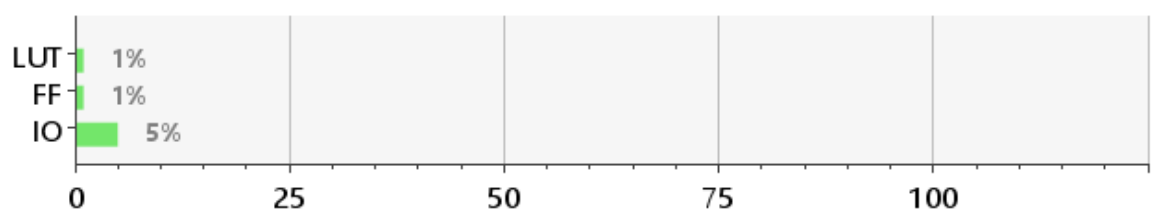

Figure 3.11: *LUT utilization with one shift register*

3.3.5 Report power utilization

From this report its possible to observe an estimation of the power consumption where 6% is for dynamic power and 94% for the static power consumption.

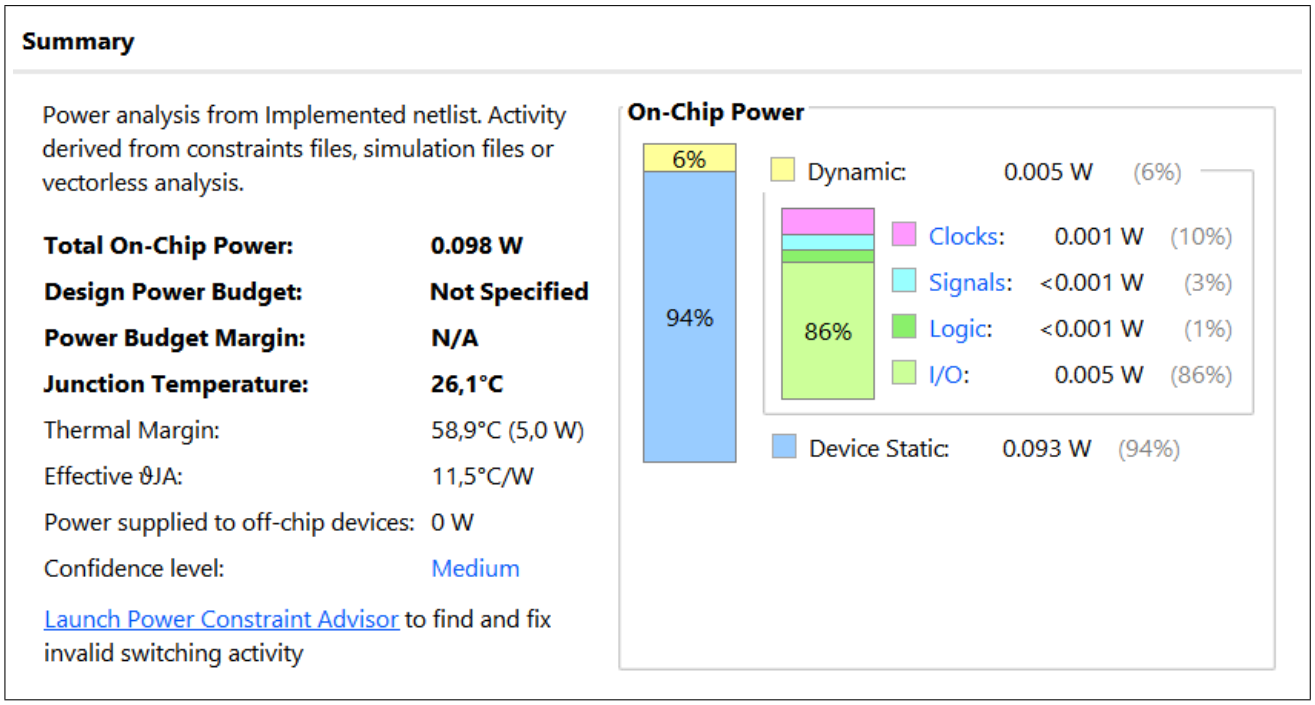

Figure 3.12: *Power consumption with one shift register*

Chapter 4

Verification and Simulation of Architecture

To test the correctness of the generator, has been implemented a simulator in C language that allows to test with any possible 16 bit input the implementation in VHDL. The simulator works in the same way for both the implementation, since the only different between them is the number of shift register used.

Through the terminal, the user can choice 16 bit to use in order to test the entire environment, without the need of modifying the code in the Test Bench for each test. The latter advantages comes from by the fact that the Test Bench reads the file 'InputVhd.txt' and writes the output in the file "OutputVhd.txt".

The C simulator will compare 'OutputVhd.txt' with the result of the simulation in order to verify the equality. By this method it's possible to test in a fast way the generator with any possibile $2^N input$.

4.1 Simulation in C language

The executable is in the same directory of GeneratorCode_tb.

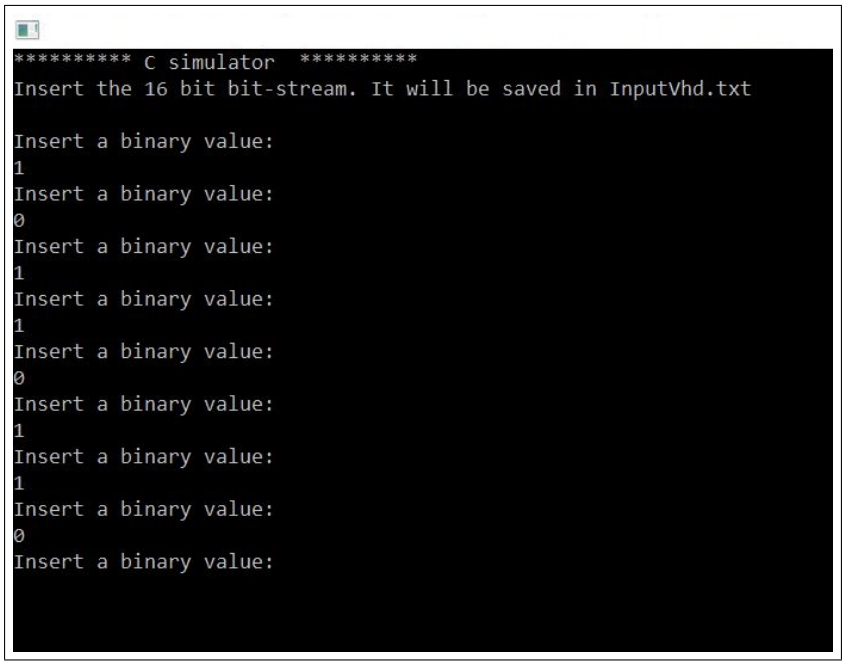

Figure 4.1: *Interface of the C simulator*

In figure 4.1 is shown the interface of the terminal. Once started, it:

- takes an input, through the terminal, decided by the user
- writes in 'InputVhd.txt' the input decided.
- simulate the generation of ck
- waits for the user to run the simulation in ModelSym
- since the Test Bench writes the output of the vhdl generator in 'OutputVhd.txt', the C simulator reads it and compare the output of the Test Bench with the output calculated

```
The result of the simulator is:
ck[0] = 1
ck[1] = 0
ck[2] = 1
ck[3] = 0
ck[4] = 1
ck[5] = 0
ck[6] = 1
ck[7] = 1
ck[8] = 0
ck[9] = 1
ck[10] = 0
ck[11] = 0
ck[12] = 0
ck[13] = 1
ck[14] = 1
ck[15] = 1
The output bit-stream has been saved in 'InputVhd.txt'
Now it's possible to run the TestBench!
 Once ended, type 'S' to verify the correctness or type 'N' to quit
```

Figure 4.2: *A possible output with '1011 0110 0110 1101'*

In figure 4.2 is shown the result of the simulation with the input '1011 0110 0110 1101'. At this point the simulator waits for the user that can run the Test Bench.

```
The output bit-stream has been saved in 'InputVhd.txt'
Now it's possible to run the TestBench!
 Once ended, type 'S' to verify the correctness or type 'N' to quit
s
VhdlOutput[0] = 1 , Simulator_C_Output[0] = 1
VhdlOutput[1] = 0 , Simulator_C_Output[1] = 0
VhdlOutput[2] = 1 , Simulator_C_Output[2] = 1
VhdlOutput[3] = 0 , Simulator_C_Output[3] = 0
VhdlOutput[4] = 1 , Simulator_C_Output[4] = 1
VhdlOutput[5] = 0 , Simulator_C_Output[5] = 0
VhdlOutput[6] = 1 , Simulator_C_Output[6] = 1
VhdlOutput[7] = 1 , Simulator_C_Output[7] = 1
VhdlOutput[8] = 0 , Simulator_C_Output[8] = 0
VhdlOutput[9] = 1 , Simulator_C_Output[9] = 1
VhdlOutput[10] = 0 , Simulator_C_Output[10] = 0
VhdlOutput[11] = 0 , Simulator_C_Output[11] = 0
VhdlOutput[12] = 0 , Simulator_C_Output[12] = 0
VhdlOutput[13] = 1 , Simulator_C_Output[13] = 1
VhdlOutput[14] = 1 , Simulator_C_Output[14] = 1
VhdlOutput[15] = 1 , Simulator_C_Output[15] = 1
```

Figure 4.3: *Compare the two outputs*

As shown in figure 4.3, once the Test Bench has been runned, typing 'S' on the terminal the simulator reads from 'OutputVhd.txt' the bit-stream created by the Test Bench, called 'VhdlOutput' in figure 4.3, and compare it with 'Simulator_C_Output' that is the output of the C simulator.

As shown in figure 4.3, the input

'1011 0110 0110 1101'

has been codified as

'1010 1011 0100 0111'

and the coding is the same for both the C simulator and the VHDL generator.

4.2 Simulation with TestBench

The Test Bench, since we use the C simulator to create the input bit-stream, becomes a reading from the file 'InputVhd.txt' and, at each clock, an updating of the signal ak_tb and is the same for both the architecture

```
entity ConvolutionalGenerator_tb is
end ConvolutionalGenerator_tb;

architecture bhv of ConvolutionalGenerator_tb is                                    -- Testbench architecture declaration
    ---------------------------------------------------------------------------
    -- Testbench constants
    ---------------------------------------------------------------------------
        constant T_CLK   : time := 10 ns;                                           -- Clock period
        constant T_RESET : time := 25 ns;                                           -- Period before the reset deassertion
        constant c_WIDTH : natural := 16;                                           -- Size of the bit-stream
    ---------------------------------------------------------------------------
    -- Testbench signals
    ---------------------------------------------------------------------------
        signal clk_tb : std_logic := '0';                                           -- clock signal, intialized to '0'
        signal rst_tb  : std_logic := '1';                                          -- reset signal
        signal ak_tb    : std_ulogic;                                               -- ak input signal to connect to the ak port of the simulator
        signal ak_out_tb    : std_ulogic;                                           -- ak output signal
        signal ck_tb    : std_logic;                                                -- ck output signal
        signal end_sim : std_logic := '1';                                          -- signal used to stop the simulation when there is nothing else to test
        signal r_InputFromSimulator : std_ulogic_vector(c_WIDTH -1  downto 0) := (others => '0');   -- signal used to read from the file 'InputVhd.txt'
    ---------------------------------------------------------------------------
   file file_VECTORS : text;                                                        -- file used to read 'InputVhd.txt'
   file file_RESULTS : text;                                                        -- file used to write 'OutputVhd.txt'
    ---------------------------------------------------------------------------
    -- Component to test
    ---------------------------------------------------------------------------
        component ConvolutionalGenerator is
        port  (
            clk                 : in std_ulogic;
            reset               : in std_ulogic;
            ak_inputGen         : in std_ulogic;
            ak_OutputGen        : out std_ulogic;
            ck_outGen           : out std_ulogic
        );
    end component;
```

Figure 4.4: *VHDL description of TestBench*

In figure 4.4 is shown the VHDL description of the generator.
The generator has 3 inputs:

- The clock.
- The reset.
- the ak_input from the bit-stream.

and 2 outputs:

- The ak_output.
- The ck_output.

```
        d_process: process(clk_tb, rst_tb)
    variable t : integer := 0; -i
    variable v_InputLine      : line;                                                    -- variable used to read a line from 'InputVhd.txt'
    variable v_OutputLine     : line;                                                    -- variable used to write a line in 'OutputVhd.txt'
    variable FileInputSimulator : std_ulogic_vector(c_WIDTH-1 downto 0);
        begin
          if(rst_tb = '1') then
             file_open(file_VECTORS, "InputVhd.txt",  read_mode);                         -- open the file 'InputVhd.txt'
             file_open(file_RESULTS, "OutputVhd.txt",  write_mode);                       -- open the file 'OutputVhd.txt'
      -- reads until the end the file 'InputVhd.txt'  --
              while not endfile(file_VECTORS) loop
                    readline(file_VECTORS, v_InputLine);
                    read(v_InputLine, FileInputSimulator);
                    r_InputFromSimulator <= FileInputSimulator;
              end loop;
                file_close(file_VECTORS);
                ak_tb <= '0';
                t := 0;
              elsif(rising_edge(clk_tb)) then
      -- for each value of t reads the next input of the bit-stream --
               case(t) is
                      when 1 => ak_tb  <= r_InputFromSimulator(15); write(v_OutputLine, ck_tb, right, 1);
                      when 2 => ak_tb  <= r_InputFromSimulator(14); write(v_OutputLine, ck_tb, right, 1);
                      when 3 => ak_tb  <= r_InputFromSimulator(13); write(v_OutputLine, ck_tb, right, 1);
                      when 4 => ak_tb  <= r_InputFromSimulator(12); write(v_OutputLine, ck_tb, right, 1);
                      when 5 => ak_tb  <= r_InputFromSimulator(11); write(v_OutputLine, ck_tb, right, 1);
                      when 6 => ak_tb  <= r_InputFromSimulator(10); write(v_OutputLine, ck_tb, right, 1);
                      when 7 => ak_tb  <= r_InputFromSimulator(9); write(v_OutputLine, ck_tb, right, 1);
                      when 8 => ak_tb  <= r_InputFromSimulator(8); write(v_OutputLine, ck_tb, right, 1);
                      when 9 => ak_tb  <= r_InputFromSimulator(7); write(v_OutputLine, ck_tb, right, 1);
                      when 10 => ak_tb <= r_InputFromSimulator(6); write(v_OutputLine, ck_tb, right, 1);
                      when 11 => ak_tb <= r_InputFromSimulator(5); write(v_OutputLine, ck_tb, right, 1);
                      when 12 => ak_tb <= r_InputFromSimulator(4); write(v_OutputLine, ck_tb, right, 1);
                      when 13 => ak_tb <= r_InputFromSimulator(3); write(v_OutputLine, ck_tb, right, 1);
                      when 14 => ak_tb <= r_InputFromSimulator(2); write(v_OutputLine, ck_tb, right, 1);
                      when 15 => ak_tb <= r_InputFromSimulator(1); write(v_OutputLine, ck_tb, right, 1);
                      when 16 => ak_tb <= r_InputFromSimulator(0); write(v_OutputLine, ck_tb, right, 1);
                      when 17 => write(v_OutputLine, ck_tb, right, 1); end_sim <= '0'; writeline(file_RESULTS, v_OutputLine); file_close(file_RESULTS);
          when others => null;
                end case;
                        t := t + 1;
              end if;
        end process;

end bhv;
```

Figure 4.5: *Reading from InputVhd.txt*

In figure 4.5 is shown the process used to:

- Update the input ak_tb at each clock.
- write in 'OutputVhd.txt' the value of ck_tb.

4.3 Wave on Model Sym

ModelSym allows to use 'Wave' in order to verify the output, when the Test Bench is run. It is not comfortable and immediate as using the terminal, like shown in figure 4.3, but is useful in order to verify that also the C simulator is working properly.

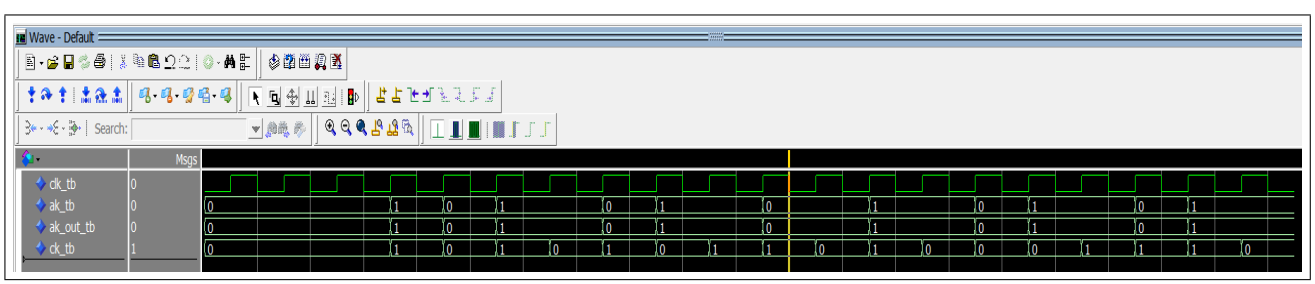

Figure 4.6: *Test with: '1011 0110 0110 1101'*

Using the wave, can be observed at each clock, the evolution of the generator.

In particular:

- ck_tb: Shows the evolution of the clock with 10 ns as period.
- ak_tb: Show the input of the bit-stream at the clock.
- ak_Out_tb: Show the output of ak at the clock. Has the same value of ak_tb.
- ck_tb: Show the output of ck, that is the code generated by the generator.

As shown with the C simulator, the input

'1011 0110 0110 1101'

has been codified as

'1010 1011 0100 0111'

Bibliography

[1] A High Throughput Hardware Architecture for Parallel Recursive Systematic Convolutional Encoders. *Gabriele Meoni, Luca Fanucci, Gianluca Giuffrida*.