UNIVERSITÀ DI PISA

MSc in Computer Engineering

Electronics and Communications Systems

Project Report:
Convolutional codes generator

Riccardo Sagramoni

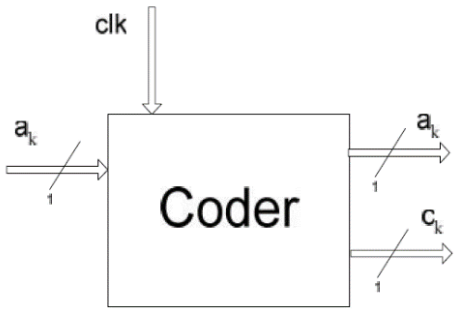
Academic Year 2021/2022

Table of Contents

Introduction

The goal of this project was to design a generator of **convolutional codes**, with the following requirements:

1. Code rate $R_c = \frac{1}{2}$
2. Constraint length $N = 11$
3. Encoding function $c_k = c_{k-8} + c_{k-10} + a_k + a_{k-3} + a_{k-4}$



The generator takes a new input bit a_k and produces two input bits per clock cycle:

- A replication of a_k
- An encoded bit c_k, computed from using the previous input (a_k , a_{k-3} , a_{k-4}) and output bits (c_{k-8} , c_{k-10})

Background: convolutional coding

A convolutional code is a type of error-correcting code (ECC), i.e. a sequence of bits used for controlling errors in data over unreliable or noisy communication channels.

The encoder uses a *sliding window* to calculate **r** parity bits by combining various subsets of bits in the window. In this project, the combining function is descripted in the third requirement. As showed in figure 1, the windows overlap and slide by 1.

The size of the window *N* is also called *constraint length* and, for this project, is stated in the second requirement. The longer the constraint length, the larger the number of parity bits that are influenced by any given message bit.

If a convolutional code that produces r parity bits per window and slides the window forward by one bit at a time, its **code rate is** $\frac{1}{r}$. Thus, our generator will produce *two parity bits per clock cycle* since the code rate is fixed to $\frac{1}{2}$.

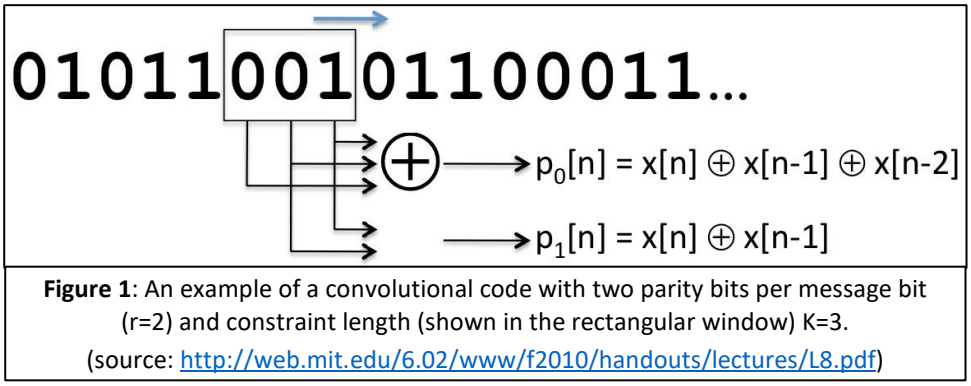


Figure 1: An example of a convolutional code with two parity bits per message bit (r=2) and constraint length (shown in the rectangular window) K=3.
(source: http://web.mit.edu/6.02/www/f2010/handouts/lectures/L8.pdf)

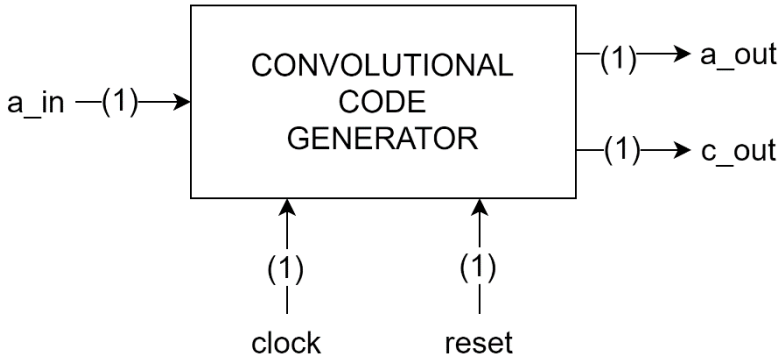
Analysis on the encoding function

$$c_k = c_{k-8} + c_{k-10} + a_k + a_{k-3} + a_{k-4}$$

As we previously stated, our encoding function uses both old input and old output bits to compute a new code bit. This classifies our generator as a **recursive** one.

Moreover, recursive generators are frequently *systematic*, i.e. they replicate the current input as one of the current code output. This is the case for our project, since we must produce two bits per clock cycle (code rate equals to ½), but only one encoding function is specified.

Description of the architecture



Ports

INPUT PORTS:

- **a_in** (1 bit): the bit stream to encode ("*a*")
- **clock** (1 bit): the clock signal of the system
- **reset** (1 bit): an asynchronous high-active reset signal

OUTPUT PORTS:

- **a_out** (1 bit): the replication of the input bit stream
- **c_out** (1 bit): the encoded bit stream

Components

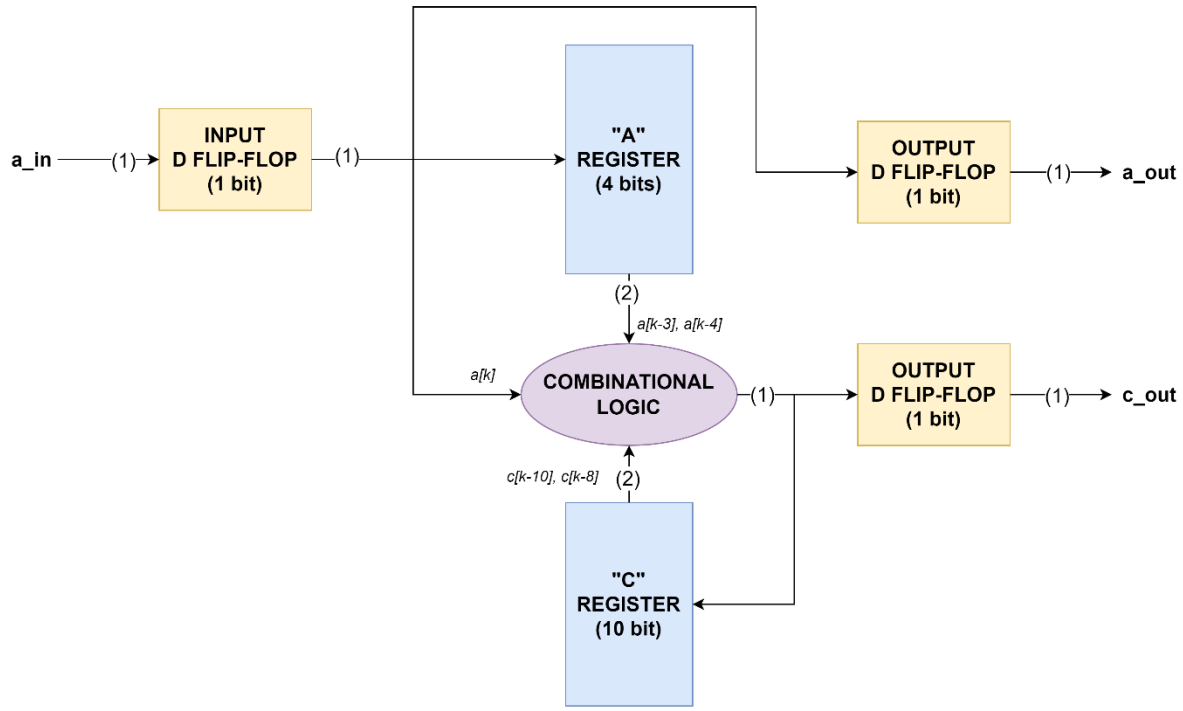
Internally the generator is composed of two types of components:

- **D Flip-Flops**: logical circuits which store and keep stable a single bit for a clock cycle.
- **Shift Registers**: logical circuits which store several bits in an "array-like" structure and shift each bit by one position at every positive edge of the clock. It's composed of a cascade of D Flip-Flops.

In particular, the convolutional code generator is composed by **two main shift registers**:

- The first one ("A" register) stores the previous values assumed by the input bit stream. It's composed of **4 DFFs**, so that it can provide a_{k-3} and a_{k-4} to the encoding function.
- The second one ("C" register) stores the previous values generated by the encoding function. It's composed of **10 DFFs**, so that it can provide c_{k-8} and c_{k-10} to the encoding function.

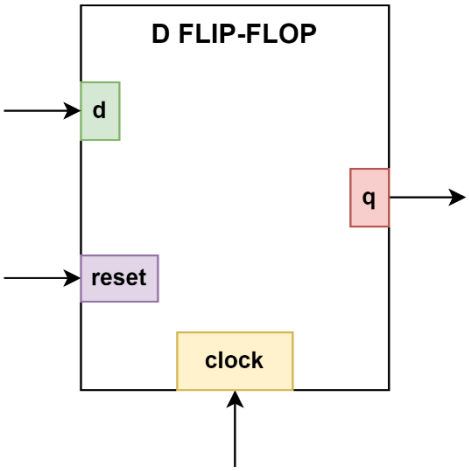
Moreover, in order to implement the **Register-Logic-Register** schema (necessary in order to correctly measure the critical path through Vivado), the system utilizes three DFFs which store and keep stable during a clock period the *a_in* input signal and the two output signals.



D Flip-Flop (DFF)

Classical data storing circuit, positive-edge triggered and with asynchronous reset.

It has threes input and one output ports:



- **clock** (input, 1 bit): the port for the clock signal.
- **reset** (input, 1 bit): asynchronous active-high signal for reset. If set to 1, it clears the current state of the Flip-Flop and forces the output bit down to zero.
- **d** (input, 1 bit): input data port. It MUST be stable in the proximity of the positive edge of the clock.
- **q** (output, 1 bit): output data port. At every positive edge of the clock, it assumes the value of the d port at that moment and remains stable for the entire clock cycle.

Shift register

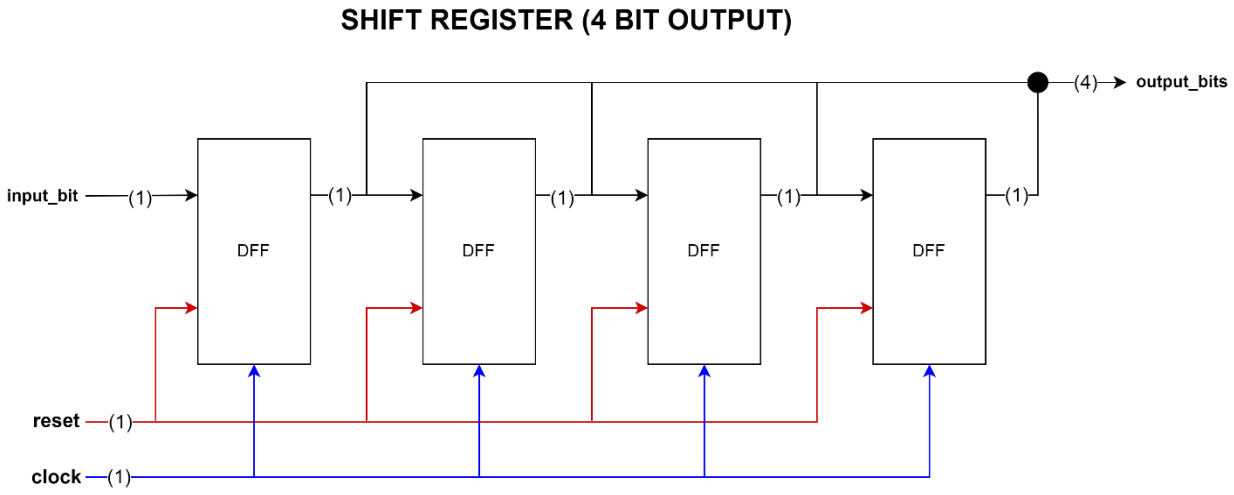
Data storing device composed by a cascade of D Flip-Flops. It stores several bits in an “array-like” structure and shift each bit by one position at every positive edge of the clock.

The number of D Flip-Flop is parameterized, and it’s equals to the amount of output bits.

The device has three input ports and one output port:

- **clock** (input, 1 bit): the port for the clock signal.

- **reset** (input, 1 bit): asynchronous active-high signal for reset. If set to 1, it clears the current state of all the Flip-Flops and forces their output bits down to zero.
- **input_bit** (input, 1 bit): input data port. It MUST be stable in the proximity of the positive edge of the clock. On a positive edge of the clock, it will be stored in the first DFF, while all the other values already stored in the register will be shifted by one "position" (i.e. to the following DFF in the cascade, if there is any).
- **output_bits** (output, N bits): output data port. It groups the output values of all the D Flip-Flop.

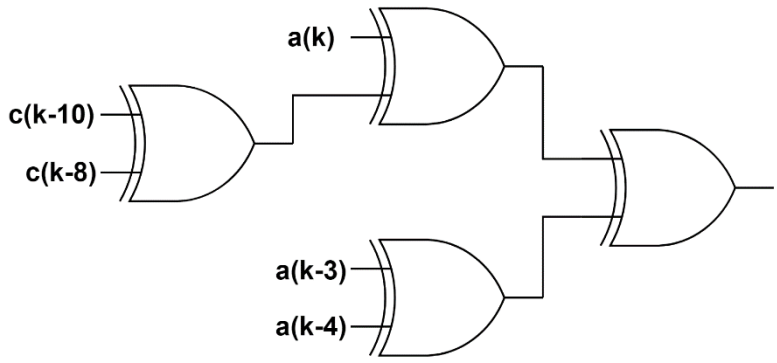


Combinational logic

The combinational logic of our system is the implementation of the encoding function

$$c_k = c_{k-8} + c_{k-10} + a_k + a_{k-3} + a_{k-4}$$

Since the function is composed of bit-to-bit sums, I implemented the function with a sequence of XOR gates. In order to minimize the delay and the critical path of the circuit, I chose a tree topology for connecting the gates.



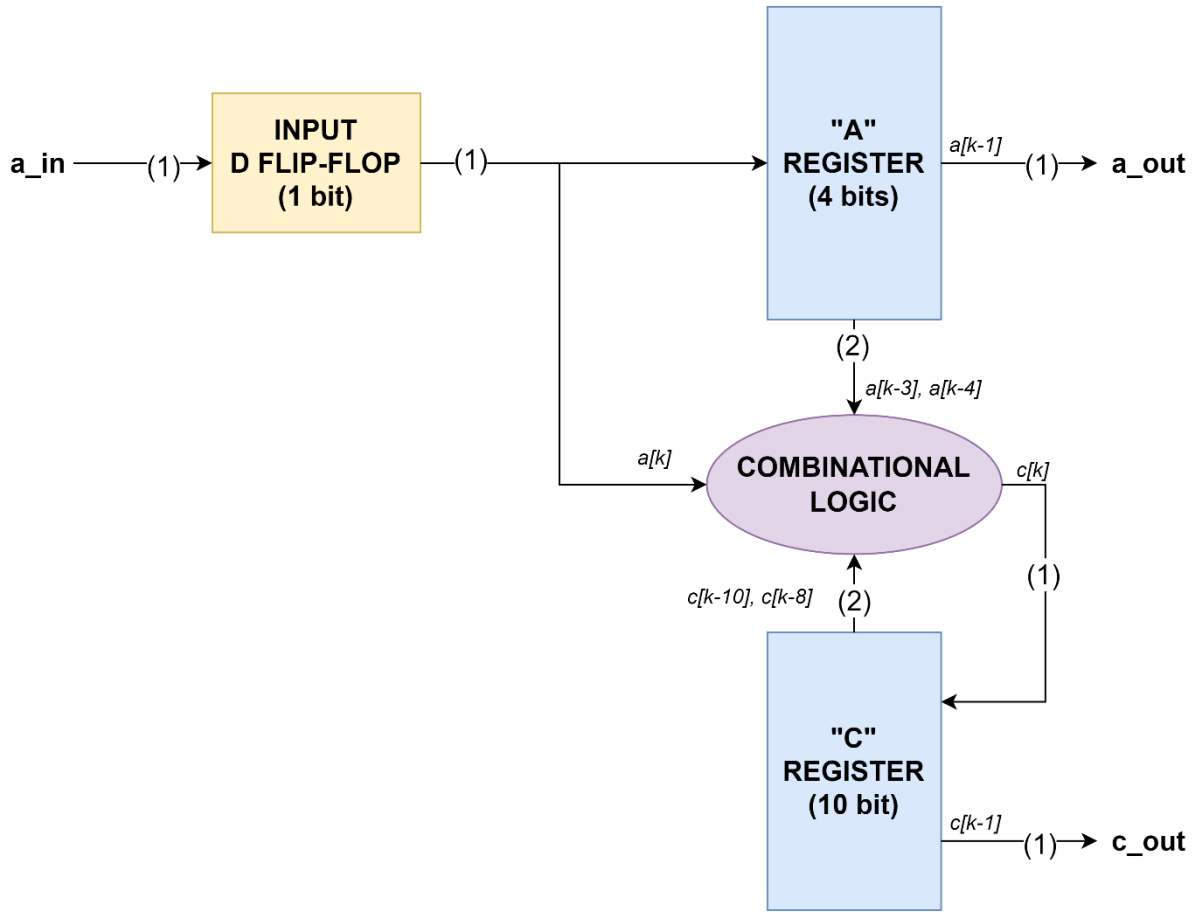
Optimization

The resources needed from the convolutional code generator could be further optimized by noticing that the two DFFs which stores the two output bits are redundant.

In fact, at clock cycle $\bar{k}$, the "output" DFFs are storing the values $a[\bar{k}-1]$ and $c[\bar{k}-1]$. These values are also stored in another part of the device, i.e. in **the first DFF of the shift registers**.

Thus, we can remove the two standalone "output" DFFs and exploit the shift registers for our purpose, by simply connecting the device output ports to the first bit of the two shift register.

In the next figure, a diagram of the optimized architecture is showed.



VHDL CODE

D Flip-Flop

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

library work;

-- Circuit which stores and keeps stable a single bit for a clock cycle.
--
-- It takes and input bit on the port "d" and stores it on a rising edge
-- of the clock. The stored bit is replicated on the "q" port for a clock cycle.
entity DFlipFlop is
    port(
        clock : in std_ulogic; -- external clock
        reset : in std_ulogic; -- reset, asynchronous, active high
        d : in std_ulogic; -- input bit
        q : out std_ulogic -- ouput bit
    );

end DFlipFlop;

-- Implementation of the D Flip-Flop
architecture beh of DFlipFlop is
begin

    DFF_proc: process(clock, reset)
        begin
            if(reset = '1') then -- Reset the circuit (output equals to zero)
                q <= '0';
            elsif(rising_edge(clock)) then -- Store a new bit
                q <= d;
            end if;
        end process;

end beh;

```

Shift Register

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

library work;

-- Data storing element. It uses a cascade of D Flip-Flop (DFF) where
-- the output of one flip-flop is connected to the input of the next.
-- In this way, each bit will move by one "position" at the beginning
-- of a clock cycle.
--
-- It takes a bit as data input per clock cycle
-- and outputs the state of the DFFs.
entity ShiftRegister is
    generic (
        size : natural := 5 -- number of DFFs
    );

    port (
        clock : in std_ulogic; -- external clock
        reset : in std_ulogic; -- reset, asynchronous, active high
        input_bit : in std_ulogic;
        output_bits : out std_ulogic_vector (size-1 downto 0)
    );
end ShiftRegister;

architecture beh of ShiftRegister is

    -- Declare D Flip-Flop component
    component DFlipFlop is
        port(
            clock : in std_ulogic; -- external clock
            reset : in std_ulogic; -- reset, asynchronous, active high
            d : in std_ulogic; -- input bit
            q : out std_ulogic -- ouput bit
        );

    end component DFlipFlop;

    -- This signal connect the output of each DFF
    -- to the input of the next one and to the output of
    -- the shift register.
    signal q_s : std_ulogic_vector (size-1 downto 0);

begin

    -- Generate D Flip-Flops
    GEN: for i in 0 to size-1 generate
        -- First DFF: input connected to the input of shift register
        FIRST: if (i = 0) generate
            FF1: DFlipFlop
            port map (
                clock => clock,
                reset => reset,
                d => input_bit,
                q => q_s(i)
            );
        end generate FIRST;

        -- All the other DFFs: input connected to the output of the previouse DFF
        INTERNAL: if (i > 0) generate
            FFi: DFlipFlop
            port map (
                clock => clock,
                reset => reset,
                d => q_s(i-1),
                q => q_s(i)
            );
        end generate INTERNAL;
    end generate GEN;

    -- Asynchronously assign the output of the DFFs
    -- to the output of the shift register
    output_bits <= q_s;

end beh;

```

Convolutional code generator

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

library work;

-- VHDL design a generator of convolutional code.
-- The chosen architecture is recursive (i.e. the future outputs depends on the
previous ones)
-- and systematic (the input bit stream is replicated as one of the output).
-- Our generator's rate is 1/2 and the system implements the relationship
-- c[k] = a[k] + a[k-3] + a[k-4] + c[k-8] + c[k-10]
entity convolutional_code_generator is
    port (
        clock : in std_ulogic;  -- external clock
        reset : in std_ulogic;  -- reset, asynchronous, active high
        a_in : in std_ulogic;   -- input bit stream
        a_out : out std_ulogic; -- 1st ouput. It's the replicated input
        c_out : out std_ulogic  -- 2nd output. It implements the relationship
    );

end convolutional_code_generator;



-- The generator is composed by two shift register, a combinational logic
-- that implements the c[k] relationship and one D-Flip-Flop for the input signal.
architecture beh of convolutional_code_generator is

    -- The shift registers are used to keep in memory the previous bits
    -- of the a[k] and c[k] streams.
    component ShiftRegister is
        generic (
            size : natural := 5
        );

        port (
            clock : in std_ulogic; -- external clock
            reset : in std_ulogic; -- reset, asynchronous, active high
            input_bit : in std_ulogic;
            output_bits : out std_ulogic_vector (size-1 downto 0)
        );
    end component ShiftRegister;
```

```
    -- The D-Flip-Flop is used to keep the input signal stable over a clock cycle
    component DFlipFlop is
        port(
            clock : in std_ulogic; -- external clock
            reset : in std_ulogic; -- reset, asynchronous, active high
            d : in std_ulogic; -- input bit
            q : out std_ulogic -- ouput bit
        );
    end component DFlipFlop;


    -- Size of the two shift-registers
    constant a_reg_size : natural := 4;
    constant c_reg_size : natural := 10;

    -- Signal from the "input" DFF
    signal a_in_dff_signal : std_ulogic;

    -- The output signals of the two shift registers.
    -- The indexes are structured so that
    -- a[k - i] = a_reg_out(i), 1 <= i <= 4
    -- c[k - i] = c_reg_out(i), 1 <= i <= 10
    -- where k is the current time.
    signal a_reg_out : std_ulogic_vector(a_reg_size downto 1);
    signal c_reg_out : std_ulogic_vector(c_reg_size downto 1);

    -- Signal generated by the encoding function
    signal conv_code_signal : std_ulogic;



begin
    -- D Flip-Flop for the input signal
    a_input_dff: DFlipFlop
    port map(
        clock => clock,
        reset => reset,
        d => a_in,
        q => a_in_dff_signal
    );

    -- Shift register that stores the last 4 a[...] values (from a[k-1] to a[k-4]).
    a_register: ShiftRegister
    generic map (
        size => a_reg_size
    )
    port map (
        clock => clock,
        reset => reset,
        input_bit => a_in_dff_signal,
        output_bits => a_reg_out
```

```
    );

    -- Shift register that stores the last 10 c[..] values
    -- (from c[k-1] to c[k-10]).
    c_register: ShiftRegister
    generic map (
        size => c_reg_size
    )
    port map (
        clock => clock,
        reset => reset,
        input_bit => conv_code_signal,
        output_bits => c_reg_out
    );



    -- Generate convolutional codes.
    -- The XOR tree is balanced in order to minimize the path
    -- between the shift register and the output DDF.
    conv_code_signal <= ( a_reg_out(3) xor a_reg_out(4) ) xor
                        ( a_in_dff_signal xor ( c_reg_out(8) xor c_reg_out(10) ) );

    -- Assign the outputs.
    a_out <= a_reg_out(1);
    c_out <= c_reg_out(1);

end beh;
```

VALIDATION

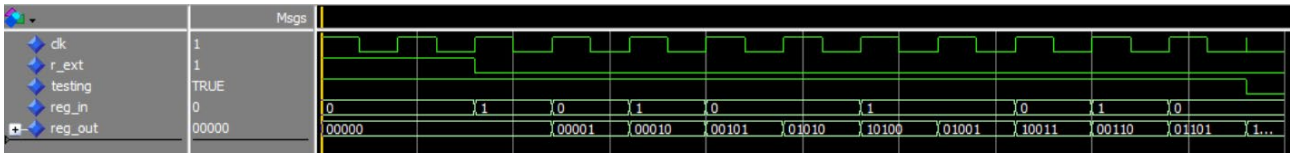
Testbench for the shift register

VHDL code

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  library work;
5
6  entity tb_shift_register is
7  end tb_shift_register;
8
9  -- Testbench for shift register
10 architecture beh of tb_shift_register is
11     constant clk_period     :   time    := 8 ns;
12
13     -- Component to test
14     component ShiftRegister is
15         generic (
16             size : natural := 5
17         );
18
19         port (
20             clock : in std_ulogic; -- external clock
21             reset : in std_ulogic; -- reset, asynchronous, active high
22             input_bit : in std_ulogic;
23             output_bits : out std_ulogic_vector (size-1 downto 0)
24         );
25     end component ShiftRegister;
26
27     -- Signals for simulation (clock, enable, reset, testing)
28     signal clk  :   std_ulogic   := '1';
29     signal r_ext    :   std_ulogic  := '1';
30     signal testing  :   boolean := true;
31
32     -- Input and output of the shift register
33     signal reg_in : std_ulogic := '0';
34     signal reg_out : std_ulogic_vector (4 downto 0);
35
```

```
36  begin
37      -- Generate clock
38      clk <= not clk after clk_period/2 when testing else '0';
39
40      reg: ShiftRegister
41      port map (
42              clock => clk,
43              reset => r_ext,
44              input_bit => reg_in,
45              output_bits => reg_out
46          );
47
48
49      -- TEST
50      stimulus: process
51      begin
52          -- Wait two clock cycles in order to reset the component's internal state
53          wait until rising_edge(clk);
54          wait until rising_edge(clk);
55          r_ext <= '0';
56
57          -- Start the testbench 1010011010
58          reg_in <= '1';
59          wait until rising_edge(clk);
60          reg_in <= '0';
61          wait until rising_edge(clk);
62          reg_in <= '1';
63          wait until rising_edge(clk);
64          reg_in <= '0';
65          wait until rising_edge(clk);
66          reg_in <= '0';
67          wait until rising_edge(clk);
68          reg_in <= '1';
69          wait until rising_edge(clk);
70          reg_in <= '1';
71          wait until rising_edge(clk);
72          reg_in <= '0';
73          wait until rising_edge(clk);
74          reg_in <= '1';
75          wait until rising_edge(clk);
76          reg_in <= '0';
77          wait until rising_edge(clk);
78
79          -- Stop the clock generator
80          testing <= false;
81      end process stimulus;
82
83  end beh;
84
```

Test on Modelsim



As we can see from the output (first DFF is the first bit from the right), the shift register behaves as intended.

C++ simulator code of the convolutional code generator

In order to properly validate the testbench of the convolutional code generator, a **C++ simulator** was written.

The simulator takes a string of bits as input of the generator and returns the correct sequence of generated bits.

Thanks to the simulator, I have been able to obtain the expected output for each input used during the testbench.

```cpp
#include <iostream>
#include <bitset>
#include <string>
using namespace std;

const int a_reg_size = 5; // Size of the register which stores the inputs
const int c_reg_size = 10; // Size of the register which stores the convolutional codes


// Generate the convolutional code sequence from an input bit string
string convolutional_code_generator (const string a_input_s)
{
    /*  a_register (it stores the sequence of old inputs)

        a[k] | a[k-1] | a[k-2] | a[k-3]
          0      1        2        3

        a(k - j) <=> a_reg[j]
    */
    bitset<a_reg_size> a_reg;

    /*  c_register (it stores the sequence of previously generated bits)

        c[k-1] | c[k-2] | c[k-3] ... c[k-10]
           0        1        2    ...   9

        c(k - j) <=> c_reg[j - 1]
    */
    bitset<c_reg_size> c_reg;

    // Generate code bits
    string convolutional_code = "";


```

```
    // For each input bit, generate a code bit
    for (unsigned int i = 0; i < a_input_s.length(); i++) {
        // Logging
        // cout << "a: " << a_reg << endl << "c: " << c_reg << endl;


        // Shift the "a" register and assign the current input to its first position
        a_reg <<= 1;
        a_reg[0] = (a_input_s.at(i) == '1');


        // Compute the new bit of the convolutional code
        // c[k] = a[k] xor a[k-3] xor a[k-4] xor c[k-8] xor c[k-10]
        bool c_k = a_reg[0] ^ a_reg[3] ^ a_reg[4] ^ c_reg[7] ^ c_reg[9];
        convolutional_code = convolutional_code.append((c_k ? "1" : "0"));

        // Shift the "c" register and assign the generated bit to its first position
        c_reg <<= 1;
        c_reg[0] = c_k;
    }

    return convolutional_code;
}
```

INPUT	OUTPUT
10101010101010101010 (“10” x10 times)	10110101110011011110
11111111111111111111 (“11” x 10 times)	11101111001010110001

Testbench for the convolutional code generator

VHDL code

Two different testbench were executed: the former tested the bit string 10101010101010, the latter tested the bit string 11111111111111.

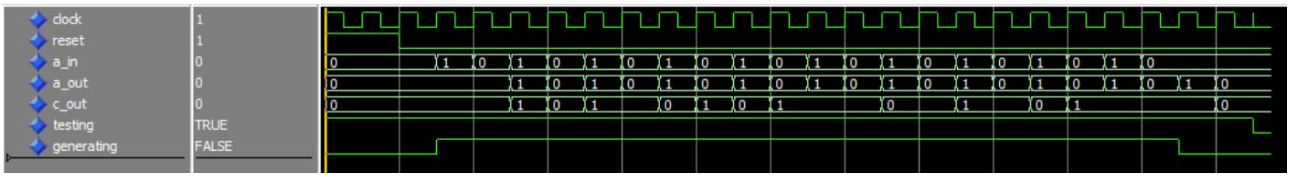
Since the code is the same apart from the lines which set the input signals, only the VHDL code of the first testbench is presented.

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

library work;

entity tb_generator_1 is
end tb_generator_1;

architecture beh of tb_generator_1 is

    constant clock_period : time := 10 ns;

    component convolutional_code_generator is
        port (
            clock : in std_ulogic; -- external clock
            reset : in std_ulogic; -- reset, asynchronous, active high
            a_in : in std_ulogic;
            a_out : out std_ulogic;
            c_out : out std_ulogic
        );
    end component convolutional_code_generator;

    signal clock : std_ulogic   := '1';
    signal reset : std_ulogic   := '1';
    signal a_in : std_ulogic := '0';
    signal a_out : std_ulogic;
    signal c_out : std_ulogic;

    signal testing : boolean := true;
    signal generating : boolean := false;

begin

    cc_generator: convolutional_code_generator
    port map (
        clock => clock,
        reset => reset,
        a_in => a_in,
        a_out => a_out,
        c_out => c_out
    );

    clock <= not clock after clock_period/2 when testing else '0';
```

```
    stimulus: process
    begin
        -- Reset the components
        wait until rising_edge(clock);
        wait until rising_edge(clock);
        reset <= '0';
        wait until rising_edge(clock);
        generating <= true;

        -- Test message '10101010101010' ('10' 10 times)
        for i in 1 to 10 loop
            a_in <= '1';
            wait until rising_edge(clock);

            a_in <= '0';
            wait until rising_edge(clock);
        end loop;

        generating <= false;
        a_in <= '0'; -- reset a_in

        -- wait two clock cycles for the complete convolutional code
        wait until rising_edge(clock);
        wait until rising_edge(clock);

        testing <= false; -- stop simulation

    end process stimulus;

end beh;
```

Test on Modelsim (1)

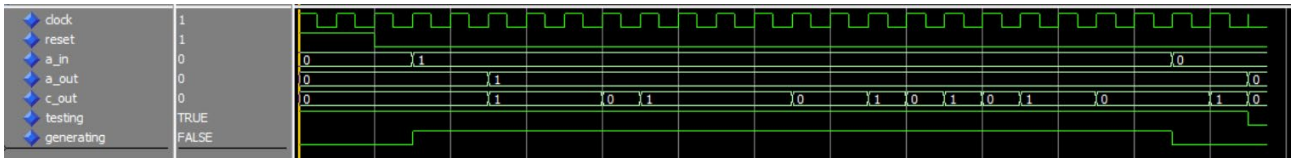
INPUT	TARGET
10101010101010101010 (“10” x10 times)	10110101110011011110



The generated output corresponds to the target. The testbench’s result is positive.

Test on Modelsim (2)

INPUT	TARGET
11111111111111111111 (“11” x 10 times)	11101111001010110001



The generated output corresponds to the target. The testbench’s result is positive.

SYNTHESIS

Synthesis through Vivado tool was made in out-of-context mode, so that it wasn't required to specify I/O pin mapping

Warnings

No warnings were generated during synthesis.

Synthesis

Status:	✓ Complete
Messages:	No errors or warnings
Part:	xc7z010clg400-1
Strategy:	Vivado Synthesis Defaults
Report Strategy:	Vivado Synthesis Default Reports
Incremental synthesis:	None

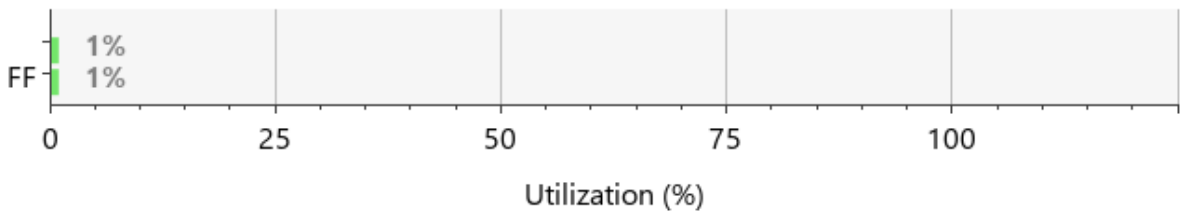
DRC

The only DRC violation is due to the fact we are not using a real ZyBO board.

Name	Details
All Violations (1)	
PS7 (1)	
Zynq requires PS7 block (1)	
PS7 (1)	
ZPS7-1 (1)	
ZPS7 #1	The PS7 cell must be used in this Zynq design in order to enable correct default configuration.

Resource utilization

Resource	Utilization	Available	Utilization %
LUT	1	17600	0.01
FF	15	35200	0.04



Timing evaluation

The timing summary with a clock period of 10 ns gave us a *WNS of 8.381 ns*.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	8,381 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:	14	Total Number of Endpoints:	14	Total Number of Endpoints:	15

All user specified timing constraints are met.

Thus, the optimal clock period is $10 - 8.381 = \mathbf{1.619\ ns}$ and the *maximum operating frequency* is **617.67 MHz**.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,000 ns	Worst Hold Slack (WHS):	0,254 ns	Worst Pulse Width Slack (WPWS):	0,309 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:	14	Total Number of Endpoints:	14	Total Number of Endpoints:	15

All user specified timing constraints are met.

Critical path

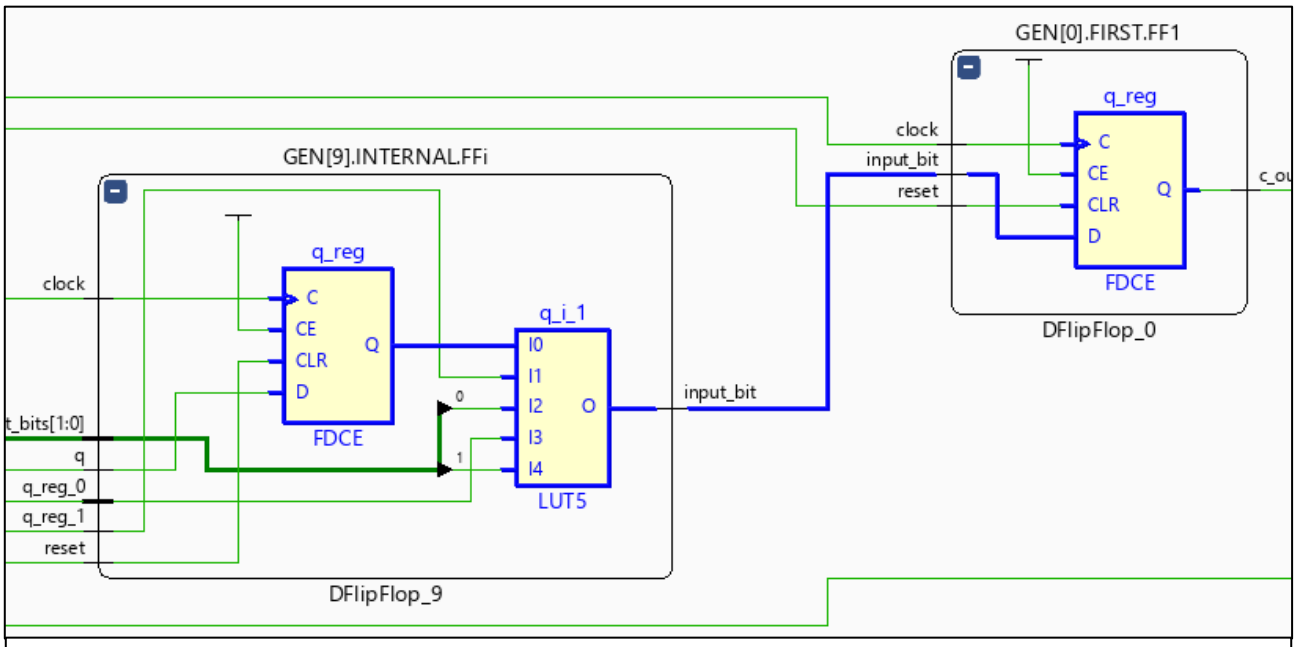
The critical path can be found by setting the clock period to 1.61 ns (lower then optimal clock period) and by analyzing the Vivado report.

Name	Slack ^1	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	R
Path 1	-0.009	1	2	1	c_register/GE...L.FFi/q_reg/C	c_register/G...FF1/q_reg/D	1.597	0.791	0.806	
Path 2	0.483	0	1	2	a_input_dff/q_reg/C	a_register/...F1/q_reg/D	0.841	0.496	0.345	
Path 3	0.483	0	1	2	a_register/GE...L.FFi/q_reg/C	a_register/G....FFi/q_reg/D	0.841	0.496	0.345	
Path 4	0.483	0	1	2	c_register/GE...L.FFi/q_reg/C	c_register/G....FFi/q_reg/D	0.841	0.496	0.345	
Path 5	0.494	0	1	1	a_register/GE....FF1/q_reg/C	a_register/G....FFi/q_reg/D	0.830	0.496	0.334	
Path 6	0.494	0	1	1	a_register/GE...L.FFi/q_reg/C	a_register/G....FFi/q_reg/D	0.830	0.496	0.334	
Path 7	0.494	0	1	1	c_register/GE....FF1/q_reg/C	c_register/G....FFi/q_reg/D	0.830	0.496	0.334	

Intra-Clock Paths: Setup

Summary	
Name	Path 1
Slack	-0.009ns
Source	c_register/GEN[9].INTERNAL.FFi/q_reg/C (rising edge-triggered cell FDCE clocked by CLOCK {rise@0.000ns fall(
Destination	c_register/GEN[0].FIRST.FF1/q_reg/D (rising edge-triggered cell FDCE clocked by CLOCK {rise@0.000ns fall@0.8
Path Group	CLOCK
Path Type	Setup (Max at Slow Process Corner)
Requirement	1.610ns (CLOCK rise@1.610ns - CLOCK rise@0.000ns)
Data Path Delay	1.597ns (logic 0.791ns (49.530%) route 0.806ns (50.470%))
Logic Levels	1 (LUT5=1)
Clock Path Skew	-0.049ns
Clock U...tainty	0.035ns

Timing summary of Path 1



Critical path (Path 1) in Schematic view

Power consumption

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