Università di Pisa

Computer Engineering

Electronics Systems

# Turbo Coding Interleaver

Project Report

Gianluca Gemini

Academic Year: 2021/2022

# Contents

# 1 Introduction

The **Turbo Coding Interleaver** is a digital circuit widely used in the **telecommunications** sector. The **turbo coding** is the application of algorithms for the correction of errors that may happen during the transmissions. It may happen that in a data flow some bits become corrupted, for example due to an interference, so these algorithms are used to **correct corrupted bits** and restore the original data flow. The error correction algorithms have best results if bits are **uniformly distributed** and are **uncorrelated**, however many consecutive bits in the data flow can be corrupted, for this reason the Interleaver is used.
The Interleaver **mixes the flow's bits** before the transmission, so that if a sequence of bits becomes corrupted the **errors** will be **distributed** in all the flow, then there is an **Deinterleaver** in the reception side that reorder the flow. Thus the Inteleaver in general receives as **input** a data flow and return as **output** the same data flow but with mixed bits.

## 1.1 Design Specification

The interleaver taken in account shall have the following requirements:

- **one bit** as **input**
- **one bit** as **output**
- is synchronized to a **T period**, such that for each T it receives a new bit
- has to manage a **1024 flow bits**, received and transmitted **one by one**
- the bit mixing have to be in accord with the following **formula**

$$x_{out}(i) = x_{in}(|45 + i * 3|_{1024})$$

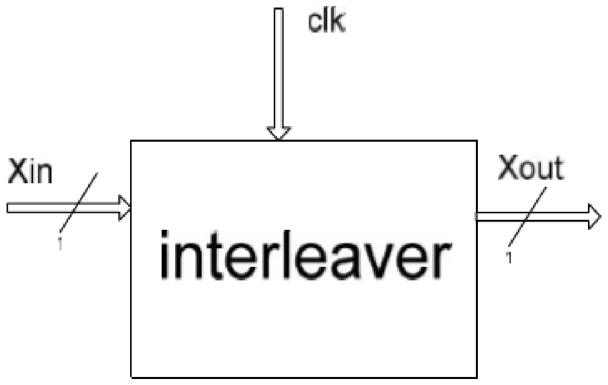


Figure 1: the Interleaver specification design

# 2 Architecture Description

## 2.1 Block diagram

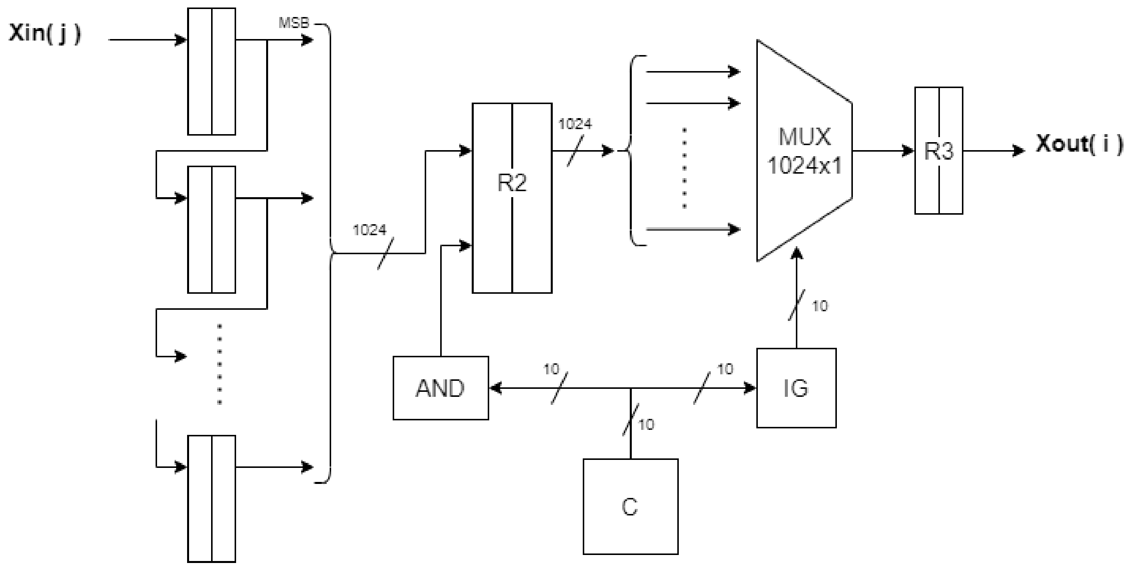


Figure 2: block diagram overview

In order to project the above architecture it's needed to start with the function to **shuffle** the bits.

$$x_{out}(i) = x_{in}(|45 + i * 3|_{1024})$$

It's possible to see that an input bit **can't be sent immediately** to output, so the first bit to transmit is the 45th of the input, the second is the 48th, the third is the 51st and so on. This meaning that it's needed to **keep in memory** the bits as long as the moment in witch they have to be transmitted. To solve this problem the circuit it's divided in **two autonomous parts**, one for storing the input bits and one to transmit them shuffled.

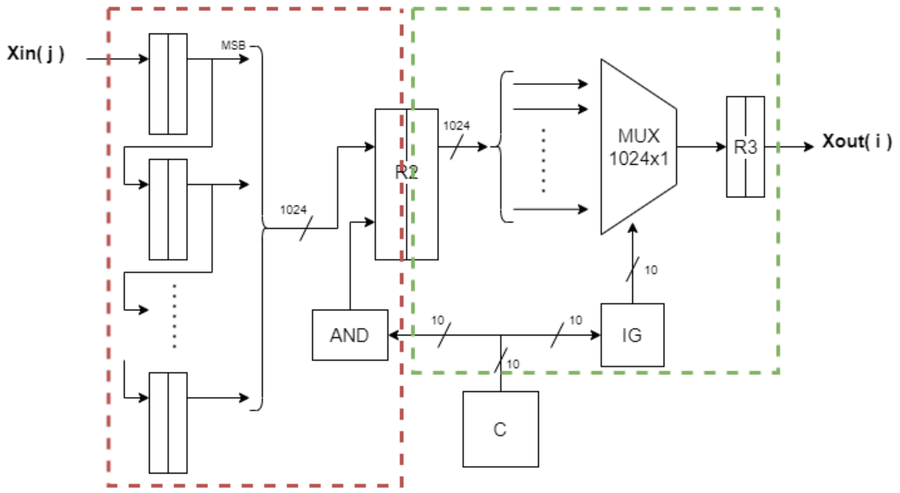


Figure 3: the two parts of the diagram

When the circuit starts **1024 clock cycles** are needed before the *Xout(i)* values are valid, infact the **R2 register** stores all the 1024 bit flow, going to **transparency** until after all these bits will be arrived from input. So while the **red part** of the device is receiving a 1024 bit block, the **green part** is transmitting the bits of the previous block.

## 2.2 The Red Part

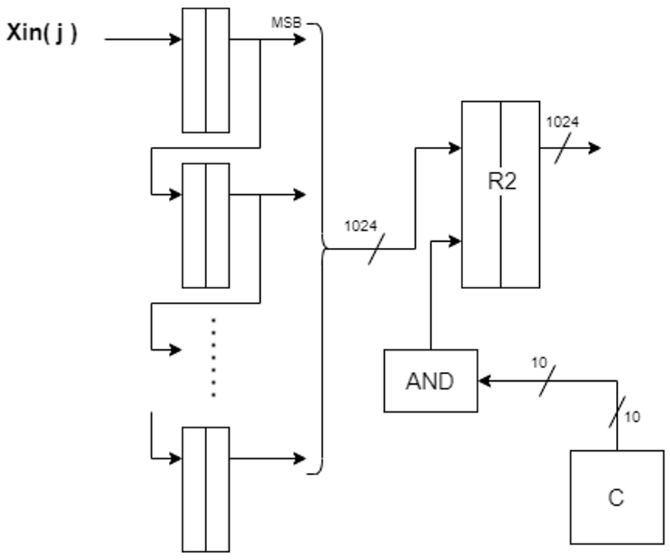


Figure 4: the red part

In order to store the input bits in the arriving order, **1024 registers** with **one bit** are used concatenated and **always in transparency**. In each clock cycle the bit are moved to the register below, after 1024 cycles the last register contains the first received bit and the first register contains the last bit. From now on the 1024 registers are named **R1**.

Note that the first register of R1 is connected with the first input bit of the 1024 bit register R2.

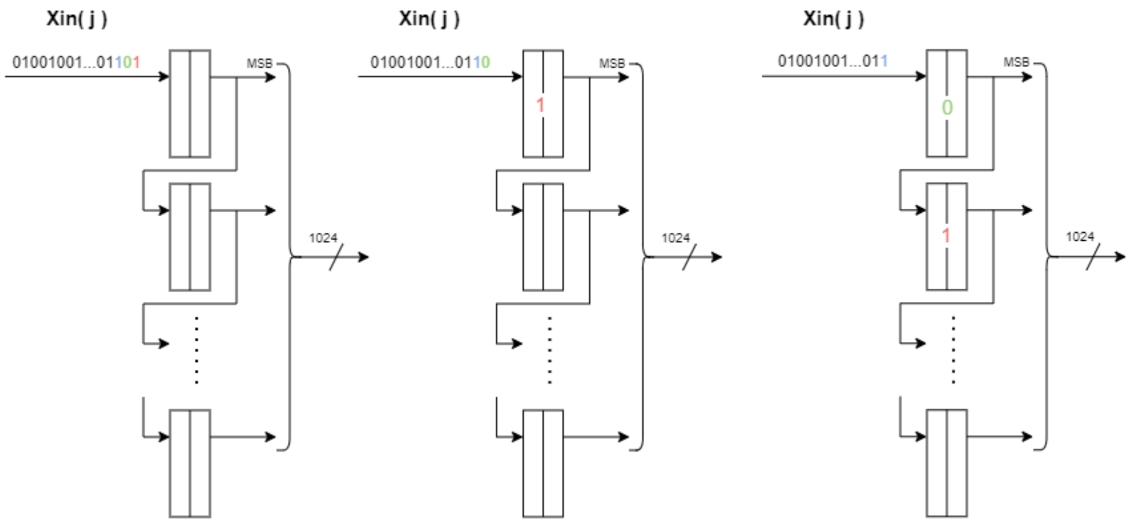


Figure 5: the R1 workflow

The **C block** is a counter witch allows to the **R2** register to go in transparency only when all registers of R1 were filled through an **enable** wire, having to count from 0 to 1023 it's a *10 bit* counter.
When the R2's enable is *1* R2 can store the 1024 bits from R1, this enable is driven to an **AND port** that takes in input the 10 bits output of the **Counter**, so when the Counter output is 1023 (*1111111111*) the AND port outputs *1* and R2 goes to transparency.

## 2.3 Counter

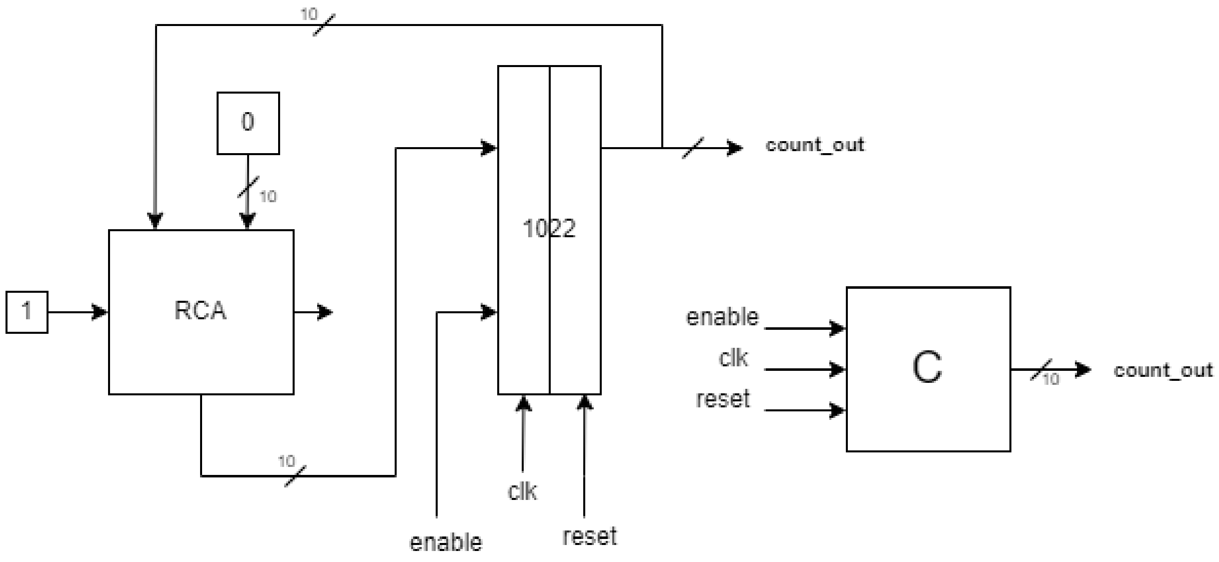


Figure 6: the counter block diagram

The **Counter**, as already mentioned, has the task to count the clock cycles from 0 to 1023, however, due to R1, it's **unsynchrinized** with the inputs, in particular is one cycle back. For this reason the 10 bit register inside the counter is **initialized** with the value *1022*, so that at the first clock the counter returns 1023 and at the second returns 0 for the overflow of the Ripple Carry Adder (RCA). The **Ripple Carry Adder** has the following inputs and outputs:

- The carry in is *1*
- The first addendum is the register output
- The second addendum is *0000000000*
- The carry out is ignored
- The sum is used as the input of the register

## 2.4 The Green Part

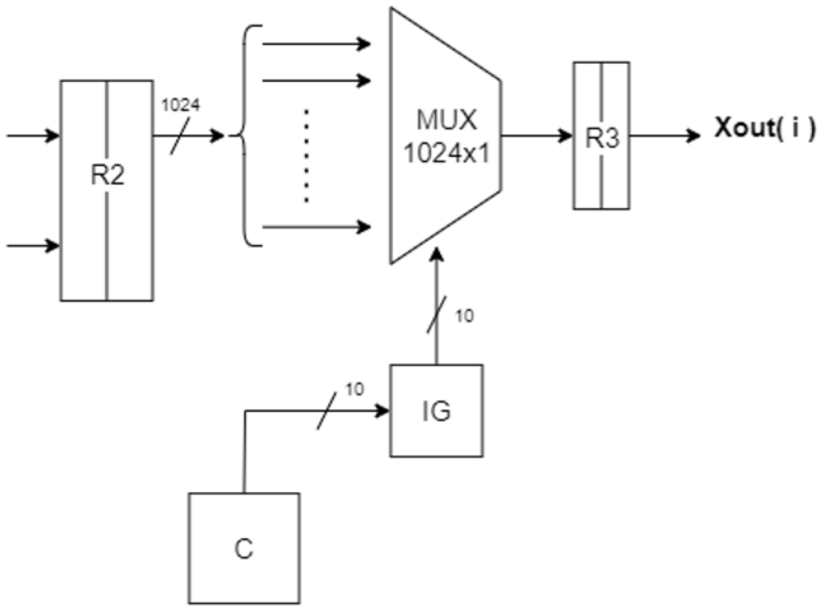


Figure 7: the green part

This second part produces the output deciding what bit has to be transmit. After that **R2** has stored an entire 1024 bits block, it will be **unchanged** for the next 1024 clock cycles, during this period the bits are be transmitted according to the **mixing function**, this function is implemented by the combinational network **Index Generator** (IG). The Counter, the same used for the first part of the circuit, generates the index $i$ and the IG outputs the index $j$ used to drive the **1024x1 multiplexer** witch links the correct bit from R2 to the output.

## 2.5 Index Generator Combinational Network

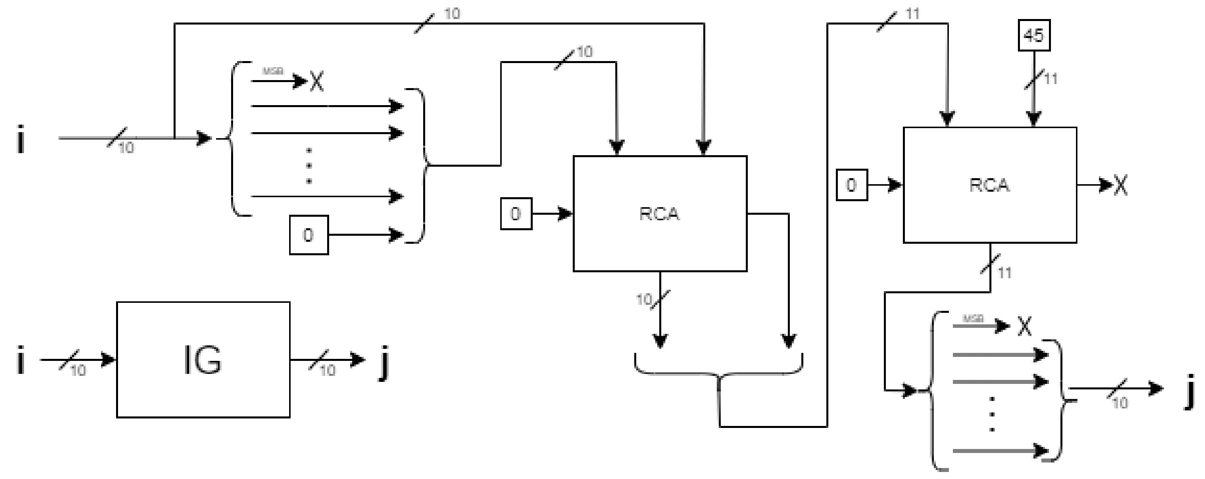


Figure 8: the Index Generator block diagram

This combinational network implements the following function

$$x_{out}(i) = x_{in}(|45 + i * 3|_{1024})$$

In order to avoid to use a multiplier, the function can rewritten as follows

$$x_{out}(i) = x_{in}(|45 + (i * 2 + i)|_{1024})$$

The operations to be performed are broken up into these parts:

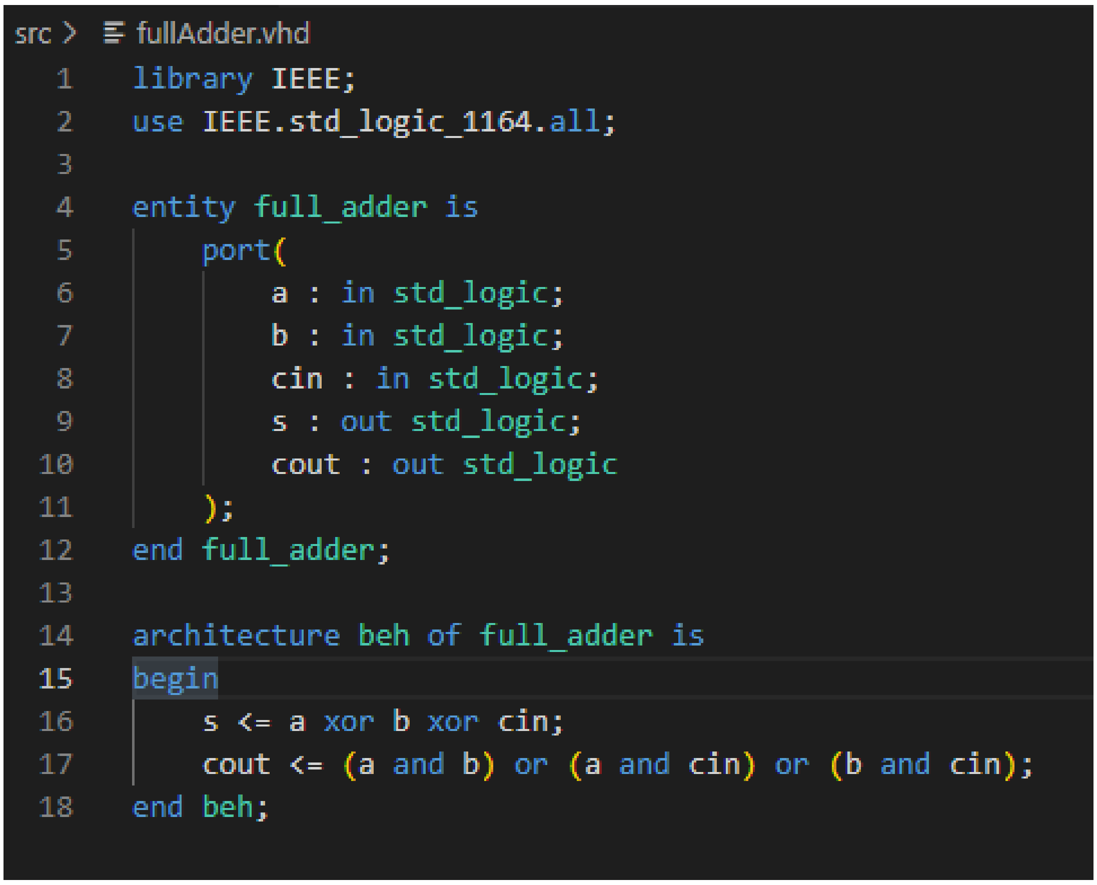
- A **left shift** of $i$ to obtain *i*2*
- Sum of $i$ and *i*2* using a **10 bit ripple carry adder**, the output is concatenated with the carry out for the **overflow**
- Another sum with an **11 bit ripple carry adder**, the addends are the sum of the previous RCA and the value *45*
- The most significant bit is removed from the result to apply the 1024 **modulus operator**
- The **carry out** of the second RCA is **ignored**

# 3 VHDL Code Analysis

The VHDL code analysis will be made starting from the base components until the entire Interleaver.

## 3.1 Full Adder

The **Full Adder** (FA) is the Ripple Carry Adder (RCA) base component, infact is a *1 bit adder*.



src > fullAdder.vhd

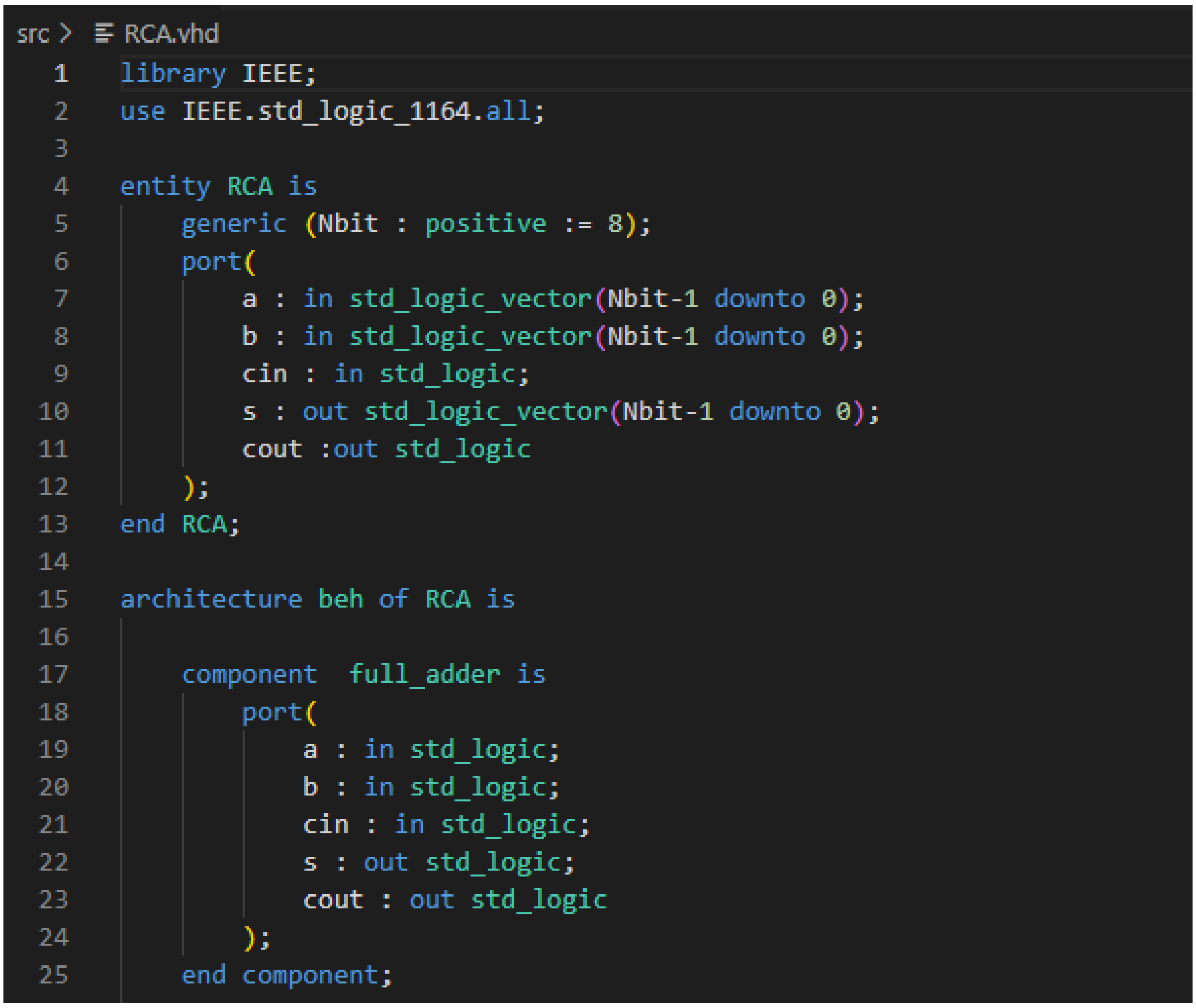
```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity full_adder is
    port(
        a : in std_logic;
        b : in std_logic;
        cin : in std_logic;
        s : out std_logic;
        cout : out std_logic
    );
end full_adder;

architecture beh of full_adder is
begin
    s <= a xor b xor cin;
    cout <= (a and b) or (a and cin) or (b and cin);
end beh;
```

Figure 9: the Full Adder source code

As can be seen, *a* and *b* are the **addends**, *cin* is the **carry in input** and *cout* is the **carry in output**, these two last are fundamental to build a Full Adder chain like the RCA. The *cout* is *1* if there is an overflow, that is when the **sum operation** produces a result not calculable on *1 bit*. The architecture describes the needed logic in order to produce the sum *s* and the output carry.

## 3.2 Ripple Carry Adder

The **Ripple Carry Adder** (RCA) is a Full Adder chain, exactly as many as bits supports.

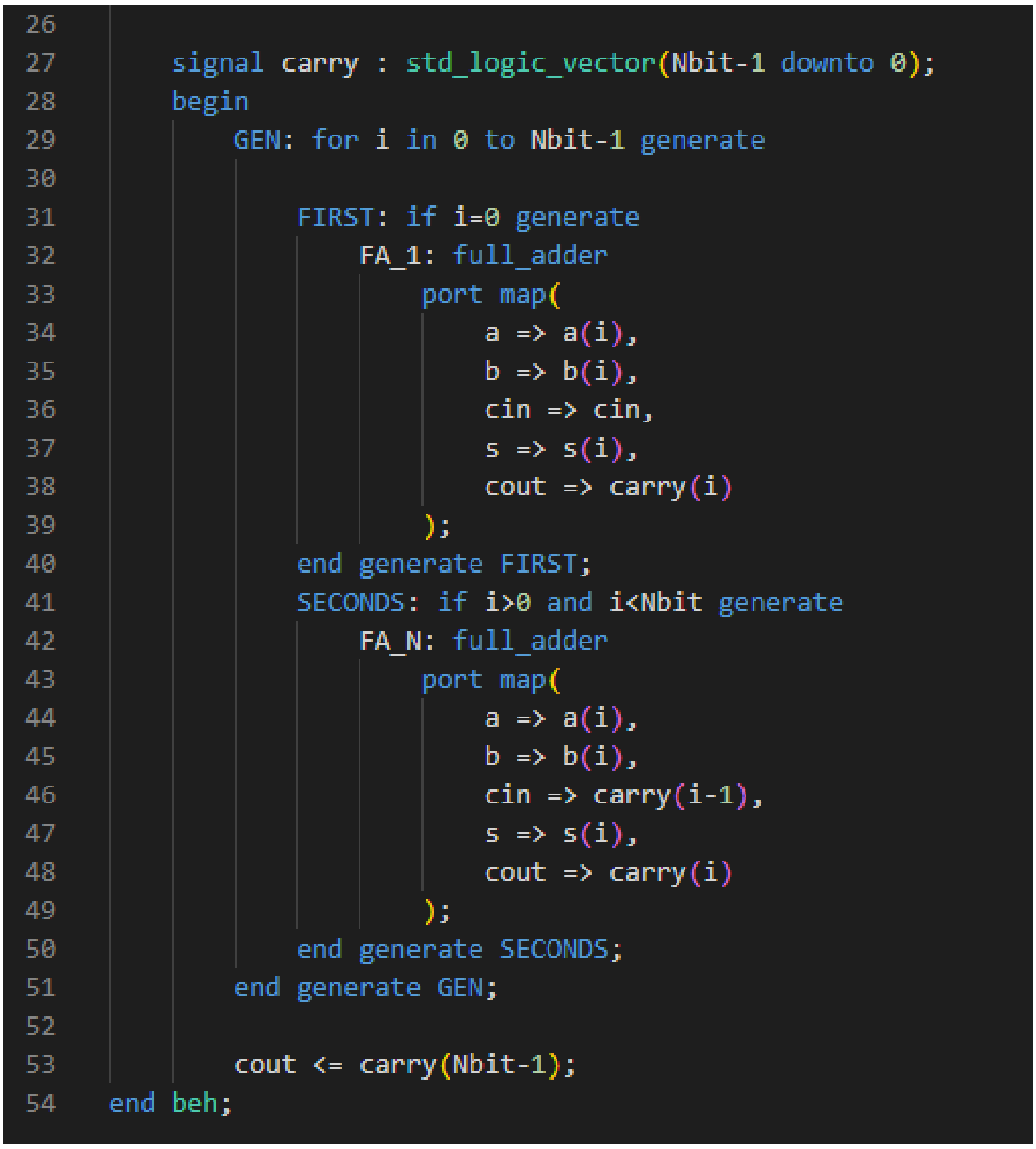


src > RCA.vhd

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

entity RCA is
    generic (Nbit : positive := 8);
    port(
        a : in std_logic_vector(Nbit-1 downto 0);
        b : in std_logic_vector(Nbit-1 downto 0);
        cin : in std_logic;
        s : out std_logic_vector(Nbit-1 downto 0);
        cout :out std_logic
    );
end RCA;

architecture beh of RCA is

    component  full_adder is
        port(
            a : in std_logic;
            b : in std_logic;
            cin : in std_logic;
            s : out std_logic;
            cout : out std_logic
        );
    end component;
```

Figure 10: the Ripple Carry Adder source code 1

It takes as **input** the *1* bit carry *cin*, the *Nbit* bits addends *a* and *b*, the **output** is the sum *s* calculated on the same bit of the addends, and the *1* bit output carry *cout* for the overflow.



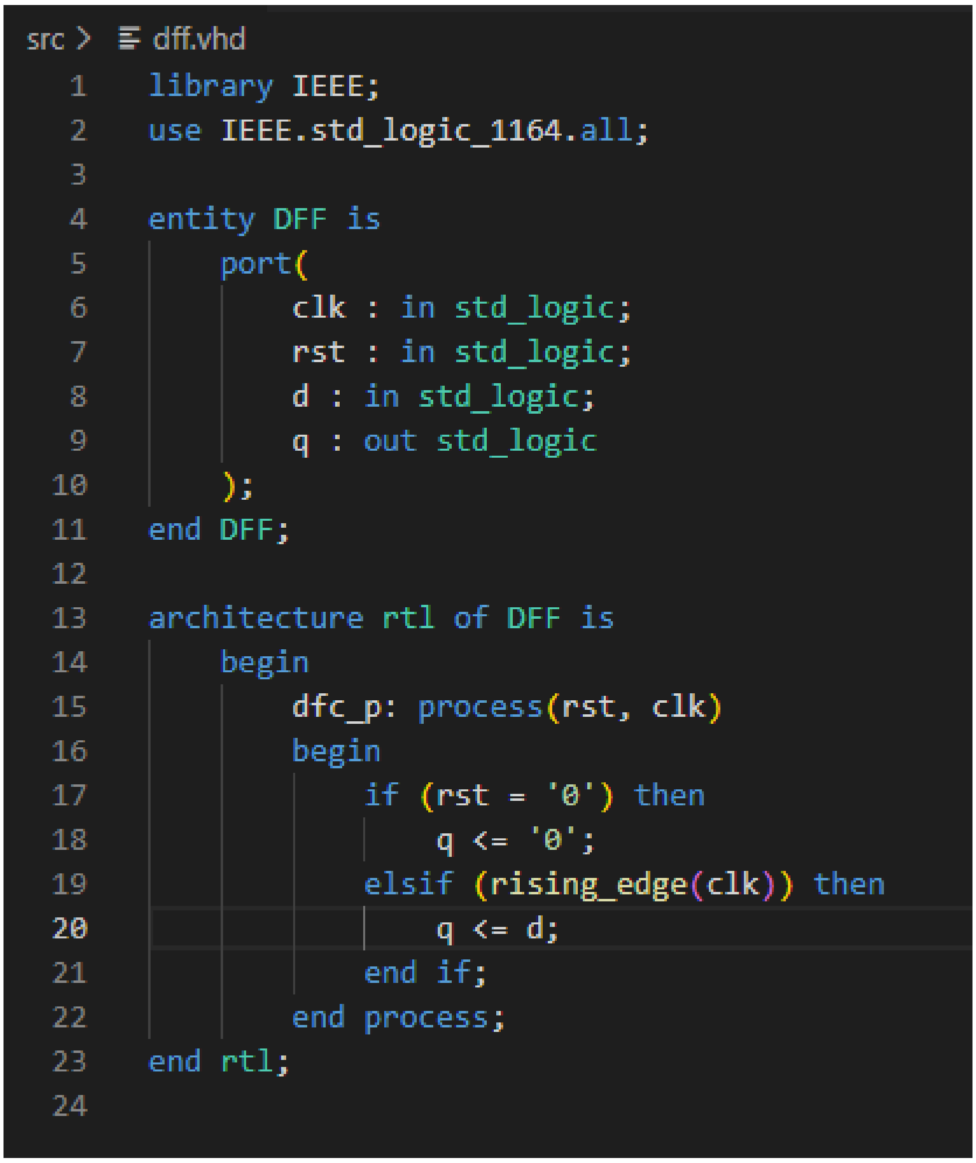
```vhdl

    signal carry : std_logic_vector(Nbit-1 downto 0);
    begin
        GEN: for i in 0 to Nbit-1 generate

            FIRST: if i=0 generate
                FA_1: full_adder
                    port map(
                        a => a(i),
                        b => b(i),
                        cin => cin,
                        s => s(i),
                        cout => carry(i)
                    );
            end generate FIRST;
            SECONDS: if i>0 and i<Nbit generate
                FA_N: full_adder
                    port map(
                        a => a(i),
                        b => b(i),
                        cin => carry(i-1),
                        s => s(i),
                        cout => carry(i)
                    );
            end generate SECONDS;
        end generate GEN;

        cout <= carry(Nbit-1);
end beh;
```

Figure 11: the Ripple Carry Adder source code 2

*Nbit* **Full Adders** are used in order to describe the RCA, the first and second addends of i-th FA are assigned respectively to the i-th bit of *a* and the i-th bit of *b*, the output carry of each FA is used as input carry of the next FA, the **input carry** of the first FA is *0*. The **output carry** of the last FA is *1* if RCA sum can't be calculated on Nbit bits, **the sum** is all **FAs' output concatenation**.

## 3.3 D Flip Flop

The **D Flip Flop** (DFF) is the needed component to **store one bit**.



src > dff.vhd

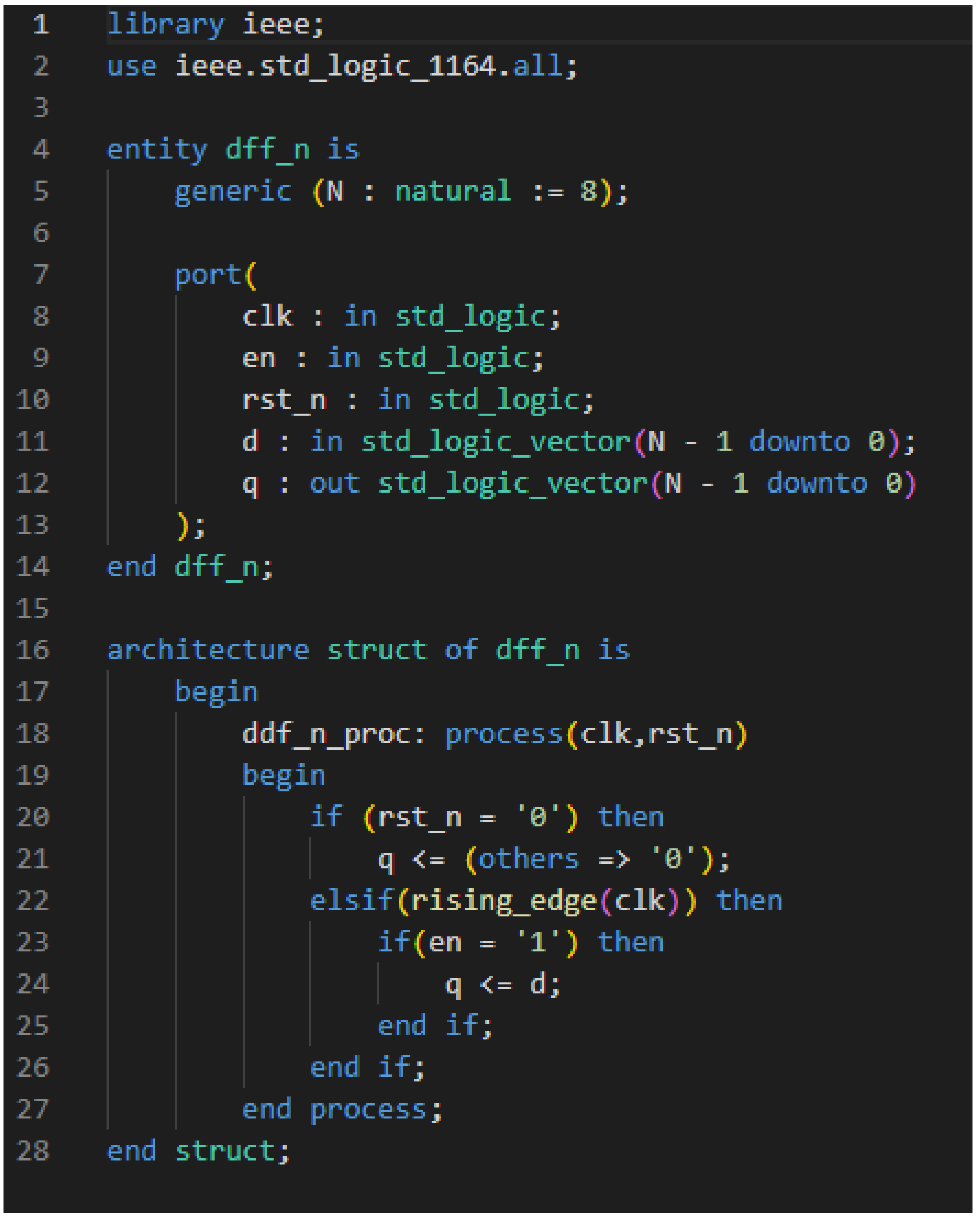
```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity DFF is
    port(
        clk : in std_logic;
        rst : in std_logic;
        d : in std_logic;
        q : out std_logic
    );
end DFF;

architecture rtl of DFF is
    begin
        dfc_p: process(rst, clk)
        begin
            if (rst = '0') then
                q <= '0';
            elsif (rising_edge(clk)) then
                q <= d;
            end if;
        end process;
end rtl;

```

Figure 12: the D Flip Flop source code

It's **synchronized** to a **edge clock** *clk* and has a **reset** *rst* that if it is *0* reinitialize the DFF to a default value. On clock arrival the DFF goes to **transparency**, that is the mode in witch the DFF can store the input bit from *d*. The **output** *q* shows the bit stored in the last transparency cycle. Generally this device has an **enabler** that is used to enable or disable the transparency mode, however in the Interleaver all 1 bit DFFs has to be **always in transparency**.

## 3.4 D Flip Flop with N bits

The **D Flip Flop with N bits** (DFF_n) is a variant of the previous DFF, infact it can **store more bits**, moreover it has the **enabler** (rows 9 and 23).
This component is used as **R2** register, that is the one has keep **in memory 1024 bits** going to **transparency** only each 1024 clock cycles, that means that during 1024 cycles the enabler *en* has to be *0*.



```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3
 4  entity dff_n is
 5      generic (N : natural := 8);
 6
 7      port(
 8          clk : in std_logic;
 9          en : in std_logic;
10          rst_n : in std_logic;
11          d : in std_logic_vector(N - 1 downto 0);
12          q : out std_logic_vector(N - 1 downto 0)
13      );
14  end dff_n;
15
16  architecture struct of dff_n is
17      begin
18          ddf_n_proc: process(clk,rst_n)
19          begin
20              if (rst_n = '0') then
21                  q <= (others => '0');
22              elsif(rising_edge(clk)) then
23                  if(en = '1') then
24                      q <= d;
25                  end if;
26              end if;
27          end process;
28  end struct;
```

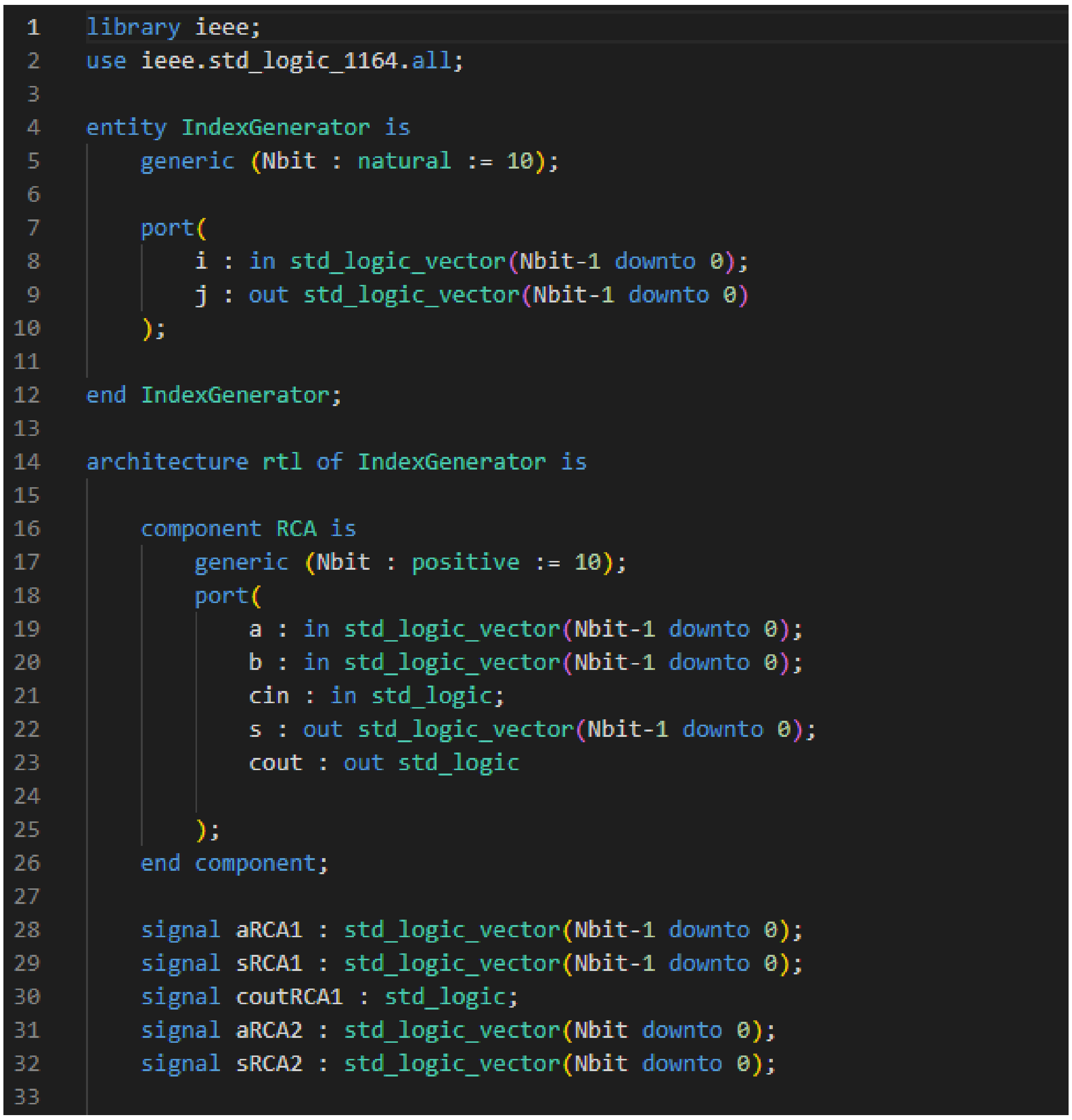
Figure 13: the DFF_n source code

## 3.5 Index Generator Combinational Network

The **Index Generator** (IG) has to implement the function

$$x_{out}(i) = x_{in}(|45 + i * 3|_{1024})$$

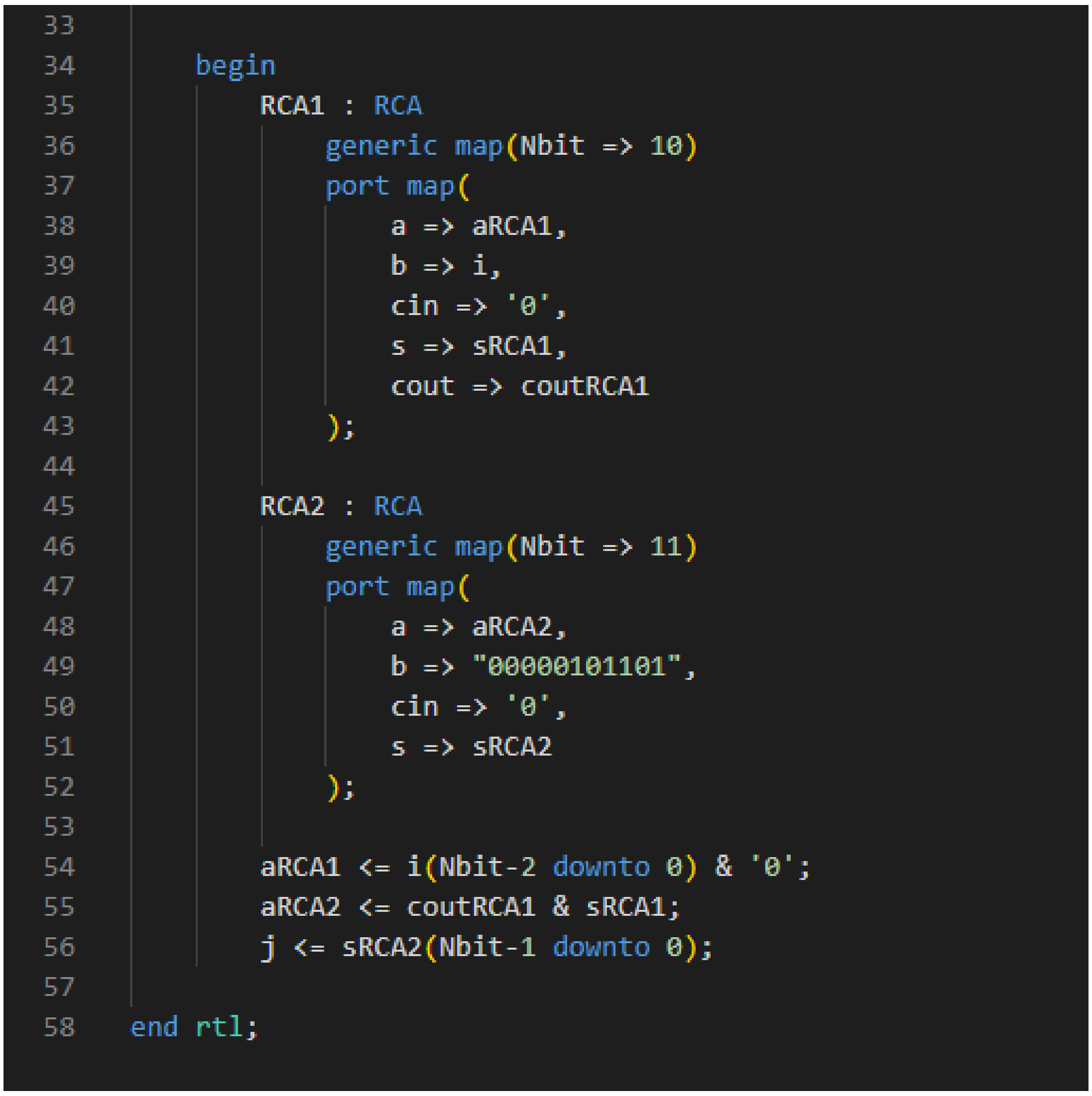
Being a **combinational network** it's not synchronized to a clock, infact the output state only depends on the input.



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

entity IndexGenerator is
    generic (Nbit : natural := 10);

    port(
        i : in std_logic_vector(Nbit-1 downto 0);
        j : out std_logic_vector(Nbit-1 downto 0)
    );

end IndexGenerator;

architecture rtl of IndexGenerator is

    component RCA is
        generic (Nbit : positive := 10);
        port(
            a : in std_logic_vector(Nbit-1 downto 0);
            b : in std_logic_vector(Nbit-1 downto 0);
            cin : in std_logic;
            s : out std_logic_vector(Nbit-1 downto 0);
            cout : out std_logic

        );
    end component;

    signal aRCA1 : std_logic_vector(Nbit-1 downto 0);
    signal sRCA1 : std_logic_vector(Nbit-1 downto 0);
    signal coutRCA1 : std_logic;
    signal aRCA2 : std_logic_vector(Nbit downto 0);
    signal sRCA2 : std_logic_vector(Nbit downto 0);

```

Figure 14: the Index generator source code 1

As shown by the block diagram seen in the previous chapter, **two RCAs** are needed, the first one on *10 bits* and the second on *11 bits*.
The **first RCA** is used to execute the operation $i*2 + i$, the **second one** it's used to **sum** the first RCA's result to *45*.
To the *54 row* it can be seen the **shift operation** to obtain $i*2$, to the *55 row* there is the concatenation of the first RCA's outputs so as to d**iscard an eventual overflow bit**, finally to the *56 row* the sum from the second RCA is used as the **IG's output**. Note that the output carry of the second RCA are not taken as output with the sum in order to apply the 1024 **modulus operator**.

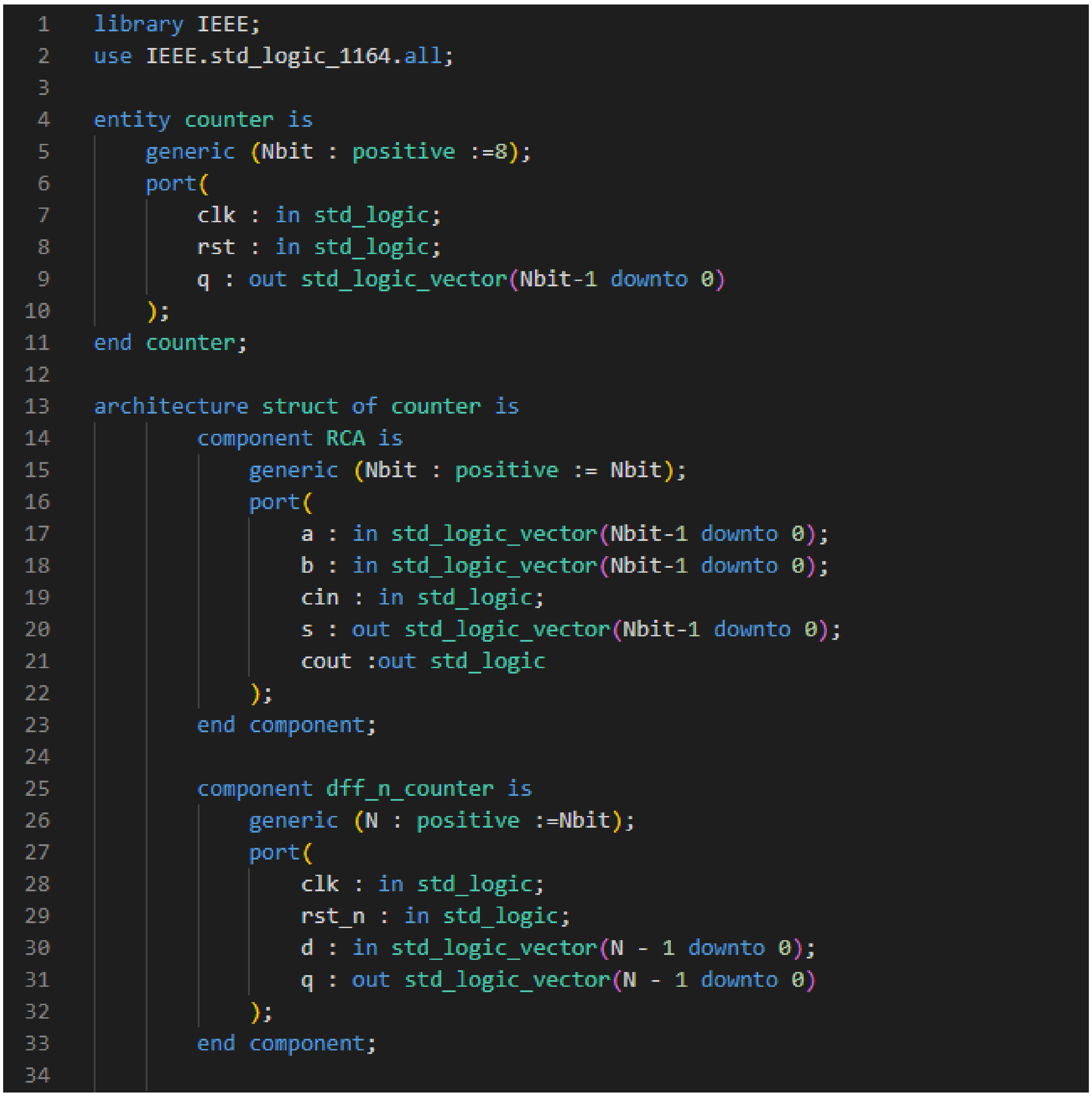


```
33
34      begin
35          RCA1 : RCA
36              generic map(Nbit => 10)
37              port map(
38                  a => aRCA1,
39                  b => i,
40                  cin => '0',
41                  s => sRCA1,
42                  cout => coutRCA1
43              );
44
45          RCA2 : RCA
46              generic map(Nbit => 11)
47              port map(
48                  a => aRCA2,
49                  b => "00000101101",
50                  cin => '0',
51                  s => sRCA2
52              );
53
54          aRCA1 <= i(Nbit-2 downto 0) & '0';
55          aRCA2 <= coutRCA1 & sRCA1;
56          j <= sRCA2(Nbit-1 downto 0);
57
58  end rtl;
```

Figure 15: the Index generator source code 2

## 3.6 Counter

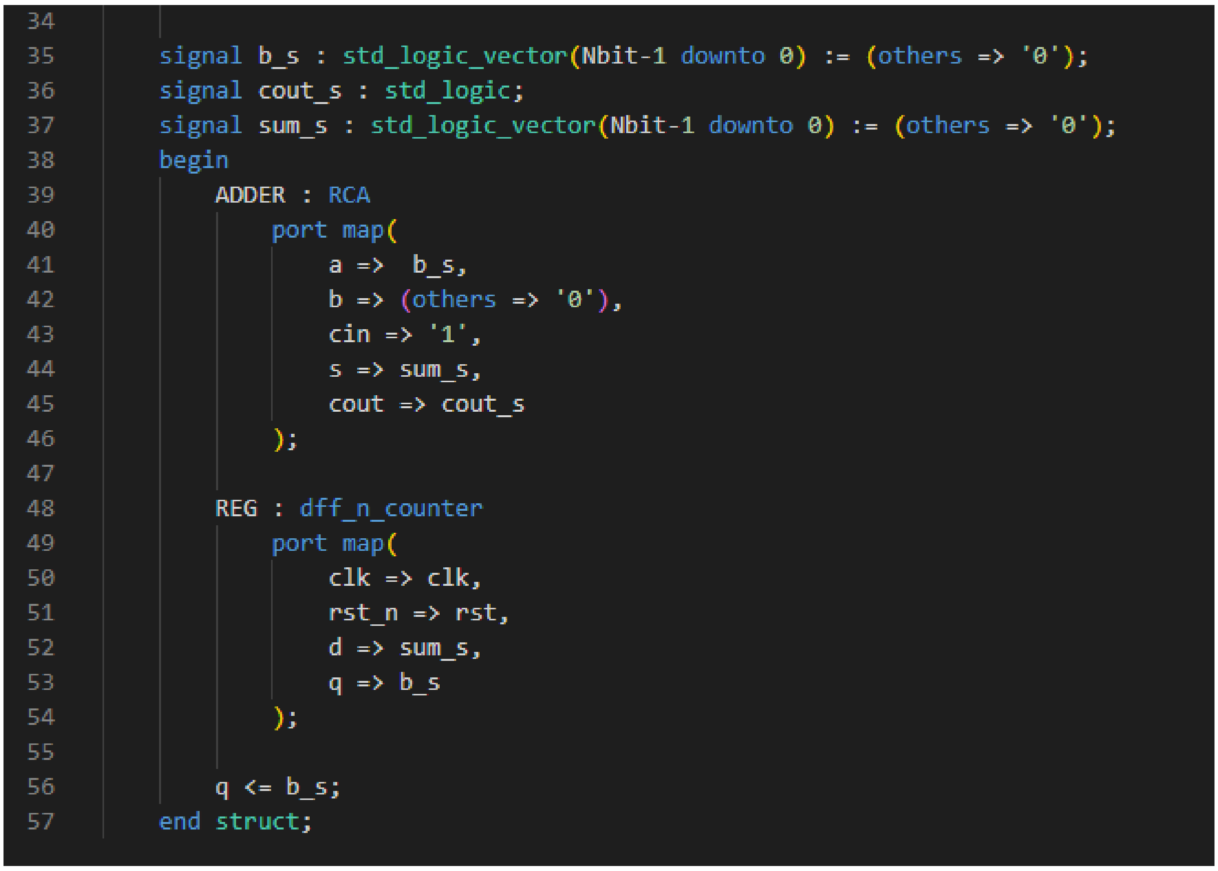
The **Counter** is used to **count** the Interleaver's input bit, and generate the **IG's input**. It needs to a 10 bits adder **RCA** and 10 bits register **DFF_n**.



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

entity counter is
    generic (Nbit : positive :=8);
    port(
        clk : in std_logic;
        rst : in std_logic;
        q : out std_logic_vector(Nbit-1 downto 0)
    );
end counter;

architecture struct of counter is
        component RCA is
            generic (Nbit : positive := Nbit);
            port(
                a : in std_logic_vector(Nbit-1 downto 0);
                b : in std_logic_vector(Nbit-1 downto 0);
                cin : in std_logic;
                s : out std_logic_vector(Nbit-1 downto 0);
                cout :out std_logic
            );
        end component;

        component dff_n_counter is
            generic (N : positive :=Nbit);
            port(
                clk : in std_logic;
                rst_n : in std_logic;
                d : in std_logic_vector(N - 1 downto 0);
                q : out std_logic_vector(N - 1 downto 0)
            );
        end component;

```

Figure 16: the counter source code 1

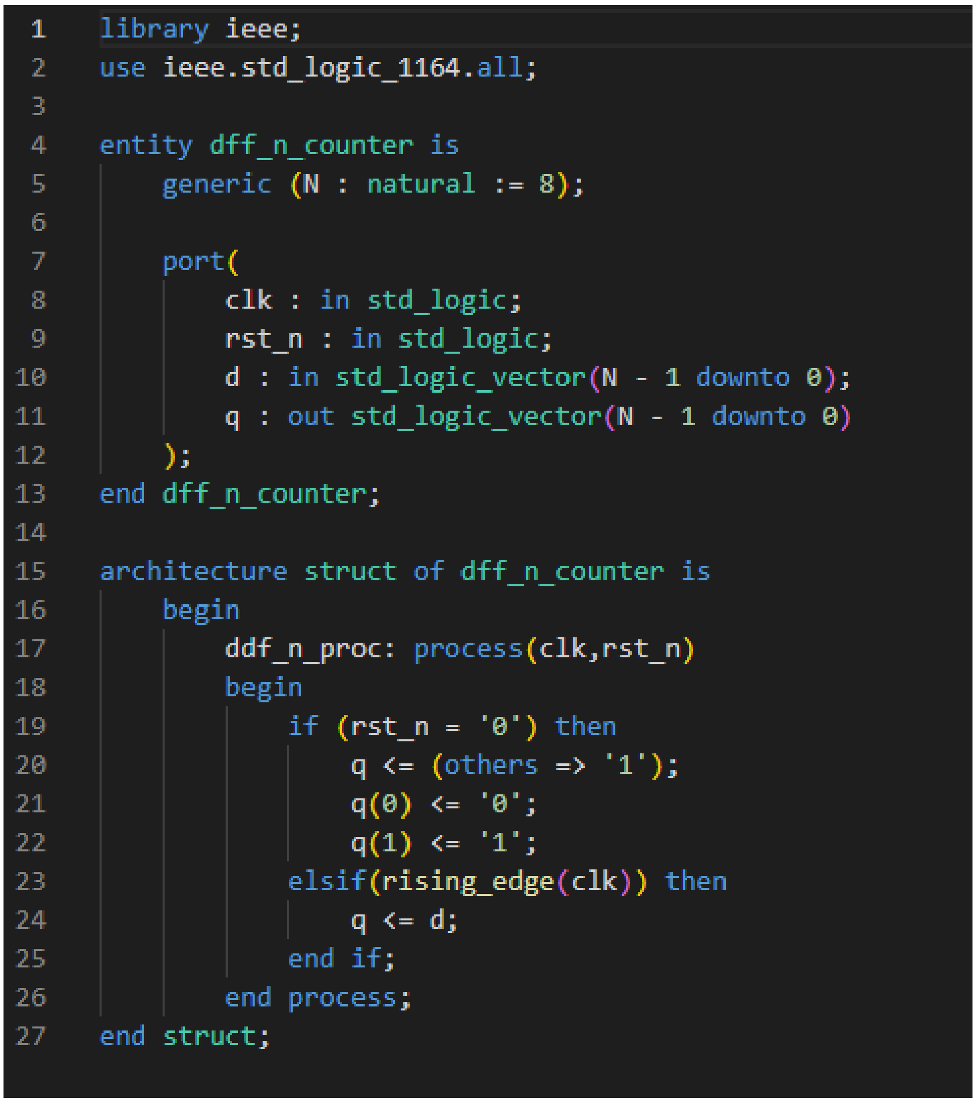
The adder's **input carry is always 1** to increment the addends. The **first addend** is *0000000000*, the **other addend** is the register's output. The **RCA's output**, excluded the carry, is the **DFF's input**.



```vhdl

    signal b_s : std_logic_vector(Nbit-1 downto 0) := (others => '0');
    signal cout_s : std_logic;
    signal sum_s : std_logic_vector(Nbit-1 downto 0) := (others => '0');
    begin
        ADDER : RCA
            port map(
                a =>  b_s,
                b => (others => '0'),
                cin => '1',
                s => sum_s,
                cout => cout_s
            );

        REG : dff_n_counter
            port map(
                clk => clk,
                rst_n => rst,
                d => sum_s,
                q => b_s
            );

        q <= b_s;
    end struct;
```

Figure 17: the counter source code 2

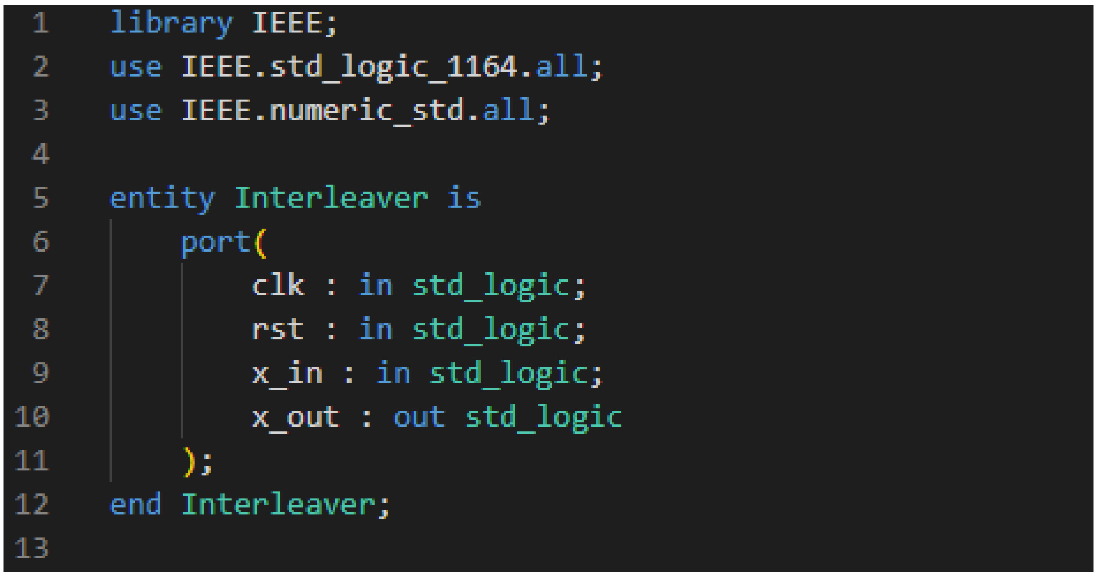
The register used in the counter doesn't have the **enabler**, so is **always in transparency mode**, infact the counter has to be **continuously working**. The register is **initialized to 1022** (*rows 20-24*) in order to solve the **synchronization problem** seen in the previous chapter.



```
1   library ieee;
2   use ieee.std_logic_1164.all;
3
4   entity dff_n_counter is
5       generic (N : natural := 8);
6
7       port(
8           clk : in std_logic;
9           rst_n : in std_logic;
10          d : in std_logic_vector(N - 1 downto 0);
11          q : out std_logic_vector(N - 1 downto 0)
12      );
13  end dff_n_counter;
14
15  architecture struct of dff_n_counter is
16      begin
17          ddf_n_proc: process(clk,rst_n)
18          begin
19              if (rst_n = '0') then
20                  q <= (others => '1');
21                  q(0) <= '0';
22                  q(1) <= '1';
23              elsif(rising_edge(clk)) then
24                  q <= d;
25              end if;
26          end process;
27  end struct;
```

Figure 18: the DFF_n inside the counter

## 3.7 Interleaver

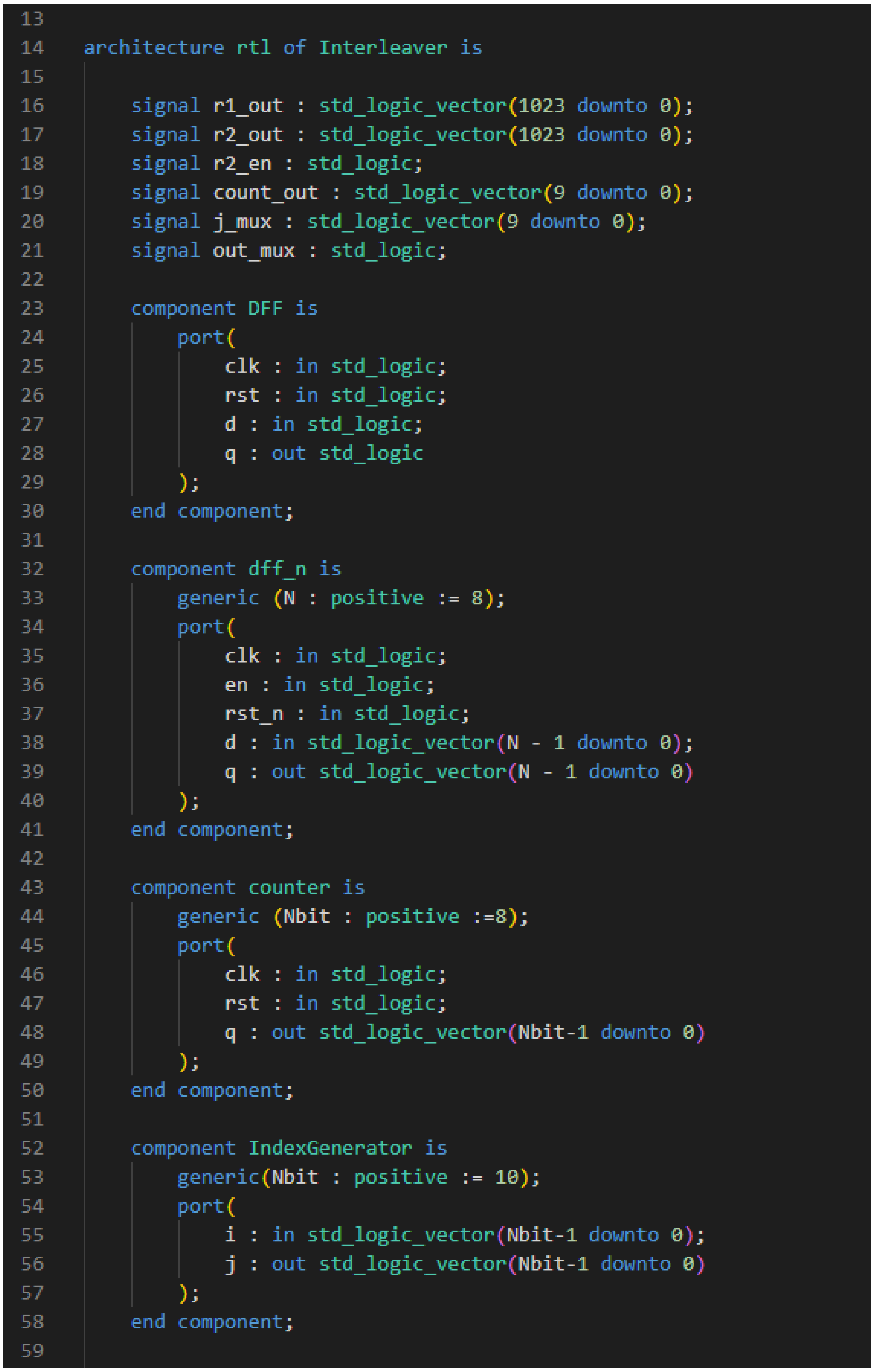


```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity Interleaver is
    port(
        clk : in std_logic;
        rst : in std_logic;
        x_in : in std_logic;
        x_out : out std_logic
    );
end Interleaver;

```

Figure 19: the Interleaver source code 1

The **Interleaver** has to have a **1 bit input and 1 bit output**, in order to **shuffle** the input bits' flow the following components are needed:

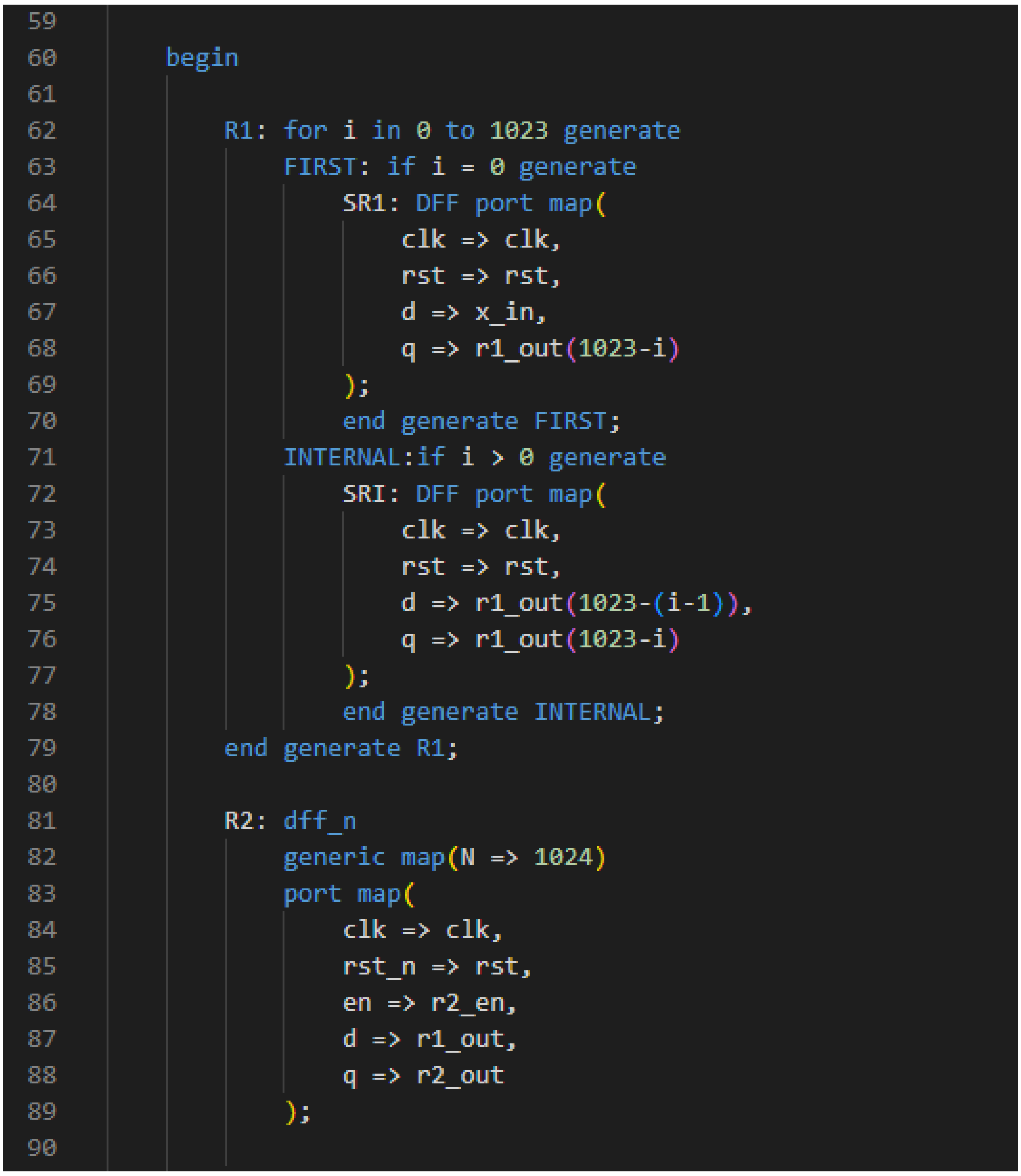
- 1 bit **1024 DFFs** (Are the R1 module) to store the entire bits' flow in the register R2.
- A 1024 bits **DFF_n** (It's the R2 register)
- The **Counter**
- A 10 inputs **AND port**
- A **1024x1 multiplexer**
- The **Index Generator**
- A 1 bit **DFF** for the output



```vhdl

architecture rtl of Interleaver is

    signal r1_out : std_logic_vector(1023 downto 0);
    signal r2_out : std_logic_vector(1023 downto 0);
    signal r2_en : std_logic;
    signal count_out : std_logic_vector(9 downto 0);
    signal j_mux : std_logic_vector(9 downto 0);
    signal out_mux : std_logic;

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

    component dff_n is
        generic (N : positive := 8);
        port(
            clk : in std_logic;
            en : in std_logic;
            rst_n : in std_logic;
            d : in std_logic_vector(N - 1 downto 0);
            q : out std_logic_vector(N - 1 downto 0)
        );
    end component;

    component counter is
        generic (Nbit : positive :=8);
        port(
            clk : in std_logic;
            rst : in std_logic;
            q : out std_logic_vector(Nbit-1 downto 0)
        );
    end component;

    component IndexGenerator is
        generic(Nbit : positive := 10);
        port(
            i : in std_logic_vector(Nbit-1 downto 0);
            j : out std_logic_vector(Nbit-1 downto 0)
        );
    end component;

```

Figure 20: the Interleaver source code 2

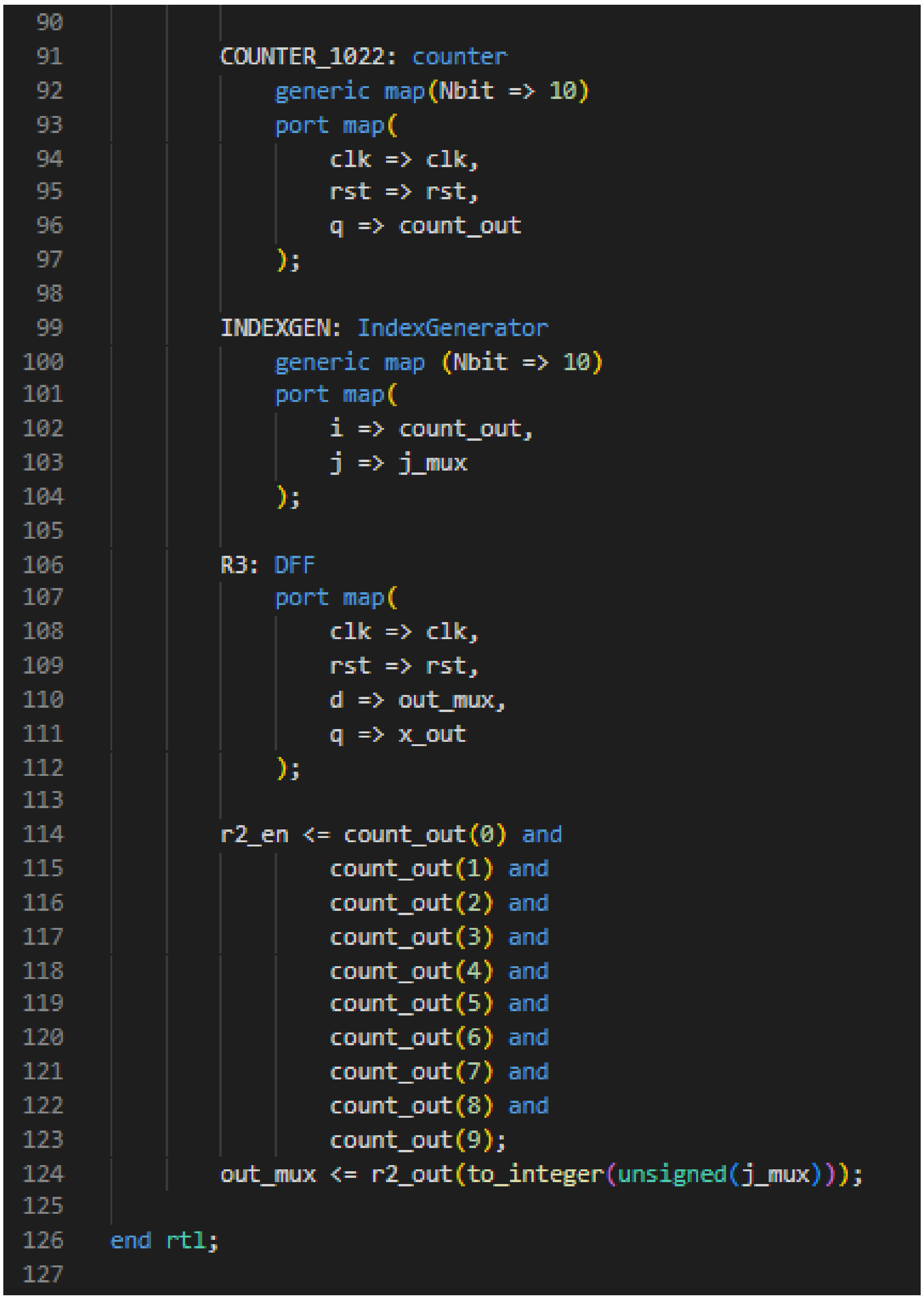
The 1024 registers of **R1** are generated assigning as input of the first of them the Interleaver's input and to all the others the output of the previous register.
Moreover the output of each register is concatenated with the other registers of R1 in order to compose the **1024 bits R2's input**, that goes to transparency only when the **enabler** is 1.



```vhdl

    begin

        R1: for i in 0 to 1023 generate
            FIRST: if i = 0 generate
                SR1: DFF port map(
                    clk => clk,
                    rst => rst,
                    d => x_in,
                    q => r1_out(1023-i)
                );
                end generate FIRST;
            INTERNAL:if i > 0 generate
                SRI: DFF port map(
                    clk => clk,
                    rst => rst,
                    d => r1_out(1023-(i-1)),
                    q => r1_out(1023-i)
                );
                end generate INTERNAL;
        end generate R1;

        R2: dff_n
            generic map(N => 1024)
            port map(
                clk => clk,
                rst_n => rst,
                en => r2_en,
                d => r1_out,
                q => r2_out
            );

```

Figure 21: the Interleaver source code 3

The **Counter**'s output is used either as the **IG**'s input (*row 102*) and the input of the **AND port** (*rows 114-123*). The **IG**'s output is used to drive the **multiplexer** (*row 126*), finally the multiplexer's output is the **R3**'s input, whose output is the same of the interleaver.



```
 90
 91         COUNTER_1022: counter
 92             generic map(Nbit => 10)
 93             port map(
 94                 clk => clk,
 95                 rst => rst,
 96                 q => count_out
 97             );
 98
 99         INDEXGEN: IndexGenerator
100             generic map (Nbit => 10)
101             port map(
102                 i => count_out,
103                 j => j_mux
104             );
105
106         R3: DFF
107             port map(
108                 clk => clk,
109                 rst => rst,
110                 d => out_mux,
111                 q => x_out
112             );
113
114         r2_en <= count_out(0) and
115                 count_out(1) and
116                 count_out(2) and
117                 count_out(3) and
118                 count_out(4) and
119                 count_out(5) and
120                 count_out(6) and
121                 count_out(7) and
122                 count_out(8) and
123                 count_out(9);
124         out_mux <= r2_out(to_integer(unsigned(j_mux)));
125
126 end rtl;
127
```

Figure 22: the Interleaver source code 4

# 4 Test Plan

In this chapter it will be shown the tests used to confirm the correct Interleaver's workflow. The main tests described are about the following components:

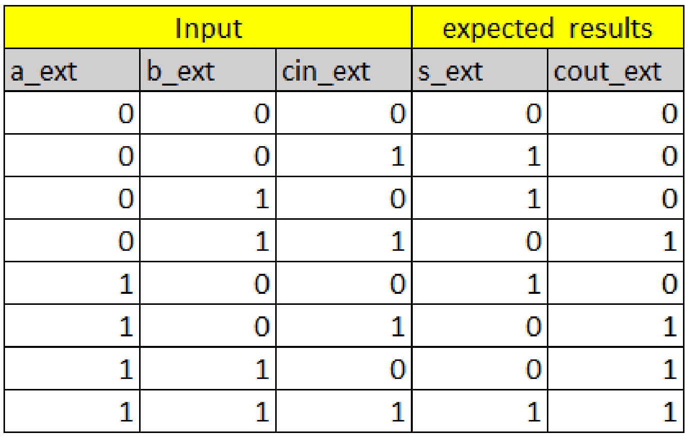
- Index Generator
- Counter
- Interleaver

The base components' tests will be seen with rapid simulations.

## 4.1 Base Components

### 4.1.1 Full Adder test

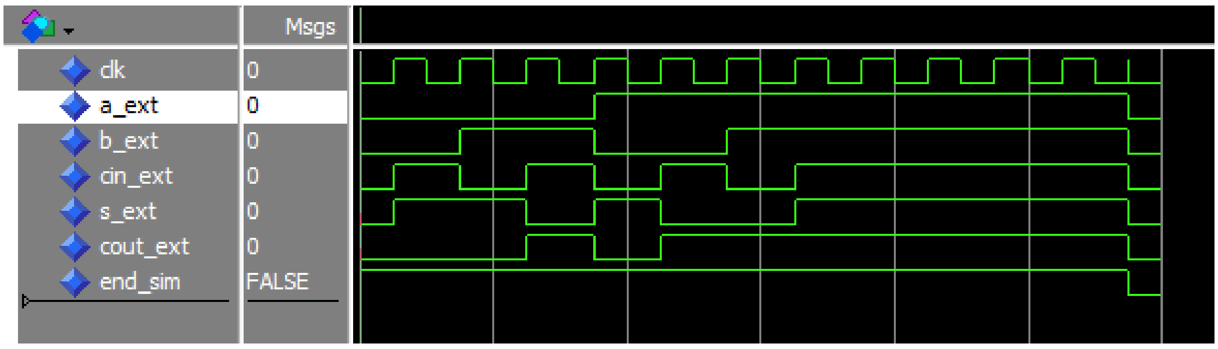
The Full Adder test consists of try some values and observe if the output is the expected result. In the table the same values used in the test are shown with their results.



| Input | | | expected results | |
|---|---|---|---|---|
| a_ext | b_ext | cin_ext | s_ext | cout_ext |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Figure 23: the Full Adder input/output table

As it can seen from the simulation the outputs are met the expected results.



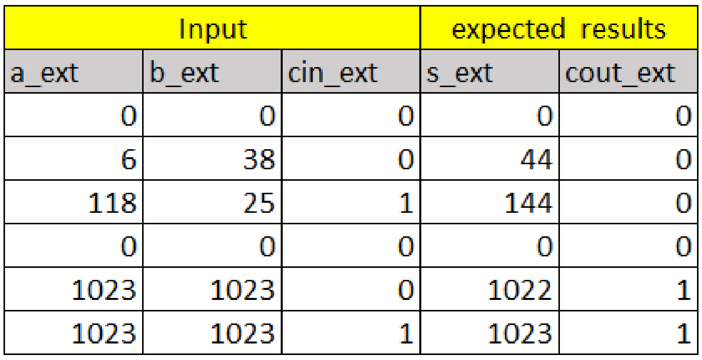
| | Msgs |
|---|---|
| clk | 0 |
| a_ext | 0 |
| b_ext | 0 |
| cin_ext | 0 |
| s_ext | 0 |
| cout_ext | 0 |
| end_sim | FALSE |

Figure 24: the Full Adder simulation

### 4.1.2 Ripple Carry Adder Test

The Ripple Carry Adder testbanch below is performed on **10 bits**, infact they are the bits needed if the component was inside the Interleaver. In order to test it are chosen values able to show the behavior either in the **normal workflow** and in case of **overflow**.

The following table shows the inputs and expected outputs.



| Input | | | expected results | |
|---|---|---|---|---|
| a_ext | b_ext | cin_ext | s_ext | cout_ext |
| 0 | 0 | 0 | 0 | 0 |
| 6 | 38 | 0 | 44 | 0 |
| 118 | 25 | 1 | 144 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 1023 | 1023 | 0 | 1022 | 1 |
| 1023 | 1023 | 1 | 1023 | 1 |

Figure 25: the Ripple Carry Adder input/output table

This is the simulation and the results are all correct.

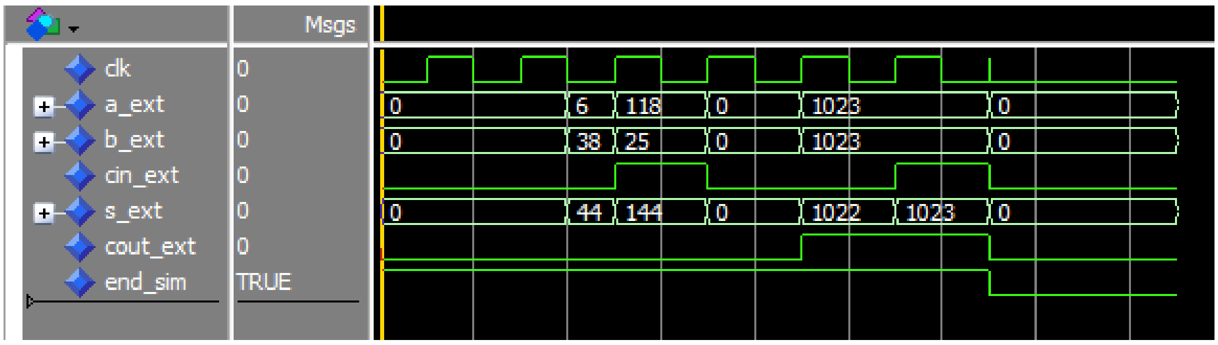


Figure 26: the Ripple Carry Adder simulation

### 4.1.3 D Flip Flop Test

The D Flip Flop test consists of try some inputs to see, for each input bit, if the component **keep it in memory** for one clock cycle, there isn't the enabler, so it is always in transparency mode.
From the simulation it can see that the output *q_tb* is equal to the input *d_tb* but with **one clock delay**, this is exactly the needed behavior for each components of the R1's 1024 registers.

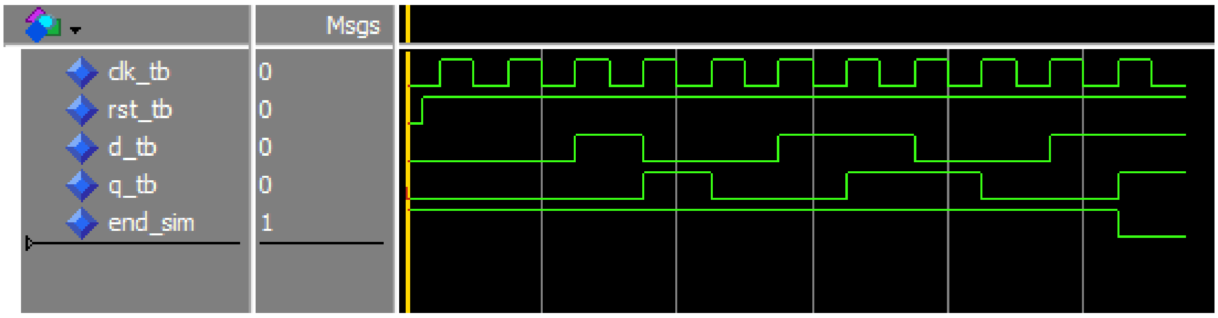


Figure 27: the D Flip Flop simulation

### 4.1.4 DFF wiht N bits Test

This test is similar to the previous but it keeps in mind the **enable presence**. For the test 10 bits are considered.

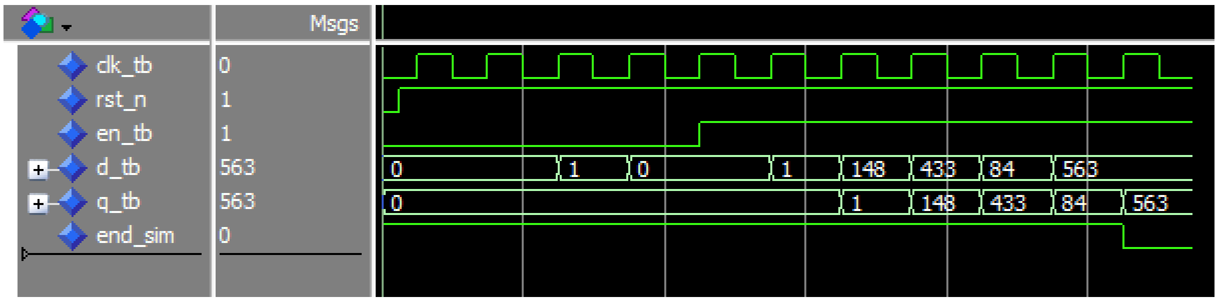
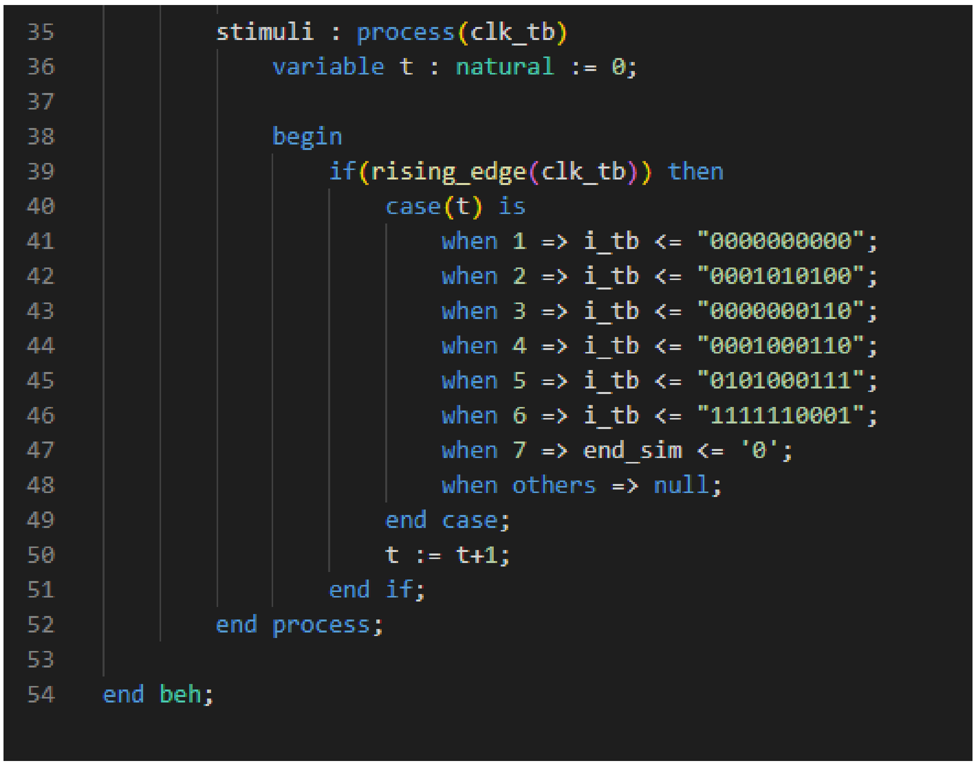


Figure 28: the DFF_n simulation

## 4.2 Index Generator Test

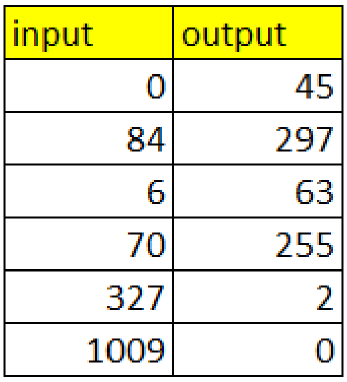
I this section the test of a fundamental Interleaver's component is shown, in particular the combinatonal network called Index Generator.



```
        stimuli : process(clk_tb)
            variable t : natural := 0;

            begin
                if(rising_edge(clk_tb)) then
                    case(t) is
                        when 1 => i_tb <= "0000000000";
                        when 2 => i_tb <= "0001010100";
                        when 3 => i_tb <= "0000000110";
                        when 4 => i_tb <= "0001000110";
                        when 5 => i_tb <= "0101000111";
                        when 6 => i_tb <= "1111110001";
                        when 7 => end_sim <= '0';
                        when others => null;
                    end case;
                    t := t+1;
                end if;
        end process;

end beh;
```

Figure 29: the Index Generator testbanch

Below there is the table with the considered inputs and relative expected outputs.



| input | output |
|---:|---:|
| 0 | 45 |
| 84 | 297 |
| 6 | 63 |
| 70 | 255 |
| 327 | 2 |
| 1009 | 0 |

Figure 30: the Index Generator input/output table

The simulation result met the table expectation.

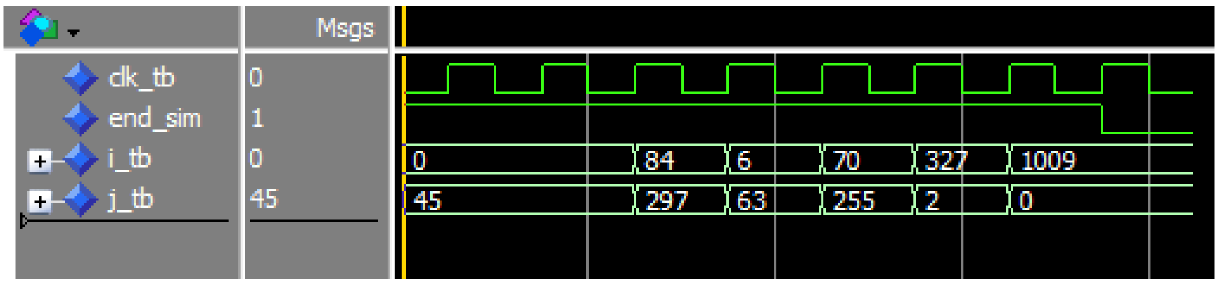
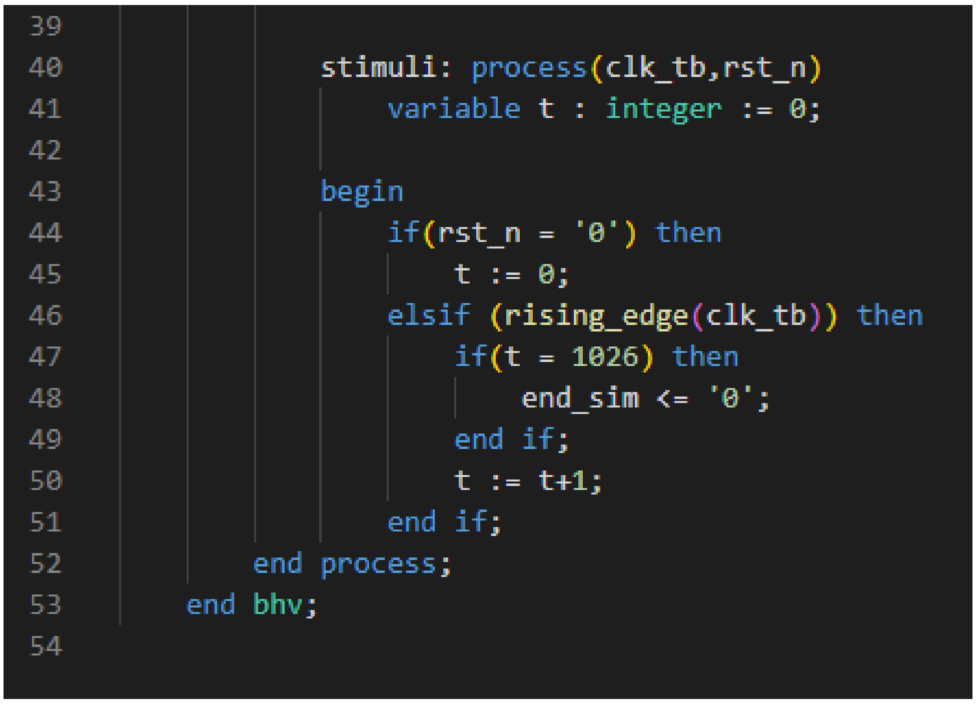


Figure 31: the Index Generator simulation

## 4.3 Counter Test

The Counter's test is done for 1026 clock cycles in order to verify the **one clock delay** at the reset and the **overflow behavior**. Indeed in the Interleaver, one transmission takes 1024 cycles, so the counter has to count from 0 to 1023 (10 bits needed) in order to identify the end and the start of a new transmission. As already mentioned the clock delay is needed to synchronize the counter with the rest of the Interleaver.



```
            stimuli: process(clk_tb,rst_n)
                variable t : integer := 0;

            begin
                if(rst_n = '0') then
                    t := 0;
                elsif (rising_edge(clk_tb)) then
                    if(t = 1026) then
                        end_sim <= '0';
                    end if;
                    t := t+1;
                end if;
        end process;
    end bhv;
```

Figure 32: the Counter testbanch

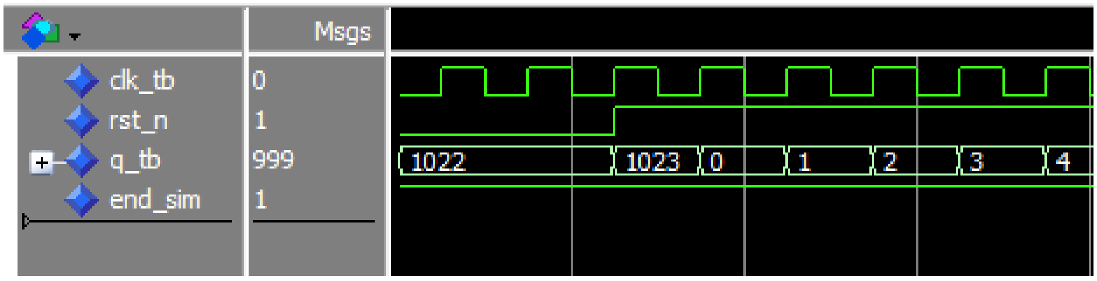


Figure 33: the Counter delay

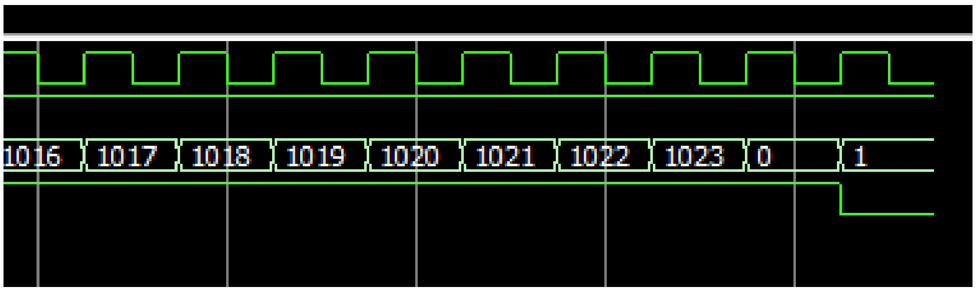


Figure 34: the Counter overflow

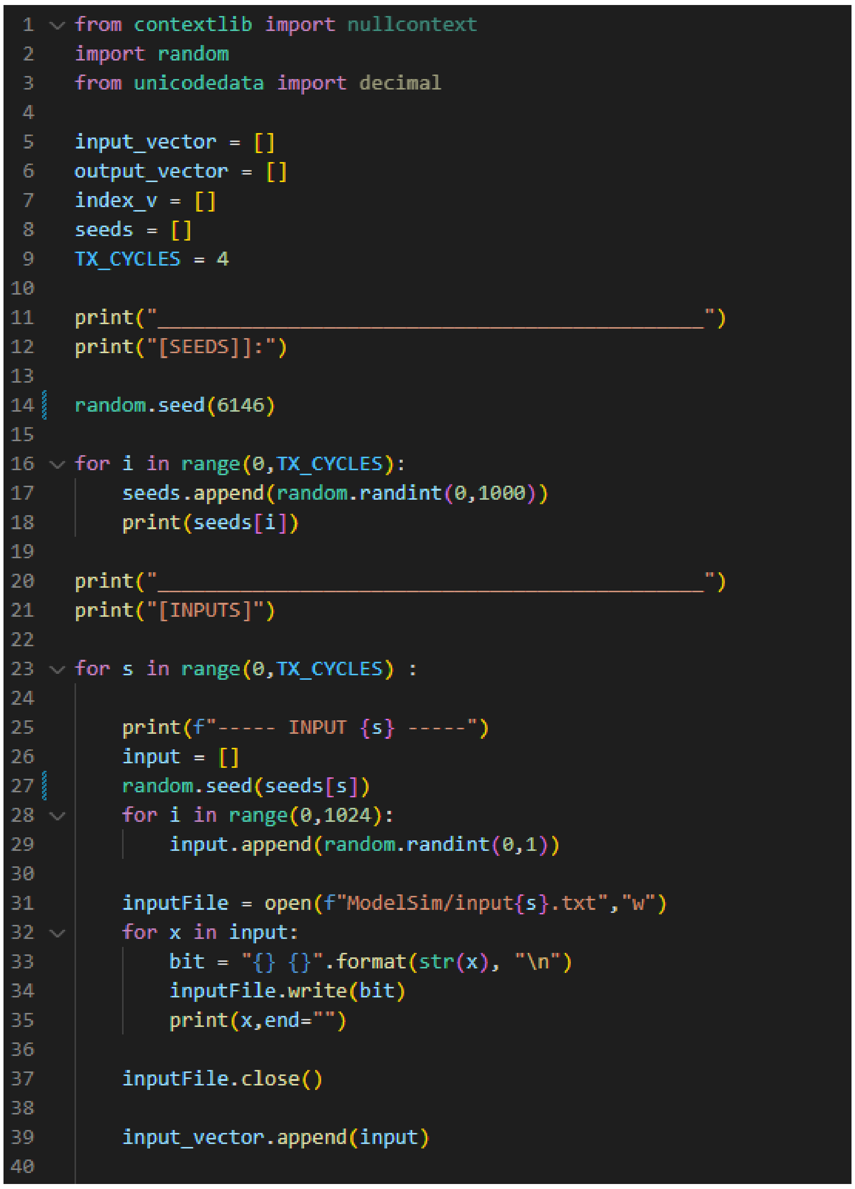
## 4.4 Interleaver Test

The Interleaver's typical workflow is a **consecutive transmissions** of 1024 bits for each, thus writing all the inputs in the testbanch and checking the outputs correctness are operations that take too long time and need too many code lines. So, in order to generate the inputs and compare the outputs, the two following **python scripts** are used:

- *interleaver_simulator.py* to **generate** the inputs and simulated outputs
- *output_validator.py* to **compare** the Interleaver's outputs with the simulated outputs

The scripts, the testbanch and the Interleaver simulation will be seen in detail.

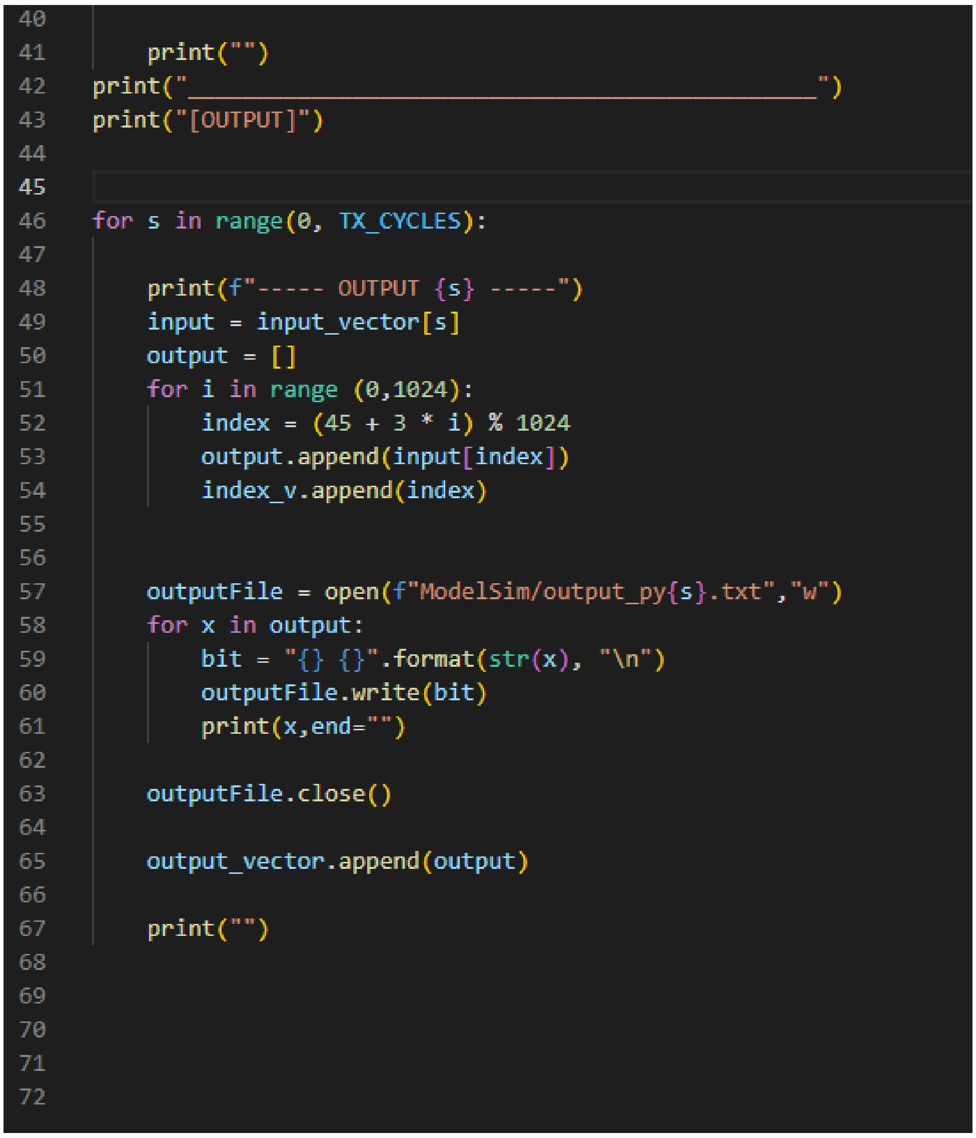
### 4.4.1 The python Interleaver simulator



```python
from contextlib import nullcontext
import random
from unicodedata import decimal

input_vector = []
output_vector = []
index_v = []
seeds = []
TX_CYCLES = 4

print("______________________________________________")
print("[SEEDS]]:")

random.seed(6146)

for i in range(0,TX_CYCLES):
    seeds.append(random.randint(0,1000))
    print(seeds[i])

print("______________________________________________")
print("[INPUTS]")

for s in range(0,TX_CYCLES) :

    print(f"----- INPUT {s} -----")
    input = []
    random.seed(seeds[s])
    for i in range(0,1024):
        input.append(random.randint(0,1))

    inputFile = open(f"ModelSim/input{s}.txt","w")
    for x in input:
        bit = "{} {}".format(str(x), "\n")
        inputFile.write(bit)
        print(x,end="")

    inputFile.close()

    input_vector.append(input)

```

Figure 35: the*interleaver_simulator.py* python code for the inputs

The portion of the python script just seen generates 4 (*TX_CYCLES*) 1024-bit streams and each stream is saved to a different file.

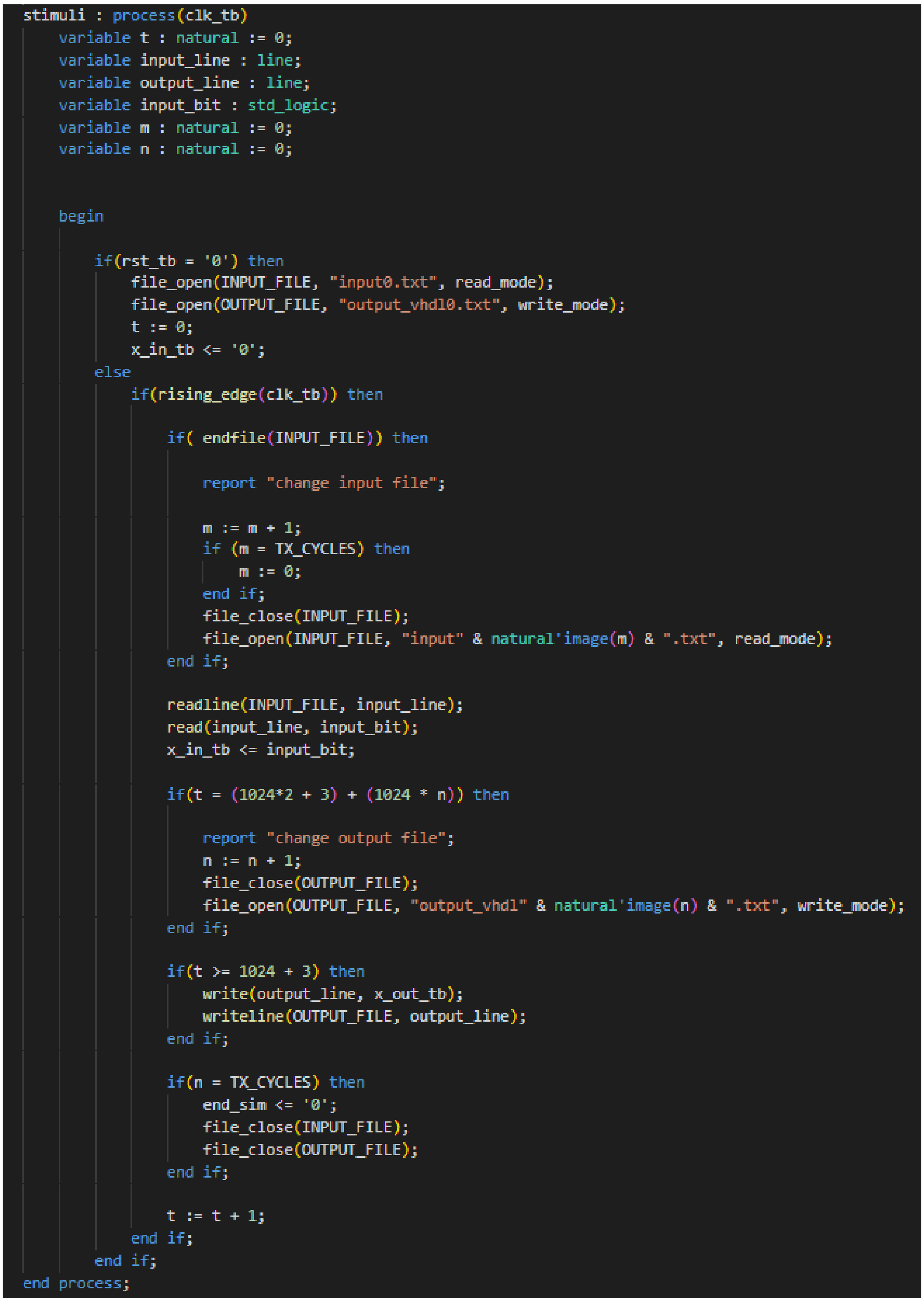


```python

    print("")
print("______________________________________________________")
print("[OUTPUT]")


for s in range(0, TX_CYCLES):

    print(f"----- OUTPUT {s} -----")
    input = input_vector[s]
    output = []
    for i in range (0,1024):
        index = (45 + 3 * i) % 1024
        output.append(input[index])
        index_v.append(index)


    outputFile = open(f"ModelSim/output_py{s}.txt","w")
    for x in output:
        bit = "{} {}".format(str(x), "\n")
        outputFile.write(bit)
        print(x,end="")

    outputFile.close()

    output_vector.append(output)

    print("")
```

Figure 36: the*interleaver_simulator.py* python code for the outputs

This last part simulates the Interleaver's workflow by generating the correct outputs, one for each input stream, these outputs are written in files that will be used to compare the results generated by the testbanch.

### 4.4.2 The Interleaver testbench



```vhdl
stimuli : process(clk_tb)
    variable t : natural := 0;
    variable input_line : line;
    variable output_line : line;
    variable input_bit : std_logic;
    variable m : natural := 0;
    variable n : natural := 0;


    begin

        if(rst_tb = '0') then
            file_open(INPUT_FILE, "input0.txt", read_mode);
            file_open(OUTPUT_FILE, "output_vhdl0.txt", write_mode);
            t := 0;
            x_in_tb <= '0';
        else
            if(rising_edge(clk_tb)) then

                if( endfile(INPUT_FILE)) then

                    report "change input file";

                    m := m + 1;
                    if (m = TX_CYCLES) then
                        m := 0;
                    end if;
                    file_close(INPUT_FILE);
                    file_open(INPUT_FILE, "input" & natural'image(m) & ".txt", read_mode);
                end if;

                readline(INPUT_FILE, input_line);
                read(input_line, input_bit);
                x_in_tb <= input_bit;

                if(t = (1024*2 + 3) + (1024 * n)) then

                    report "change output file";
                    n := n + 1;
                    file_close(OUTPUT_FILE);
                    file_open(OUTPUT_FILE, "output_vhdl" & natural'image(n) & ".txt", write_mode);
                end if;

                if(t >= 1024 + 3) then
                    write(output_line, x_out_tb);
                    writeline(OUTPUT_FILE, output_line);
                end if;

                if(n = TX_CYCLES) then
                    end_sim <= '0';
                    file_close(INPUT_FILE);
                    file_close(OUTPUT_FILE);
                end if;

                t := t + 1;
            end if;
        end if;
end process;
```

Figure 37: the Interleaver testbench source code

The testbanch takes the **inputs** from the files generated by the *interleaver_simulator.py* script, the input file is changed every time it is ended.
The **output** are written on files that are substituted every 1024 clock cycles, by starting with the cycle 1024 + 3. Indeed, as seen in the previous chapters, the 1024-bit first output stream is **not valid** because the output produced by the Interleaver is always relative to the previous input stream.
For this reason, the first valid output stream is the one that starts with the **second 1024-bit period**.
In the testbench the variable ***t*** is used to count the cycles, since is a variable and not a signal its value is assigned before all signals in the cycle, so at the first clock $t$ is 1 not 0, thus the Counter's output is **2 cycles back** than $t$. Moreover, due the **delay** of the register **R3** is needed to add 1 more cycle to write the Interleaver's output on file, so the first writing is at $t = 1024 + 3$.

### 4.4.3 The Interleaver's simulation

Here it's possible to see the **Interleaver workflow**, the first image represents the start of the first stream, the second represents the moment in witch the first output bit is valid, and the third shows the workflow at the second output stream. The **testbach signals** are only the *clk_tb*, *rst_tb*, *x_in_tb*, *x_out_tb* and *end_sim*, the others are internal to the Interleaver and they are shown to understand better the simulation.

- clk_tb : is the clock signal.
- rst_tb : is the reset signal.
- x_in_tb : is the input bit of the Interleaver.
- x_out_tb : is the R3's output and the output bit of the Interleaver.
- q : is the R2 register's output.
- i : is the Counter's output and the IG's input.
- j : is the IG's output.

In the following part of the simulation it can be see that the Counter's output $i$ is 0 at the second clock cycle, so $t$ is 2. Notice that the first three inputs bit from *x_in* are 0.

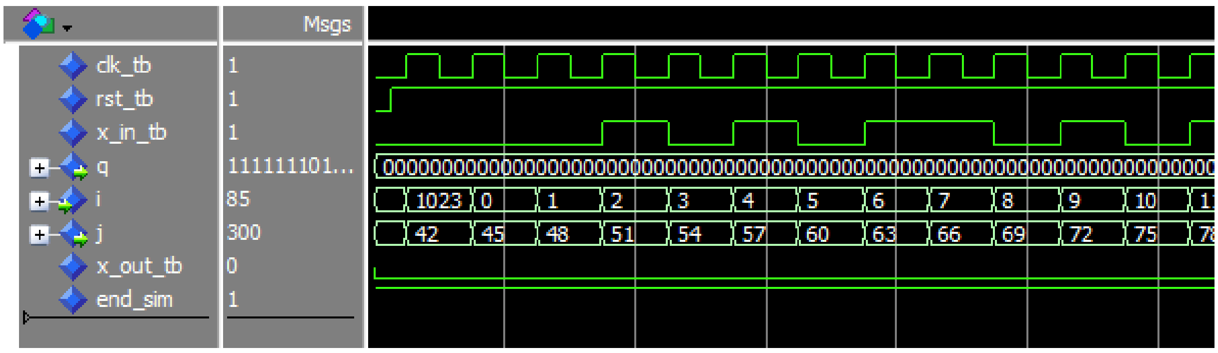


Figure 38: the Interleaver starting workflow

At the end of the fist period when $i$ is 0, $t$ is 1026, so when the output from *x_out* is ready $t$ is $1024 + 3$. Notice that the first output bit from *x_out* is 1.

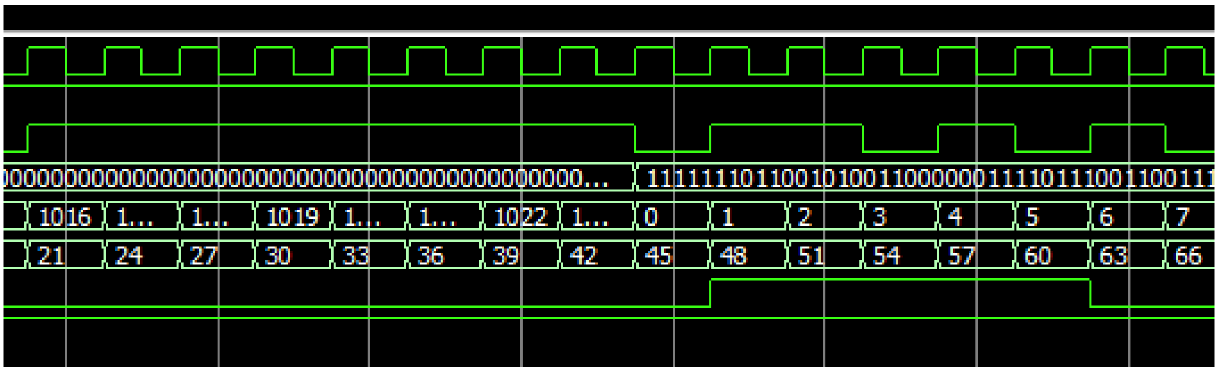


Figure 39: the Interleaver first output

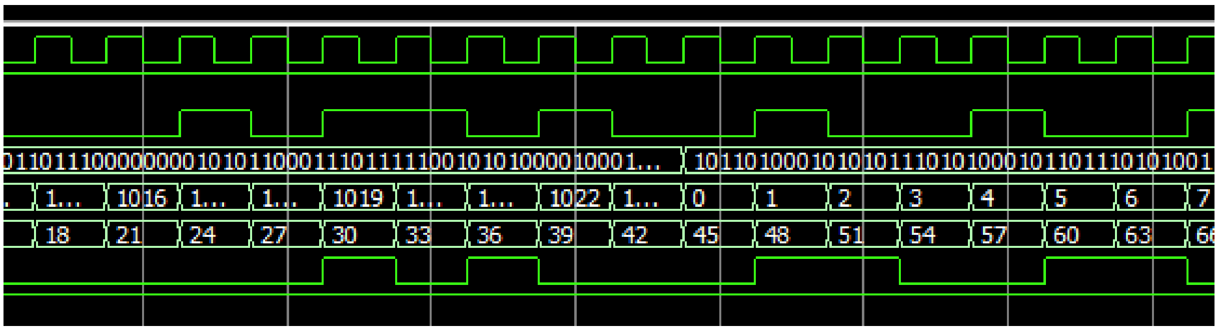
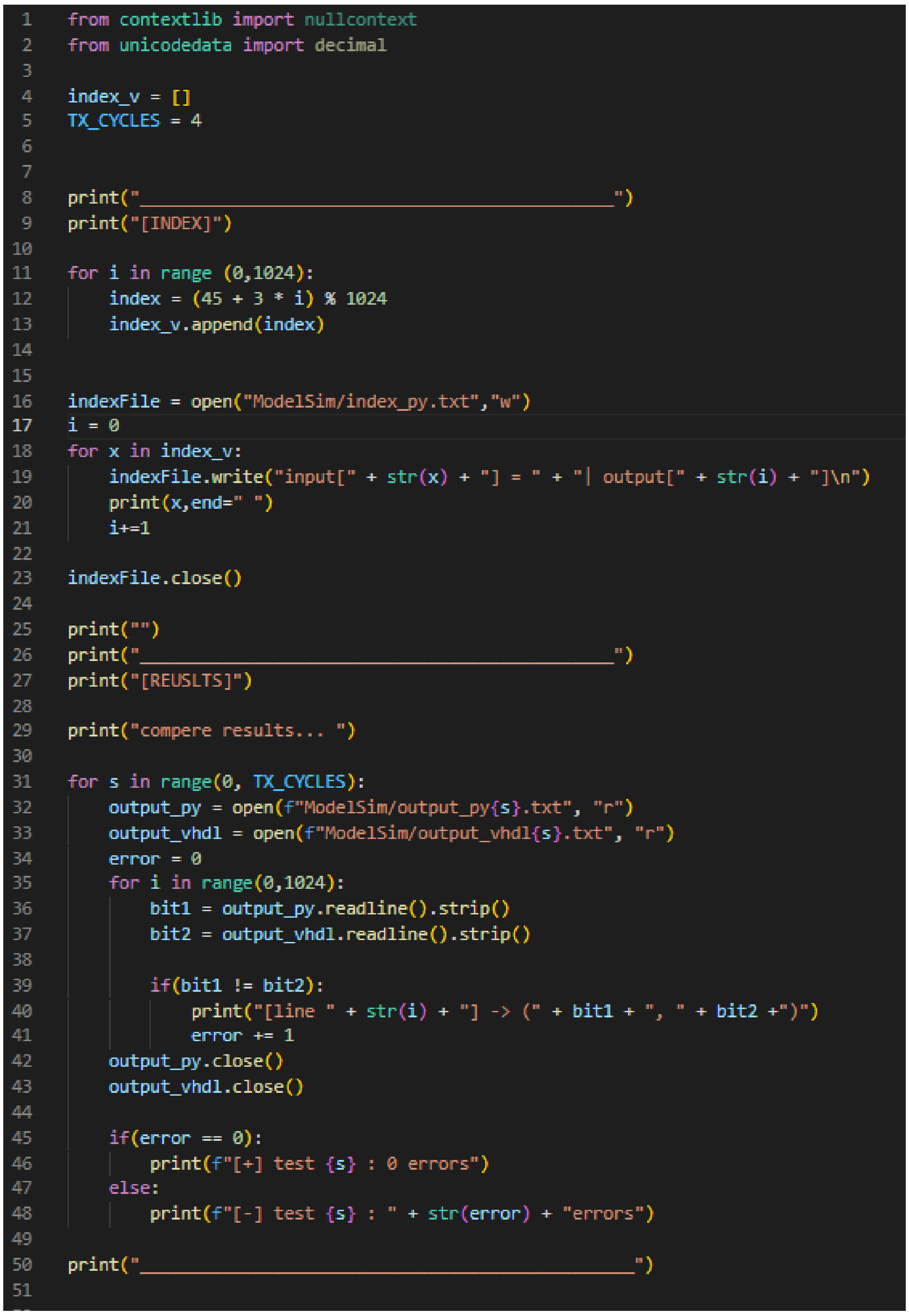


Figure 40: the Interleaver normal workflow

### 4.4.4 The python Output Validator

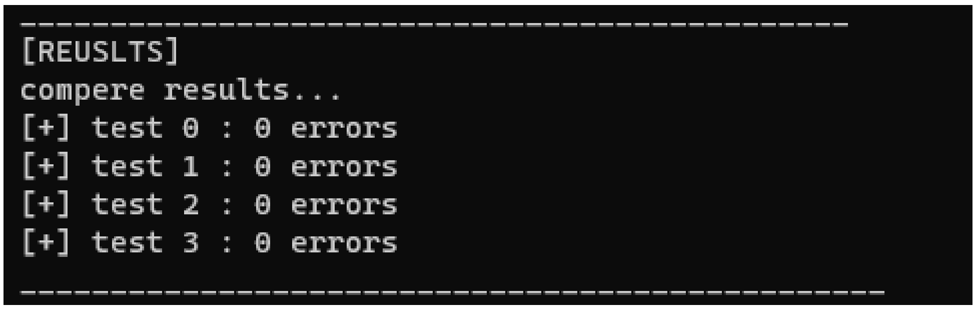
The below script is the *output_validator.py*, it's mainly used to **compare** the output files generated by the *interleaver_simulator.py* script and by the Interleaver's testbanch. Moreover it is used also to generate all the values of the Index Generator in order to inspect easily the streams in case there is an error.



```python
from contextlib import nullcontext
from unicodedata import decimal

index_v = []
TX_CYCLES = 4


print("_______________________________________________")
print("[INDEX]")

for i in range (0,1024):
    index = (45 + 3 * i) % 1024
    index_v.append(index)


indexFile = open("ModelSim/index_py.txt","w")
i = 0
for x in index_v:
    indexFile.write("input[" + str(x) + "] = " + "| output[" + str(i) + "]\n")
    print(x,end=" ")
    i+=1

indexFile.close()

print("")
print("_______________________________________________")
print("[REUSLTS]")

print("compere results... ")

for s in range(0, TX_CYCLES):
    output_py = open(f"ModelSim/output_py{s}.txt", "r")
    output_vhdl = open(f"ModelSim/output_vhdl{s}.txt", "r")
    error = 0
    for i in range(0,1024):
        bit1 = output_py.readline().strip()
        bit2 = output_vhdl.readline().strip()

        if(bit1 != bit2):
            print("[line " + str(i) + "] -> (" + bit1 + ", " + bit2 +")")
            error += 1
    output_py.close()
    output_vhdl.close()

    if(error == 0):
        print(f"[+] test {s} : 0 errors")
    else:
        print(f"[-] test {s} : " + str(error) + "errors")

print("_________________________________________________")
```

Figure 41: the *output_validator.py* python code

Following there is the result about the four bit-streams considered in all the tests.



```
-----------------------------------------------
[REUSLTS]
compere results...
[+] test 0 : 0 errors
[+] test 1 : 0 errors
[+] test 2 : 0 errors
[+] test 3 : 0 errors

-------------------------------------------------
```

Figure 42: the validator result

# 5 Vivado Report

In this chapter the **synthesis** result will be shown, the synthesis has been done through the **Xilinx Vivado** Software and has as target the FPGA device **xc7z010clg400-1** of **Zynq-700** family.

## 5.1 Elaborated Design Analysis

The following schematics show the elaborated design done by Vivado using the Interleaver source files. This design is coherent with the block diagrams seen in the architecture description chapter.
From the following schematic's portions it can see this:

- the 1 bit wire *x_in* in input to the **R1**'s registers in figure 43
- the column o**f 1024 R1's registers** in figure 44
- the **Counter** and the 10 bit **AND port** composed by nine 2 bit AND port in figure 45
- the **R2** and **R3** registers, and the **multiplexer** in figure 46
- The **Index Generator** in figure 47

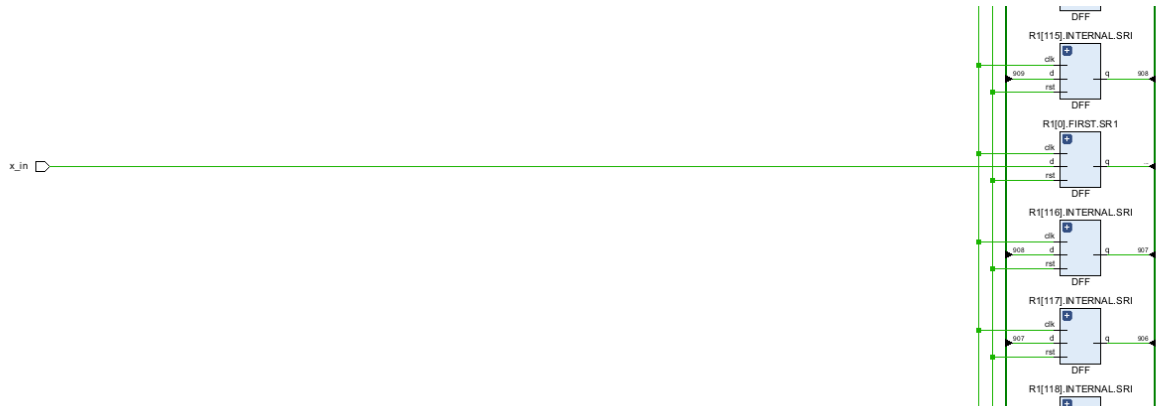


Figure 43: the schematic input

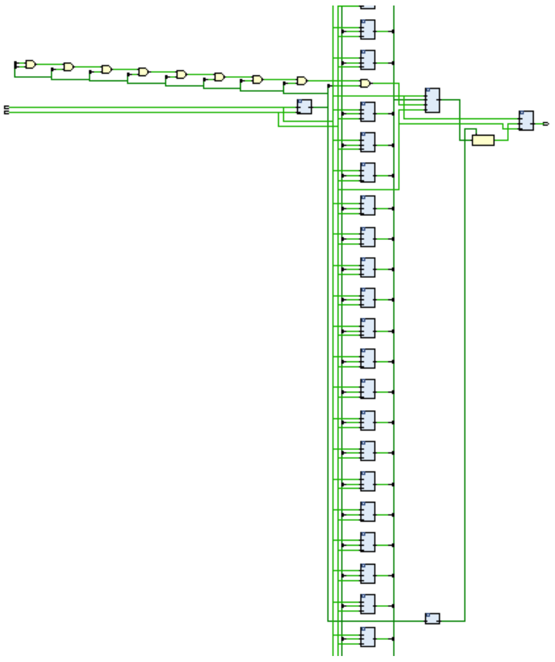


Figure 44: the schematic overview

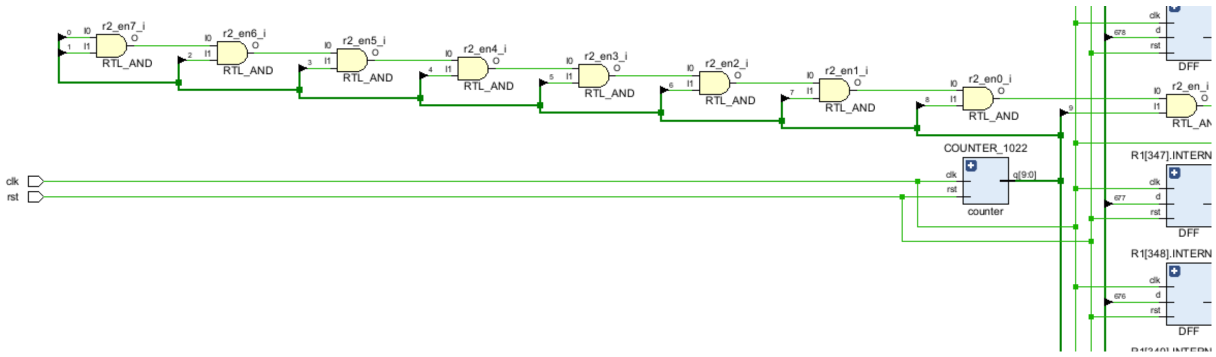


Figure 45: the Counter and the AND port

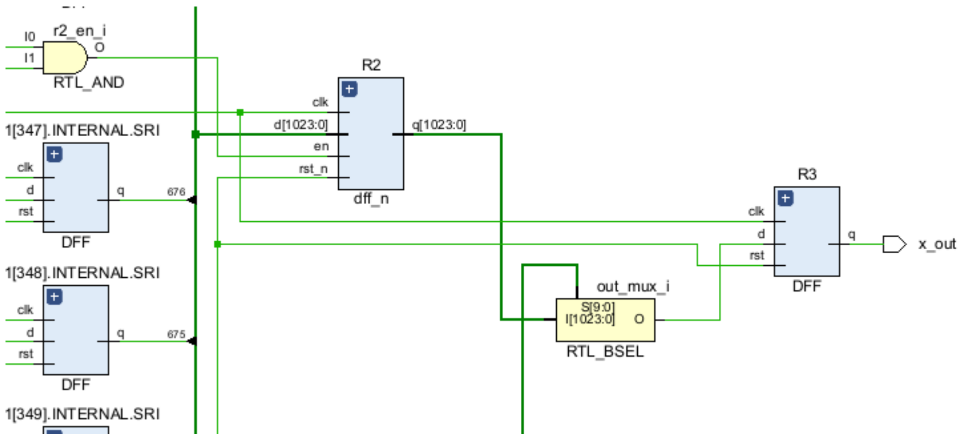


Figure 46: the R2 and R3 registers, and the multiplexer

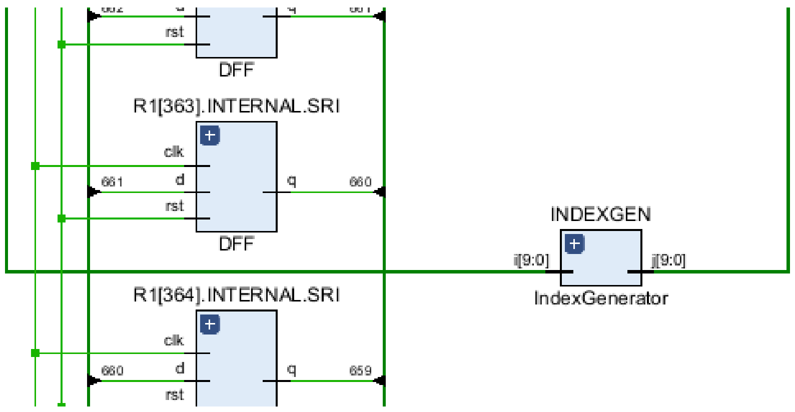
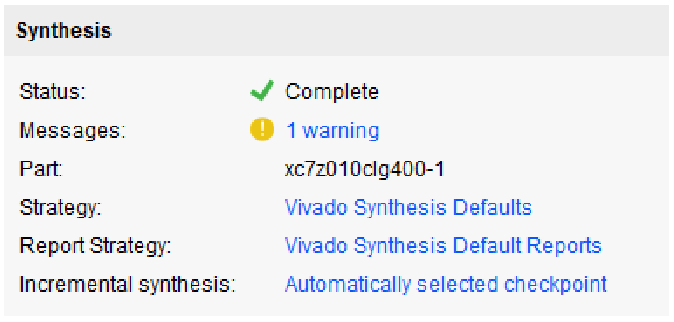


Figure 47: the Index Generator

## 5.2 Synthesis Analysis

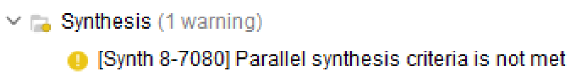
Here it can see the synthesis result using one constraint, that is a ***8ns* clock**.



**Synthesis**

| | |
|---|---|
| Status: | ✓ Complete |
| Messages: | 1 warning |
| Part: | xc7z010clg400-1 |
| Strategy: | Vivado Synthesis Defaults |
| Report Strategy: | Vivado Synthesis Default Reports |
| Incremental synthesis: | Automatically selected checkpoint |

Figure 48: the synthesis report



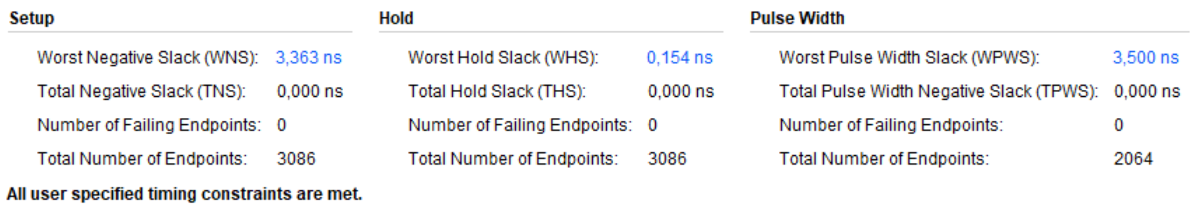
Synthesis (1 warning)
[Synth 8-7080] Parallel synthesis criteria is not met

Figure 49: the warning message

The **warning** indicates that the design is not large enough to benefit from parallel synthesis, so it's not a problem.

## 5.3 Timing Report and Critical Path

Observing the timing reports below, the clock value is enough for the project, infact all **negative slacks** are positives. The **worst negative slack** is *3,363ns*, it's meaning that it's possible to make the clock faster than this value.



| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 3,363 ns | Worst Hold Slack (WHS): | 0,154 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: | 3086 | Total Number of Endpoints: | 3086 | Total Number of Endpoints: | 2064 |

**All user specified timing constraints are met.**

Figure 50: the timing report

This is the **synthesized design** and the **critical path** is the one highlighted in blue.

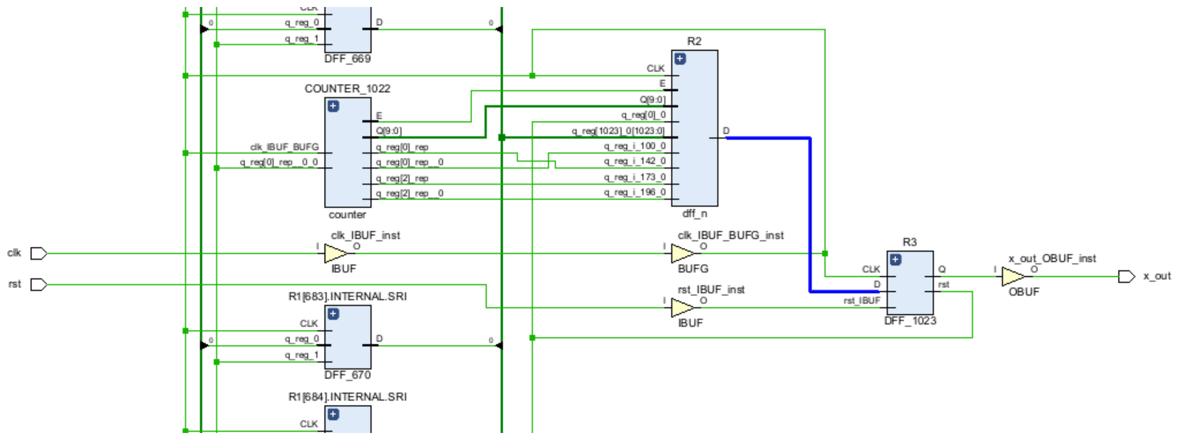
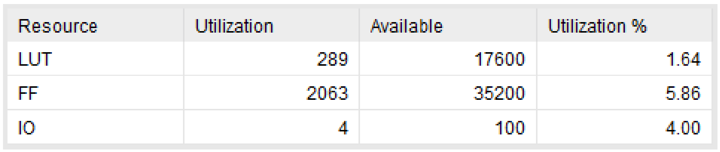


Figure 51: the synthesized design and the critical path

## 5.4 Utilization Analysis

The utilization of the resources of the Zynq device would be as follows:



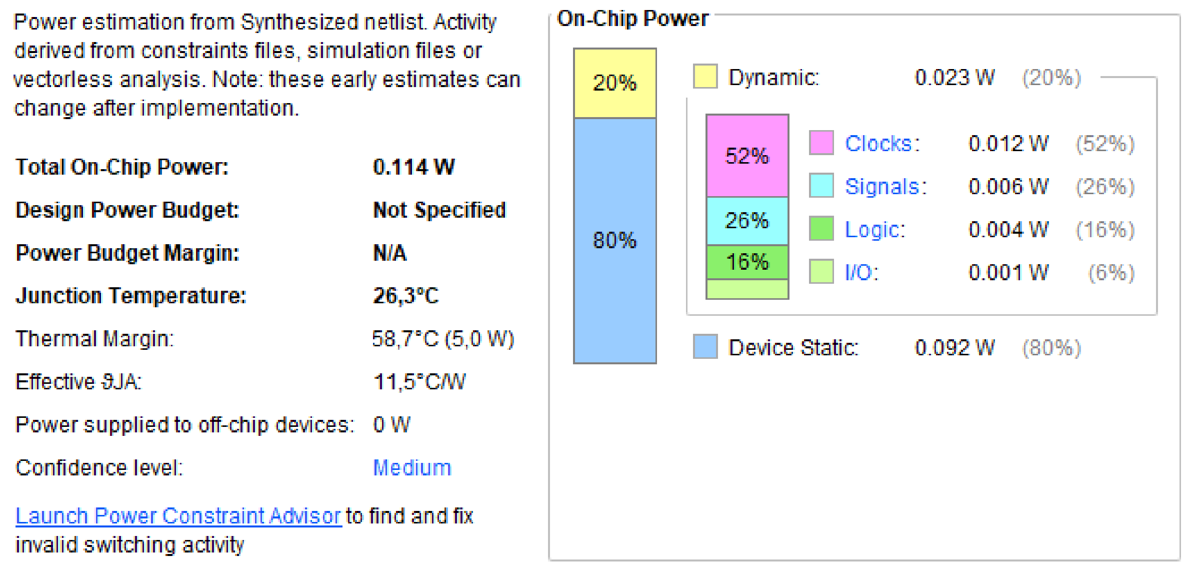
| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 289 | 17600 | 1.64 |
| FF | 2063 | 35200 | 5.86 |
| IO | 4 | 100 | 4.00 |

Figure 52: The utilization report

The **Look-Up tables** resources needed are 289, the 1.64% of the available, the needed **Flip Flop** are 2063, so is the 5.86% and the **IO ports** are 4, the 4% of the available. Thus, the device can surely implement the Interleaver.

## 5.5 Power Consumption Analysis

In this report the power consumption estimation is shown.



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:** | **0.114 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26,3°C** |
| Thermal Margin: | 58,7°C (5,0 W) |
| Effective ϑJA: | 11,5°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Medium |

Launch Power Constraint Advisor to find and fix invalid switching activity

**On-Chip Power**

| | | | |
|---|---|---|---|
| | Dynamic: | 0.023 W | (20%) |
| | Clocks: | 0.012 W | (52%) |
| | Signals: | 0.006 W | (26%) |
| | Logic: | 0.004 W | (16%) |
| | I/O: | 0.001 W | (6%) |
| | Device Static: | 0.092 W | (80%) |

Figure 53: the power consumption report

The general consumption is around *100mW* and the percentage of **dynamic power** consumption is much lower than the **static** one, so the **switching activity** is low. In addiction, dynamic power consumption is dominated by the clock switching activity.

# 6 Conclusions

The Interleaver Turbo Coding workflow is strongly relative to the **input reception rate**, infact the **clock frequency** can't be greater or lower than the bit rate. If the clock is faster then the input reception, the same bit will be taken more times, otherwise if is slower some bits will be lose.
So in order to use the Interleaver is needed to have the following cautions:

- The first bit of in input stream of 1024 bits have to be received when the Counter's output is 1022.
- The first bit of an output stream of 1024 bits is the one available when the Counter's output is 1
- The clock frequency have to be the same as the input rate reception
- The first 1024 bit output stream after the reset is not valid
- The Interleaver workflow is continuous, so it is necessary ignore the ouput bits by starting 1024 + 2 cycles after the last bit of the last input stream.
- An output stream is relative to the input stream of the previous period of 1024 cycles
- At the start of each series of input streams it is necessary reset the Interleaver to synchronize the Counter