
UNIVERSITÀ DI PISA

Computer Engineering
Electronic and Communication Systems

# ERROR CORRECTION CODE

## Project Report

Federica Perrone

# Summary

## Introduction

An error correcting code is an algorithm for expressing a sequence of numbers such that any errors, which are introduced, can be detected and corrected (within certain limitations) based on the remaining numbers.

The error correcting codes are used for controlling errors in data over unreliable or noisy communication channels.

The central idea is the sender encodes the message with redundant information in the form of an ECC. The redundancy allows the receiver to detect a limited number of errors that may occur anywhere in the message, and often to correct these errors without retransmission.

The two main categories of ECC codes are block codes and convolutional codes.

Hamming codes are a family of linear error-correcting block codes. Richard Hamming invented Hamming codes in 1950 as a way of automatically correcting errors introduced by punched card readers. The scheme invented by Hamming adds additional parity bits (k) to the information bits (n), and can self-detect and self-correct any single-event effect (SEE) error that occurs during transmission. Once the location of the error is identified and located, the code inverts the bit, returning it to its original form.

Hamming codes form the foundation of other more complex error correction schemes. The original scheme allows **single-error correction single-error detection** (**SECSED**), but with an addition of one parity bit, an extended Hamming version allows **single-error correction and double-error detection** (**SECDED**).

The extended Hamming code is popular in computer memory systems.

### Overview of Hamming Code

Hamming codes tend to follow the process illustrated in Figure 1. The input is errorless information of n-bits long which is sent to the encoder. The encoder then applies Hamming theorems, calculates the parity bits (k), and attaches them to the received information data, to form a codeword of (n + k)-bits. The processed information which contains additional parity bits is now ready for storage or transmission.

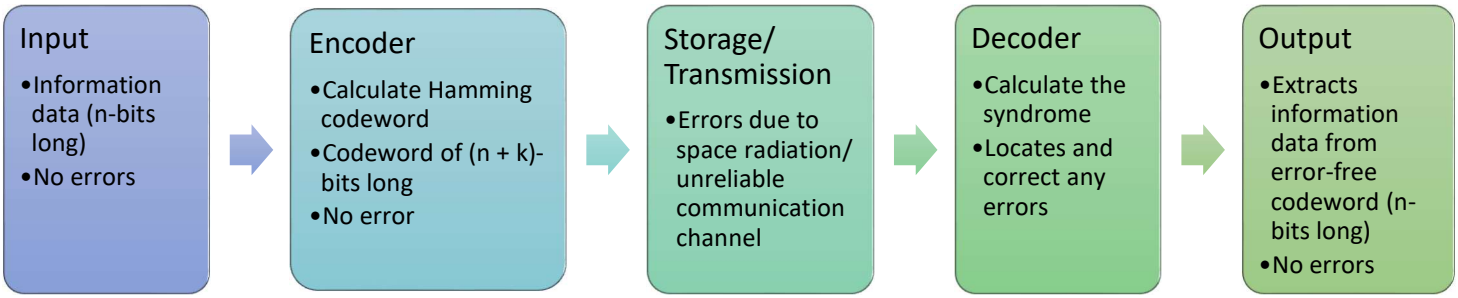


*Figure 1- General layout of Hamming code*

The decoder is responsible for checking and correcting any errors contained within the requested data. This is done by applying the Hamming theorem to calculate the syndrome. The decoder checks, locates, and corrects the errors contained in the codeword before extracting the new error-free information data.

### General Algorithm

As mentioned previously, Hamming code uses parity bits to perform error detection and correction. Bit position (codeword) is dependent on the amount of data bits protected. Parity bit positions are placed according to, 2 to the power of parity bit:

$$2^0 = 1, \quad 2^1 = 2, \quad 2^2 = 4, \quad 2^3 = 8, \quad 2^4 = 16, \quad \dots$$

A general algorithm can be deduced from the following description:

1. Number the bits starting from 1: bit 1, 2, 3, 4, 5, 6, 7, etc.
2. Write the bit numbers in binary: 1, 10, 11, 100, 101, 110, 111, etc.
3. All bit positions that are powers of two (have a single 1 bit in the binary form of their position) are parity bits: 1, 2, 4, 8, etc. (1, 10, 100, 1000)
4. All other bit positions, with two or more 1 bits in the binary form of their position, are data bits.
5. Each data bit is included in a unique set of 2 or more parity bits, as determined by the binary form of its bit position:
   - Parity bit 1 ($P_1$) covers all bit positions which have the least significant bit set: bit 1 (the parity bit itself $P_1$), 3, 5, 7, 9, etc. (all the odd numbers);
   - Parity bit 2 ($P_2$) covers all bit positions which have the second least significant bit set: bits 2 ($P_2$), 3, 6, 7, 10, 11, etc. (sets of 2);
   - Parity bit 4 ($P_4$) covers all bit positions which have the third least significant bit set: bits 4 ($P_4$), 5, 6, 7, 12, 13, 14, 15, etc. (sets of 4);
   - Parity bit 8 ($P_8$) covers all bit positions which have the fourth least significant bit set: bits 8 ($P_8$), 9, 10, 11, 12, 13, 14, 15, etc. (sets of 8);
   - Parity bit 16 ($P_{16}$) covers all bit positions which have the fifth least significant bit set: bits 16 ($P_{16}$), 17, 18, 19, 20, 21, 22, 23, etc. (sets of 16);

In the following table (Table 1), a typical codeword layout is shown.

| **Bit position** | | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Encoded data bits** | | **$P_1$** | **$P_2$** | **$D_1$** | **$P_4$** | **$D_2$** | **$D_3$** | **$D_4$** | **$P_8$** | **$D_5$** | **$D_6$** | **$D_7$** | **$D_8$** | **$D_9$** | **$D_{10}$** | **$D_{11}$** | **$P_{16}$** |
| **Encoded data coverage** | **$P_1$** | | | ✓ | | ✓ | | ✓ | | ✓ | | ✓ | | ✓ | | ✓ | |
| | **$P_2$** | | | ✓ | | | ✓ | ✓ | | | ✓ | ✓ | | | ✓ | ✓ | |
| | **$P_4$** | | | | | ✓ | ✓ | ✓ | | | | | | ✓ | ✓ | ✓ | |
| | **$P_8$** | | | | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| | **$P_{16}$** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

*Table 1- bit layout of Hamming code*

The layout makes each column have a unique parity bit combination, for each bit position. This unique parity bit combination in known as the syndrome value. The syndrome allows errors to be located and corrected.

## Hamming Encoder

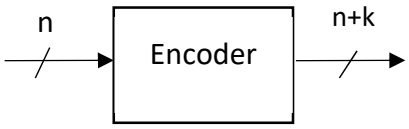


*Figure 2 - Encoder block diagram*

The Hamming encoder is responsible for generating the codeword (n + k - bits long) from the information bits. Once generated the codeword contains both the information bits and parity bits. The codeword is calculated as shown in the following figure (Figure 3).

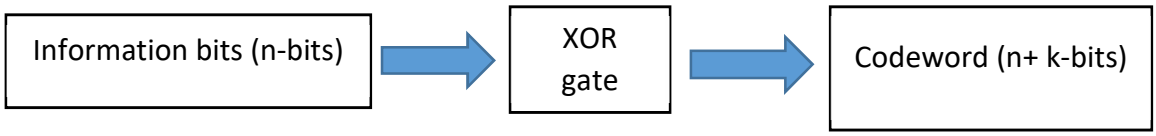


*Figure 3 - Graphical expression of Hamming encoder*

## Hamming Decoder

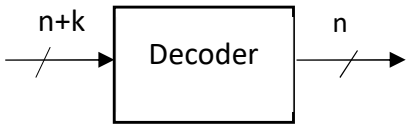


*Figure 4 - Decoder block diagram*

The Hamming decoder is responsible for generating the syndrome (k-bits long) from the codeword (n +k -bits long). Once generated, the syndrome contains the error pattern that allows the error to be located and corrected. The syndrome is calculated as shown in the following figure (Figure 5).

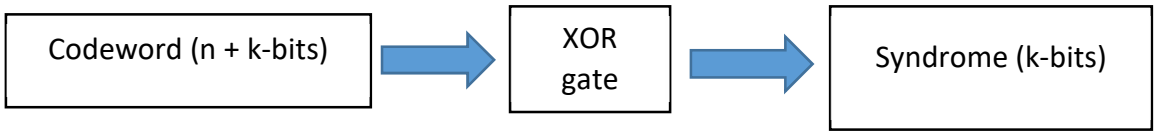


*Figure 5 - Graphical expression of Hamming decoder*

## Extended Hamming Code

The extended Hamming code makes use of an additional parity bit, which increases the Hamming code capabilities to SECDED.

In Table 1, $P_{16}$ is the added parity bit that allows double error detection.

## Applications

The Error Correcting Codes (ECC), and more specifically Hamming codes, are used in many applications, such us:

- Telecommunication industry;
- Computer memory, modems and embedded processors;
- Nano Satellites.

## Architecture description

In this chapter will be discussed deeply the architecture of the Error Correcting Code (ECC).

The general structure could be summarized by the following schema (Figure 6):

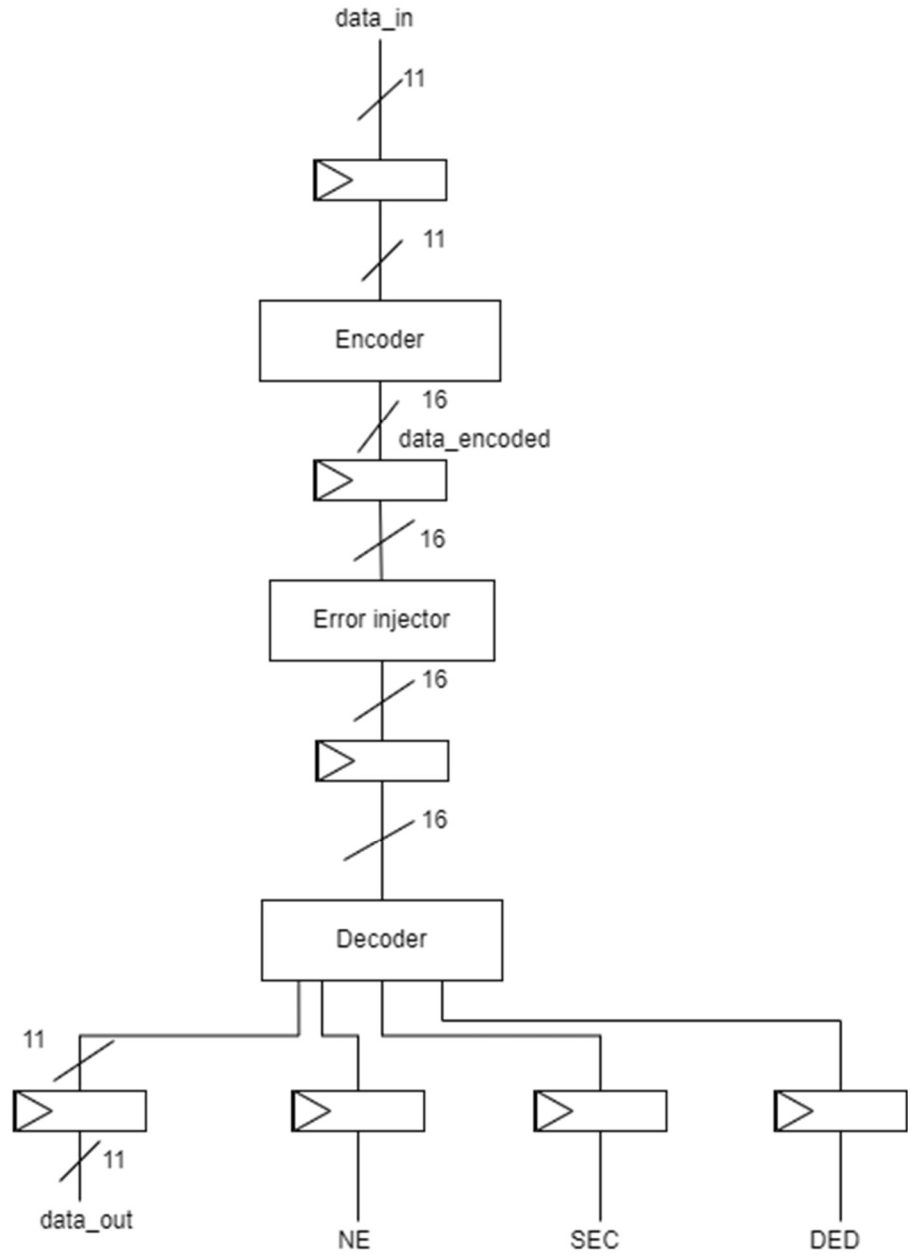


*Figure 6 - General schema*

## Encoder Architecture

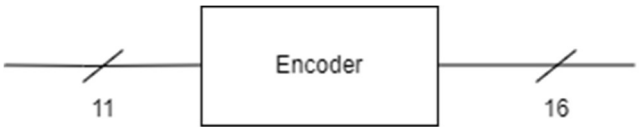


*Figure 7 - Encoder block diagram*

As shown in Figure 7, the data word (11 bits long) is applied as an input to the encoder circuit, which performs XOR operations on the given data word and thus the required parity bits (5 bits long) are generated. In this way, the output bits of the encoder consist 16 bits, i.e. 11-bits of data (from $D_1$ to $D_{11}$) and 5-bits of parity ($P_1$, $P_2$, $P_4$, $P_8$, $P_{16}$).

The 5 parity bits, for 11 data bits, are calculated as follows:

$$P_1 = D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7 \oplus D_9 \oplus D_{11}$$

$$P_2 = D_1 \oplus D_3 \oplus D_4 \oplus D_6 \oplus D_7 \oplus D_{10} \oplus D_{11}$$

$$P_4 = D_2 \oplus D_3 \oplus D_4 \oplus D_8 \oplus D_9 \oplus D_{10} \oplus D_{11}$$

$$P_8 = D_5 \oplus D_6 \oplus D_7 \oplus D_8 \oplus D_9 \oplus D_{10} \oplus D_{11}$$

$$P_{16} = P_1 \oplus P_2 \oplus D_1 \oplus P_4 \oplus D_2 \oplus D_3 \oplus D_4 \oplus P_8 \oplus D_5 \oplus D_6 \oplus D_7 \oplus D_8 \oplus D_9 \oplus D_{10} \oplus D_{11}$$

The following figure (Figure 8) shows the encoder circuit:

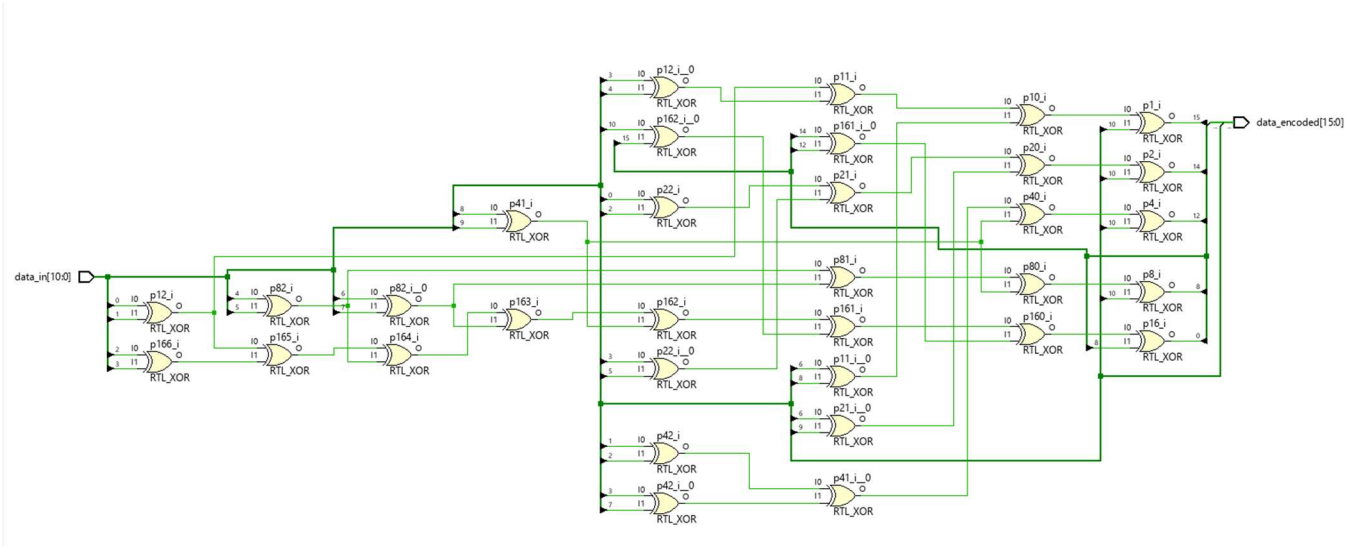


*Figure 8 - Encoder circuit*

## Decoder Architecture

The decoder must be able to detect a double error and correct a single error (SECDED →single error correction, double error detection).

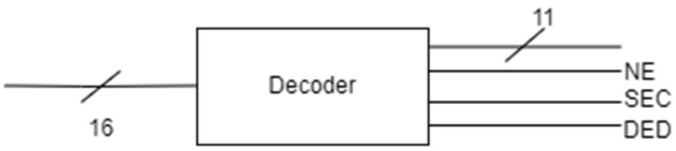


*Figure 9 - Decoder block diagram*

As shown in Figure 9, the decoder has the code word as input (16-bits long) and outputs the original word (possibly correct) (11-bits long), a NE bit indicating that there was no error, a SEC bit indicating single correct error and a DED bit indicating double error detection.

In order to correct a single error and detect a double error the decoder must calculate control bits as follows:

$$C_1 = P_1 \oplus D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7 \oplus D_9 \oplus D_{11}$$

$$C_2 = P_2 \oplus D_1 \oplus D_3 \oplus D_4 \oplus D_6 \oplus D_7 \oplus D_{10} \oplus D_{11}$$

$$C_4 = P_4 \oplus D_2 \oplus D_3 \oplus D_4 \oplus D_8 \oplus D_9 \oplus D_{10} \oplus D_{11}$$

$$C_8 = P_8 \oplus D_5 \oplus D_6 \oplus D_7 \oplus D_8 \oplus D_9 \oplus D_{10} \oplus D_{11}$$

$$P = P_1 \oplus P_2 \oplus D_1 \oplus P_4 \oplus D_2 \oplus D_3 \oplus D_4 \oplus P_8 \oplus D_5 \oplus D_6 \oplus D_7 \oplus D_8 \oplus D_9 \oplus D_{10} \oplus D_{11} \oplus P_{16}$$

$$C = C_8\ C_4\ C_2\ C_1$$

In general, the can be the following four cases:

1. C=0 and P=0 → No error occurred, so the code word is taken as valid information;
2. C≠0 and P=1 → A single bit error occurred that can be detected and corrected;
3. C≠0 and P=0 → Double bit error occurred that can be detected, but cannot be corrected, so the code word is taken as invalid information;
4. C=0 and P=1 → A single bit error occurred in the bit $P_{16}$, that can be detected and corrected.

In the first case, since there was not no error, the output bit NE is set to 1.

In the second case, it is necessary to find the position of the wrong bit so that it can be corrected. Looking at C as a 4-bits word, its decimal decoding gives us exactly the position of the wrong bit. In this way, it is possible to flip the wrong bit and to output the corrected word. Furthermore, the output bit SEC is set to 1.

In the third case, since there was double bit error, the output bit DED is set to 1.

In the fourth case, the correction of the wrong bit is easy, because we know already the wrong bit, that is $P_{16}$. As in the second case, also in this case the output bit SEC is set to 1.

The following figure (Figure 10) shows the decoder circuit:

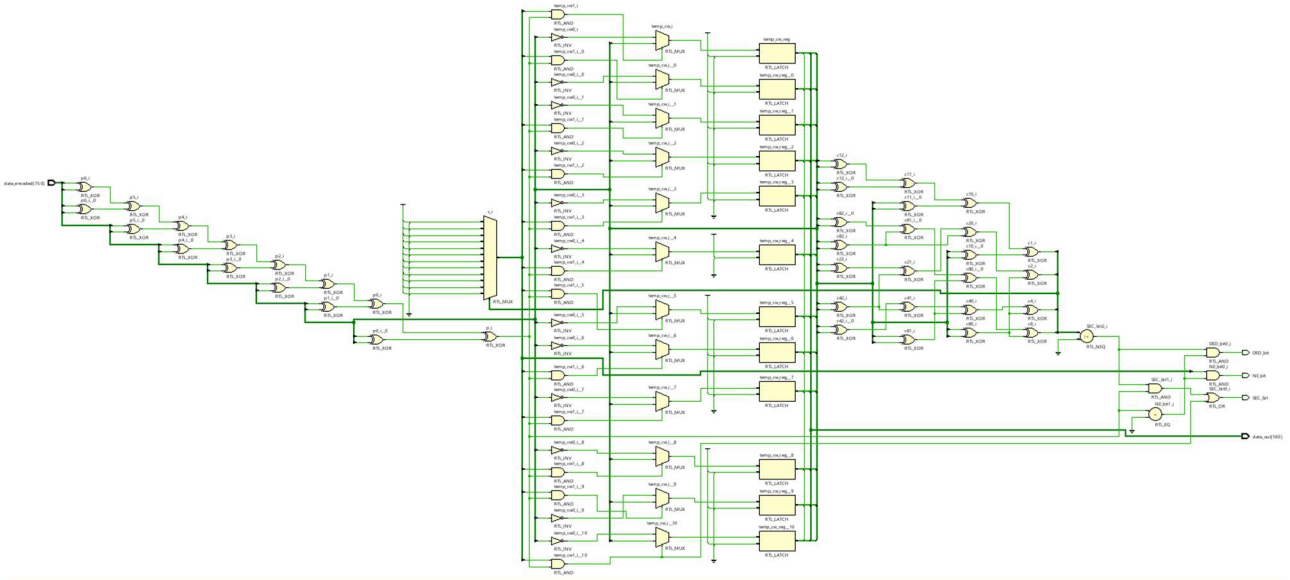

*Figure 10 - Decoder circuit*

## Error Injector

The Error injector is implemented by flipping the bits of the code word on the flow, in the testbench. In this way, the behavior of an unreliable transmission channel or a storage is simulated.

# VHDL Code

In this chapter will be presented the main modules that compose the architecture of the **Error Correcting Code** (**ECC**).

## Modules List

The following modules were created:

- ECC
  - Encoder
  - Decoder
  - DFF_N

A **bottom-up strategy** was followed in order to build up the architecture.

## ECC

The main hardware description of the architecture. This module will connect all the other modules in order to create the correct architecture.

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

entity ECC is
    port(
        clk : in std_logic;
        resetn : in std_logic;
        data_in : in std_logic_vector(10 downto 0);
        data_out : out std_logic_vector(10 downto 0);
        NE_bit : out std_logic;
        SEC_bit : out std_logic;
        DED_bit : out std_logic
    );

end ECC;

architecture beh of ECC is
    signal output_inputreg : std_logic_vector(10 downto 0);
    signal output_encoder : std_logic_vector(15 downto 0);
    signal output_encoder_reg : std_logic_vector(15 downto 0);
    signal output_decoder_data : std_logic_vector(10 downto 0);
    signal output_decoder_NE : std_logic_vector(0 downto 0);
    signal output_decoder_SEC : std_logic_vector(0 downto 0);
    signal output_decoder_DED : std_logic_vector(0 downto 0);
    signal output_decoderreg_data : std_logic_vector(10 downto 0);
    signal output_decoderreg_NE : std_logic_vector(0 downto 0);
    signal output_decoderreg_SEC : std_logic_vector(0 downto 0);
    signal output_decoderreg_DED : std_logic_vector(0 downto 0);

    component Encoder is
        port(
            data_in : in std_logic_vector(10 downto 0);
            data_encoded : out std_logic_vector(15 downto 0)
        );
    end component Encoder;

    component Decoder is
        port(
            data_encoded : in std_logic_vector(15 downto 0);
            data_out : out std_logic_vector(10 downto 0);
```

```
            NE_bit : out std_logic;
            SEC_bit : out std_logic;
            DED_bit : out std_logic
        );
    end component Decoder;

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

    begin

        --register for data_in bit
        input_reg: DFF_N
        generic map( N => 11)
        port map(
            clk     => clk,
            a_rstn =>  resetn,
            en      => '1',
            d       => data_in,
            q       => output_inputreg
        );

        --encoder
        encoder_block: Encoder
        port map(
            data_in => output_inputreg,
            data_encoded => output_encoder
        );

        --register for output of the encoder
        out_encoder_reg: DFF_N
        generic map( N => 16)
        port map(
            clk     => clk,
            a_rstn =>  resetn,
            en      => '1',
            d       => output_encoder,
            q       => output_encoder_reg
        );

        --decoder
        decoder_block: Decoder
        port map(
            data_encoded => output_encoder_reg,
            data_out => output_decoder_data,
            NE_bit => output_decoder_NE(0),
            SEC_bit => output_decoder_SEC(0),
            DED_bit => output_decoder_DED(0)
        );

        --register for output of the decoder (data)
        out_decoder_reg_data: DFF_N
        generic map( N => 11)
        port map(
```

```
            clk     => clk,
            a_rstn =>  resetn,
            en      => '1',
            d       => output_decoder_data,
            q       => output_decoderreg_data
        );

        data_out <= output_decoderreg_data;

        --register for output of the decoder (NE)
        out_decoder_reg_NE: DFF_N
        generic map( N => 1)
        port map(
            clk     => clk,
            a_rstn =>  resetn,
            en      => '1',
            d       => output_decoder_NE,
            q       => output_decoderreg_NE
        );

        NE_bit <= output_decoderreg_NE(0);

        --register for output of the decoder (SEC)
        out_decoder_reg_SEC: DFF_N
        generic map( N => 1)
        port map(
            clk     => clk,
            a_rstn =>  resetn,
            en      => '1',
            d       => output_decoder_SEC,
            q       => output_decoderreg_SEC
        );

        SEC_bit <= output_decoderreg_SEC(0);

        --register for output of the decoder (DED)
        out_decoder_reg_DED: DFF_N
        generic map( N => 1)
        port map(
            clk     => clk,
            a_rstn =>  resetn,
            en      => '1',
            d       => output_decoder_DED,
            q       => output_decoderreg_DED
        );

        DED_bit <= output_decoderreg_DED(0);

    end beh;
```

## Encoder

This module has the task of creating the code word (16-bits long), having the 11-bits as input.

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

entity Encoder is
    port(
        data_in : in std_logic_vector(10 downto 0);
        data_encoded : out std_logic_vector(15 downto 0)
```

```
      );
end Encoder;

architecture beh of Encoder is
    signal p1 : std_logic;
    signal p2 : std_logic;
    signal p4 : std_logic;
    signal p8 : std_logic;
    signal p16 : std_logic;

    begin
        -- generation of parity bits
        p1 <= data_in(0) xor data_in(1) xor data_in(3) xor data_in(4) xor
              data_in(6) xor data_in(8) xor data_in(10);

        p2 <= data_in(0) xor data_in(2) xor data_in(3) xor data_in(5) xor
              data_in(6) xor data_in(9) xor data_in(10);

        p4 <= data_in(1) xor data_in(2) xor data_in(3) xor data_in(7) xor
              data_in(8) xor data_in(9) xor data_in(10);

        p8 <= data_in(4) xor data_in(5) xor data_in(6) xor data_in(7) xor
              data_in(8) xor data_in(9) xor data_in(10);

        -- generation of extra parity bit
        p16 <= data_in(0) xor data_in(1) xor data_in(2) xor data_in(3) xor
               data_in(4) xor data_in(5) xor data_in(6) xor data_in(7) xor
               data_in(8) xor data_in(9) xor data_in(10)  xor p1 xor p2 xor
               p4 xor p8;

        --set output bits
        data_encoded <= p16 & data_in(10) & data_in(9) & data_in(8) & data_in(7)
                        & data_in(6) & data_in(5) & data_in(4) &p8 & data_in(3)
                        & data_in(2) & data_in(1) & p4 & data_in(0) & p2 & p1;

    end beh;
```

## Decoder

This module has the task of decoding the code word. In more detail, it has the task of detecting a double bit error and correcting a single bit error.

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

entity Decoder is
    port(
        data_encoded : in std_logic_vector(15 downto 0);
        data_out : out std_logic_vector(10 downto 0);
        NE_bit : out std_logic;
        SEC_bit : out std_logic;
        DED_bit : out std_logic
    );
end Decoder;

architecture beh of Decoder is
    signal c1 : std_logic;
    signal c2 : std_logic;
    signal c4 : std_logic;
    signal c8 : std_logic;
    signal p : std_logic;
    signal c : std_logic_vector(3 downto 0);
```

```
    signal temp_cw : std_logic_vector(15 downto 0);

    begin

--*********************************************************************************
--                              ERROR DETECTION
--*********************************************************************************

        -- generation of control bits
        c1 <= (data_encoded(0) xor data_encoded(2)) xor (data_encoded(4) xor
              data_encoded(6)) xor (data_encoded(8) xor data_encoded(10)) xor
              (data_encoded(12) xor data_encoded(14));

        c2 <= (data_encoded(1) xor data_encoded(2)) xor (data_encoded(5) xor
              data_encoded(6)) xor (data_encoded(9) xor data_encoded(10)) xor
              (data_encoded(13) xor data_encoded(14));

        c4 <= (data_encoded(3) xor data_encoded(4)) xor (data_encoded(5) xor
              data_encoded(6)) xor (data_encoded(11) xor data_encoded(12)) xor
              (data_encoded(13) xor data_encoded(14));

        c8 <= (data_encoded(7) xor data_encoded(8)) xor (data_encoded(9) xor
              data_encoded(10)) xor (data_encoded(11) xor data_encoded(12)) xor
              (data_encoded(13) xor data_encoded(14));

        p <= (data_encoded(0) xor data_encoded(1)) xor (data_encoded(2) xor
             data_encoded(3)) xor (data_encoded(4) xor data_encoded(5)) xor
             (data_encoded(6) xor data_encoded(7)) xor (data_encoded(8) xor
             data_encoded(9)) xor (data_encoded(10) xor data_encoded(11)) xor
             (data_encoded(12) xor data_encoded(13)) xor (data_encoded(14) xor
             data_encoded(15));

        c <= c8 & c4 & c2 & c1;


--*********************************************************************************
        --      ERROR CORRECTION
--*********************************************************************************

        -- when c=0 and p=1 there was an error in p16
        temp_cw(15) <= not(data_encoded(15)) when c="0000" and p='1' else
                       data_encoded(15);

        --it must be taken into account that the position of the bits in
          data encoded is shifted by 1 as it goes from 0 to 15

        --when c="1111" (15) there was an error in 14-bit
        temp_cw(14) <= not(data_encoded(14)) when c="1111" and p='1' else
                       data_encoded(14);

        --when c="1110" (14) there was an error in 13-bit
        temp_cw(13) <= not(data_encoded(13)) when c="1110" and p='1' else
                       data_encoded(13);

        --when c="1101" (13) there was an error in 12-bit
        temp_cw(12) <= not(data_encoded(12)) when c="1101" and p='1' else
                       data_encoded(12);

        --when c="1100" (12) there was an error in 11-bit
        temp_cw(11) <= not(data_encoded(11)) when c="1100" and p='1' else
                       data_encoded(11);
```

```
        --when c="1011" (11) there was an error in 10-bit
        temp_cw(10) <= not(data_encoded(10)) when c="1011" and p='1' else
                       data_encoded(10);

        --when c="1010" (10) there was an error in 9-bit
        temp_cw(9) <= not(data_encoded(9)) when c="1010" and p='1' else
                      data_encoded(9);

        --when c="1001" (9) there was an error in 8-bit
        temp_cw(8) <= not(data_encoded(8)) when c="1001" and p='1' else
                      data_encoded(8);

        --when c="0111" (7) there was an error in 6-bit
        temp_cw(6) <= not(data_encoded(6)) when c="0111" and p='1' else
                      data_encoded(6);

        --when c="0110" (6) there was an error in 5-bit
        temp_cw(5) <= not(data_encoded(5)) when c="0110" and p='1' else
                      data_encoded(5);

        --when c="0101" (5) there was an error in 4-bit
        temp_cw(4) <= not(data_encoded(4)) when c="0101" and p='1' else
                      data_encoded(4);

        --when c="0011" (3) there was an error in 2-bit
        temp_cw(2) <= not(data_encoded(2)) when c="0011" and p='1' else
                      data_encoded(2);

        temp_cw(7) <= data_encoded(7);
        temp_cw(3) <= data_encoded(3);
        temp_cw(1 downto 0) <= data_encoded(1 downto 0);


        --set outputs
        data_out <= temp_cw(14 downto 8) & temp_cw(6 downto 4) & temp_cw(2);

        NE_bit <= '1' when c="0000" and p='0' else '0';
        SEC_bit <= '1' when (c/="0000" and p='1') or (c="0000" and p='1') else
                   '0';
        DED_bit <= '1' when c/="0000" and p='0' else '0';

    end beh;
```

## Test-plan

In order to verify the correctness of the system the following tests were made:

1. **Unit tests**: following the bottom-up strategy, each sub-module, after completing the implementation, has a dedicated testbench in order to check the correctness of the single sub-module in isolation.
2. A test with **two Device Under Test** (**DUT**): the encoder and the decoder. In this way, it was possible to insert errors on the flow in the decoder input.

### Unit tests

#### ECC

The ECC testbench is not shown because it is not relevant, in fact what is given in input is returned in output (it is not possible to insert the errors in the decoder input due to how the ECC is structured).

In order to test the correct functioning of the overall architecture, it is necessary to do a testbench in which there are two Device Under Test (see later).

#### Encoder

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

--entity declaration
entity Encoder_tb is
end Encoder_tb;

--architecture body
architecture rtl of Encoder_tb is

    constant clk_period : time := 8 ns;

    component Encoder is
        port(
            data_in : in std_logic_vector(10 downto 0);
            data_encoded : out std_logic_vector(15 downto 0)
        );
    end component;

    signal clk : std_logic := '0';
    signal data_in: std_logic_vector(10 downto 0) := (others =>
                    '0');
    signal data_encoded: std_logic_vector(15 downto 0);
    signal testing : boolean := true;

    begin
        clk <= not clk after clk_period/2 when testing else '0';

        dut: Encoder
        port map(
            data_in => data_in,
```

```
            data_encoded => data_encoded
        );

        stimulus : process
        begin
            data_in <= (others => '0');
            wait for 16 ns;

            data_in <= "01010101011";
            --expected output "0010101011010101" -> 10965
            wait for 16 ns;

            data_in <= "11100011100";
            --expected output "1111000101101000" -> 61800
            wait for 16 ns;

            data_in <= "10101010101";
            --expected output "0101010100101101" -> 21805
            wait for 16 ns;

            data_in <= "00000000000";
            --expected output "0000000000000000" -> 0
            wait for 16 ns;

            data_in <= "11111111111";
            --expected output "1111111111111111" -> 65535
            wait for 16 ns;

            wait until rising_edge(clk);
            testing <= false;
        end process;

        end rtl;
```

In the Figure 11 is shown the output of the encoder simulation obtained with Modelsim.

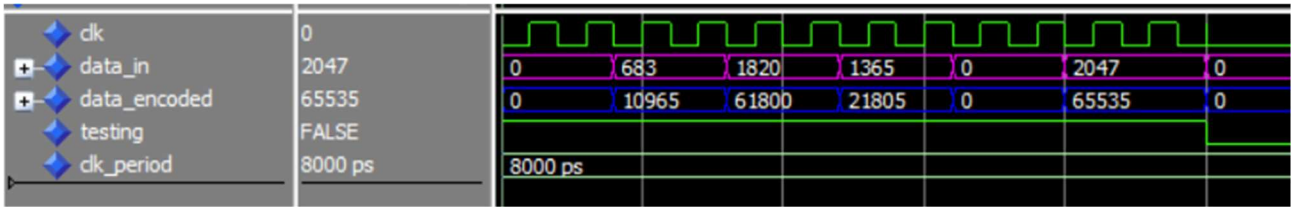


*Figure 11 - Encoder simulation*

## Decoder

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

--entity declaration
entity Decoder_tb is
end Decoder_tb;
```

```
--architecture body
architecture rtl of Decoder_tb is

    constant clk_period : time := 8 ns;

    component Decoder is
        port(
            data_encoded : in std_logic_vector(15 downto 0);
            data_out : out std_logic_vector(10 downto 0);
            NE_bit : out std_logic;
            SEC_bit : out std_logic;
            DED_bit : out std_logic
        );
    end Component;

    signal clk : std_logic := '0';
    signal data_encoded: std_logic_vector(15 downto 0) := (others
                         => '0');
    signal data_out: std_logic_vector(10 downto 0);
    signal NE_bit : std_logic;
    signal SEC_bit : std_logic;
    signal DED_bit : std_logic;
    signal testing : boolean := true;

    begin
        clk <= not clk after clk_period/2 when testing else '0';

        dut: Decoder
        port map(
            data_encoded => data_encoded,
            data_out => data_out,
            NE_bit => NE_bit,
            SEC_bit => SEC_bit,
            DED_bit => DED_bit
        );

        stimulus : process
        begin
            data_encoded <= (others => '0');
            wait for 32 ns;

            -- correct input-> "0010101011010101" (10965)

            --assume that there was not an error
            -- expected out -> data_out= "01010101011" (683),
                               NE=1, SEC=DED=0
            data_encoded <= "0010101011010101";
            wait for 32 ns;

            --assume that there was a single error in p16
            -- expected out -> data_out= "01010101011" (683),
                               NE=0, SEC=1, DED=0
```

```
        data_encoded <= "0010101011010100";
        wait for 32 ns;

        --assume that there was a single error in d10
        -- expected out -> data_out= "01010101011" (683),
                           NE=0, SEC=1, DED=0
        data_encoded <= "0000101011010101";
        wait for 32 ns;

        --assume that there was a single error in d11
        -- expected out -> data_out= "01010101011" (683),
                           NE=0, SEC=1, DED=0
        data_encoded <= "0110101011010101";
        wait for 32 ns;

        --assume that there was a single error in d3
        -- expected out -> data_out= "01010101011" (683),
                           NE=0, SEC=1, DED=0
        data_encoded <= "0010101011110101";
        wait for 32 ns;

        --assume that there was a single error in p8
        -- expected out -> data_out= "01010101011" (683),
                           NE=0, SEC=1, DED=0
        data_encoded <= "0010101001010101";
        wait for 32 ns;

        --assume that there was a double error in p8 and d1
        -- expected out -> data_out=invalid info, NE=0, SEC=0,
                           DED=1
        data_encoded <= "0010101001010001";
        wait for 32 ns;

        --assume that there was a double error in d10 and d9
        -- expected out -> data_out=invalid info, NE=0, SEC=0,
                           DED=1
        data_encoded <= "0010110011010101";
        wait for 32 ns;

        wait until rising_edge(clk);
        testing <= false;
    end process;

    end rtl;
```

In the Figure 12 is shown the output of the decoder simulation obtained with Modelsim.

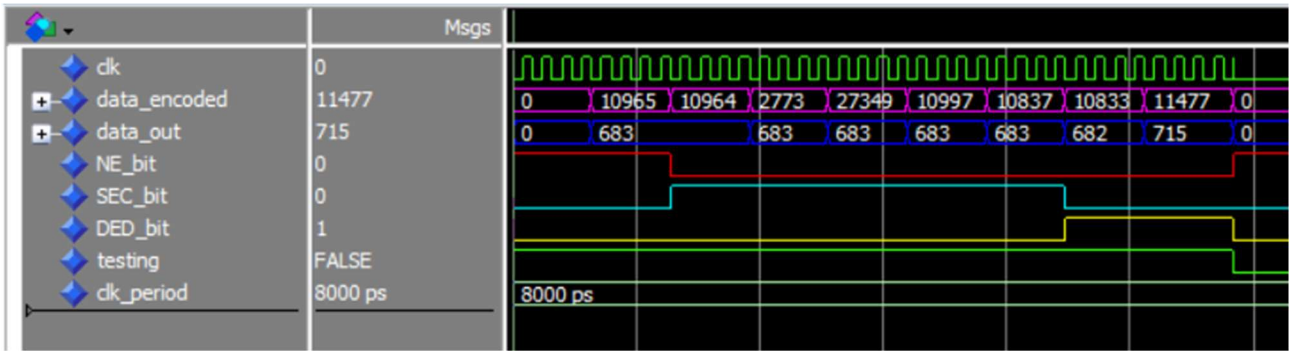


*Figure 12 - Decoder simulation*

## Final test

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

--entity declaration
entity Enc_Dec_tb is
end Enc_Dec_tb;

--architecture body
architecture rtl of Enc_Dec_tb is

    constant clk_period : time := 8 ns;

    component Encoder is
        port(
            data_in : in std_logic_vector(10 downto 0);
            data_encoded : out std_logic_vector(15 downto 0)
        );
    end component;

    component Decoder is
        port(
            data_encoded : in std_logic_vector(15 downto 0);
            data_out : out std_logic_vector(10 downto 0);
            NE_bit : out std_logic;
            SEC_bit : out std_logic;
            DED_bit : out std_logic
        );
    end component;

    signal clk : std_logic := '0';
    signal data_in: std_logic_vector(10 downto 0) := (others =>
                    '0');
    signal data_encoded_out: std_logic_vector(15 downto 0) :=
                             (others => '0');
    signal data_encoded_in: std_logic_vector(15 downto 0) :=
                            (others => '0');
    signal data_out: std_logic_vector(10 downto 0);
```

```
    signal NE_bit : std_logic;
    signal SEC_bit : std_logic;
    signal DED_bit : std_logic;
    signal testing : boolean := true;

    begin
        clk <= not clk after clk_period/2 when testing else '0';

        i_dut1: Encoder
        port map(
            data_in => data_in,
            data_encoded => data_encoded_out
        );

        i_dut2: Decoder
        port map(
            data_encoded => data_encoded_in,
            data_out => data_out,
            NE_bit => NE_bit,
            SEC_bit => SEC_bit,
            DED_bit => DED_bit
        );

        stimulus : process
        begin
            data_in <= (others => '0');
            data_encoded_in <= (others => '0');
            wait for 32 ns;

            --assume that there was not an error
            -- expected out -> data_out= "01010101011" (683),
                               NE=1, SEC=DED=0
            data_in <= "01010101011";
            data_encoded_in <= data_encoded_out;
            wait for 32 ns;

            --assume that there was a single error in p16
            -- expected out -> data_out= "01010101011" (683),
                               NE=0, SEC=1, DED=0
            data_in <= "01010101011";
            data_encoded_in <= not(data_encoded_out(15)) &
                               data_encoded_out(14 downto 0);
            wait for 32 ns;

            --assume that there was a single error in d10
            -- expected out -> data_out= "01010101011" (683),
                               NE=0, SEC=1, DED=0
            data_in <= "01010101011";
            data_encoded_in <= data_encoded_out(15 downto 14) &
                               not(data_encoded_out(13)) &
                               data_encoded_out(12 downto 0);
            wait for 32 ns;
```

```
--assume that there was a single error in d11
-- expected out -> data_out= "01010101011" (683),
                    NE=0, SEC=1, DED=0
data_in <= "01010101011";
data_encoded_in <= data_encoded_out(15) &
                   not(data_encoded_out(14)) &
                   data_encoded_out(13 downto 0);
wait for 32 ns;

--assume that there was a single error in d3
-- expected out -> data_out= "01010101011" (683),
                    NE=0, SEC=1, DED=0
data_in <= "01010101011";
data_encoded_in <= data_encoded_out(15 downto 6) &
                   not(data_encoded_out(5)) &
                   data_encoded_out(4 downto 0);
wait for 32 ns;

--assume that there was a single error in p8
-- expected out -> data_out= "01010101011" (683),
                    NE=0, SEC=1, DED=0
data_in <= "01010101011";
data_encoded_in <= data_encoded_out(15 downto 8) &
                   not(data_encoded_out(7)) &
                   data_encoded_out(6 downto 0);
wait for 32 ns;

--assume that there was a double error in p8 and d1
-- expected out -> data_out=invalid info, NE=0, SEC=0,
                    DED=1
data_in <= "01010101011";
data_encoded_in <= data_encoded_out(15 downto 8) &
                   not(data_encoded_out(7)) &
                   data_encoded_out(6 downto 3) &
                   not(data_encoded_out(2)) &
                   data_encoded_out(1 downto 0);
wait for 32 ns;

--assume that there was a double error in d10 and d9
-- expected out -> data_out=invalid info, NE=0, SEC=0,
                    DED=1
data_in <= "01010101011";
data_encoded_in <= data_encoded_out(15 downto 14) &
                   not(data_encoded_out(13)) &
                   not(data_encoded_out(12)) &
                   data_encoded_out(11 downto 0);
wait for 32 ns;

--assume that there was a double error in d4 and d7
-- expected out -> data_out=invalid info, NE=0, SEC=0,
                    DED=1
```

```
            data_in <= "01010101011";
            data_encoded_in <= data_encoded_out(15 downto 11) &
                               not(data_encoded_out(10)) &
                               data_encoded_out(9 downto 7) &
                               not(data_encoded_out(6)) &
                               data_encoded_out(5 downto 0);
            wait for 32 ns;

            wait until rising_edge(clk);
            testing <= false;
        end process;

    end rtl;
```

In the Figure 13 is shown the output of the final architecture simulation obtained with Modelsim.

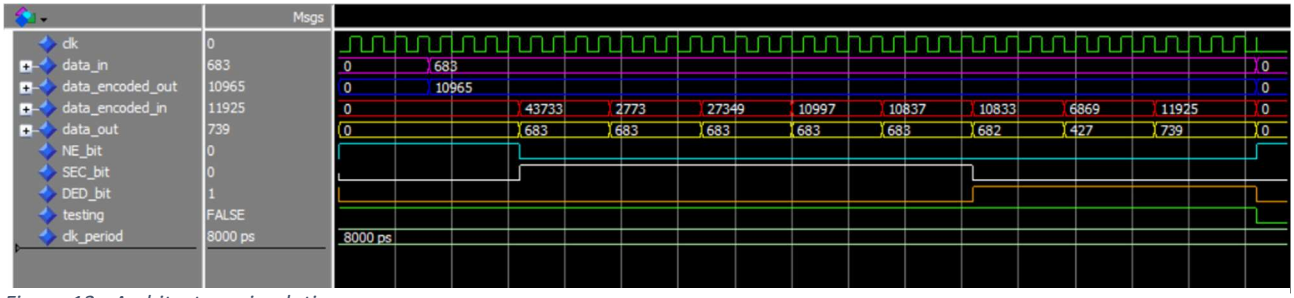


*Figure 13 - Architecture simulation*

# Synthesis

In this chapter, will be presented the results obtained by creating a project with **Xilinx VIVADO** by selecting the **Zybo Zynq-7000** (xc7z010clg400-1) as working device. The Implementation phase has not been performed because it is not required from the assignment.

## RTL Analysis

Before heading with the Synthesis a preliminary double-check of the correctness of the system has been made by simply opening the schema obtained by the **Elaborated Design**. No problem has been found at this stage.

## Timing Report

A timing constraint has been added (clock with period of 8ns), and after running the Synthesis command, the following Timing Report has been displayed (Figure 14):

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 3,821 ns | Worst Hold Slack (WHS): | 0,161 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: | 30 | Total Number of Endpoints: | 30 | Total Number of Endpoints: | 42 |

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

*Figure 14 - Timing Report*

As we can see, the Worst Negative Slack (WNS) is positive, so we can drive the board at a higher frequency than 125MHz. We can calculate the maximum frequency as:

$$f_{max} = \frac{1}{T_{clk} - WNS} = 239.2\ MHz$$

$T_{clk}$ is given by the Zybo Board, which operates with 125MHz and, for this reason, will grant a $T_{clk} = {}^{1}/_{125MHz} = 8ns$.

The WNS is determined by the Critical Path of the architecture, which is shown in the following picture (Figure 15):

| Name | Slack ^1 | Levels | Routes | High Fanout | From | To | Total Delay | Logic D |
|---|---|---|---|---|---|---|---|---|
| Path 1 | 3.821 | 3 | 4 | 14 | out_encoder_reg/q_reg[15]/C | out_decoder_reg_data/q_reg[0]/D | 4.028 | ( |
| Path 2 | 3.821 | 3 | 4 | 14 | out_encoder_reg/q_reg[15]/C | out_decoder_reg_data/q_reg[10]/D | 4.028 | ( |
| Path 3 | 3.821 | 3 | 4 | 14 | out_encoder_reg/q_reg[15]/C | out_decoder_reg_data/q_reg[1]/D | 4.028 | ( |
| Path 4 | 3.821 | 3 | 4 | 14 | out_encoder_reg/q_reg[15]/C | out_decoder_reg_data/q_reg[2]/D | 4.028 | ( |
| Path 5 | 3.821 | 3 | 4 | 14 | out_encoder_reg/q_reg[15]/C | out_decoder_reg_data/q_reg[3]/D | 4.028 | ( |
| Path 6 | 3.821 | 3 | 4 | 14 | out_encoder_reg/q_reg[15]/C | out_decoder_reg_data/q_reg[4]/D | 4.028 | ( |
| Path 7 | 3.821 | 3 | 4 | 14 | out_encoder_reg/q_reg[15]/C | out_decoder_reg_data/q_reg[5]/D | 4.028 | ( |
| Path 8 | 3.821 | 3 | 4 | 14 | out_encoder_reg/q_reg[15]/C | out_decoder_reg_data/q_reg[6]/D | 4.028 | ( |
| Path 9 | 3.821 | 3 | 4 | 14 | out_encoder_reg/q_reg[15]/C | out_decoder_reg_data/q_reg[7]/D | 4.028 | ( |
| Path 10 | 3.821 | 3 | 4 | 14 | out_encoder_reg/q_reg[15]/C | out_decoder_reg_data/q_reg[8]/D | 4.028 | ( |

*Figure 15 - Critical Path Report*

We can state that the Decoder module for the output data calculation has the most impact on the critical path.

## Resource Utilization Report

The resource utilized by the architecture synthesized are the following (Figure 16):

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 28 | 17600 | 0.16 |
| FF | 41 | 35200 | 0.12 |
| IO | 27 | 100 | 27.00 |

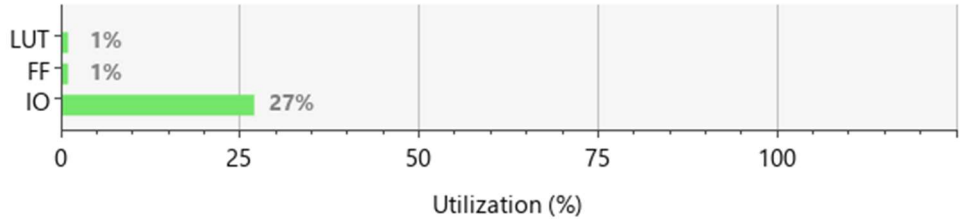


*Figure 16 - Resource Utilization Report*

## Power Consumption Report

The Power Consumption Report is the following (Figure 17):

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:** | **0.102 W** |
| **Design Power Budget:** | **Not Specified** |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **26,2°C** |
| Thermal Margin: | 58,8°C (5,0 W) |
| Effective ϑJA: | 11,5°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

Launch Power Constraint Advisor to find and fix invalid switching activity

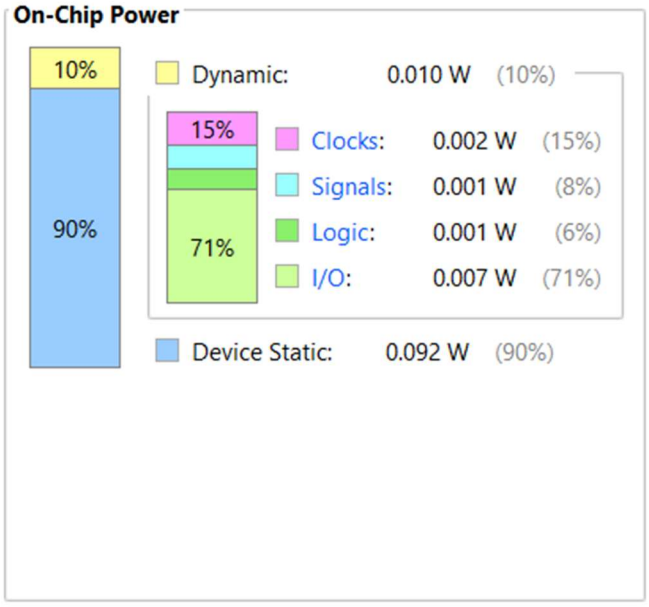


*Figure 17 - Power Consumption Report*

As we can see, a total of 0.102W of power are needed, which is divided by 10% into dynamic power and 90% into static power. For the dynamic power, consumption the most relevant contribute is from the I/O.

## Warning Messages

The tool did not report any relevant warnings, other than those relating to the non-assignment of the variables to the I/O pin of the selected board.

## Conclusion

In general, this type of Error Correcting Code is very useful, as it is possible to detect two errors and correct one. It is, therefore, a simple solution to implement and rather effective, ideal for those applications which are not prone to errors in themselves and which require further control that is not too heavy.