

Computer Engineering

Electronic and Communication Systems

Error correction code

Project Report

Federica Perrone

Summary

[Introduction 3](#_Toc90885117)

[Overview of Hamming Code 3](#_Toc90885118)

[General Algorithm 4](#_Toc90885119)

[Hamming Encoder 5](#_Toc90885120)

[Hamming Decoder 5](#_Toc90885121)

[Extended Hamming Code 5](#_Toc90885122)

[Applications 5](#_Toc90885123)

[Architecture description 6](#_Toc90885124)

[Encoder Architecture 7](#_Toc90885125)

[Decoder Architecture 7](#_Toc90885126)

[VHDL Code 10](#_Toc90885127)

[Modules List 10](#_Toc90885128)

[Test-plan 11](#_Toc90885129)

[Synthesis 12](#_Toc90885130)

[Conclusion 13](#_Toc90885131)

# 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:

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 (P1) covers all bit positions which have the least significant bit set: bit 1 (the parity bit itself P1), 3, 5, 7, 9, etc. (all the odd numbers);
* Parity bit 2 (P2) covers all bit positions which have the second least significant bit set: bits 2 (P2), 3, 6, 7, 10, 11, etc. (sets of 2);
* Parity bit 4 (P4) covers all bit positions which have the third least significant bit set: bits 4 (P4), 5, 6, 7, 12, 13, 14, 15, etc. (sets of 4);
* Parity bit 8 (P8) covers all bit positions which have the fourth least significant bit set: bits 8 (P8), 9, 10, 11, 12, 13, 14, 15, etc. (sets of 8);
* Parity bit 16 (P16) covers all bit positions which have the fifth least significant bit set: bits 16 (P16), 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 | | **P1** | **P2** | **D1** | **P4** | **D2** | **D3** | **D4** | **P8** | **D5** | **D6** | **D7** | **D8** | **D9** | **D10** | **D11** | **P16** |
| Encoded data coverage | **P1** |  |  | ✓ |  | ✓ |  | ✓ |  | ✓ |  | ✓ |  | ✓ |  | ✓ |  |
| **P2** |  |  | ✓ |  |  | ✓ | ✓ |  |  | ✓ | ✓ |  |  | ✓ | ✓ |  |
| **P4** |  |  |  |  | ✓ | ✓ | ✓ |  |  |  |  |  | ✓ | ✓ | ✓ |  |
| **P8** |  |  |  |  |  |  |  |  | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |  |
| **P16** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

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

n

Encoder

n+k

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

Information bits (n-bits)

XOR gate

Codeword (n+ k-bits)

### Hamming Decoder

Figure 4 - Decoder block diagram

n+k

Decoder

n

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).

Codeword (n + k-bits)

XOR gate

Syndrome (k-bits)

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, P16 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).

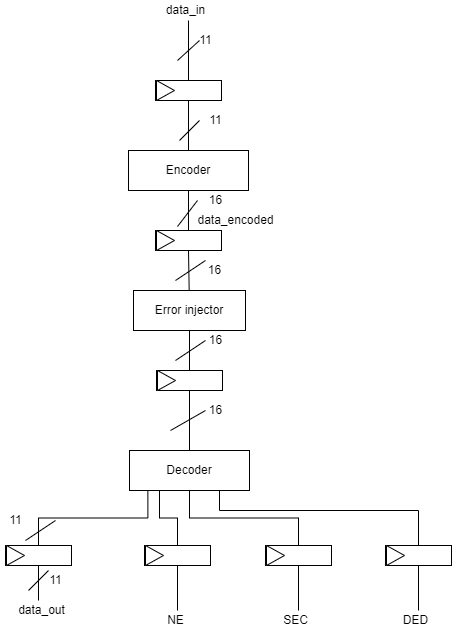
The general structure could be summarized by the following schema (Figure 6):

Figure - General schema

## Encoder Architecture

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 D1 to D11) and 5-bits of parity (P1, P2, P4, P8, P16).

Figure 7 - Encoder block diagram

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

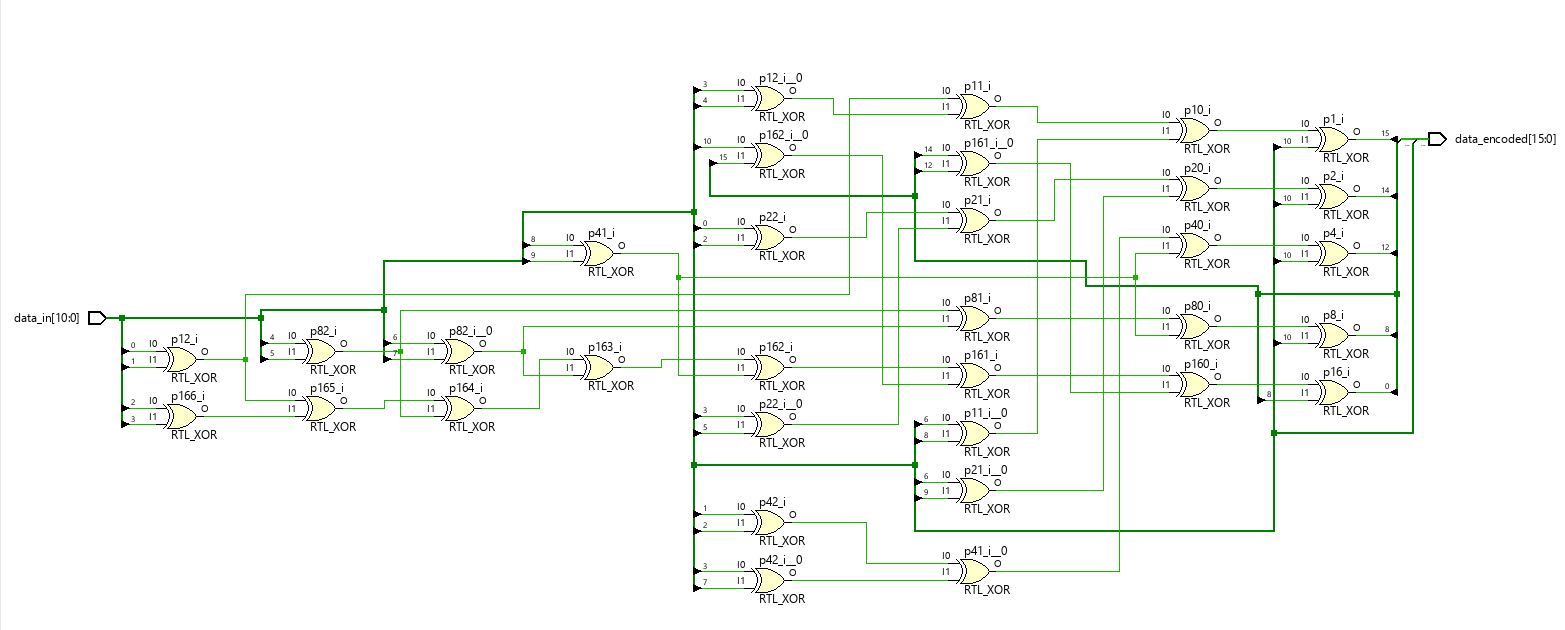
The following figure (Figure 8) shows the encoder circuit:

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).

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.

Figure 9 - Decoder block diagram

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

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 P16, 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 P16. As in the second case, also in this case the output bit SEC is set to 1.

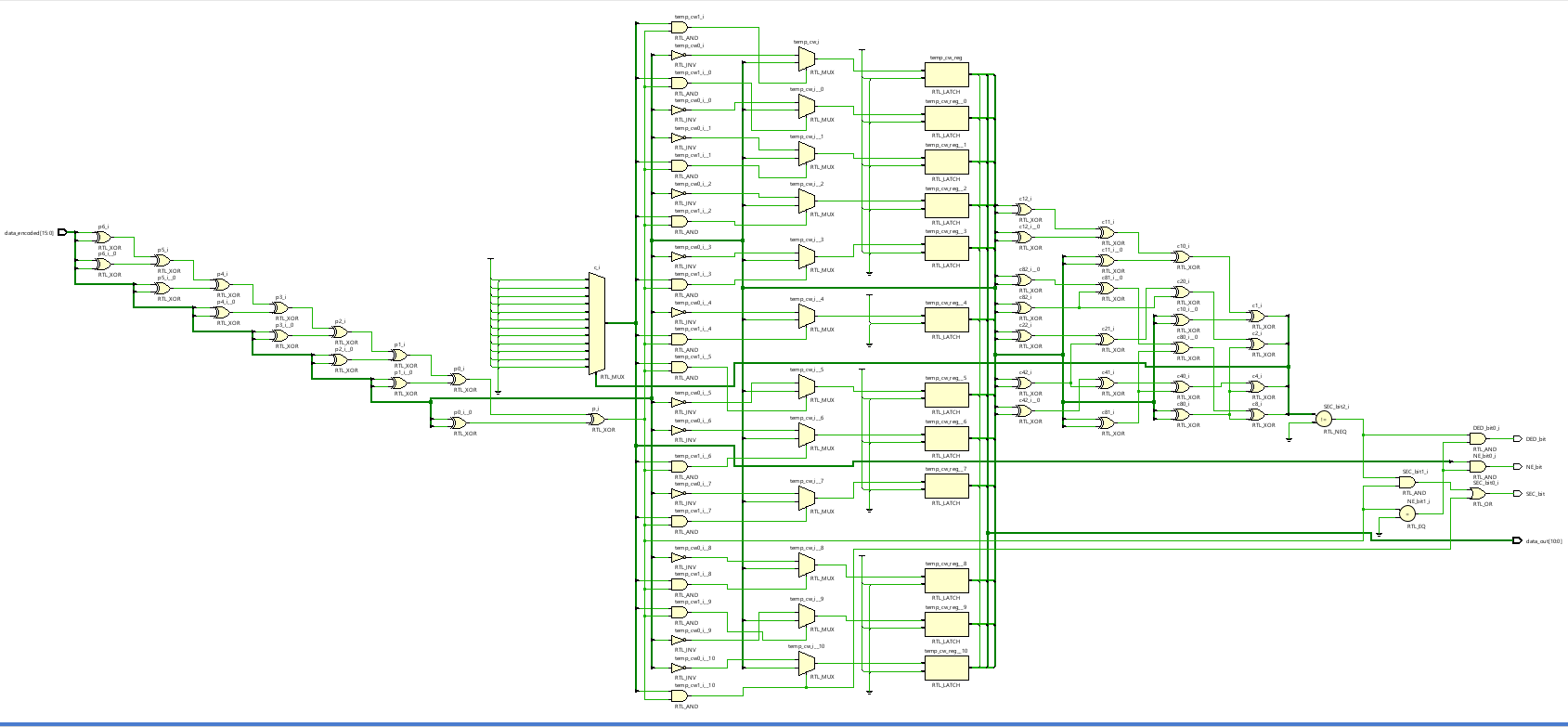
The following figure (Figure 10) shows the decoder circuit:

Figure - Decoder circuit

# 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.

# 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.

# Synthesis

# Conclusion