

# Project - Error Correcting Code

Mariano Basile

25th January 2016

## Contents

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Descrizione del problema . . . . .	3
1.2	Codice di Hamming . . . . .	4
1.3	L'algoritmo . . . . .	4
1.3.1	Calcolo dei bit di parità Px . . . . .	5
1.4	Architettura . . . . .	6
1.5	CODICE VHDL . . . . .	7
1.6	Sintesi . . . . .	11
1.7	Codificatore di Hamming in C++ . . . . .	14

# 1 Introduzione

## 1.1 Descrizione del problema

In genere per poter codificare  $N$  simboli distinti (parole) con un codice binario, occorrono  $n \geq \lceil \log_2 N \rceil$  bit. Tuttavia, nei casi pratici, quello che si costruisce è un codice cosiddetto ridondante, ovvero la codifica degli  $N$  simboli distinti avviene su  $n' = n + m$  bit, usando cioè  $m$  bit aggiuntivi rispetto agli  $n$  bit strettamente richiesti. L'aggiunta di bit di ridondanza permette di costruire codici che consentono di rilevare e correggere eventuali errori di trasmissione dovuti (ad esempio) alla presenza di rumore sul canale di trasmissione. Si hanno due tipi di codici ridondanti:

- Codici a rilevazione di errore - Consentono di individuare la presenza di un errore;
- Codici a correzione di errore - Consentono non solo di individuare la presenza di un errore, ma anche di identificarne la posizione in modo da poterlo correggere;

La probabilità di errore sul singolo bit inviato sul canale è legata a diversi fattori quali la potenza media del segnale in ricezione, alla bit rate di trasmissione, allo spettro di densità di potenza del rumore presente sul canale, al tipo di canale di comunicazione in questione (con rumore additivo, moltiplicativo, ecc.). Lo svantaggio di una codifica ridondante è che si devono trasmettere più bit di quanti ne siano effettivamente necessari per rappresentare il messaggio quindi, a parità di altre condizioni, cresce il tempo richiesto per la trasmissione. Esistono diverse tipologie di codici che si distinguono per il modo con cui introducono ridondanza nel messaggio. In particolare, i codici possono essere divisi in 2 grandi classi:

- codici a blocchi
- codici convoluzionali.

La classe dei codici a blocchi può essere ulteriormente suddivisa in sottoclassi, come mostrato nella figura sottostante:

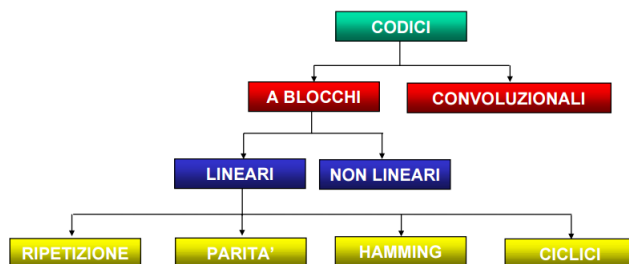


Figure 1: Tipologie di codici

## 1.2 Codice di Hamming

Il codice di Hamming appartiene alla categoria dei *codici correttore lineari*. Consideriamo il caso in cui un decodificatore sia in grado di rilevare e correggere un singolo bit errato sulla parola ricevuta e sia in grado di rilevare (ma non correggere) un doppio errore. Per una ricezione corretta la distanza di Hamming di ogni bit tra le due parole deve essere 0. Ma cosa si intende per **distanza di Hamming**? La distanza di Hamming fra due parole (in genere ci si riferisce alla distanza di Hamming tra la parola trasmessa e quella ricevuta), si ottiene contando il numero di bit diversi in posizioni corrispondenti. È banalmente ottenibile attraverso un XOR bit a bit e successivo conteggio dei bit a 1. Se due parole hanno una distanza di Hamming uguale a  $d$ , ci vogliono esattamente  $d$  errori su singoli bit per trasformare l'una nell'altra. **La distanza minima**  $d_{min}$  è la minima distanza di Hamming fra tutte le possibili coppie di parole. Possiamo quindi affermare che se un codice ha distanza minima  $d_{min}$ , allora, se  $d_{min}$  è un numero dispari, il codice è in grado di rilevare e correggere  $\lfloor \frac{d}{2} \rfloor$  errori, altrimenti, se  $d_{min}$  è pari, esso rileva e corregge  $(\frac{d}{2}-1)$  errori e rileva soltanto (senza correggere)  $\frac{d}{2}$  errori.

## 1.3 L'algoritmo

Il codice di Hamming fornisce un algoritmo per generare codici ridondanti correttivi, tali che sia immediata l'indicazione degli eventuali bit errati nella parola. È generato aggiungendo a ciascuna parola alcuni bit di parità. Siano:

- $n$ , bit della parola (informazione);
- $r$ , bit di ridondanza;
- $n'=n+r$ , bit della CODEWORD generata;

Ognuna delle  $2^n$  parole, ha  $n'$  CODEWORD errate a distanza di Hamming 1, ottenute cambiando un bit alla volta nella parola originaria. Ognuna delle  $2^n$  parole richiede quindi  $(n'+1)$  configurazioni di bit dedicate. **Per cui il minimo numero di bit di ridondanza  $r$  sarà dato dal più piccolo  $r$  che risolve:**

$$2^{n'} \geq 2^n (n' + 1)$$

$$2^n \cdot 2^r \geq 2^n (n + r + 1)$$

$$2^r \geq (n + r + 1)$$

In ingresso al nostro sistema abbiamo una parola su  $n=11$  quindi il **più piccolo  $r$  che risolve la disuguaglianza di cui sopra è 4**. Nel nostro caso siamo di fronte a un codice di Hamming con distanza  $d=4$  in quanto  $n=11$  e  $n'+1=16$ . Abbiamo quindi bisogno di ulteriore bit di parità. Dove andranno inseriti i bit di parità nella CODEWORD in uscita?

In un codice di Hamming tutti i bit in posizione  $x$ , con  $x$  potenza di 2, sono bit di parità ( $P_x$ ), gli altri sono bit di informazione della parola.

I bit di parità saranno quindi nelle posizioni  $2^0, 2^1, 2^2, 2^3, 2^4$  rispettivamente 1, 2, 4, 8, 16.

Gli 11 bit dati  $\mathbf{Dx}$  di ciascuna parola vengono traslati in modo da lasciare libere le posizioni potenze di 2 in cui verranno inseriti i bit di parità.

La CODEWORD generata avrà quindi la seguente struttura:

Posizione	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Codifica	P1	P2	D1	P4	D2	D3	D4	P8	D5	D6	D7	D8	D9	D10	D11	PO

Figure 2: **Struttura della CODEWORD trasmessa**

### 1.3.1 Calcolo dei bit di parità $Px$

Una volta stabiliti quanti bit di parità sono necessari rimane da stabilire come calcolarli. La procedura è standard: **per ogni bit di informazione in posizione  $i$  all'interno della CODEWORD, bisogna esprimere  $i$  come somma di potenze di 2. Supponiamo che  $i$  sia uguale alla somma  $a+b+c$  con  $a, b$  e  $c$  potenze di 2, allora il bit di informazione  $D_i$  entra nel calcolo dei bit di parità  $P_a, P_b, P_c$ .** Tale procedimento va iterato fino a quando tutti i bit di informazione della CODEWORD non sono stati esaminati. A questo punto il sigolo bit di parità è ottenuto attraverso l'operazione di XOR tra i bit 'selezionati'. Come si può notare, il numero di bit di parità cresce in maniera molto più lenta rispetto a quello dei bit di informazione. Vi si aggiunge un bit di parità alla CODEWORD ogni qual volta questa raggiunge una lunghezza che è una potenza di 2 (esempio: 1, 2, 4, 8, 16 . . . . bit). **Il nostro è un codice di Hamming a distanza 4 per cui valgono tutte le considerazioni fatte finora per il calcolo dei bit di parità P1,P2,P4,P8 mentre P16, da noi indentificato con PO, è ottenuto dall'OR esclusivo di tutti i bit (di informazione e di parità) appena generati.** In sintesi:

- $P1 = D1 \oplus D2 \oplus D4 \oplus D5 \oplus D7 \oplus D9 \oplus D11$
- $P2 = D1 \oplus D3 \oplus D4 \oplus D6 \oplus D7 \oplus D10 \oplus D11$
- $P4 = D2 \oplus D3 \oplus D4 \oplus D8 \oplus D9 \oplus D10 \oplus D11$
- $P8 = D5 \oplus D6 \oplus D7 \oplus D8 \oplus D9 \oplus D10 \oplus D11$
- $PO = P1 \oplus P2 \oplus D1 \oplus P4 \oplus D2 \oplus D3 \oplus D4 \oplus P8 \oplus D5 \oplus D6 \oplus D7 \oplus D8 \oplus D9 \oplus D10 \oplus D11$

Il codice di Hamming viene utilizzato in genere per fare rilevazione e correzione di errori nelle memorie.

## 1.4 Architettura

Il codificatore di Hamming è stato modellato in VHDL come un'entità "ENCODER" avente:

- In ingresso una WORD su 11 bit;
- In uscita una CODEWORD su 16 bit;

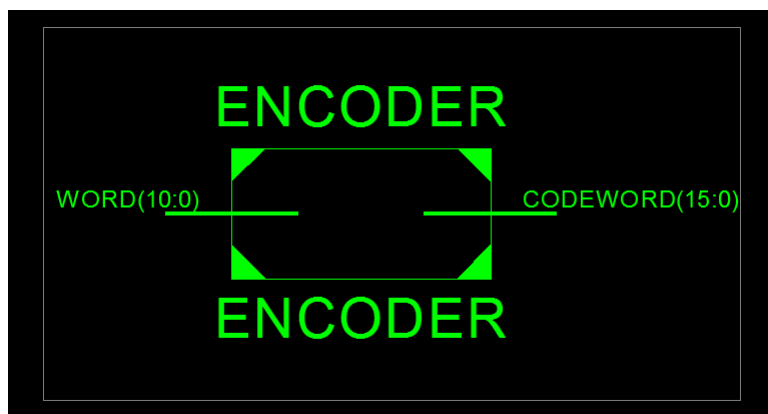


Figure 3: VHDL Entity "ENCODER"

**Questa blackbox è stata implementata come un circuito puramente combinatorio**, ottenuto nel seguente modo:

- Si procede prima col calcolare i bit di parità P1,P2,P4,P8: per ognuno si esegue una operazione di XOR che interessa solo alcuni bit informazione (quelli presentati alla fine del paragrafo 1.3.1);
- Si calcola il bit di parità PO attraverso uno XOR di tutti i bit (informazione e parità appena generati).
- Si determina la CODEWORD in uscita attraverso il concatenamento di tutti i bit (informazione e parità) secondo lo schema in Figura 2 (la posizione 1 si riferisce al bit meno significativo).

La rete combinatoria ottenuta è la seguente:

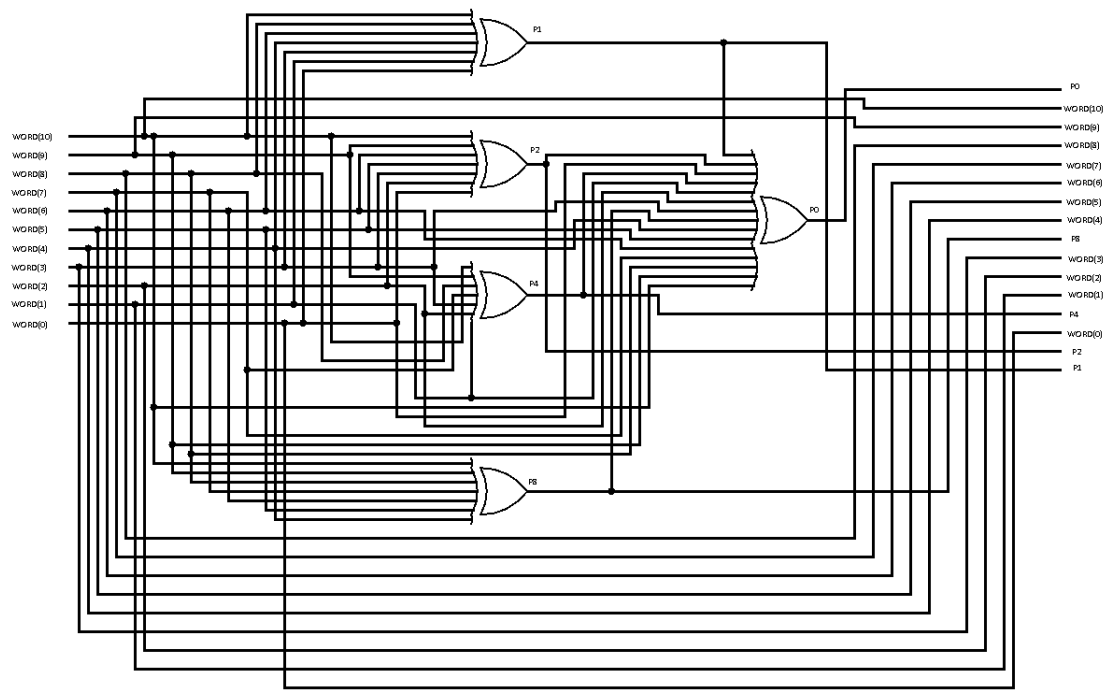


Figure 4: Rete Combinatoria per il codificatore di Hamming

## 1.5 CODICE VHDL

L'architettura poc'anzi esposta è stata implementata in VHDL per mezzo di una architettura *di tipo behavioural*. Il progetto si compone di due file, rispettivamente:

- ENCODER.vhd
- ENCODER\_TEST.vhd

```

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

ENTITY ENCODER is

    port(
        WORD      : in  std_logic_VECTOR (10 downto 0);
        CODEWORD   : out std_logic_VECTOR (15 downto 0)
    );

END ENCODER;

-- Describing functionality by a process()
ARCHITECTURE BEHAVIOURAL OF ENCODER IS
BEGIN
    ENCODING:process(WORD)
        variable P1,P2,P4,P8,PO:std_logic;
    begin
        --Computing parity bits Pi i=1,2,4,8.
        P1:= WORD(0) XOR WORD(1) XOR WORD(3) XOR WORD(4) XOR WORD(6) XOR WORD(8) XOR WORD(10);

        P2:= WORD(0) XOR WORD(2) XOR WORD(3) XOR WORD(5) XOR WORD(6) XOR WORD(9) XOR WORD(10);

        P4:= WORD(1) XOR WORD(2) XOR WORD(3) XOR WORD(7) XOR WORD(8) XOR WORD(9) XOR WORD(10);

        P8:= WORD(4) XOR WORD(5) XOR WORD(6) XOR WORD(7) XOR WORD(8) XOR WORD(9) XOR WORD(10);

        --Another parity bit PO (code's distance equals to 4) obtained by applying EXCLUSIVE OR
        --between all bits(data and parity ones).
        PO:= P1 XOR P2 XOR WORD(0) XOR P4 XOR WORD(1) XOR WORD(2) XOR WORD(3) XOR P8 XOR WORD(4)
            XOR WORD(5) XOR WORD(6) XOR WORD(7) XOR WORD(8) XOR WORD(9) XOR WORD(10);

        --Defining output
        CODEWORD <= PO & WORD(10) & WORD(9) & WORD(8) & WORD(7) & WORD(6) & WORD(5) &
            WORD(4) & P8 & WORD(3) & WORD(2) & WORD(1) & P4 & WORD(0) & P2 & P1;

    end process ENCODING;
END BEHAVIOURAL;

```

Figure 5: **ENCODER.vhd**

Per verificare che il codificatore di Hamming funzioni come da specifiche, è stato implementato il seguente test plan al fine di fornire degli input e verificare gli output corrispondenti. A tale scopo sono stati individuati dei *timing*, degli istanti temporali in cui fornire dei segnali in ingresso al sistema per poterne verificare la risposta. È stato quindi definito un particolare segnale di input chiamato *clk*, non richiesto dal codificatore, essendo una pura rete combinatoria, al fine di raggiungere tale scopo.



```

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

ENTITY ENCODER_tb IS
END ENCODER_tb;

ARCHITECTURE ENCODER_TEST OF ENCODER_tb IS

    COMPONENT ENCODER

    port(
        WORD      : in  std_logic_VECTOR (10 downto 0);
        CODEWORD   : out std_logic_VECTOR (15 downto 0)
    );

    END COMPONENT;

    -----
    --Just a logical test, but I may need some timing info after the synthesis tool
    --It's a good idea to set these timing info at this time.
    CONSTANT MckPer  : TIME      := 200 ns;  -- Master Clk period
    CONSTANT TestLen : INTEGER   := 24;      -- No. of Count (MckPer/2) for test

    -- I N P U T      S I G N A L S

    SIGNAL  clk  : std_logic := '0';
    SIGNAL  word : std_logic_VECTOR (10 downto 0) := "000000000000";

    -- O U T P U T      S I G N A L S

    SIGNAL  codeword : std_logic_VECTOR (15 downto 0);

```

Figure 6: TestBench di Verifica (parte 1)

In prima istanza si è incluso il design da testare, si è quindi definito il component ENCODER. Il test è solo un test logico, tuttavia lo scopo è quello di avere, successivamente a questa fase di simulazione logica, tale design su un gate fisico vero e proprio (FPGA per esempio). Sarebbe necessario a quel punto definire un nuovo testbench su un'altra architettura, ottenuta come risultato del processo di sintesi, dove però i timing risultano necessari. Per non definire un nuovo testbench, le informazioni sul timing sono state riportate in questa definizione, così che lo stesso file possa essere in seguito riutilizzato. È doveroso comunque rimarcare il concetto che in questa fase di progetto siamo *'technology independent'*.

La costante *TestLen* viene utilizzata dalla variabile *count* per interrompere la chiamata al processo. I segnali di ingresso sono:

- *clk*
- *word* (che ha lo stesso nome definito per la parola in ingresso all'encoder), inizializzato con 11 bit a 0.

```

-- SERVICE SIGNALS TO CREATE DATA SET OF THE TIMING
SIGNAL clk_cycle : INTEGER;
SIGNAL Testing: Boolean := True;

BEGIN
  -- I need first to put the design under the test
  ENCODER_INSTANCE : ENCODER PORT MAP(word,codeword);

  -- Create the clock: period of 200ns, with duty cycle of 0.5 and runs until Testing variable is true.
  clk    <= NOT clk AFTER MckPer/2 WHEN Testing ELSE '0';

  -- At the raising and falling edge of the clock the following process is executed
  TestingProcess: PROCESS(clk)
  VARIABLE count: INTEGER:= 0;
  BEGIN
    --In order to obtain the clk_cycle i need to divide variable count by 2
    --(Process is executed both at the raising and the falling edge of the clock).
    clk_cycle <= (count+1)/2;
    -- Run the simulation exactly TestLen times (after clk is set to zero)
    CASE count IS
      WHEN 3  => word <= "000000000001";
      WHEN 6  => word <= "000000000010";
      WHEN 9  => word <= "000000000011";
      WHEN 12 => word <= "000000000100";
      WHEN 15 => word <= "000000000101";
      WHEN 18 => word <= "10010100100";
      WHEN 21 => word <= "11111111111";
      WHEN (TestLen - 1) => Testing <= False;
      WHEN OTHERS => NULL;
    END CASE;
    count:= count + 1;
  END PROCESS TestingProcess;
END ENCODER_TEST;

```

Figure 7: TestBench Di Verifica(parte 2)

Nel testbench ho anzitutto bisogno di definire quale è il design per cui sto effettuando il test, quindi la prima istruzione dopo la keyword BEGIN serve a richiamare il componente. Viene in seguito creato il clock, al fine di generare stimuli differenti a tempi differenti. Quest'ultimo è implementato per mezzo di un operatore NOT, avrà un duty cycle del 50%, e verrà generato fino a quando la variabile *Testing*, definita precedentemente, non verrà settata al valore *false*.

Ogni volta che vi è un fronte in salita o un fronte in discesa del clock viene eseguito un processo definito *TestingProcess*: tale processo verifica ogni volta il valore della variabile *count* (che rappresenta *due volte il ciclo di clock*). **Tale test permette di distinguere i vari *timing steps* durante i quali fornire words differenti al codificatore.** Quando *count* (inizializzato a 0) raggiunge un valore pari a *TestLen-1*, la variabile *Testing* è settata a false e la simulazione termina. La simulazione dura quindi esattamente *TestLen cycles*.

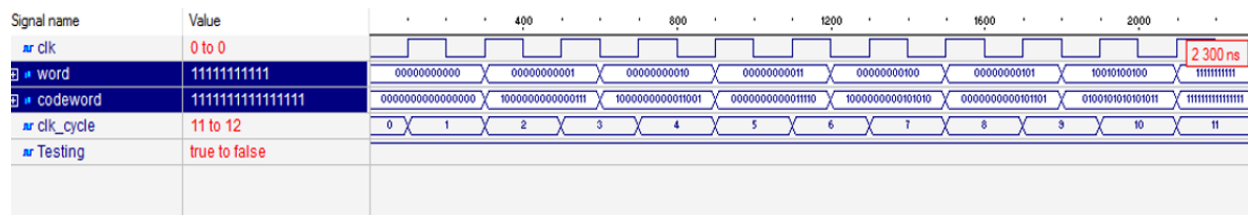


Figure 8: Simulazione Logica

## 1.6 Sintesi

Il progetto è stato poi successivamente sintetizzato su Xilinx Zync attraverso il tool ISE. Di seguito mostriamo alcuni screenshot riepilogativi:

ENCODER Project Status (01/23/2016 - 17:52:21)			
<b>Project File:</b>	HAMMING_ECC_ENCODER.xise	<b>Parser Errors:</b>	No Errors
<b>Module Name:</b>	ENCODER	<b>Implementation State:</b>	Synthesized
<b>Target Device:</b>	xc7z010-3dgg400	• <b>Errors:</b>	No Errors
<b>Product Version:</b>	ISE 14.7	• <b>Warnings:</b>	No Warnings
<b>Design Goal:</b>	Balanced	• <b>Routing Results:</b>	
<b>Design Strategy:</b>	<a href="#">Xilinx Default (unlocked)</a>	• <b>Timing Constraints:</b>	
<b>Environment:</b>	<a href="#">System Settings</a>	• <b>Final Timing Score:</b>	

Device Utilization Summary (estimated values)				<a href="#">[-]</a>
Logic Utilization	Used	Available	Utilization	
Number of Slice LUTs	10	17600	0%	
Number of fully used LUT-FF pairs	0	10	0%	
Number of bonded IOBs	27	100	27%	

Detailed Reports						<a href="#">[-]</a>
Report Name	Status	Generated	Errors	Warnings	Infos	
<a href="#">Synthesis Report</a>	Current	sab 23. gen 17:52:21 2016	0	0	0	
Translation Report						

Figure 9: Sintesi su Xilinx Zync

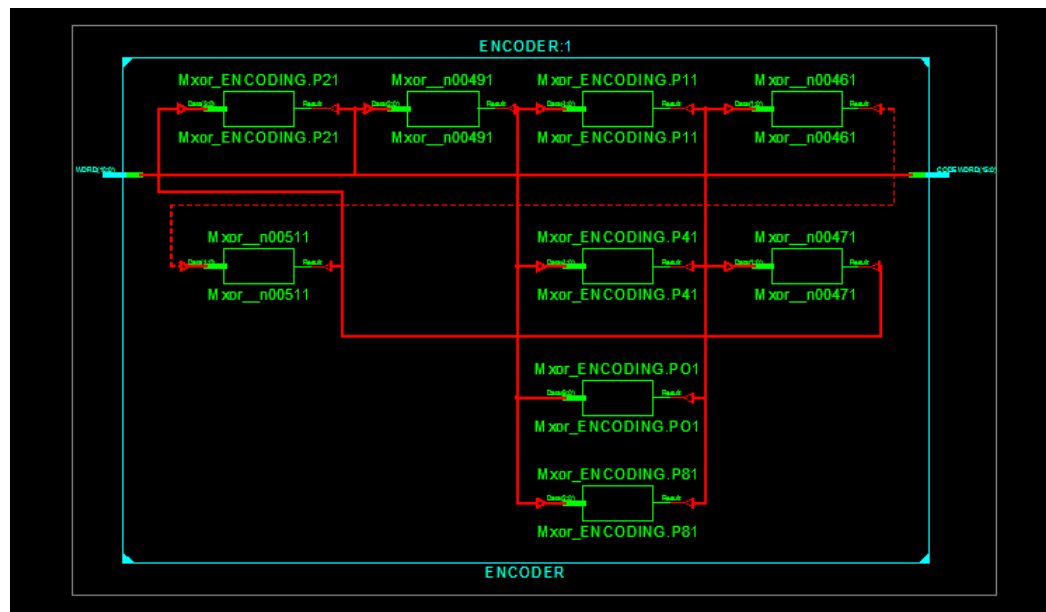


Figure 10: **Rappresentazione schematica del codice sorgente sintetizzato**

Presentiamo ora un summary circa la device utilization in termini di slices utilizzati:

```
Selected Device : 7z010c1g400-3

Slice Logic Utilization:
Number of Slice LUTs:           10 out of 17600    0%
Number used as Logic:           10 out of 17600    0%

Slice Logic Distribution:
Number of LUT Flip Flop pairs used: 10
Number with an unused Flip Flop: 10 out of 10    100%
Number with an unused LUT: 0 out of 10    0%
Number of fully used LUT-FF pairs: 0 out of 10    0%
Number of unique control sets: 0

IO Utilization:
Number of IOs: 27
Number of bonded IOBs: 27 out of 100    27%
```

Figure 11: **Numero di slices utilizzati dall'encoder**

Presentiamo infine un report relativo al timing:

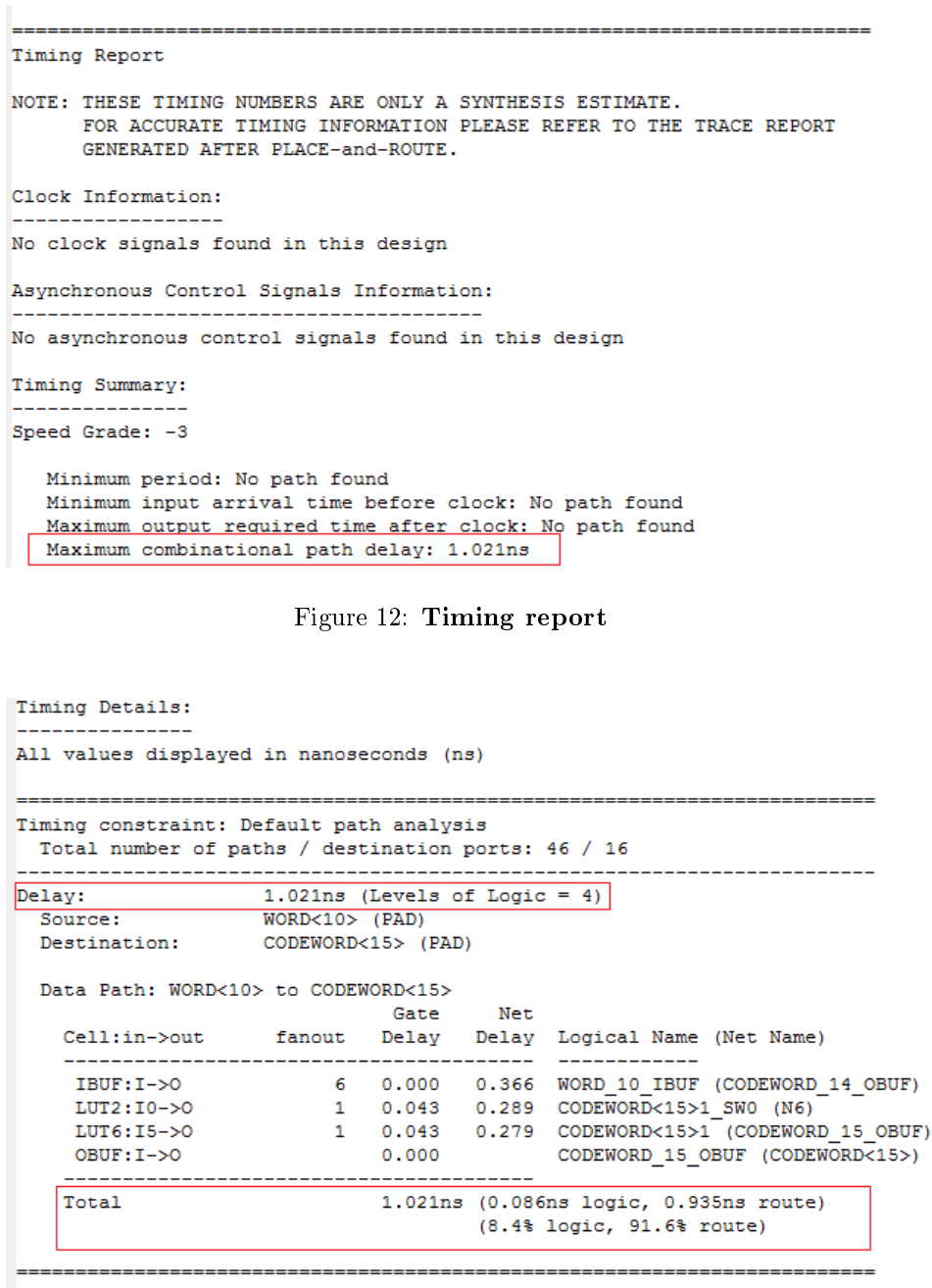
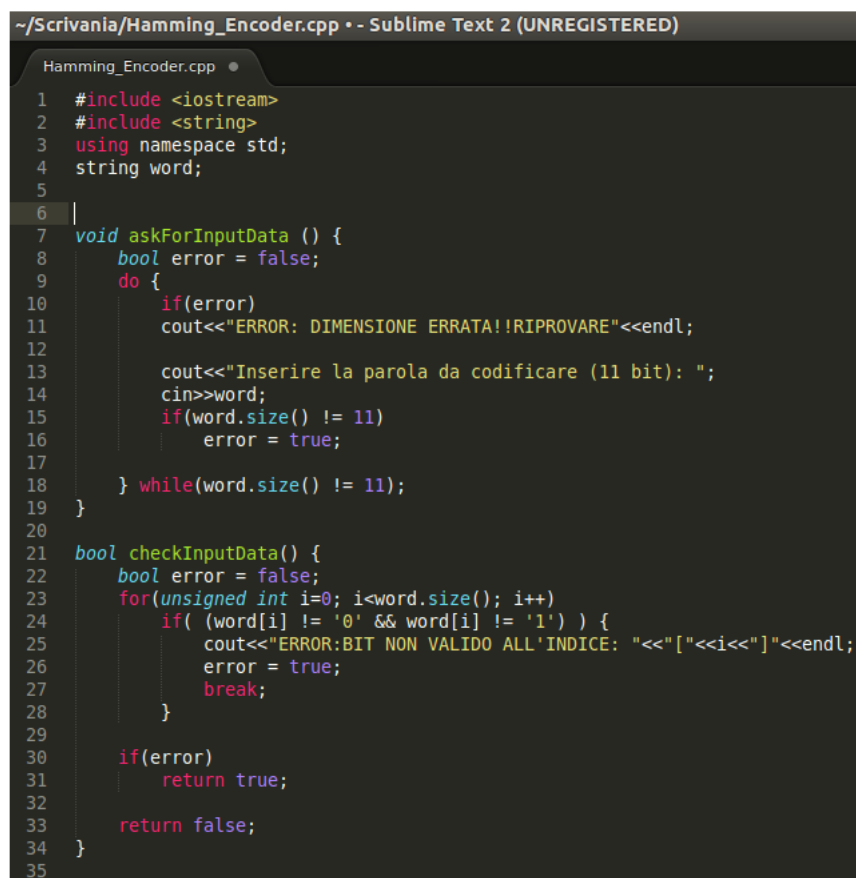


Figure 13: Maximum combination path delay: *1.021ns*

## 1.7 Codificatore di Hamming in C++

Qui di seguito, per concludere, presentiamo un programma in *C++* che calcola la codifica di Hamming prendendo in ingresso una word su 11 bit. Prima di determinare la codeword in uscita, secondo l'algoritmo presentato al paragrafo 1.3, l'input viene sottoposto a due test:

1. Verifica della lunghezza della word immessa (su 11 bit).
2. Controllo sui singoli bit immessi.



```
~/Scrivania/Hamming_Encoder.cpp - Sublime Text 2 (UNREGISTERED)
Hamming_Encoder.cpp
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  string word;
5
6
7  void askForInputData () {
8      bool error = false;
9      do {
10         if(error)
11             cout<<"ERROR: DIMENSIONE ERRATA!!RIPROVARE"<<endl;
12
13         cout<<"Inserire la parola da codificare (11 bit): ";
14         cin>>word;
15         if(word.size() != 11)
16             error = true;
17     } while(word.size() != 11);
18 }
19
20
21 bool checkInputData() {
22     bool error = false;
23     for(unsigned int i=0; i<word.size(); i++)
24         if( (word[i] != '0' && word[i] != '1') ) {
25             cout<<"ERROR:BIT NON VALIDO ALL'INDICE: "<<"["<<i<<"<<endl;
26             error = true;
27             break;
28         }
29
30     if(error)
31         return true;
32
33     return false;
34 }
35
```

Figure 14: *C++* source code (part 1)

```

Hamming_Encoder.cpp
36 int main() {
37
38     cout<<"\t\t\t\t=====HAMMING ENCODER===="<<endl<<endl;
39
40     askForInputData();
41
42     while(checkInputData())
43         askForInputData();
44
45     //Determining parity bit
46     unsigned int P1,P2,P4,P8,P0;
47
48     P1 = word[0] ^ word[2] ^ word[4] ^ word[6] ^ word[7] ^ word[9] ^ word[10];
49     P2 = word[0] ^ word[1] ^ word[4] ^ word[5] ^ word[7] ^ word[8] ^ word[10];
50     P4 = word[0] ^ word[1] ^ word[2] ^ word[3] ^ word[7] ^ word[8] ^ word[9];
51     P8 = word[0] ^ word[1] ^ word[2] ^ word[3] ^ word[4] ^ word[5] ^ word[6];
52
53     P0 = P1 ^ P2 ^ word[0] ^ P4 ^ word[1] ^ word[2] ^ word[3] ^ P8 ^ word[4] ^ word[5] ^ word[6] ^ word[7] ^ word[8] ^ word[9] ^ word[10];
54
55     //Determining CODEWORD
56     char codeword[17] = { (char)P0, word[0], word[1], word[2], word[3], word[4], word[5], word[6], (char)P8, word[7],
57                          word[8], word[9], (char)P4, word[10], (char)P2, (char)P1, '\0'};
58
59     cout<<"CODEWORD DA TRASMETTERE: "<<codeword<<endl;
60
61     return 0;
62 }

```

Figure 15: *C++* source code (part 2)

Di seguito presentiamo una possibile esecuzione:

```

mariano@mariano-Aspire-5742G: ~/Scrivania
mariano@mariano-Aspire-5742G:~/Scrivania$ g++ -Wall -o Hamming Hamming_Encoder.c
mariano@mariano-Aspire-5742G:~/Scrivania$ ./Hamming
=====HAMMING ENCODER=====

Inserire la parola da codificare (11 bit): 00000000001
CODEWORD DA TRASMETTERE: 100000000000111
mariano@mariano-Aspire-5742G:~/Scrivania$

```

Figure 16: Programma in *C++* che calcola la codifica di Hamming