Scuola universitaria professionale
della Svizzera italiana

**Dipartimento
Tecnologie
Innovative**

Progetto di semestre invernale

IN-05/06-SI-11

EDAC
ERROR DETECTION AND CORRECTION

Studenti: Davide Dellagana
Patrick Di Domenico
Relatore: Paolo Ceppi
Correlatore: Ivan Defilippis

10 marzo 2006

Ringraziamo i collaboratori del SSL che che ci hanno aiutato nella realizzazione di questo progetto.

Indice

1 Riassunto / Abstract **8**

2 Descrizione **9**
2.1 Informazioni tecniche 9
2.1.1 Obiettivi da raggiungere 9
2.1.2 Compiti da eseguire 9
2.1.3 Tecnologie da utilizzare 10

3 Introduzione **11**

4 Requisiti e specifiche **12**

5 Studio delle soluzioni **13**
5.1 Metodi analizzati 13
5.1.1 Hamming 13
5.1.2 Binary Golay codes 14
5.1.3 Convolutional code 16
5.1.4 Reed-Solomon codes 16
5.2 Conclusioni 17
5.3 Applicazioni implementate 18

6 Design / concezione **19**
6.1 Implementazione in VHDL 19
6.1.1 Hamming 19
6.1.2 CRC 23
6.1.3 EDAC 27
6.2 Sintesi 27
6.2.1 Hamming 28
6.2.2 CRC 36
6.3 Dimostratore in Java 38
6.3.1 Hamming 38
6.3.2 CRC 47

7 Test **53**
7.1 Implementazione in VHDL . . . 53
7.1.1 Hamming . . . 53
7.1.2 CRC . . . 54
7.1.3 Test package . . . 63
7.2 Dimostratore in Java . . . 63

8 Conclusioni **64**

A Introduzione ai codici per la correzione degli errori **66**
A.1 Introduzione generale . . . 66
A.2 Errori nel campo delle trasmissioni . . . 67

B Aritmetica modulare e campi finiti **69**
B.1 Aritmetica modulare . . . 69
B.1.1 Definizione di corpo (copiata da wikipedia) . . . 71
B.2 Campi di Galois (Galois Fields) . . . 72

C Codici sorgente in VHDL e generatore test in C **75**
C.1 Hamming . . . 75
C.1.1 Codificatore . . . 75
C.1.2 Decodificatore . . . 78
C.1.3 Correttore . . . 81
C.1.4 Test . . . 83
C.2 CRC . . . 95
C.2.1 Codificatore . . . 95
C.2.2 Decodificatore . . . 97
C.2.3 Correttore . . . 99
C.2.4 Test . . . 203
C.3 Edac . . . 224
C.3.1 Edac package . . . 224
C.3.2 Edac . . . 226

D Codici sorgente Java (dimostratore) **250**
D.1 Metodi . . . 250
D.1.1 Hamming . . . 250
D.1.2 CRC . . . 288
D.2 Generatore case per correzione CRC . . . 487
D.2.1 Classe divisione polinomiale . . . 487
D.2.2 Generatore case per correzione CRC . . . 490
D.2.3 Test divisione polinomiale . . . 500
D.2.4 Test generatore case per correzione CRC . . . 504
D.3 Convertitore . . . 506
D.4 EDAC applet . . . 513

E Codici sorgente esistenti **517**
E.1 Golay(23,12,7) . . . 517
E.2 Golay(23,12,7) con routine per stampa tabella decodifica . . . 517
E.3 Golay(24,12,8) . . . 517
E.4 Reed-Solomon . . . 517

F Allegati applicazioni EDAC esistenti **518**
F.1 Allegato 1 . . . 518
F.2 Allegato 2 . . . 518
F.3 Allegato 3 . . . 518
F.4 Allegato 4 . . . 518
F.5 Allegato 5 . . . 519
F.6 Alleganto 6 . . . 519
F.7 Allegato 7 . . . 519

Elenco delle figure

5.1 Schema Golay(23,12,7) . . . 15
5.2 Diagramma di flusso Reed-Solomon . . . 17

6.1 Schema generale Hamming . . . 20
6.2 Schema codifica Hamming . . . 20
6.3 Schema decodifica Hamming . . . 21
6.4 Schema correzione Hamming . . . 22
6.5 Schema codifica CRC . . . 24
6.6 Schema decodifica CRC . . . 25
6.7 Schema correzione CRC . . . 26
6.8 Schema generale EDAC . . . 27
6.9 Schema generale Hamming (RTL netlist) . . . 28
6.10 Codificatore Hamming (RTL netlist) . . . 29
6.11 Decodificatore Hamming (RTL netlist) . . . 30
6.12 Correttore Hamming (RTL netlist) . . . 31
6.13 Schema generale Hamming (Gate netlist) . . . 32
6.14 Codificatore Hamming (Gate netlist) . . . 33
6.15 Decodificatore Hamming (Gate netlist) . . . 34
6.16 Correttore Hamming (Gate netlist) . . . 35
6.17 Schema generale CRC (RTL netlist) . . . 36
6.18 Correttore CRC (RTL netlist) . . . 36
6.19 Schema generale CRC (Gate netlist) . . . 37
6.20 Implementazione codifica Hamming nel dimostratore . . . 41
6.21 Implementazione inserimento errrori dimostratore Hamming . . . 43
6.22 Implementazione decodifica Hamming nel dimostratore . . . 45
6.23 Impl. ricon. info. orig. in Hamming nel dimostratore . . . 46
6.24 Implementazione codifica CRC nel dimostratore . . . 48
6.25 Implementazione inserimento errori CRC nel dimostratore . . . 50
6.26 Implementazione decodifica CRC nel dimostratore . . . 51
6.27 Impl. ricon. info. orig. CRC nel dimostratore . . . 52

7.1 Diagramma di flusso test bench Hamming . . . 54
7.2 Diagramma di flusso generazione case per corr. CRC parte 1 . . . 56
7.3 Diagramma di flusso generazione case per corr. CRC parte 2 . . . 57

7.4 Diagramma di flusso generazione case per corr. CRC parte 3 . 58
7.5 Diagramma di flusso test CRC con 0 errori 59
7.6 Diagramma di flusso test CRC con 1 errore 60
7.7 Diagramma di flusso test CRC con 2 errori 61
7.8 Diagramma di flusso test CRC con 3 errori 62

Elenco delle tabelle

5.1 Tavola posizionamento bit Hamming 14

7.1 Struttura file pattern per test codice di Hamming 53

B.1 Tavole somma e moltiplicazione *modulo 6* 70
B.2 Tavola moltiplicazione *modulo 6* con *MCD* 70
B.3 Tavola moltiplicazione *modulo 7* con *MCD* 71
B.4 Tavola somma $GF(2^2)$ 73
B.5 Tavola moltiplicazione $GF(2^2)$ 73

Capitolo 1

Riassunto / Abstract

Questo progetto di semestre, tratta il tema della correzione degli errori nella memorizzazione dei dati in un calcolatore di un satellite. La prima parte di questo progetto, tratta l'implementazione di due metodi per la correzione degli errori (Hamming e CRC) nel linguaggio di descrizione hardware VHDL; la seconda parte tratta l'implementazione di un dimostratore, utilizzabile su un PC per mostrare il funzionamento di vari metodi per la correzione degli errori. All'interno di questa documentazione si trova anche del materiale supplementare: analisi di altri metodi, teoria sull'aritmetica modulare, teoria sui campi di Galois, ...

In this project we work to implement a error's detection and correction's system for a satellite's computer. This project is divided in two differents parts, the first part implement two errors correcting's codes (Hamming and CRC) with the VHDL language; the second part implement a Java dimostrator that (for example) a professor can use for teaching some errors correcting's codes. This paper documentation contain also various material about others error correcting codes, the theory about modular's arithmetic, the theory about Galois Field,....

Capitolo 2

Descrizione

La detezione e correzione automatica di errori sui dati di memoria di calcolatori di bordo è vista come un aspetto importante per le prossime missioni spaziali di Almasat (http://www.almasat.org).

SUPSI Space Lab e Almasat intendono collaborare anche in questo settore.

Più in generale, le tecnologie di detezione e correzione degli errori trovano molte applicazioni nei calcolatori e nei dischi, si pensi ai sistemi RAID.

2.1 Informazioni tecniche

2.1.1 Obiettivi da raggiungere

Un algoritmo (Coder-Decoder e correttore) implementato, simulato, sintetizzato e verificato.

2.1.2 Compiti da eseguire

Raccogliere e analizzare documentazione sugli algoritmi di codifica per detezione e correzione di errori su dati binari (v. corsi di matematica discreta, corsi di telematica, tecnica digitale, ecc.) in prospettiva di una codifica per sintesi in hardware. Inizialmente Almasat punta ad un'implementazione dell'algoritmo su FPGA.

Ricerca documentazione di applicazioni e implementazioni in ambito spaziale (p.e. http://www.smallsat.org/sessions/abstracts/session6-abstracts).

Codifica di un algoritmo completo e relativo testbench per validare il funzionamento del sistema in diverse situazioni.

Durante la fase di analisi del problema sarà molto probabilmente necessario

visitare il gruppo Almasat a Forlì per conoscere meglio l'ambito applicativo e chiarire i dettagli dei requisiti.

2.1.3 Tecnologie da utilizzare

- UP1/UP2
- Simili/Aldec
- Symplify
- Teorie della codifica
- Errori

Capitolo 3

Introduzione

Questo progetto di semestre, tratta il tema della detezione e correzione degli errori nel calcolatore di bordo di un satellite.
Il nostro lavoro si è svolto in due tappe:

- nella prima abbiamo analizzato diverso materiale, al fine di trovare dei metodi adatti al nostro scopo;
- nella seconda abbiamo implementato due metodi, Hamming e CRC.

L'implementazione l'abbiamo fatta:

- in VHDL, per potere sintettizzarli su di un FPGA, come richiesto dalle specifiche (a pagina 9);
- in Java (sotto forma di applet) per realizzare un dimostratore di vari metodi per la correzione degli errori.

Capitolo 4

Requisiti e specifiche

Le specifiche ed i requisiti di questo progetto, sono contenuti nel capitolo 2 a pagina 9. Nei vari colloqui avuti con il docente responsabile durante lo svolgimento del lavoro, abbiamo approfondito le specifiche richieste.
I seguenti punti, riassumono le varie specifiche approfondite:

- implementazione (in ogni caso), del metodo di Hamming;
- analisi dei metodi: Golay (23,12,7), Golay (24,12,8), Convolutional Code, Reed-Solomon;
- studio delle possibili varianti di implementazione del dimostratore (applicazione o applet, scrittura risultati su file, ...).

Capitolo 5

Studio delle soluzioni

Prima di iniziare l'implementazione in VHDL ed in Java, abbiamo analizzato diversi metodi e, dopo una valutazione abbiamo selezionato: il metodo di *Hamming* ed il *CRC*.
In questo capitolo presentiamo i metodi analizzati e le applicazioni già sviluppate in questo campo.

5.1 Metodi analizzati

5.1.1 Hamming

All'informazione originale, aggiunge dei bit di parità. Questi bit vengono aggiunti in posizioni ben determinate all'interno della codeword. Oltre a questi appena calcolati, viene aggiunto un'ulteriore bit, che controlla la parità della codeword ottenuta, permettendo così in fase di decodifica di determinare se ci sono 2 errori all'interno della codeword.
Il codice di Hamming si costruisce nel seguente modo:

1. si determina la quantità dei bit di controllo con la seguente formula $n \leq 2^{n-k} + 1$ (la simbologia è quella definita nell'appendice A a pagina 66);

2. si inseriscono i bit di controllo. Il bit di controllo c_i, viene inserito nella posizione 2^{i-1} e controlla a partire dalla posizione 2^{i-1} compresa, gruppi di 2^{i-1} bit, presi alternativamente in successione nella parola del codice.

La seguente tabella mostra:

- come posizionare i bit del messaggio e quelli di parità nella codeword;

- di che bits controlla la parità ogni bit di controllo.

	k_3	k_2	k_1	c_3	k_0	c_2	c_1
c_1	x		x		x		
c_2	x	x			x		
c_3	x	x	x				

Tabella 5.1: Tavola posizionamento bit Hamming

Nella fase di decodifica, si ricalcolano i bit di parità e si fa un confronto con quelli della codeword. Si ottiene così la posizione del bit errato, che si può quindi correggere. Utilizzando anche l'ottavo bit, si può determinare se la codword contiene 2 errori (questo procedimento è spiegato nella sezione 6.1.1 a pagina 21).

5.1.2 Binary Golay codes

Esistono due codici binari di golay:

codice binario di Golay esteso: questo codice codifica delle parole binarie di 12 bit e ne restituisce delle parole da 24 bit. Se in una parola ci sono 3 errori il codice è in grado di correggerli, se gli errori sono 4 il codice è in grado solamente di intercettarli.

codice binario di Golay perfetto: le parole vengono codificate con 23 bit, viene creato a partire dal codice di Golay esteso cancellandone un bit. Al contrario se si vuole ottenere il codice esteso, bisogna aggiungere un bit di parità.

E' possibile codificare un byte usando la modulazione 8 a 12 (es. per comunicare su di un bus). 8 bit si condividono la larghezza di banda con 4 bit di telemetria.
Questo metodo è usato nel campo delle telecomunicazioni (es. trasferimento dati da satellite, trasmissioni ad alta frequenza, ...).
In un primo momento, abbiamo studiato il codice di Golay (23,12,7). Non l'abbiamo implementato, perchè non siamo stati in grado di comprendere completamente il funzionamento.
Lo studio di questo codice, l'abbiamo fatto aiutandoci con il codice sorgente *golay23.c* scritto da Robert Morelos-Zaragoza (si trova come allegato alla sezione E.1 a pagina 517).
Il seguente schema rappresenta come viene implementato il codice nel suddetto programma.

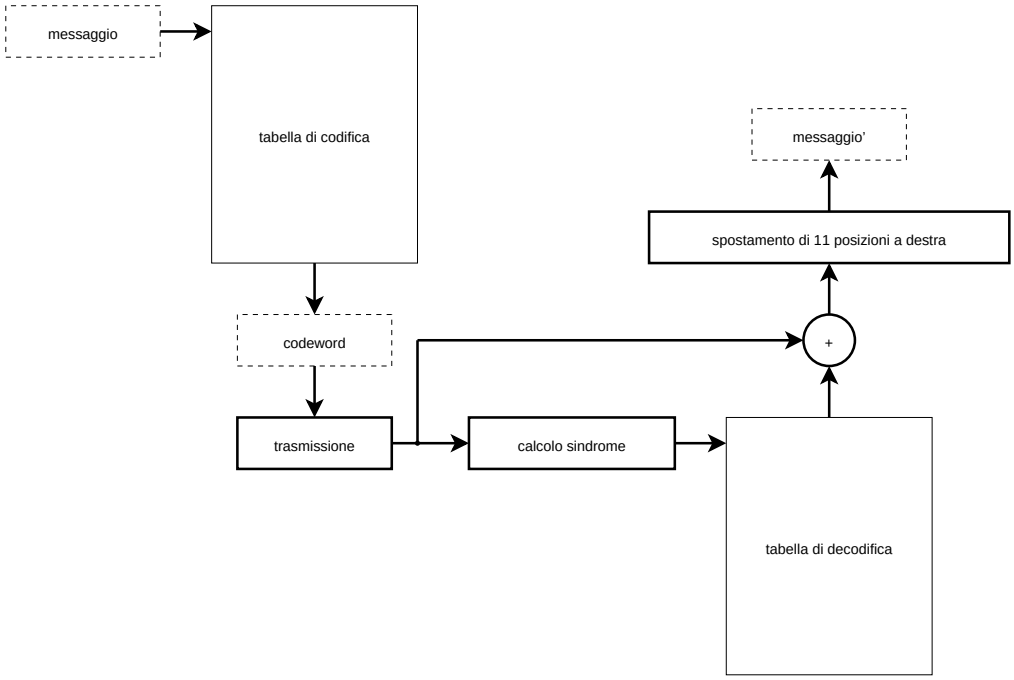

Figura 5.1: Schema Golay(23,12,7)

La tabella di codifica viene costruita prendendo tutti i possibili messaggi, moltiplicandoli per 2^{11} e calcolandone la sindrome del risultato. Questa tabella contiene 4096 elementi.
La costruzione della tabella di decodifica, non è molto chiara, infatti non ce l'abbiamo fatta a comprenderne il procedimento. Una cosa che siamo riusciti a capire, è che la costruzione è suddivisa in 3 parti: la prima per le codeword contenenti 1 errore, la seconda parte per le codeword contenenti 2 errori e la terza per le codeword contenenti 3 errori.
L'altro aspetto poco chiaro ma centrale di questo programma è il modo ci calcolare la sindrome. Questo è la parte di codice in C del suddetto programma, usata per calcolare la sindrome.

```
#define X22    0x00400000
#define X11    0x00000800
#define MASK12 0xfffff800
#define GENPOL 0x00000c75

long get_syndrome(long pattern)
{
  long aux = X22, aux2;

  if (pattern >= X11)
    while (pattern & MASK12)
```

```
    {
      while (!(aux & pattern))
        aux = aux >> 1;
      pattern ^= (aux/X11) * GENPOL;
    }
  return (pattern);
}
```

All'inizio pensavamo che questa funzione facesse la divisione polinomiale del pattern per il GENPOL, ma facendo dei confronti con test fatti a mano, ci siamo accorti che non è così.
Abbiamo quindi implementato il metodo CRC, con lo stesso polinomio generatore.
Il codice di Golay (24,12,8) (che si trova come allegato alla sezione E.3 a pagina 517), non usa la funzione get_syndrome come il (23,12,7), ma lavora con delle matrici.

5.1.3 Convolutional code

Questo metodo viene usato prevalentemente nel campo delle telecomunicazioni (i simboli usati in questa sottosezione non sono quelli spiegati nell'appendice A a pagina 66).

- Ogni stringa di m bit in entrata viene trasformata in un simbolo di n bit. Il rapporto del codice viene calcolato come m/n ($n \geq m$).
- La trasformazione è una funzione dell'ultimo simbolo k, dove k è la lunghezza limite del codice. Questa trasformazione può essere eseguita con una FSM.

Il vantaggio nell'uso di questo codice sta nella decodifica. Esistono diversi algoritmi per decodificare questo codice. Per piccoli valori di k viene utilizzato frequentemente l'algoritmo *Viterbi* che fornisce quasi sempre la massima performance ed è altamente parallelizzabile.

5.1.4 Reed-Solomon codes

La correzione degli errori con il metodo Reed-Solomon è uno schema di codifica che crea dapprima una forma polinomiale dei simboli (dati da codificare) e poi manda una traccia sovracampionata del polinomio invece dei simboli stessi. Il processo appena descritto viene fatto perchè grazie all'informazione ridondante contenuta nei dati sovracampionati è possibile ricostruire il polinomio originale e così i dati fino ad un certo numero di errori. Questo metodo lavora su dei blocchi di dati.
Le proprietà dei codici Reed-Solomon li rendono particolarmente utili per

delle applicazioni in cui gli errori "si producono" a raffica.
Questi codici vengono usati in molte applicazioni per il salvataggio dei dati e non (es. CD, DAT, DVD, wireless, satellite comunication, digital television DVB, high speed modem ADSL, ...).
Il seguente diagramma di flusso mostra come funziona l'algoritmo che implementa il codice *Reed-Solomon*. Il codice sorgente *rs.c*, su di sui ci siamo basati per disegnare questo diagramma di flusso, è stato scritto da Simon Rockliff (si trova come allegato alla sezione E.4 a pagina 517.

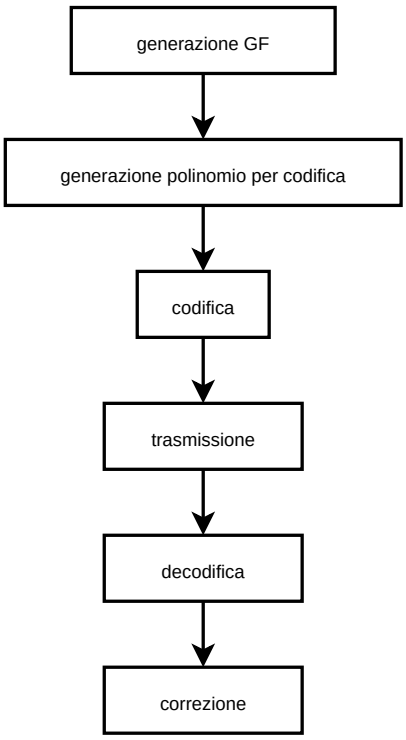

Figura 5.2: Diagramma di flusso Reed-Solomon

5.2 Conclusioni

Da quest'analisi abbiamo tratto le seguenti conclusioni:

- il codice di Hamming, lo implementiamo, perchè è il più semplice. Nella versione "base" non è molto efficiente, ma con certi accorgimenti lo diventa;
- i codici di Golay, in particolare il (23,12,7), ci sembrava fattibile da implementare. Purtroppo nel corso dell'analisi, abbiamo riscontrato

dei problemi nel calcolo della sindrome e nella decodifica. Per questi motivi, l'abbiamo scartato;

- il codice convoluzionale, non abbiamo potuto analizzarlo, perchè abbiamo riscontrato problemi nella ricerca del materiale;
- il codice Reed-Solomon, l'abbiamo scelto, perchè viene già usato in applicazioni simili alla nostra. Un'altro vantaggio è la sua flessibile capacità di correzione (possibilità di scegliere quanti errori correggibili). Purtroppo dalla nostra analisi, la sua implementazione, è risultata troppo complicata.

5.3 Applicazioni implementate

Dalle ricerche effettuate, abbiamo trovato le seguenti implementazioni simili alla nostra. I documenti sono allegati all'appendice F a pagina 518.

1. contiene una descrizione di un sistema EDAC implementato alla Pennsylvania State University;
2. contiene un sistema implementato da Amsat, su di un satellite;

3-7. contengono informazioni (non molto approfondite) di varie applicazioni.

Capitolo 6

Design / concezione

6.1 Implementazione in VHDL

6.1.1 Hamming

Per quest'implementazione, utilizziamo il codice di Hamming $(7, 4, 3)$. Il numero massimo di errori detettabili da questo codice è:

$$e_d = 3 - 1 = 2$$

Per la detezione del secondo errore, bisogna aggiungere un bit di parità alla codeword. Il codice diventa quindi Hamming $(8, 4, 3)$. Il numero di errori correggibili è:

$$e_c = \frac{3-1}{2} = 1$$

L'algoritmo di codifica del metodo di Hamming, calcola a partire dai 4 bit del messaggio originale 3 bit di parità. Questi bit vengono aggiunti al messaggio originale formando cosi le codeword. A questo punto viene calcolato il valore del bit di parità che viene aggiunto alla codeword per la detezione del secondo errore in fase di decodifica. La codeword viene salvata in memoria. Al momento della lettura, l'algoritmo di decodifica è in grado di determinare, dall'informazione fornita dalla codeword: se essa non contiene errori, se ne contiene 1 o se ne contiene 2. Nel caso che contiene più di 2 errori, quest'algoritmo non è in grado di intercettarli. Nel caso che contiene 1 errore, esso viene corretto. Il fatto che in presenza di 2 errori, il decodificatore lo segnala, è utile perchè quest'informazione può venir utilizzata ad esempio per fare un reset del sistema.
In questo progetto, abbiamo deciso di lavorare parità pari.

Il seguente schema, mostra come abbiamo implementato questo metodo.

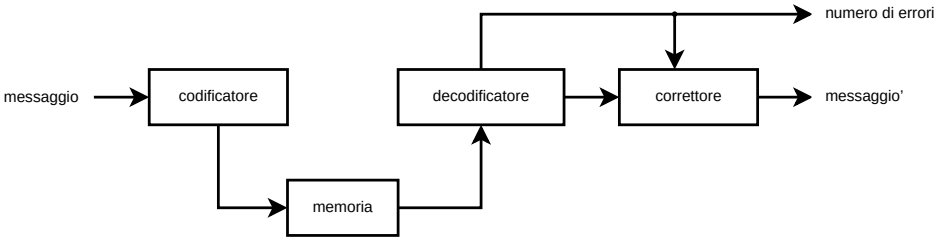

Figura 6.1: Schema generale Hamming

Codifica

La codifica è illustrata nel seguente schema.

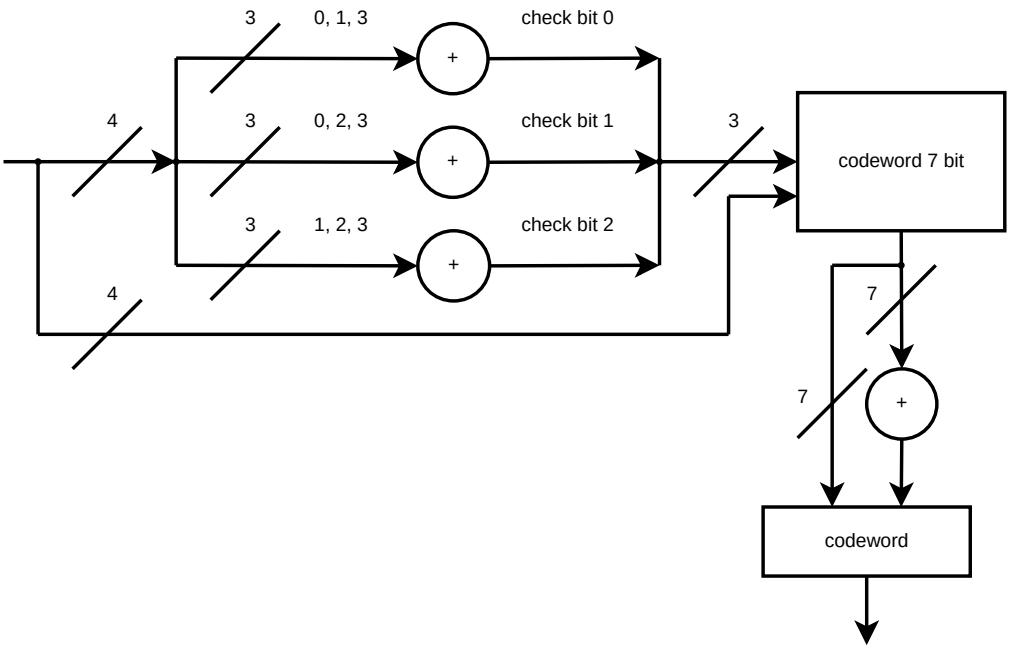

Figura 6.2: Schema codifica Hamming

Come si può vedere il calcolo dei 3 bit di parità dei gruppi e del bit di parità della codeword di 7 bit, viene fatto semplicemente con delle porte xor. Il posizionamento dei bit di parità e quelli del messaggio originale all'interno della codeword, viene spiegato nella sezione 5.1.1 a pagina 14.

Decodifica

La decodifica è illustrata nel seguente schema.

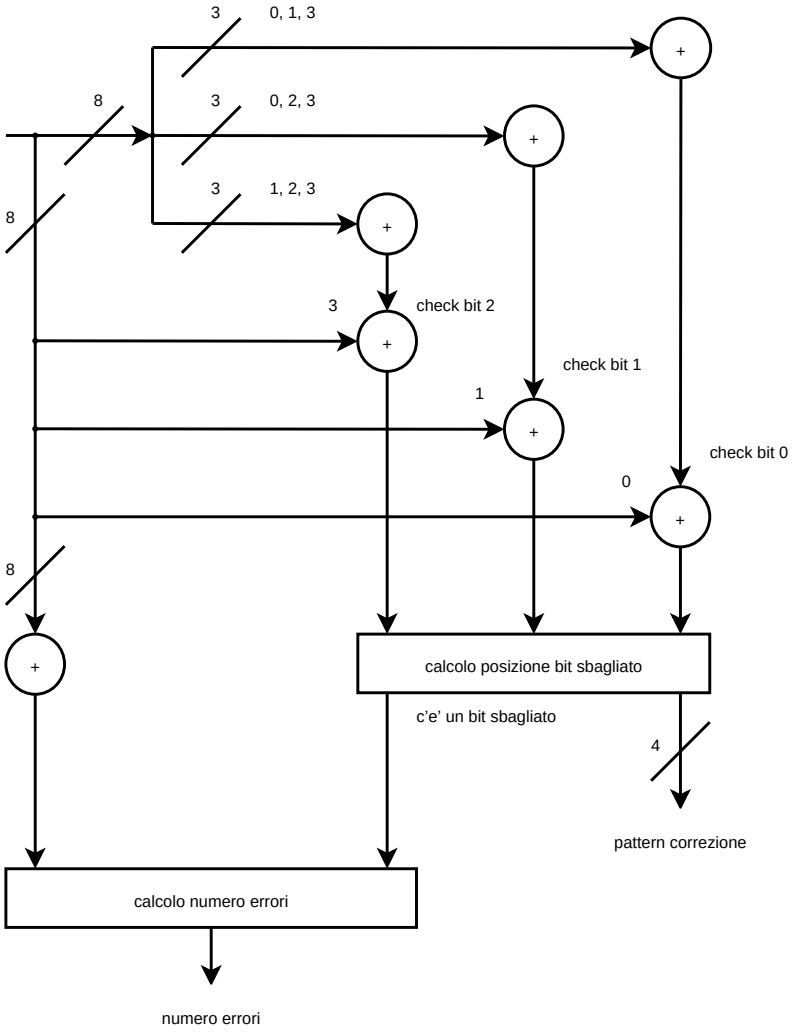

Figura 6.3: Schema decodifica Hamming

La decodifica consiste nel ricalcolare i bit di parità dei gruppi e quello globale e di determinare in seguito il numero di errori e la posizione dell'errore. I tre check bit calcolati al momento della decodifica, in ordine crescente, rappresentano il numero in formato binario della posizione dell'errore.
La determinazione del numero di errori presenti nella parola, avviene nel seguente modo.

0 errori: se il calcolo della posizione del bit sbagliato ed il bit di parità, determinano che non ci sono errori;

1 errore: se il calcolo della posizione del bit sbagliato determina che c'è un'errore e anche il bit di parità lo conferma oppure se il calcolo della posizione del bit sbagliato determina che ci sono errori e invece il bit di parità determina che non ce ne sono (errore nel bit di parità);

2 errori: se il calcolo della posizione del bit sbagliato determina che non ci sono errori, ma il bit di parità determina che ce ne sono.

Nel calcolo del numero di errori, vengono presi in considerazione anche quelli nei 4 bit di parità. Nel caso che il decodificatore comunica al correttore che ha trovato 1 errore, quest'ultimo deve controllare che il pattern per la correzione degli errori sia diverso da 0, altrimenti l'errore si trova nel bit di parità (caso da non considerare).

Correzione

La correzione è illustrata nel seguente schema.

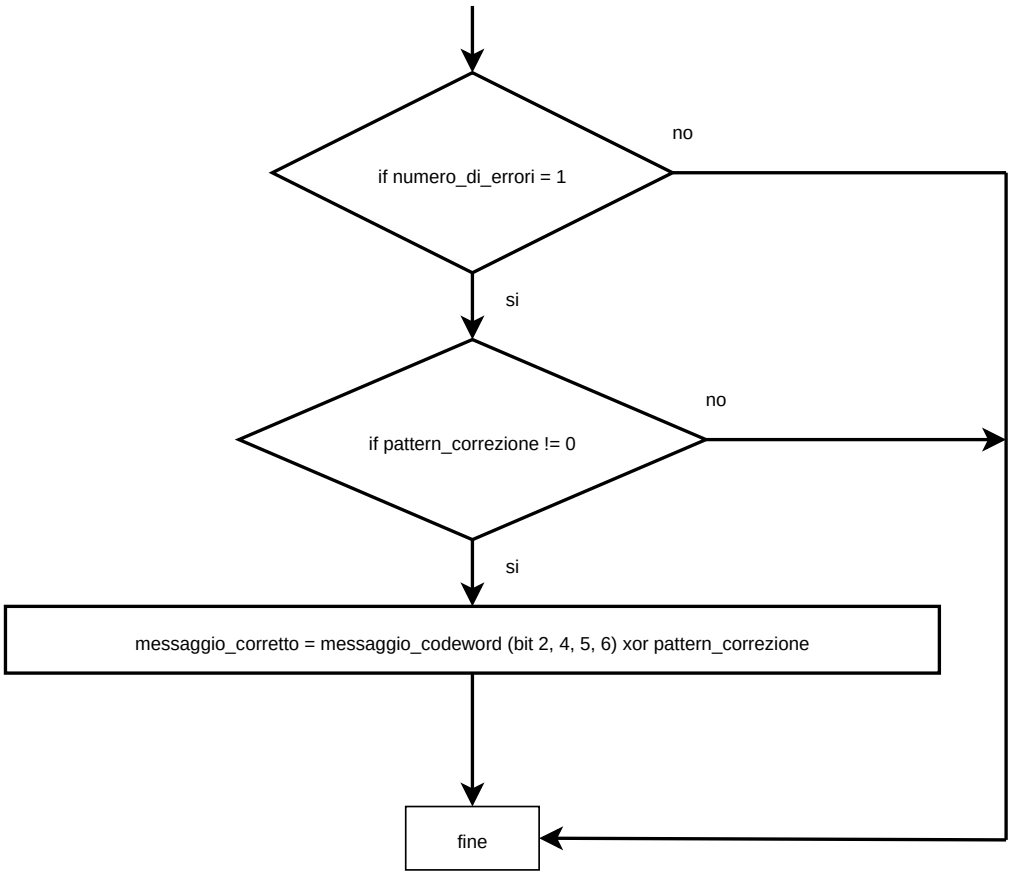

Figura 6.4: Schema correzione Hamming

6.1.2 CRC

La nostra implementazione del CRC in VHDL, usa il seguente polinomio generatore.

$$x^{11} + x^{10} + x^6 + x^5 + x^4 + x^2 + x^0 = 110001110101 = 0\text{xC75}$$

I messaggi sono lunghi 12 bit, le codeword saranno quindi composte da 23 bit. Usiamo la seguente funzione, per calcolare la divisione polinomiale.

Funzione per il calcolo della *divisione polinomiale*

Questa funzione è contenuta nel package *EdacDef*. Il procedimento che abbiamo implementato è il seguente:

- creiamo le seguenti variabili temporanee:
  - il vettore *TempDiv* di tipo std_logic_vector e di dimensione uguale a quella del divisore. Questo vettore conterrrà il risultato della divisione ad ogni passo, lo inizializziamo con i bit più significativi del dividendo;
  - la variabile *TempPos* di tipo integer. Questa variabile contiene la posizione del prossimo bit da trasferire dal dividendo nella posizione meno significativa del vettore TempDiv, e la inizializziamo;
- a questo punto discriminiamo i tre possibili casi:
  - la dimensione del dividendo è uguale alla dimensione del divisore. Controlliamo il valore del primo bit del dividendo: se è 0, non viene fatto niente, se è 1, nel vettore TempDiv viene messo il suo valore attuale xor il divisore;
  - la dimensione del dividendo è maggiore di quella del divisore. In questo caso ripetiamo le seguenti operazioni fino a che il valore della variabile TempPos è maggiore uguale a 0.
    - viene controllato il valore del bit più significativo del vettore TempDiv: se è uguale a 0, non viene fatto niente, se è uguale a 1, nel vettore viene messo il valore in esso contenuto xor il divisore;
    - viene spostato a sinistra di una posizione il contenuto del vettore TempDiv e nella posizione meno significativa viene messo il valore del bit del dividendo alla posizione TempPos;
    - viene decrementata la variabile TempPos;

    A questo punto viene controllato il valore del bit più significativo del vettore TempDiv: se è uguale a 1, nel vettore viene messo il valore in esso contenuto xor il divisore, se è uguale a 0, non viene fatto niente;

- la dimensione del dividendo è minore della dimensione del divisore. Questo caso non l'abbiamo previsto, quindi non facciamo nessuna operazione.

- la funzione restituisce l'informazione contenuta nei bit del vettore TempDiv tranne quello nella posizione più significativa.

Per limitare il numero di operazioni da effettuare, ogni volta che viene fatta l'operazione xor, non vengono presi in considerazione il bit più significativo del primo operando e quello più significativo del secondo, siccome l'informazione che verrebbe calcolata non serve.

Codifica

La codifica è illustrata nel seguente schema.

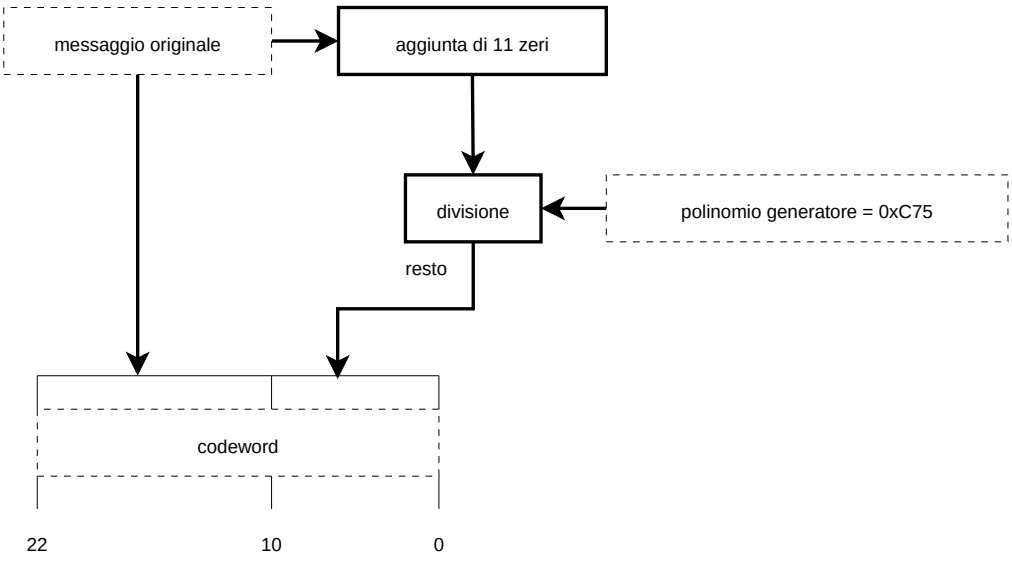

Figura 6.5: Schema codifica CRC

L'aggiunta di 11 zeri, corrispone alla moltiplicazione per 2^{11}.

Decodifica

La decodifica è illustrata nel seguente schema.

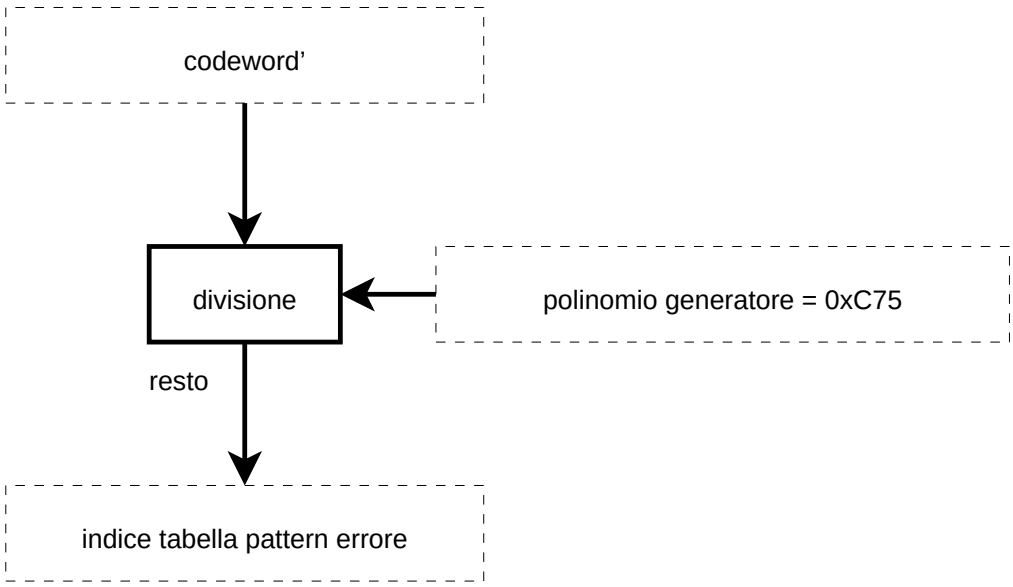

Figura 6.6: Schema decodifica CRC

La decodifica consiste nel calcolare il resto della divisione della codeword ricevuta per il polinomio generatore. Il resto della divisione, ci servirà nella fase di correzione per ottenere dalla tabella di decodifica il pattern d'errore. La costruzione di questa tabella è spiegata nella sottosezione 7.1.2 a pagina 54.

Correzione

La correzione è illustrata nel seguente schema.

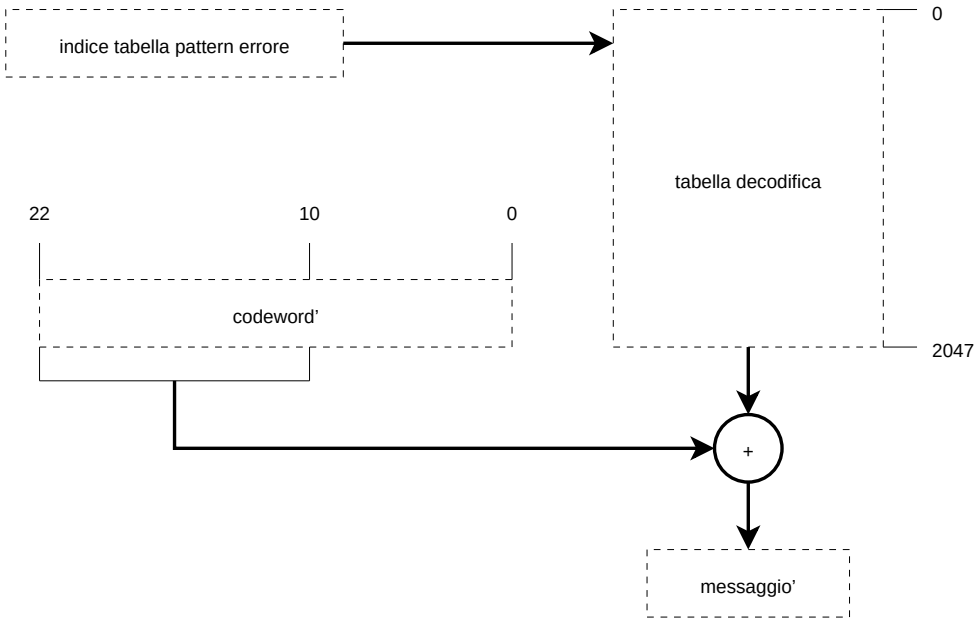

Figura 6.7: Schema correzione CRC

L'uso di questi due metodi all'interno del nostro EDAC, viene illustrato nella prossima sezione.

6.1.3 EDAC

L'implementazione del componente EDAC, non l'abbiamo portata a termine, perchè non conosciamo l'architettura dove verrà utilizzato e non abbiamo avuto sufficiente tempo.
Abbiamo iniziato a definire:

- l'entità con le principali "connessioni" con il mondo esterno (segnale di clock, bus in, bus out, operazione da effettuare, ...);
- l'architettura in cui dichiariamo i componenti del metodo di Hamming, alcuni segnali interni, il port mapping dei componenti ed il processo principale.

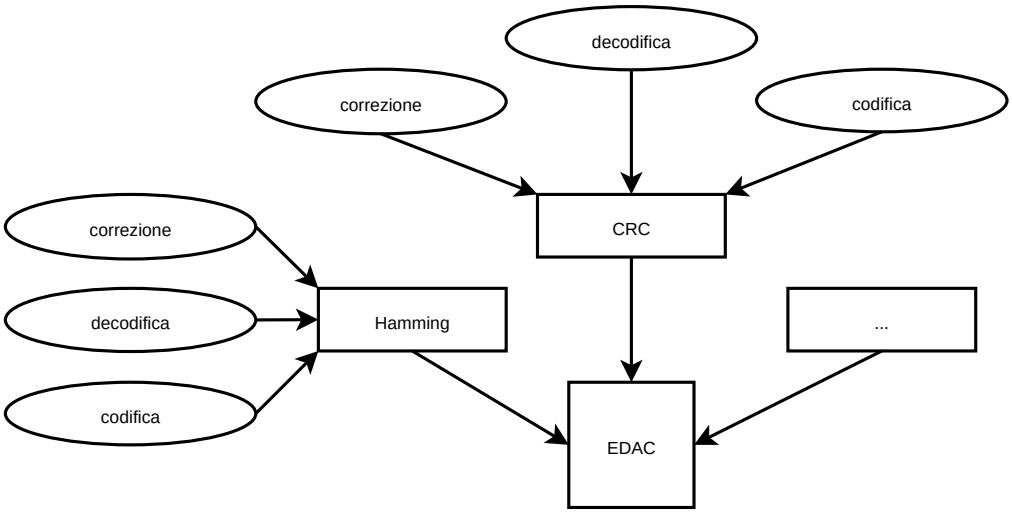

Figura 6.8: Schema generale EDAC

6.2 Sintesi

Per sintetizzare le implementazioni che abbiamo sviluppato in VHDL (HammingSyn.vhd e DivSyn.vhd), abbiamo utilizzato il tool *Symplify Pro 8.2.1* della *Symplicity*. Abbiamo utilizzato come componente di riferimento l'FPGA *FLEX EPF10K20RC240-4* della *Altera*.
In questa sezione, mostriamo alcuni schemi della sintesi, i file di report si trovano nel cd allegato a questa documentazione.

6.2.1 Hamming

RTL netlist

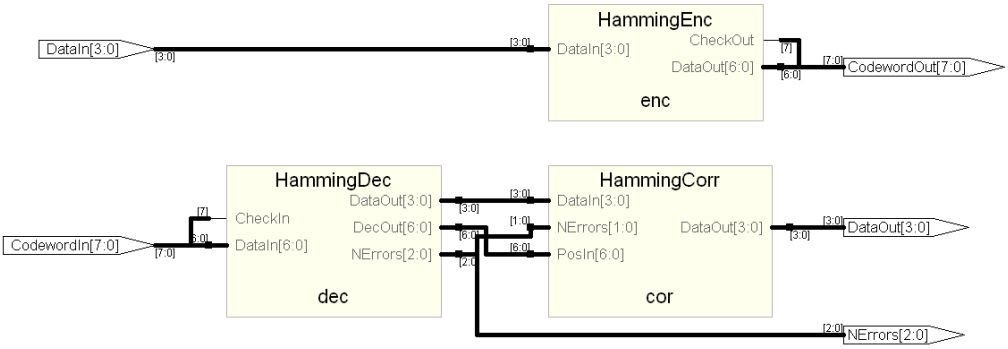

Figura 6.9: Schema generale Hamming (RTL netlist)

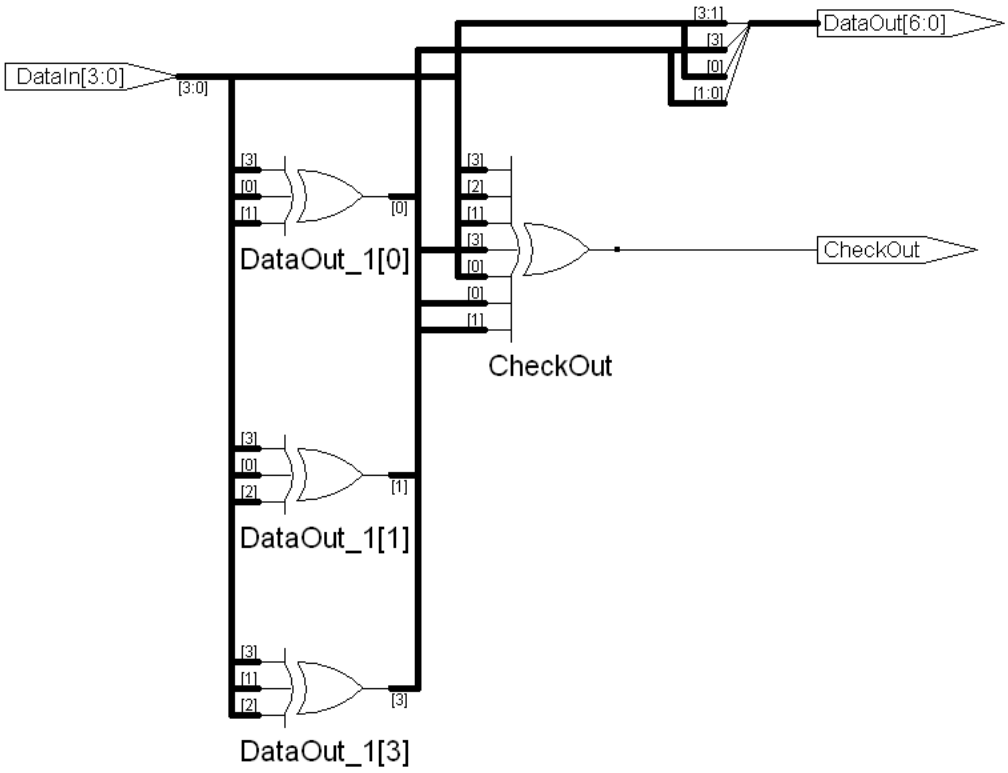

Figura 6.10: Codificatore Hamming (RTL netlist)

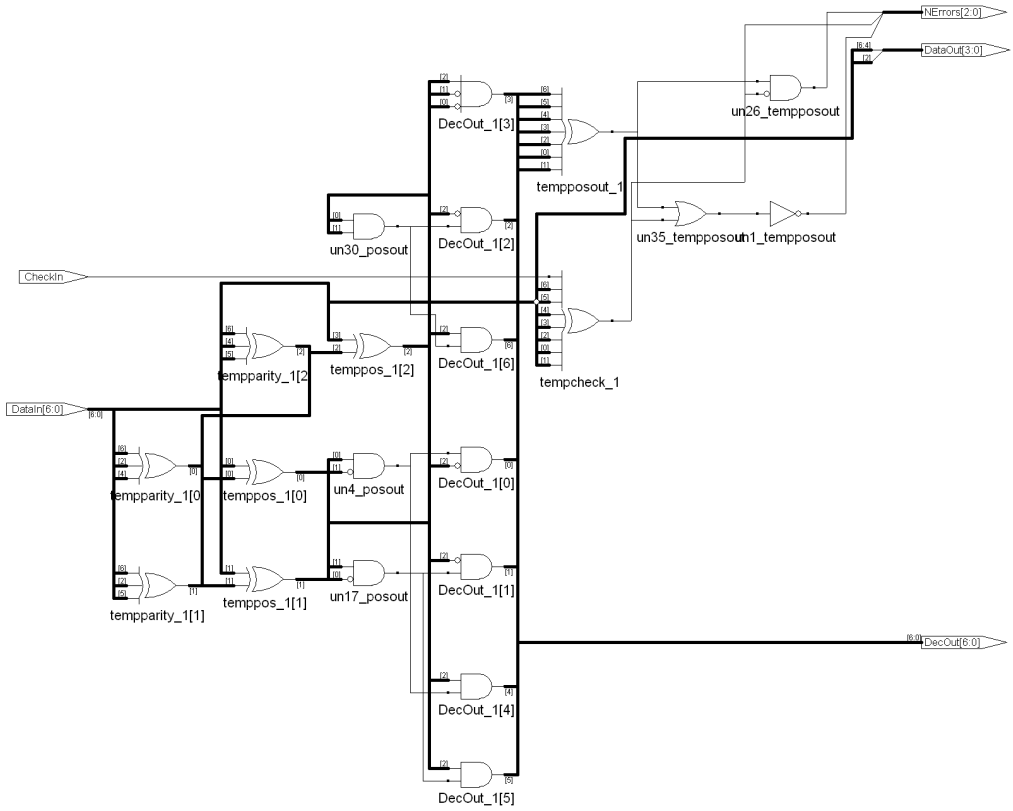

Figura 6.11: Decodificatore Hamming (RTL netlist)

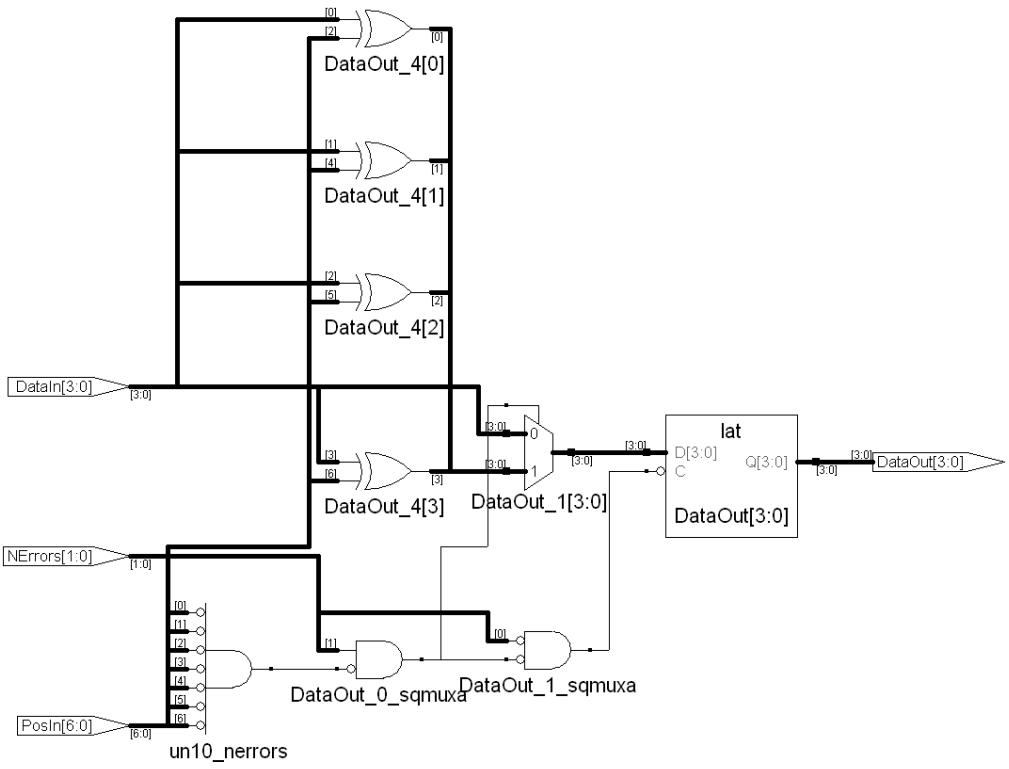

Figura 6.12: Correttore Hamming (RTL netlist)

Gate netlist

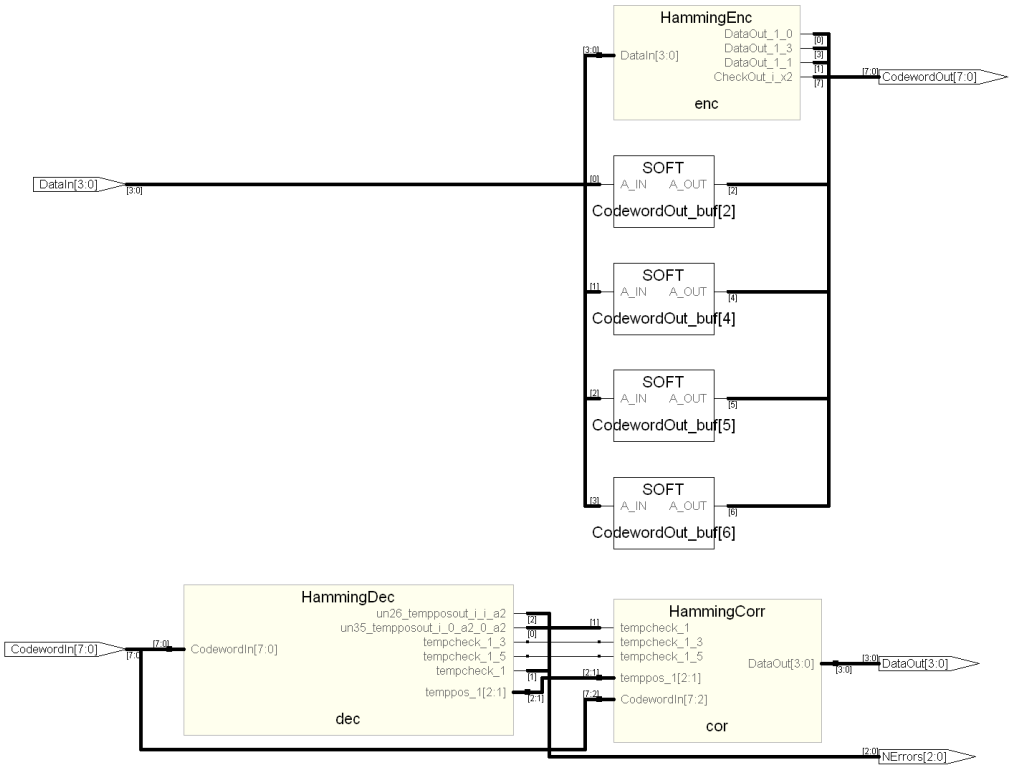

Figura 6.13: Schema generale Hamming (Gate netlist)

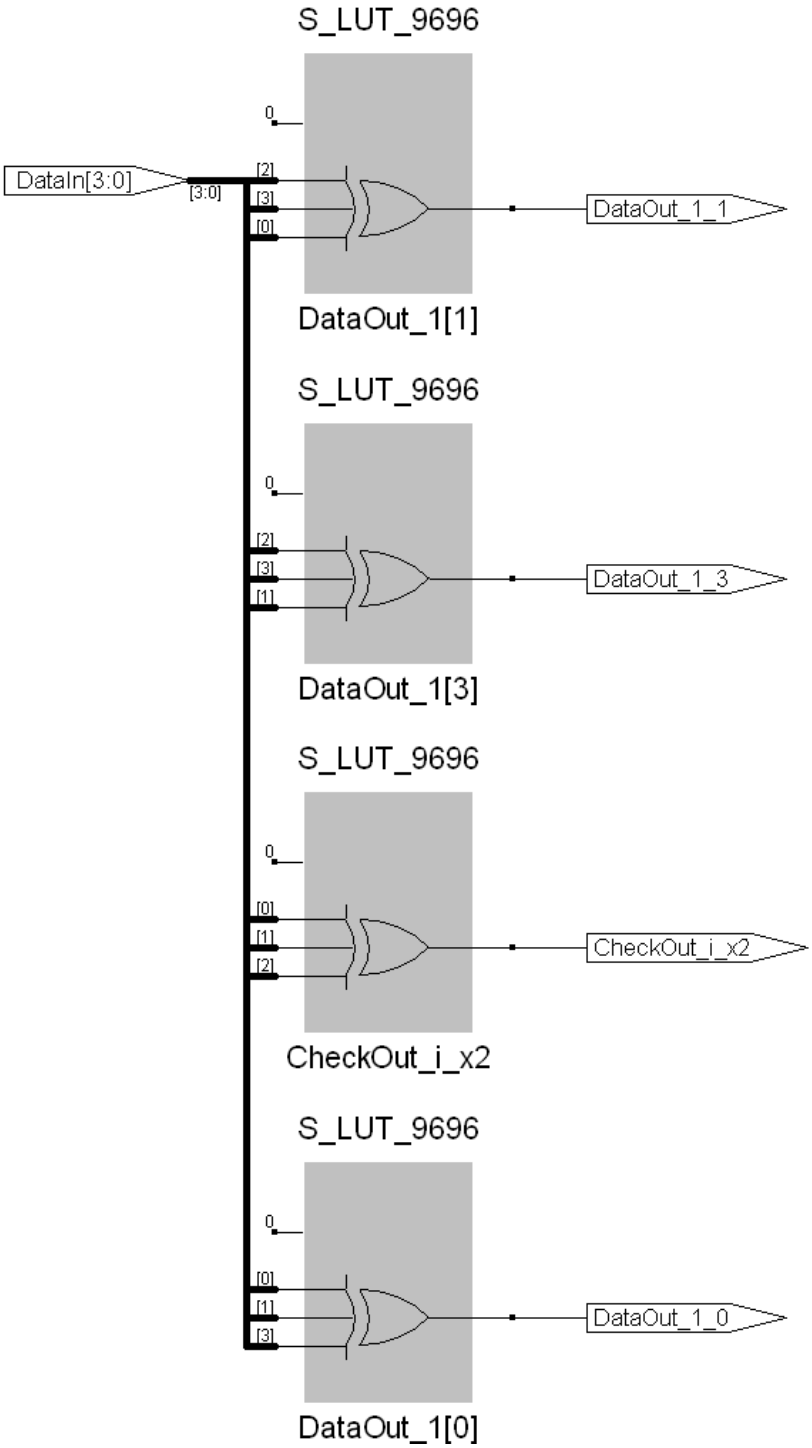

Figura 6.14: Codificatore Hamming (Gate netlist)

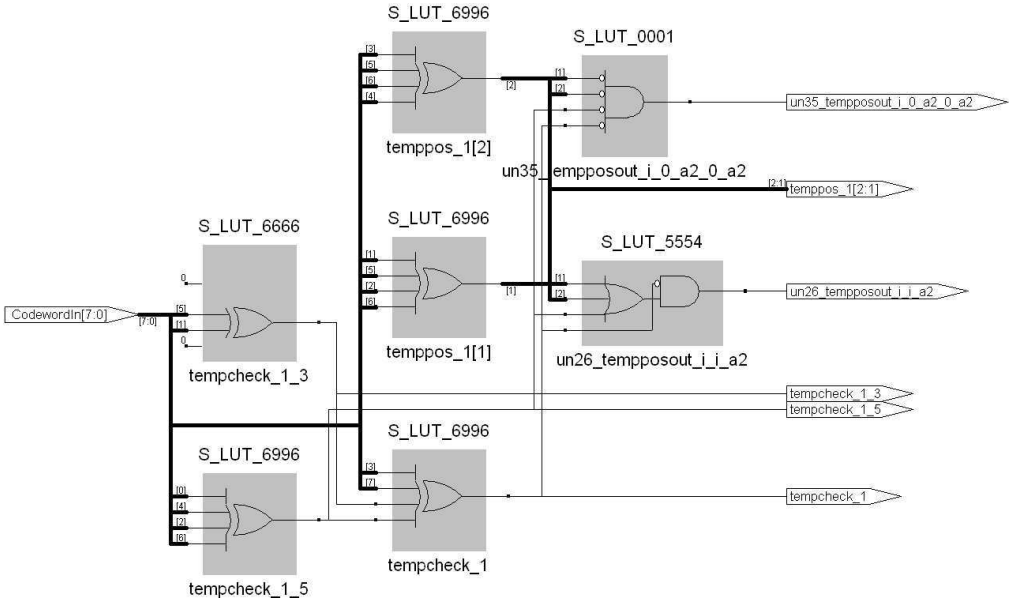

Figura 6.15: Decodificatore Hamming (Gate netlist)

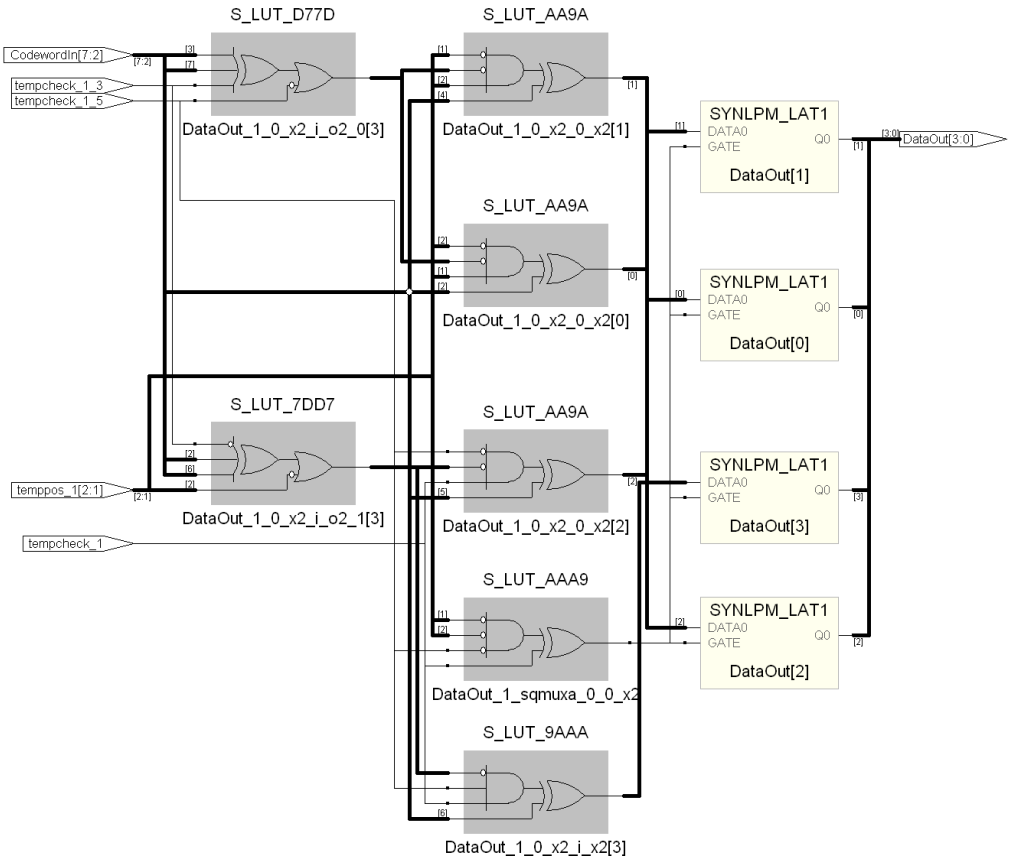

Figura 6.16: Correttore Hamming (Gate netlist)

6.2.2 CRC

RTL netlist

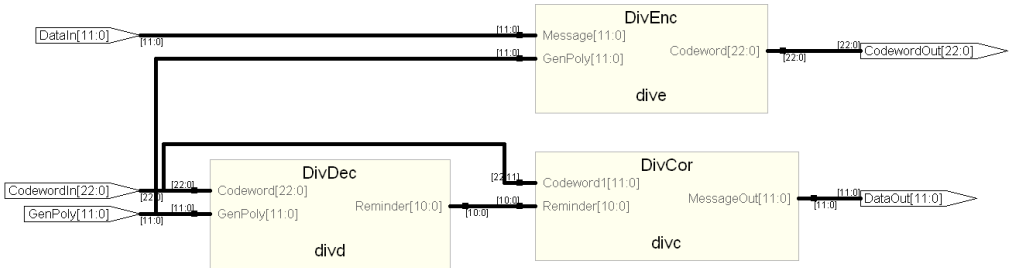

Figura 6.17: Schema generale CRC (RTL netlist)

Gli schemi relativi al codificatore e al decodificatore, non li riportiamo, perchè sono troppo grandi, si trovano in ogni caso nel cd allegato.

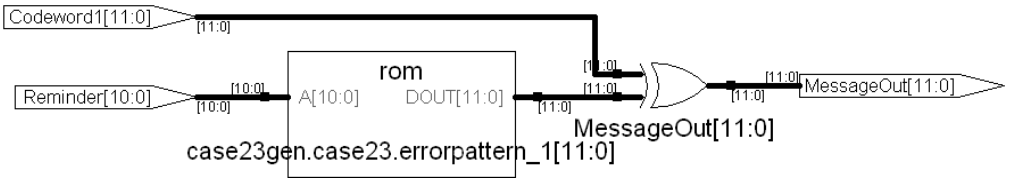

Figura 6.18: Correttore CRC (RTL netlist)

Gate netlist

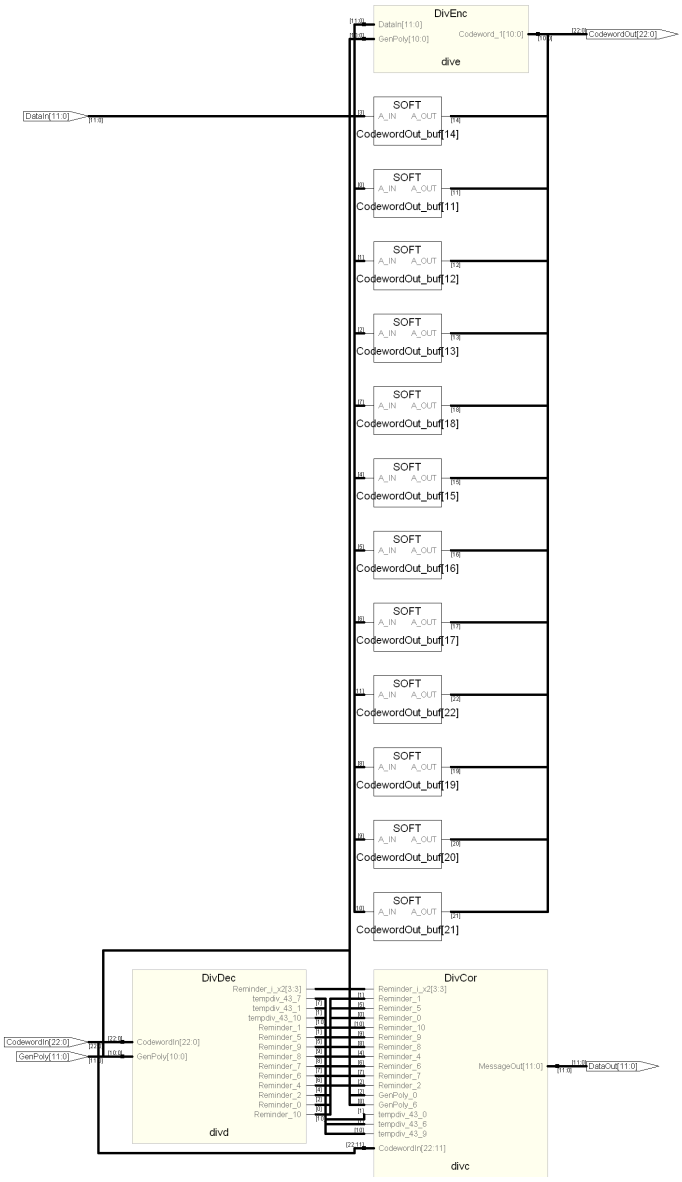

Figura 6.19: Schema generale CRC (Gate netlist)

Gli schemi relativi al codificatore, al decodificatore e al correttore, non li riportiamo, perchè sono troppo grandi, si trovano in ogni caso nel cd allegato.

6.3 Dimostratore in Java

L'idea di realizzare un dimostratore dei nostri metodi di correzione, usati per la ricerca, è quella di dare la possibilità di testare e quindi provare come funzionano i vari algoritmi di correzione, visualizzando graficamente il lavoro svolto da un metodo di correzione. Quindi, attraverso un'interfaccia grafica scritta in java, mettiamo a disposizione di tutti il funzionamento di un metodo che viene largamente impiegato nelle varie strumentazioni quotidiane. Il metodo fino ad ora implementato graficamente è quello di Hamming e il CRC. Il metodo di Hamming è inoltre, il metodo più conosciuto e facilmente implementabile. Contrariamente all'implementazione in vhdl, dove abbiamo utilizzato unicamente un Hamming(7,4) (poi diventato un 8,4), con il nostro applet in java volevamo dare la possibilità di inserire la dimensione dell'informazione a piacimento, implicando così di dover, ad ogni ciclo, ricalcolare la quantità di bit di ridondanza necessari. Così come la codeword che era formata diversamente a seconda della dimensione dei bit dell'informazione.
Questo dimostratore (applet) è stato compilato e testato con la versione 1.5 di Java della Sun.

6.3.1 Hamming

Parte grafica

La parte grafica dell'applet è composta da una finestra principale (l'applet appunto) che, a piacimento, potrebbe venir sostituita da un semplice programma in java che non giri necessariamente sul web, contenente un menu a tendina dove poter scegliere il metodo di correzione da testare. Una volta scelto il metodo si apre una nuova finestra, che è quella principale del metodo che conterrà il pannello vero e proprio dove girerà poi l'algoritmo con tutte le sue funzioni grafiche.
Il pannello di Hamming è composto nel modo seguente. Come prima possibilità di scelta si può decidere di selezionare la modalità binaria o esadecimale di IN e OUT dei vari campi di testo dell'area di lavoro, poi abbiamo inserito una tendina dove bisogna specificare quanti bit di informazione si vogliono spedire e una che da la possibilità di inserire una parola già esistente dalla tendina di sinistra nel prossimo campo che è la vera e propria area per l'inserimento della parola da spedire (sia in binario che esadecimale) facendo attenzione all'ordine di inserimento dei vari bit di info. Una volta arrivati fino a qui si può procedere con la codifica che calcolerà e inserirà al posto giusto i vari bit di controllo scrivendo la codeword nell'apposito campo di testo, contemporaneamente si visualizzerà nel campo "errore" un vettore di tutti 0 di lunghezza uguale alla codeword che permette di inserire o meno l'errore nella codeword iniziale, cambiando o meno il bit che si desidera alterare. Una volta introdotto l'eventuale errore viene visualizzata la nuova

codeword in ricezione modificata, nella casella di testo apposita. A questo punto si procede con la decodifica che calcolerà e, se necessario, sostituirà il bit invertito precedentemente dall'errore visualizzando la codeword corretta e la parola originariamente spedita nelle apposite caselle di testo.

Funzionamento Per seguire una ciclo regolare del programma è necessario per prima cosa selezionare, dai bottoni posti in cima alla pagina, la modalità di svolgimento dello stesso: binario oppure esadecimale. Una volta effettuata la scelta apparirà nella tendina di sinistra, quella dei messaggi da inviare, una serie di messaggi binari o esadecimali già pronti per essere elaborati. Mentre nella tendina al centro compariranno il numero di bit da spedire, se si è scelta la modalità binaria, o il numero di caratteri altrimenti. A questo punto si può immettere nell'apposito campo il messaggio che si vuole inviare, scegliendolo tra quelli già pronti oppure immettendolo da tastiera. Si ricordi che nel caso si selezioni uno già preconfezionato, la quantità di bit o di caratteri viene selezionata automaticamente, mentre se si desidera immetterne uno manualmente bisogna specificare la quantità desiderata, dopo aver comunque selezionato la modalità. Il procedimento di default accetta una parola binaria di 4 bit. Nel caso di insuccesso la parola immessa si colora di rosso per segnalarlo, quindi bisogna correggerla.
Quando tutto è a posto si può cliccare su CODIFICA la quale operazione andrà a scrivere il messaggio spedito nella coda dei messaggi già spediti e produrrà la codeword nell'apposita casella di testo. Contemporaneamente si visualizzerà un vettore in formato binario, sia se si è scelta la modalità binaria che quella esadecimale, di lunghezza corrispondente alla codeword (in binario). Questo permetterà di scegliere esattamente quale bit si voglia alterare.
Una volta cambiato il bit, e ciccando su INTRODUCI, apparirà la nuova codeword. A questo punto è possibile procedere con la decodifica che calcolerà se e quale bit è stato alterato e restituirà, stampandoli negli appositi campi di testo, la codeword e la parola decodificata. In caso di successo, l'area di testo verrà colorata di verde, oppure di rosso altrimenti.

Parte algoritmica

Per quanto riguarda la parte algoritmica abbiamo realizzato una classe java apposita contenente i vari metodi necessari al nostro scopo. La classe è strutturata in linea di massima nel seguente modo: ci sono metodi "ordinari" utilizzati per ricevere/trasmettere la quantità di bit di informazione, per ricevere/trasmettere l'informazione stessa, per calcolare la quantità di bit della codeword (dopo aver saputo la quantità di bit di check) e per ricevere/trasmettere l'errore; abbiamo un metodo per calcolare la quantità di bit di controllo e uno che ci da le posizioni di questi bit di controllo; poi abbiamo implementato, partendo dalla sola conoscenza teorica del metodo di Ham-

ming, il metodo che calcola la codifica; con lo stesso principio abbiamo scritto la decodifica; un metodo che serve solamente, conoscendo la lunghezza totale della codeword, a costruire il vettore dell'errore settato a zero; abbiamo un metodo che restituisce al codeword modificata dall'eventuale errore inserito; e per ultimo un metodo che dalla codeword restituisce la parola (corretta oppure no) originariamente spedita.

Codifica Per iniziare la codifica c'è bisogno della grandezza della parola da spedire. Questo permetterà all'algoritmo di calcolare la quntità e la posizione dei bit di check, oltre che alla lunghezza della codeword totale.
Il programma riceve in input una stringa di testo, che bisogna trasformare in un vettore di interi per l'elaborazione. Per fare ciò è necessario per prima cosa convertirla in un array di caratteri. Prima di inviare il messaggio alla codifica è necessario invertire il vettore così da avere il giusto inserimento nella codeword dal bit meno significativo a quello più significativo. A questo punto si passa il messaggio all'algoritmo e si fa calcolare la codeword codificata. Per il calcolo della codeword, abbiamo bisogno del messaggio da inviare e la sua relativa lunghezza, il vettore delle posizioni di check e il numero di bit di check, e naturalmente la lunghezza totale della codeword.
Per prima cosa viene inserito il vettore dell'info da spedire in un vettore temporaneo di codeword, nelle posizioni che non siano quelle dei bit di check. Ora inizia la vera e propria ricerca del valore del bit di check.
Un annidamento di for permette di scansionare la codeword temporanea e di calcolarne l'esatto valore del bit di ridondanza partendo dal principio della teoria della codifica di Hamming. In pratica ad ogni ciclo più esterno viene fatto un ciclo più interno basato sulla posizione attuale del bit di check, in modo da avere ogni volta un ciclo sui bit che servono a calcolare il bit di controllo relativo. Ogni ciclo corrispondente a questi bit, avrà un suo ciclo più interno ad ogni passo siccome i bit da controllare vanno a gruppi a dipendenza del valore del bit che si sta controllando.
Quando viene fatto il conteggio dei bit per ogni gruppetto ed ogni ciclo del bit di ridondanza, se ne calcola la parità per poter stabilire se il bit di check sarà un 1 o uno 0. Da qui viene quindi calcolata la codeword relativa e riconvertita in stringa, in modo da poterla immettere nella casella di testo corrispondente. Nel caso si scelga la modalità esadecimale, abbiamo inserito una classe appositamente allo scopo di convertire, nel caso opportuno, una stringa da esadecimale a binaria. In pratica ogni volta che si introduce una stringa esadecimale questa viene convertita in un array di caratteri e poi spedita al convertitore che provvederà a convertire larray in una stringa binaria, con la quale sarà possibile riprendere il lavoro esattamente come per la modalità binaria. Una volta calcolata la codeword la si converte in un vettore di caratteri da spedire al convertitore che restituirà una stringa esadecimale, pronta da essere scritta nella casella di testo corrispondente.

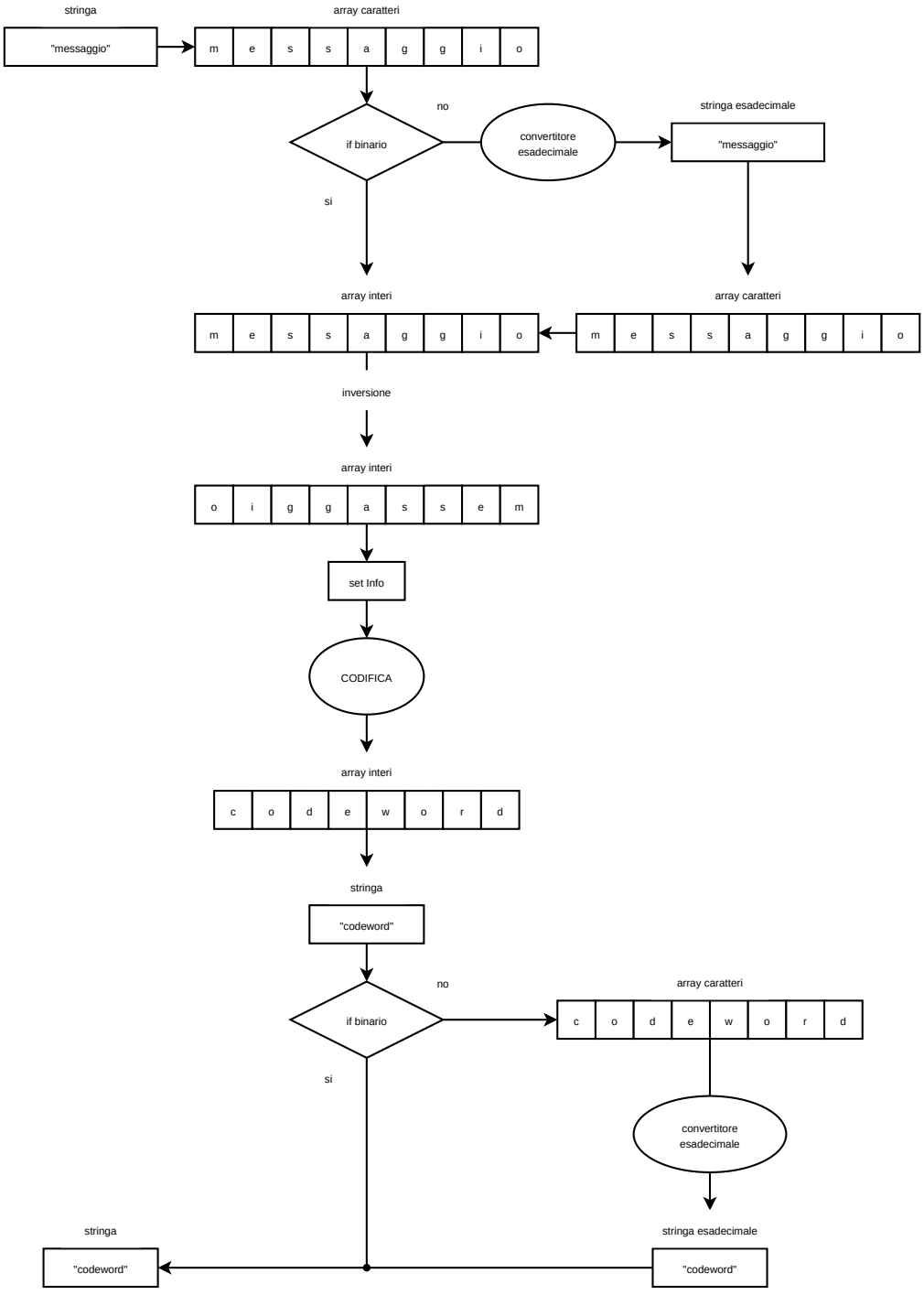

Figura 6.20: Implementazione codifica Hamming nel dimostratore

Inserimento errori La stringa di errore sarà sempre di lunghezza della codeword binaria, sia in modalità binaria che esadecimale. Il vettore di default è composto unicamente da zeri, ciò vuol dire nessun errore nella codeword, nel caso si modifichi un bit (ricordiamo che questo dimostratore di Hamming è in grado di rilevare e correggere unicamente errori singoli), l'algoritmo provvederà a modificare la codeword in corrispondenza di quel bit e di riscriverla nella casella di testo corrispondente. Le operazioni di conversione sono pressochè identiche a quelle nella codifica, ovvero: la stringa convertita in array di caratteri e poi in array di interi, prima di essere mandata all'algoritmo che restituirà la nuova codeword che sarà a sua volta riconvertita in stringa. Per quanto riguarda la modalità esadecimale, al momento che si riceve la nuova codeword dall'algoritmo la si converte in array di caratteri, per poi essere convertita, dal metodo apposito, in una stringa esadecimale.

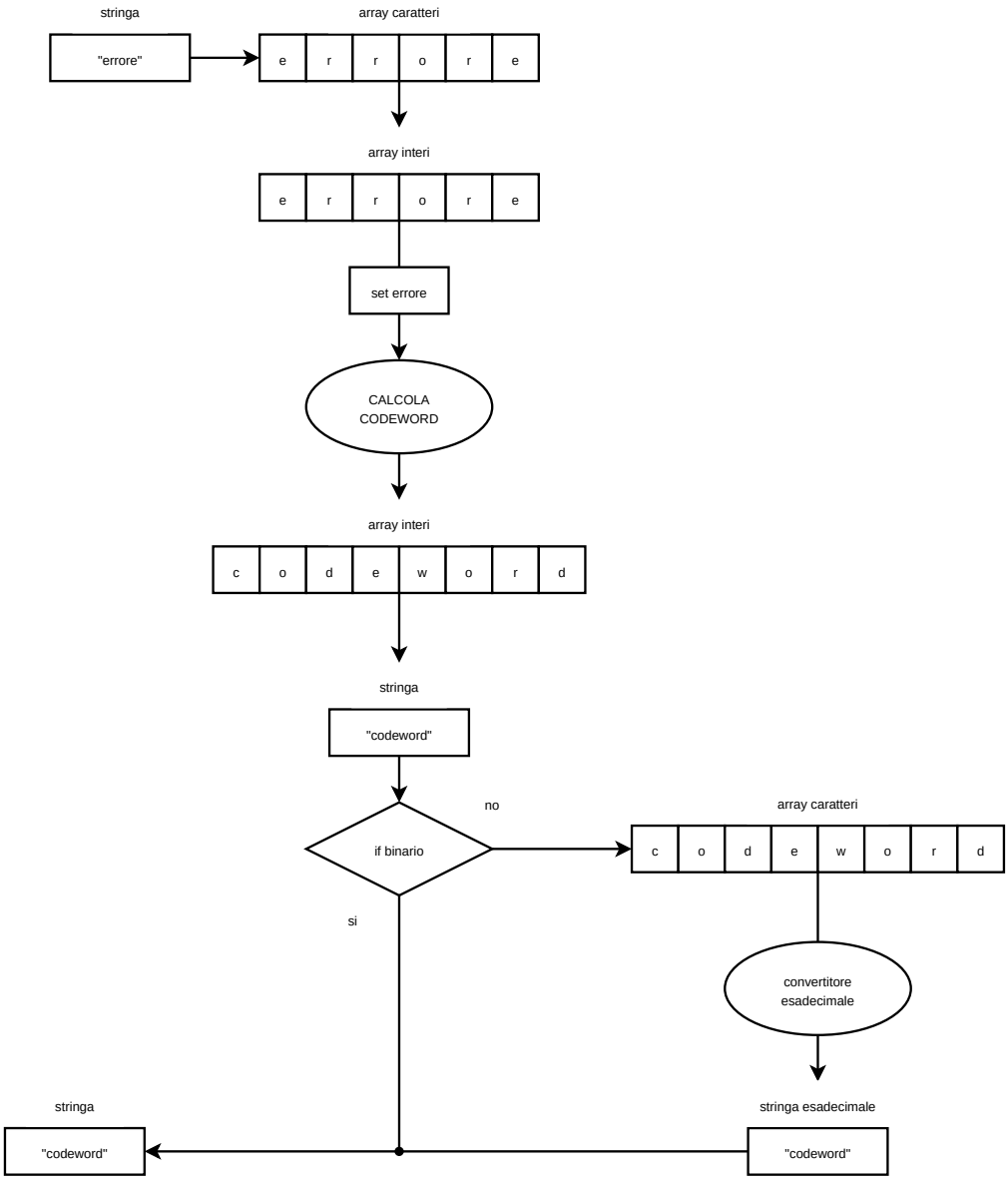

Figura 6.21: Implementazione inserimento errrori dimostratore Hamming

Decodifica La decodifica di Hamming, consiste nel prelevare la stringa della codeword ricevuta ed elaborarla in modo molto simile che per la codifica. Al termine delle operazioni si calcolerà il valore che permetterà di localizzare il bit errato, grazie al conteggio dei bit di parità. Quindi il bit corrispondente verrà modificato e la decodifica restituirà la codeword decodificata. Quest'ultimo viene riconvertita in stringa e nel caso ci trovassimo in modalità esadecimale, verrà nuovamente convertita in array di caratteri per rielaborarla e ottenere la stringa esadecimale che varrà inserita nell'apposita casella di testo. Una volta ottenuta la codeword decodificata manca l'ultima operazione che consiste nel estrarre il messaggio inizialmente spedito. Una volta ottenuto questo array di interi, viene rigirato, per la priorità dei bit, e convertito in stringa e inserito nell'apposita sezione. Nel caso si tratti di modalità esadecimale quando si riceve l'info come vettore di interi, la si converte in stringa e successivamente in vettore di caratteri. Questo vettore di caratteri viene mandato al convertitore che ritorna la stringa esadecimale. Quest'ultima bisogna riconvertirla in array di caratteri, invertirla e convertirla nuovamente in stringa pronta da stampare a video nell'apposita casella di testo.

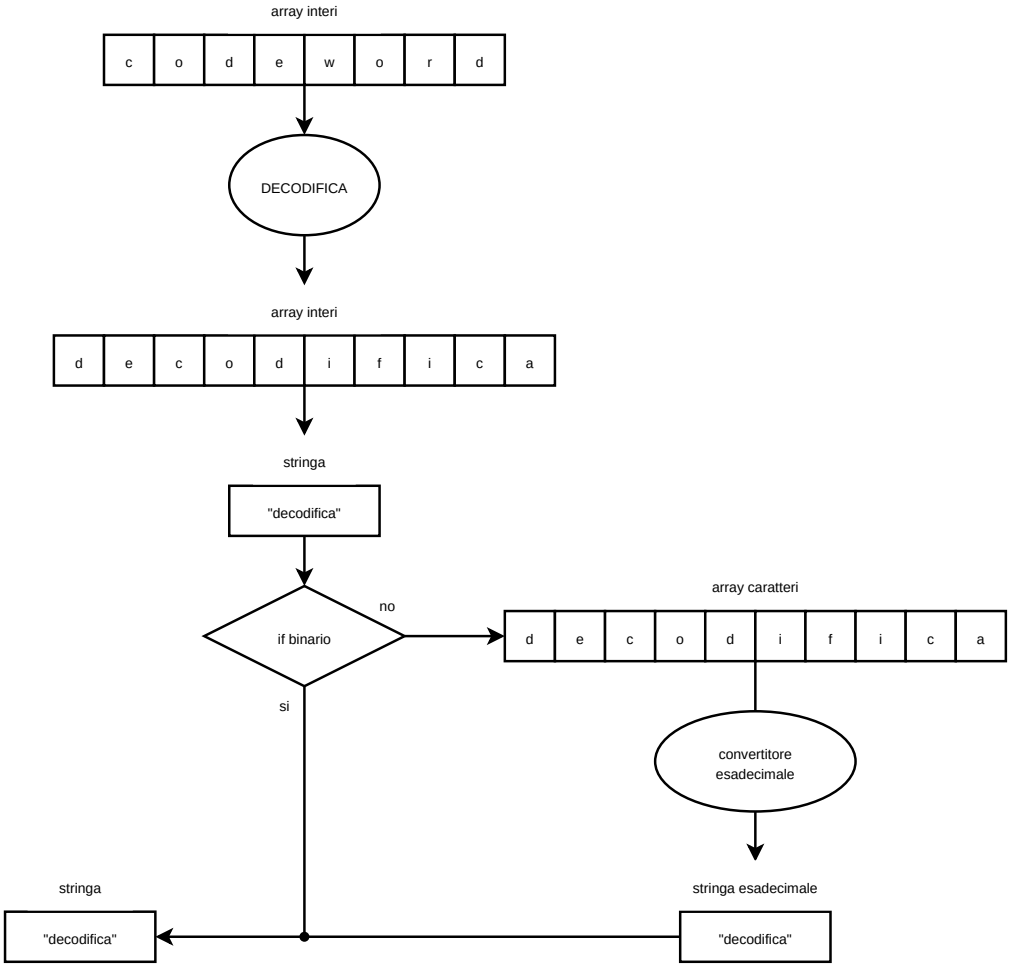

Figura 6.22: Implementazione decodifica Hamming nel dimostratore

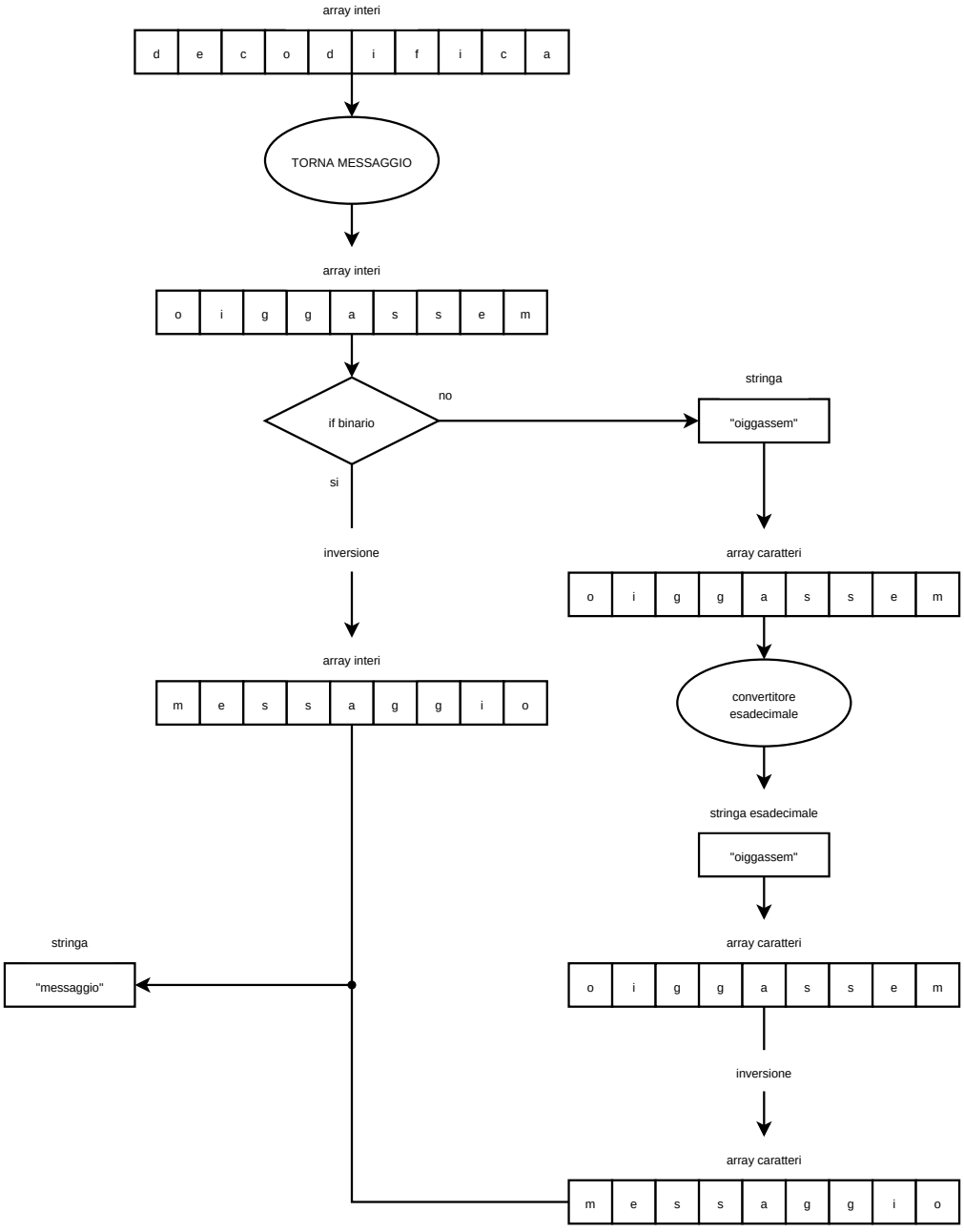

Figura 6.23: Impl. ricon. info. orig. in Hamming nel dimostratore

6.3.2 CRC

Parte grafica

Il funzionamento del Crc, per quanto riguarda la grafica, è simile a quello di Hamming ma semplificato molto dal fatto che esiste solo la possibilità di inserire i dati in modalità binaria e che è possibile codificare unicamente messaggi di lunghezza 12 bit. Nel primo campo va inserito il messaggio da spedire e nel caso non si inserisca correttamente viene colorata in rosso la stringa e bisogna correggerla. Una volta inserita la parola si può cliccare su CODIFICA e apparirà nella casella sottostante la codeword codificata di 23 bit costituita dai 12 bit dell'info e dagli 11 bit del resto della divisione del Crc. Contemporaneamente verrà visualizzato il vettore di errore inizializzato a zero nel quale sarà possibile andare a modificare uno più bit di errore nella codeword corrispondente. A questo punto, una volta premuto su INTRODUCI, si crea la codeword ricevuta contenente l'errore identica all'originale. La DECODIFICA riempirà la casella di testo immediatamente dopo e quella successiva contenenti rispettivamente il resto della divisione e il messaggio inviato. Se verrà colorato in rosso significa che gli errori non sono stati corretti, erano superiori a due, mentre se sarà verde è andato tutto come previsto.

Parte algoritmica

Codifica La codifica del CRC prevede in pratica di inserire il messaggio da inviare in una temporanea codeword, di tutti zeri, nella parte più significativa e nella restante parte si inseriranno i bit del resto della divisione di questa temporanea codeword con il polinomio generatore. Il programma riceve in input la stringa del messaggio, che verrà convertita in array di caratteri e poi successivamente in array di interi. A questo punto si inverte il vettore e si setta l'info nell'algoritmo. Il metodo di codifica preleva dal metodo apposito l'info da elaborare e come prima operazione la inserisce nel vettore di codeword e poi invia al divisore quest'ultima assieme al polinomio generatore. La divisione ritorna il resto di questa divisione che verrà poi inserito nella parte meno significativa del vettore di codeword. Ottenuto il vettore di codeword, si converte in stringa e lo si stampa nell'apposita casella di testo.

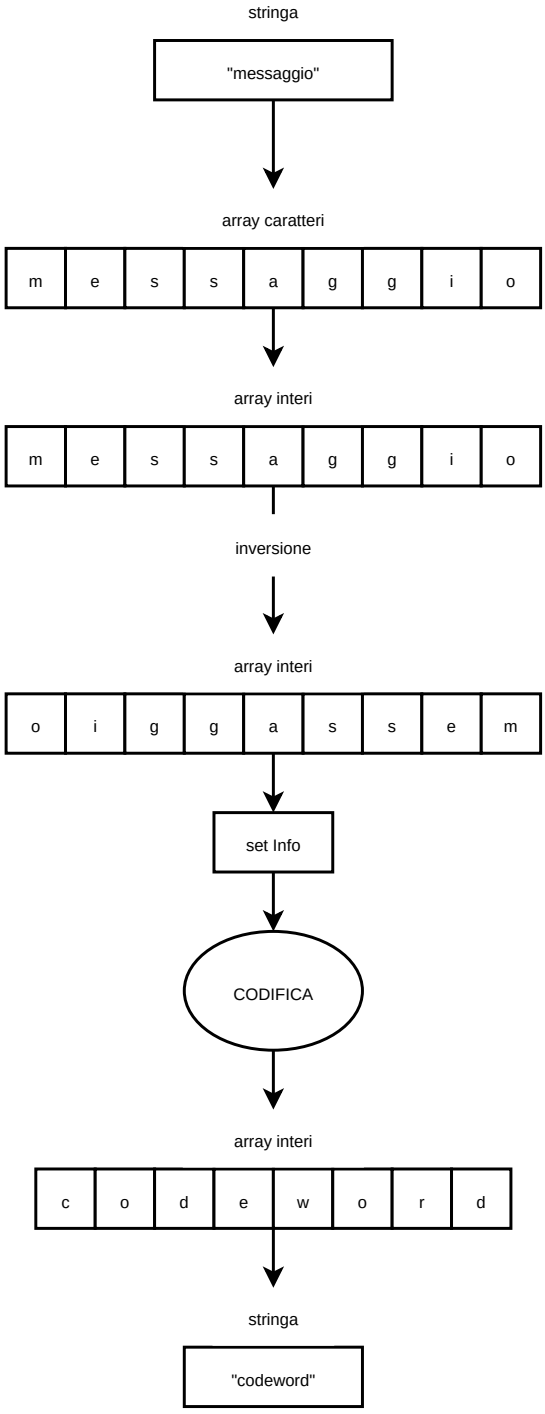

Figura 6.24: Implementazione codifica CRC nel dimostratore

Inserimento errori Una volta generata la codeword, apparirà nella casella apposita anche una stringa di tutti zeri nella quale si andranno a sostituire i bit che si desidera alterare nella codeword. Il procedimento di input e output è simile come in precedenza e come per Hamming. Settato l'errore si ricalcola la nuova codeword, modificata eventualmente dall'errore, si fanno le dovute conversioni e la si scrive nella casella di testo apposita.

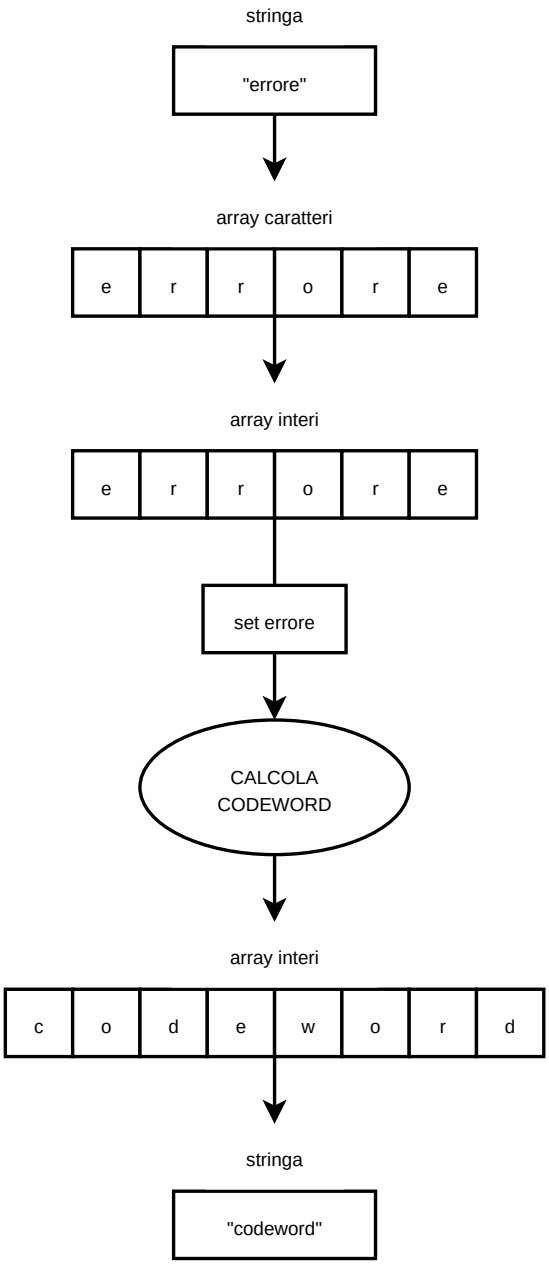

Figura 6.25: Implementazione inserimento errori CRC nel dimostratore

Decodifica Per quanto riguarda la decodifica del Crc, l'algoritmo prevede di ricevere in input la codeword ricevuta e il polinomio divisore e produrre come output il resto della divisione tra questi due. Gli inserimenti nel campo di testo, e quello che riguarda la dinamica degli input output, è simile al precedente sistema. La differenza sostanziale appunto è che verrà visualizzato il resto della divisione, che è quello che determina se ci sono errori o meno, al posto della codeword decodificata come avviene per Hamming. In fine dalla decodifica, viene calcolato il pattern di errore e la codeword viene corretta. L'ultima parte dell'algoritmo prevede di restituire il messaggio originale possibilmente corretto e di visualizzarlo graficamente sull'applet.

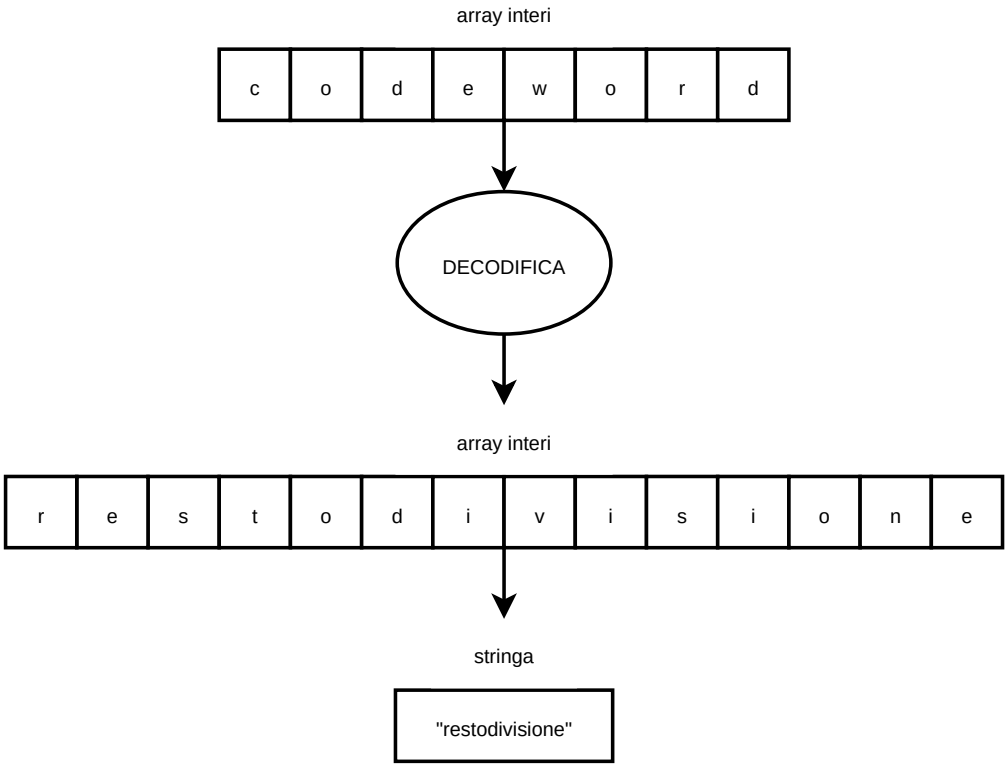

Figura 6.26: Implementazione decodifica CRC nel dimostratore

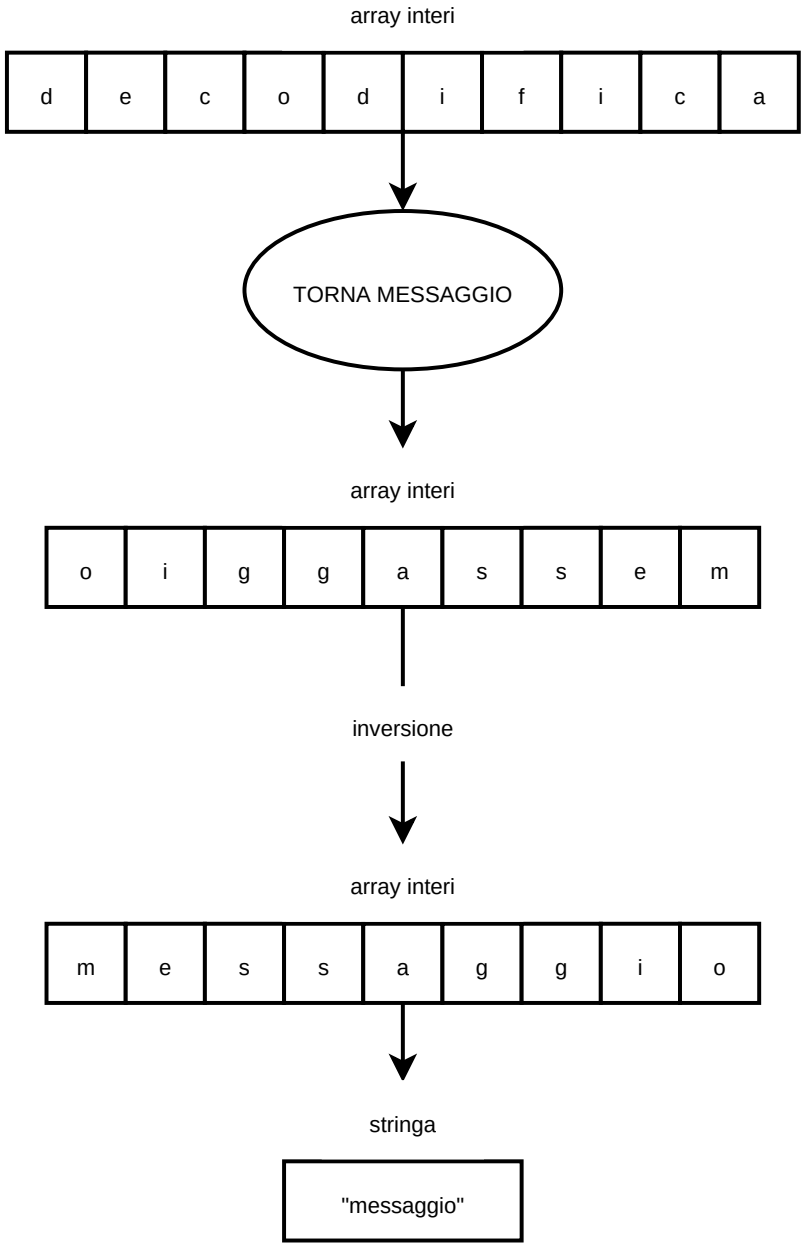

Figura 6.27: Impl. ricon. info. orig. CRC nel dimostratore

Capitolo 7

Test

7.1 Implementazione in VHDL

7.1.1 Hamming

Per il test del metodo di Hamming, abbiamo creato un'entità e un'architettura (strutturale) all'interno della quale istanziamo e colleghiamo i vari componenti (codificatore, inserimento errori, decodificatore e correttore). Questo componente *HammingTest.vhd*, viene istanziato nel test bench *Hamming_Tb.vhd*. Il test avvine basandosi sui valori calcolati dal programma *patterngen.c* che si trova nella sezione C.1.4 a pagina 83. Il suddetto programma, genera un file con la seguente struttura.

messaggio	pattern errore	umero errori
000	00000101	1
...	...	...

Tabella 7.1: Struttura file pattern per test codice di Hamming

Il test bench legge questo file riga per riga e usa le informazioni per testare il metodo.
I messaggi che si possono passare all'algoritmo per creare le codeword sono $2^4 = 16$. Il programma pattergen, genera tutti i possibili casi con i pattern di errore contenenti: 0 errori, 1 errore e 2 errori. Il numero di casi, l'abbiamo calcolato nel seguente modo.

$$C_n^k = \frac{n!}{k! \cdot (n-k)!} = \binom{n}{k}$$

$$C_8^0 = 1,\ C_8^1 = 8,\ C_8^2 = 28$$

$$n_c = 1 \cdot 16 + 8 \cdot 16 + 28 \cdot 16 = 16 + 128 + 448 = 592$$

Il seguente diagramma di flusso mostra come funziona il test.

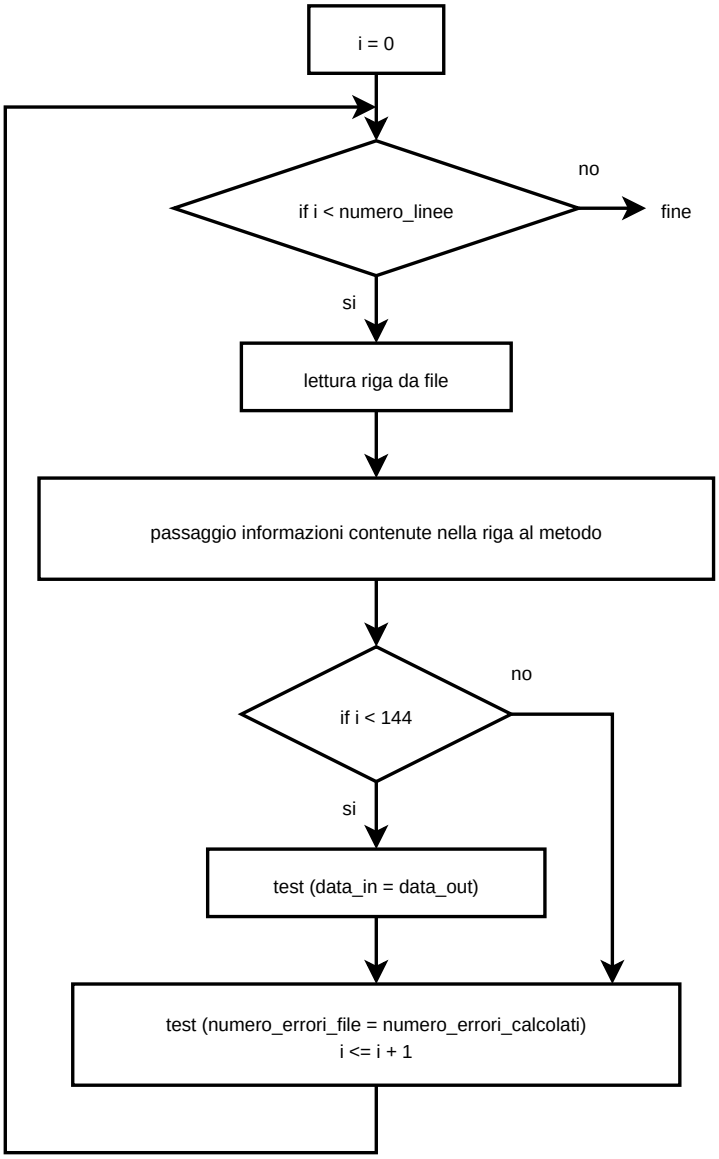

Figura 7.1: Diagramma di flusso test bench Hamming

7.1.2 CRC

Per il test CRC, abbiamo creato un'entità e un'architettura (strutturale) all'interno della quale istanziamo e colleghiamo i vari componenti (codificatore, inserimento errori, decodificatore e correttore). Questo componente *DivTest.vhd*, viene istanziato nel test bench *Div_Tb.vhd*. Nel test bench, facciamo diversi test, per verificare il corretto funzionamento del metodo.

Questi test sono necessari, perchè dobbiamo verificare empiricamente, se con il polinomio generatore 0xC75 si possono correggere fino a 3 errori (come con il codice di Golay (23,12,7)). Abbiamo fatto due tipi di prove: il primo inserendo gli errori nei 12 bit più significativi delle codeword (i bit del messaggio originale) mentre il secondo inserendo gli errori in tutti e 23 i bit della codeword. I seguenti calcoli, mostrano quanti casi dobbiamo testare.
I messaggi possibili sono in totale: $2^{12} = 4'096$.
Con l'inserimento di errori solamente nei 12 bit del messaggio, si ottengono:

$$C_{12}^0 = 1,\ C_{12}^1 = 12,\ C_{12}^2 = 66,\ C_{12}^3 = 220$$

$$n_c = 1 \cdot 4'096 + 12 \cdot 4'096 + 66 \cdot 4'096 + 220 \cdot 4096 = 1'224'704$$

Con l'inserimento di errori in tutti i 23 bit, si ottengono:

$$C_{23}^0 = 1,\ C_{23}^1 = 23,\ C_{23}^2 = 253,\ C_{23}^3 = 1'771$$

$$n_c = 4'096 + 94'208 + 1'036'288 + 7'254'016 = 8'388'608$$

Per ognuno dei due tipi di prova elencati, generiamo come prima cosa la tabella (sotto forma di case) per la correzione. Questa tabella contiene le copie: resto divisione / pattern di errore (i 12 bit relativi al messaggio). In seguito testiamo l'algoritmo con tutti i casi possibili (messaggi originali e pattern di errore), al fine di avere la conferma che funzioni.
I seguenti diagrammi di flusso, mostrano come funziona la generazione del case per il primo tipo di prova. Per il secondo, è sufficiente fare le stesse operazioni ma con le variabili i che iniziano da 0.

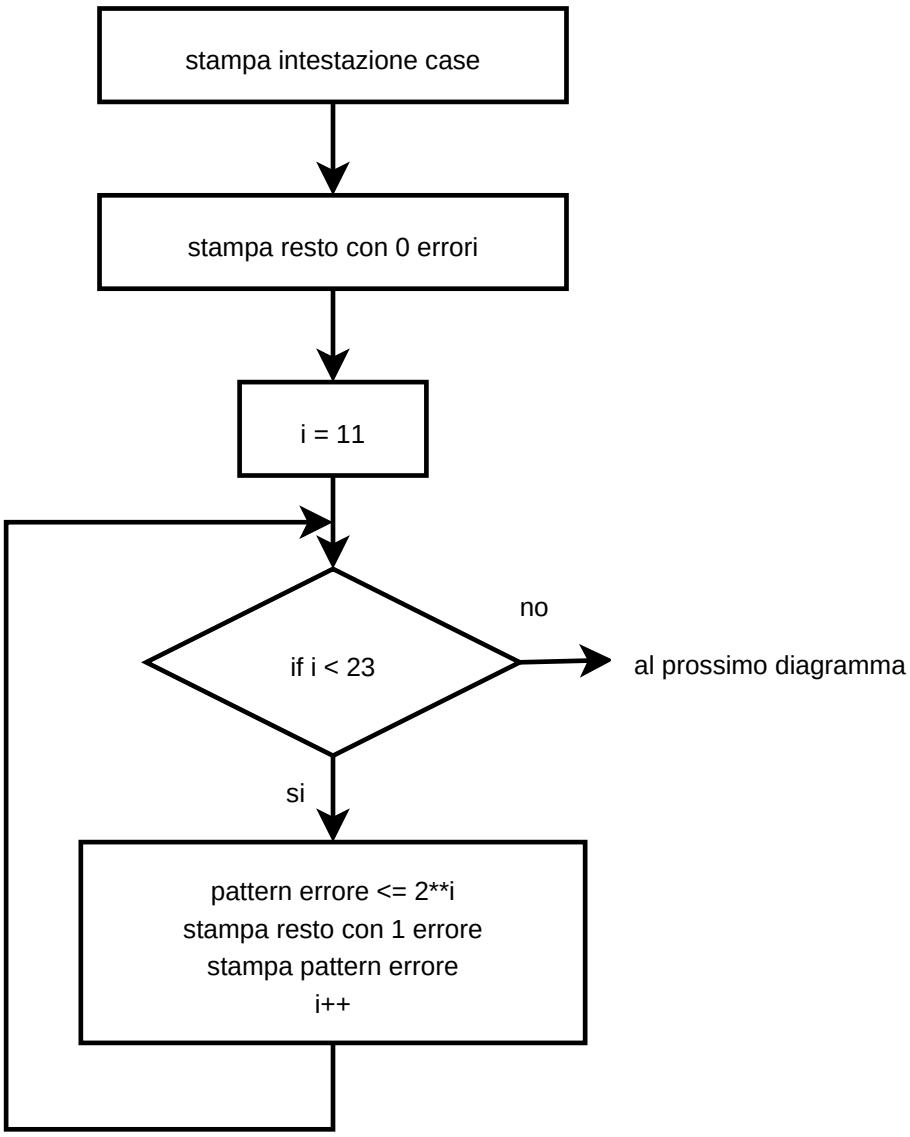

Figura 7.2: Diagramma di flusso generazione case per corr. CRC parte 1

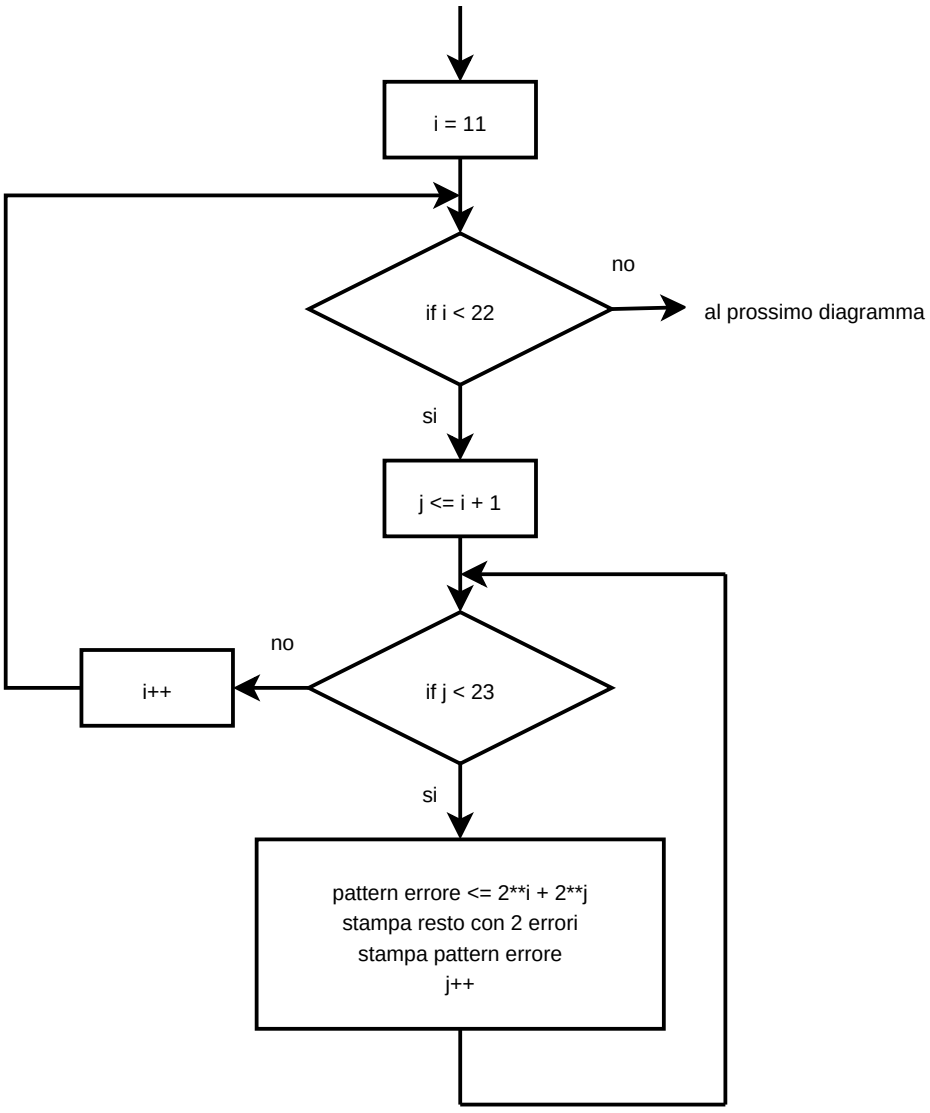

Figura 7.3: Diagramma di flusso generazione case per corr. CRC parte 2

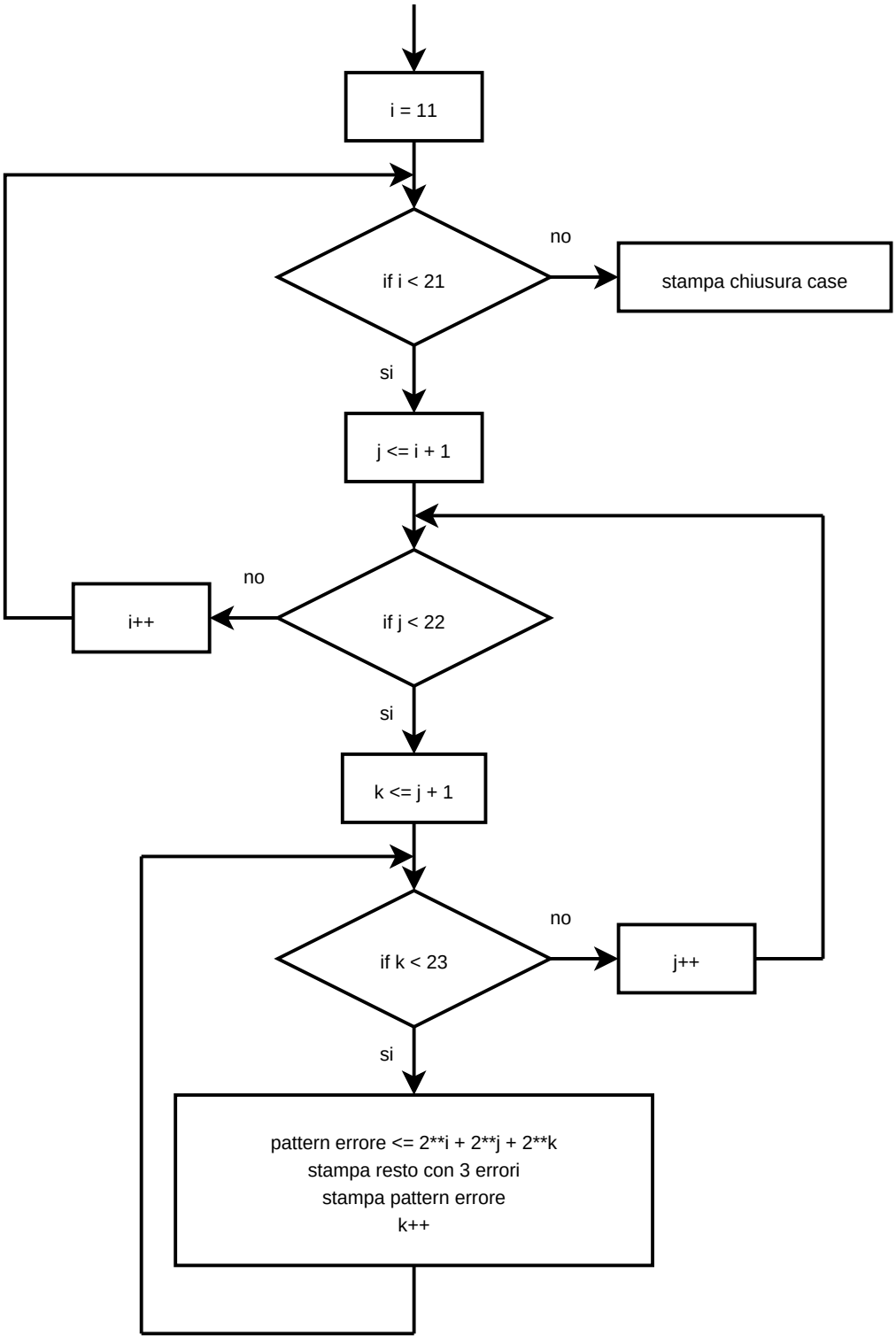

Figura 7.4: Diagramma di flusso generazione case per corr. CRC parte 3

I seguenti diagrammi di flusso, mostrano come funziona il test per il primo tipo di prova. Per il secondo, è sufficiente fare le stesse operazioni ma con le variabili j che iniziano da 0.

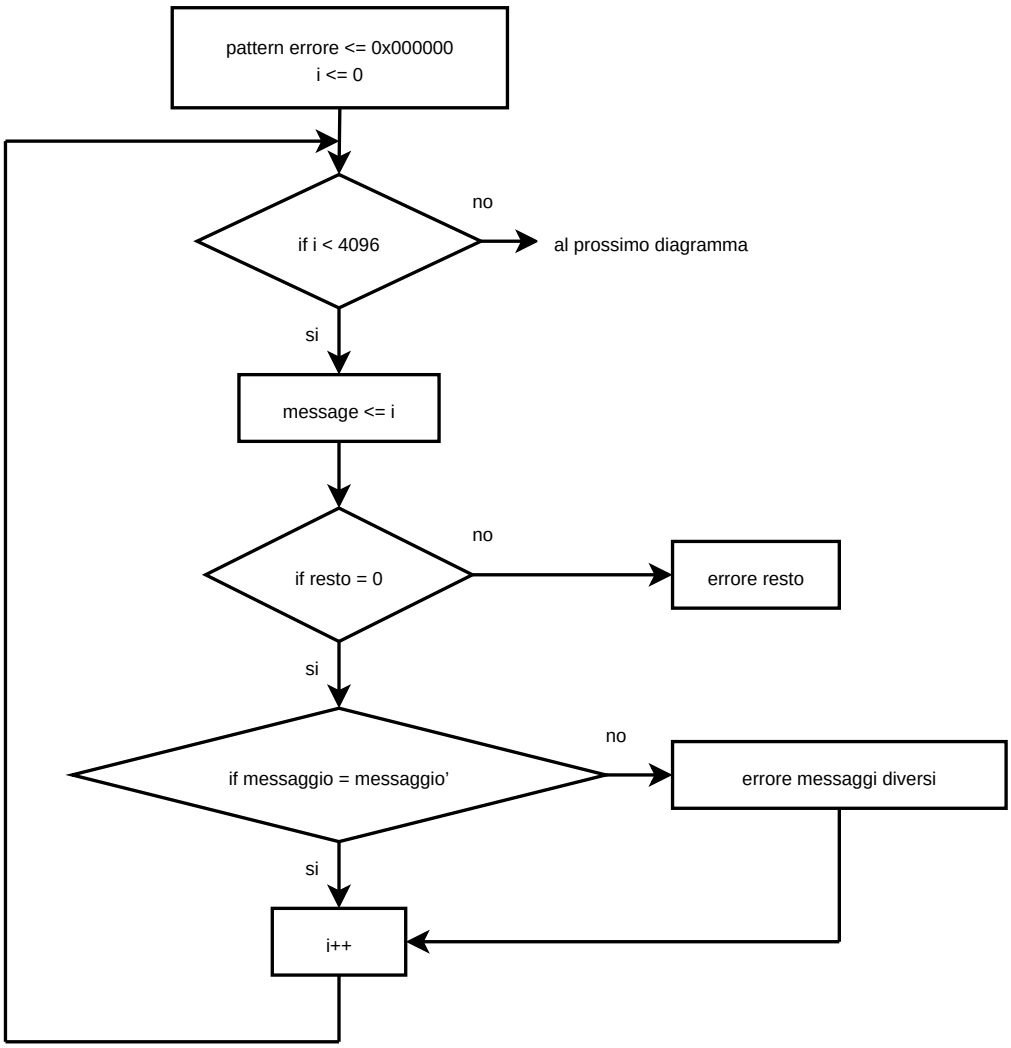

Figura 7.5: Diagramma di flusso test CRC con 0 errori

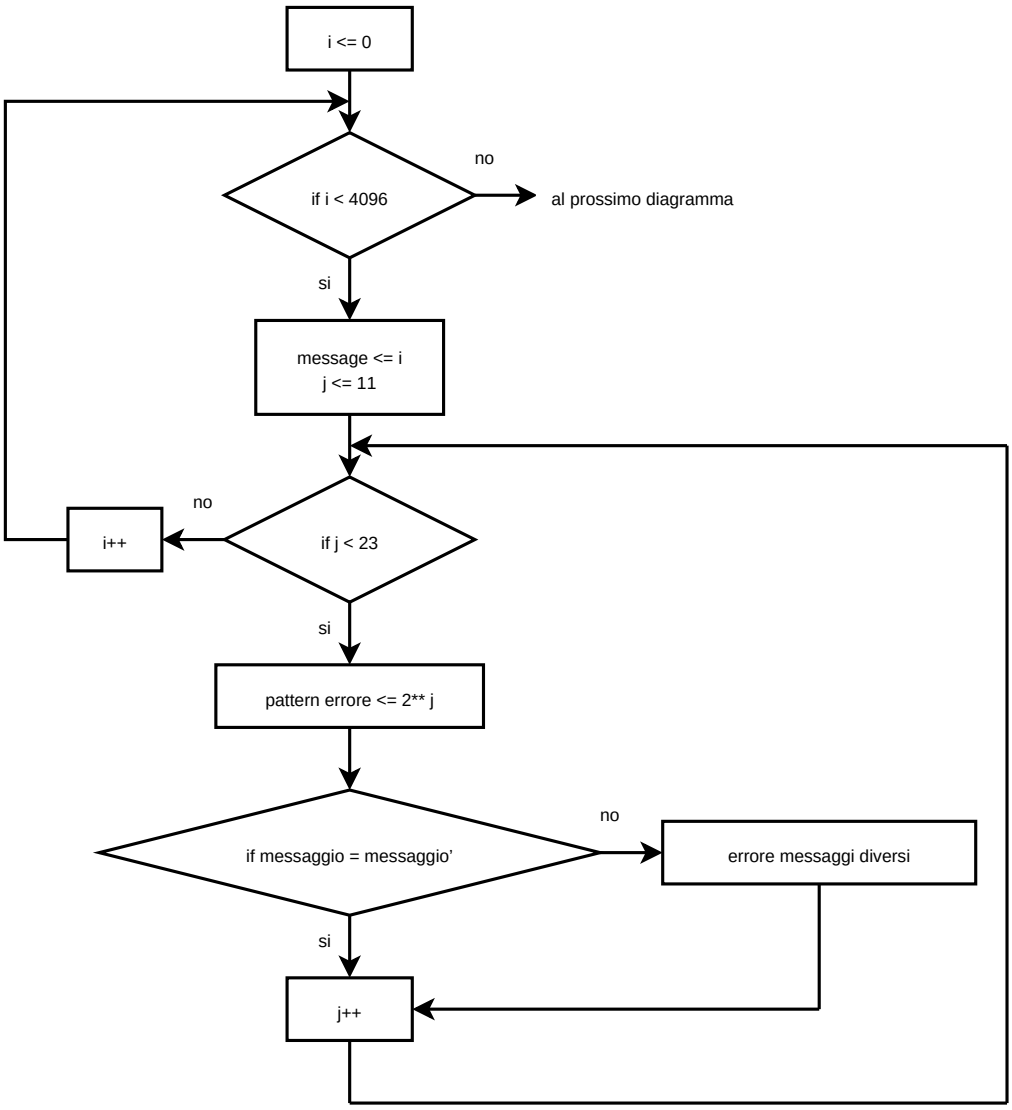

Figura 7.6: Diagramma di flusso test CRC con 1 errore

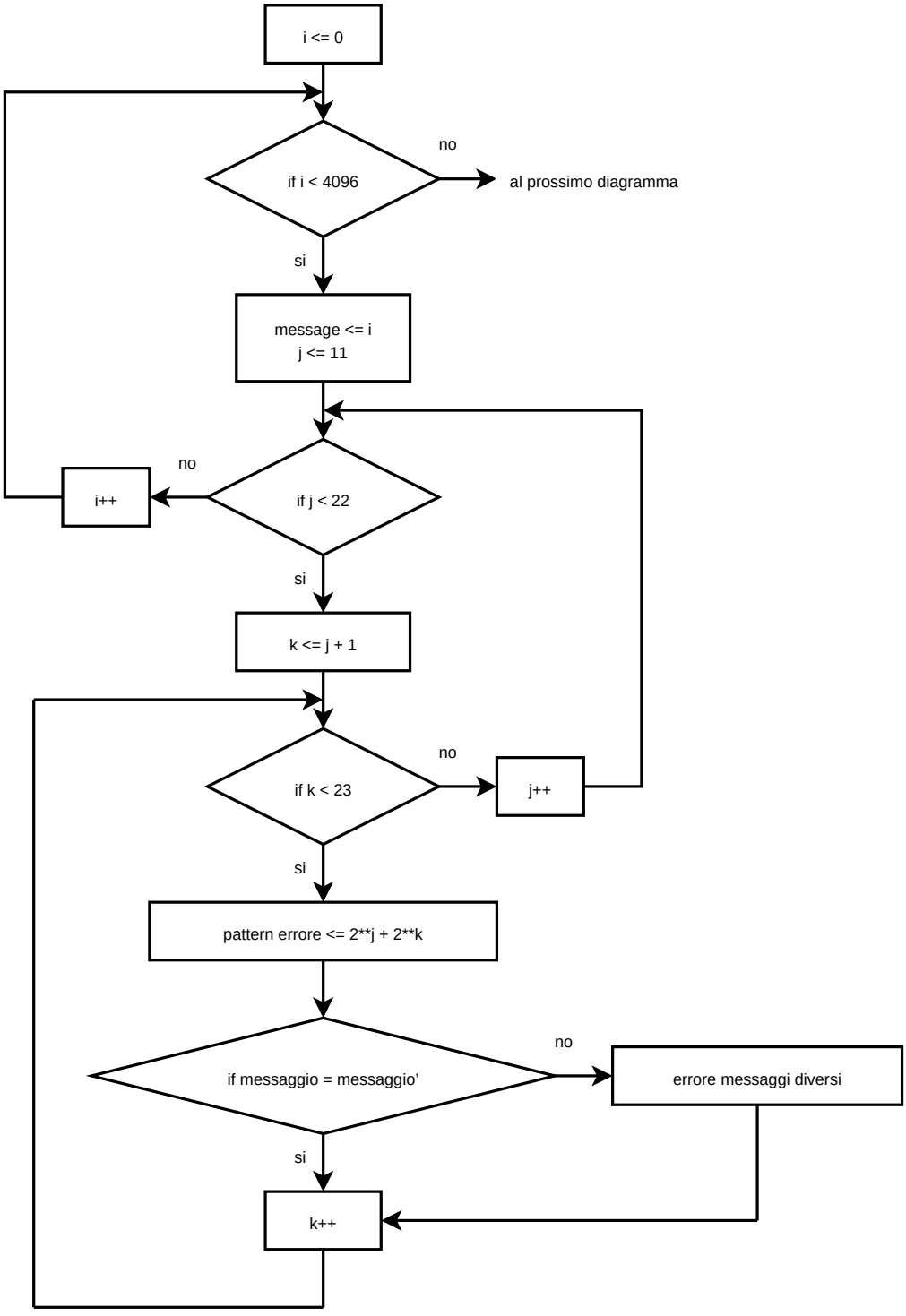

Figura 7.7: Diagramma di flusso test CRC con 2 errori

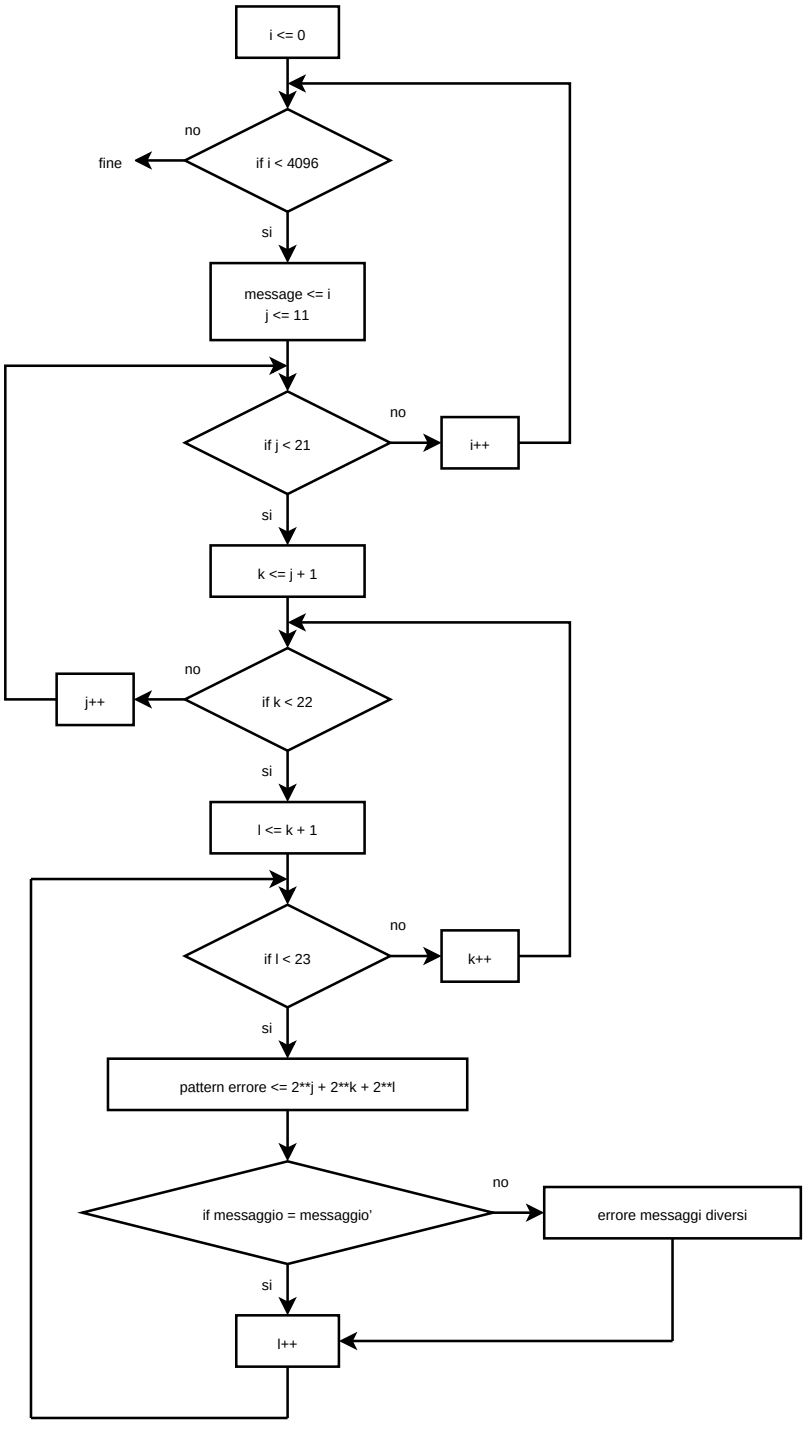

Figura 7.8: Diagramma di flusso test CRC con 3 errori

7.1.3 Test package

In questo package sono definite delle costanti e delle funzioni usate per i test dei due metodi. Non le descriviamo, perchè per la comprensione del loro funzionamento, sono sufficienti i commenti nel codice.

7.2 Dimostratore in Java

Per il dimostratore abbiamo implementato unicamente il test per il generatore del case per la correzione nel CRC, che si trovano nell'appendice D.2.4 a pagina 504.

Capitolo 8

Conclusioni

Eccoci giunti al termine di questo progetto di semestre. Siamo molto soddisfati del lavoro che abbiamo fatto, perchè ci ha permesso di apprendere molte cose nuove e, di approfondirne altre che avevamo studiato in precedenza. Questo progetto ci ha permesso di apprendere un metodo di lavoro molto simile a quello che dovremmo adottare quando inizieremo ad esercitare la nostra professione. Per esempio, il fatto di dover raccogliere diverso materiale su un certo argomento e di dover ricavare da esso la conoscenza necessaria per lo svolgimento di un dato progetto, sarà un possibile compito che dovremmo essere in grado di fare. In conclusione, vogliamo dire che anche se non siamo riusciti a completare l'implementazione dell'EDAC a causa della mancanza di conoscenza sull'architettura dove dovrà venir implementato e del tempo a disposizione, siamo soddisfati e speriamo che questo progetto venga continuato da altre persone, che possano sfruttare quanto abbiamo prodotto.

Bibliografia

[1] Robert H. Morelos-Zaragoza, *The Art of Error Correcting Coding*, Wiley, 2002, Edizione Aprile 2002

[2] Erozan M. Kurtas and Bane Vasic, *Advanced Error Control Techniques for Data Storage Systems*, CRC and Taylor & Francis, 2006

[3] *http://www.eccpage.com*

[4] *http://www.wikipedia.org*

[5] *http://www.stefanvhdl.com/vhdl/html/*

Appendice A

Introduzione ai codici per la correzione degli errori

A.1 Introduzione generale

In molti campi della tecnica, c'è necessità di trasmettere delle informazioni in formato digitale (bit). Nei canali di trasmissione, purtroppo è quasi sempre presente del rumore oppure intervengono dei disturbi esterni, che alterano il flusso dei dati trasmesso. Per ovviare a questo problema, vengono utilizzati i *codici per la correzione degli errori*. In sostanza questi codici calcolano a partire dall'informazione che vogliamo trasmettere (da qui in avanti chiamata *messaggio originale*), un certo numero di bit (da qui in avanti chiamati *ridondanza*) da aggiungere ad essa. La *ridondanza* servirà al momento della ricezione per detettare e (in certi casi) correggere eventuali errori intervenuti nel canale di trasmissione.
L'insieme di tutti i *messaggi originali* + le rispettive *ridondanze* viene chiamato *codice per la correzione degli errori*. Un *codice per la correzione degli errori* è composto da parole (da qui in avanti chiamate *codeword*).
I simboli del *messaggio originale*, della *ridondanza* e della *codeword* sono numeri che possono assumere i valori 0 o 1. L'insieme $\{0, 1\}$ è definito come, $GF(2^1)$ [1].
Esistono due tipi di codici per la correzione degli errori:

- **codici a blocco** il *messaggio originale* viene suddiviso in blocchi di k bit, successivamente codificati in *codewords* di n bit.
- **codici convoluzionali** il *messaggio originale* viene preso globalmente e letto mediante una finestra che scorre su di esso nell'ordine della codifica.

[1] campo di Galois di ordine 2 (i campi di Galois li spieghiamo nella sezione B.2 a pagina 72)

In quest'appendice trattiamo solamente i *codici a blocco*.
La fase di codifica, viene rappresentata mediante la seguente espressione.

$$f : GF(2^k) \rightarrow GF(2^n)$$

La funzione f, è la procedura di codifica. L'insieme dei 2^k vettori codificati di lungehzza n è chiamato *codice* di lunghezza n e dimensione k, viene espresso come codice $[n, k]$. Il rapporto k/n viene chiamato *code rate*.
La *distanza di Hamming* o distanza minima è un numero che rappresenta, considerando due vettori di ugual dimensione, in quante posizioni essi differiscono (es. $\{001, 000\}$ distanza di Hamming 1). A questo punto nella definizione di un codice viene inserita anche la *distanza di Hamming* tra le *codeword*. Il codice viene quindi definito $[n, k, d]$. Il numero massimo di errori che può correggere un codice, lo si può calcolare con la seguente formula.

$$e_c = \frac{d-1}{2}$$

Il numero massimo di errori che può detettare un codice, lo si può calcolare con la seguente formula.

$$e_d = d - 1$$

A.2 Errori nel campo delle trasmissioni

Abbiamo voluto inserire questa sezione nell'appendice, perchè ci è sembrato importante mostrare un campo d'applicazione in cui l'uso dei codici per la correzione degli errori è fondamentale.
Un generico sistema di comunicazione può essere schematizzato, secondo la teoria dell'informazione, con un modello costituito da 5 parti:

una sorgente di informazione: un'entità logica o fisica (persona o dispositivo) che genera messaggi o sequenze di messaggi che saranno inviati ad una destinazione; i messaggi possono essere di svariata natura;

un trasmettitore: dispositivo che trasforma i messaggi in segnali adatti ad essere trasmessi sul canale; tale trasformazione può essere semplicemente una trasduzione oppure una serie di operazioni quali quelle di codifica e modulazione, a dipendenza del tipo di canale;

un canale: mezzo (es. linea di rame, fibra, etere, ...) utilizzato per trasferire il segnale dal trasmettitore al ricevitore. In genere sul canale sono presenti disturbi (rumore) che possono corrompere il segnale trasmesso fino ad impedirne la corretta ricezione da parte del ricevitore;

un ricevitore: dispositivo che generalmente esegue le stesse operazioni del trasmettitore, ma in ordine inverso, e ,in aggiunta, decide in base a quanto ricevuto quale messaggio possa essere stato realmente trasmesso;

una destinazione: un'entità (persona o dispositivo) alla quale è indirizzato il messaggio trasmesso.

La trasmissione di un messaggio da un canale ad un altro, può essere "infettata" da errori dipendenti da condizioni anomali. Molto spesso, la causa degli errori può essere attribuita a disturbi (rumore). Per questo motivo è importante disporre di metodi che consentano di rilevare automaticamente la presenza di errori in un messaggio. In tal caso, un modo di ovviare alle conseguenze dei disturbi può essere quello di ripetere la trasmissione del messaggio, nella speranza di essere più fortunati. La possibilità di rilevare automaticamente la presenza di un errore nel messaggio ricevuto consiste nella presenza di regole, che sono inserite nel messaggio originario e sono conosciute dal fatto che esse sono note a priori a chi riceve il messaggio. I disturbi possono provocare l'infrazione di tali regole e pertanto mettere in luce così la presenza di errori. La presenza di un errore può essere rilevata semplicemente analizzando il messaggio ricevuto e mettendo in luce l'infrazione di una o più regole. Spesso però gli errori si presentano in "gruppi", in quanto provocati da disturbi che durano nel tempo. Per esempio si pensi ad un disturbo di natura elettrica. Un modo molto semplice di essere ragionevolmente sicuri che il messaggio ricevuto sia effettivamente quello corretto trasmesso dal mittente, è quello di trasmetterlo più volte e scegliere quello ricevuto maggiormente. Una soluzione sicuramente meno onerosa è quella di codificare il messaggio originario impiegando un codice a rivelazione di errore e richiederne la ritrasmissione solo in caso di errore. Oppure avere un codice in grado di correggere automaticamente il messaggio errato. Esiste un numero molto vasto di codici rilevatori e correttori di errore.

Appendice B

Aritmetica modulare e campi finiti

B.1 Aritmetica modulare

Come introduzione all'aritmetica modulare, facciamo il seguente esempio: prendiamo due numeri, 17 e 47 e li dividiamo per 6.

$$17 : 6 = 2 \quad \text{resto} \quad 5$$

$$47 : 6 := 7 \quad \text{resto} \quad 5$$

In questo caso si dice che 17 è *congruente* a 47 in modulo 6, in altre parole 17 e 47 hanno lo stesso resto se divisi per 6.

$$17 \equiv 47 \pmod 6$$

A questo punto possiamo prendere tutti i numeri che divisi per 6 danno resto 5 e raggrupparli in un'insieme chiamato *classe di resto*.

$$\overline{5} = \{\ldots, 17, \ldots, 47, \ldots\}$$

L'insieme delle 6 classi di resto (insieme finito), viene rappresentato nel seguente modo:

$$\mathbb{Z}_6 = \{\overline{0}, \overline{1}, \overline{2}, \overline{3}, \overline{4}, \overline{5}\}$$

ogni suo elemento è un sottoinsieme di $\mathbb{Z}$.
Facciamo qualche esempio di aritmetica *modulo 6*.

$$14 + 27 = 41 \Leftrightarrow \overline{2} + \overline{3} = \overline{5}$$

Come si può notare, sommando due numeri, la classe di resto del risultato è uguale alla somma delle classi di resto dei due numeri.

+	0	1	2	3	4	5
0	**0**	1	2	3	4	5
1	1	2	3	4	5	**0**
2	2	3	4	5	**0**	1
3	3	4	5	**0**	1	2
4	4	5	**0**	1	2	3
5	5	**0**	1	2	3	4

∗	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	**1**	2	3	4	5
2	0	2	4	0	2	4
3	0	3	0	3	0	3
4	0	4	2	0	4	2
5	0	5	4	3	2	**1**

Tabella B.1: Tavole somma e moltiplicazione *modulo 6*

Queste due tavole mostrano in grassetto: per la somma gli 0 (elemento neutro della somma), per la moltiplicazione gli 1 (elemento neutro della moltiplicazione). Si può notare come $2 + 4 = 0$, quindi il 4 è l'inverso di 2 (-2 nell'insieme $\mathbb{Z}$). La tabella della moltiplicazione, mostra come non esiste la divisione per tutti i *moduli*, perchè, per esempio in questo caso (*modulo 6*), non tutti i numeri hanno un'inverso.
Nella tabella seguente, si può vedere che i numeri che hanno un'inverso in *modulo 6* sono quelli che hanno il *massimo comun divisore* con 6, uguale a 1.

∗	0	1	2	3	4	5	MCD
0	0	0	0	0	0	0	0
1	0	**1**	2	3	4	5	**1**
2	0	2	4	0	2	4	2
3	0	3	0	3	0	3	3
4	0	4	2	0	4	2	2
5	0	5	4	3	2	**1**	**1**

Tabella B.2: Tavola moltiplicazione *modulo 6* con *MCD*

Nel caso che usiamo l'aritmetica *modulo 7*, esistono tutti gli inversi della moltiplicazione, come mostrato nella seguente tabella.

$*$	0	1	2	3	4	5	6	MCD
0	0	0	0	0	0	0	0	0
1	0	**1**	2	3	4	5	6	1
2	0	2	4	6	**1**	3	5	1
3	0	3	6	2	5	**1**	4	1
4	0	4	**1**	5	2	6	3	1
5	0	5	3	**1**	6	4	2	1
6	0	6	5	4	3	2	**1**	1

Tabella B.3: Tavola moltiplicazione *modulo 7* con *MCD*

B.1.1 Definizione di corpo (copiata da wikipedia)

Un **corpo** è un insieme K munito di due operazioni binarie, che chiamiamo **somma** e **prodotto** e indichiamo rispettivamente con $+$ e $*$, che godono delle seguenti proprietà:

- somma
  - Per ogni coppia di elementi a, b appartenenti a K, la loro somma $a + b$ appartiene a K; si dice che K è *chiuso* rispetto alla somma.
  - La somma è *associativa*; cioè per ogni terna di elementi a, b, c appartenenti a K, vale: $(a + b) + c = a + (b + c)$.
  - Esiste un unico elemento z appartenente a K *neutro* ripetto alla somma, cioè tale che $a + z = z + a = a$.
  - Per ogni elemento a di K esiste un elemento *opposto* b tale che $a + b = b + a = z$.
  - La somma è *commutativa*, cioè per ogni coppia di elementi a, b di K, vale: $a + b = b + a$.
- prodotto
  - Per ogni coppia di elementi a, b appartenenti a K, il loro prodotto $a \cdot b$ appartiene a K; si dice che K è *chiuso* rispetto al prodotto.
  - Il prodotto è *associativo*; cioè per ogni terna di elementi a, b, c appartenenti a K, vale: $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
  - Esiste un unico elemento e (diverso da z) appartenente a K *neutro* ripetto al prodotto, cioè tale che $a \cdot e = e \cdot a = a$.
  - Per ogni elemento a diverso da z esiste un elemento inverso b tale che $a \cdot b = e$.
- Somma e prodotto godono delle proprietà distributive, cioè per ogni terna a, b, c di elementi di K vale: $a \cdot (b + c) = a \cdot b + a \cdot c$.

Generalmente, si indica con 0 l'elemento neutro della somma (z) e con 1 l'elemento neutro del prodotto(e).
In un corpo sono risolubili in modo unico le equazioni $a \cdot x = b$, $x \cdot a = b$ per ogni a, b appartenenti a K con a diverso da 0.
Un corpo è un anello con unità in cui esiste l'inverso di ogni numero diverso da 0 (proprietà B3). Un corpo commutativo, in cui valga cioè la proprietà commutativa del prodotto, è un campo.

B.2 Campi di Galois (Galois Fields)

I *GF* sono dei corpi finiti, contenenti elementi composti dai coefficienti di una delle possibili "combinazioni" del polinomio generatore. Essi vengono rappresentati nel seguente modo.

$$GF(p^n)$$

La base p è un numero primo, mentre $n-1$ indica il grado del polinomio generatore del campo ed il numero di coefficienti di un'elemento. La quantità di elementi che compongono il campo, la si calcola con la seguente formula.

$$n_{elementi} = p^n$$

Il polinomio generatore, viene rappresentato nel seguente modo.

$$g(x) = a_{n-1} \cdot x^{n-1} + a_{n-2} \cdot x^{n-2} + \ldots + a_1 \cdot x^1 + a_0 \cdot x^0$$

$$a_{n-1}, \ldots, a_0 \in \mathbb{Z}_p$$

Quando lavoriamo con i *GF*, possiamo fare tutte le operazioni ereditate dall'insieme $\mathbb{Z}$, in particolare ci interessano l'addizione e la moltiplicazione.
L'addizione di due elementi, consiste nel sommare i due vettori contenenti i coefficienti dei polinomi che li rappresentano in *modulo p*.
Prima di spiegare come si fa la moltiplicazione, dobbiamo scegliere un polinomio irriducibile di grado n.

$$p(x) = x^n + c_{n-1} \cdot x^{n-1} + \ldots + c_0 \cdot x^0$$

$$c_{n-1}, \ldots, c_0 \in \mathbb{Z}_p$$

Il risultato della moltiplicazione, è l'elemento rappresentato dal resto della divisione polinomiale fra la moltiplicazione dei due polinomi (che rappresentno gli elementi) ed il polinomio irriducibile.

$$(a(x) \cdot b(x)) \bmod p(x)$$

Per chiarire il tutto facciamo il seguente esempio prendendo un

$$GF(2^2)$$

Questo campo è composto da 4 elementi, il polinomio generatore è di grado 1 ed i coefficienti del polinomio generatore appartengono all'insieme $\mathbb{Z}_2 = \{0, 1\}$.

$$g(x) = a_1 \cdot x^1 + a_0 \cdot x^0 = a_1 \cdot x + a_0$$

Gli elementi del campo sono i seguenti:

$$GF(2^2) = \{0 \cdot x + 0, 0 \cdot x + 1, 1 \cdot x + 0, 1 \cdot x + 1\}$$

$$GF(2^2) = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$$

Definiamo per comodità, gli elementi, nel seguente modo:

$$GF(2^2) = \{\underline{0}, \underline{1}, \underline{2}, \underline{3}\}$$

L'elemento $\underline{0}$ è quello neutro della somma, mentre il secondo ($\underline{1}$) è quello neutro della moltiplicazione. Come per l'aritmetica modulare vista in precedenza, possiamo scrivere le tavole della somma e della moltiplicazione in $GF(2^2)$.

$+$	$\underline{0}$	$\underline{1}$	$\underline{2}$	$\underline{3}$
$\underline{0}$	$\underline{0}$	$\underline{1}$	$\underline{2}$	$\underline{3}$
$\underline{1}$	$\underline{1}$	$\underline{0}$	$\underline{3}$	$\underline{2}$
$\underline{2}$	$\underline{2}$	$\underline{3}$	$\underline{0}$	$\underline{1}$
$\underline{3}$	$\underline{3}$	$\underline{2}$	$\underline{1}$	$\underline{0}$

Tabella B.4: Tavola somma $GF(2^2)$

A questo punto scegliamo il polinomio irriducibile da usare per la moltiplicazione.

$$p(x) = x^2 + x + 1$$

Il seguente calcolo mostra come costruire gli elementi della tavola della moltiplicazione.

$$\underline{2} \cdot \underline{3} = (x \cdot (x + 1)) \bmod (x^2 + x + 1) \Rightarrow \bmod = 1 \Rightarrow \underline{1}$$

$*$	$\underline{0}$	$\underline{1}$	$\underline{2}$	$\underline{3}$
$\underline{0}$	$\underline{0}$	$\underline{0}$	$\underline{0}$	$\underline{0}$
$\underline{1}$	$\underline{0}$	$\underline{1}$	$\underline{2}$	$\underline{3}$
$\underline{2}$	$\underline{0}$	$\underline{2}$	$\underline{3}$	$\underline{1}$
$\underline{3}$	$\underline{0}$	$\underline{3}$	$\underline{1}$	$\underline{2}$

Tabella B.5: Tavola moltiplicazione $GF(2^2)$

La base p del campo è un'elemento primitivo, infatti come si può notare nel seguente esempio essa può generare tutto il campo.

$$2^0 = \underline{1}$$

$$2^1 = \underline{2}$$

$$2^2 = \underline{3}$$

I seguenti calcoli, mostrano come sia possibile costruire tutti gli elementi del campo con un elemento primitivo.

$$2^3 = 2 \cdot 2^2 = \underline{2} \cdot \underline{3} = 6 = \underline{1}$$

$$2^4 = 2^2 \cdot 2^2 = \underline{3} \cdot \underline{3} = 9 = \underline{2}$$

$$\dots$$

Ogni $GF(p^n)$ ha degli elementi primitivi.