Specifica del Progetto di Laboratorio
Architettura degli Elaboratori I

Ivan Masnari
matricola: 909607, Turno: B
ivan.masnari@studenti.unimi.it

1 Descrizione Generale

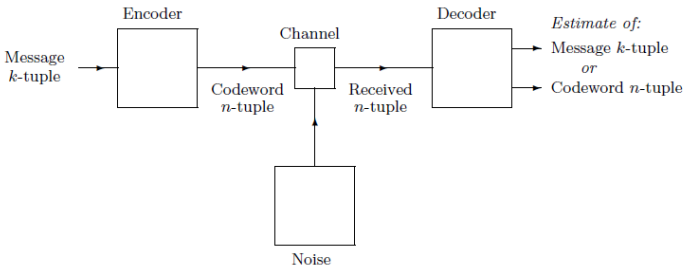

Figura 1: modello di comunicazione secondo Shannon.

Il progetto avrà come suo scopo la realizzazione di un modello di comunicazione attraverso un canale *non-fedele*, come in Figura 1. Ciò che distingue un canale *non-fedele* da uno *fedele* è la presenza di interferenze che alterano la composizione della parola trasmessa. La scelta progettuale di dotare il mio modello di un canale *non-fedele* mi da l'opportunità di mostrare alcune proprietà interessanti della codifica scelta per i messaggi inviati dall'utente. La principale caratteristica che la miae codifica dovrà soddisfare sarà quella di permetterci, nonostante gli artefatti dovuti alle interferenze, una comunicazione quanto più efficiente e affidabile. Per essere efficiente, il trasferimento dell'infromazione deve avvenire in breve tempo senza un uso proibitivo delle risorse. Per essere affidabile, invece, il dato ricevuto dopo la trasmissione deve essere quanto più somigliante al messaggio inviato. Il tentativo di conciliare queste due esigenze ha dato vita alla moderna *Coding Theory*[1], fondata da Claude Shannon e Richard Hamming nella prima metà del secolo scorso. Il modello qui proposto ha due scopi: il primo è quello di mostrare che i due *desiderata* vengono assolti, il secondo è quello di dare corpo a teorie considerate astratte. Il progetto sarà composto da più unità funzionali:

1. Encoder

2. Canale *non-fedele*

3. Decoder

1.1 Encoder

L'encoder permette l'aggiunta di 3 bit di controllo al messaggio di 4 bit scelto dall'utente. La scelta dei 3 bit e la loro posizione non è affatto casuale. Per gusto antiquario, ho scelto di utilizzare la codifica $C[=4/7]$ introdotta da Shannon nel suo paper sull'argomento[2]. Questi bit aggiuntivi non modificano il contenuto informativo del messaggio, ma permettono di identificare al più due errori di trasmissione e di correggerne al più uno. Per questo motivo il codice qui utilizzato viene detto *1-correcting-code*.

1.2 Canale *non-fedele*

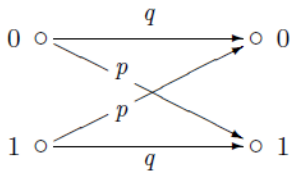

Figura 2: Canale binario simmetrico.

Il canale è il mezzo di propagazione dell'informazione. Solitamente si assume che il canale sia un ambiente non-cooperativo (se non apertamente ostile). Un esempio importante è fornito dal *canale simmetrico m-ario*, così chiamato in quanto ha in input e output un alfabeto di m simboli. Il canale è caratterizzato da un singolo parametro p, che identifica la probabilità che dopo la trasmissione di un simbolo x_i venga ricevuto il simbolo x_j, dove $j \neq i$. In simboli,

$$p = \text{Prob}(x_i \mid x_j), \text{ per } j \neq i.$$

Correlato a questa avremo la probabilità

$$s = (m-1)p$$

che, dopo averlo trasmesso, x_j non venga ricevuto correttamente e la probabilità

$$q = 1 - s = 1 - (m-1)p$$

che, invece, x_j sia ricevuto correttamente. Si definisce mSC(p) l'm-ario canale simmetrico con *probabilità di transizione p*. Il canale è detto simmetrico nel senso che Prob($x_i \mid x_j$) non dipende dai particolari valori di x_i e x_j, ma solo dal fatto che non siano uguali. Data la natura binaria del segnale, nel nostro progetto saremo interessati in particolare al 2-ario canale simmetrico(Figura 2), altrimenti chiamato BSC(p), dove $p = s$.
Assumo, inoltre, che il canale di trasmissione sia capace di operare con una discreta accuratezza. Ovvero, assumo che sebbene degli errori nella trasmissione siano possibili, questi errori non saranno troppo gravi. Altrimenti, il problema sarebbe più di design che di codifica. Per il nostro BSC(p) questa ipotesi implica che un pattern di errori composto di un piccolo numero di simboli è più probabile di uno con un grande numero di simboli.

1.3 Decoder

Il decoder è il device che ha la funzione di ricevere il segnale nella forma di una n-tupla dal canale, di discernere se sono avvenuti errori nella trasmissione ed, eventualmente, di correggerli. Suppongo di sapere che per ogni tupla $\mathbf{y}$ e ogni parola appartenente al nostro codice $\mathbf{x}$ la probabilità Prob($\mathbf{y} \mid \mathbf{x}$) sia la probabilità di ricevere $\mathbf{y}$ dopo che $\mathbf{x}$ è stato trasmesso. Il nostro decoder dovrà, quindi, ricercare la *codeword* $\mathbf{x}$ tale che Prob($\mathbf{y} \mid \mathbf{x}$) venga massimizzata. Questo genere di decoding viene spesso chiamato **Maximum Likelihood Decoding** (abbreviato MLD) ed è una forma di *decoding completo*: dobbiamo essere sempre essere in grado di decodificare il messaggio con una adeguata *codeword*.
Tuttavia, questa operazione può non risultare sempre possibile, in quanto, magari, il messaggio risulta troppo danneggiato. Per ovviare al problema indebolisco la richiesta di MLD nel seguente modo: data una particolare tupla, il decoder restituirà o una codeword o un particolare simbolo $\propto$ che può essere inteso come "sono avvenuti errori nella trasmissione che non possono essere corretti".

2 Implementazione

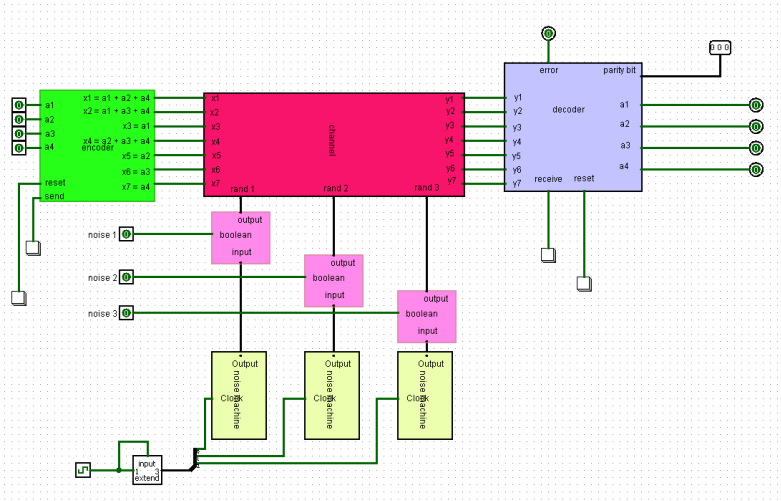

Figura 3: Implementazione del progetto

Nell'implementazione su Logisim ho cercato di aderire, per quanto possibile, all'immagine proposta in Figura 1. Ogni blocco individuato da Shannon nel modello di comunicazione ha una sua controparte funzionale nel mio progetto. Non sorprende, quindi, che questa sezione riprodurrà nella forma la sezione precedente, fatto salvo il maggior dettaglio dato agli aspetti di costruzione dei singoli moduli.

2.1 Encoder

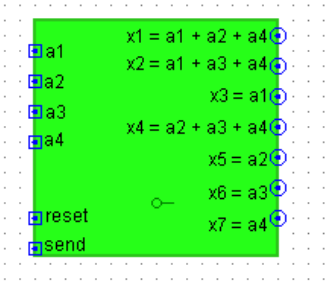

Figura 4: Encoder

Input:

- a_i t.c. $i \in \{1,2,3,4\}$:= bit che veicolano l'informazione.
- *send* := segnale di invio della trasmissione.

- *reset* := segnale di annullamento.

Output:

- x_i t.c. $i \in \{1,2,4\}$:= parity-check bit.
- x_i t.c. $i \in \{3,5,6,7\}$:= bit che veicolano l'informazione.

Funzione:
Questo modulo si occupa di trasformare il vettore **a** di 4 bit in input in un vettore **x** di 7 bit. La sua azione può essere considerata algebricamente identica al prodotto del vettore in input con una matrice detta *matrice generatrice*, che chiameremo **G**, definita come segue:

$$\mathbf{G} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

A livello circuitale, ho ottenuto tale funzionalità mettendo in XOR i bit a_1,a_2,a_4 per ottenere x_1, i bit a_1,a_3,a_4 per ottenere x_2 e i bit a_2,a_3,a_4 per ottenere x_4. I rimanenti bit del vettore di output, essendo gli effettivi bit di informazione, sono direttamente collegati al vettore di input. Inoltre, ho aggiunto dei Latch S-R come gate per permettere all'utente di lanciare il segnale attraverso l'apposito pulsante *send* quando correttamente formulato. Il segnale di *reset* riporta a zero tutti i Latch.

2.2 Canale *non-fedele*

2.2.1 Canale

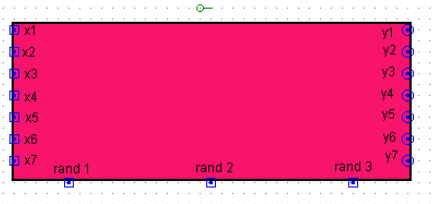

Figura 5: Canale

Input:

- x_i t.c. $i \in \{1,2,3,4,5,6,7\}$:= vettore inviato (privo di errori).
- $rand_i$ t.c. $i \in \{1,2,3\}$:= vettori di errore.

Output:

- y_i t.c. $i \in \{1,2,4\}$:= vettore ricevuto (potenzialmente affetto da errori).

Funzione:

La funzione di questo modulo è triviale: ritrasmettere il messaggio dall'encoder al decoder. Tuttavia, se uno (o più) dei vettori *rand* è un vettore non nullo, si produrrà un (o più) errore nel messaggio trasmesso in output.

2.2.2 Gate

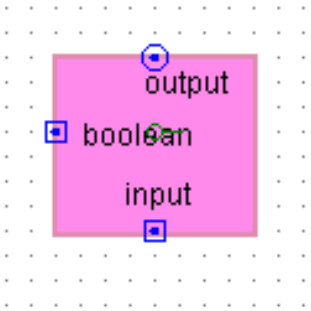

Figura 6: Gate

Input:

- *input* := vettore in input.
- *boolean* := segnale di check.

Output:

- *output* := vettore di output.

Funzione:
Se il segnale *boolean* è 0, allora **output** sarà uguale al vettore nullo. Altrimenti, **output** = **input**. Questo componente permette all'utente di decidere quanti segnali di errore permettere in una data esecuzione.

2.2.3 Macchina del rumore

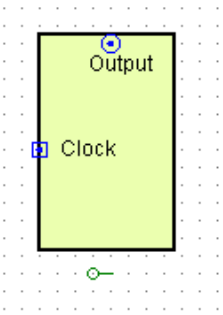

Figura 7: Macchina del Rumore

Input:

- *clock* := segnale di clock.

Output:

- *output* := vettore di output.

Funzione:
La macchina del rumore è di gran lunga il modulo più complesso. La sua funzione è quella di produrre un vettore unario **e** random. Siccome, sviluppando in Logisim, non abbiamo accesso alla "vera" casualità, dobbiamo sfruttare le risorse del sistema per crearne una ragionevole simulazione. A questo scopo ho utilizzato i cosiddetti LSFR, ovvero *linear feedback shift register*, che sono una tipologia di registri di traslazione i cui dati in ingresso sono prodotti da una funzione lineare dello stato interno. Le uniche funzioni lineari di singoli bit sono lo XOR e lo XNOR; perciò è un registro di traslazione i cui bit in ingresso sono prodotti dall'OR esclusivo (XOR) di alcuni bit memorizzati all'interno dei registri. In particolare, ho sfruttato questi LSFR in due modi: da una parte, un LSFR da 16 bit (Fig.8) produce ad ogni colpo di clock un bit pseudo-random, dall'altra, altri tre LSFR da 3 bit ciascuno(Fig.9), concorrono a specificare i tre bit di selezione di un demultiplexer(Fig.10) che direzionerà il bit pseudo-random su uno delle sette posizioni del vettore **output**. Tale architettura mi è stata suggerita da un articolo trovato su Internet[3].

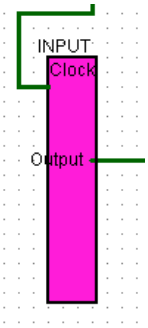

Figura 8: LSFR 16 bit

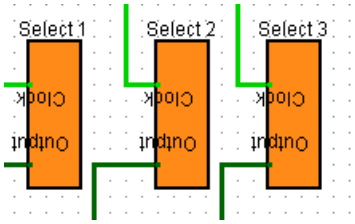

Figura 9: LSFR 3 bit

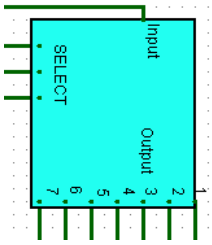

Figura 10: Demultiplexer

Per ottenere vettori errore realmente randomici, ho dovuto avere un'attenzione specifica riguardo alla struttura interna degli LSFR. Per prima cosa, i valori dei registri che concorrono a produrre il bit di ingresso non sono scelti a caso, ma secondo un pattern che massimizza la lunghezza della sequenza di bit ad ogni aggiornamento dei registri[4] (nel caso di 16 bit, 2^{15}). In secondo luogo, il fatto che la posizione dove il bit generato pseudo-randomicamente va a sommarsi al vettore nullo viene a sua volta generata pseudo-randomicamente permette di eliminare la linearità (e quindi la predicibilità) intrinseca alla struttura degli LSFR.

2.3 Decoder

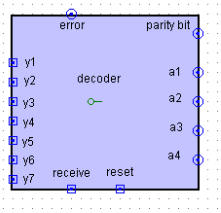

Figura 11: Decoder

Input:

- y_i t.c. $i \in \{1,2,3,4,5,6,7\}$:= vettore inviato + possibili errori.
- *receive* := segnale di ricevuta trasmissione.
- *reset* := segnale di reset.

Output:

- a_i t.c. $i \in \{1,2,3,4\}$:= bit che veicolano l'informazione.
- *error* := segnale di errore.

- *parity* := 3 bit di parità.

Funzione:
Il decoder svolge due funzioni principali: rileva l'errore nella trasmissione e cerca di correggerlo. Per fare questo il modulo sfrutta la natura lineare della codifica del messaggio. Il primo sottomodulo in Figura 12, infatti, opera il controllo di parità moltiplicando il vettore di entrata **y** per la cosiddetta *matrice di parità* **H**, così definita:

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Il vettore **z** ottenuto viene comunemente chiamato vettore **sindrome**. Se tale vettore è il vettore nullo, significa che quella che è stata ricevuta era una parola che apparteneva al codice: il segnale di errore resta a 0 e non avvengono correzioni alla parola in entrata. In caso contrario, supponendo che sia avvenuto un errore su un singolo bit, otterremo un vettore non nullo che verrà passato al secondo sottomodulo (in Figura 13). Il modulo provvederà a identificare il vettore errore e a sommarlo al vettore **y**. Questo avviene perchè algebricamente[5] vale la seguente equazione:

$$\mathbf{Hy} = \mathbf{H}(\mathbf{x} + \mathbf{e}) = \mathbf{Hx} + \mathbf{He} = 0 + \mathbf{He} \tag{1}$$

Non è difficile mostrare che solo gli errori di signolo bit possono essere corretti. Alternativamente, il codice può rilevare al più due errori semplicemente notando che il segnale di errore è a 1. In Figura 14 un esempio di errore occorso su due bit che non può essere corretto dal nostro codice, ma solo rilevato.

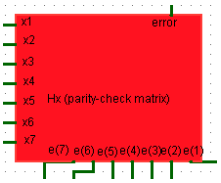

Figura 12: Controllo di parità

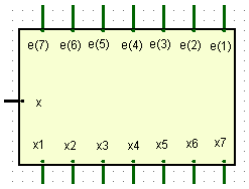

Figura 13: Correzione dell'errore

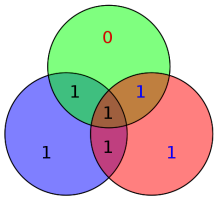

Figura 14: Esempio di errore su due bit

3 Bibliografia

[1] Hall J. I., *Notes on Coding Theory*, (2010)
[2] Shannon C. E., *A Mathematical Theory of Communication*, (1948)
[3] Sharma D., Khalid A., Parashar S., *Cryptographically Secure Linear feedback shift Register*, in International Journal of Advanced Research in Computer Engineering & Technology (IJARCET), Volume 3 Issue 10, October 2014
[4] Goresky M., Klapper A., *Algebraic Shift Register Sequences*, (2009)
[5] Pinter C.C., A Book of Abstract Algebra, (1990)