Prova Finale di Reti Logiche
Prof. Gianluca Palermo - Anno 2021/2022
Matteo Luigi Giovanni Rigat (Codice Persona 10633610 - Matricola 909204)

POLITECNICO
MILANO 1863

Ingegneria Informatica
Corso di Laurea Triennale - Milano Leonardo

Introduzione

I codici convoluzionali vengono utilizzati per ottenere un trasferimento affidabile di dati in applicazioni quali la trasmissione di video digitale, radio, telefonia mobile e comunicazioni via satellite.

Il processo di codifica consiste nell'aggiungere bit di ridondanza al messaggio che si vuole trasmettere.
In fase di ricezione, la presenza di tali bit permette di rilevare e correggere eventuali errori introdotti nel messaggio dal rumore presente sul canale.

La ridondanza però causa un peggioramento dell'efficienza di trasmissione e quindi si rende necessario un compromesso tra la diminuzione della probabilità di errore e la riduzione dell'efficienza di trasmissione.

Il modulo da implementare deve leggere la sequenza da codificare da una memoria con indirizzamento al byte: ogni singola parola è un byte.
La quantità di parole W da codificare è memorizzata nell'indirizzo 0 della RAM.
I byte della sequenza W sono memorizzati dall'indirizzo 1.
La sequenza di byte è trasformata nella sequenza di bit da elaborare, su questo flusso viene applicato il codice convoluzionale (ogni bit viene codificato con 2 bit).
Lo stream di uscita Z (Z=2W) viene poi memorizzato a partire dall'indirizzo 1000 (mille).

Esempio1: (Sequenza lunghezza 2)
W: 10100010 01001011
Z: 11010001 11001101 11110111 11010010

INDIRIZZO MEMORIA	VALORE	COMMENTO
0	2	\\ Byte lunghezza sequenza di ingresso
1	162	\\ primo Byte sequenza da codificare
2	75	
[...]		
1000	209	\\ primo Byte sequenza di uscita
1001	205	
1002	247	
1003	210	

Il convolutore è una macchina sequenziale sincrona, in particolare una macchina di Mealy, in cui l'uscita dipende sia dallo stato sia dall'ingresso.
Il funzionamento del processo di codifica è raffigurato nella seguente figura 1:

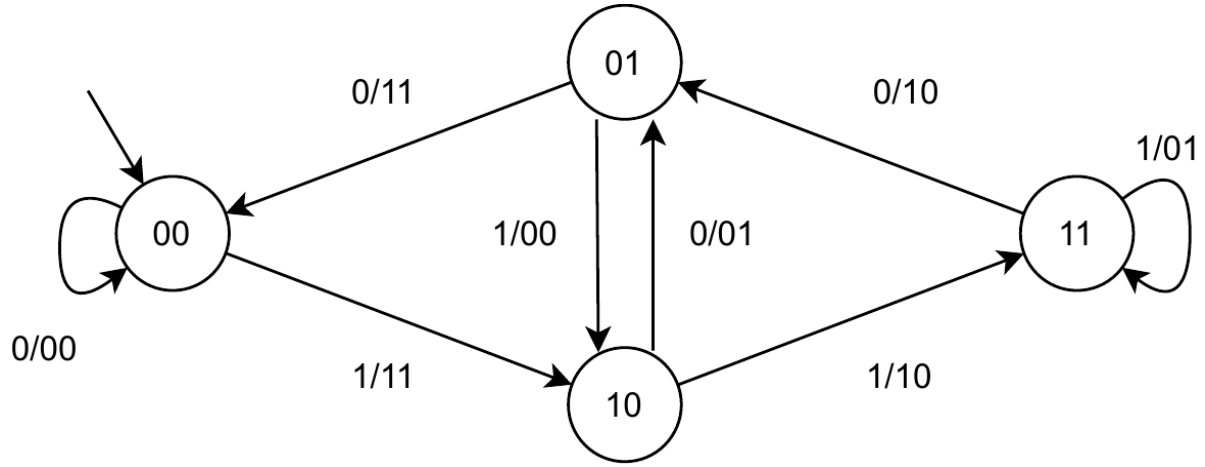

Fig. 1

Architettura

Visto che il processo di codifica consiste in una macchina a stati, ho scelto di inglobare questa in una macchina a stati più grande, comprendente, tra gli altri, gli stati di reset, lettura dei valori e stampa della codifica.

La macchina a stati che ho progettato, descritta nel dettaglio poco più avanti, è la seguente.

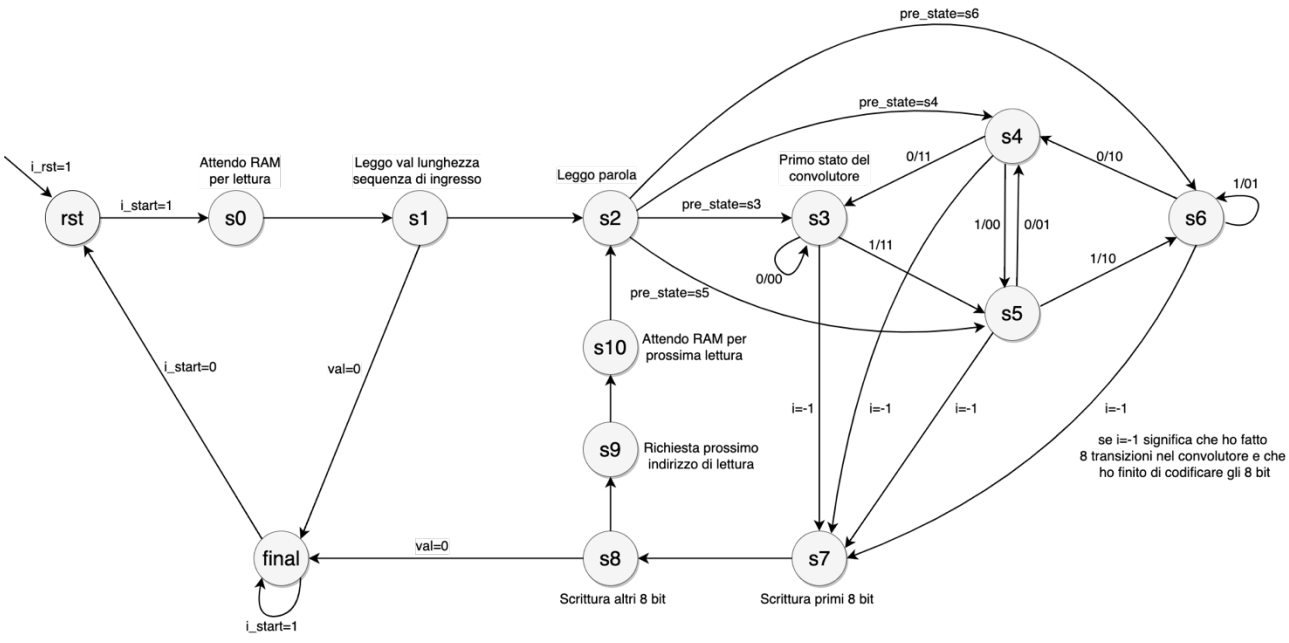

Fig. 2

Come si può notare, ci sono stati che possono essere accorpati; tuttavia, per una maggiore chiarezza del progetto, ho deciso di lasciare gli stati come mostrato in figura 2.

Inoltre, la macchina è stata progettata in modo che in qualsiasi momento, quando si riceve in ingresso il segnale i_rst alto, viene inizializzata e rimane pronta per una nuova codifica (più avanti viene mostrato come caso limite in Risultati Sperimentali).

Descrizione degli stati

- rst (stato di reset)
  1. La macchina ha ricevuto in ingresso il segnale di i_reset, è stata inizializzata per una nuova codifica.
  2. Finché il segnale i_start rimane basso, la macchina rimane in attesa in questo stato.
  3. Appena i_start diventa alto, si è pronti per cominciare una nuova codifica: viene abilitato o_en per poter leggere la lunghezza della sequenza di ingresso (val).

- s0 (attesa RAM)
  1. Stato di attesa della RAM per poter poi leggere da i_data il numero di parole da codificare al ciclo di clock successivo.
  2. Nel frattempo, richiedo l'indirizzo della prima parola che poi leggerò nello stato s2.

- s1 (lettura val)
  1. Posso finalmente leggere val, controllo che non sia uguale a zero, non avrei nessuna parola da codificare, se così fosse passo direttamente allo stato finale.
  2. In questo stato attendo inoltre la RAM affinché mi restituisca la prima parola.

- s2 (lettura parola)
  1. Leggo la parola.
  2. Decremento val così da sapere quando ho finito le parole in ingresso.
  3. Inizializzo le variabili i e j che mi serviranno negli stati del convolutore.
  4. Se è la prima parola che leggo il prossimo stato sarà s3 (di default il primo stato del convolutore) altrimenti il prossimo sarà l'ultimo stato del convolutore che la parola precedente ha visitato (salvato in pre_state).

- Stati del convolutore → s3, s4, s5, s6
  1. Codifico il bit della parola in base alle regole del codice convoluzionale e memorizzo i due bit in un segnale di 16 bit (difatti doppio della parola da 8 bit).
  2. Tengo conto di quanti stati del convolutore ho visitato, se è l'ottavo esco e vado al primo stato di stampa, inoltre salvo lo stato corrente (pre_state) che mi servirà per riprendere la codifica della parola successiva.

- s7 (primo stato di stampa)
  1. Mando in memoria i primi 8 bit della codifica.
  2. Inoltre, incremento di uno l'indirizzo di lettura, mi preparo per la prossima parola in ingresso.

- s8 (secondo stato di stampa)
  1. Mando in memoria gli ultimi 8 bit della codifica facendo attenzione ad incrementare di 1 o_address per poter scrivere al giusto indirizzo della RAM.
  2. Controllo se ho finito le parole in ingresso, infatti ho precedentemente decrementato val; se si vado nello stato finale, altrimenti ritorno allo stato s9 per leggere la parola successiva e ripetere il ciclo.

- s9 (richiesta prossima parola)

1. Mando alla RAM il prossimo indirizzo di ingresso per leggere la prossima parola.
2. Inoltre, incremento di due l'indirizzo di scrittura, mi preparo per le prossime due parole da scrivere in memoria.

- s10 (attendo RAM)
  1. Non succede nulla, serve solo per aspettare un ciclo di clock per poter poi leggere un'altra parola.
  2. Ritorno allo stato s2 (lettura parola).

- final
  1. Ho finito di codificare la sequenza di parole in ingresso, abilito o_done per notificarlo.
  2. Attendo in questo stato finché non ricevo i_start=0, a questo punto abbasso o_done e mi sposto nello stato di reset, pronto per una nuova sequenza da codificare.

Per la realizzazione di questo progetto ho deciso di non dividere i vari moduli e tenere tutto all'interno di un unico processo; in quanto essendo poche le istruzioni presenti in uno stato, ho preferito non separarle in modo da capire meglio cosa succede all'interno di ogni singolo stato in poche righe di codice.

Il componente progettato, con i bus dei segnali in entrata e uscita, è quindi raffigurato nel seguente disegno.

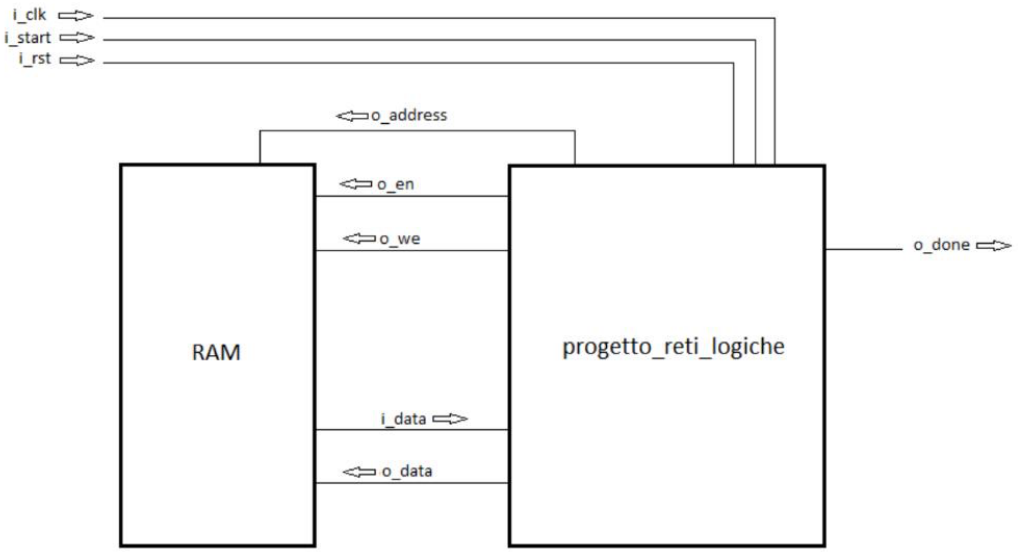

Fig. 3

All'interno di progetto_reti_logiche i segnali utilizzati sono solo quelli indispensabili

```
signal i : integer range -1 to 7;          // Conta gli 8 bit della parola in ingresso per poi finire la codifica se i=-1
signal j : integer range 0 to 15;          // Conta i 16 bit dello stream di uscita salvato in risultato
signal val : integer range 0 to 255;       // Lunghezza sequenza di ingresso

signal parola : std_logic_vector(7 downto 0);          // Salva la parola in ingresso
signal risultato : std_logic_vector(15 downto 0);      // Salva le due parole di uscita
signal in_address : std_logic_vector(15 downto 0);     // Tiene traccia dell'ultimo indirizzo di lettura
signal out_address : std_logic_vector(15 downto 0);    // Tiene traccia dell'ultimo indirizzo di scrittura

signal state, pre_state: state_type;      // State è lo stato corrente, pre_state tiene traccia dell'ultimo stato del
                                             convolutore visitato
```

Risultati sperimentali

Testbench

Dopo aver implementato il componente e testato il suo funzionamento, ho riportato qui i risultati di casi limite in simulazione post sintesi che ritengo rilevanti.

1. Reset

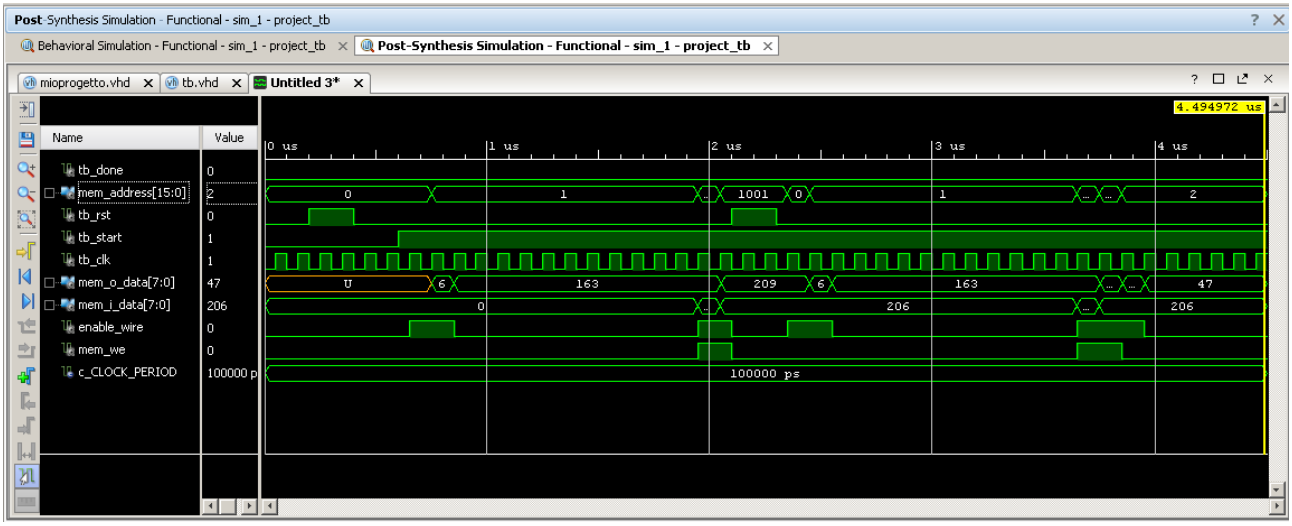

Fig. 4

Come detto, la macchina deve essere in grado di tornare allo stato di reset in qualsiasi momento se viene dato il segnale i_reset=1.

Nella foto sopra riportata si vede infatti che al secondo i_reset, la macchina viene inizializzata: l'indirizzo di lettura viene riportato a 0 mentre quello di scrittura a 1000 (purtroppo nell'immagine per mancanza di spazio vengono mostrati dei puntini) e siccome i_start rimane alto, il componente prosegue poi con una nuova codifica.

2. Sequenza minima

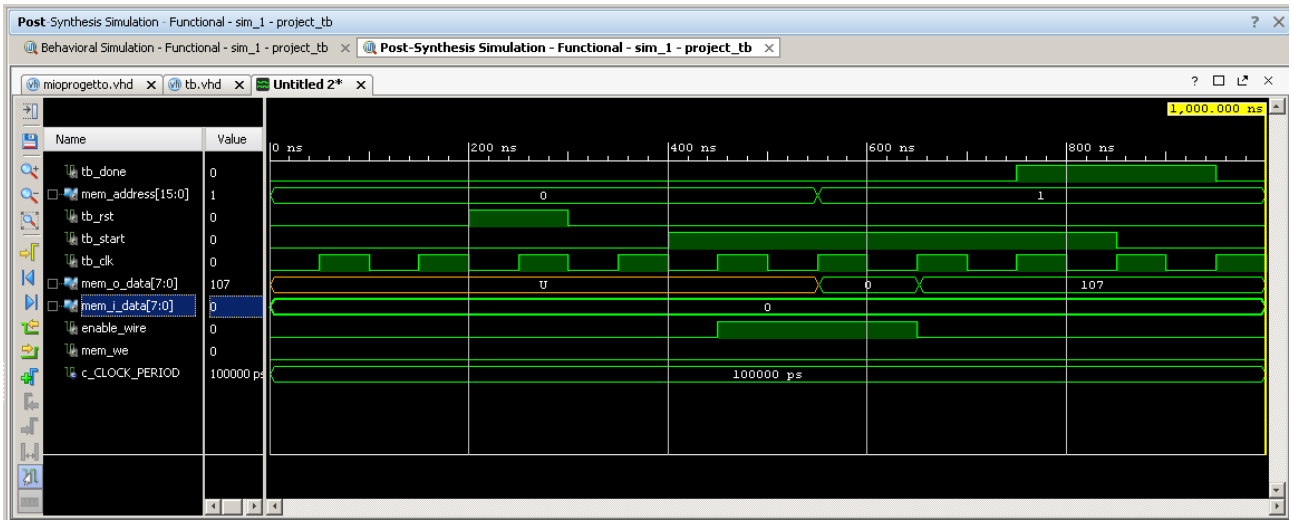

Fig. 5

Questo test serve a verificare cosa succede quando alla macchina viene detto che non ci sono parole in entrata mentre, dall'indirizzo 1 in poi, ci sono parole pronte alla lettura.

Come si vede (Fig. 5), al ciclo di clock successivo alla lettura del valore zero viene alzato il segnale o_done, nonostante 107 (codificato in binario) sia pronto alla lettura.

3. Sequenza massima

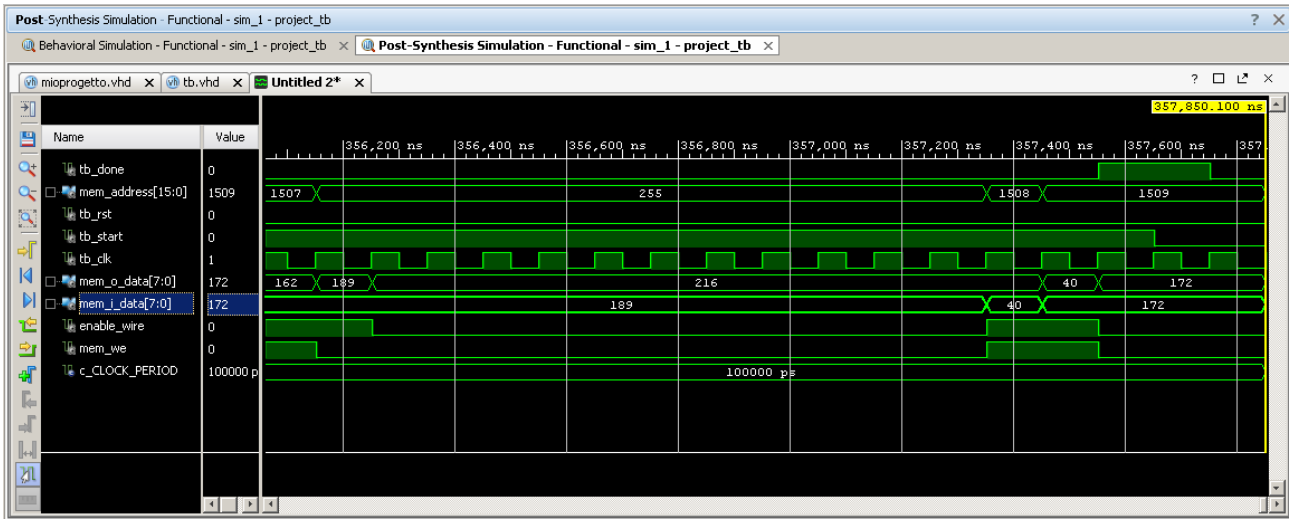

Fig. 6

Da specifica, la macchina deve saper codificare 255 Byte in un'unica sequenza.

Di tutta la simulazione ho catturato la parte finale proprio per verificare sia l'ultimo indirizzo di lettura, che risulta essere 255, sia gli ultimi due indirizzi di scrittura che sono 1508 e 1509.

Infatti 255*2 = 510, considerando che si parte a scrivere da 1000, all'indirizzo 1509 si trova la 510° parola.

4. Re_encode

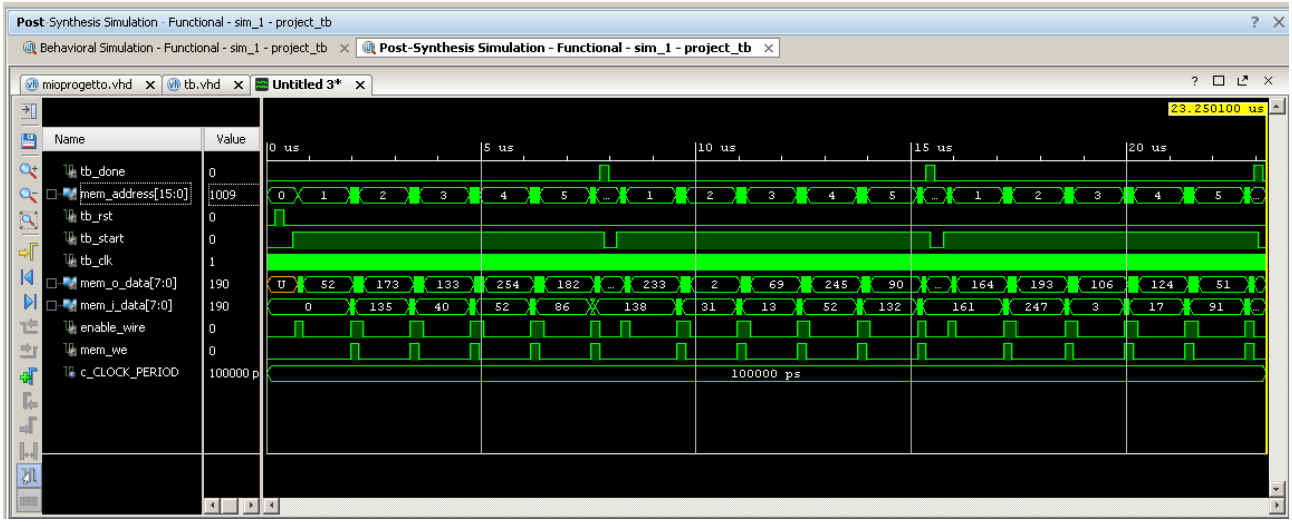

Fig. 7

Questo test verifica se la macchina riesce a codificare più sequenze di ingresso, una dietro l'altra.

In questo testbench (Fig. 7) si vede che alla fine di ogni codifica di una sequenza, viene alzato il segnale o_done, si attende che i_start diventi basso per poi inizializzare la macchina e iniziare una nuova codifica.

Grazie a quest'ultimo test mi sono ricordato di inizializzare i valori nello stato finale prima di iniziare una nuova codifica, in precedenza invece la macchina continuava la convoluzione sugli indirizzi di lettura e scrittura dell'ultima sequenza.

Un altro testbench che mi è stato utile è stato banalmente il primo, che non riporto perché superfluo, ma i cui valori di lettura e scrittura si possono vedere nell'esempio nella sezione "Introduzione". Grazie alle prime simulazioni mi sono reso conto di dover introdurre degli stati intermedi per aspettare i dati in arrivo dalla RAM, che non sono, come inizialmente pensavo, disponibili già al ciclo di clock successivo.

Report di sintesi

1. Report_utilization

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	172	0	134600	0.13
LUT as Logic	172	0	134600	0.13
LUT as Memory	0	0	46200	0.00
Slice Registers	106	0	269200	0.04
Register as Flip Flop	106	0	269200	0.04
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Fig. 8

Rispetto ad una prima versione del progetto, eliminando segnali intermedi ed alcuni assegnamenti e inizializzazioni superflue, è interessante notare come il numero di flip flop sia sceso da più di 200 alla metà di quelli iniziali, determinando una notevole riduzione di costi in termini di elementi di memoria. È un bene, inoltre, che non ci siano latch.

2. Report_timing

```
Timing Report

Slack (MET) :             96.482ns  (required time - arrival time)
  Source:                 parola_reg[7]/C
                            (rising edge-triggered cell FDRE clocked by clock  {rise@0.000ns fall@50.000ns period=100.000ns})
  Destination:            state_reg[1]/D
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@50.000ns period=100.000ns})
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            100.000ns  (clock rise@100.000ns - clock rise@0.000ns)
  Data Path Delay:        3.367ns  (logic 0.999ns (29.670%)  route 2.368ns (70.330%))
  Logic Levels:           3  (LUT4=1 LUT6=2)
```

Fig. 9

Qui è interessante notare lo slack, ovvero il periodo di clock inutilizzato dal nostro componente, con un periodo di clock di 5ns, ad esempio, il nostro componente funzionerebbe lo stesso. Infatti, di 100ns disponibili ne vengono utilizzati 3.518ns, un ottimo risultato.

Conclusioni

Scopo del progetto è la progettazione di un encoder per la codifica convoluzionale di una stringa di bit, codificando ogni bit in entrata con due bit in uscita.

La macchina progettata risponde efficacemente a questa richiesta, utilizzando meno del 5% del periodo di clock a disposizione dei testbench assegnati.
Inoltre, il componente ha passato numerosi test sia scritti manualmente per confermarne il funzionamento in casi limite, sia generati in maniera pseudo-casuale per verificarne la sua persistenza nel soddisfare i requisiti.

Progettare il modulo come macchina a stati ha permesso di dividere tutti i vari compiti in stati diversi, rendendo l'architettura di facile comprensione e scalabile per una futura modifica.

Infine, anche se non richiesto dalla specifica, ho testato il codice scritto in post implementation timing simulation, che ne prova il funzionamento su un fpga, utilizzando ritardi sui bus reali, e non più ideali.