POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

PROVA FINALE - PROGETTO DI RETI LOGICHE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Mykhalio Shpakov - 10656977 / 937848
Davide Preatoni - 10696246 / 939259,

Advisor:
Prof. Gianluca Palermo

Co-advisors:

Dr. Antonio Miele

Academic year:
2021-2022

Abstract: VHDL (acronimo di VHSIC Hardware Description Language, dove "VHSIC" è la sigla di Very High Speed Integrated Circuits) è lo strumento fondamentale per la progettazione dei moderni circuiti integrati digitali. Lo scopo è stato quello di strutturare ed implementare tramite VHDL un codice convoluzione 1/2 che rispetti la specifica assegnata.

Key-words: codice convoluzionale, VHDL, affidabilità della trasmissione

1. Introduzione

La necessità di comunicare da parte della società ha portato sempre più un aumento dei canali per la comunicazione rendendo critico il problema dell'integrità dei dati trasmessi. Nella trasmissione digitale possono esserci interferenze che introducono rumore. In questo caso il ricevitore deve avere la possibilità di determinare il dato effetivamente ricevuto, prima che sia stato affetto dai rumori di canale.

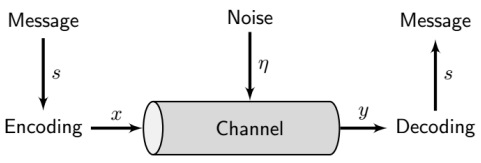

Figure 1

Una possibile soluzione è l'utilizzo di codici convoluzionali, detti CC (illustrati come *Encoding* in Figure 1). Essi vengono solitamente implementati tramite circuiti logici sequenziali e permettono al ricevitore di recuperare il segnale trasmesso.

Lo scopo è stato quello di progettare, descrivere e sintetizzare un modulo hardware, tramite VHDL per *Xilinx FPGA* [1] che realizzi un CC 1/2.

Questo report è organizzato come segue. Nella sezione 2 è presente un'ulteriore spiegazione teorica di cosa sia un codice convoluzionale e delle possibili applicazioni concrete. Nella sezione 3 troviamo la specifica di progetto. Nella sezione 4 riportiamo dettagliatamente le nostre scelte progettuali. Nella sezione 5 riportiamo i risultati sperimentali, le quali conclusioni discusse infine nella sezione 6.

2. Codice Convoluzionale

Nel campo delle telecomunicazioni un codice convoluzionale è un codice per la correzione degli errori.
In particolare abbiamo che:

- l'input viene portato in ingresso come una stringa di m bit ed è codificato in un output a n bit. Dove m/n è il rapporto (rate) di codifica.
- la trasformazione è una funzione degli ultimi k simboli d'informazione, dove k è la lunghezza dei vincoli del codice (constraint length).

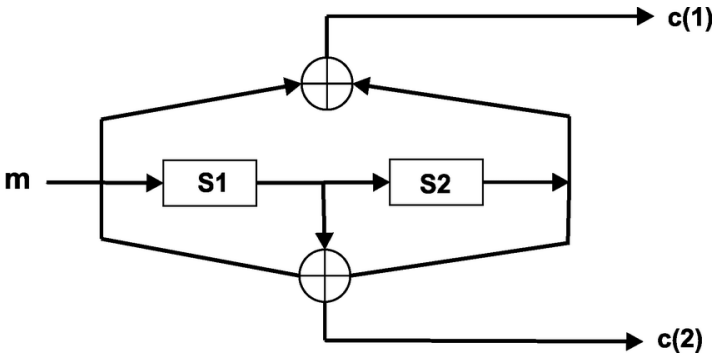

Figure 2: Codice convoluzionale 1/2

Nella Figure 2 viene mostrata una rete sequenziale che implementa un codice convoluzionale 1/2. Tale circuito prende 1 bit in ingresso (*m* nell'immagine) e mediante l'operazione logica XOR produce 2 bit in uscita c(1) e c(2). I segnali intermedi sono le uscite dei registri S1 , S2 che sono a loro volta i precedenti 'm' propagati nei vari cicli di clock.

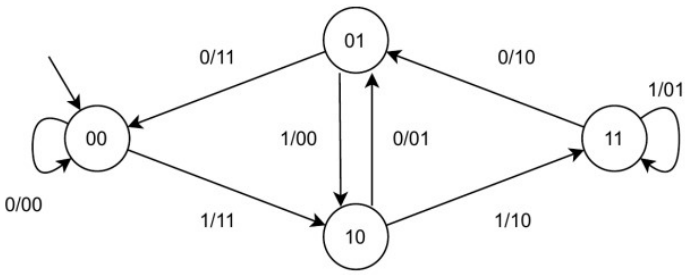

Figure 3: Diagramma della macchina a stati del codice convoluzionale mostrato in Figure 2

La caratteristica principale delle reti sequenziali è il concetto di *Stato*; Il circuito sequenziale può essere rappresentato con la macchina a stati finiti il cui diagramma viene mostrato in Figure 3. Lo stato è costituito dai 2 bit dei registri S1, S2 mentre l'ingresso è definito dal bit m che viene fornito come input al CC. L'uscita essendo costituita dai 2 bit c(1) e c(2), dipende anche dall'ingresso applicato nello stato stesso, il che vuol dire che la macchina a stati in esame è la *macchina di Mealy*[2]

[1] Field Programmable Gate Array (abbreviato con FPGA) è un circuito integrato le cui funzionalità logiche di elaborazione sono appositamente programmabili e modificabili.

[2] Macchina di Mealy è una macchina a stati finiti la cui funzione d'uscita dipende sia dall'ingresso che dallo stato corrente.

Per ogni macchina a stati si può costruire il diagramma a traliccio come uno mostrato in Figure 4. Tale diagramma mostra le transizioni tra gli stati per una specifica sequenza di bit in ingresso e le uscite scritte per ogni transizione. Lo si può utilizzare anche al contrario; confrontando le uscite scritte con quelle sulle transizioni si può determinare qual è stata la sequenza di bit applicata all'ingresso del codice convoluzionale.

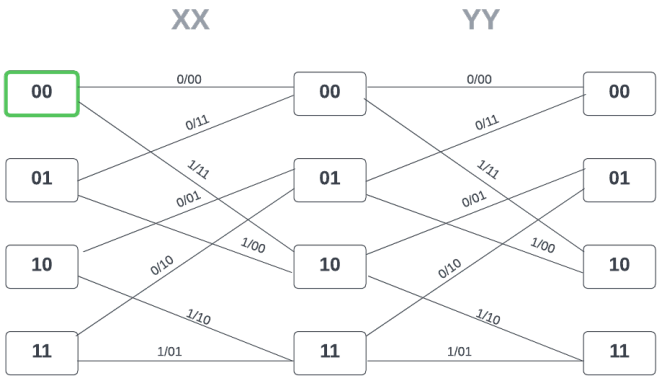

Figure 4: Diagramma a traliccio

Spesso accade che la sequenza d'uscita del codice convoluzionale può subire delle variazioni a causa dei rumori presenti nel mezzo di comunicazione. In tal caso occorre stabilire la sequenza che è stata codificata sul lato trasmittente. Se la lunghezza dei vincoli (constraint length) è relativamente piccola è opportuno utilizzare **l'algoritmo di Viterbi** per questo scopo. Immaginiamo che XXYY sia la sequenza dei bit ricevuti (con XX e YY intendiamo qualsiasi 4 bit). Si parte dallo stato di reset, evidenziato in verde sull'immagine Figure 4, e si seguono le transizioni possibili da quello stato. Ad ogni transizione viene associato un costo che è la somma del costo della transizione precedente più il peso della transizione in esame.

$$c_k = c_{k-1} + p_k$$

Ci sono diversi modi per associare il peso ad una transizione, uno di questi è l'uso della distanza di hamming tra la coppia di bit che viene scritta in uscita dalla FSM per quella transizione e la coppia di bit ricevuti.

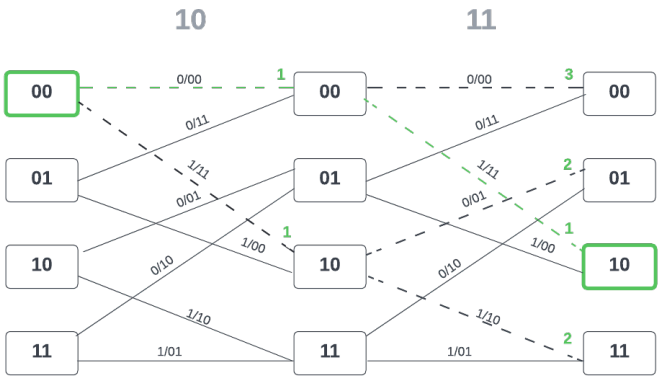

Figure 5: Esempio Algoritmo di Vitebri con diagramma a traliccio

Dopo che tutti i bit ricevuti sono stati analizzati lo scopo è quello di scegliere la sequenza di transizioni con il minor costo possibile. Seguendo tale percorso si ottiene la sequenza più probabile dei bit che sono stati forniti all'ingresso del nostro codice convoluzionale. Nella Figure 5 si può vedere un'esempio di applicazione di Vitebri Algorithm.
Con le linee trateggiate denotiamo le transizioni esaminate. Ad ogni transizione associamo un costo, scritto in verde. Le linee trateggiate verdi indicano il percorso con il costo minore. Tale percorso definisce la sequenza d'ingresso più probabile che è stata fornita al codice convoluzionale sul lato trasmittente, che è uguale a "01".

3. Specifica di progetto

Il modulo deve leggere dalla memoria una sequenza continua di W parole, ognuna di 8 bit e deve scrivere in memoria una sequenza continua di Z parole, ognuna da 8 bit.

- La memoria ha indirizzamento al byte.
- La quantità di parole W da codificare è memorizzata nell'indirizzo 0 della memoria mentre il primo byte della sequenza W è memorizzato all'indirizzo 1.
- La dimensione massima della sequenza di ingresso è 255 byte.
- Lo stream di uscita Z deve essere memorizzato a partire dall'indirizzo 1000.

Ognuna delle parole di ingresso viene serializzata, in modo tale da generare un flusso continuo U da 1 bit. Serializzazione avviene dal bit più significativo a quello meno significativo. Su questo flusso d'ingresso viene applicato il codice convoluzionale 1/2. Questa operazione genera in uscita un flusso continuo Y. Il flusso Y è ottenuto come concatenamento alternato dei due bit di uscita. La sequenza delle parole d'uscita Z è la parallelizzazione su 8 bit, del flusso continuo Y.

3.1. Interfaccia del Componente

Il componente da descrivere deve avere la seguente interfaccia.

```
entity project_reti_logiche is
      port (
            i_clk     : in std_logic;
            i_rst     : in std_logic;
            i_start   : in std_logic;
            i_data    : in std_logic_vector(7 downto 0);
            o_address : out std_logic_vector(15 downto 0);
            o_done    : out std_logic;
            o_en      : out std_logic;
            o_we      : out std_logic;
            o_data    : out std_logic_vector (7 downto 0)
      );
end project_reti_logiche;
```

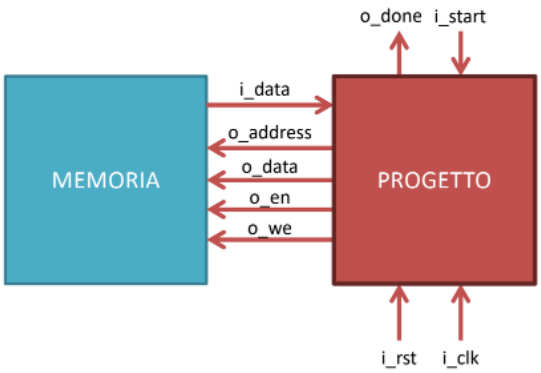

Figure 6: Interfaccia del componente con la memoria

In particolare:

- i_clk è il segnale di *clock* in ingresso.
- i_rst è il segnale di *reset* che riporta i segnali allo stato iniziale.
- i_start è il segnale di *start*, cioè di inizio del processo.
- i_data è il segnale che arriva dalla memoria in seguito ad una richiesta di lettura.
- o_address è il segnale di uscita che manda l'indirizzo alla memoria.
- o_done è il segnale di uscita che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria.
- o_en è il segnale di *enable* che controlla l'attivazione della memoria per poter comunicare (sia in lettura che in scrittura).
- o_we è il segnale di *write enable* che controlla l'attivazione della memoria per poter scrivere.
- o_data è il segnale di uscita dal componente, rappresenta la parola da scrivere.

3.2. Algoritmo convolutivo

Dato in ingresso il flusso $U(k)$, il modulo produce $P_1(k)$ e $P_2(k)$, secondo le seguenti equazioni:

$$P_1(k) = U(k) \oplus U(k-2)$$

$$P_2(k) = U(k) \oplus U(k-1) \oplus U(k-2)$$

I singoli bit prodotti sono poi concatenati, producendo il flusso d'uscita $Y(k)$:

$$Y(k) = P1(k).P2(k)$$

3.3. Protocollo di elaborazione

Il modulo funzionale HW deve rispettare il seguente protocollo

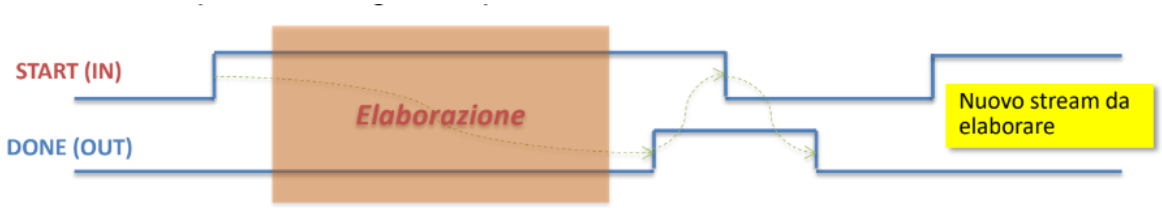

Figure 7: Protocollo di elaborazione del modulo

1. Il modulo partirà nella elaborazione quando un segnale *start* in ingresso verrà portato a 1.
2. Il segnale di *start* rimarrà alto fino a che il segnale di *done* non verrà portato alto.
3. Al termine della computazione (compresa la scrittura del risultato in memoria), il modulo da progettare deve alzare il segnale *done* che notifica la fine dell'elaborazione.
4. Il segnale *done* deve rimanere alto fino a che il segnale di *start* non è riportato a 0. Un nuovo segnale *start* non può essere dato fin tanto che *done* non è stato riportato a zero.
5. Una volta *start* è stato riportato a 0, anche segnale *done* viene riportato a 0 e il modulo si mette in attesa del prossimo segnale di inizio.
6. Quando viene rialzato il segnale di *start*, il modulo dovrà ripartire con la codifica della sequenza successiva.

Il modulo deve essere progettato considerando che prima della prima codifica verrà sempre dato il *reset* al modulo. Invece, come descritto nel protocollo precedente, una seconda elaborazione non dovrà attendere il reset del modulo ma solo la terminazione della elaborazione. L'aggiornamento della memoria con la nuova sequenza d'ingresso W da codificare avviene mentre il segnale *start* è basso.

4. Architettura

L'architettura generale del modulo consiste di due *Datapath* e una *Macchina a stati*.

1. Datapath - Module: consiste nel calcolo effettivo dei bit convoluti e nella scrittura in memoria.
2. Datapath - Address: gestitsce gli indirizzi.
3. FSM gestisce il funzionamento generale.

Il componente e' stato progettato e simulato utilizzando Xilinx Vivado; l'FPGA target e' lo Xilinx Artix-7 xc7a200tfbg484-1.

4.1. Macchina a stati

Il valore di default di tutti i segnali che la macchina a stati scrive in uscita è pari a 0. Lo stato S0 è sia lo stato iniziale che di reset.

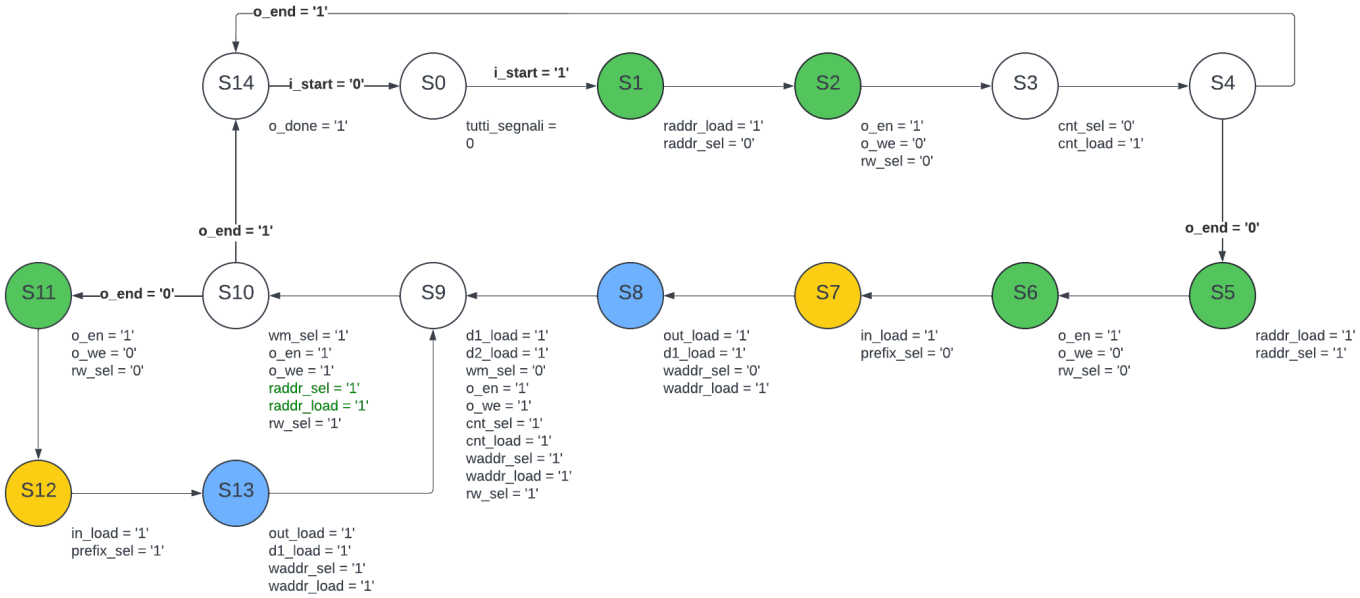

Figure 8: Versione iniziale della macchina a stati

La versione iniziale della macchina a stati è raffigurata sull'immagine Figure 8.
Qui è presente una distinzione tra stati anche se svolgono operazioni molto simili tra di loro:

- S1 e S5 svolgono il calcolo dell'indirizzo di lettura.
- S2 e S6 e S11 leggono dalla memoria.
- S7 e S12 memorizzano l'input del CC e selezionano il prefisso.
- S8 e S13 memorizzano L'output del CC e svolgono il calcolo per il nuovo indirizzo di scrittura.

Questa distinzione compare a causa di valori diversi delle uscite tra stati simili. In particolare:

- raddr_sel per S1 e S5
- prefix_sel per S7 e S12
- waddr_sel per S8 e S13

Osservato che questi stati sarebbero equivalenti in assenza di uscite. Abbiamo scelto di eliminare i segnali raddr_sel per poter unire S1 e S5, prefix_sel per unire S7 e S12, waddr_sel per unire S8 e S13. A tal proposito abbiamo aggiunto dei registri rsel_reg, psel_reg e wsel_reg con lo scopo di memorizzare il valore di selezione. Per quanto riguarda invece gli stati S2 e S6; necessitiamo di avere un segnale che ci dica se abbiamo già effettuato una prima lettura del primo byte per poter effetturare una riduzione degli stati.
Nel caso sia la lettura del primo byte dobbiamo passare nello stato S3 e memorizzare il numero di parole da codificare, altrimenti dobbiamo passare nello stato S7 e memorizzare la parola da codificare stessa. Perciò abbiamo introdotto il segnale o_is_first_read che vale 1 se l'ingresso di selezione del MUX R è pari a 0, siccome l'ingresso di selezione assume 0 solo per il calcolo dell'indirizzo del primo byte che dobbiamo leggere.

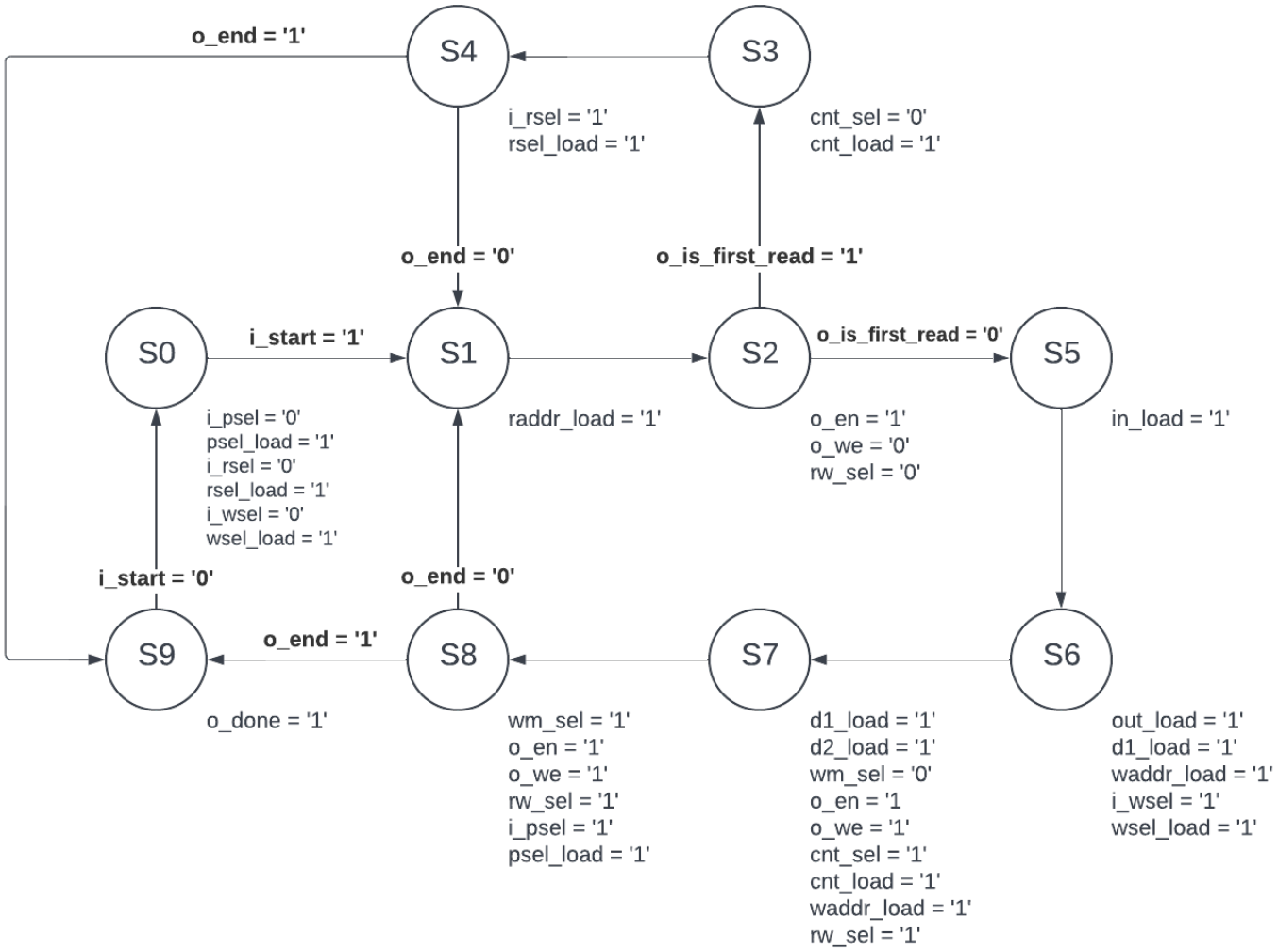

Figure 9: Macchina a stati ridotta

In risultato che abbiamo ottenuto è una FSM con un numero ridotto di stati ed è raffigurata in Figure 9.
Di seguito viene riportato il ruolo di ciascun stato.

4.1.1. IDLE State(S0)

Il primo stato a cui si passa in seguito al segnale di *start* basso e dal quale parte l'elaborazione della sequenza W di parole. Propaga gli ingressi di selezione "0" nei registri waddr_sel, raddr_sel, prefix_sel per permettere di iniziare una nuova elaborazione.

4.1.2. Stato 1(S1)

E' lo stato in cui si passa in seguito al segnale *start* alto o al segnale *end* basso che indica che l'elaborazione delle sequenza W non è ancora stata completata. Calcola l'indirizzo da quale leggere il prossimo byte della memoria. Ciò avviene con il valore salvato nel registro rsel_reg fornito al MUX R.

4.1.3. Stato 2(S2)

Lo stato che legge dalla memoria il byte nell'indirizzo memorizzato da reg_raddr.

4.1.4. Stato 3(S3)

E' lo stato in cui si passa a seguito della lettura del primo byte dalla memoria. Attiva il registro reg_cnt che salva il numero di parole da processare.

4.1.5. Stato 4(S4)

Propaga l'ingresso di selezione "1" nel registro rsel_reg per permettere di calcolare gli indirizzi successivi di lettura.

4.1.6. Stato 5(S5)

Si passa in seguito alla lettura della memoria successiva al primo indirizzo. Attiva il registro reg_in che salva la parola da codificare con il prefisso fornito dal mux P.

4.1.7. Stato 6(S6)

Memorizza l'output del CC nel registro reg_out. Propaga l'ultimo bit della parola d'ingresso al registro D1. Calcola l'indirizzo sul quale scrivere il prossimo byte nella memoria. Ciò avviene con il valore salvato nel registro wsel_reg. Propaga l'ingresso di selezione "1" nel registro wsel_reg per calcolare gli indirizzi successivi di scrittura negli stati prossimi.

4.1.8. Stato 7(S7)

Stato che scrive nella memoria il byte più significativo del risultato della codifica. Memorizza gli ultimi 2 bit della parola d'ingresso nei registri D1 e D2. Calcolo il numero di parole ancora da trasformare dopo che questa è stata trasformata e lo memorizza nel registro reg_cnt

4.1.9. Stato 8(S8)

Stato che scrive nella memoria il byte meno significativo del risultato della codifica. Propaga l'ingresso di selezione "1" nel registro psel_reg per permettere di concatenare i valori salvati in D1 e D2 alla parola successiva da trasformare.

4.1.10. Stato 9(S9)

Stato in cui si passa al seguito dell'elaborazione finita correttamente. L'elaborazione è finita correttamente quando tutte le parole della sequenza d'ingresso W sono state codificate ed il loro risultato scritto nella memoria.

NOME	TIPO	VALORE INIZIALE	DESCRIZIONE
cur_state, next_state	S	S0	Memorizza lo stato corrente/prossimo
in_load, out_load	STD_LOGIC	0	Attiva il registro reg_in/reg_out che salva input/output del codice convoluzionale
d1_load, d2_load	STD_LOGIC	0	Attivano i registri D1, D2 che memorizzano gli ultimi 2 bit della parolla d'ingresso
i_psel	STD_LOGIC	0	Ingresso del registro psel_reg
psel_load	STD_LOGIC	1	Attiva il registro psel_reg che fornisce ingresso di selezione del MUX P
wm_sel	STD_LOGIC	0	Dice quale dei 2 byte scrivere nella memoria
cnt_sel	STD_LOGIC	0	Ingresso di selezione del MUX C
cnt_load	STD_LOGIC	0	Attiva il registro reg_cnt che memorizza il numero di parole ancora da elaborare
i_rsel, i_wsel	STD_LOGIC	0	Ingresso del registro rsel_reg/wsel_reg
rsel_load, wsel_load	STD_LOGIC	1	Attivano i registri rsel_reg/wsel_reg che forniscono ingressi di selezione al mux R/W
raddr_load, waddr_load	STD_LOGIC	0	Attivano i registri reg_raddr/reg_waddr che memorizzano l'indirizzo di lettura/scrittura successivo
rw_sel	STD_LOGIC	0	Ingresso di selezione del MUX A che fornisce o_address
o_end	STD_LOGIC	1	Dice se tutte le parole della sequenza d'ingresso sono state processate
o_is_first_read	STD_LOGIC	1	Serve per distinguere tra la lettura del numero di parole e la lettura della parola da codificare stessa

Tabella riassuntiva dei segnali interni

4.2. Datapath del modulo convoluzionale

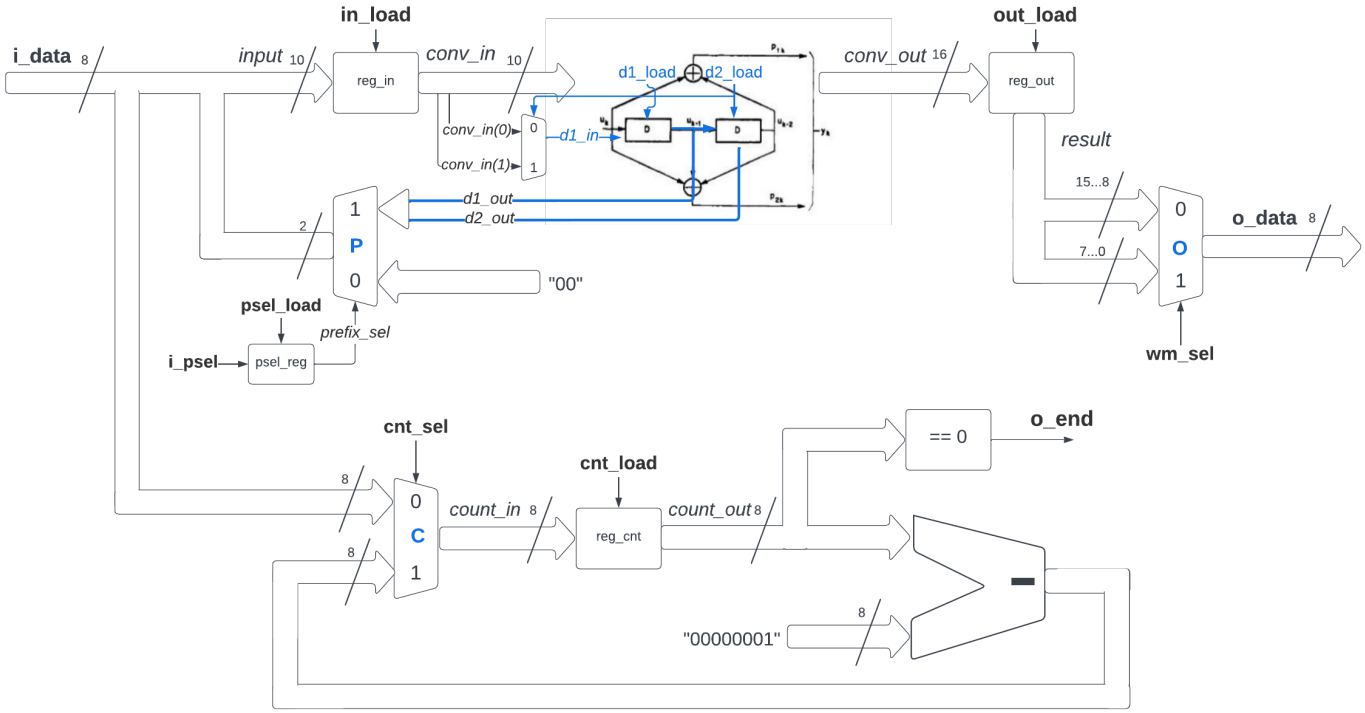

Figure 10: Datapath del modulo convoluzionale

Il modulo è stato descritto con:

- la collezione di process che implementano i registri reg_in, reg_out, d1, d2, reg_cnt e psel_reg
- le equazioni dataflow che implementano la parte combinatoria restante del circuito

Una delle scelte progettuali è stata quella di implementare il codice convoluzionale che codifichi gli 8 bit in parallelo in un singolo ciclo di clock. Per far ciò è stato necessario introdurre il concetto di prefisso (i 2 bit meno significativi della parola codificata precedente). Tale prefisso va aggiunto alla parola in ingresso e viene creato il segnale input fornito all'ingresso del codice convoluzionale. Questo permette di eseguire la codifica continua della sequenza delle parole W e così trarre il beneficio del parallelismo.

NOME	TIPO	VALORE INIZIALE (dopo il reset)	DESCRIZIONE
input	STD_LOGIC_VECTOR(10)	U	Parola della sequenza W concatenata con 2 bit meno significativi della parola precedente oppure con "00" nel caso sia la prima parola da codificare
conv_in	STD_LOGIC_VECTOR(10)	0	Vettore all'ingresso del codice convoluzionale
conv_out	STD_LOGIC_VECTOR(16)	0	Vettore all'uscita del codice convoluzionale
prefix_sel	STD_LOGIC	0	Segnale di selezione del mux P. Uguale a 0 per la prima parola da codificare
d1_in	STD_LOGIC	0	Fornisce penultimo/ultimo bit della parola da codificare
d1_out, d2_out	STD_LOGIC	0	Memorizzano 2 ultimi bit della parola d'ingresso da codificare
result	STD_LOGIC_VECTOR(16)	0	Risultato prodotto dal codice convoluzionale
count_in	STD_LOGIC_VECTOR(8)	U	Memorizza il numero di parole iniziale da processare o count_out decrementato di 1
count_out	STD_LOGIC_VECTOR(8)	0	Memorizza il numero di parole ancora da processare

4.3. Datapath del modulo per calcolo degli indirizzi

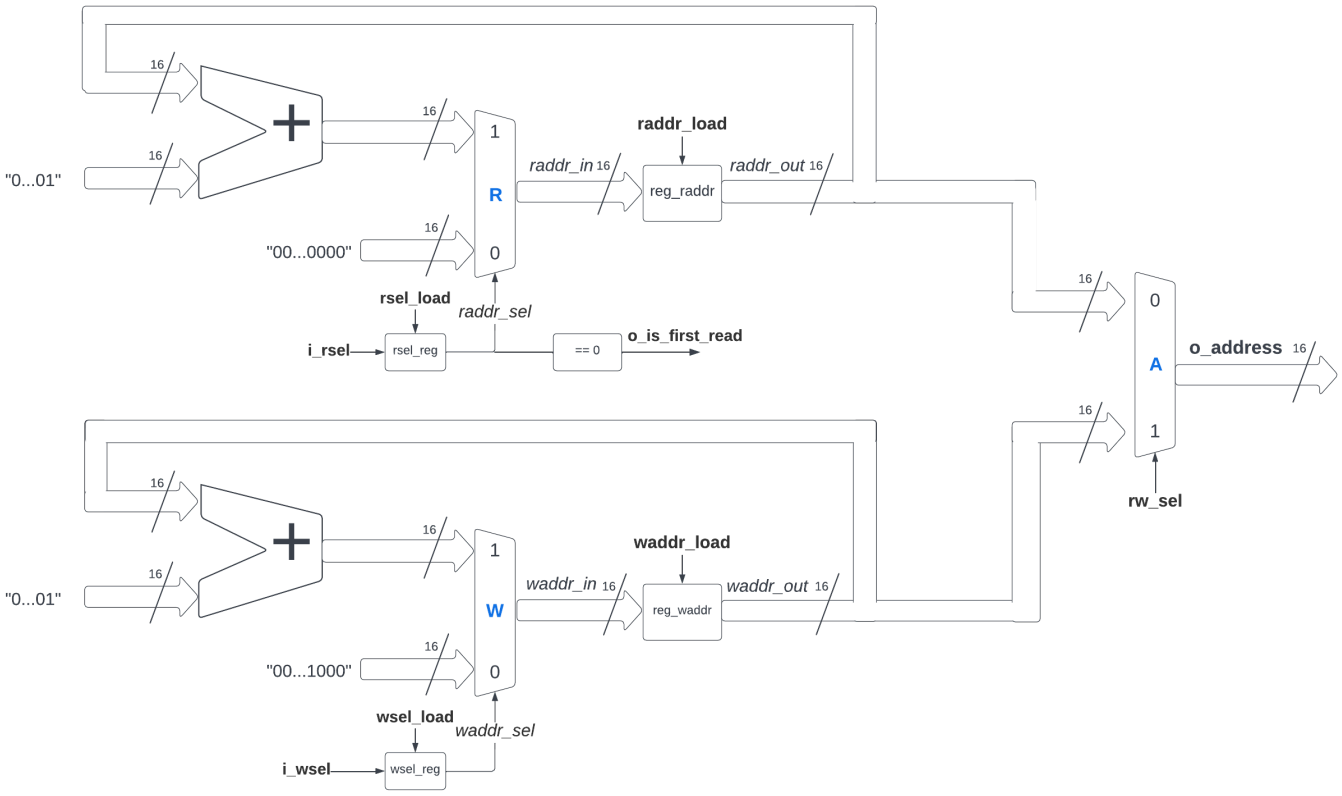

Figure 11: Datapath del modulo che calcola gli indirizzi

Il modulo è stato descritto con:

- la collezione di process che implementano i registri reg_raddr, reg_waddr, rsel_reg e wsel_reg
- le equazioni dataflow che implementano la parte combinatoria restante del circuito

Una delle scelte progettuale è stata quella di utilizzare i registri rsel_reg e wsel_reg che memorizzano gli ingressi di selezione dei MUX R e W. Questa decisione è stata supportata da un' osservazione: i segnali raddr_sel e waddr_sel sono pari a '0' solo per il calcolo del primo indirizzo della lettura/scrittura, dopodiché vengono settati a 1 e il loro valore non cambia per tutta la durata dell'elaborazione della sequenza d'ingresso W corrente. La stessa osservazione è stata fatta per il segnale prefix_sel mostraro in Figure 10 che è pari a '0' solo per la prima parola da codificare.

5. Risultati sperimentali

La progettazione è stata supportata dalla fase di sperimentazione; testando il componente con gli opportuni testbench, sia nei casi limite che nei casi standard. Il componente supera tutti i test a cui 'e stato sottoposto sia in simulazione *behavioural* che *post-syntesis* (+10 mila testbench).

5.1. Report di sintesi

Il componente finale risulta essere correttamente sintetizzabile ed il report di sintesi per l'FPGA target sopracitato risulta essere:

- LUT: 0,07% del totale
  - Datapath - Module: 36
  - Datapath - Address: 59
- FF: 0,03% del totale
  - Datapath - Module: 37
  - Datapath - Address: 34

Il Worst Negative Slack (WNS) al clock di 100 ns (dalla specifica) risulta essere di 96.967 ns.

5.2. Simulazioni

Il componente è correttamente simulabile al clock specificato di 100 ns, anche se è stato verificato che funziona correttamente a clock molto minori. Di seguito sono riportati i casi limiti superati di questo componente.

5.3. Testbench - Reset

Questo tb supporta il corretto funzionamento del componente dopo un reset asincrono.

5.4. Testbench - Codifica multipla

Questo tb supporta il il corretto funzionamento del componente su multiple codifiche consecutive senza reset.

5.5. Testbench - Codifica di una sequenza minima

Questo tb supporta il corretto funzionamento del componente quando la parola W, che rappresenta il numero di byte -words- da codificare, è uguale a zero.

5.6. Testbench - Codifica di una sequenza massima

Questo tb supporta il corretto funzionamento del componente quando la parola W, è uguale a 255 (massimo valore da specifica).

5.7. Testbench - Clock inferiore rispetto alla specifica

Questo test serve a verificare che anche con un constraint del clock più restrittivo funzioni tutto correttamente [15 ns]. Il componente progettato supera con successo anche questo testbench.

5.8. Testbench - generici

Per verificare un corretto comportamento da parte del componente si è scelto di utilizzare un generatore di test che ha simulato più di 10.000 prove in modo casuale, anche in questo caso il componente non ha riscontrato problemi sia in simulazione behavioural e post-synthesis.

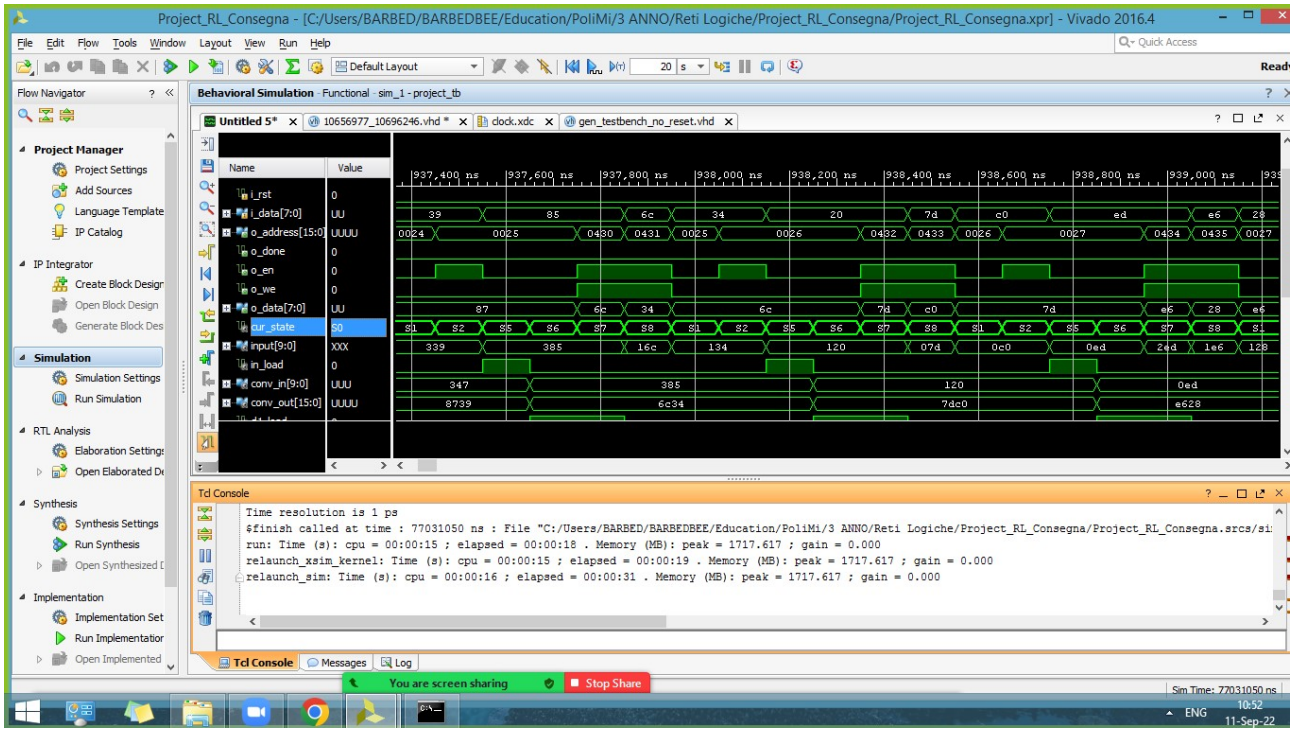

Figure 12: Forma d'onda durante i 10.000 test

6. Conclusioni

Riteniamo che il componente HW descritto rientri nelle specifiche richieste superando tutti i tb come descritto nella sezione precedente. Abbiamo compreso quanto sia importante una buona analisi della specifica e un'attenzione ai dettagli durante la fase di progettazione, ancor prima di pensare ai compoenenti fisici da implementare.