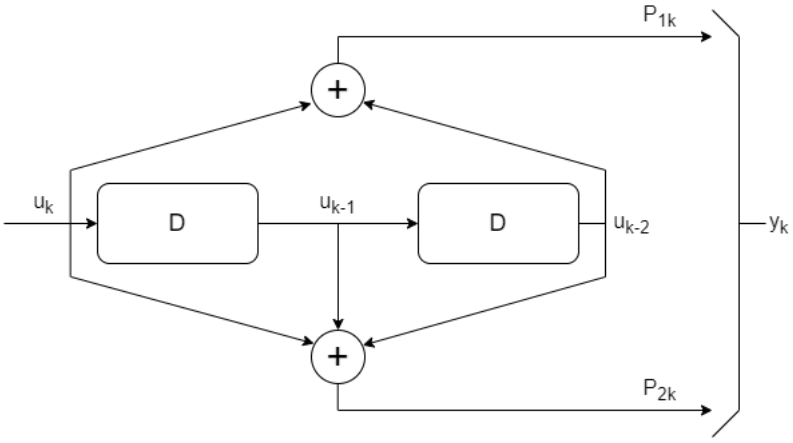
Prova Finale Reti Logiche

Prof. Gianluca Palermo

Lorenzo Paleari 10673890
Emanuele Paci 10681377

A.A. 2021-2022



Indice

1 Introduzione

L'obiettivo del progetto è implementare un componente hardware descritto in VHDL che si interfacci con una memoria. Il componente deve riuscire ad elaborare delle sequenze fornite dalla memoria applicando il codice convoluzionale $\frac{1}{2}$. Quest'ultimo è sfruttato nel campo delle telecomunicazioni allo scopo di ottenere un trasferimento dei dati affidabile mediante una trasformazione m/n, nella quale m bit in ingresso vengono convertiti in n bit di uscita.

1.1 Specifica

Il componente riceve in ingresso una sequenza continua di W parole (8 bit) e restituisce in uscita una sequenza continua di Z parole che rispetterà il rapporto definito dal codice convoluzionale producendo in uscita il doppio delle parole ricevute in ingresso (Z = 2*W). Ogni bit u_k viene preso ed elaborato dal codificatore convoluzionale generando due bit p_{1k} e p_{2k} che sono poi concatenati per generare un flusso continuo y_k che andrà a formare la sequenza continua Z di parole.

Il modulo deve partire quando un segnale START viene posto a 1. Alla fine dell'elaborazione il componente deve portare un segnale DONE a 1, il quale verrà posto a 0 solo quando il segnale di START viene riportato a 0. Una volta compiuti questi passaggi il segnale START potrà essere riportato a 1, cominciando una nuova elaborazione. Prima della prima codifica verrà sempre dato un RESET al modulo, mentre per le successive non è necessario attendere il segnale di RESET; il componente dovrà quindi essere in grado di tornare allo stato iniziale autonomamente alla fine di ogni elaborazione e prima della codifica successiva.

1.2 Interfaccia

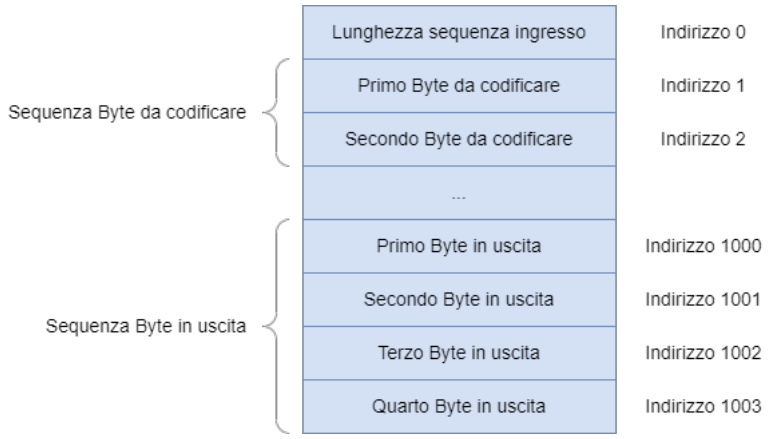
Il componente da descrivere ha la seguente interfaccia:

```
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;
```

- il nome del modulo deve essere `project_reti_logiche`.
- `i_clk` è il segnale di CLOCK in ingresso generato dal TestBench.

- `i_rst` è il segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START.
- `i_start` è il segnale di START generato dal Test Bench.
- `i_data` è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura.
- `o_address` è il segnale (vettore) 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 da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura).
- `o_we` è il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter scriverci. Per leggere da memoria esso deve essere 0.
- `o_data` è il segnale (vettore) di uscita dal componente verso la memoria.

1.3 Dati e Memoria

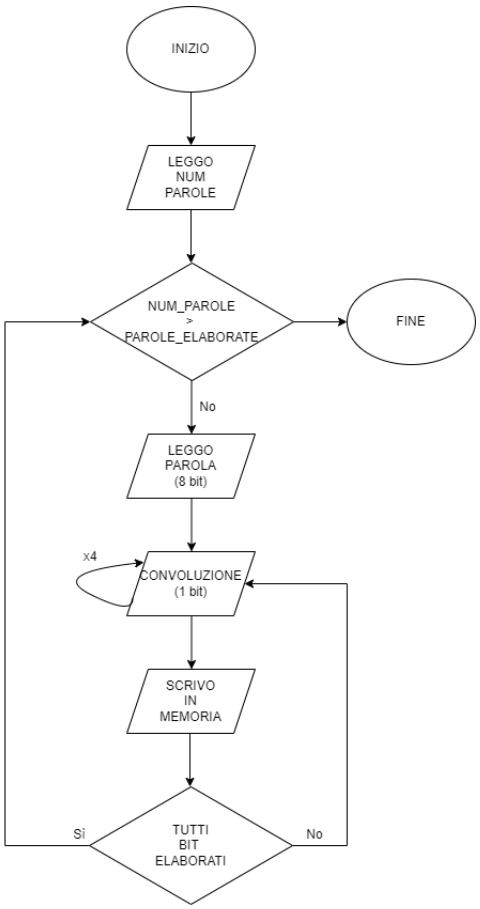


I dati, ciascuno da 8 bit (1 parola) sono memorizzati e andranno poi salvati in una memoria con indirizzamento al Byte. L'indirizzo 0 è usato per memorizzare la quantità delle parole da codificare, può variare da un valore minimo di 0 ad un valore massimo di 255. Gli indirizzi dal 1 a N sono usati per leggere i valori delle parole da codificare, N corrisponde al valore letto nell'indirizzo 0 Gli indirizza dal 1000 in avanti sono utilizzati per scrivere i valori delle parole codificate rispettando l'ordine di ingresso (Verranno usati il doppio degli indirizzi rispetto a quanti usati per i dati letti).

2 Analisi della specifica

Analizzando la specifica, abbiamo modellizzato un algoritmo che descriva la sequenza di azioni che il modulo deve compiere per ottenere il risultato richiesto. In seguito abbiamo poi ottenuto una macchina a stati in funzione dell'algoritmo, che siamo andati ad implementare in VHDL.

Come primo passo viene letto il numero di parole da codificare. Successivamente, si controlla se tutte le parole sono state elaborate. In caso di esito positivo l'algoritmo ha terminato la sua esecuzione, altrimenti legge dalla memoria la prossima parola da codificare. Applica quindi il codice convoluzionale alla prima metà della parola letta ottenendo una parola completa in uscita che andrà ad essere salvata in memoria. Viene poi controllato se è stata elaborata anche la seconda metà della parola. In caso di esito positivo si tornerà al controllo del numero di parole elaborate, altrimenti verrà applicato nuovamente il codice convoluzionale alla parte mancante. Nella figura sottostante è rappresentato l'algoritmo da noi ideato



3 Design

Per implementare la nostra FSM abbiamo deciso di descrivere il componente con due processi:

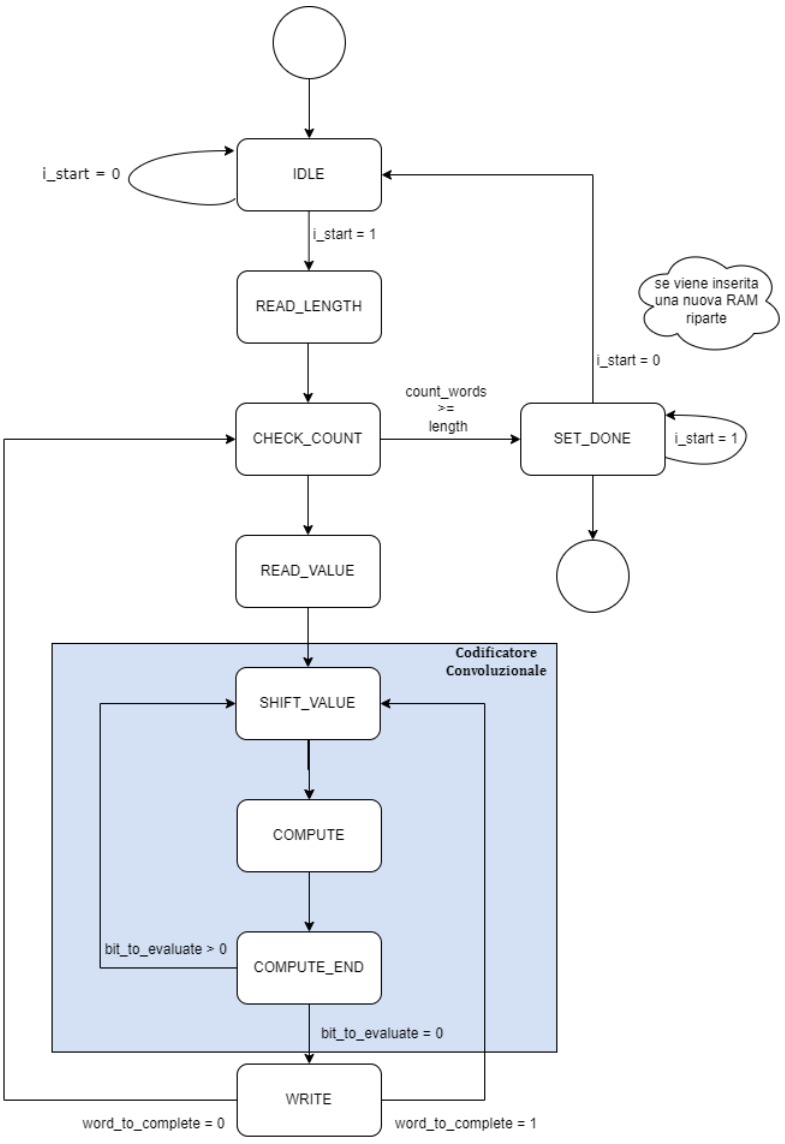
- Il primo rappresenta la parte sequenziale della macchina e quindi come vengono manipolati i registri. Si occupa anche di gestire eventuali reset asincroni e di inizializzare tutte le variabili.
- Il secondo invece implementa la macchina a stati che analizza i segnali in ingresso e lo stato corrente per determinare il prossimo stato in cui evolverà il sistema.

3.1 Descrizione degli stati

- `IDLE`
  È lo stato iniziale della macchina, attende che il segnale `i_start` vada a '1'. La macchina può tornare in questo stato a seguito di un segnale di reset.
- `READ_LENGTH`
  Leggo da RAM la quantità di parole da elaborare dall'indirizzo iniziale e copia il contenuto di `i_data` in un segnale che viene mantenuto per tutta la durata dell'esecuzione.
- `CHECK_COUNT`
  Controlla se il numero di parole elaborate è maggiore o uguale del valore letto inizialmente. In caso affermativo, pone il segnale `o_done` a '1' e passa allo stato `SET_DONE`. Altrimenti incrementa l'indirizzo di memoria e passa allo stato successivo.
- `READ_VALUE`
  In questo stato vengono lette le parole da elaborare una alla volta, ripetendo l'operazione finché non vengono elaborati i dati.
- `SHIFT_VALUE`
  Stato in cui ha inizio l'applicazione dell'algoritmo convoluzionale, il quale consiste in questo passaggio nello shift dei bit u_k, u_{k-1} e u_{k-2}.
- `COMPUTE`
  In questo stato vengono calcolati i bit di uscita e vengono serializzati in un segnale che verrà poi stampato al raggiungimento degli 8 bit.
- `COMPUTE_END`
  Controlla se sono già stati elaborati 4 bit in ingresso (che corrisondono ad una parola in uscita). In caso affermativo si passa allo stato `WRITE`, altrimenti viene aggiornato il numero di bit elaborati e si ritorna a `SHIFT_VALUE`.
- `WRITE`
  Stato in cui si scrive in memoria la parola elaborata. In seguito, controlla se sono state

salvate in memoria due parole. In caso positivo, l'elaborazione della parola in ingresso è terminata e si torna a `CHECK_COUNT`. Diversamente è stata elaborata solo metà della parola in ingresso, quindi si riprende la convoluzione dallo stato `SHIFT_VALUE`.

- `SET_DONE`
  Stato finale della FSM. Ripristina tutti i segnali in preparazione di un eventuale successivo nuovo flusso di dati. Quando `i_start` viene portato a '0', si passa allo stato `IDLE` e la macchina è pronta per cominciare una nuova elaborazione o per terminare.



4 Risultati Sperimentali

4.1 Sintesi

Il componente è perfettamente sintetizzabile: la sintesi non presenta alcun errore o warning. In seguito è riportata la tabella Slice Logic presa dal report di utilizzo

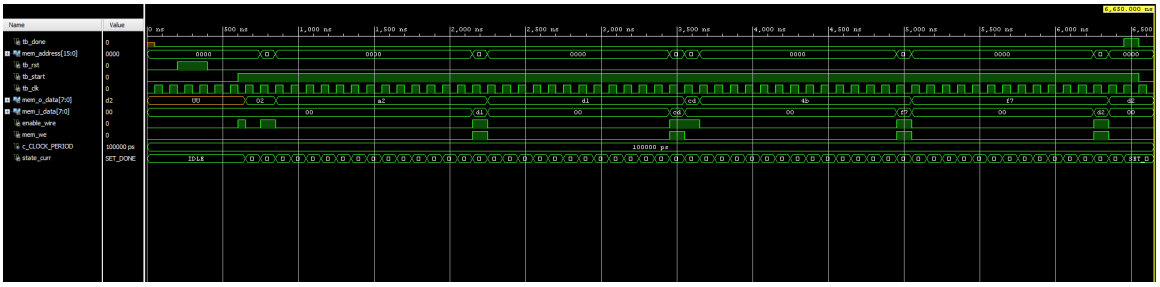
Site Type	Used	Fixed	Available	Util%
Slice LUTs*	119	0	134600	0.09
LUT as Logic	119	0	134600	0.09
LUT as Memory	0	0	46200	0.00
Slice Registers	76	0	269200	0.03
Register as Flip Flop	76	0	269200	0.03
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

4.2 Simulazioni

Abbiamo verificato il corretto funzionamento del componente realizzato, sottoponendolo innanzitutto al test bench fornitoci, e successivamente abbiamo individuato e testato possibili corner case, oltre ad aver simulato il componente con ulteriori test casualmente generati. In seguito la descrizioni di alcuni test.

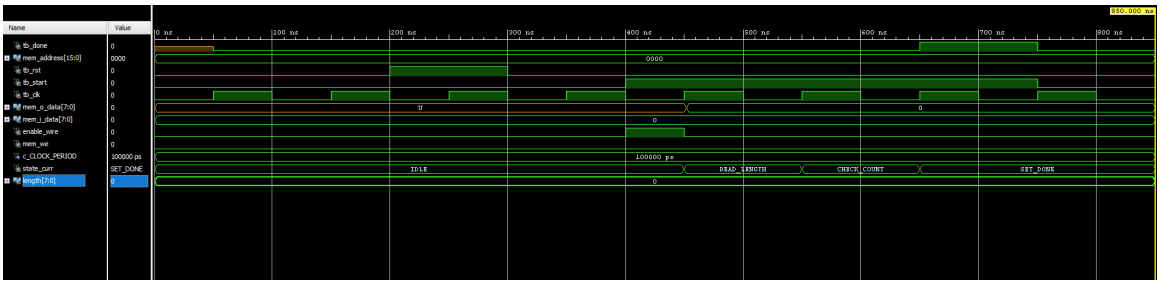
4.2.1 Test Bench Generici

Con questi tipi di test siamo andati a verificare il corretto funzionamento del nostro componente in una situazione di utilizzo normale.



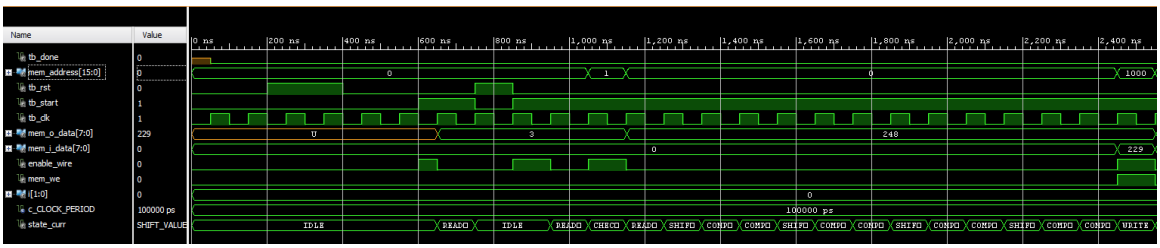
4.2.2 Test Bench Sequenza Minima

Questo test va a verificare che il componente sia in grado di lavorare correttamente anche se riceve come quantità di parole da elaborare il valore "00000000"; deve passare allo stato di `SET_DONE` senza scrivere niente in memoria.



4.2.3 Test Bench Reset Asincrono

Il test copre la possibilità di chiamare un segnale di reset asincrono. Il componente deve essere in grado di leggerlo in qualsiasi stato si trovi e di ricominciare la computazione dall'inizio evitando che quest'ultima venga compromessa.



4.2.4 Test Bench Ulteriori

Oltre a quelli precedentemente elencati, siamo anche andati a verificare il corretto funzionamento di altri due casi:

- Test Sequenza Massima
  Il test va a verificare che il componente funzioni con la dimensione massima della sequenza di ingresso, che corrisponde a 255 parole.

- Test Multiple Computazioni
  Abbiamo testato la possibilità di non terminare subito l'esecuzione alla conclusione di una computazione, ma di richiedere ulteriori elaborazioni su memorie successivamente fornite.

5 Conclusioni

Il nostro componente supera con successo tutti i test elencati precedentemente sia in Behavioral che in Post-Synthesis Functional.
Sfruttando un costraint[1], abbiamo verificato che il nostro componente è in grado di funzionare con un clock massimo di 100ns, precisamente abbiamo ottenuto uno slack di 95.905ns, che corrisponde ad un clock di poco più di 4ns.

[1] create_clock -period 100 -name clock [get_ports i_clk].
Dopo una simulazione in Post-Synthesis Functional, può essere chiamato in Console il comando report_timing che restituisce lo slack.