Prova Finale di Reti Logiche

Davide Galbiati

Giuseppe Gentile

A. A. 2021-2022

Matricole: 934913 – 932227
Codici Persona: 10706728 – 10680798
Docente: Fabio Salice

Indice

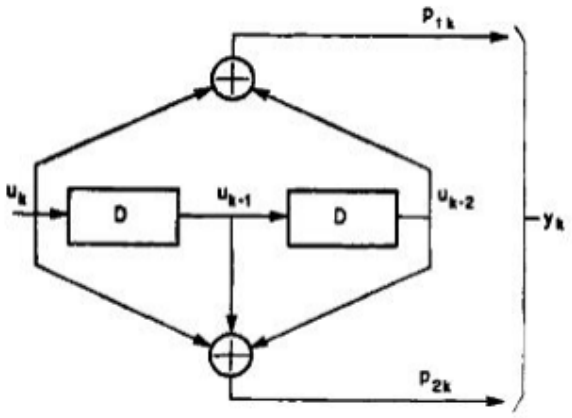
1. Introduzione

1.1 Obiettivo

La specifica della "*Prova Finale (Progetto di Reti Logiche)"* A.A.2021/2022 chiede di implementare un modulo HW (descritto in VHDL) che si interfacci con una memoria e che segua la seguente specifica.
Il modulo riceve in ingresso una sequenza continua di W parole, ognuna di 8 bit, e restituisce in uscita una sequenza continua di Z parole, ognuna da 8 bit.
Ognuna delle parole di ingresso viene serializzata; in questo modo viene generato un flusso continuo U da 1 bit.
Su questo flusso viene applicato il codice convoluzionale ½ (ogni bit viene codificato con 2 bit) secondo lo schema riportato in figura; questa operazione genera in uscita un flusso continuo Y.
Il flusso Y è ottenuto come concatenamento alternato dei due bit di uscita.
Utilizzando la notazione riportata in figura, il bit uk genera i bit p_{1k} e p_{2k} che sono poi concatenati per generare un flusso continuo y_k (flusso da 1 bit).
La sequenza d'uscita Z è la parallelizzazione, su 8 bit, del flusso continuo y_k.
Al componente viene richiesto di:

1. Accedere al primo indirizzo della memoria RAM per leggere il numero di parole della sequenza W. La lunghezza massima della sequenza è di 255 parole.
2. Leggere ogni parola della sequenza.
3. Calcolare le due nuove parole ottenute dalla singola parola letta.
4. Scrivere il risultato nella memoria RAM e ripetere il procedimento del punto 4 per tutte le parole della sequenza.

Inoltre, l'implementazione deve essere in grado di gestire i segnali START e DONE che identificano rispettivamente l'inizio e la fine della elaborazione della sequenza.
L'implementazione deve poi essere sintetizzata con target FPGA



Codificatore convoluzionale con tasso di trasmissione $\frac{1}{2}$.

1.2 Ipotesi progettuali

Abbiamo ipotizzato che il numero di byte da leggere (valore contenuto nella prima cella in memoria) sia del valore massimo di 255. Se il primo numero in memoria è uguale a zero la macchina si porta istantaneamente nello stato DONE (si assume la memoria settata con valore iniziali zero per ogni indirizzo).

I valori in ingresso non hanno ulteriori bisogni di controlli. Il segnale di start è chiamato sempre in maniera corretta.

1.2.1 Descrizione della memoria

Il modulo comunica con la RAM indirizzata al byte.

La memoria, inoltre, ha una configurazione iniziale così strutturata:

- nel primo indirizzo il numero di byte da leggere da memoria (questo escluso), indichiamo il contenuto della cella come *N*
- negli indirizzi [1, *N*] ci sono i byte da leggere e mandare in input alla FSM
- negli indirizzi [1000, 1000 + 2**N* - *1]* l'output della FSM, col corretto ordine per ogni cella

Address 1003	**210**
Address 1002	**247**
Address 1001	**205**
Address 1000	**209**
[..]	**…**
Address 2	**75**
Address 1	**162**
Address 0	**2**

Tabella 1: Rappresentazione della memoria

1.2.2 Interfaccia del componente

```
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;
```

i_clk	Segnale di clock in ingresso generato dal TestBench
i_rst	Inizializza la macchina pronta per ricevere il primo segnale di start
i_start	Segnale di start generato dal TestBench
i_data	Vettore che arriva dalla memoria in seguito a una richiesta di lettura
o_address	Vettore di uscita che manda l'indirizzo alla memoria
o_done	Segnale di uscita che segnala la fine dell'elaborazione e il dato scritto in memoria
o_en	Segnale da mandare alla memoria per poter comunicare (lettura e scrittura)
o_we	Segnale da mandare alla memoria per scrivere (=1), o per leggere (=0)
o_data	Vettore di uscita dal componente verso la memoria

Tabella 2: Interfaccia del componente

2. Implementazione

2.1 Descrizione ad Alto Livello

L'idea discorsiva del flusso del progetto è percorribile seguendo i punti:

1. Legge il primo indirizzo e salva il numero di byte da leggere in *num_bytes*
2. Se il valore è zero termina settando il segnale o_done al valore alto
3. Se *num_bytes* è diverso da zero lo moltiplica per due e passa a leggere nell'indirizzo di memoria 1
4. Per ogni byte in input si effettuano i seguenti passi:
    - 4.1 Si memorizza lo stato corrente
    - 4.2 Se la scrittura del byte non è ancora terminata:
        - 4.2.1 Scrivo su un vettore di uscita temporaneo il primo bit modificato
        - 4.2.2 Incremento i contatori di lettura e scrittura
        - 4.2.3 Passaggio allo stato successivo per il secondo bit da scrivere
        - 4.2.4 Ripetere il procedimento al punto 4.2.2
        - 4.2.5 Lettura del bit seguente
    - 4.3 Altrimenti la macchina entra nello stato di preparazione alla scrittura in memoria, dopo aver salvato lo stato in cui sarei andato se avessi dovuto leggere (*temp_state*)
        - 4.3.1 Scrivo effettivamente in memoria
        - 4.3.2 Controllo se ho scritto l'ultimo byte
        - 4.3.3 Se lo è termino, se non lo è riparto dal punto 4

2.2 Macchina a Stati Finiti

2.2.1 Rappresentazione grafica

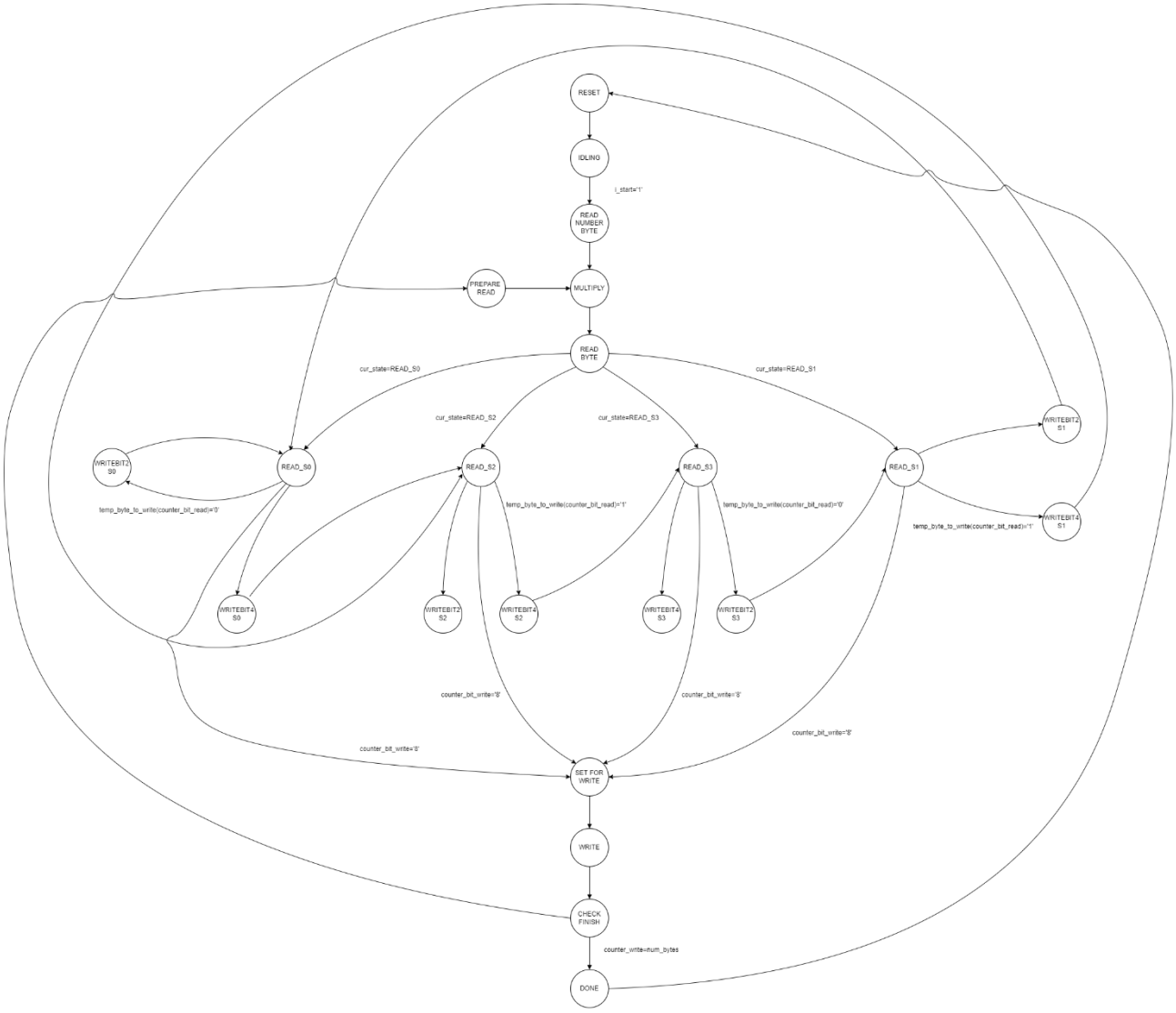


Figura 1: Rappresentazione grafica della macchina a stati finiti

2.2.2 Tabella degli Stati

IDLING	Stato iniziale, quando si riceve i_start si avanza in *READ_NUMBER_BYTE.*
READ_NUMBER_BYTE	Viene letto il primo indirizzo di RAM, memorizzato in un registro con 9 bit (uno aggiuntivo per lo shift). Si imposta *o_address* a "00000001" per leggere il primo byte da elaborare. Si avanza in *MULTIPLY.*
MULTIPLY	Se il numero di byte da leggere è 0 manda in *DONE*, altrimenti moltiplica per due e avanza in *READ_BYTE* (imposta *temp_state* a *READ_S0*, caso di partenza).
READ_BYTE	Sposta il byte da leggere in un vettore *temp_byte_to_read*, dove verranno effettuate le elaborazioni, e avanza in *temp_state.*
READ_S0	Primo stato del codificatore. Se ha finito di scrivere avanza in *SET_FOR_WRITE.* Se invece legge zero imposta a zero il bit corrente e avanza in *WRITEBIT2_S0*, altrimenti imposta a uno e avanza in *WRITEBIT4_S0*. Salva inoltre in *temp_state* lo stato corrente.
WRITEBIT2_S0	Imposta a zero il bit corrente e avanza in *READ_S0.*
WRITEBIT4_S0	Imposta a uno il bit corrente e avanza in *READ_S2.*
READ_S1	Se ha finito di scrivere avanza in *SET_FOR_WRITE.* Se invece legge zero imposta a uno il bit corrente e avanza in *WRITEBIT2_S1*, altrimenti imposta a zero e avanza in *WRITEBIT4_S1*. Salva inoltre in *temp_state* lo stato corrente.
WRITEBIT2_S1	Imposta a uno il bit corrente e avanza in *READ_S0.*
WRITEBIT4_S1	Imposta a zero il bit corrente e avanza in *READ_S2.*
READ_S2	Se ha finito di scrivere avanza in *SET_FOR_WRITE.* Se invece legge zero imposta a zero il bit corrente e avanza in *WRITEBIT2_S2*, altrimenti imposta a uno e avanza in *WRITEBIT4_S2*. Salva inoltre in *temp_state* lo stato corrente.
WRITEBIT2_S2	Imposta a uno il bit corrente e avanza in *READ_S1.*
WRITEBIT4_S2	Imposta a zero il bit corrente e avanza in *READ_S3.*
READ_S3	Se ha finito di scrivere avanza in *SET_FOR_WRITE.* Se invece legge zero imposta a uno il bit corrente e avanza in *WRITEBIT2_S3*, altrimenti imposta a zero e avanza in *WRITEBIT4_S3*. Salva inoltre in *temp_state* lo stato corrente.
WRITEBIT2_S3	Imposta a uno il bit corrente e avanza in *READ_S1.*
WRITEBIT4_S3	Imposta a zero il bit corrente e avanza in *READ_S3.*
PREPARE_READ	Inizializza *o_address* col contatore dei byte letti e il registro *o_we*
SET_FOR_WRITE	Azzera il contatore delle scritture e imposta *o_address* all'indirizzo 1000 + contatore delle scritture. Inoltre, dopo aver messo *o_we* ed *o_en HIGH,* manda in *WRITE.*
WRITE	Manda in *o_data* il vettore costruito nel codificatore e azzera quest'ultimo, avanza in CHECK_FINISH
CHECK_FINISH	Se ha finito di leggere tutta la sequeza avanza in DONE. Se ho finito di leggere il byte corrente manda in *PREPARE_READ* per il prossimo byte, altrimenti sono a metà byte e riprendo da *temp_state*
DONE	Alza *o_done* e avanza in RESET
RESET	Azzera tutti i registri

Tabella 3: Tabella degli stati

2.2.3 Segnali interni e registri

cur_state	Stato attuale della macchina.
temp_state	Stato salvato prima di salvare il byte in uscita dal convolutore. Quando ricomincia a leggere un nuovo byte riparte da questo stato.
counter_read	Numero di byte letti.
counter_write	Numero di byte scritti in memoria.
num_bytes	Numero di byte da leggere. Contenuto del primo indirizzo di memoria.
temp_byte_to_read	Vettore su cui viene salvato il byte da leggere.
temp_byte_to_write	Vettore su cui viene costruito il byte da scrivere
counter_bit_write	Posizione del nastrino di uscita della macchina. Si posiziona sui bit della parola in output e la costruisce. Dominio [0,8].
counter_bit_read	Posizione del nastrino di lettura della macchina. Si posiziona sui singoli bit della parola in ingresso. Dominio [0,8].

Tabella 4: Segnali interni e registri

2.3 Schema dell'implementazione

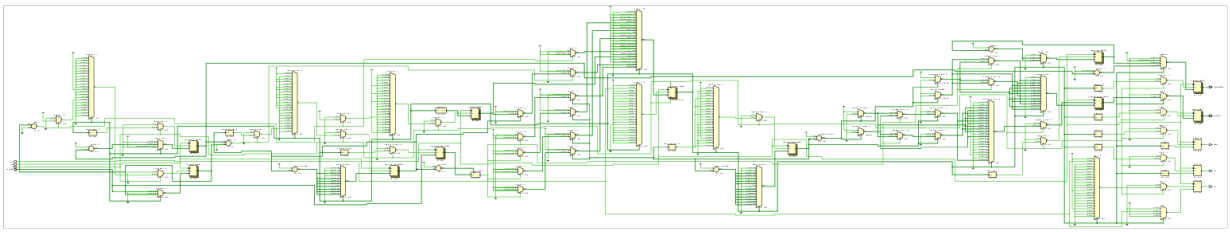


Figura 2: Schema dell'implementazione

3. Test benches

I test effettuati sono stati pensati per verificare transizioni critiche o possibili configurazioni di memoria estreme. In particolare, i test bench utilizzati comprendono le seguenti situazioni:

- Vari test generati casualmente tramite un generatore di sequenze di parole in python
- Caso specifico in cui il numero di parole della sequenza è minimo, cioè 0
- Caso specifico in cui il numero di parole della sequenza è massimo, cioè 255
- Caso con più sequenze con scrittura dei rispettivi risultati nella stessa memoria
- Caso con più sequenze con scrittura dei rispettivi risultati in memorie diverse

L'analisi dei risultati dei test sopracitati ha consentito l'avanzamento ottimale della scrittura del codice fino alla sua versione finale.

Per tutti i test riportati è stata effettuata la simulazione behavioural e successivamente la simulazione functional e timing post-synthesis, tutte con successo.

I risultati rilevanti sono riportati nella sezione 4.

4. Risultati Sperimentali

4.1 Report di sintesi

Dal punto di vista dell'area, la sintesi riporta il seguente utilizzo dei componenti:

- LUT: 149 (0,36% di utilizzo dell'area totale)
- FF: 100 (0,12% di utilizzo dell'area totale)

Si è posta particolare attenzione in fase di scrittura del codice affinché non venissero utilizzati Latch.

4.2 Risultato dei Test Benches

L'implementazione rispetta le specifiche, funzionando ad un periodo di clock pari a 100 ns.
Si riportano i risultati di tempo legati agli ultimi 4 test riportati nella sezione 3.

- Simulazione 1 sequenza di lunghezza 0 (Functional Post-Synthesis): 1900100 ps
- Simulazione 2 sequenza di lunghezza 255 (Functional Post-Synthesis): 663800100 ps
- Simulazione 3 sequenze con stessa RAM (Functional Post-Synthesis): 41200100 ps
- Simulazione 4 sequenze con diverse RAM (Functional Post-Synthesis): 34400100 ps

5. Risultati Sperimentali

Si ritiene che il componente progettato rispetti le specifiche, fatto sostenuto e verificato dalla esaustiva serie di test eseguiti tramite test benches creati sia manualmente, per esaminare specifiche casistiche, sia casualmente, per verificare situazioni di stress del componente.
L'architettura è stata pensata facendo riferimento ad un'unica FSM senza componenti esterni, evitando l'utilizzo di Latch che avrebbero potuto portare alla creazione di cicli infiniti.