Prova Finale (Progetto di Reti Logiche)

Jonathan Sciarrabba (Codice Persona: 10675342 - Matricola: 933553)

a.a. 2021-2022

Indice

1 Introduzione

1.1 Obiettivo del progetto

Realizzare un componente descritto in VHDL che data una sequenza di parole, ciascuna di 8 bit, applichi ad essa il codice convoluzionale 1/2. L'algoritmo consiste nel far corrispondere a ciascun bit letto in ingresso 2 bit in uscita.

1.2 Specifiche

Il modulo da realizzare riceve in ingresso una sequenza U di parole, ognuna di 8 bit, e restituisce in uscita una sequenza di parole Z, alla sequenza U viene applicato il codice convoluzionale 1/2 perciò la lunghezza di Z sarà di $2 * U$.
Il codice da applicare al flusso U segue il seguente diagramma di macchina a stati finiti:

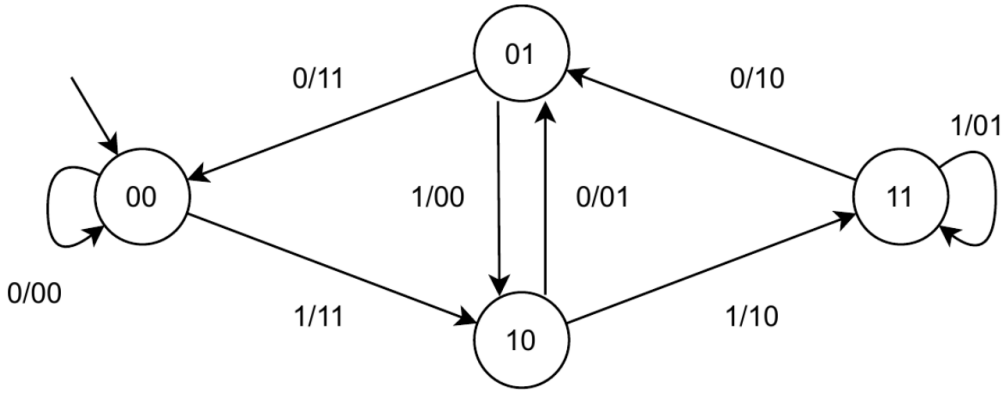

Figura 1: Rappresentazione della macchina a stati che realizza il codice convoluzionale

Un esempio di funzionamento con byte in ingresso: 11010110, si noti che la serializzazione avviene da sinistra verso destra perciò $T_0 = 1$, $T_1 = 1$, $T_2 = 0$, ecc.
In questo esempio U corrisponde appunto alla sequenza 11010110 che verrà codificata nel seguente modo:

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7
U	1	1	0	1	0	1	1	0
Z	11	10	10	00	01	00	10	10

Tabella 1: Esempio di codifca

La sequenza Z in uscita sarà dunque 1110100001001010 che corrisponde ai 2 byte:

- 11101000
- 01001010

che andranno scritti in memoria.
È presente inoltre un vincolo sul periodo di clock, il componente deve funzionare con un periodo di clock di almeno 100 ns.

1.3 Descrizione memoria

La memoria da cui il componente descritto legge e scrive è istanziata all'interno del TestBench, il suo funzionamento segue le linee guida della User Guide di VIVADO.
I dati all'interno della memoria sono indirizzabili al byte e sono contenuti secondo questo schema:

Numero di parole da codificare	←	Indirizzo 0
Prima parola di U	←	Indirizzo 1
Seconda parola di U	←	Indirizzo 2
...		
Prima parola di Z	←	Indirizzo 1000
Seconda parola di Z	←	Indirizzo 1001

Tabella 2: Rappresentazione schematica della memoria

1.4 Interfaccia del componente

Il componente da descrivere ha una interfaccia così definita:

```
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;
```

Nello specifico:

- `i_clk`: segnale di `CLOCK` generato dal TestBench
- `i_rst`: segnale di `RESET` che inizializza il componente pronto a ricevere il segnale di `START`
- `i_start`: segnale di `START` che dà inizio alla codifica
- `i_data`: segnale (vettore) in ingresso che rappresenta il byte letto dalla memoria
- `o_address`: segnale (vettore) che manda l'indirizzo alla memoria
- `o_done`: segnale di `DONE` che il componente invia al TestBench quando termina la codifica e scrittura di tutte le parole
- `o_en`: segnale di `ENABLE` che abilita l'accesso alla memoria sia in lettura che in scrittura
- `o_we`: segnale di `WRITE ENABLE` che abilita la scrittura in memoria
- `o_data`: segnale (vettore) di uscita che rappresenta il byte da scrivere in memoria

2 Architettura

2.1 Macchina a stati sincrona

Nel modulo realizzato la parte che si occupa di gestire i segnali di input e output rispetto al TestBench si comporta come una macchina a stati finiti che esegue i suoi cambi di stato sul fronte di salita del `CLOCK`.
Di seguito uno schematico per comprendere meglio il suo funzionamento:

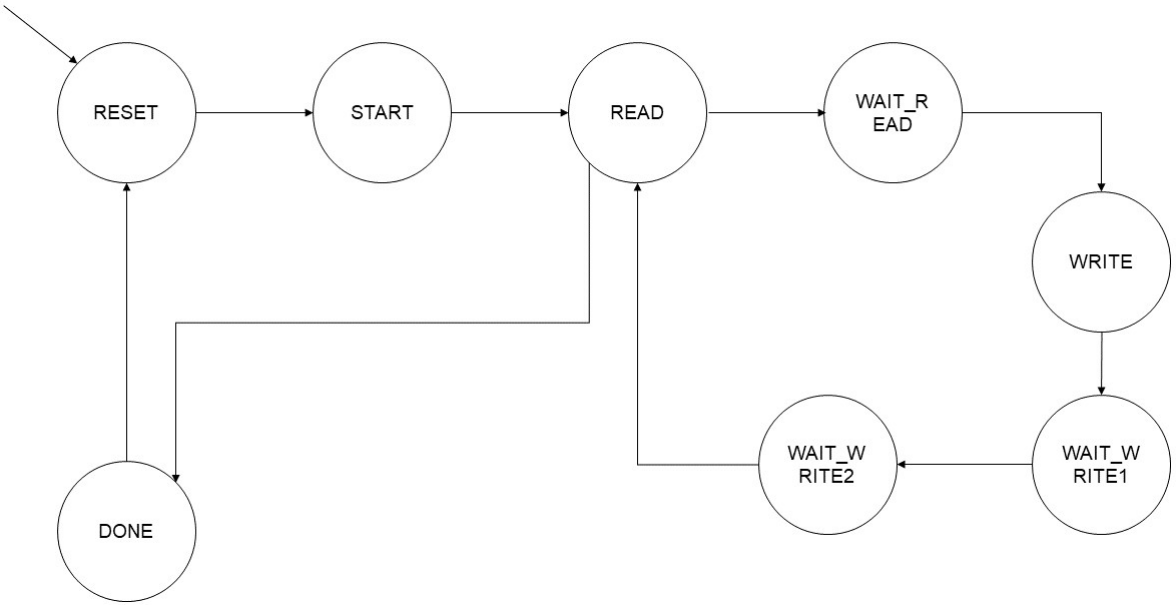

Figura 2: Schematico FSM gestione input/output

- RESET - stato iniziale, pronto a ricevere il segnale di `START`;
- START - stato transitorio per consentire l'accesso corretto alla memoria;
- READ - stato che assegna gli indirizzi da leggere alla memoria;
- WAIT_READ - stato che attende la disponibilità del dato in output dalla memoria;
- WRITE - stato che esegue il calcolo del codice convoluzionale 1/2;
- WAIT_WRITE1 - stato di attesa per la scrittura del primo byte;
- WAIT_WRITE2 - stato di attesa per la scrittura del secondo byte;
- DONE - stato di terminazione dell'elaborazione, invia il segnale di `DONE`.

2.2 Macchina a stati asincrona

Il calcolo del codice convoluzionale vero e proprio viene eseguito da una FSM asincrona scritta in una *procedure* richiamata ogni qualvolta si ha un nuovo byte da elaborare che ritorna alla macchina a stati sincrona una sequenza (vettore) di 16 bit che corrispondono ai 2 byte da dover scrivere in memoria. Il componente riproduce esattamente la FSM presentata in precedenza (vedi Figura 1).
Ecco un esempio di codifica effettuata dal modulo:

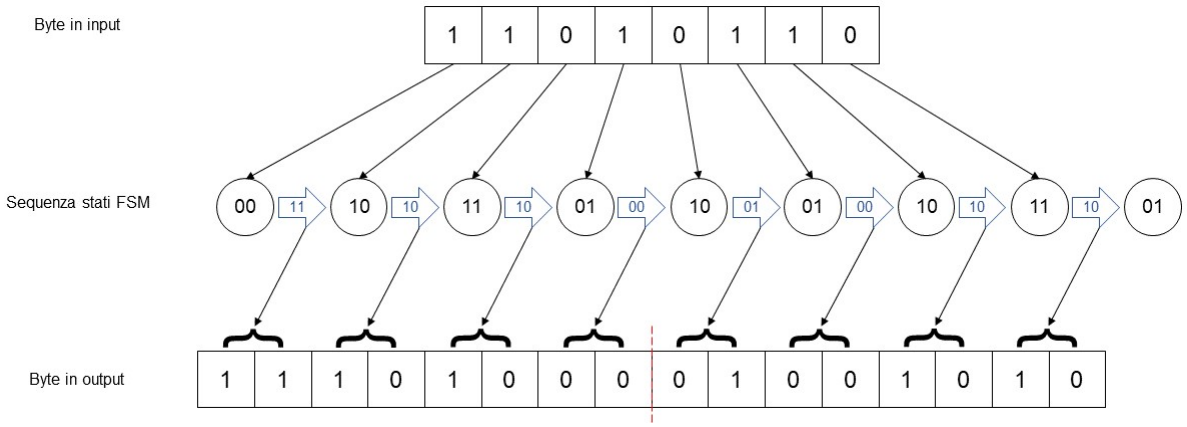

Figura 3: Esempio di codifica

3 Risultati Sperimentali

Sono stati effettuati diversi test per verificare il corretto funzionamento del componente.
Nello specifico sono stati effettuati test che andassero a verificare il comportamento desiderato anche nei cosiddetti *corner case*.

3.1 Testbench memoria vuota

Sequenza di parole da leggere di lunghezza 0 byte:
L'obiettivo del test è quello di verificare il corretto funzionamento del modulo nel caso in cui la sequenza di parole da codificare sia di lunghezza minima (valore 00000000 all'indirizzo 0 della memoria).

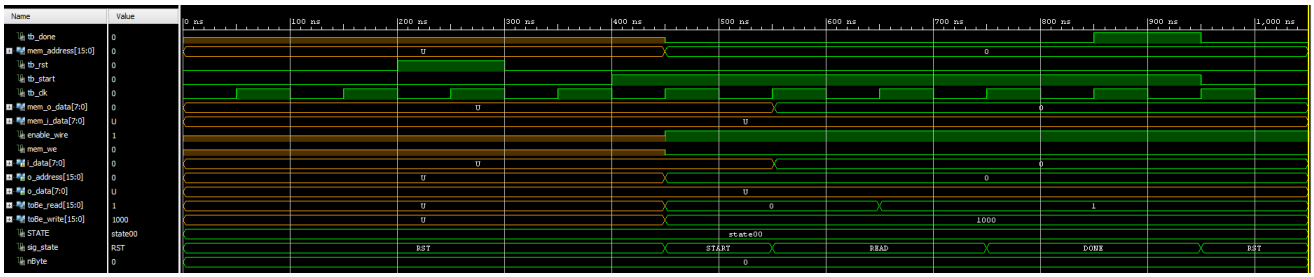

Figura 4: Forme d'onda del TestBench con sequenza di lunghezza minima

3.2 Testbench memoria piena

Sequenza di parole da leggere di lunghezza 255 byte:
L'obiettivo del test è quello di verificare il corretto funzionamento del modulo nel caso in cui la sequenza di parole da codificare sia di lunghezza massima (valore 11111111 all'indirizzo 0 della memoria).

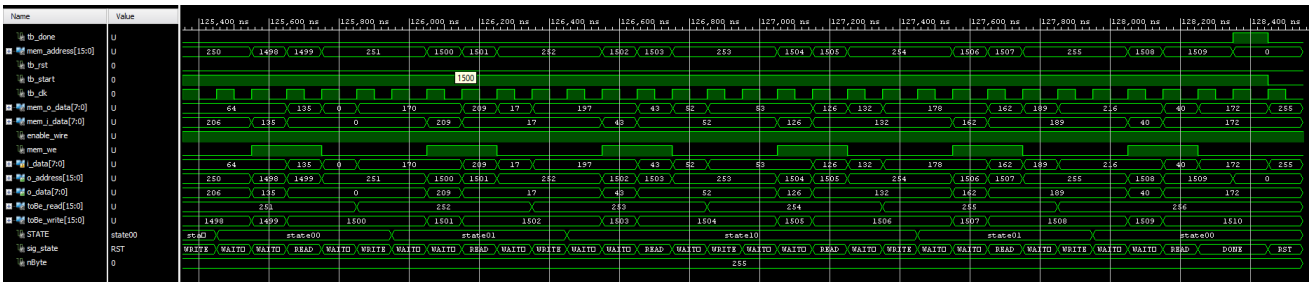

Figura 5: Forme d'onda del TestBench con sequenza di lunghezza massima (solo parte finale)

3.3 Testbench codifiche sequenziali

Segnali di `START` multipli:
L'obiettivo del test è quello di verificare il corretto funzionamento del modulo nel caso in cui al termine dell'elaborazione venga dato un nuovo segnale di `START` con conseguente cambio di contenuto in memoria.

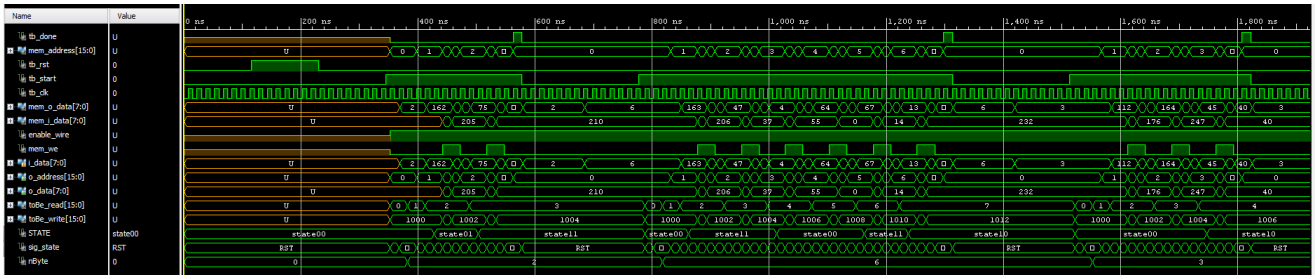

Figura 6: Forme d'onda del TestBench con multipli segnali di START

3.4 Testbench reset multipli

Segnali di `RESET` multipli:
L'obiettivo del test è quello di verificare il corretto funzionamento del modulo nel caso in cui vengano dati segnali di `RESET` (sincroni) casualmente durante l'elaborazione.

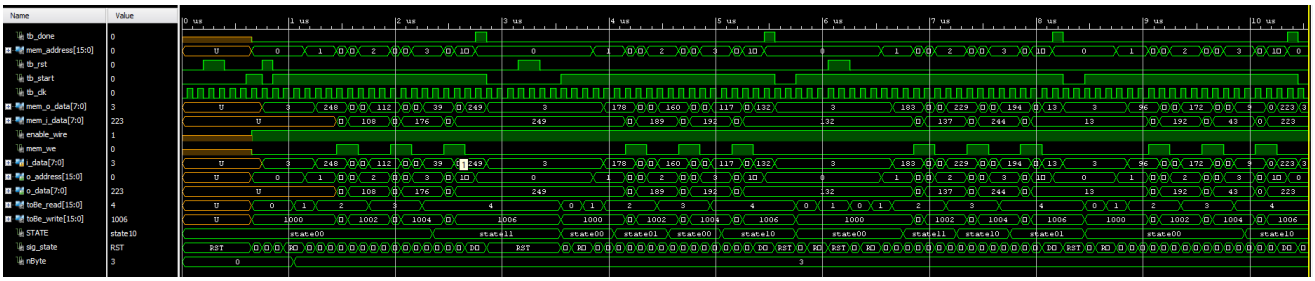

Figura 7: Forme d'onda del TestBench con multipli segnali di RESET

3.5 Testbench con valori randomici

Test con valori randomici per maggior copertura:
L'obiettivo del test è quello di coprire il maggior numero di scenari possibili, verificando quindi più possibili percorsi di esecuzione. Sono stati quindi eseguiti numerosi test con valori casuali in memoria.

4 Conclusione

Il componente descritto ha concluso con successo tutti i test effettuati in modalità *Behavioral*, *Post-Synthesis Functional* e *Post-Synthesis Timing*.
Di seguito sono presentati i tempi di esecuzione nei casi limite:

- Caso ottimo: 1.4 µs
- Caso pessimo: 128.6 µs

Il caso ottimo corrisponde al caso in cui la sequenza da codificare sia di 0 parole, Il caso peggiore invece corrisponde al caso in cui la sequenza da codificare sia di 255 parole.

4.1 Report timings

Dopo la sintesi è possibile verificare i timings del modulo e si può notare come il ciclo di clock possa scendere fino a circa 5 ns senza causare malfunzionamenti, siccome il *Worst Negative Slack* è di circa 95 ns su 100 ns di periodo di clock disponibile.

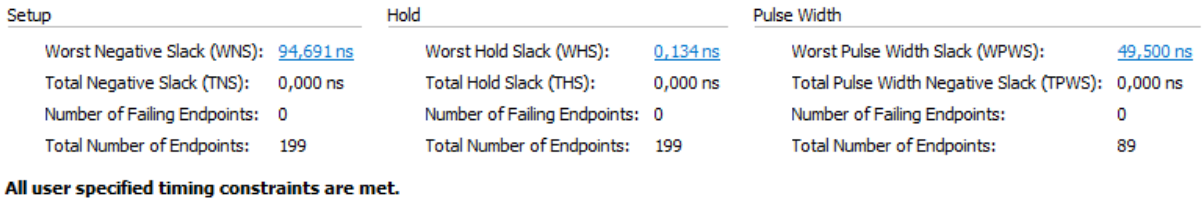

Figura 8: Report timings all'interno di VIVADO

4.2 Report utilization

Sempre dopo la sintesi è possibile verificare l'utilizzo di *LUT, Flip-Flop e Latch*. Quest'ultimi non sono presenti nel componente sintetizzato poiché darebbero origine a un circuito non più puramente combinatorio.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	105	0	134600	0.08
LUT as Logic	105	0	134600	0.08
LUT as Memory	0	0	46200	0.00
Slice Registers	88	0	269200	0.03
Register as Flip Flop	88	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

Figura 9: Report utilization all'interno di VIVADO