Prova Finale (Progetto Di Reti Logiche)

Carlo Sgaravatti (Codice Persona 10660072 - Matricola 937539)

Consegna : 1 Aprile 2022

Indice

1 Introduzione

1.1 Scopo del progetto

Il progetto è basato sulle codifiche convoluzionali, che sono utilizzate, nell'ambito della trasmissione delle informazioni, per rilevare e correggere gli errori di trasmissione. Nello specifico, l'obiettivo è sviluppare un modulo hardware che implementi un codice convoluzionale 1/2, ovvero: per ogni bit di ingresso vengono generati due bit in uscita.
Il modulo riceve in ingresso una sequenza di n parole da 8 bit, che devono essere codificate in una sequenza di 2n parole da 8 bit. In particolare, si richiede al componente di:

1. Leggere il numero di parole da codificare, accedendo ad una memoria RAM
2. Per ogni parola da codificare:
    - leggere la parola dalla memoria RAM
    - calcolarne la codifica
    - scrivere il risultato nella memoria RAM

La Figura 1 riporta il criterio con cui viene fatta la codifica; in particolare, considerando la sequenza di parole come un'unica sequenza di bit seriali, per ogni bit di ingresso, i bit di uscita sono costruiti nel seguente modo: detto `u_k` il bit di ingresso serializzato al tempo k

- il primo bit di uscita (`p_1k`) è lo xor dell'ultimo e del terzultimo bit della sequenza di ingresso (rispettivamente `u_k` e `u_k-2`)
- il secondo bit di uscita (`p_2k`) è lo xor degli ultimi tre bit della sequenza d'ingresso (`u_k`, `u_k-1` e `u_k-2`)

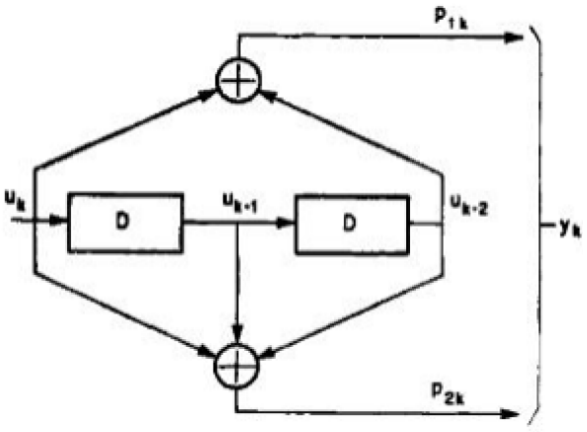

Figura 1: La codifica dell'uscita

1.2 Interfaccia del componente

L'interfaccia del componente da descrivere è la seguente:

```
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;
```

Dove:

- `i_clk` è il segnale di `CLOCK` in ingresso, generato dal testbench
- `i_rst` è il segnale di `RESET` che inizializza la macchina per poter ricevere il primo segnale di `START` (ovvero la prima sequenza di parole) ed è generato dal testbench
- `i_start` è il segnale di `START` generato dal testbench che, quando portato al livello logico alto, fa partire la computazione di una sequenza di parole
- `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 in uscita che comunica la fine dell'elaborazione della sequenza di parole
- `o_en` è il segnale di `ENABLE` da dover mandare alla memoria per potrer comunicare con essa (sia in lettura che in scrittura)
- `o_we` è il segnale di `WRITE ENABLE` a dover mandare alla memoria per poterci scrivere (se pari a 1). Per leggere dalla memoria deve essere settato a 0
- `o_data` è il segnale (vettore) di uscita che contiene la parola da scrivere in memoria

Il modulo partirà nell'elaborazione di una sequenza di parole quando `i_start` verrà portato ad 1 e terminerà l'elaborazione portando `o_done` a 1. Per iniziare una nuova elaborazione, `i_start` dovrà essere portato a 0 e si dovrà attendere che `o_done` venga portato a 0.

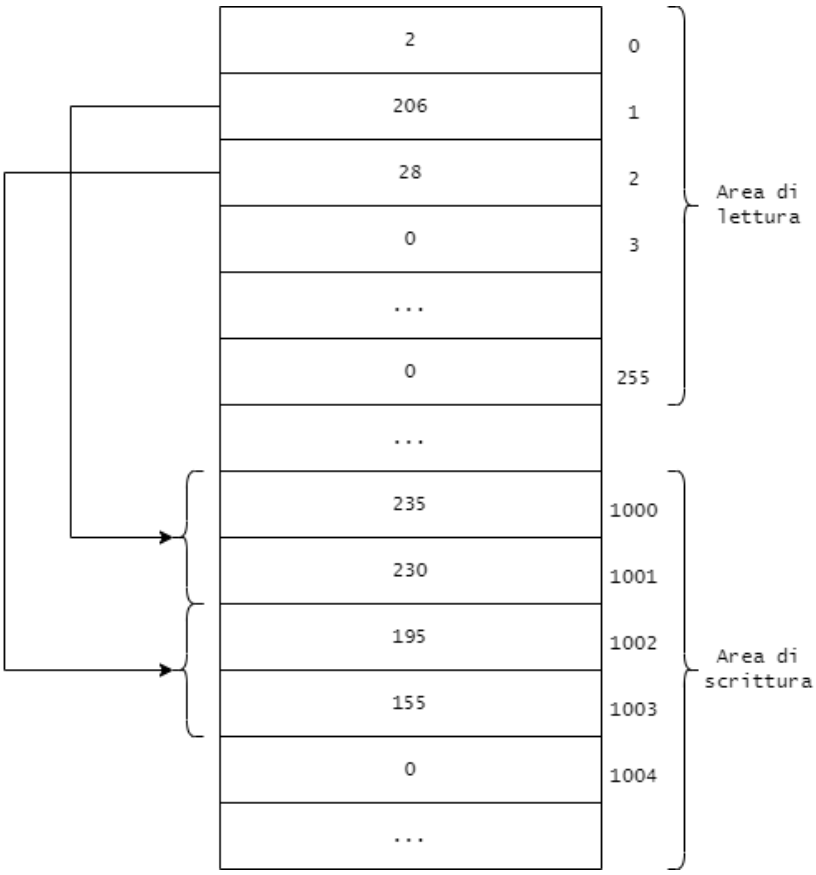

Figura 2: Esempio di utilizzo della memoria

1.3 Descrizione della memoria

I dati sono memorizzati in una memoria RAM con indirizzamento al byte, avente la seguente struttura:

- l'indirizzo 0 contiene il numero n di parole da 8 bit contenute nella sequenza di parole da elaborare (con n compreso tra 0 e 255)
- gli indirizzi da 1 a n (per n maggiore di 0) contengono le parole da codificare
- gli indirizzi da 1000 a (2n - 1) + 1000 sono utilizzati per la scrittura delle parole codificate in uscita dal componente da progettare; in particolare, la codifica dell'i-esima parola in ingresso (contenuta nell'indirizzo i) viene scritta negli indirizzi 2i + 998 e 2i + 999

La Figura 2 riporta un esempio con n = 2.

2 Architettura

2.1 Schema dell'implementazione

L'architettura è stata progettata in maniera modulare, in particolare sono stati sviluppati i seguenti tre moduli:

- `FSM Principale`: modulo che si interfaccia con la memoria (lettura e scrittura) e che si occupa di gestire il protocollo di inizio e fine della computazione di ogni sequenza di parole (gestito dai segnali i_start e o_done). Il modulo rappresenta il top-level component del progetto.
- `Serializzatore/Deserializzatore`: modulo che si occupa di serializzare la parola da elaborare, allo scopo di produrre l'ingresso del Convolutore, e di deserializzare i singoli bit prodotti dal Convolutore per formare le uscite da scrivere in memoria

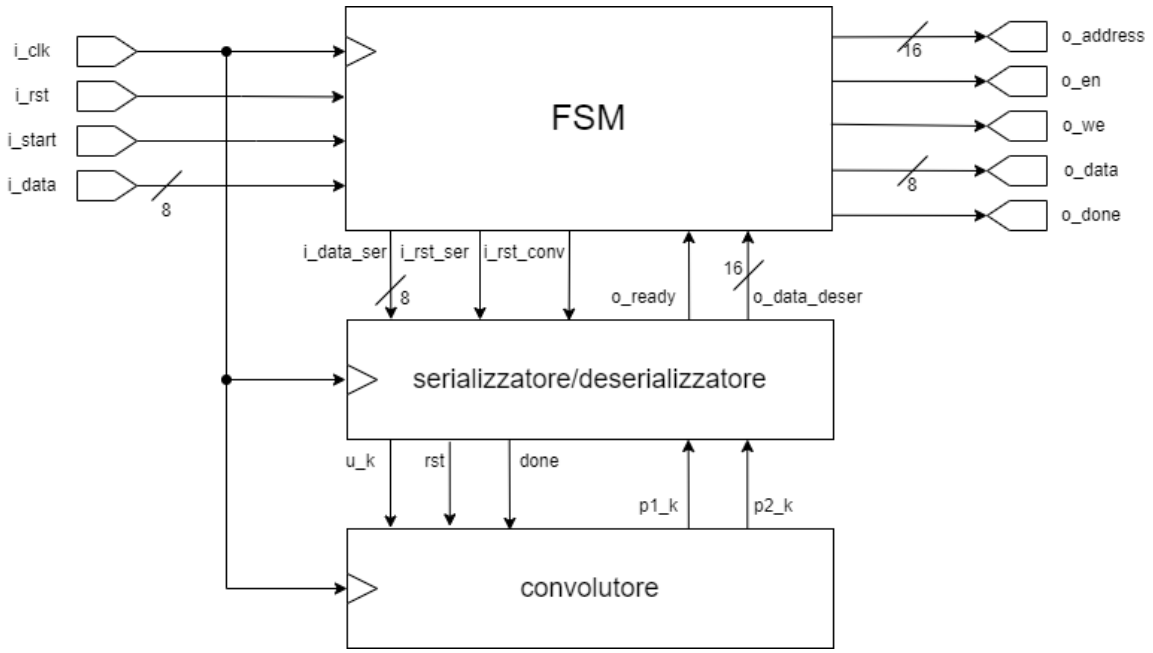

Figura 3: Lo schema ad alto livello dell'implementazione

- `Convolutore`: modulo che si occupa di codificare i singoli bit da produrre in uscita

La Figura 3 mostra uno schema ad alto livello dell'implementazione.
Tutti e tre i moduli sono macchine a stati ed hanno il medesimo segnale di `CLOCK`; inoltre, tutte e tre sono realizzate tramite due processi: `state_output` e `delta_lambda`. Il primo è il processo sequenziale che ha il compito di asserire le uscite e di cambiare lo stato interno sul fronte di salita del clock. Il secondo è il processo combinatorio che calcola le uscite e lo stato interno validi per il prossimo fronte di salita.

2.2 FSM principale

La FSM principale è una macchina a stati avente i seguenti 11 stati:

- `RST`: stato di attesa in cui la macchina aspetta che `i_start` venga portato ad 1 per poter iniziare la computazione di una nuova sequenza di parole in ingresso. Nel caso in cui venga alzato `i_rst` si ritorna in questo stato.
- `START`: stato in cui viene preparato l'indirizzo iniziale in uscita per poter leggere la lunghezza della sequenza di ingresso. Viene, inoltre, preparato il reset del Convolutore (poiché si è all'inizio di una nuova computazione).
- `READ_NUM_BYTE`: stato di attesa della RAM, si aspetta un ciclo di clock per permettere alla RAM di poter asserire l'uscita correttamente.
- `SAVE_NUM_BYTE`: stato in cui si salva all'interno di un registro a 8 bit, detto `num_word_in`, il valore contenuto nella prima cella di memoria della RAM, che rappresenta il numero di parole in ingresso. Tale registro servirà poi per identificare la fine della computazione della sequenza di parole.
- `SET_ADDR`: stato in cui si prepara l'indirizzo di memoria dal quale verrà letta la prossima parola. L'indirizzo di lettura viene memorizzato nel registro `o_address_read` e, prima di modificarlo (viene incrementato di uno ad ogni lettura), gli 8 bit meno significativi di tale registro vengono confrontati con `num_word_in`: se essi sono uguali allora la computazione è finita in quanto sono già state lette tutte le parole.
- `READ_WORD`: stato in cui si attende che la RAM asserisca l'uscita correttamente.

- `SAVE_WORD`: stato in cui la parola in uscita dalla RAM viene salvata nel registro a 8 bit `data_ser`, che costituirà l'ingresso del Serializzatore nel ciclo di clock successivo . In questo stato viene inoltre preparato il reset del serializzatore (il quale, dal ciclo di clock, successivo, inizierà a serializzare la parola in ingresso).

- `CALC_OUT`: stato in cui si attende che la codifica delle due parole da scrivere in RAM sia pronta. La codifica della parola viene salvata in un apposito registro a 16 bit, detto `o_data_tmp`, che conterrà la concatenazione delle due parole da scrivere in memoria.

- `MEM_WRITE_FIRST`: stato in cui si prepara la scrittura della prima parola in memoria (contenuta in `o_data_tmp(15 downto 8)`), la cui scrittura effettiva avverrà nel ciclo di clock successivo.

- `MEM_WRITE_SECOND`: stato in cui si prepara la scrittura della seconda parola in memoria (contenuta in `o_data_tmp(7 downto 0)`), la cui scrittura effettiva avverrà nel ciclo di clock successivo.

- `DONE`: stato che sancisce la fine della computazione. In questo stato, il segnale `o_done` viene portato al livello logico alto e rimane tale fino a che il segnale `i_start` non diventa 0.

Di seguito è riportato il diagramma degli stati, dove sono state fatte le seguenti abbreviazioni per alcuni degli stati:

READ_NUM_BYTE:	READ_NB
SAVE_NUM_BYTE:	SAVE_NB
SET_ADDR:	SET_ADD
READ_WORD:	READ_WR
SAVE_WORD:	SAVE_WR
MEM_WRITE_FIRST:	MEM_WR1
MEM_WRITE_SECOND:	MEM_WR2

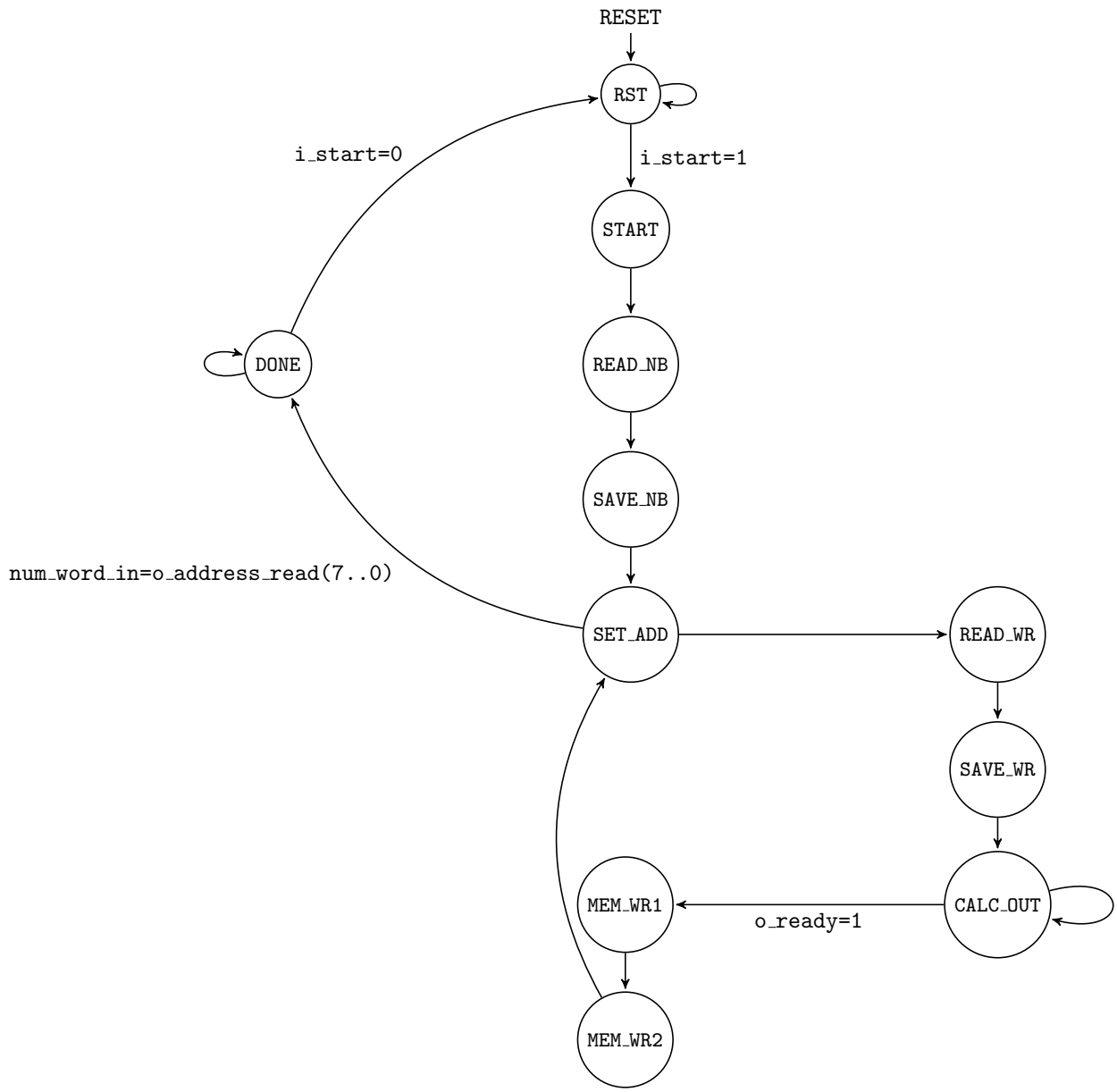

Figura 4: Diagramma degli stati della Macchina a Stati Finiti

2.3 Serializzatore/Deserializzatore

L'interfaccia del Serializzatore/Deserializzatore è la seguente:

```
entity serializzatore_deserializzatore is
     port (
          i_data_ser :  in std_logic_vector(7 downto 0);
          i_rst_ser :  in std_logic;
          i_clk :  in std_logic;
          i_rst_conv :  in std_logic;
          o_data_deser :  out std_logic_vector(15 downto 0);
          o_ready :  out std_logic
     );
end serializzatore_deserializzatore;
```

In particolare:

- `i_data_ser` è il byte da serializzare, proveniente dalla memoria
- `i_rst_ser` è il segnale di `RESET` del serializzatore
- `i_clk` è il segnale di `CLOCK` ed è lo stesso della FSM principale
- `i_rst_conv` è il segnale di `RESET` del Convolutore, che viene propagato dal Serializzatore/Deserializzatore verso il Convolutore
- `o_data_deser` è un segnale a 16 bit che forma il concatenamento delle due parole da scrivere in memoria alla fine dell'elaborazione di `i_data_ser` da parte del Convolutore
- `o_ready` è il segnale che notifica la fine dell'elaborazione del byte alla FSM principale

La presenza di due segnali di `RESET` è dovuta al fatto che il Serializzatore/Deserializzatore e il Convolutore devono esser resettati in due momenti diversi: il primo deve essere resettato quando inizia la computazione di una nuova parola, il secondo quando inizia la computazione di una nuova sequenza di parole. I due segnali vengono entrambi asseriti dalla FSM principale in maniera sincrona, dunque non avverrà mai un reset asincrono né del Serializzatore/Deserializzatore né del Convolutore.
La macchina a stati esegue un ciclo di conteggio da 0 a 9:

- il ciclo da 0 a 7 si occupa di serializzare il dato in ingresso, fornendo il bit di ingresso del convolutore: quando il conteggio è 0 si prepara il bit più significativo di `i_data_ser` (ovvero `i_data_ser(7)`) come ingresso del Convolutore, quando il conteggio è 7 si prepara il bit meno significativo di `i_data_ser` (ovvero `i_data_ser(0)`) come ingresso del Convolutore; dunque, al ciclo di conteggio i viene preparato `i_data_ser(7-i)` come ingresso del Convolutore. Poichè la modifica del'ingresso vero e proprio del Convolutore avviene sul fronte di salita del clock, la computazione effettiva del Convolutore avviene nel ciclo di conteggio da 1 a 8.
- il ciclo da 2 a 9 si occupa di raccogliere i due bit di uscita del Convolutore nel segnale `o_data_deser`; i bit vengono deserializzati in `o_data_deser` partendo dal bit più significativo (`o_data_deser(15)`).

Alla fine del conteggio, il modulo porta il segnale `o_ready` a 1, dunque la macchina si mantiene nello stesso stato (senza produrre nuovi ingressi per il Convolutore) fino a che non avviene un reset. Inoltre, il modulo comunica con il Convolutore tramite un segnale `done`, che viene alzato quando il conteggio è 8 (e viene mantenuto alto fino a quando il conteggio non ritorna a 0) e che notifica al Convolutore di non modificare più le sue uscite e il suo stato interno fino a quanto tale segnale non ritorna a 0.

2.4 Convolutore

Il Convolutore è una macchina a stati avente la seguente interfaccia:

```
entity convolutore is
     port (
          u_k :  in std_logic;
          clk :  in std_logic;
          rst :  in std_logic;
          done :  in std_logic;
          p1_k :  out std_logic;
          p2_k :  out std_logic
     );
end convolutore;
```

Dove:

- `u_k` è il bit d'ingresso da codificare
- `clk` è il segnele di `CLOCK`
- `rst` è il segnale di `RESET`

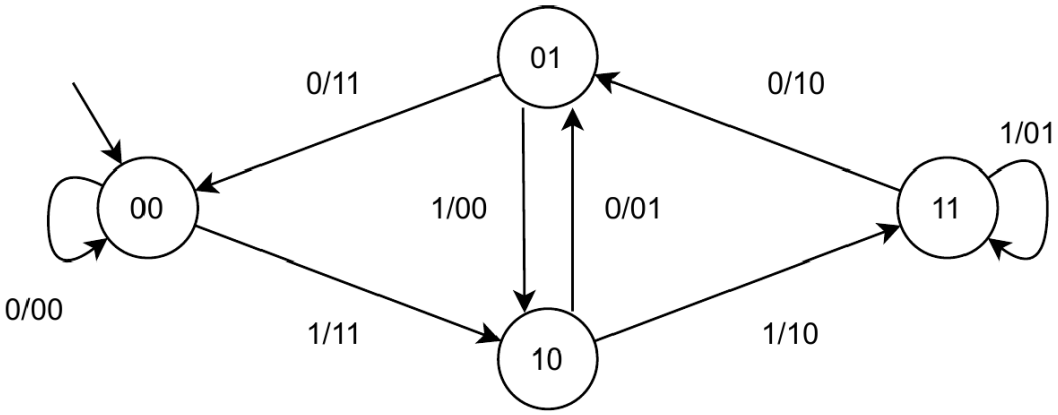

Figura 5: Il diagramma degli stati del Convolutore con `done` uguale a 0

- `done` è un segnale proveniente dal Serializzatore/Deserializzatore: se è 1 allora la serializzazione è finita, dunque il Convolutore non cambia più il suo stato e non modifica più le sue uscite, se è a 0 allora il Convolutore si comporta normalmente
- `p1_k` e `p2_k` sono i due bit di uscita, che rappresentano la codifica di `u_k`.

La Figura 5 riporta il diagramma degli stati del Convolutore, considerando `done` pari a 0.

2.5 Scelte progettuali e ottimizzazioni

L'architettura è stata realizzata in maniera modulare per poter per poter separare le funzionalità di calcolo della codifica, di serializzazione della parola di ingresso e di interfacciamento con la memoria; questo ha permesso di parallelizzare la serializzazione e il calcolo della codifica dell'uscita.
Le ottimizzazioni hanno riguardato soprattutto la FSM principale; in particolare, si è scelto di utilizzare un set di registri per memorizzare l'indirizzo di lettura della memoria (incrementato di uno ad ogni lettura) e di non memorizzare l'indirizzo di scrittura, in quanto quest'ultimo viene ricavato da quello di lettura. Questo ha permesso di risparmiare sull'utilizzo di qualche registro, poiché l'eventuale memorizzazione dell'indirizzo di scrittura richiederebbe ulteriori registri per poterlo incrementare di 1 ad ogni scrittura.
Un'ulteriore ottimizzazione riguarda il processo di scrittura: la scrittura effettiva in memoria della seconda parola avviene quando la FSM principale si trova nello stato `SET_ADDR`, in questo modo la scrittura della parola e la preparazione dell'indirizzo di memoria successivo da cui leggere avvengono nello stesso stato; lo stesso avviene anche con la prima parola da scrivere, che viene scritta in memoria quando la FSM principale sta preparando l'indirizzo di scrittura della seconda parola da scrivere. Questo ha consentito di risparmiare due cicli di clock per ogni parola da codificare in uscita.

3 Risultati sperimentali

3.1 Report di sintesi

Il componente risulta correttamente simulabile con l'utilizzo di 144 Lookup Tables e di 100 Flip Flop, come lo si può notare dalla Figura 6. Si è fatta particolare cautela nella scrittura del codice per evitare l'uso di Latch.

3.2 Simulazioni

Sono stati effettuati alcuni test per verificare condizioni di funzionamento ritenute significative; in particolare, oltre al testbench d'esempio, sono stati scritti i seguenti testbench:

- Indirizzo 0 della RAM contenente la parola "00000000": test che rappresenta il caso limite in cui la lunghezza della sequenza di ingresso è nulla

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	144	0	63400	0.23
LUT as Logic	144	0	63400	0.23
LUT as Memory	0	0	19000	0.00
Slice Registers	100	0	126800	0.08
Register as Flip Flop	100	0	126800	0.08
Register as Latch	0	0	126800	0.00
F7 Muxes	0	0	31700	0.00
F8 Muxes	0	0	15850	0.00

Figura 6: Il report di utilizzo

- Segnale di `RESET` asincrono: il segnale di reset viene asserito dal testbench nel mezzo della computazione, con l'obiettivo di verificare che la macchina riprenda la computazione dall'inizio
- Lunghezza della sequenza di ingresso pari a 255: l'obiettivo del test è verificare se il modulo progettato è in grado di gestire il caso limite in cui la lunghezza della sequenza di ingresso è massima
- Due sequenze di parole: test che mira a verificare il corretto funzionamento del protocollo per la gestione più sequenze di parole

Oltre ai test descritti, sono stati usati anche ulteriori testbench generati casualmente, mantenendo la validità della specifica. Per tutti i test è stata effettuata la simulazione behavioral ed in seguito le simulazioni functional e timing post-synthesis, tutte con successo. Si è, inoltre, provato a variare anche il periodo di clock: il componente ha mantenuto il suo funzionamento anche a 1 ns nella simulazione Functional Post-Synthesis e a 5 ns nella simulazione Timing Post-Synthesis.

4 Conclusioni

Si ritiene che il componente progettato rispetti la specifica, il che è stato verificato mediante testing estensivo sia in pre-sintesi che in post-sintesi. I test effettuati hanno evidenziato alcune criticità nella soluzione iniziale, consentendo di sviluppare la soluzione fino alla sua versione finale. Lavorare alla soluzione dei problemi ha permesso di comprendere bene il funzionamento di Vivado e di migliorare la capacità di risolvere i problemi in maniera efficiente. In particolare, l'analisi delle forme d'onda è stata importante per sviluppare al meglio il progetto.