Terzo anno Ing. Informatica Politecnico di Milano

Progetto Finale di Reti Logiche

Giovanni Barbiero - 10722159

28 marzo 2022

1 Introduzione

Lo scopo del progetto è quello di descrivere in linguaggio VHDL e di sintetizzare un componente hardware che legga i dati dalla memoria, effettui una codifica convoluzionale con rapporto $\frac{1}{2}$ ed infine scriva il risultato in memoria.

1.1 Descrizione generale

La memoria da cui il componente legge e a cui scrive i dati è sincrona, a indirizzamento a byte e con uno spazio degli indirizzi a 16 bit. Il componente, servendosi della memoria, segue in ordine i seguenti passaggi:

1. legge all'indirizzo di memoria `0x0000` il numero di parole che dovrà codificare.
2. partendo dall'indirizzo `0x0001`, serializza le parole ottenendo una sequenza di bit.
3. su questa sequenza applica la codifica convoluzionale $\frac{1}{2}$ raddoppiando, così, il numero di bit.
4. la nuova sequenza di bit viene infine scritta in memoria a partire dall'indirizzo `0x03E8`.

La sequenza di parole da codificare è al massimo di 255, inoltre il componente deve funzionare correttamente con un periodo di clock di almeno `100 ns`.

1.2 Codifica convoluzionale

Il codificatore convoluzionale con tasso di trasmissione $\frac{1}{2}$ è un componente che per ogni bit che riceve in ingresso genera la sua codifica in 2 bit. Si può rappresentare come una macchina a stati di Mealy con un bit per l'ingresso e 2 bit per l'uscita come in Figura 1.

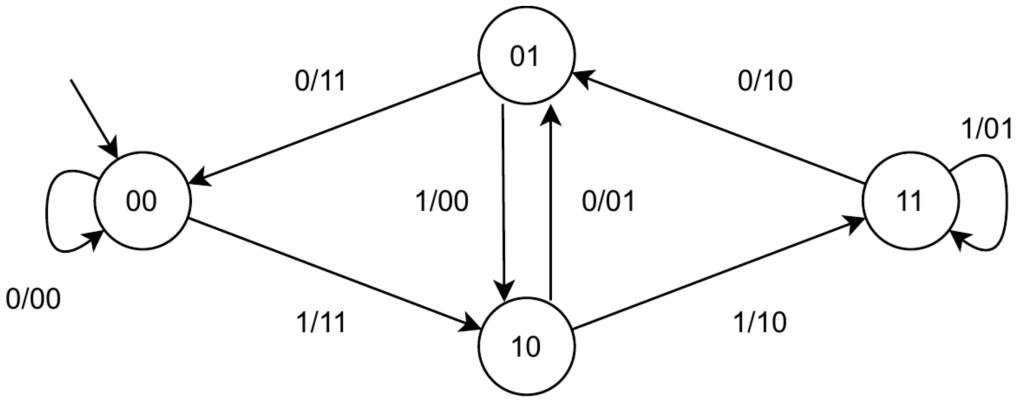

Figura 1: Codifica convoluzionale con rapporto $\frac{1}{2}$.

1.3 Esempio di funzionamento

Il componente legge dalla memoria (Tabella 1) all'indirizzo `0x0000` il numero di parole totale da codificare. Successivamente partendo dall'indirizzo `0x0001` legge la prima parola, la quale viene serializzata e il singolo bit viene dato in ingresso al codificatore convoluzionale come in Tabella 3. Quest'ultimo produce due bit in ogni istante `t`, i quali vengono deserializzati e scritti in memoria a partire dall'indirizzo `0x03E8`.
Una volta codificata la prima parola il codificatore continua con le successive fino ad averle codificate tutte. Il contenuto della memoria alla fine della codifica è riportato in Tabella 2.

Indirizzo	Contenuto
0x0000	00000010
0x0001	10100010
0x0002	01001011

Tabella 1: Contenuto iniziale memoria.

Indirizzo	Contenuto
0x03E8	11010001
0x03E9	11001101
0x03EA	11110111
0x03EB	11010010

Tabella 2: Contenuto finale memoria.

t	0	1	2	3	4	5	6	7
in	1	0	1	0	0	0	1	0
out_1	1	0	0	0	1	0	1	0
out_2	1	1	0	1	1	0	1	1
t	8	9	10	11	12	13	14	15
in	0	1	0	0	1	0	1	1
out_1	1	1	0	1	1	0	0	1
out_2	1	1	1	1	1	1	0	0

Tabella 3: Codifica convoluzionale al tempo t.

2 Architettura

Il componente possiede 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;
```

L'elaborazione inizia quando viene alzato il segnale `i_start` e termina quando il componente alza il segnale `o_done`. Solo successivamente il segnale `i_start` dovrà essere abbassato.
Il componente è stato strutturato in vari moduli e in una macchina a stati che ne coordina l'interoperabilità al fine di ottenere un corretto funzionamento secondo le specifiche. In Figura 2 si può osservare come i moduli sono connessi tra loro. A tutti i moduli, compresa la macchina a stati, vengono forniti il segnale di clock `i_clk` e il segnale di reset `i_rst`.

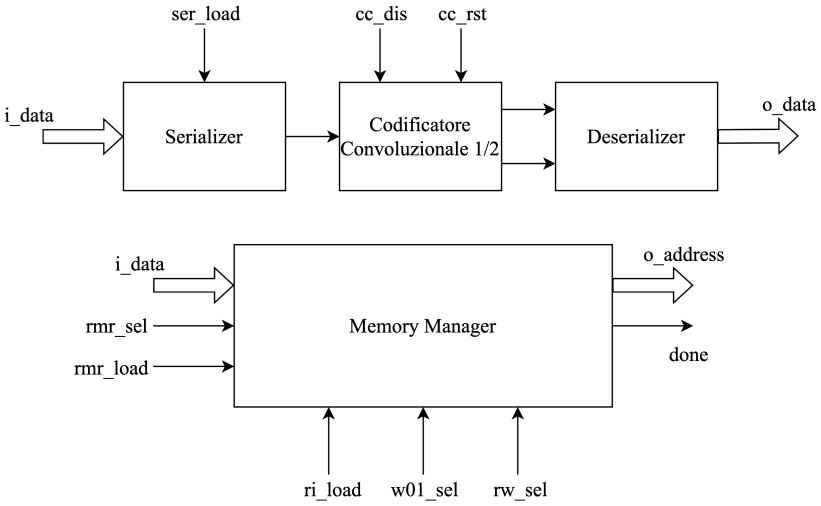

Figura 2: Connessioni tra i moduli.

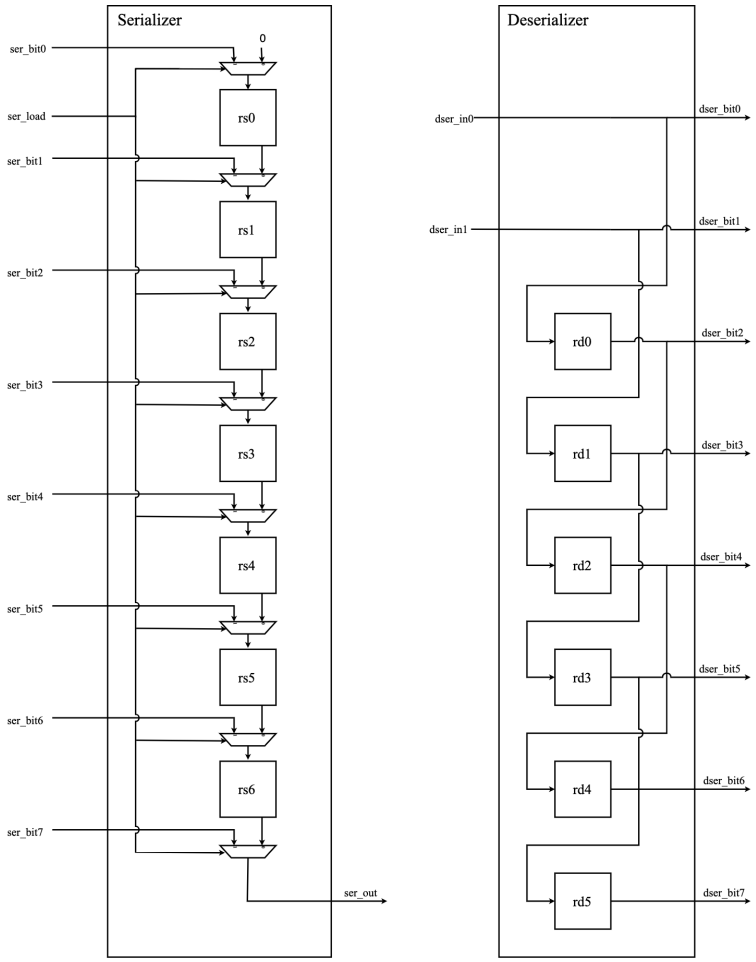

Figura 3: Schema implementativo di serializzatore e deserializzatore.

2.1 Serializzatore

Il serializzatore riceve in ingresso i dati dalla memoria in `i_data` e seguendo il segnale di clock mette in uscita un bit alla volta i dati letti partendo dal più significativo.
Come mostrato in modo dettagliato in Figura 3, il serializzatore è un registro a scorrimento di tipo PISO (parallel input-serial output) a 8 bit dove l'ultimo flip flop è stato rimosso per risparmiare un ciclo di clock. Quando il segnale `ser_load = 1` viene memorizzato `i_data`, altrimenti viene effettuato lo shift.

2.2 Deserializzatore

Il deserializzatore riceve in ingresso due bit `dser_in0` e `dser_in1` e seguendo il segnale di clock appende ai bit presenti i due nuovi bit.
Come mostrato in modo dettagliato in Figura 3, il deserializzatore è un registro a scorrimento di tipo SIPO (serial input-parallel output) a 8 bit con uno shift di 2 bit alla volta e i primi due flip flop rimossi per risparmiare un ciclo di clock.
L'uscita del deserializzatore viene poi direttamente collegata ad `o_data` per poter essere scritta in memoria quando i segnali `o_en` e `o_we` vengono alzati.

2.3 Codificatore convoluzionale

Il codificatore convoluzionale con rapporto $\frac{1}{2}$ è l'implementazione della macchina a stati in Figura 1.
Il segnale `cc_dis` serve per bloccare l'avanzamento della macchina a stati, mentre il segnale `cc_rst` per resettare la macchina a stati e portarla allo stato iniziale `00`. Questo risulta l'unico modulo ad essere scollegato dal segnale `i_rst`, però, come verrà mostrato in seguito, quando la macchina a stati si trova nello stato di reset allora alza il segnale `cc_rst` resettando così anche il codificatore convoluzionale.

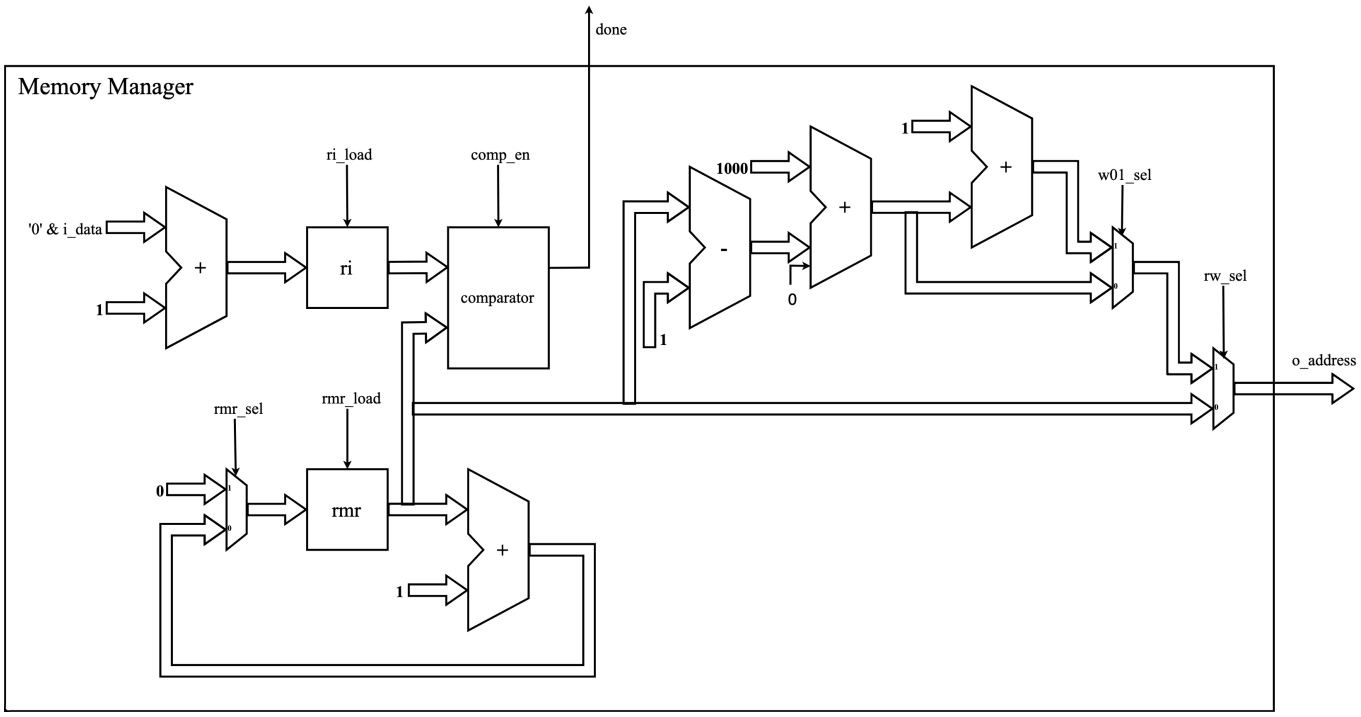

Figura 4: Struttura interna del gestore della memoria.

2.4 Gestore della memoria

Il gestore della memoria ha il compito di tener traccia di quante parole devono essere lette e gestire gli indirizzi di memoria sia per la lettura che per la scrittura.
La struttura interna viene mostrata in Figura 4.
Tramite il segnale `ri_load` viene salvato il numero di parole totale da leggere +1 nel registro `ri` a 9 bit. Nel

registro `rmr` viene salvato l'indirizzo della parola corrente e tramite il segnale `rmr_load` ne viene aggiornato il valore. Quest'ultimo in base a `rmr_sel` può essere il valore precedente incrementato di uno oppure `0`. Il segnale `rmr_sel` viene utilizzato per questo scopo, cioè di riportare il contatore a zero per poter consentire la codifica di nuovi flussi senza reset.
Dato che per ogni parola letta se ne scrivono due, si può calcolare l'indirizzo dove scrivere il risultato della codifica come $\texttt{0x03E8} + 2 \cdot \texttt{n}$ e $\texttt{0x03E8} + 2 \cdot \texttt{n} + 1$ dove `n` corrisponde al valore del registro `rmr`.
Il segnale `w01_sel` serve per selezionare in quale tra i due indirizzi scrivere il risultato della codifica, mentre il segnale `rw_sel` serve per selezionare tra gli indirizzi di lettura e quelli di scrittura.
Inoltre, in modo asincrono, viene effettuato il confronto tra i valori dei registri `ri` e `rmr`. Se questi ultimi coincidono significa che sono state lette tutte le parole, quindi il segnale interno `done` viene portare a `1`.
I registri `ri` e `rmr` sono a 9 bit poiché le parole massime che il componente deve poter codificare sono 255 che richiede almeno 9 bit per essere codificato in binario.

2.5 Macchina a stati

La macchina a stati (vedi Figura 5) ha lo scopo di gestire i segnali di comando in input al componente e di modificare i segnali interni per consentire la corretta interoperabilità dei vari moduli.
È una macchina a stati di Moore ed è formata dai seguenti stati:

- `IDLE`: Stato iniziale in cui si resetta sia il contatore degli indirizzi di memoria sia il codificatore convoluzionale. Inoltre questo stato rappresenta anche lo stato di reset della macchina.
- `READ_0`: Quando viene alzato il segnale `i_start` si arriva in questo stato. Viene richiesto alla memoria il contenuto all'indirizzo `0x0000` e il codificatore convoluzionale viene disabilitato.
- `SAVE`: In questo stato viene salvato il valore che arriva dalla memoria nel registro `ri` e il codificatore rimane disabilitato.
- `READ_I`: Si richiede alla memoria il contenuto dell'indirizzo salvato nel registro `rmr` e si continua a tener disabilitato il codificatore convoluzionale.
- `COMP_00`: Viene caricata la parola nel serializzatore e il codificatore convoluzionale, ora abilitato, ne codifica il primo bit.
- `COMP_01`: Passaggio intermedio di codifica della parola: viene codificato il secondo bit.
- `COMP_02`: Passaggio intermedio di codifica della parola: viene codificato il terzo bit.
- `WRITE_0`: Passaggio intermedio di codifica della parola: viene codificato il quarto bit. Inoltre la codifica della prima metà della parola viene scritta in memoria all'indirizzo $\texttt{0x03E8} + 2 \cdot \texttt{rmr}$.
- `COMP_10`: Passaggio intermedio di codifica della parola: viene codificato il quinto bit.
- `COMP_11`: Passaggio intermedio di codifica della parola: viene codificato il sesto bit.
- `COMP_12`: Passaggio intermedio di codifica della parola: viene codificato il settimo bit.
- `WRITE_1`: Ultimo passaggio di codifica della parola: viene codificato l'ottavo bit. Inoltre la codifica della seconda metà della parola viene scritta in memoria all'indirizzo $\texttt{0x03E8} + 2 \cdot \texttt{rmr} + 1$.
- `WAIT_S`: Se la macchina a stati si trova nello stato di `READ_I` e il segnale `done` è alto allora si arriva in questo stato. Si continua a rimanere in questo stato con `o_done = 1` fintanto che `i_start` rimane alzato. Quando `i_start` viene abbassato la macchina a stati si porta nello stato `IDLE`.

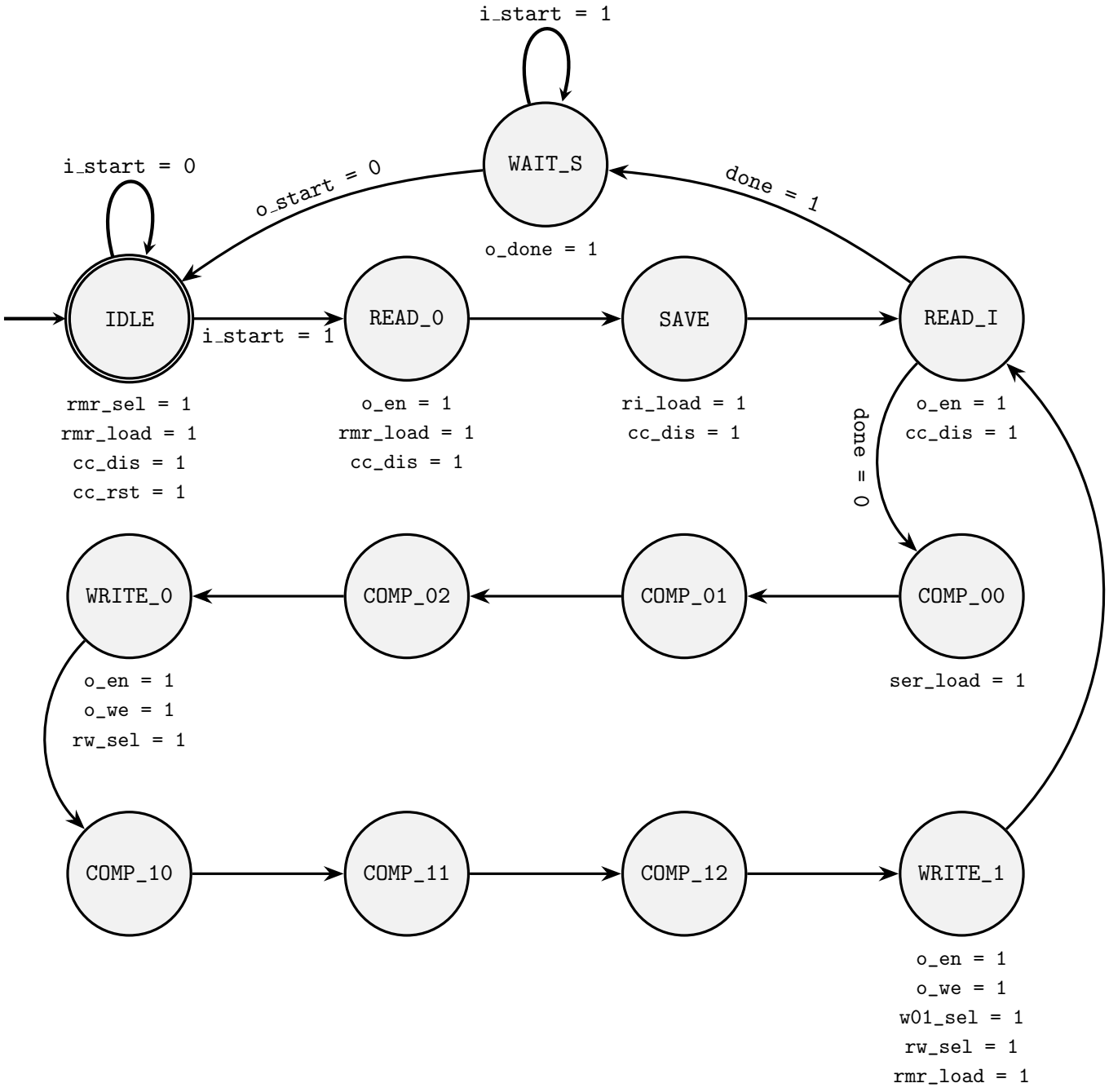

Figura 5: Macchina a stati finiti di Moore. Negli stati tutti i segnali che sono stati omessi sono uguali a 0.

3 Risultati sperimentali

3.1 Sintesi

Effettuando la sintesi risulta il seguente utilizzo:

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	43	0	0	134600	0.03
LUT as Logic	43	0	0	134600	0.03
LUT as Memory	0	0	0	46200	0.00
Slice Registers	46	0	0	269200	0.02
Register as Flip Flop	46	0	0	269200	0.02
Register as Latch	0	0	0	269200	0.00
F7 Muxes	0	0	0	67300	0.00
F8 Muxes	0	0	0	33650	0.00

Come si può notare dal report sull'utilizzo, non sono stati inferiti dei latch, i quali, se non intenzionalmente inseriti, avrebbero potuto compromettere il corretto funzionamento del componente.
Secondo le specifiche di progetto il componente deve poter funzionare con un periodo di clock di almeno `100ns`. Analizzando il report sul timing (Figura 6) possiamo notare che il Worst Negative Slack (WNS) è di `96.744ns`, il quale sta ad indicare che il componente potrebbe funzionare correttamente anche con periodi di clock molto più bassi.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	96,744 ns	Worst Hold Slack (WHS):	0,134 ns	Worst Pulse Width Slack (WPWS):	49,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	72	Total Number of Endpoints:	72	Total Number of Endpoints:	47

All user specified timing constraints are met.

Figura 6: Report timing del componente sintetizzato.

3.2 Simulazioni

Sono stati creati dei test bench per testare il corretto funzionamento del componente secondo le specifiche anche nei casi limite. Il componente passa tutti i test sia in Behavioral Simulation che in Post-Synthesis Functional Simulation.

3.2.1 Test Bench 1: Sequenza minima e massima

Il componente si comporta correttamente anche per la codifica di sequenza minima e massima.
Nel caso della sequenza minima viene scritto zero all'indirizzo `0x0000`. In questo modo il componente, come in Figura 7, dopo aver letto che il numero di parole da elaborare è zero, alza il segnale `o_done` e termina la codifica.
Nel caso di sequenza massima viene inserito 255 all'indirizzo `0x0000`. Come mostrato in Figura 8, il componente riesce a codificare tutte le parole compresa l'ultima.

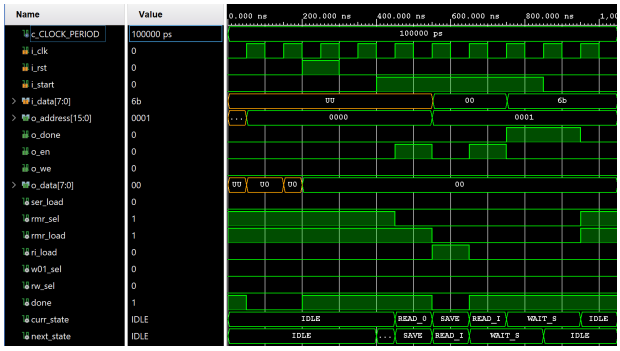

Figura 7: Simulazione test bench 1: sequenza minima.

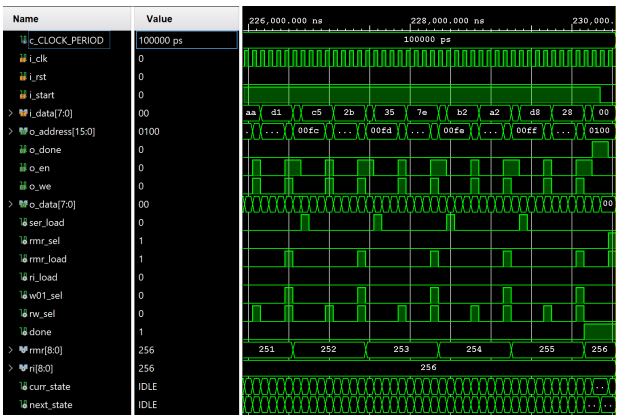

Figura 8: Simulazione test bench 1: sequenza massima (solo parte finale).

3.2.2 Test Bench 2: Reset

Il componente riceve durante la codifica un segnale di reset e, come in Figura 9, riesce a ricominciare la codifica correttamente dopo aver ricevuto un nuovo segnale di start.

Figura 9: Simulazione test bench 2: reset.

3.2.3 Test Bench 3: Codifica continua

In questo test bench viene testata la capacità del componente di codificare più sequenze senza ricevere un reset. Come mostrato in Figura 10, il componente una volta terminata una sequenza riesce a iniziare correttamente la codifica di una nuova dopo aver ricevuto un nuovo segnale di start.

3.2.4 Test Bench 4: Stress Test

Il componente viene sottoposto ad uno stress test facendogli codificare 10000 sequenze di dimensione variabile tra 0 e 255, rispettivamente minimo e massimo di parole codificabili dal componente in una singola sequenza.
Questo test bench ha portato alla scoperta di un errore nella gestione del segnale `o_done`. Inizialmente `o_done` veniva collegato direttamente al segnale `done`, però questo lo portava in rari casi ad alzarsi erroneamente mentre la codifica era ancora in corso. È stato possibile risolvere questo problema semplicemente separando i due segnali e facendo in modo che fosse la macchina a stati a modificare `o_done`.

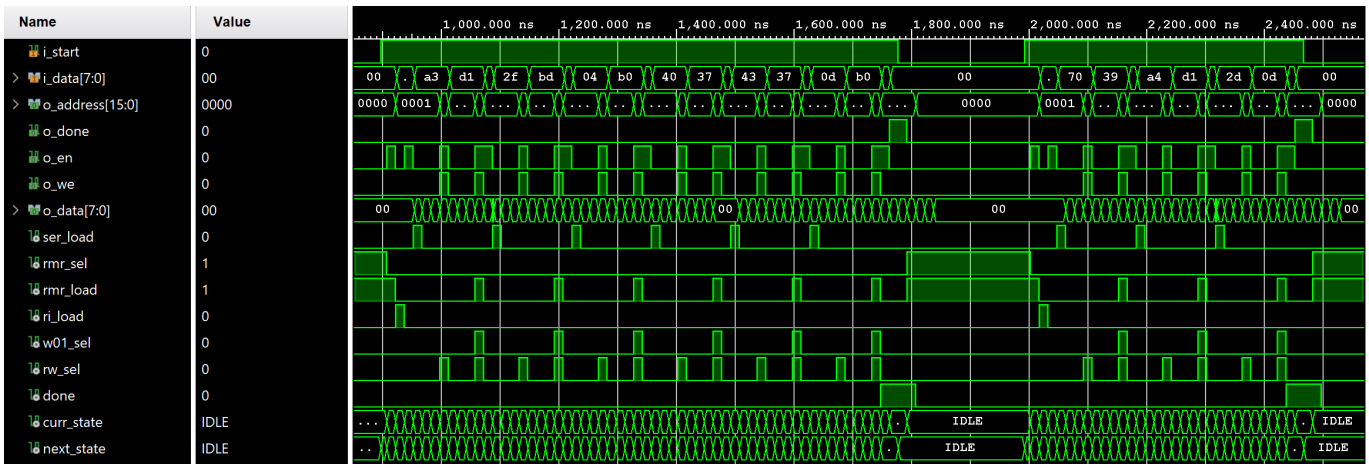

Figura 10: Simulazione test bench 3: codifica continua.

4 Conclusioni

Si ritiene che l'architettura del componente che è stata progettata rispetti pienamente le specifiche. Ciò è stato verificato e confermato dai vari test effettuati. Inoltre l'architettura è stata pensata per diminuire il più possibile l'overhead alla codifica utilizzando il minimo numero di stati possibili e per minimizzare l'utilizzo di risorse. Infine il componente è stato suddiviso in più moduli per rendere più semplice la sua progettazione e per facilitare l'identificazione e correzione degli errori.