POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Prova finale – Reti Logiche

PROGETTO PROVA FINALE DI RETI LOGICHE
CODIFICATORE CONVOLUZIONALE

Alice Portentoso

Matricola:	934939
Codice Persona:	10664207
Anno Accademico:	2021-2022

Sommario

1. Introduzione 3
2. Codificatore Convoluzionale 3
2.1 Come nasce 3
2.2 Come funziona 3
2.3 Esempio 4
3. Specifica 5
3.1 Interfaccia del componente 5
3.2 Memoria 6
4. Architettura 6
4.1 Segnali 6
4.2 Stati 7
4.3 Diagrammi dei moduli 8
5. Risultati sperimentali 10
5.1 Report di sintesi 10
5.2 Test Bench 10
6. Conclusioni 11

1. Introduzione

Lo scopo di questo progetto è la realizzazione di un modulo hardware composto di un codificatore convoluzionale (descritto in seguito) e la sua interfaccia con una memoria. Il modulo legge dalla memoria *N* parole, ognuna di 8 bit, e scrive in memoria *2*N* parole.

2. Codificatore Convoluzionale

2.1 Come nasce

Un problema critico nell'ambito delle telecomunicazioni è l'affidabilità dei segnali trasmessi. Durante il flusso di trasmissione digitale alcune interferenze possono introdurre rumore, che rende impossibile per il destinatario ricevere e interpretare correttamente il dato.

Nel 1948 Shannon dimostrò che gli errori indotti dal rumore possono essere ridotti notevolmente utilizzando opportune codifiche del segnale senza sacrificare la velocità della trasmissione del canale. Shannon si limitò a dimostrarne l'esistenza teorica e negli anni successivi si assistette a un enorme sviluppo di algoritmi per la codifica e decodifica di segnali.

2.2 Come funziona

I codici di correzione degli errori possono essere a blocchi e convoluzionali. Mentre i primi sono senza memoria e consistono nella suddivisione in blocchi del messaggio in ingresso, nei secondi la codifica del bit dipende dall'ingresso e dallo stato, che a sua volta dipende dai due bit precedenti, seguendo lo schema mostrato in Figura 1.

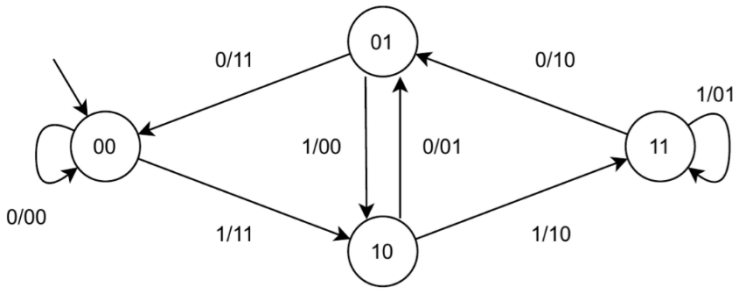


Figura 1: Macchina a stati del Codificatore Convoluzionale

I codici convoluzionali si differenziano per il Code Rate cioè il rapporto tra il numero di bit di ingresso e di uscita. Questo vale sempre *1/K* con *K* numero di porte XOR. Nel nostro caso viene usato un convolutore *1/2* quindi per *N* parole in ingresso avremo *2*N* parole in uscita.

Il codificatore è composto di due Flip Flop di tipo D che consentono di tenere memoria dei due bit precedenti. Come mostrato nella Figura 2 il modulo calcola due OR esclusivi e il flusso finale viene ottenuto concatenando in modo alternato i bit di uscita.

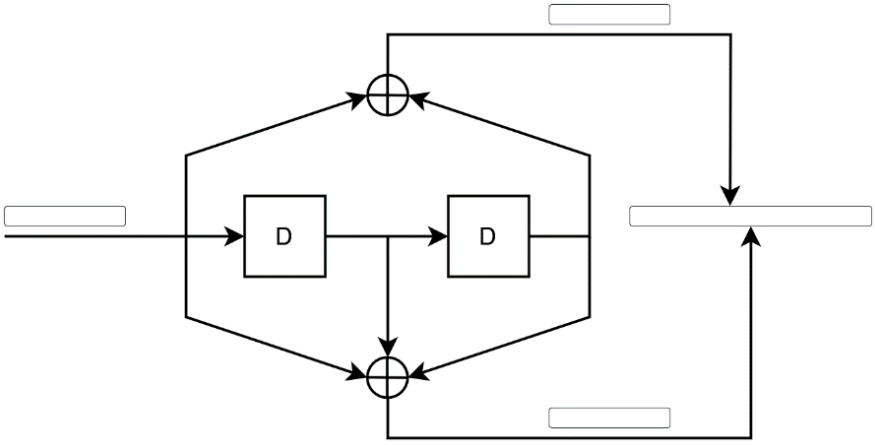


Figura 2: Codificatore Convoluzionale con Code Rate ½

2.3 Esempio

Sia *U(t)* il bit di ingresso al tempo *t*, *U(t-1)* il bit memorizzato nel primo Flip Flop e *U(t-2)* nel secondo. Siano *P1(t)* e *P2(t)* i bit di uscita rispettivamente della prima e della seconda porta XOR e *Y(t)* l'uscita del convolutore.

A	B	C	A⊕B⊕C	A⊕C
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	1
1	0	0	1	1
1	0	1	0	0
1	1	0	0	1
1	1	1	1	0

Figura 3: Tabella di verità operatore XOR

Allora un esempio di funzionamento del modulo è il seguente, dove le frecce indicano l'operatore XOR a 2 o 3 operandi come mostrato in Figura 3.

t	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***
U	1	0	1	0	0	0	1	0
P1	1	0	0	0	1	0	1	0
P2	1	1	0	1	1	0	1	1

Y 1 1 0 1 0 0 0 1 1 1 0 0 1 1 0 1

3. Specifica

Viene richiesta l'implementazione di un modulo hardware del convolutore appena descritto e la sua interfaccia con una memoria.

Come mostrato in Figura 4 l'elaborazione inizia quando viene portato a 1 il segnale d'ingresso *i_start*. Una volta terminata la codifica di una sequenza di parole e dopo averla scritta in memoria, il modulo deve portare alto il segnale *o_done*. Questo deve rimanere alto finché il segnale *i_start* non viene riportato basso. Se, a questo punto, il modulo riceve un nuovo segnale *i_start*, ricomincia la fase di codifica.

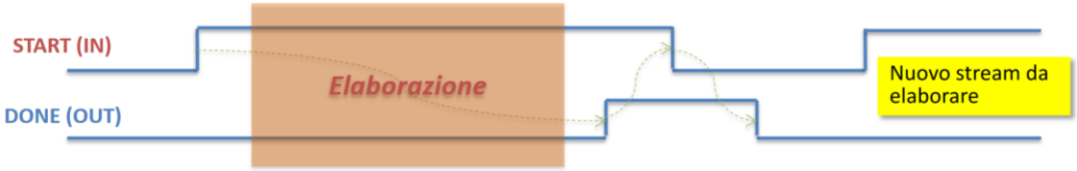
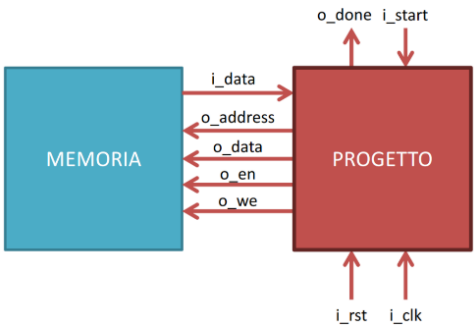


Figura 4: Protocollo di elaborazione

Questo significa che il modulo da implementare deve poter codificare più flussi in sequenza. Inoltre, prima della prima codifica, verrà sempre dato il segnale di reset, non necessario invece tra una codifica e l'altra.

3.1 Interfaccia del componente

Il componente da descrivere deve avere 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;
```

- I segnali *i_clk* e *i_rst* sono i segnali di clock e di reset generati dal TestBench.
- Il segnale *i_start* viene generato dal TestBench e indica l'inizio della codifica.
- Il segnale *i_data*, di 8 bit, arriva dalla memoria in seguito a una richiesta di lettura.
- Il segnale *o_address*, di 16 bit, è il segnale di uscita che manda alla memoria l'indirizzo in cui si vuole leggere o scrivere.
- Il segnale *o_done* è il segnale di uscita che comunica la fine dell'elaborazione.

- Il segnale *o_en* abilita l'accesso alla memoria, sia in lettura sia in scrittura.
- Il segnale *o_we*, di write enable, abilita la scrittura in memoria, mentre deve essere 0 durante la lettura.
- Il segnale *o_data*, di 8 bit, è il segnale di uscita dal componente verso la memoria.

3.2 Memoria

Il modulo da implementare legge lo stream da una memoria con indirizzamento al byte. La memoria contiene all'indirizzo 00000000 il numero *N* di parole da codificare, mentre la sequenza di parole è memorizzata in ordine dall'indirizzo 00000001 all'indirizzo *N*. Dopo l'elaborazione il modulo deve scrivere le parole generate in ordine a partire dall'indirizzo $(1000)_{10}$ all'indirizzo $(2*N)_{10}$.

4. Architettura

4.1 Segnali

```
type S is (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11);
signal curr_state, next_state : S;
signal num_words : std_logic_vector (7 downto 0);
signal o_r1 : std_logic_vector (7 downto 0);
signal o_r2 : std_logic_vector (7 downto 0);
signal address_in : std_logic_vector (15 downto 0);
signal address_o : std_logic_vector (15 downto 0);
signal conv: std_logic_vector (15 downto 0);
signal pre_bit: std_logic_vector (1 downto 0);
```

Figura 5: Dichiarazione dei segnali

- I segnali *curr_state* e *next_state* indicano rispettivamente lo stato corrente e lo stato prossimo e possono assumere uno tra i 12 stati esistenti.
- Il segnale *num_words*, di 8 bit, viene letto dalla memoria nell'indirizzo 0 durante lo stato S3 e contiene la quantità di parole da codificare. Questo segnale viene anche usato come contatore, poiché viene decrementato a ogni parola letta e la codifica termina quando questo segnale diventa minore di 0.
- Il segnale *o_r1*, di 8 bit, indica l'output del registro *r1*. Questo assume il valore d'ingresso *i_data* durante lo stato S5, 0 negli altri casi.
- Il segnale *pre_bit*, di 2 bit, memorizza gli ultimi 2 bit della parola precedente necessari per il calcolo convoluzionale della parola corrente.
- Il segnale *conv*, di 16 bit, è il segnale di uscita del codice convolutore. Questo viene calcolato con un insieme di combinazioni logiche di tipo XOR dei segnali *o_r1* e *pre_bit.*

- Il segnale *o_r2*, di 8 bit, indica l'output del registro *r2*. Questo assume in modo alternato i primi 8 bit e gli ultimi 8 del segnale *conv*. In particolare assume *conv* (15..8) nello stato S7 e *conv* (7..0) nello stato S9.
- La gestione degli indirizzi si compone di due segnali di supporto *address_in* e *address_o*. Il primo parte da 0 e incrementa di uno per ogni parola letta, durante lo stato S4; il secondo parte da 0 e incrementa di uno per ogni parola scritta, quindi di due per ogni parola letta, durante gli stati S7 e S9. Il segnale *o_address* viene settato uguale a *address_o* se lo stato è di scrittura, a *address_in* se lo stato è di lettura.

4.2 Stati

La macchina a stati del modulo viene mostrata in Figura 5. Sono stati omessi per semplicità i segnali *o_done*, *o_en*, *o_we* che valgono 0 quando non diversamente specificato e vengono mostrati solo nello stato S0.

In Figura 5 vengono mostrati in blu gli stati di lettura da memoria, in verde gli stati di scrittura in memoria e in giallo lo stato finale.

- S0: stato di attesa perché il segnale *i_start* diventi 1.
- S1: stato in cui vengono settati gli indirizzi di partenza di *address_in* e *address_o*.
- S2: stato di lettura nell'indirizzo 0 salvato in *num_words*.
- S3: stato per controllare che ci siano ancora parole da codificare.
- S4: stato per incrementare *address_in* e salvare gli ultimi 2 bit della parola utili per la codifica successiva.
- S5: stato per memorizzare la parola in input nel registro *r1*.
- S6: stato per il calcolo convoluzionale.
- S7: stato per salvare la prima parola del calcolo convoluzionale nel registro *r2*.
- S8: stato per scrivere in uscita la prima parola del calcolo convoluzionale.
- S9: stato per salvare la seconda parola del calcolo convoluzionale nel registro *r2*.
- S10: stato per scrivere in uscita la seconda parola del calcolo convoluzionale.
- S11: stato di terminazione. Si entra in questo stato quando non ci sono più parole da codificare. Qui *o_done* viene settato a 1 ed è anche uno stato di attesa per la prossima codifica, che può iniziare solo quando *i_start* viene riportato basso.

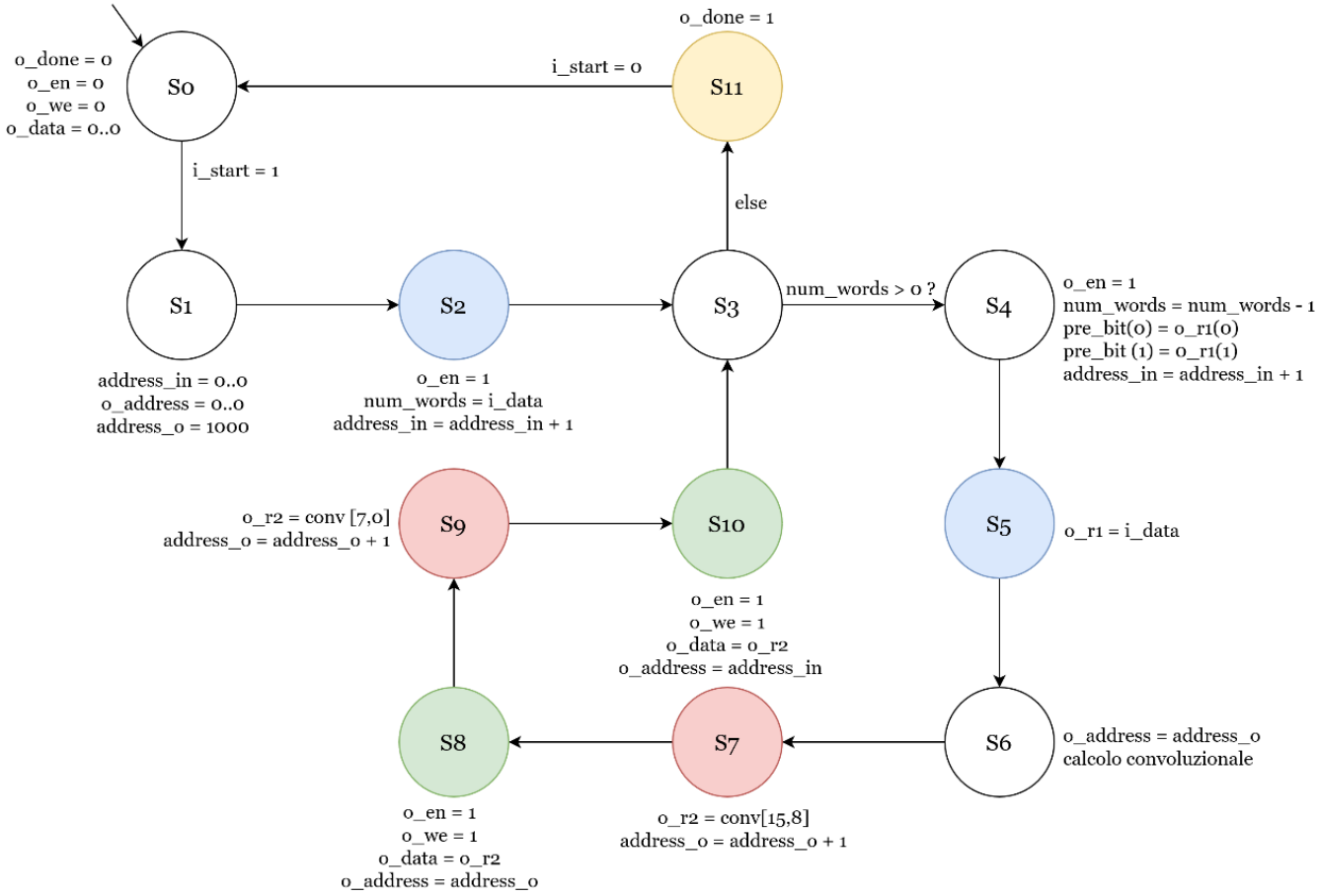


Figura 5: Macchina a stati

4.3 Diagrammi dei moduli

Per semplificarne la lettura il diagramma è stato diviso in diagramma del modulo convoluzionale, in Figura 6, e in quello del modulo per il calcolo degli indirizzi, in Figura 7. Tutti i multiplexer e i registri vengono controllati unicamente dallo stato corrente. Sono omessi per comodità di lettura il segnale di clock e il segnale di reset, per il quale tutti i segnali vengono resettati a 0.

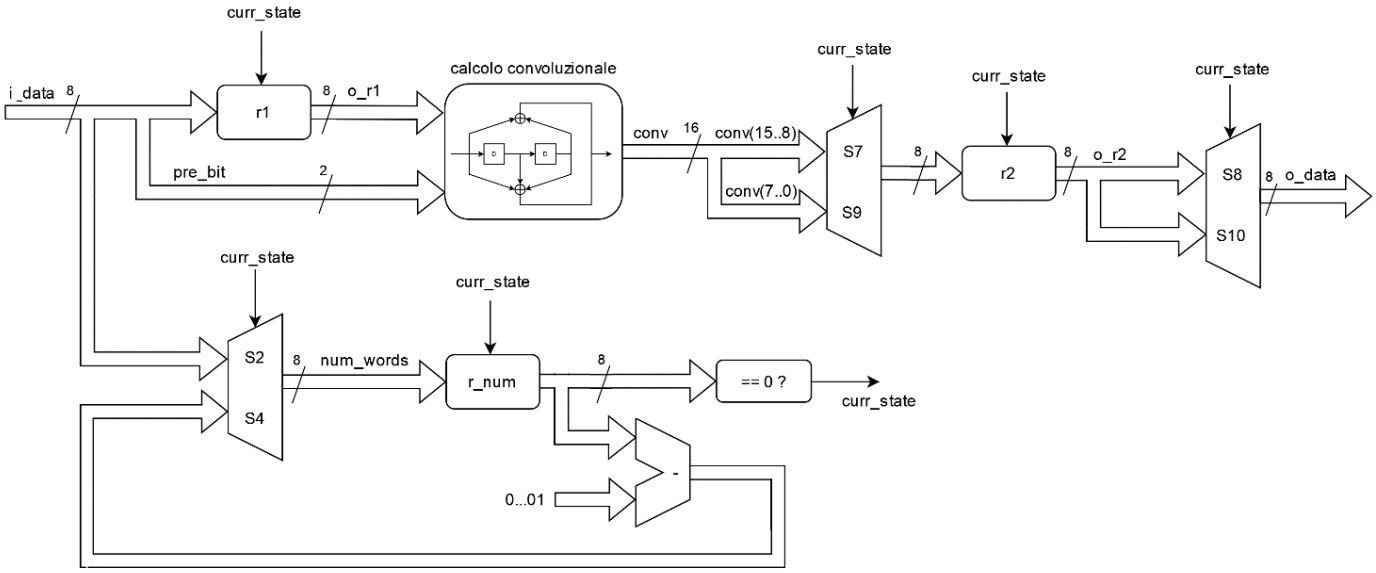


Figura 6: Diagramma modulo convoluzionale

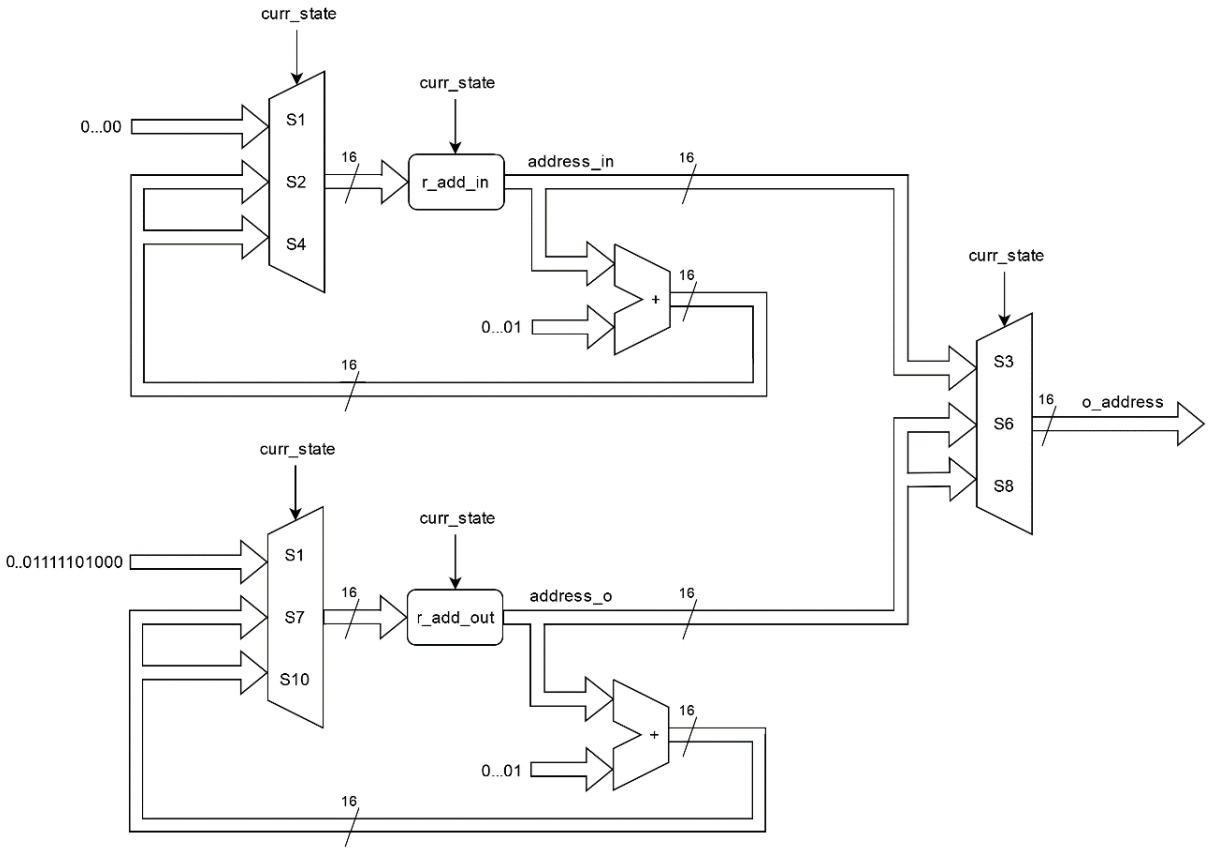


Figura 7: Diagramma modulo indirizzi

5. Risultati sperimentali

5.1 Report di sintesi

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	137	0	134600	0.10
LUT as Logic	137	0	134600	0.10
LUT as Memory	0	0	46200	0.00
Slice Registers	94	0	269200	0.03
Register as FF	94	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

Ref Name	Used	Funct Category
FDCE	86	Flop & Latch
LUT5	36	LUT
LUT6	33	LUT
LUT	30	LUT
OBUF	27	IO
LUT4	27	LUT
LUT3	11	LUT
IBUF	11	IO
LUT2	9	LUT
FDPE	8	Flop & Latch
CARRY4	8	CarryLogic
BUFG	1	Clock

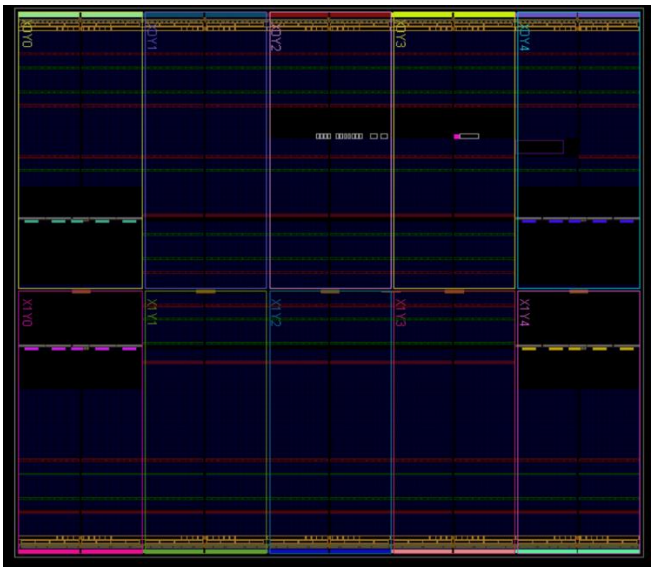


Figura 8: Sintesi del modulo

5.2 Test Bench

- *tb_base*: il primo test bench verifica il corretto funzionamento con un'unica codifica. Il diagramma dei segnali per questo test è mostrato nella Figura 9.
- *tb_multiple*: verifica il corretto funzionamento con più flussi di codifica consecutivi.
- *tb_min:* verifica il corretto funzionamento con sequenza di lunghezza nulla.
- *tb_max*: verifica il corretto funzionamento con sequenza di lunghezza massima, l'indirizzo 0 contiene "11111111" e viene quindi richiesta la codifica di 255 parole.

- *tb_reset*: verifica il corretto funzionamento nel caso in cui si ricevi un segnale di reset asincrono durante la codifica.
- *tb_reset_multiple*: verifica il corretto funzionamento con segnale di reset in più flussi di codifica. Il diagramma dei segnali per questo test è mostrato nella Figura 10.

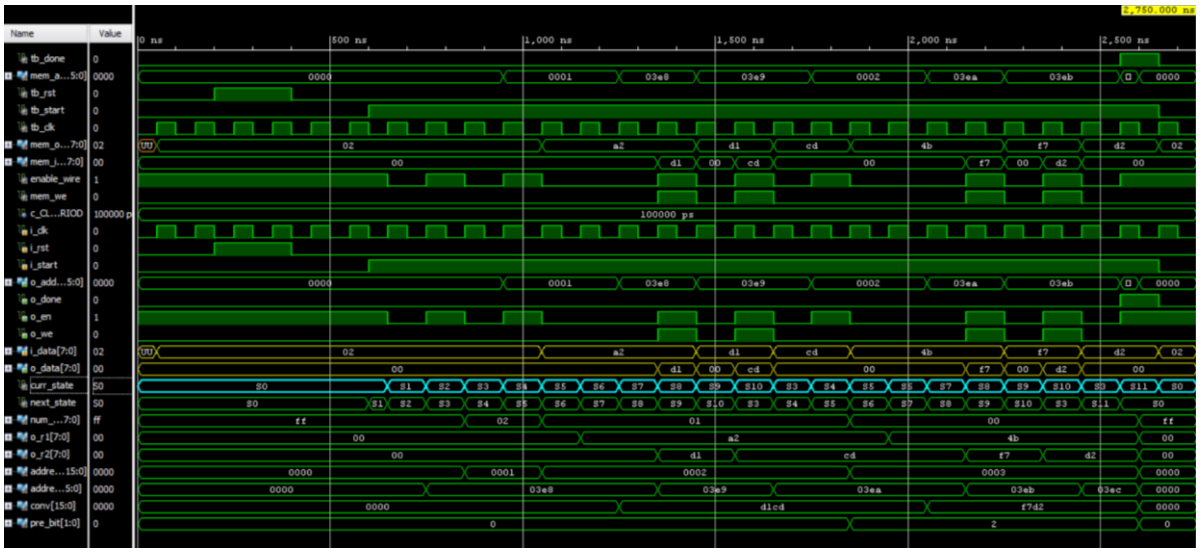


Figura 9: Diagramma dei segnali per *tb_base*

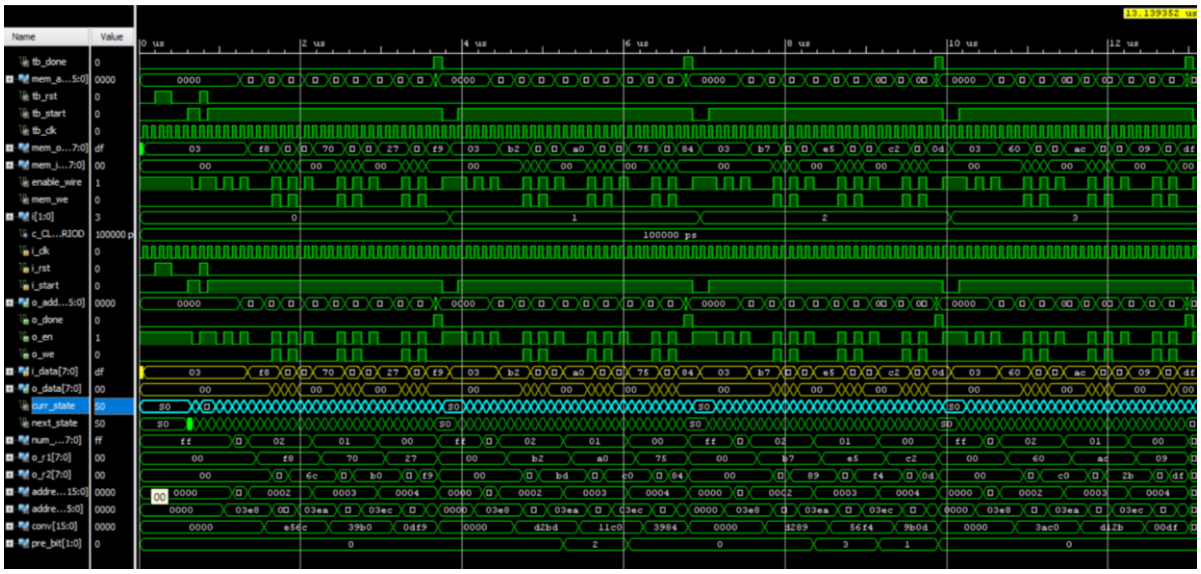


Figura 10: Diagramma dei segnali per *tb_reset_multiple*

6. Conclusioni

Il modulo implementato rispetta le specifiche richieste e si comporta correttamente per tutti i casi di test presi in analisi, sia per la codifica standard che per i casi limite considerati.