PROVA FINALE (PROGETTO DI RETI LOGICHE)

*Prof. Fabio Salice (Anno 2021/2022)*

Progetto svolto in gruppo dai seguenti studenti:

* Matteo Fiorentino (Codice Persona 10686260 – Matricola 937077)
* Luca Longinotti (Codice Persona 10707342 – Matricola 937218)

*INDICE*

[1. INTRODUZIONE 1](#_Toc112866098)

[**1.1.** **Scopo del progetto e specifiche generali** 1](#_Toc112866099)

[**1.2.** **Interfaccia del componente** 1](#_Toc112866100)

[**1.1.** **Dati e descrizione della memoria** 2](#_Toc112866101)

[2. DESIGN 2](#_Toc112866102)

[**2.1.** **Stati della macchina** 2](#_Toc112866103)

[**2.1.1.** **START** 2](#_Toc112866104)

[**2.1.2.** **READ\_START** 2](#_Toc112866105)

[**2.1.3.** **WAIT\_R\_START\_1** 2](#_Toc112866106)

[**2.1.4.** **WAIT\_R\_START\_2** 3](#_Toc112866107)

[**2.1.5.** **W1** 3](#_Toc112866108)

[**2.1.6.** **W2** 3](#_Toc112866109)

[**2.1.7.** **PK** 3](#_Toc112866110)

[**2.1.8.** **WORD\_CONSTRUCTION** 3](#_Toc112866111)

[**2.1.9.** **W3** 3](#_Toc112866112)

[**2.1.10.** **WRITE\_WORD** 3](#_Toc112866113)

[**2.1.11.** **WRITE\_WORD\_2** 3](#_Toc112866114)

[**2.1.12.** **WAIT\_W\_WORD** 3](#_Toc112866115)

[**2.1.13.** **DONE** 3](#_Toc112866116)

[**2.2.** **Scelte progettuali** 4](#_Toc112866117)

[3. RISULTATI DEI TEST 5](#_Toc112866118)

[4. CONCLUSIONI 6](#_Toc112866119)

[**4.1.** **Risultati della sintesi** 6](#_Toc112866120)

[**4.2.** **Ottimizzazioni** 7](#_Toc112866121)

# **INTRODUZIONE**

## **Scopo del progetto e specifiche generali**

Lo scopo del progetto è quello di implementare un modulo HW, descritto in VHDL, che, ricevuta in ingresso una sequenza continua di W parole, ognuna di 8 bit, restituisce in uscita una sequenza continua di Z parole, ognuna da 8 bit.

In primis viene generato un flusso continuo U da 1 bit, di lunghezza 8\*W, in seguito alla serializzazione di ognuna delle parole in ingresso.

Successivamente, su questo flusso, viene applicato il codice convoluzionale ½ (ogni bit viene codificato con 2 bit); tale operazione genera un flusso continuo Y, di lunghezza 8\*W\*2, in uscita.

La sequenza d’uscita finale Z è la parallelizzazione, su 8 bit, del flusso continuo Y.

## **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;

In particolare:

* il nome del modulo deve essere project\_reti\_logiche
* i\_clk è il segnale di CLOCK in ingresso generato dal TestBench;
* i\_rst è il segnale di RESET che inizializza la macchina pronta per ricevere il primo

segnale di START;

* i\_start è il segnale di START generato dal Test Bench;
* 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 di uscita che comunica la fine dell’elaborazione e il dato di uscita

scritto in memoria;

* o\_en è il segnale di ENABLE da dover mandare alla memoria per poter comunicare

(sia in lettura che in scrittura);

* o\_we è il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter

scriverci. Per leggere da memoria esso deve essere 0;

* o\_data è il segnale (vettore) di uscita dal componente verso la memoria.

## **Dati e descrizione della memoria**

I dati, ciascuno di dimensione 8 bit, sono memorizzati in una memoria con indirizzamento al byte:

* L’indirizzo 0 è usato per memorizzare la quantità di parole W da codificare;
* L’indirizzo 1 è usato per memorizzare il primo byte della sequenza W;
* A partire dall’indirizzo 1000 è memorizzato lo stream di uscita Z;
* La dimensione massima della sequenza di ingresso è 255 byte.

# **DESIGN**

Quando il segnale i\_start in ingresso viene portato a 1, il modulo sviluppato inizia l’elaborazione spostandosi dallo stato START al primo stato di computazione (READ\_START).

Una volta terminata la computazione, dopo aver scritto il risultato in memoria, il segnale o\_done viene alzato a 1; tale segnale rimane alto fino a quando i\_start non è stato riportato a 0.

Solo a questo punto il modulo torna nello stato IDLE in attesa che i\_start torni alto.

Il modulo, inoltre, è progettato in modo che venga effettuato sempre un reset prima della prima codifica; per questo scopo è presente il segnale i\_rst; tuttavia, una seconda elaborazione non dovrà attendere il reset del modulo ma solo la terminazione dell’elaborazione.

## **Stati della macchina**

La macchina scelta è composta da 13 stati, ciascuno di essi presentato qui in breve.

### **START**

Stato iniziale del nostro modulo: ogni ciclo di clock controlla se il segnale *i\_start* è stato portato a 1, se si il modulo passa allo stato **READ\_START**, se invece *i\_start* rimane uguale a 0 il modulo rimane nello stato **START**.

### **READ\_START**

In questo stato viene settato l’indirizzo a cui andare a leggere in memoria per ottenere i dati necessari alla computazione che sono il numero di parole da leggere e la prima parola; ovviamente non essendo possibile leggere due dati diversi contemporaneamente questo stato verrà attraversato inizialmente 2 volte (una per dato); successivamente nella computazione, lo stato verrà attraversato una volta per ogni parola da leggere. La scelta dell’indirizzo di lettura viene effettuata attraverso i due segnali *got\_n\_words* e *got\_input\_buffer*.

### **WAIT\_R\_START\_1**

Questo stato controlla se il modulo ha ottenuto entrambi i dati necessari alla computazione (numero di parole da leggere e la parola corrente): se il controllo risulta positivo il modulo passa direttamente allo stato **PK** mentre in caso contrario passa allo stato **WAIT\_R\_START\_2**.

### **WAIT\_R\_START\_2**

In questo stato vengono salvati, nei rispettivi registri, i dati: numero di parole da leggere e la parola corrente passando poi rispettivamente allo stato **W1** o **W2**.

### **W1**

Stato che controlla se il numero di parole da leggere è uguale a 0: in caso affermativo il segnale *o\_done* viene portato a 1 e il modulo passa direttamente allo stato **DONE**, in caso contrario il modulo tornerà allo stato **READ\_START** per poter settare l’indirizza a cui leggere la prima parola della sequenza.

### **W2**

Semplice stato di wait dove si attende la fine delle attività di lettura aspettando la risposta della memoria.

### **PK**

Stato in cui vengono “costruiti” i due bit derivanti dal passaggio nel convolutore. Questi due bit verranno usati nella costruzione della sequenza in uscita attraverso concatenazione dei due.

### **WORD\_CONSTRUCTION**

Stato dove viene “costruita” la sequenza (16 bit per ogni parola letta), che dovrà essere poi salvata in memoria; se la costruzione è stata completata si passerà allo stato **WRITE\_WORD**, mentre in caso contrario si passerà allo stato **W3**.

### **W3**

In questo stato blocchiamo l’aggiornamento dei Flip-Flop e successivamente ritorniamo allo stato **PK**.

### **WRITE\_WORD**

Prima parte della fase di scrittura in cui viene settato l’indirizzo in cui salvare i primi 8 bit della sequenza creata in **WORD\_CONSTRUCTION.**

### **WRITE\_WORD\_2**

Seconda parte della fase di scrittura in cui viene scritta la seconda metà della sequenza creata in **WORD\_CONSTRUCTION.**

### **WAIT\_W\_WORD**

Stato in cui si controlla se sono state lette ed elaborate tutte le parole richieste: in caso affermativo si passerà allo stato **DONE**, in caso negativo si tornerà allo stato **READ\_START** per leggere la parola successiva.

### **DONE**

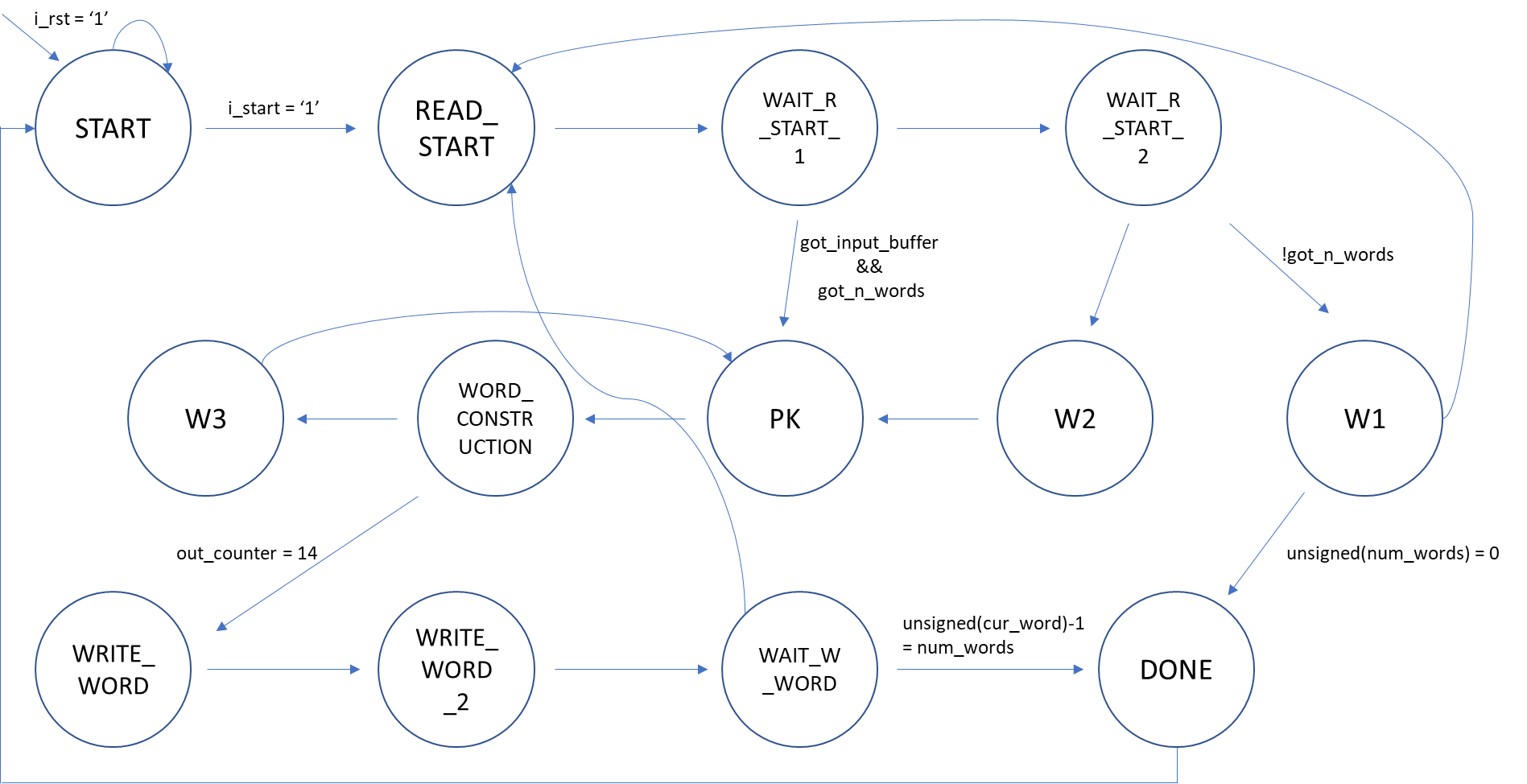
Stato finale del nostro modulo in cui il segnale *o\_done* viene settato ad 1 e che riporterà il modulo al suo stato iniziale in attesa di una nuova computazione.

## **Scelte progettuali**

Il nostro componente è formato da due processi: Il primo è quello responsabile del funzionamento della FSM: questo viene aggiornato ogni fronte di salita del ciclo di clock e permette al modulo di passare da uno stato all’altro.

Il secondo processo è quello responsabile del comportamento dei FF.

L’algoritmo costruito è stato pensato per convertire interamente, una ad una, le parole da 8 bit in sequenze da 16 bit utilizzando contatori delle celle di memoria dove leggere o scrivere i dati (utilizzando poi operazioni di somma per l’aggiornamento di essi) e flag come, ad esempio, *got\_n\_words* e *got\_input\_buffer* per confermare l’effettiva acquisizione dei dati.

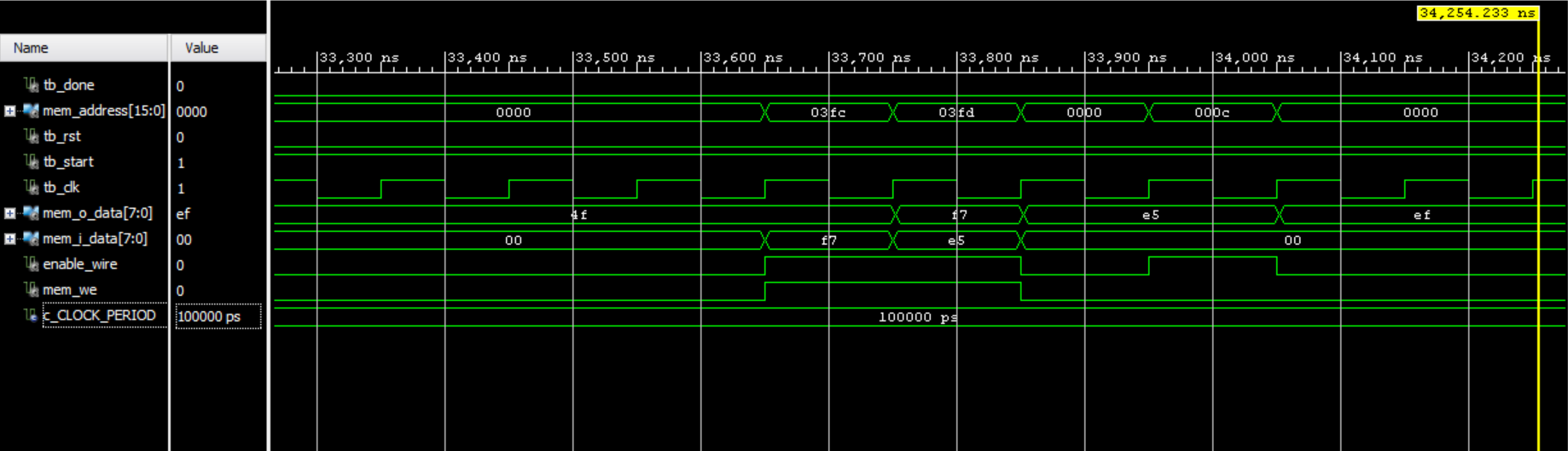
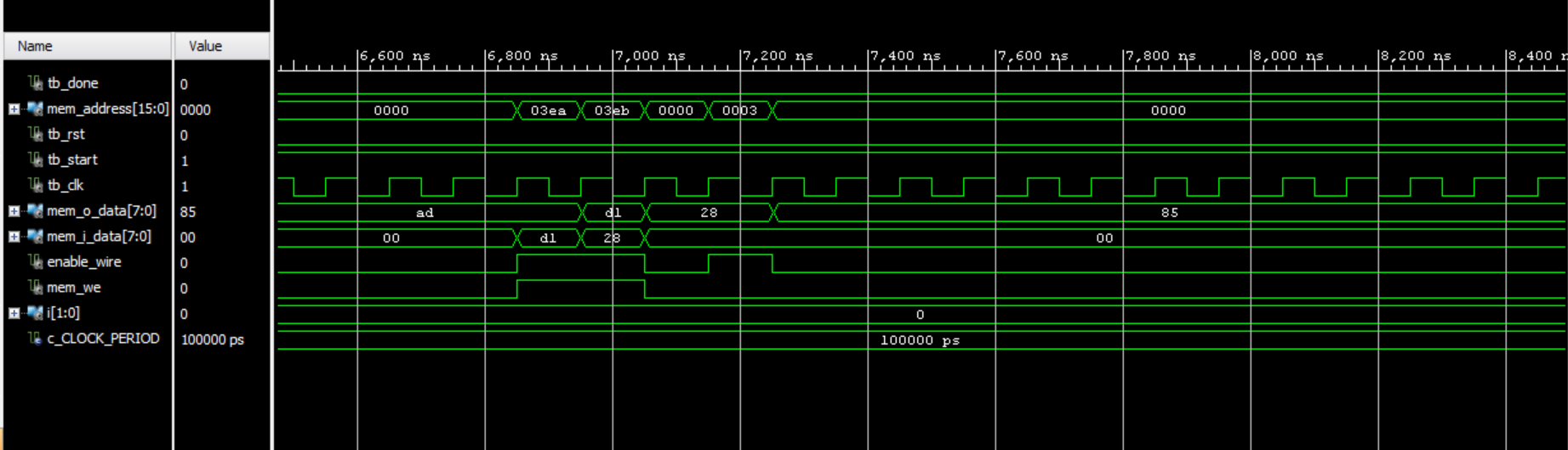
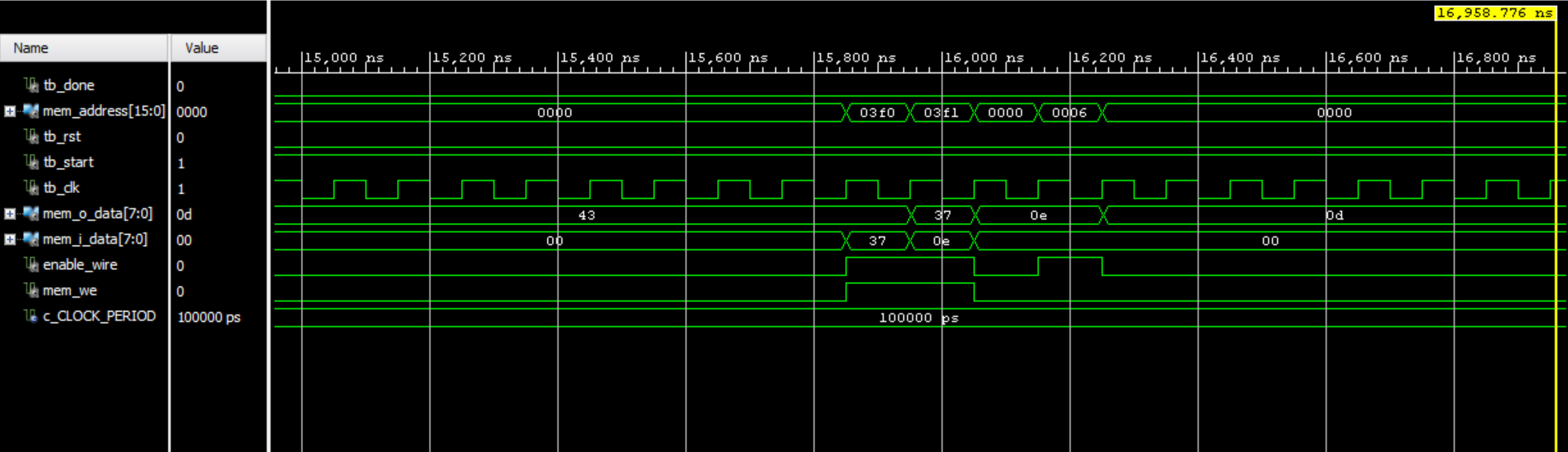
****

*MACCHINA A STATI IMPLEMENTATA*

# **RISULTATI DEI TEST**

Il nostro componente è stato testato con vari test bench che ne verificavano il corretto funzionamento in situazioni standard e nei corner case. Di seguito verranno presentati i test più significativi effettuati con i relativi andamenti dei segnali.

1. Immagine che contiene testo, schermo

   Descrizione generata automaticamente**tb\_seq\_min**: in questo test veniva controllato se, ricevendo in ingresso un numero di parole da processare uguale a zero (ovvero RAM(0) = "00000000"), non venisse scritto niente in memoria.
2. **tb\_seq\_max**: in questo test veniva controllato se, ricevendo in ingresso un numero di parole da processare uguale al massimo possibile (ovvero RAM(0) = "11111111"),il modulo funzionasse correttamente.
3. **tb\_re\_encode**: in questo test veniva controllato se, a seguito di più flussi consecutivi in ingresso venissero tutti gestiti correttamente.
4. **tb\_reset:** in questo test viene controllato se a seguito di un segnale di reset asincrono la computazione non venga compromessa, ma bensì rincominci facendo tornare la FSM allo stato iniziale e resettando tutti i registri come dovrebbe accadere.

# **CONCLUSIONI**

## **Risultati della sintesi**

Il componente sintetizzato supera correttamente tutti i test specificati nelle due simulazioni richieste: *Behavioral* e *Post-Synthesis Functional*.

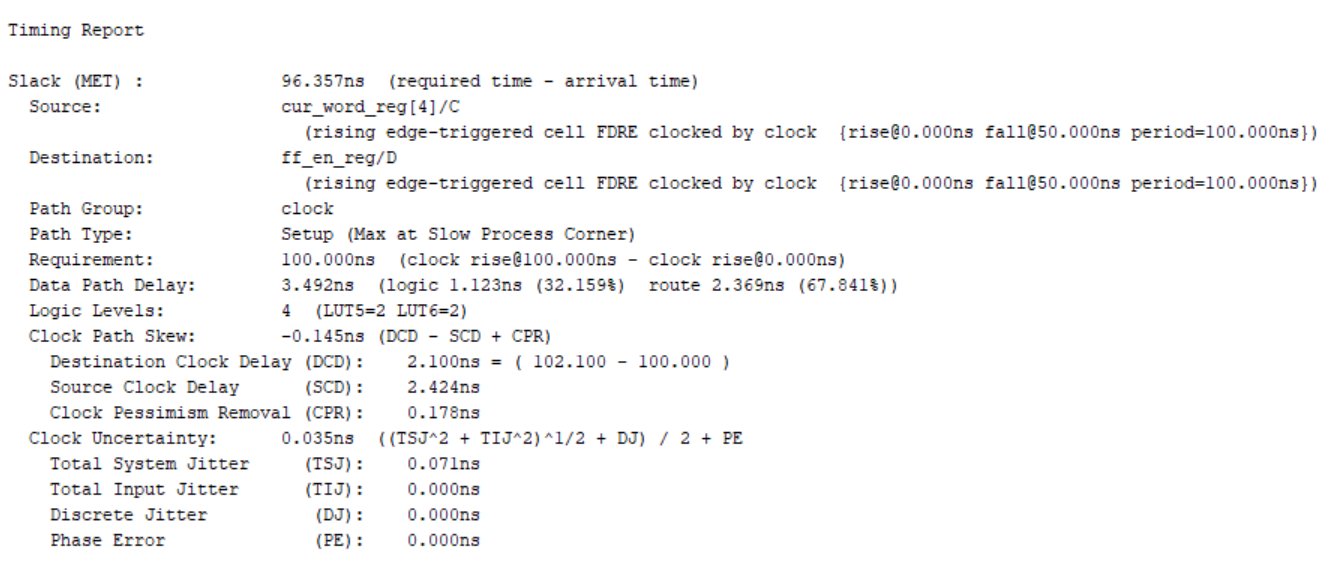
Il dispositivo utilizza 127 *Look Up Table* e 103 *Flip Flop*, senza alcun *inferred latch*.

Immagine che contiene tavolo

Descrizione generata automaticamente

*UTILITAZION REPORT*

Il componente, inoltre, rimane ampiamente dentro il tempo di clock richiesto; qui di seguito sono riportati i risultati ottenuti con un ciclo di clock di 100ns.



*TIMING REPORT*

## **Ottimizzazioni**

Le ottimizzazioni attuate al codice sono state principalmente volte alla riduzione del numero di stati, ad esempio sono stati condensati insieme gli stati relativi all’acquisizione del numero di parole e delle parole; inoltre, le restanti ottimizzazioni hanno ridotto il numero di registri iniziali, i quali durante lo sviluppo si sono poi rivelati superflui.

