Prova Finale (Progetto di Reti Logiche)

Prof. Gianluca Palermo - Anno 2021/2022

Alessandro Romito (Codice Persona 10661916 - Matricola 934080)

Giugno 2022

Indice

Introduzione

1.1 Scopo del progetto

Descrivere in VHDL e sintetizzare il componente hardware che, ricevendo in ingresso una sequenza continua di parole, serializza ed applica il codice convoluzionale 1/2 per ogni bit, parallelizzando poi in uscita il flusso continuo di bit elaborati.
Lo scopo consiste anche nell'interfacciamento con una memoria in cui sono memorizzati i dati e in cui verranno scritti i byte elaborati.

1.2 Specifiche generali

La scelta dell'utilizzo del codice convoluzionale è relativo ai sistemi trasmissivi. Lo scopo è di ottenere un trasferimento di dati affidabile.
Il modulo del progetto riceve in ingresso una sequenza continua di W parole (al massimo 255), ognuna di 8 bit e restituisce una sequenza continua di Z parole, ognuna di 8 bit. Alla sequenza di ingresso viene applicato il codice convoluzionale 1/2, ciò significa che ogni byte in ingresso corrisponde a due in uscita.
Di seguito viene illustrato il codificatore convoluzionale:

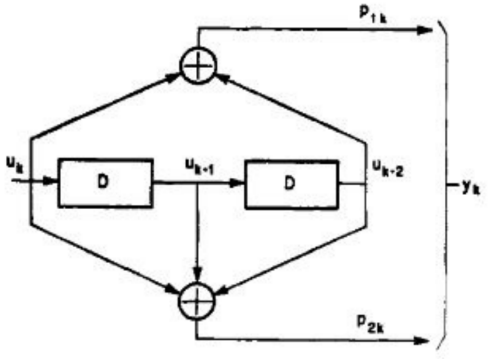

Figura 1.1: Codificatore convoluzionale con tasso di trasmissione 1/2

Per ogni bit in ingresso, in uscita si alternano p1k e p2k.
Come riportato nell'immagine sottostante, il modulo parte quando il segnale DONE è a 0 e il segnale START in ingresso viene portato a 1. Il segnale START rimane alto finchè il segnale DONE non verrà portato alto. Al termine dell'elaborazione il segnale DONE verrà alzato direttamente dal modulo notificando in questo modo la fine dell'elaborazione. Il segnale DONE rimarrà alto fino a quando il segnale START non verrà portato a 0. Seguendo di nuovo tutti i passi si potrà compiere una nuova elaborazione.

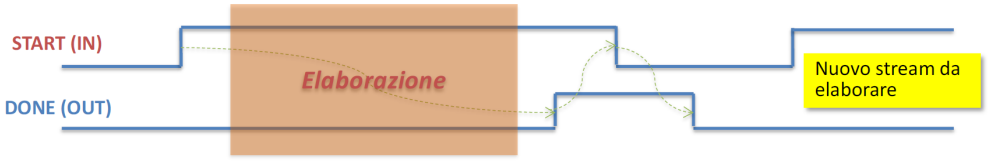

Figura 1.2: I segnali di *i_start* e *o_done* durante la computazione

Il constraint utilizzato per la gestione del clock nell'ambiente di sviluppo Vivado è stato implementato nel modo seguente:

```
create_clock -period 100 -name clock -waveform {0 50} [get_ports i_clk]
```

1.3 Interfaccia del componente

Di seguito l'interfaccia del componente:

```
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:

- **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.

1.4 Descrizione e uso memoria

Ogni lettura da memoria o memorizzazione in memoria viene effettuata al byte, quindi muovendo 8 bit per volta. Di seguito viene specificato lo spazio associato al contenuto della memoria:

- L'indirizzo 0 contiene il byte rappresentante la lunghezza della sequenza di ingresso
- Gli indirizzi da 1 a "LUNGHEZZA_SEQUENZA" contengono i byte della sequenza da codificare
- Gli indirizzi da 1000 a (1000 + "LUNGHEZZA_SEQUENZA" - 1) contengono i byte elaborati e quindi scritti in uscita.

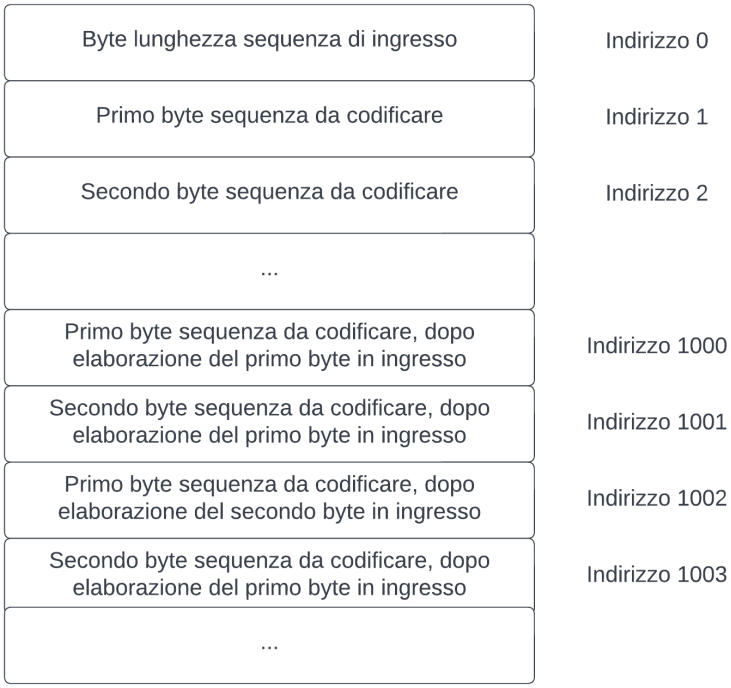

Figura 1.3: Rappresentazione uso della memoria

Architettura

L'architettura del componente realizzato prevede due entity distinte:

- Datapath: ha al suo interno il registro principale, tutta la parte che elabora il dato di ingresso, il registro d'uscita e un contatore che si occupa di far terminare l'esecuzione a sequenza terminata;
- Macchina a Stati Finiti (FSM): gestisce le fasi operative, il caricamento dei dati nei registri, la lettura e la scrittura in memoria.

2.1 Datapath

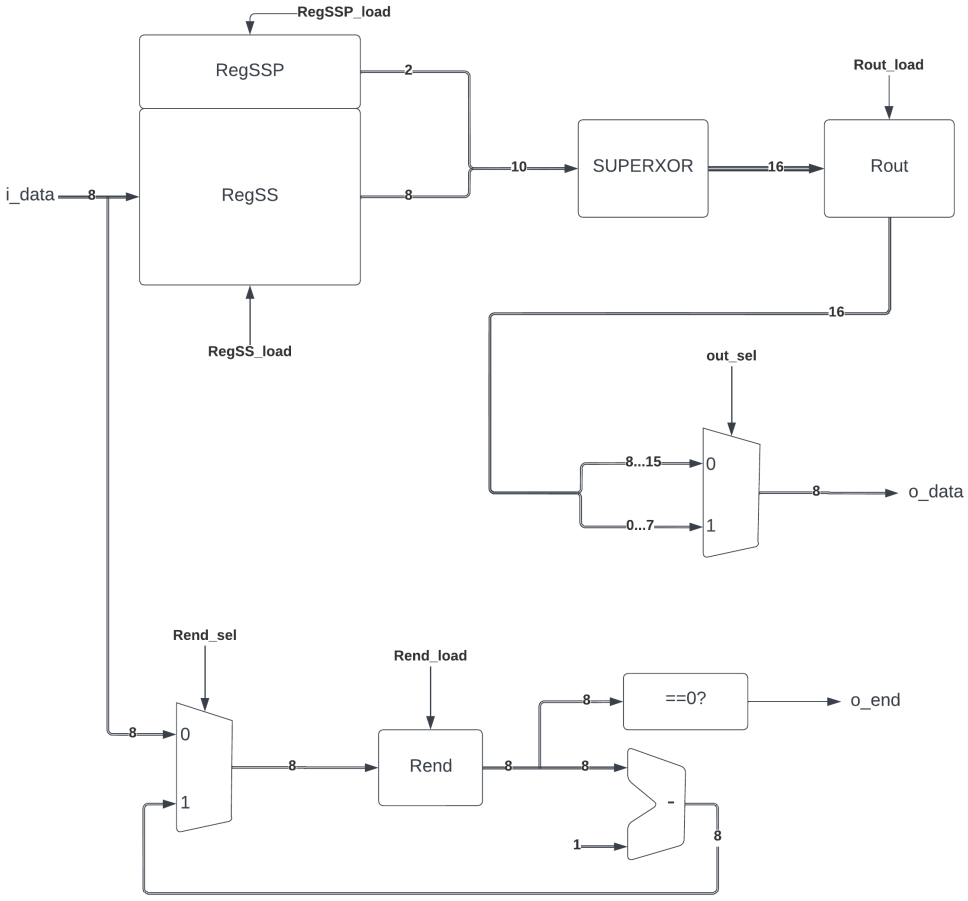

Figura 2.1: Datapath

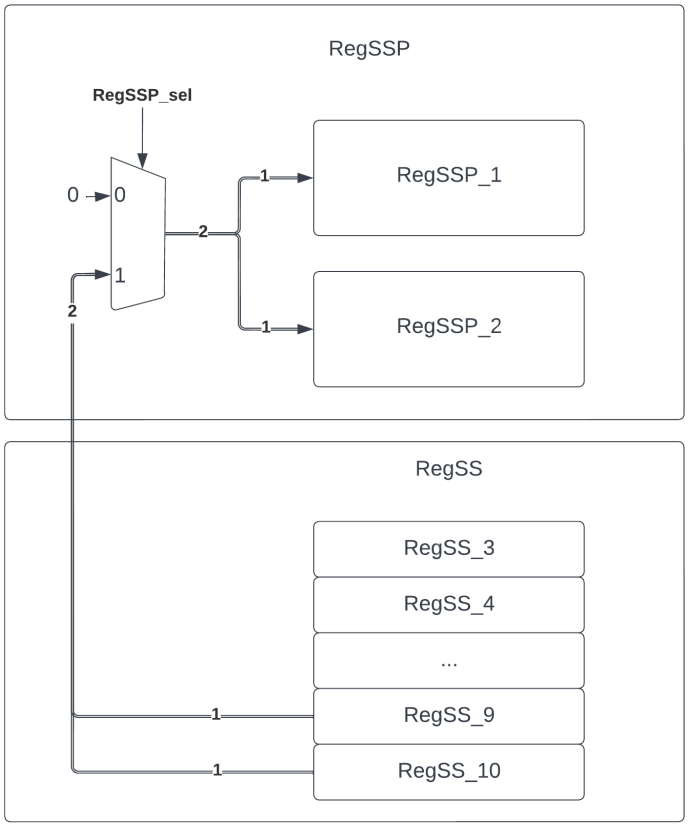

Figura 2.2: Specifica costruzione registro SSP

```
f_xor(0) <= o_regSS(2) xor o_regSS(1) xor o_regSS(0);
f_xor(1) <= o_regSS(2) xor o_regSS(0);
f_xor(2) <= o_regSS(3) xor o_regSS(2) xor o_regSS(1);
f_xor(3) <= o_regSS(3) xor o_regSS(1);
f_xor(4) <= o_regSS(4) xor o_regSS(3) xor o_regSS(2);
f_xor(5) <= o_regSS(4) xor o_regSS(2);
f_xor(6) <= o_regSS(5) xor o_regSS(4) xor o_regSS(3);
f_xor(7) <= o_regSS(5) xor o_regSS(3);
f_xor(8) <= o_regSS(6) xor o_regSS(5) xor o_regSS(4);
f_xor(9) <= o_regSS(6) xor o_regSS(4);
f_xor(10) <= o_regSS(7)xor o_regSS(6) xor o_regSS(5);
f_xor(11) <= o_regSS(7) xor o_regSS(5);
f_xor(12) <= o_regSSP(0) xor o_regSS(7) xor o_regSS(6);
f_xor(13) <= o_regSSP(0) xor o_regSS(6);
f_xor(14) <= o_regSSP(1) xor o_regSSP(0) xor o_regSS(7);
f_xor(15) <= o_regSSP(1) xor o_regSS(7);
```

Figura 2.3: Codice relativo a costruzione SUPERXOR

2.2 Macchina a stati

2.2.1 START state

Stato iniziale che si raggiunge anche quando il segnale di reset (**i_rst**) viene portato alto. Attende che il segnale **i_start** sia portato alto per procedere.

2.2.2 READ_LENGTH state

Stato in cui si inizializzano il counter ed i registri, inoltre prepara la lettura portando il segnale **o_en** alto ed associando l'indirizzo di memoria "0" a **o_address**.

2.2.3 WAIT_READ_LENGTH state

Stato in cui si memorizza il dato (8 bit) contenuto dell'indirizzo "0" nel counter.

2.2.4 FIX_COUNT state

Stato in cui avviene l'avvio dell'algoritmo di elaborazione. In questo stato, se il segnale **o_done** dovesse essere alto, avverrebbe la transizione immediata verso lo stato DONE.

2.2.5 READ_BYTE state

Stato in cui si abilita la ricezione in memoria del byte richiesto per elaborare le uscite.

2.2.6 WAIT_READ_BYTE state

Stato in cui si memorizza il dato (8 bit) nei registri, in modo da preparare i bit serializzati già elaborati.

2.2.7 WRITE_BYTE_1 state

Stato in cui si prepara la scrittura in memoria del primo byte elaborato.

2.2.8 WAIT_WRITE_BYTE_1 state

Stato in cui si scrive il primo byte elaborato in memoria.

2.2.9 WRITE_BYTE_2 state

Stato in cui si prepara la scrittura in memoria del secondo byte elaborato.

2.2.10 WAIT_WRITE_BYTE_2 state

Stato in cui si scrive il secondo byte elaborato in memoria.

2.2.11 DONE state

Stato in cui si alza il segnale **o_done** e si attende che il segnale **i_start** venga abbassato prima di tornare in START.

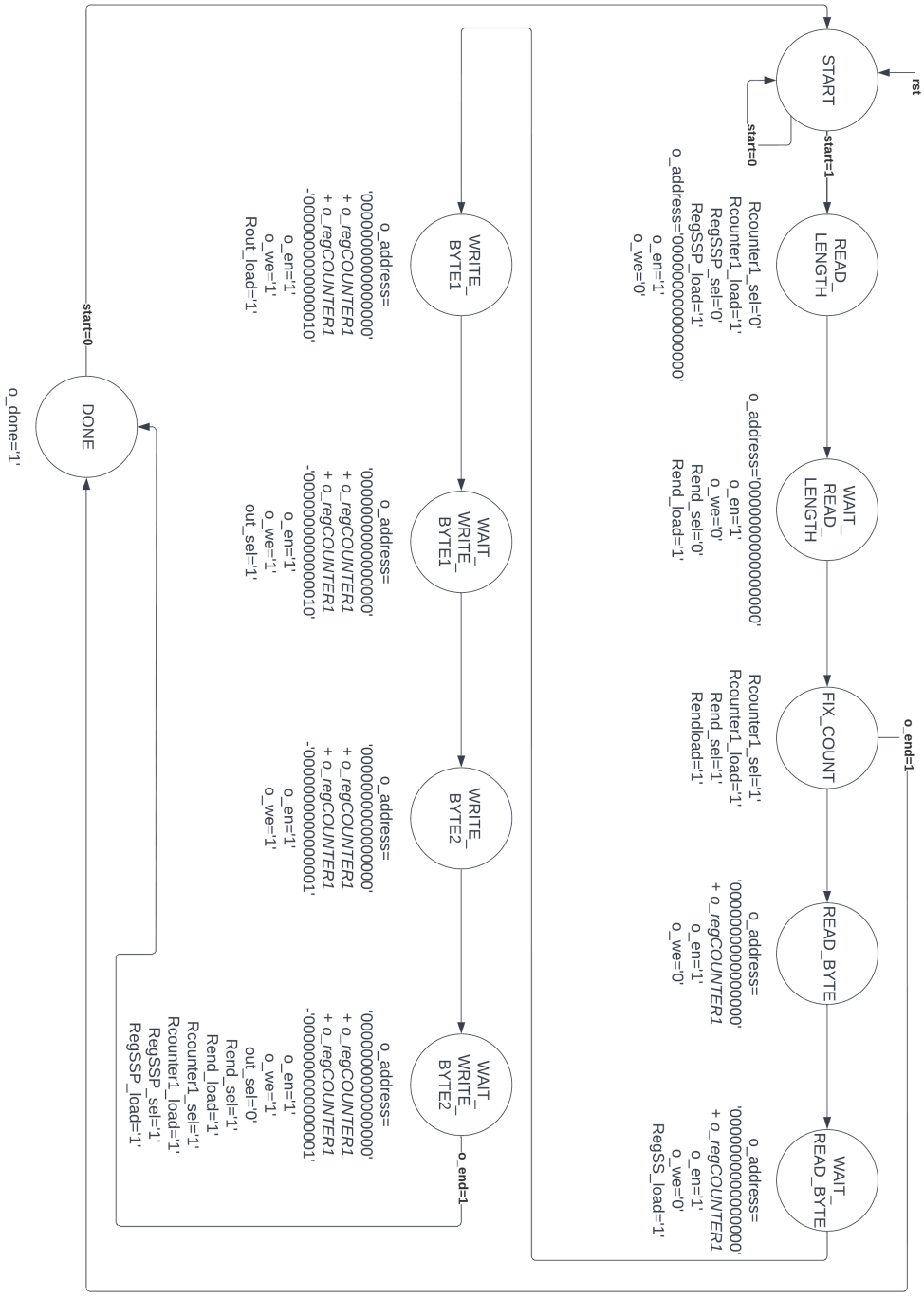

Figura 2.4: Macchina a Stati Finiti

2.3 Scelte progettuali

Come già anticipato, ho deciso di utilizzare due entity perchè personalmente credo che la modularità di un componente sia una proprietà fondamentale. Con la scelta progettuale effettuata si potrebbe intercambiare il datapath in uso con un altro che segua la stessa logica per lunghezza della sequenza, lettura e scrittura senza alcun tipo di problema. Inoltre durante la fase di debug del codice è risultato più facile e veloce mantenere i due componenti separati.

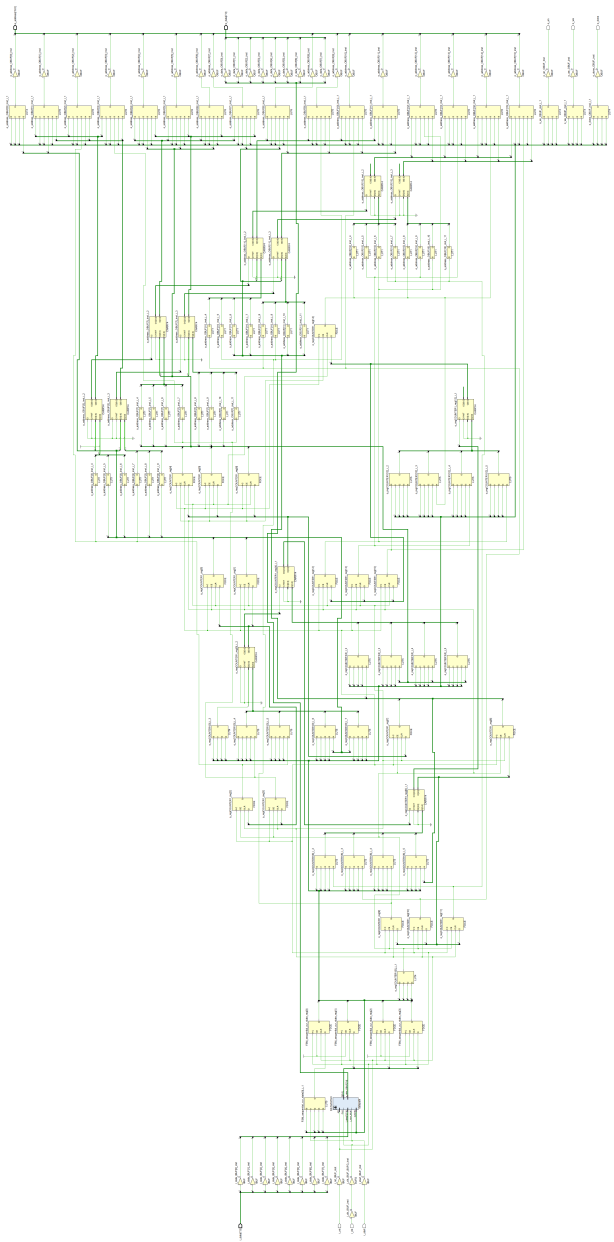

Figura 2.5: Schema di sintesi del componente

Risultati Sperimentali

3.1 Report di sintesi

Il modulo sintetizzato supera senza problemi tutti i test specificati nelle seguenti simulazioni: *Behavioral*, *Post-Synthesis Functional* e *Post-Synthesis Timing*.
FPGA utilizzata: xc7a200tfbg484-1.
Il modulo rientra pienamente nei vincoli forniti in quanto non utilizza latch ed il Worst Negative Slack è inferiore a 100ns.

3.1.1 Utilization report

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	102	0	134600	0.08
LUT as Logic	102	0	134600	0.08
LUT as Memory	0	0	46200	0.00
Slice Registers	54	0	269200	0.02
Register as Flip Flop	54	0	269200	0.02
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Figura 3.1: Utilization Report, sintesi

3.1.2 Timing report

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	97.113 ns	Worst Hold Slack (WHS):	0.142 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:	94	Total Number of Endpoints:	94	Total Number of Endpoints:	55

All user specified timing constraints are met.

Figura 3.2: Timing Report, sintesi

3.2 Test

La simulazione con il test bench fornito è andata a buon fine. Il componente ha superato anche test critici fatti ad hoc e test creati attraverso un generatore scritto in python. Il superamento implica la corretta esecuzione in *Behavioral Simulation* e *Post-Synthesis Functional Simulation*.
Di seguito riporto i test più critici effettuati per verificare la corretta esecuzione.

3.2.1 Test bench sequenza massima (255 byte in ingresso)

Viene utilizzata tutta la sequenza di ingresso disponibile in memoria seguendo la specifica (255 byte). Viene verificata l'affidabilità del modulo durante un'elaborazione a pieno carico.

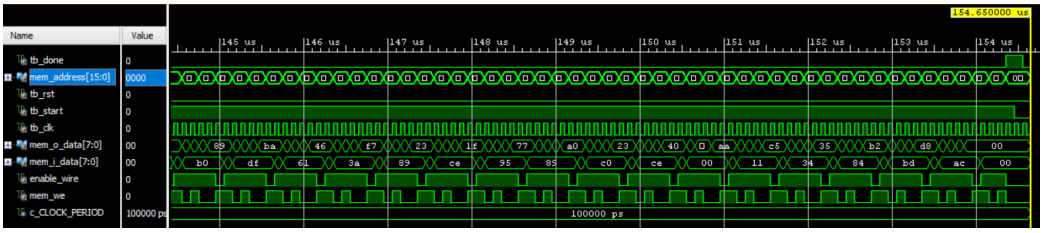

Figura 3.3: Test Bench con 255 byte in ingresso.

3.2.2 Test bench sequenza minima (0 byte in ingresso)

Tutti i byte della memoria sono inizializzati a "0". L'esecuzione si interrompe dopo aver letto la lunghezza della sequenza (0). Tutti i byte della memoria da indirizzo 1000 in poi hanno valore "0".

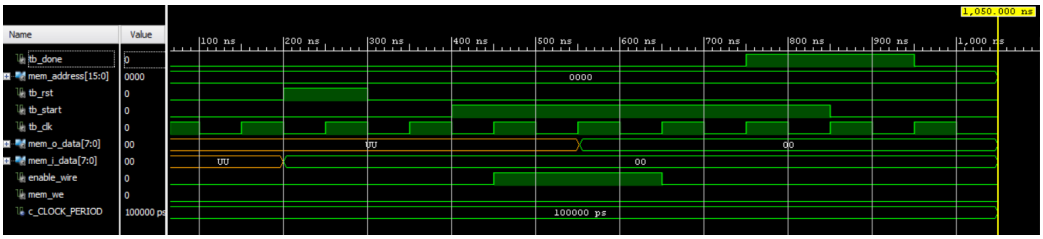

Figura 3.4: Test Bench con 0 byte in ingresso.

3.2.3 Test bench con uso di reset

Viene restato il corretto funzionamento del modulo dopo aver portato alto il segnale di controllo *i_rst*. In condizioni reali potrebbe essere portato alto principalmente in caso di malfunzionamento o riavvio del dispositivo.

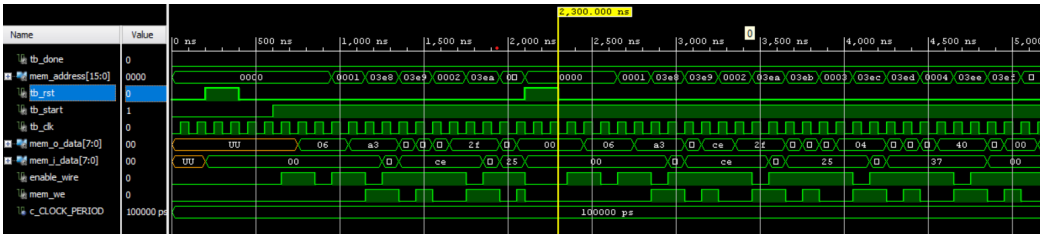

Figura 3.5: Test Bench con utilizzo del segnale di reset.

3.2.4 Test bench con diverse sequenze da elaborare

Vengono testate tre sequenze con valori diversi una dall'altra. Questa Test Bench è importante per il superamento di un punto importante della specifica fornita per la progettazione del componente, cioè la possibilità di finire un'elaborazione ed iniziarne un altra. Non solo si assicura del corretto funzionamento dei segnali *i_ start* e *o_ done*, ma anche della corretta inizializzazione dei registri di ingresso e del counter di memoria e di fine sequenza.

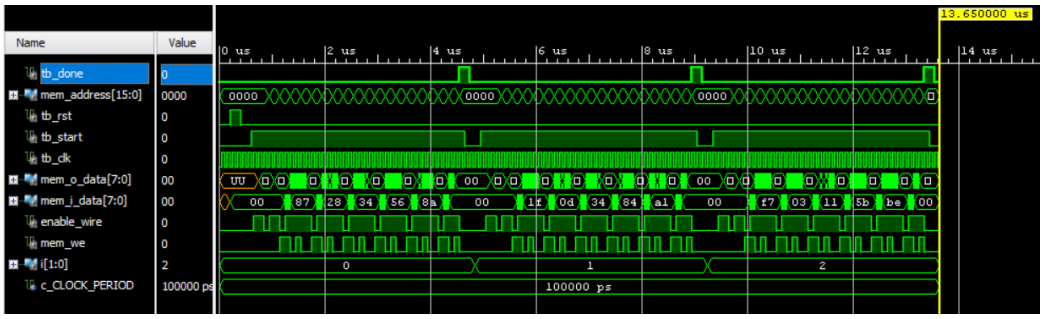

Figura 3.6: Test Bench con tre sequenze da elaborare.

Conclusioni

Il modulo è in grado di funzionare correttamente e nei vincoli forniti. Sicuramente c'è margine di miglioramento sia per il numero di componenti utilizzate, sia per il Worst Negative Slack. La parte più impegnativa è stata progettare il datapath e la macchina a stati senza avere un'interfaccia che in real time testasse il codice.
Personalmente trovo VHDL un linguaggio molto difficile da capire all'inizio, ma col passare del tempo ho trovato molto semplice fare debug e ottimizzare tempi e componenti utilizzati. Infatti eliminare l'utilizzo dei latch, una volta finito il progetto, è stato un lavoro di soli 10 minuti.
Trovo inoltre che sia essenziale per un laureando in ingegneria informatica della triennale conoscere e saper utilizzare questo tipo di linguaggio.