POLITECNICO
MILANO 1863

PROVA FINALE DI RETI LOGICHE
Docente: Palermo Gianluca – AA 2023/2024

Andrea Tonoli
Codice Persona 10764952 – Matricola 961130

INDICE

1. INTRODUZIONE

1.1 Scopo e Specifica del Progetto

Lo scopo del progetto è quello di implementare un componente hardware descritta in VHDL che legge e completa una sequenza di parole memorizzate in una memoria.

La sequenza verrà presa dalla memoria e conterrà una serie di *i_k* parole che si troveranno a intervalli di 2 byte partendo dall' indirizzo iniziale *i_add* (ovvero: *i_add*, *i_add*+2, i_*add*+4, ...).
Per completare la sequenza il componente dovrà svolgere due operazioni:

1) Per ogni parola letta, dovrà venire generato un "valore di credibilità" *C*, che verrà memorizzato nel byte successivo alla parola stessa (ad esempio, se la parola si trova all'indirizzo *add*, il valore *C* sarà memorizzato in *add*+1). Il valore di C varierà in base al valore della parola *W* che lo precede:
    - se *W* != 0, il valore di credibilità verrà inizializzato a 31;
    - se *W* = 0, il valore di credibilità sarà uguale all'ultimo valore di credibilità registrato meno 1 ($C_{new} = C_{old} - 1$).
2) Se una parola letta ha valore zero, dovrà essere sostituita con l'ultima parola letta con valore valido (diverso da zero) incontrata in precedenza.

ESEMPIO
Sequenza di Partenza: **128** 0 **64** 0 **0** 0 **11** 0 **0** 0 **0** 0
Sequenza Finale: **128** 31 **64** 31 **64** 30 **11** 31 **11** 30 **11** 29

Caso Particolare:
Se il primo dato della sequenza è pari a zero, il suo valore rimane tale e il valore di credibilità deve essere posto a zero. Lo stesso succede fino al raggiungimento del primo dato della sequenza con valore diverso da zero.

ESEMPIO
Sequenza di Partenza: **0** 0 **0** 0 **0** 0 **11** 0 **55** 0 **0** 0
Sequenza Finale: **0** 0 **0** 0 **0** 0 **11** 31 **55** 31 **55** 30

1.2 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_add    : in std_logic_vector(15 downto 0);
    i_k      : in std_logic_vector(9 downto 0);

    o_done   : out std_logic;
    o_mem_addr : out std_logic_vector(15 downto 0);
    i_mem_data : in std_logic_vector(7 downto 0);
    o_mem_data : out std_logic_vector(7 downto 0);
    o_mem_we  : out std_logic;
    o_mem_en  : out std_logic
  );
end project_reti_logiche;
```

- Segnali di Input
  - → i_clk: segnale di CLOCK generato dal TestBench
  - → i_rst: segnale di RESET che inizializza la macchina
  - → i_start: segnale di START che da il via all'elaborazione
  - → i_add: segnale (vettore) che indica l'indirizzo dal quale parte la sequenza da elaborare
  - → i_k: segnale (vettore) che rappresenta la lunghezza della sequenza da elaborare
  - → i_mem_data: segnale (vettore) che arriva dalla memoria e che contiene il dato in seguito ad una richiesta di lettura
- Segnali di Output
  - → o_done: segnale di DONE che comunica la fine dell'elaborazione
  - → o_mem_addr: segnale (vettore) che manda l'indirizzo di memoria
  - → o_mem_data: segnale (vettore) che va verso la memoria e contiene il dato che verrà successivamente scritto
  - → o_mem_we: segnale di WRITE ENABLE da mandare alla memoria (=1) per poterci scrivere (=0) per poterci leggere
  - → o_mem_en: segnale di ENABLE da dover mandare alla memoria per poter comunicare

2. ARCHITETTURA

2.1 Segnali di Controllo

Il componente hardware opera con molteplici segnali, tra cui tre principali di controllo: *RESET (i_rst)*, *START (i_start)* e *DONE (o_done)*.
Tutti i segnali del modulo, inclusi quelli di ingresso e uscita, sono interpretati in modo sincrono sul fronte di salita del *CLOCK (i_clk)*, con l'unica eccezione del segnale asincrono di RESET. L'attivazione di quest'ultimo provoca l'immediata reinizializzazione del modulo, dunque, tutti i registri interni vengono azzerati e l'uscita *DONE* viene forzata a 0, indipendentemente dallo stato del *CLOCK*.

Il flusso operativo inizia con un *RESET* obbligatorio, seguito da un ciclo *START-elaborazione-DONE*, che può ripetersi senza necessità di ulteriori *RESET*. Il segnale *START* rimane alto durante l'intera fase di elaborazione e fino all'attivazione del segnale *DONE*, il quale indica il completamento dell'elaborazione. *DONE* resta alto fino a quando *START* non viene disattivato, garantendo una chiara transizione tra le fasi e assicurando l'esecuzione corretta di possibili elaborazioni multiple.

2.2 Segnali Interni

I segnali interni utilizzati per lo svolgimento del progetto sono:

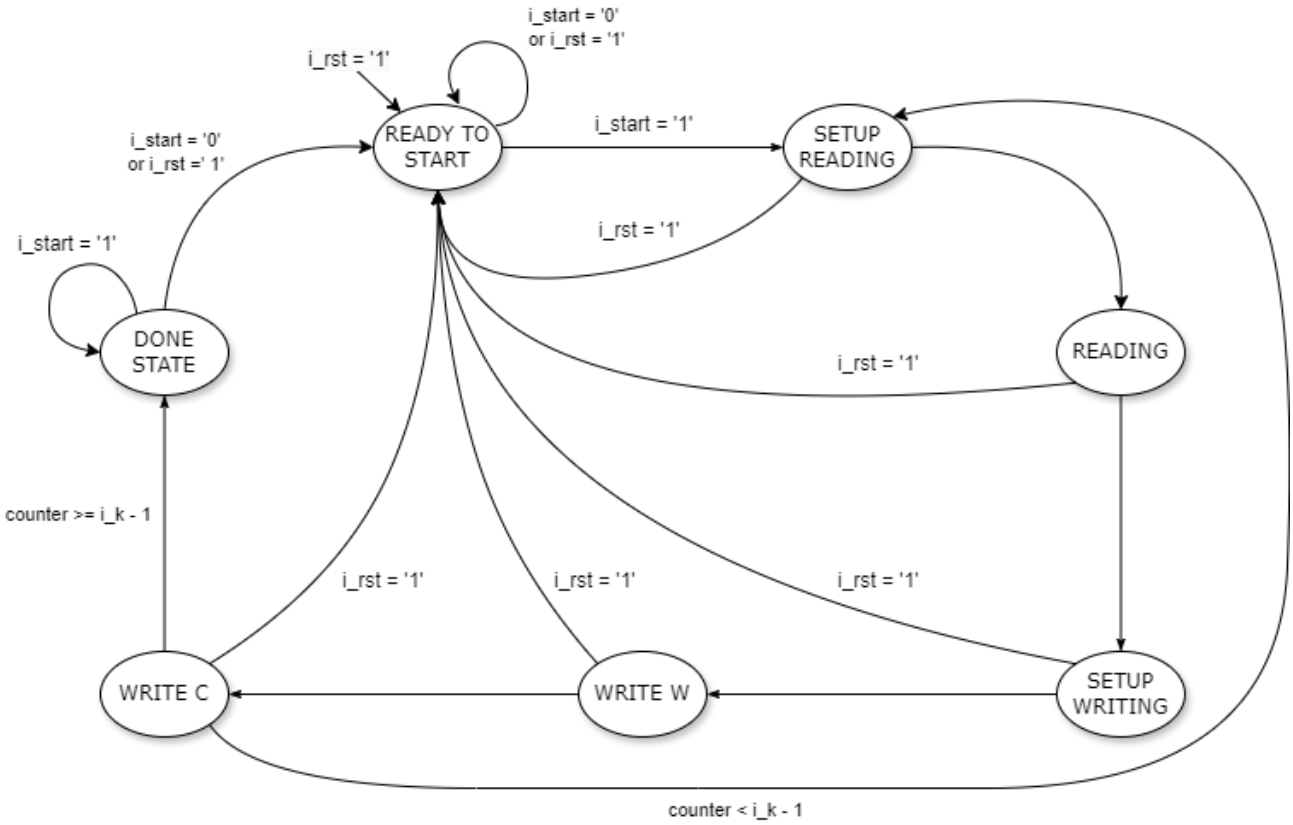
NOME	TIPO	DESCRIZIONE
counter	unsigned(9 downto 0)	tiene il conto di quante parole sono già state lette
w	std_logic_vector(7 downto 0)	contiene il valore della parola nella posizione attuale
last_valid_w	std_logic_vector(7 downto 0)	ultima parola letta valida (diversa da 0)
c	unsigned(7 downto 0)	valore di credibilità associato all’ultima parola letta
curr_state	States	stato attuale della fsm
next_state	States	stato futuro della fsm

2.3 I Processi

Per garantire una gestione efficace della FSM, il progetto è stato organizzato in tre processi distinti. Questa suddivisione permette una chiara separazione delle responsabilità tra i vari compiti, semplificando il controllo del sistema e rendendo il codice più comprensibile.

- Processo ***fsm_next_state***:
  questo processo è responsabile esclusivamente della logica di transizione degli stati. In questo modo, le decisioni relative ai cambiamenti di stato risultano separate dalle operazioni svolte, rendendo più semplice l'interpretazione del flusso di controllo.
- Processo ***fsm_curr_state***:
  questo processo gestisce l'aggiornamento dello stato corrente della FSM. Se il segnale di reset è attivo, la FSM viene riportata nello stato iniziale. In caso contrario, ad ogni fronte di salita del clock, lo stato viene aggiornato con il valore di next_state.
- Processo ***fsm_operations***:
  questo processo gestisce tutte le operazioni pratiche associate a ciascuno stato, come la gestione degli accessi alla memoria e l'aggiornamento dei dati. Separando queste azioni dalla logica di transizione degli stati, si assicura una gestione chiara e organizzata delle operazioni, evitando interferenze con il controllo degli stati.

2.4 Macchina a Stati Finiti (FSM)

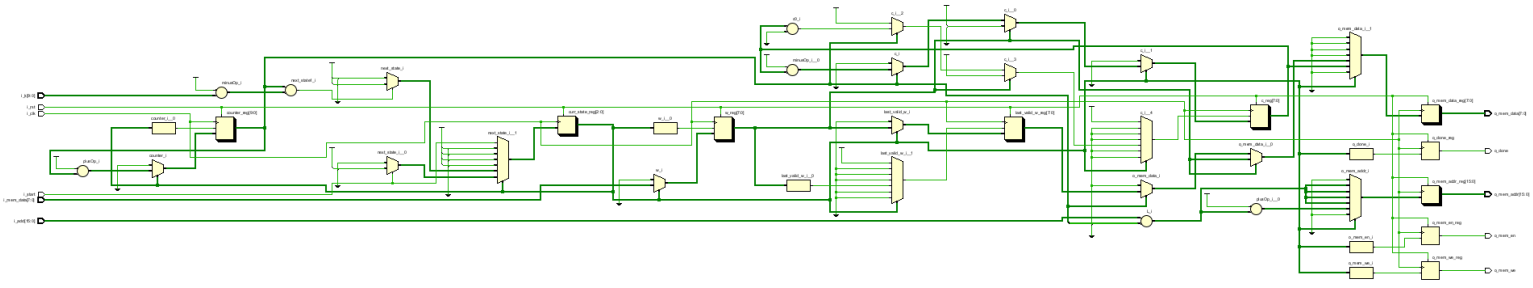


Ogni stato ha un ruolo specifico nel ciclo di lettura, elaborazione e scrittura in memoria. Di seguito viene fornita una descrizione dettagliata di ogni stato e delle sue operazioni principali.

1. READY_TO_START
   Stato iniziale della FSM, per accedervi la prima vota è obbligatoria l'attivazione del segnale di reset. Qui la FSM resetta tutti i segnali interni e attende l'attivazione del segnale di start, una volta attivo, il sistema si prepara per l'elaborazione della sequenza. Questo è anche lo stato a cui la FSM ritorna dopo ogni reset.
2. SETUP_READING
   Prepara la lettura dalla memoria abilitando il segnale *o_mem_en* mantenendo comunque basso il segnale *o_mem_we* e calcolando l'indirizzo di memoria in base all'indirizzo *i_add* e al contatore di parole lette. L'indirizzo viene poi assegnato a *o_mem_addr* per essere utilizzato in seguito.
3. READING
   Questo stato ha il solo scopo di far attendere alla FSM un ciclo di clock per garantire che il dato letto dalla memoria all'indirizzo *o_mem_addr* venga trasferito correttamente nel segnale *i_mem_data*, rendendolo pronto per l'elaborazione successiva.

4. SETUP_WRITING
   Prepara l'accesso in scrittura alla memoria abilitando il segnale *o_mem_we* mantenendo comunque attivo il segnale *o_mem_en*.
   la parola letta rappresentata da *i_mem_data* viene memorizzata nel segnale *w*.
5. WRITE_W
   Si occupa di scrivere la parola *w* nella memoria. Se *w* è zero e il contatore è a zero, scrive zero nella memoria, mentre, se il contatore è diverso da zero, scrive l'ultima parola valida precedentemente letta *last_valid_w*, e diminuisce il valore di credibilità (a meno che non sia già a zero). Se invece *w* è diverso da zero, il dato *w* viene scritto nella memoria e *last_valid_w* viene aggiornato con il nuovo valore. In questo caso, la credibilità *c* viene riportata a 31.
6. WRITE_C
   Si occupa di scrivere il valore di credibilità *c* nella memoria. Per fare ciò, l'indirizzo di memoria viene aggiornato per puntare alla cella successiva rispetto a quella dove è stata scritta la parola *w*. Una volta completata la scrittura del valore di *c*, il contatore counter viene incrementato per passare al dato successivo. Se il contatore ha raggiunto il limite specificato da *i_k*, la FSM si prepara a terminare l'esecuzione passando allo stato DONE_STATE, altrimenti torna allo stato SETUP_READING per iniziare una nuova iterazione.
7. DONE_STATE
   Indica la fine del ciclo di elaborazione. I segnali di controllo della memoria vengono disabilitati mentre il segnale *o_done* viene portato a '1' per segnalare il completamento dell'operazione. La FSM rimane in questo stato fino a quando *i_start* non viene riportato a '0', momento in cui ritorna allo stato iniziale READY_TO_START, pronta per elaborare una nuova sequenza.

2.5 Risultato Finale dell'Architettura



3. Risultati Sperimentali

3.1 Report Utilization

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	82	0	0	134600	0.06
LUT as Logic	82	0	0	134600	0.06
LUT as Memory	0	0	0	46200	0.00
Slice Registers	64	0	0	269200	0.02
Register as Flip Flop	64	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

Il report di utilizzo fornisce un riepilogo delle risorse utilizzate durante la sintesi del progetto. In particolare, sono stati utilizzati **64 Flip-Flop**, distribuiti come registri, **senza alcun utilizzo di latch**, il che indica che la logica sequenziale è stata implementata correttamente.

3.2 Report Timing

```
Slack (MET) :             14.881ns  (required time - arrival time)
  Source:                 counter_reg[0]/C
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@10.000ns period=20.000ns})
  Destination:            o_mem_addr_reg[12]/D
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@10.000ns period=20.000ns})
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            20.000ns  (clock rise@20.000ns - clock rise@0.000ns)
  Data Path Delay:        4.968ns  (logic 2.977ns (59.924%)  route 1.991ns (40.076%))
  Logic Levels:           7  (CARRY4=5 LUT2=1 LUT5=1)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.100ns = ( 22.100 - 20.000 )
    Source Clock Delay      (SCD):    2.424ns
    Clock Pessimism Removal (CPR):    0.178ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

Il timing report conferma che il design soddisfa i vincoli temporali della specifica, con uno **Slack positivo** di 14.881 ns su un clock di 20 ns. Questo significa che il percorso critico richiede solo 5.119 ns, garantendo un ampio margine rispetto al periodo di clock.

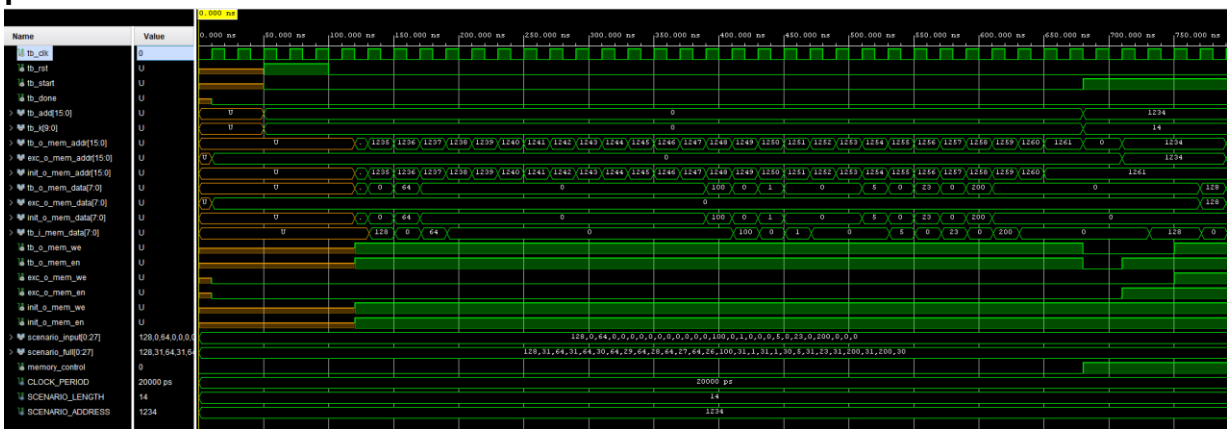
3.3 Risultati dei Test

Di seguito sono riportati i diversi casi di test a cui ho sottoposto il componente per verificarne il corretto funzionamento. Tutti i test sono stati superati con successo, sia in behavioral che in post sintesi.

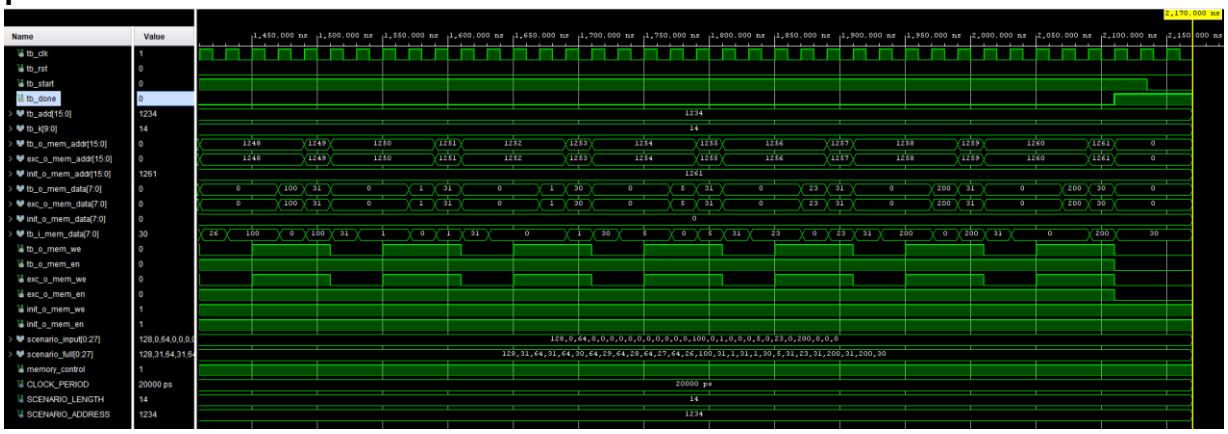
1) Caso Base

Questo è il TestBench fornito dal professore. Poiché non include casi limite, è stato utilizzato come test di base per verificare il corretto funzionamento della componente.

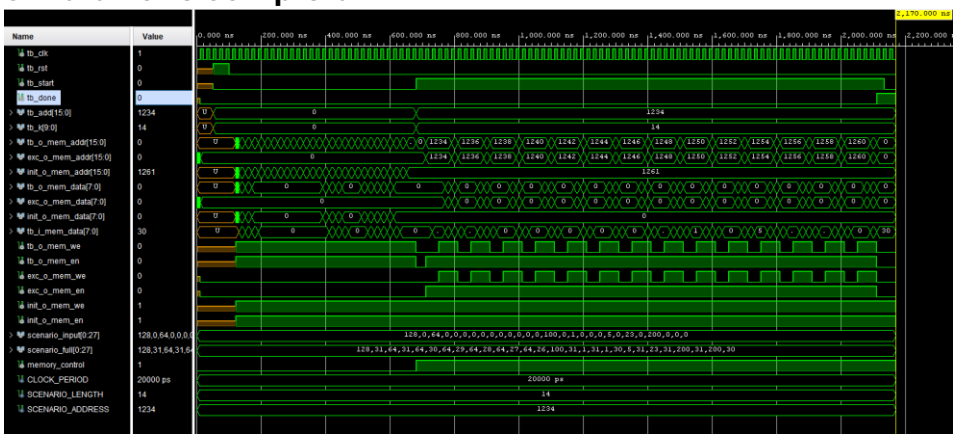
parte iniziale simulazione



parte finale simulazione



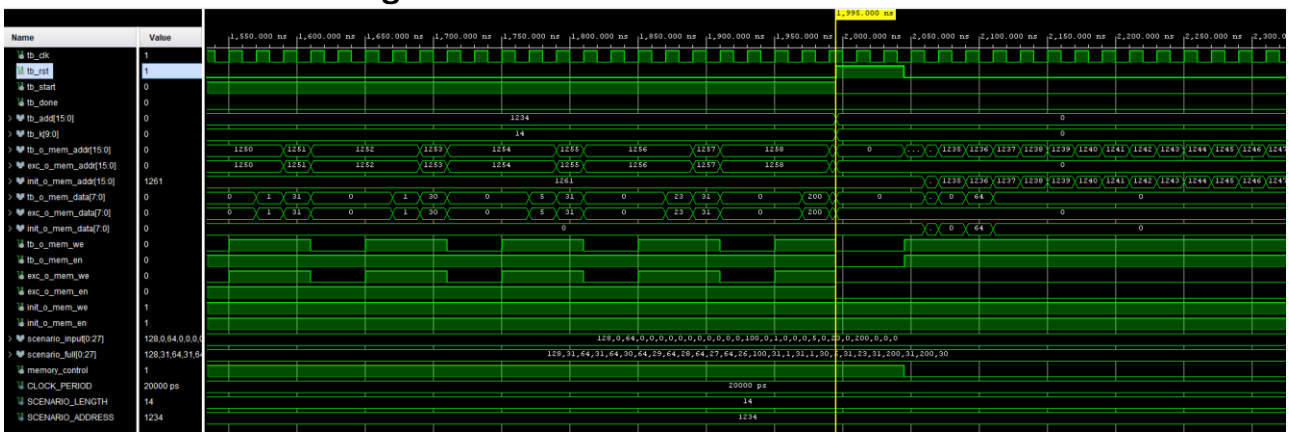
simulazione completa



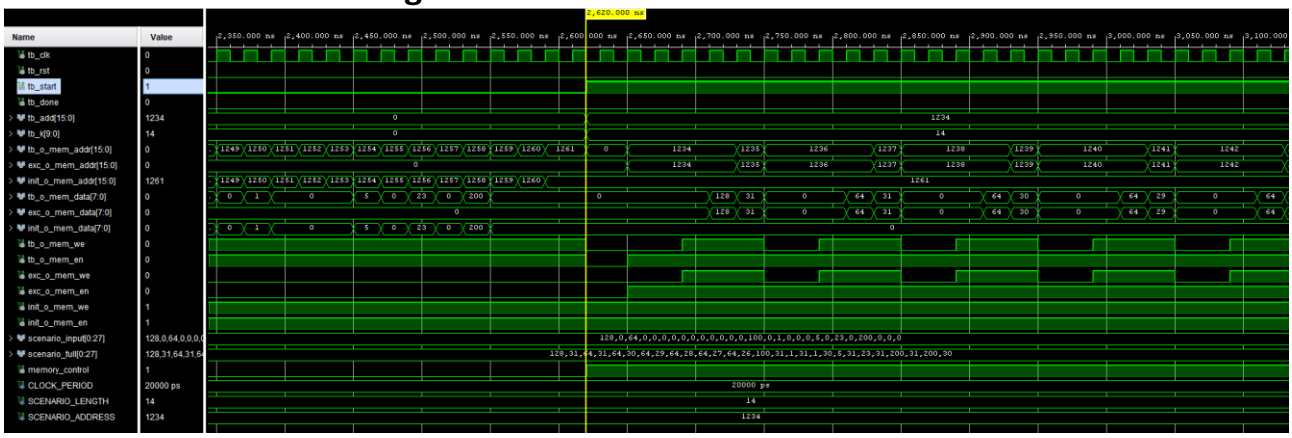
2) Reset

Questo TestBench verifica che il componente reagisca in modo adeguato durante un *reset* e che riprenda correttamente l'esecuzione in caso di un nuovo segnale di *start*.

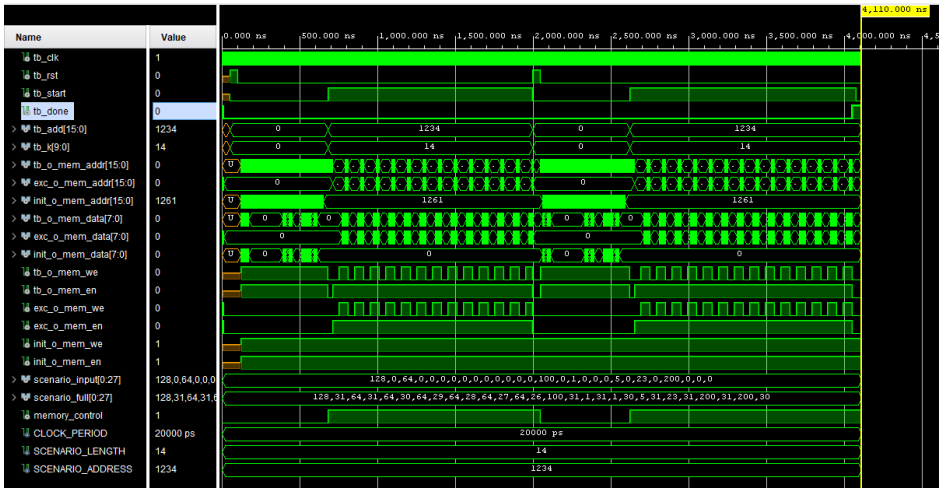
istante in cui si alza il segnale di reset



istante in cui si rialza il segnale di start



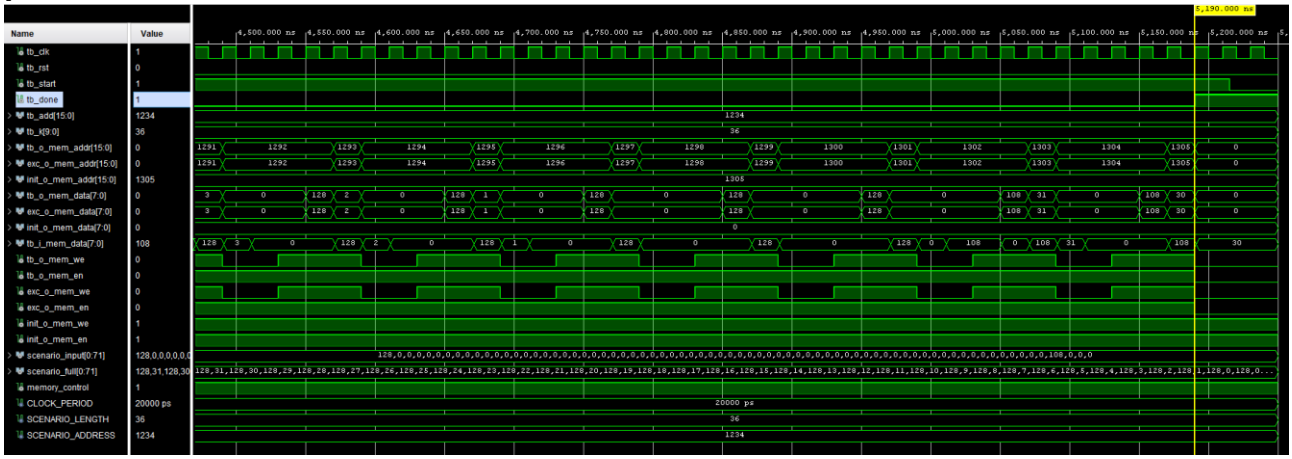
simulazione completa



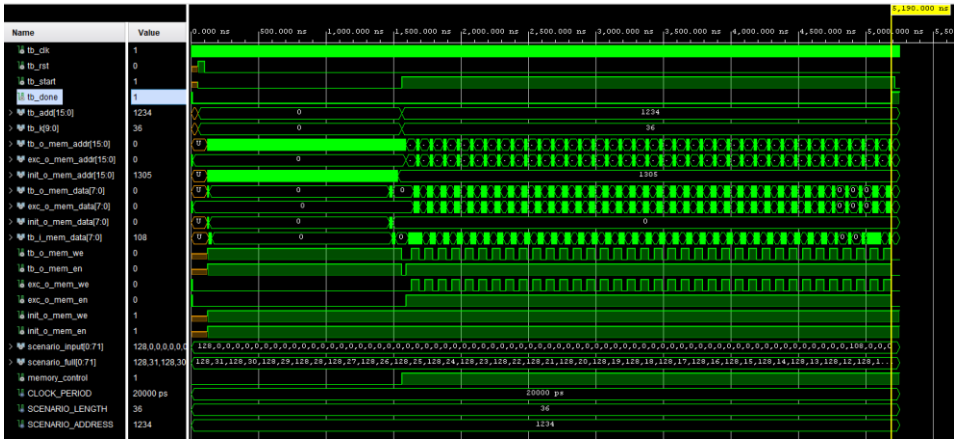
3) C=0

Con questo TestBench si vuole verificare che il valore di credibilità rispetti le specifiche, assicurando che non scenda mai sotto lo zero e che ritorni al valore massimo anche dopo aver raggiunto lo zero.

parte in cui c diventa 0



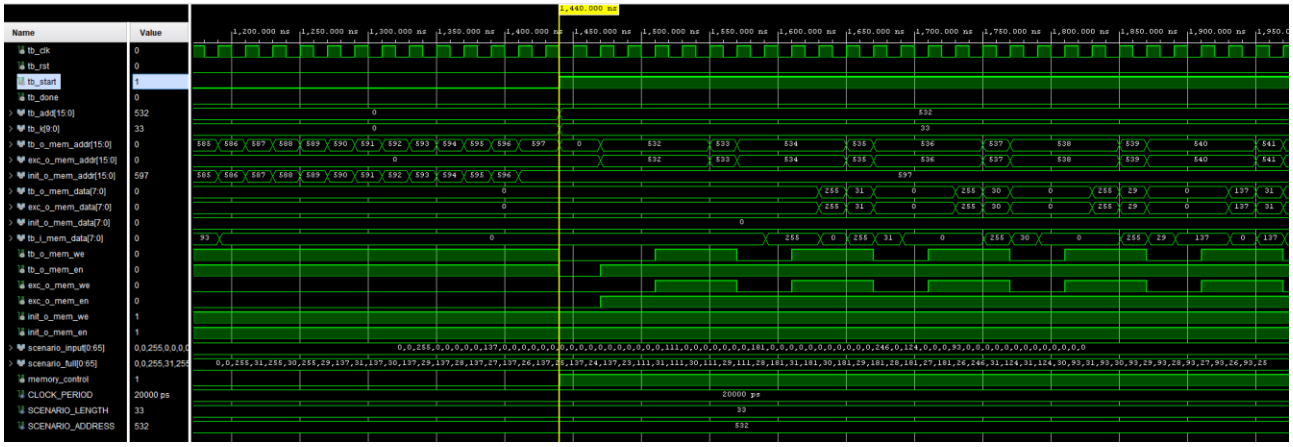
simulazione completa



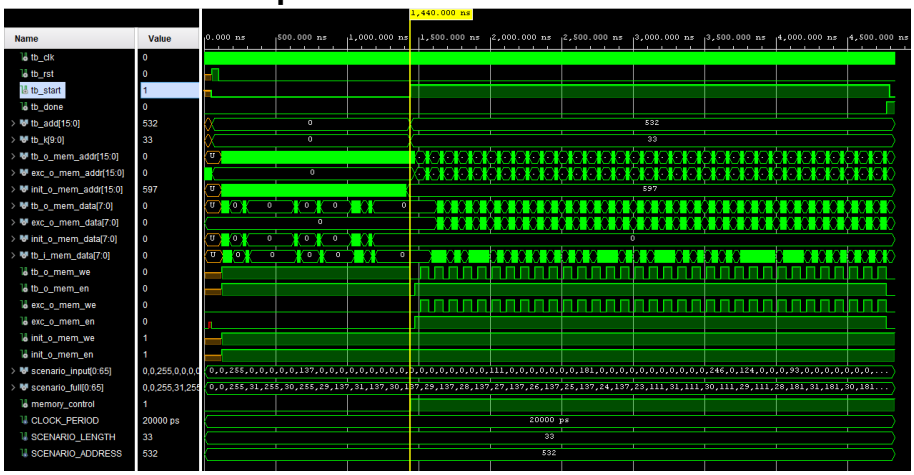
4) Prima Parola uguale a 0

Con questo TestBench si dimostra che il componente rispetta la specifica quando la parola iniziale è zero, garantendo che il valore di credibilità rimanga zero finché non viene trovata la prima parola valida.

inizio elaborazione sequenza



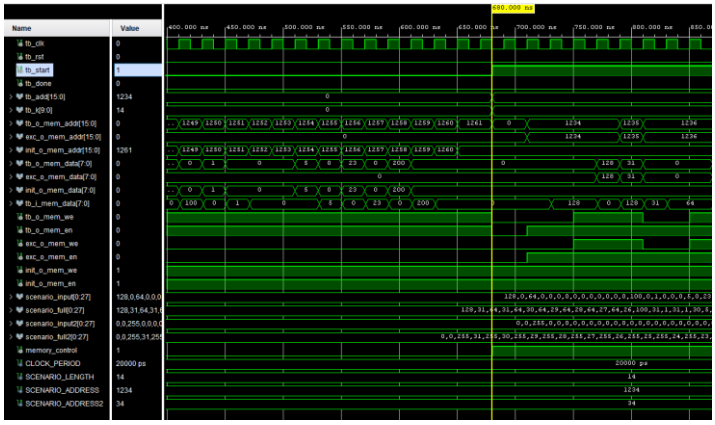
Simulazione completa



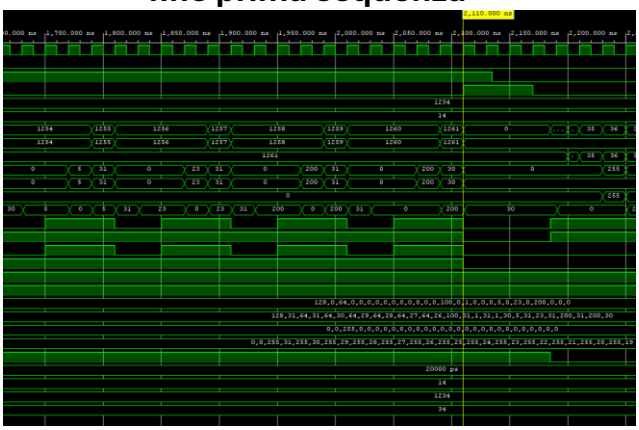
5) 2 Sequenze

Con questo TestBench si intende verificare che il componente sia in grado di elaborare correttamente più sequenze in serie, rispettando i segnali di start e done, e assicurando che questi si comportino conforme alle specifiche.

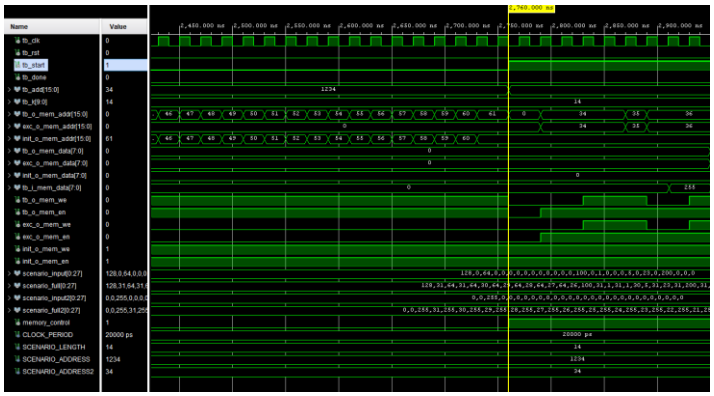
inizio prima sequenza



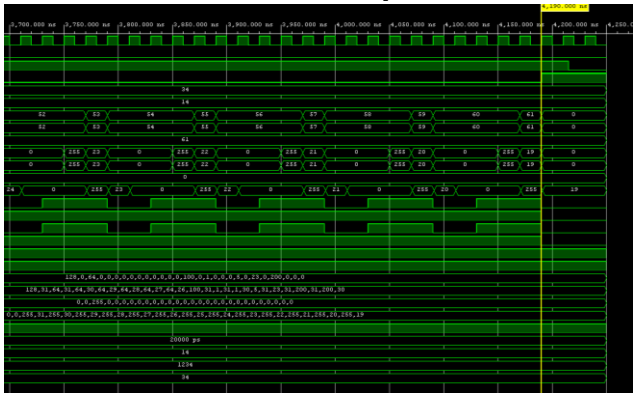
fine prima sequenza



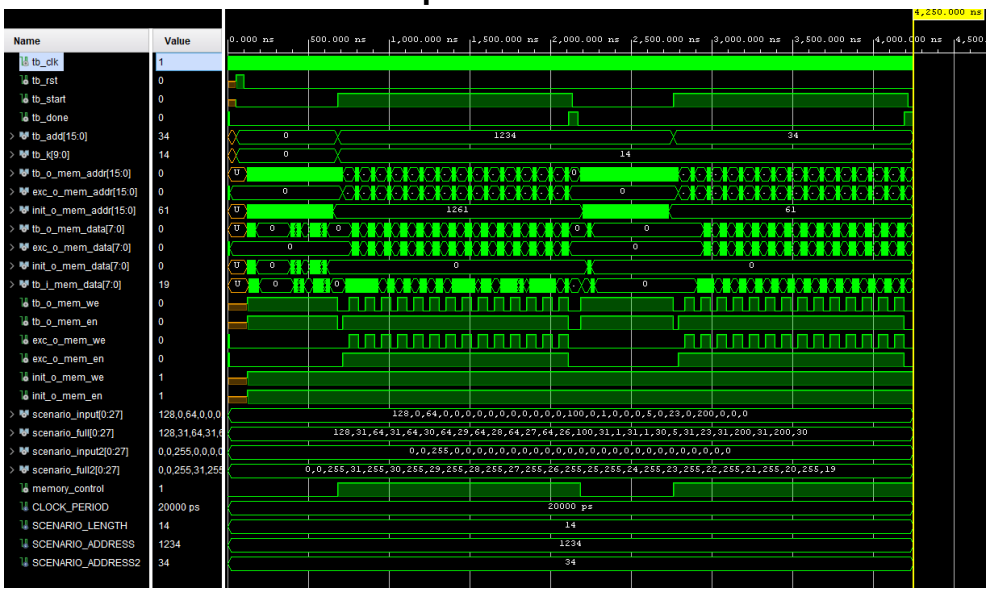
inizio seconda sequenza



fine seconda sequenza



simulazione completa



4. Conclusioni

Questo progetto ha segnato la mia prima esperienza concreta nel campo della programmazione hardware e mi ha permesso di mettere in pratica le conoscenze teoriche acquisite durante il corso.

Affrontare questo progetto ha comportato diverse sfide, legate soprattutto alla mia inesperienza. In particolare, ho incontrato difficoltà nei seguenti ambiti:

1) **Adattamento alla logica dei sistemi hardware**:
   A differenza della programmazione software, dove le modifiche sono immediatamente visibili, nella programmazione hardware ogni cambiamento richiede un ciclo di clock per diventare effettivo.
2) **Apprendimento del tool di sviluppo e del linguaggio VHDL**:
   L'acquisizione delle competenze necessarie per utilizzare efficacemente il tool di sviluppo e il linguaggio VHDL ha rappresentato un impegno notevole.

Ritengo che, sebbene impegnativa, questa esperienza sia stata estremamente istruttiva e mi abbia fornito una solida base per affrontare future sfide nel campo della programmazione hardware.