Prova Finale di Reti Logiche

Andrea Mastroberti -10736595
Diego Lecchi - 10681646

POLITECNICO
MILANO 1863

Indice

Capitolo 1

Introduzione

1.1 Panoramica del progetto

Il progetto consiste nello sviluppo di un componente specificato in VHDL che rispetti la specifica esplicata nel seguito. Il componente si interfaccia con una memoria ed ha il compito di completare una sequenza di byte presente in essa con valori di credibilità.

1.2 Tool utilizzati

L'applicativo utilizzato per sviluppare il componente e per la sua sintesi è la versione 2023.2 di Xilinx Vivado. L'FPGA impiegata è stata quella consigliata, ovvero Artix-7 FPGA xc7a200tfbg484-1.

1.3 Interfaccia del componente e requsiti

Il componente opera attraverso la seguente interfaccia

```
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
```

La specifica è definita a partire dall'abbassamento segnale di reset `i_rst`, in seguito a cui il segnale di start `i_start` verrà alzato, e rimarrà alto fino a che il componente non porti `o_done` ad 1. Il segnale `o_done` dovrà rimanere alto fintanto che `i_start` non venga portato a 0. Si assumerà sempre che il segnale di reset venga dato prima del primo start.

In particolare, una volta iniziata l'elaborazione, su `i_k` verrà indicato il numero di parole da elaborare, mentre su `i_add` verrà posto l'indirizzo di memoria da cui inizierà la sequenza di parole.
La memoria si divide in indirizzi pari contenenti valori nel range `[0, 255]` e di indirizzi dispari contenenti valori di credibilità nel range `[0, 31]`, quest'ultimi all'inizio dell'elaborazione sono sempre 0 (se su `i_add` è presente ADD, si intendono dispari gli indirizzi ADD + 1, ADD + 3, ..., ADD + $2 \cdot k - 1$ e per pari ADD, ADD + 2, ..., ADD + $2 \cdot k - 2$).

Si distinguono due casi: nel caso in cui il valore letto sull'indirizzo pari sia diverso da 0, nell'indirizzo dispari seguente il componente dovrà inserire il massimo valore di credibilità, ovvero 31; nel caso invece in cui venga letto 0 su un indirizzo pari, esso dovrà essere rimpiazzato con il valore dell'indirizzo pari precedente e nell'indirizzo dispari successivo inserito il valore di credibilità precedente decrementato di 1.
Il comportamento specificato nel caso di valore 0 nel primo indirizzo della sequenza è di mantenere invariata la sequenza fino al raggiungimento del primo valore in indirizzo pari diverso da zero.

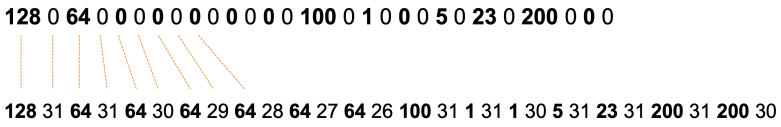


Figura 1.1: Esempio di funzionamento

1.4 Struttura della memoria

La memoria impiegata è derivata dalla user guide di VIVADO, e presenta la seguente interfaccia

```
clk : in std_logic;
we : in std_logic;
en : in std_logic;
addr : in std_logic_vector(15 downto 0);
di : in std_logic_vector(7 downto 0);
do : out std_logic_vector(7 downto 0)
```

L'indirizzamento delle parole avviene tramite il segnale a 16 bit `addr`, mentre i segnali `we, en` sono rispettivamente gli enable in scrittura e lettura. Infine, `di, do` contengono i dati in input ed in output alla memoria.

1.5 Esempio di funzionamento

Di seguito un esempio di elaborazione del componente, assumendo `K = 000A`, `ADD = 0064`.

Indirizzo	Parola (prima)	Parola (dopo)
0064	33	33
0065	00	1F
0066	00	33
0067	00	1E
0068	39	39
0069	00	1F
006A	18	18
006B	00	1F
006C	00	18
006D	00	1E
006E	18	18
006F	00	1D
0070	7E	7E
0071	00	1F
0072	00	7E
0073	00	1E
0074	C0	C0
0075	00	1F
0076	00	C0
0077	00	1E
...	...	...

Capitolo 2

Architettura

2.1 Segnali utilizzati

- `state_curr, state_prev` Memorizzano lo stato corrente e quello precedente della FSM. N.B. `state_prev` assume solo valori in **S_1**, **S_2**, **S_3**, **S_21**.
- `trust` Memorizza il valore di credibilità da assegnare.
- `jump` Contiene l'incremento da aggiungere al valore di `start_add` per indirizzare il prossimo valore da elaborare.
- `k` Valore letto su `i_k`.
- `start_add` Valore letto su `i_add`.
- `value` Valore letto su `i_mem_data`.
- `last_value` Ultimo valore letto su `i_mem_data` diverso da 0.

2.2 Descrizione della FSM

L'archiettetura sviluppata per l'implementazione del componente è una FSM costituita dai seguenti stati:

- **S_IDLE**: Qui la macchina è in uno stato di attesa. Quando il segnale di reset `i_rst` è alto, la macchina ritorna asincronamente a questo stato. Vengono inizializzate variabili e segnali, e si aspetta il segnale `i_start` per avviare l'esecuzione.
- **S_SETUP**: Questo stato prepara i valori per una nuova esecuzione. Viene salvato l'indirizzo di partenza `start_add` per la lettura/scrittura in memoria e viene attivato il segnale di enable `o_mem_en` per consentire la lettura della memoria. Se (k > 0), passa allo stato di attesa per la lettura dalla memoria **S_READMEM_WAIT**; altrimenti, imposta il segnale `o_done` per indicare la fine dell'esecuzione e passa allo stato **S_DONE**.

- **S_READ_MEM_WAIT**: Stato di attesa necessario affinchè la memoria abbia tempo per rispondere alla richiesta del componente, ovvero garantisce un ciclo di clock di attesa. Lo stato successivo è **S_READMEM**.

- **S_READMEM**: In questo stato viene letto il valore in output alla memoria (`i_mem_data`) e viene salvato in `value`. Lo stato che segue è **S_READMEM_CHOICE**.

- **S_READMEM_CHOICE**: Questo stato è uno "switch" decisionale che dirige alternativamente in **S_1**, **S_2** o **S_3**:
  se il valore letto è diverso da 0 la credibilità verrà impostata a 31 (`trust = 31`), `jump` incrementato di uno per puntare all'indirizzo dispari e lo stato successivo sarà **S_1** (`state_curr = S_1`). Se invece il valore letto è 0 e, o si sta leggendo il primo valore (`jump = 0`), o si proviene da una sequenza di soli 0 (`state_prev = S_3`), allora incrementa `jump` di 2 per puntare al valore da leggere successivo e si passa allo stato **S_3**. Altrimenti se il valore letto è 0 ma non è il primo valore letto (`jump` $\neq$ `0`), allora lo stato successivo sarà **S_2**.

- **S_1**: Salva il valore appena letto di `value` in `last_value` in previsione del fatto che il valore successivo da leggere potrebbe essere 0, imposta `state_prev = S_1` per ricordarsi il ramo della FSM da cui proviene, pone su `o_mem_addr` il valore `ADD + jump` per scrivere il valore di credibilità, ponendo `trust` (che è 31) su `o_mem_data`. Inoltre abilita la memoria alla scrittura ponendo `o_mem_we` a 1 e passa allo stato successivo **S_WRITEMEM**.

- **S_2**: Questo stato si occpua della sovrascrittura del valore 0 all indirizzo pari appena letto con l'ultimo valore diverso da 0 salvato in `last_value`, mentre la scrittura del valore di credibilità verrà fatta successivamente nello stato **S_21**. Nel dettaglio: imposta `state_prev = S_2`, incrementa `jump` di 1 (dato aggiornato che verrà utilizzato in **S_21**), se `trust` non è uguale a 0 lo decrementa, abilita la scrittura in memoria ponendo `o_mem_we` a 1 ed infine pone `last_value` su `o_mem_data` (`o_mem_addr` punta correttamente allo zero appena letto da sovrascrivere). Lo stato che segue è **S_WRITEMEM**.

- **S_3**: Questo stato gestisce il caso in cui la prima sequenza letta sia una serie di 0. Siccome non è necessaria alcuna scrittura, verifica se sono stati letti tutti i valori della sequenza tramite `k > jump/2`: se ciò è falso la sequenza è terminata, `o_done` viene posto a 1 e lo stato successivo sarà **S_DONE**, altrimenti viene posto su `o_mem_addr` l'indirizzo del valore successivo utilizzando `start_add` e sommandolo al valore `jump` calcolato nello stato precedente, per poi impostare **S_READMEM_WAIT** come stato successivo.

- **S_WRITE_MEM**: Attende un ciclo di clock affinchè la memoria concluda la scrittura. Se lo stato precedente era **S_2**, sposta `jump` affinchè punti al valore di credibilità. Lo stato successivo è **S_WRITEMEM_CHOICE**.

- **S_WRITEMEM_CHOICE**: Questo stato si occupa di scegliere il prossimo stato in base a `state_prev`: se è uguale a **S_2** allora lo stato successivo sarà **S_21**, altrimenti, se la sequenza è terminata pone `o_done` a 1 e passa a **S_DONE**. Se invece la

sequenza non è terminata (`k > jump/2`) `o_mem_addr` viene posto al prossimo indirizzo da leggere e lo stato successivo è **S_READMEM_WAIT**. In entrambi i casi se `state_prev` $\neq$ `S_2`, `o_mem_we` viene posto a 0.

- **S_21**: Viene scritto in memoria il valore di trust e impostato `state_prev` a **S_21**, concludendo l'elaborazione dello stato **S_2**. Il prossimo stato è **S_WRITE_MEM**.
- **S_DONE**: Stato di fine elaborazione. In corrsispondeza del raggiungimento di questo stato, il segnale `o_done` è posto alto. Il ritorno allo stato di idle avverrà nel momento in cui `i_start` sarà posto basso.

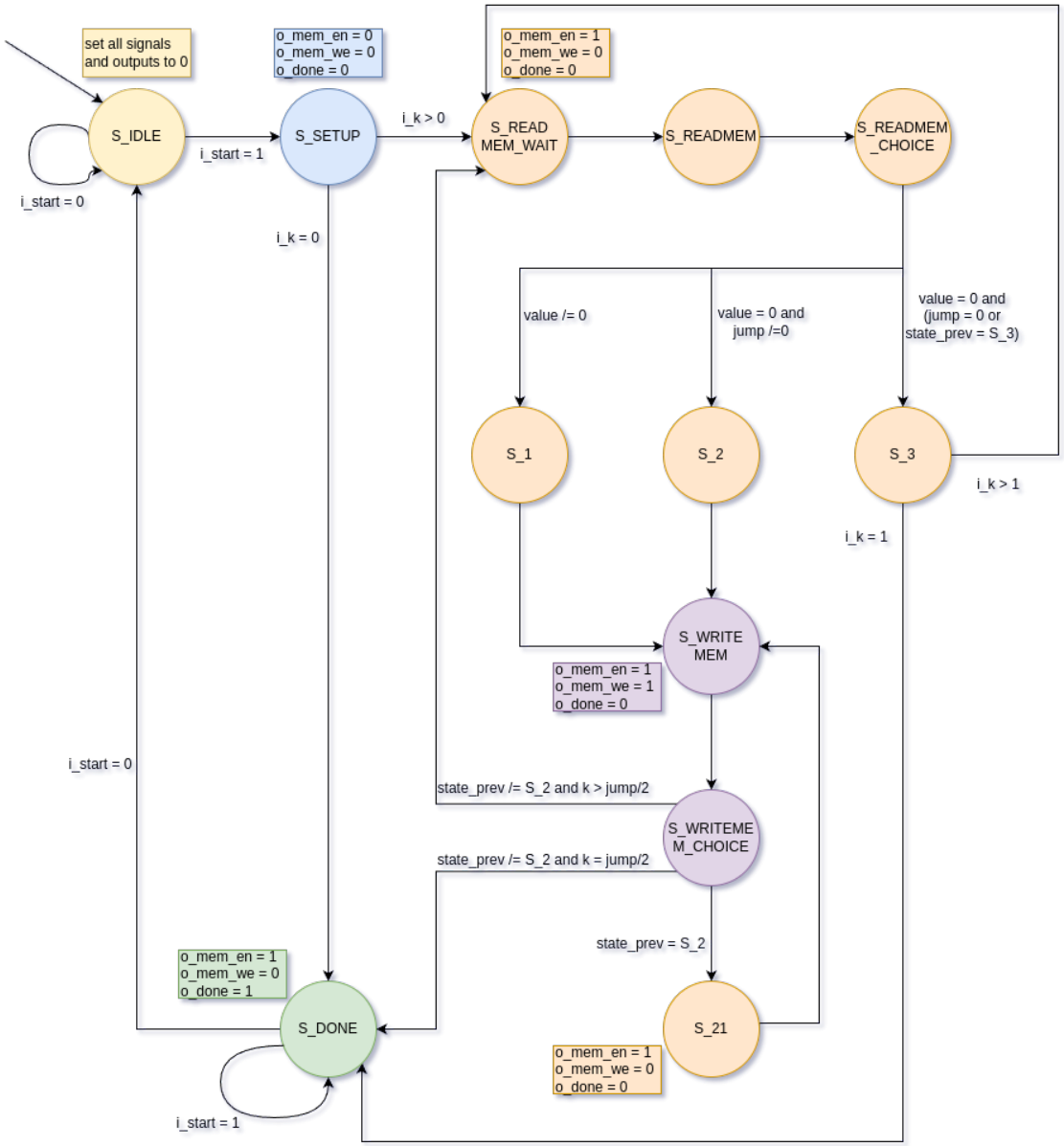


Figura 2.1: Rappresentazione della FSM

Capitolo 3

Risultati sperimentali

3.1 Report Utilization

Output di `report_utilization`: non sono presenti latch.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	110	0	0	134600	0.08
LUT as Logic	110	0	0	134600	0.08
LUT as Memory	0	0	0	46200	0.00
Slice Registers	98	0	0	269200	0.04
Register as Flip Flop	98	0	0	269200	0.04
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

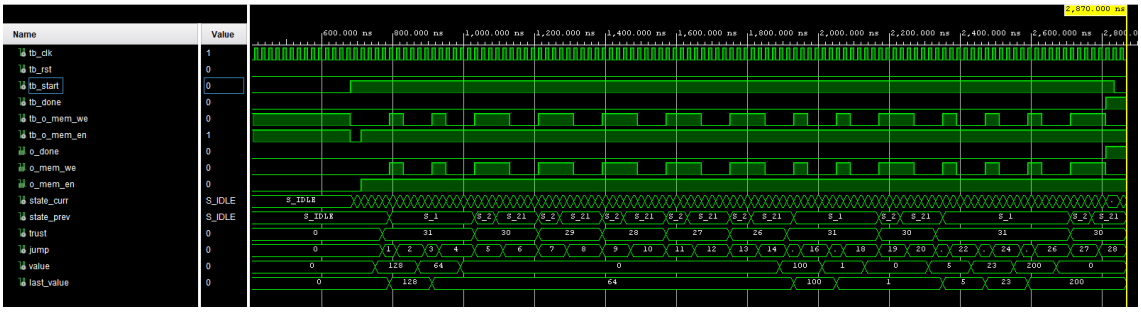
3.2 Report Timing

Lo slack time è la differenza tra il tempo di arrivo del segnale ed il massimo tempo di propagazione: è quindi auspicabile che lo slack time sia positivo. Il clock è di 20ns.

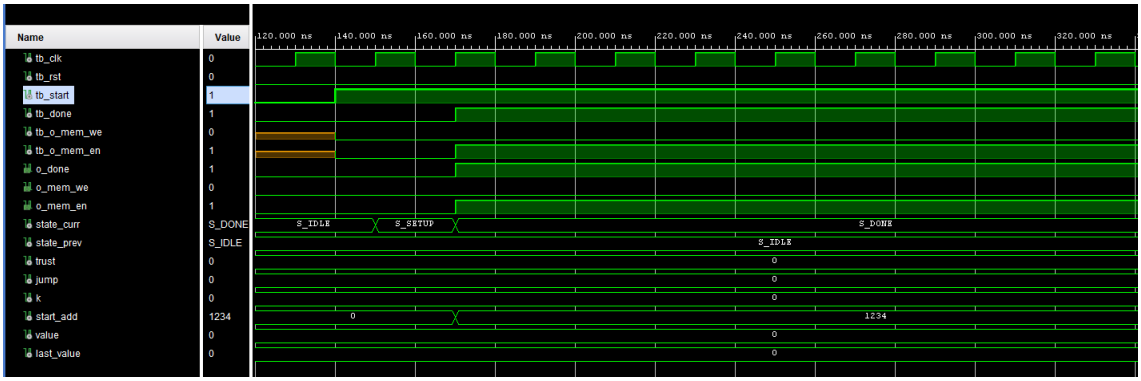
```
Timing Report

Slack (MET) :             14.667ns  (required time - arrival time)
```

3.3 Simulazioni

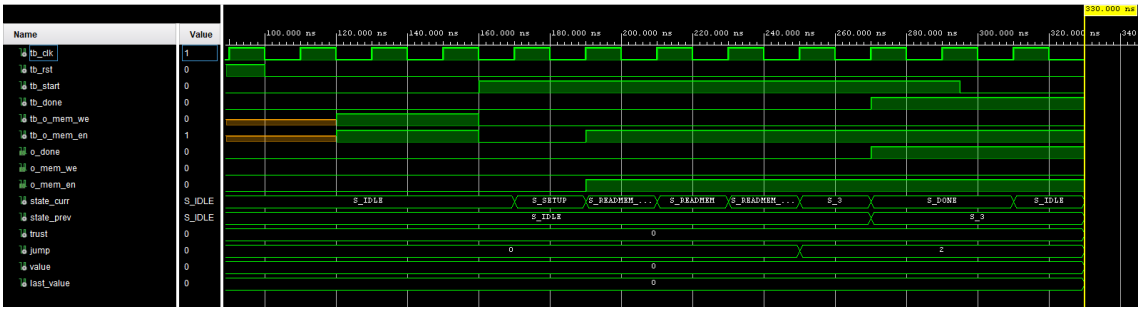
Il componente progettato è stato testato per garantirne il corretto funzionamento utilizzando testbench finalizzati ad esplorare edge case. I seguenti casi di testi vogliono quindi esplorare il comportamento del componente in situazioni estreme e critiche:

1. 

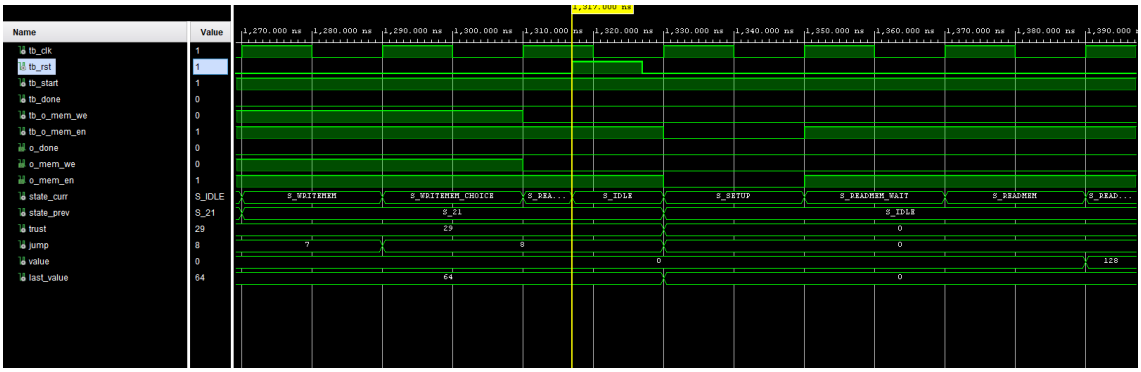
Testbench fornito: il componente passa il testbench fornito con successo.

2. 

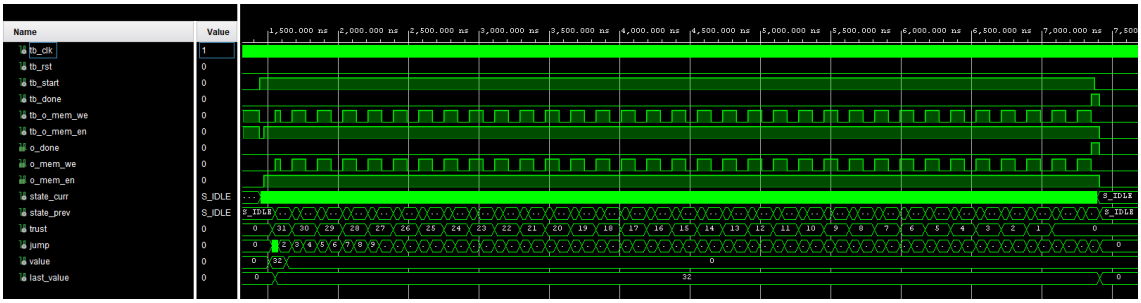
k = 0: Edge case in cui il numero di parole nella sequenza è nullo: il componente va direttamente in **S_DONE** per poi ritornare in **S_IDLE** non appena si abbassa il segnale `i_start`.

3. 

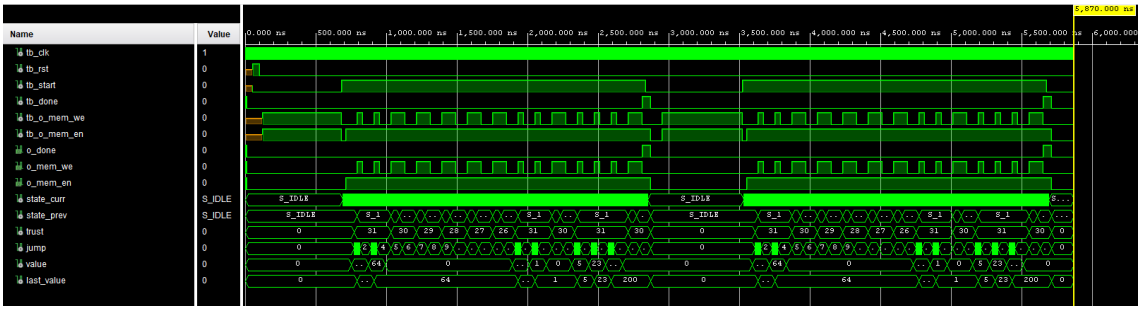
k = 1, value = 0: Questo testbench è stato realizzato appositamente per testare il corretto funzionamento del ramo `i_k = 1` dello stato **S_3**. Correttamente, non modifica la memoria e va in **S_DONE**.

4. 

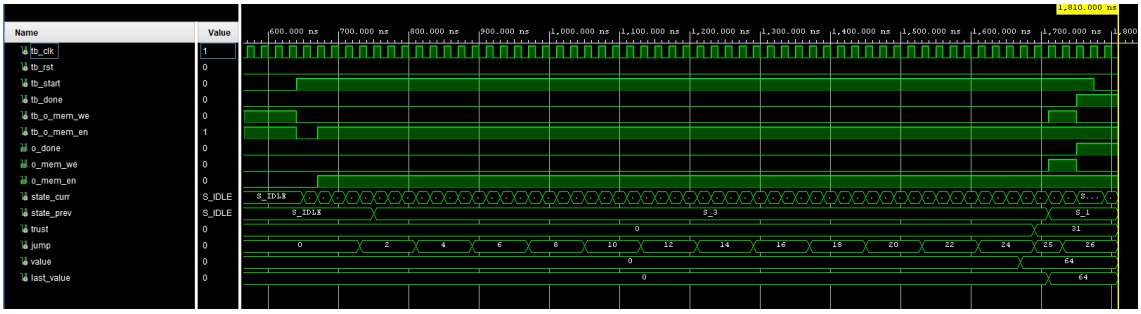
Sudden reset: questo testbench analizza il caso di reset asincrono improvviso. Il componente si comporta correttamente e si sposta immediatamente in **S_IDLE**.

5. 

Trust decreasing to 0: In questo testbench viene analizzato il caso in cui il valore di credibilità decresca fino a zero e si verifica che la credibilità non va sotto lo zero come da specifica.

6. 

Two consecutive sequences: Testbench in cui vengono prodotte in input due sequenze consecutive, si verifica che non è necessario il segnale di reset per far partire la seconda elaborazione, dopo la prima elaborazione, appena si abbassa il segnale `i_start`, la macchina torna nello stato **S_IDLE** dove attende l'alzata del segnale `i_start` per l'elaborazione successiva che viene eseguita come previsto.

7. 

Zeros sequence then value: Una sequenza di zero seguita da un valore. Il componente risponde correttamente.

Capitolo 4

Conclusioni

Il componente realizzato rispetta le specifiche e supera i test forniti sia in Behavioral Simulation che in Post-Synthesis Functional Simulation.

Nel gestire una sequenza con una serie di valori 0 iniziali, in principio lo stato **S_3** veniva utilizzato solo per il primo valore ed i successivi eventuali zeri venivano sovrascritti attraverso lo stato S2. Tuttavia, modificando le condizioni nello stato **READ_MEM_CHOICE** siamo riusciti ad ottimizzare l'elaborazione utilizzando lo stato **S_3** anche per gli zeri successivi nella sequenza, risparmiando 5 cicli di clock ogni valore 0 letto.