Prova Finale di Reti Logiche

Docente: Fornaciari William

Studente : Capodanno Mario
Codice Persona: 10804856

Studente: Michele Dussin
Codice Persona: 10809989

Anno Accademico 2023-2024

Contents

Chapter 1

Introduzione

1.1 Specifica del progetto

Il progetto descritto in questa relazione riguarda lo sviluppo di un componente VHDL che implementa una macchina a stati finiti (FSM) per l'elaborazione di una sequenza di dati memorizzati in una memoria RAM. Dalla specifica, viene richiesta la lettura di una sequenza di K parole W memorizzate in una memoria RAM secondo un certo ordine, partendo da un indirizzo specifico che viene fornito in input (valore ADD da specifica). I valori W sono compresi tra 0 e 255 con i valori pari a 0 che indicano 'dati non specificati o non affidabili', come nel caso della lettura di outlier da un sensore. Il compito è di sostituire i valori non speicificati con l'ultimo valore valido letto e di aggiungere un parametro di "credibilità" che diminuisce con ogni valore 0 consecutivo fino a un massimo di 31.

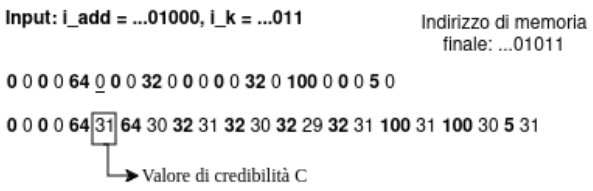


Figure 1.1: Esempio di funzionamento del componente richiesto

La nostra implementazione si basa su una macchina a stati che per gestisce autonomamente le operazioni principali richieste dalla specifica, quali la lettura e scrittura in memoria, la gestione dei contatori e la determinazione dello stato per la prossima transizione.

1.2 Interfaccia del componente

Il componente descritto presenta la seguente interfaccia:

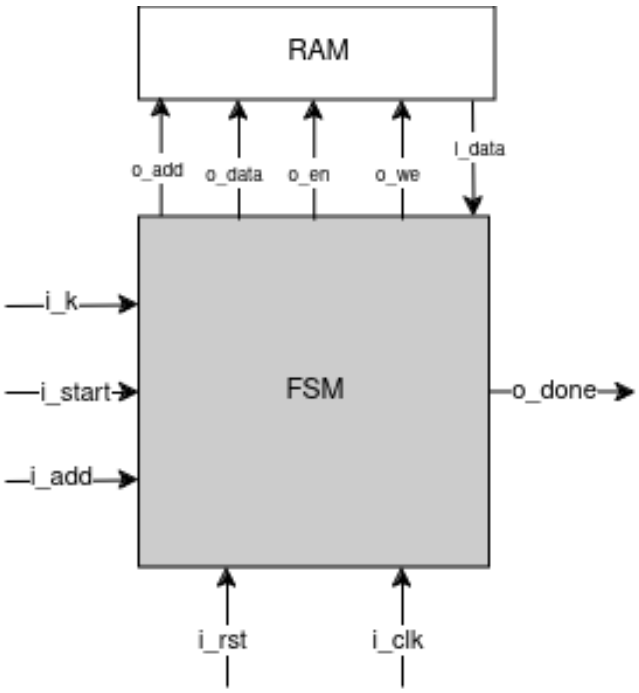


Figure 1.2: Architettura del componenente (*gli stati della fsm sono descritti nel prossimo capitolo*)

```
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;
```

Figure 1.3: Entity del componente

Il componente è composto da tre ingressi primari (START, ADD e K) e da un'uscita primaria a 1 bit (DONE). Dispone di un unico segnale di clock CLK e di un segnale di reset RESET, tutti sincroni e interpretati sul fronte del clock. L'uscita DONE deve essere 0 all'inizio del processo. Il modulo deve essere progettato in modo che il segnale RESET sia sempre dato prima del primo segnale START=1 e che il modulo venga resettato ogni volta che viene dato. Il modulo aggiorna la sequenza e i corrispondenti valori di confidenza, con zero dati rima-

Segnale	Direzione	Descrizione
i_clk	Input	Segnale di *clock* in ingresso generato dal *Test Bench*.
i_rst	Input	Segnale di *reset* che inizializza la macchina, pronta a ricevere il primo segnale di *start*.
i_start	Input	Segnale di *start* generato dal *Test Bench*.
i_k	Input	Segnale (vettore) *W* generato dal *Test Bench*, rappresentante la lunghezza della sequenza.
i_add	Input	Segnale (vettore) *ADD* generato dal *Test Bench* che rappresenta l'indirizzo da cui parte la sequenza da elaborare.
i_mem_data	Input	Segnale (vettore) che arriva dalla memoria e contiene il dato in seguito a una richiesta di lettura.
o_done	Output	Segnale di uscita che comunica la fine dell'elaborazione.
o_mem_addr	Output	Segnale (vettore) di uscita che manda l'indirizzo alla memoria.
o_mem_data	Output	Segnale (vettore) che va verso la memoria e contiene il dato che verrà successivamente scritto.
o_mem_en	Output	Segnale di *enable* da inviare alla memoria per poter comunicare (sia in lettura che in scrittura).
o_mem_we	Output	Segnale di *write enable* da inviare alla memoria (=1) per poter scrivere. Per leggere dalla memoria, esso deve essere 0.

Table 1.1: Tabella dei segnali di input e output del progetto.

nenti e zero valori di confidenza impostati fino al raggiungimento del primo punto di dati non nullo.

Chapter 2

Architettura

2.1 Segnali principali

Spiegazione dei Segnali Interni

Nel progetto della macchina a stati, sono stati utilizzati diversi segnali interni per gestire correttamente le operazioni di lettura e scrittura nella memoria RAM. Il segnale `saved_W`, un vettore di `std_logic` a 8 bit, viene utilizzato per memorizzare temporaneamente il dato letto dalla RAM. Questo segnale è importante per garantire che i dati corretti vengano scritti in memoria, specialmente quando il dato letto ha un valore diverso da zero e il bit meno significativo (`lsb`) è pari a 1.

```
TYPE states IS (IDLE, FETCH_INITIAL_DATA, ASK_READ_RAM, WAIT_READ_RAM, READ_W_RAM, WRITE_RAM, DONE);
signal curr_state : states := IDLE;

signal saved_W     : std_logic_vector(7 downto 0) := (others => '0');
signal counter_K   : std_logic_vector(15 downto 0) := (others => '0');
signal counter_Add : std_logic_vector(15 downto 0) := (others => '0');
signal counter_31  : std_logic_vector(4 downto 0) := "11111";
signal end_sng     : std_logic := '0';
signal lsb         : std_logic := '0';
```

Figure 2.1: Segnali definiti su vivado

Il segnale `counter_K`, anch'esso un vettore di `std_logic` a 16 bit, funge da contatore che tiene traccia del numero di cicli di lettura effettuati dalla RAM. Esso viene incrementato ad ogni ciclo e serve per calcolare l'indirizzo di memoria successivo da cui leggere o su cui scrivere.
Un altro segnale critico è `counter_Add`, che memorizza l'indirizzo di memoria attualmente in uso, combinando l'indirizzo base fornito in ingresso (`i_add`) con il valore di `counter_K`. Questo consente di accedere sequenzialmente alla memoria durante il processo di elaborazione.
Il segnale `counter_31`, un vettore di `std_logic` a 5 bit, è utilizzato per contare fino a 31, un valore che viene decrementato ogni volta che viene eseguita una scrittura in memoria. Questo contatore assicura che i valori di credibilità siano corretti e ritornino a 31 quando la parola W è diversa da zero.

Infine, il segnale `end_sng`, di tipo `std_logic`, è un flag che indica il completamento della sequenza di elaborazione, mentre `lsb` è un segnale che rappresenta il bit meno significativo dell'indirizzo di memoria corrente e viene utilizzato per determinare se l'indirizzo corrente è di tipo ADD+2 o ADD+1.
Questi segnali interni lavorano insieme per coordinare il flusso di dati tra il modulo e la memoria, assicurando che le operazioni vengano eseguite correttamente e in modo sincrono con il clock di sistema.

2.2 Stati della FSM

1. **IDLE**: Questo è lo stato iniziale della macchina a stati. Quando il sistema è in `IDLE`, la macchina non compie nessuna operazione. In questo stato, il componente attende che il segnale di avvio `i_start` venga impostato a '1' per iniziare l'elaborazione. Se `i_start` è '0', il componente rimane in `IDLE`. Quando `i_start` è impostato a '1', la macchina passa allo stato `FETCH_INITIAL_DATA` per iniziare il processo di calcolo degli indirizzi di memoria e lettura dei dati iniziali.

2. **FETCH_INITIAL_DATA**: In questo stato la macchina calcola l'indirizzo della memoria da cui leggere il dato iniziale sommando l'indirizzo base `i_add` con il valore del contatore `counter_K`. Questo indirizzo viene quindi salvato nel contatore degli indirizzi `counter_Add`. Se il valore di `counter_K` ha raggiunto il massimo valore possibile, ottenuto concatenando `i_k` con '0', la macchina considera terminata l'elaborazione e passa allo stato `DONE`. In caso contrario, il modulo continua il processo e passa allo stato `ASK_READ_RAM` per richiedere la lettura dalla memoria RAM.

3. **ASK_READ_RAM**: Durante questo stato, la macchina attiva il segnale `o_mem_en` per abilitare la lettura dalla memoria RAM. Il segnale `o_mem_we` viene mantenuto a '0' per indicare che non si sta eseguendo alcuna scrittura. La macchina inoltre determina il valore del bit meno significativo (`lsb`) dell'indirizzo di memoria, confrontando `counter_Add(0)` con `i_add(0)`. Questo confronto è utile per decidere se ci troviamo in uno degli indirizzi ADD+2*(K-1), ovvero se si sta leggendo la parola `W` o il dato successivo. Dopo aver effettuato queste operazioni, la macchina passa allo stato `WAIT_READ_RAM`.

4. **WAIT_READ_RAM**: : Questo è uno stato di transizione in cui la macchina attende che i dati siano disponibili per essere letti dalla memoria RAM. Non viene effettuata alcuna operazione specifica, ma è necessario per garantire che la lettura dei dati sia completata correttamente prima di procedere. Dopo questa attesa, la macchina passa allo stato `READ_W_RAM` per processare i dati letti dalla RAM.

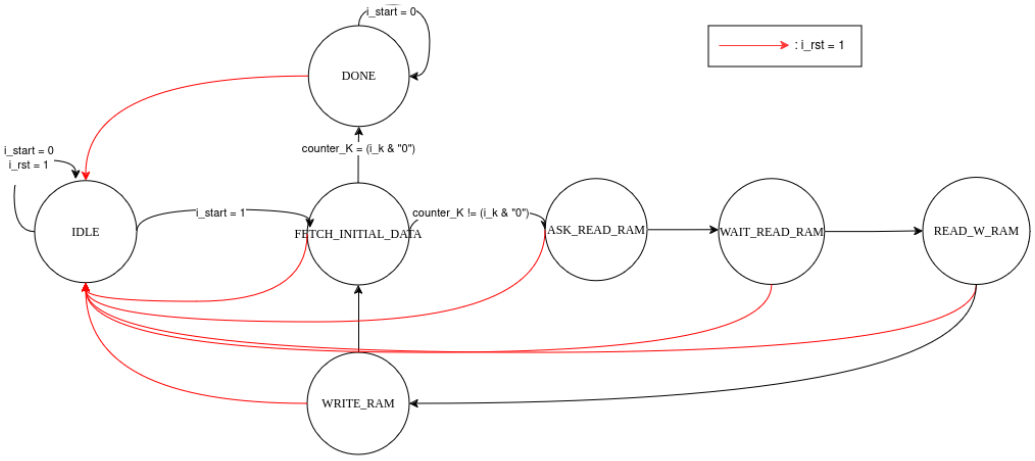


Figure 2.2: Rappresentazione degli stati della FSM

5. **READ_W_RAM**: : Se il valore di `lsb` è '1', il modulo verifica se il dato letto (`i_mem_data`) è zero. Se è zero e anche `saved_W` è zero, la macchina disattiva il segnale `o_mem_en`, indicando che la RAM non deve essere più abilitata, e il contatore `counter_31` viene resettato a "11111". Se `i_mem_data` non è zero, il dato viene salvato in `saved_W` e la macchina prepara la scrittura del dato nella RAM. Se invece `lsb` è '0', la macchina verifica se `saved_W` è diverso da zero e, se lo è, prepara i dati per la scrittura in RAM impostando `o_mem_data` al valore del contatore `counter_31` concatenato con tre zeri. Dopo queste operazioni, la macchina passa allo stato `WRITE_RAM`.

6. **WRITE_RAM**: : Durante questo stato, i dati preparati vengono scritti nella memoria RAM. Dopo aver completato la scrittura, i segnali di controllo della RAM (`o_mem_we` e `o_mem_en`) vengono disattivati per evitare ulteriori scritture. La macchina ritorna quindi allo stato `FETCH_INITIAL_DATA` per continuare l'elaborazione dei successivi indirizzi di memoria, ripetendo il ciclo finché non viene raggiunto lo stato finale.

7. **DONE**: : Questo stato indica la conclusione dell'elaborazione da parte della macchina a stati. Quando la macchina entra in `DONE`, il segnale `o_done` viene impostato a '1' per indicare che il processo è completato. La macchina rimane in questo stato fino a quando il segnale `i_start` non viene riportato a '0'. Quando `i_start` diventa '0', la macchina ritorna allo stato `IDLE`, pronta per un nuovo ciclo di elaborazione. Se `i_start` non viene disattivato, la

macchina rimane nello stato `DONE`, mantenendo i segnali nella configurazione di fine processo.

In conclusione, l'architettura progettata soddisfa pienamente la specifica descritta. Grazie alla gestione della macchina a stati finiti (FSM) e all'uso di segnali di controllo e contatori, il modulo garantisce che il segnale `DONE` sia inizialmente a 0 e che l'elaborazione inizi correttamente al ricevimento del segnale `START`. Ogni nuova elaborazione viene preceduta da un reset dei segnali interni e di quelli di output. Inoltre, l'aggiornamento dei valori di credibilità avviene come richiesto, con una gestione corretta degli edge case.

Chapter 3

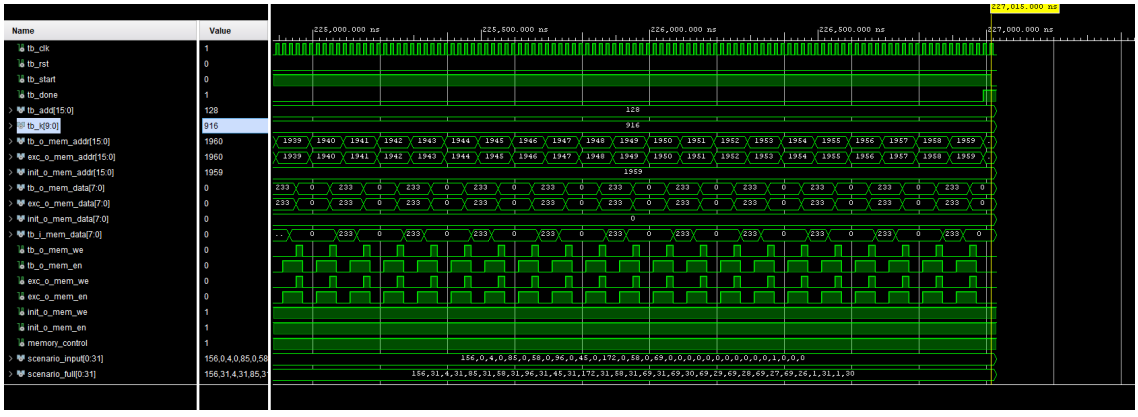
Risultati Sperimentali

3.1 Sintesi

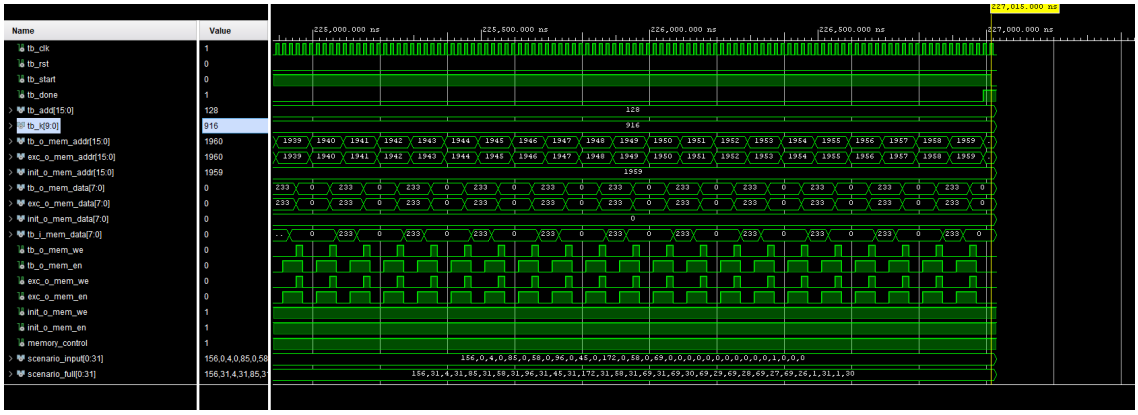
3.2 Simulazioni e Testbench

Per verificare il corretto funzionamento del componente sintetizzato, dopo averlo testato col test bench di esempio, abbiamo sottoposto il componente ad altri 15 testbench (tra i quali simulazioni di edge case e test generati automaticamente). Di seguito è fornita una breve desctizione dei test bench più significativi utilizzati con l'andamento dei segnali durante la simulazione:

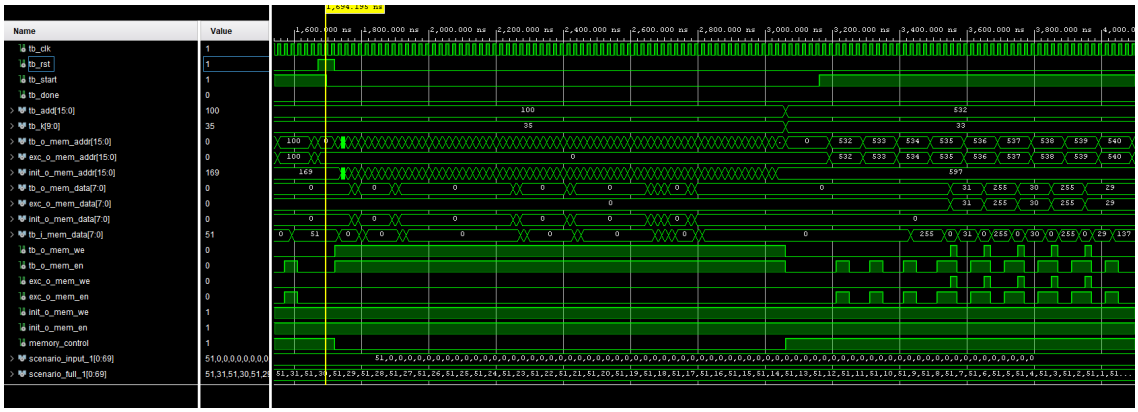
1. **Multiple sequenze di fila** : appena segnata la fine del processo di elaborazione con `o_done` una sequenza passa alla prossima senza necessità di porre `i_rst` = '1' del componente.



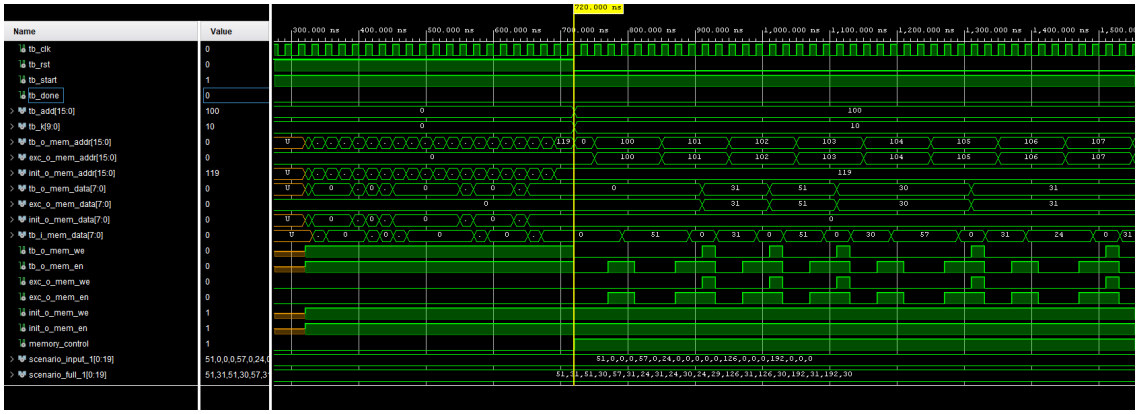
2. **Overflow contatore K** : per il conteggio degli indirizzi di memoria 'visitati' abbiamo deciso di utilizzare un contatore modulo K*2 con numero di bit pari a quello di `i_k`. Questa scelta può comportare alla possibilità che si verifichi un overflow se `i_k` è molto grande. Questo testbench verifica cosa accade con valori elevati di parole W.



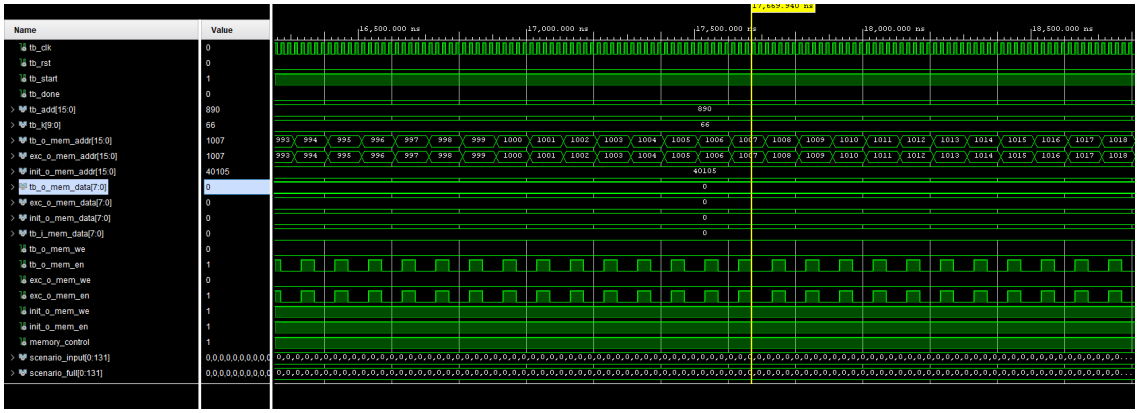
3. **Reset durante start** : durante l'elaborazione della sequenza se `i_rst` sale ad '1' allora il componente tornerà al primo stato(`IDLE`) della FSM preparandosi alla ricezione di una nuova sequenza e resettando i segnali interni e di output.



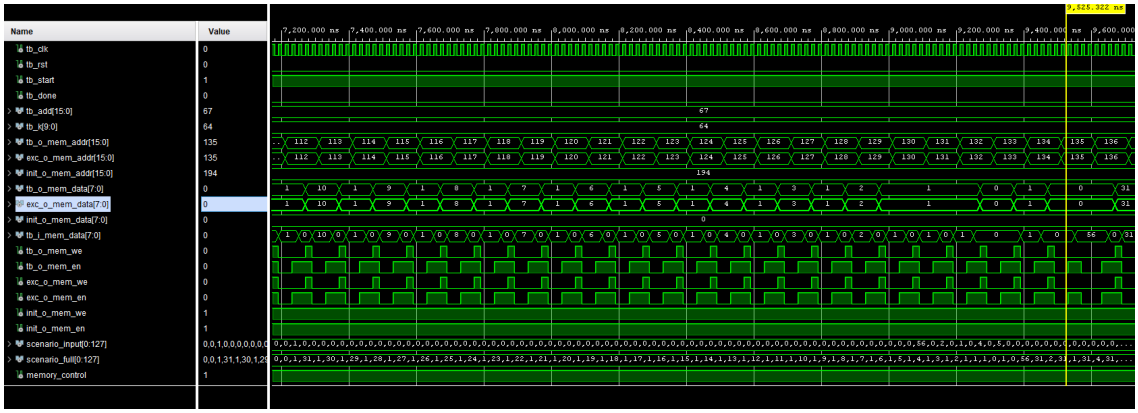
4. **Start durante reset** : se `i_start` rimane alto anche quando `i_rst` è a '1' allora il componente aspetta che `i_rst` si abbassi per poter partire con l'elaborazione della sequenza di dati.



5. **Sequenza solo '0'** : come da specifica, se inizialmente la sequenza presenta solo zeri, allora tali valori rimangono tali (quindi anche i rispettivi valori di credibilità), fino a che non si raggiunge una parola W diversa da zero.



6. **31+ Parole uguali** : se durante l'elaborazione della sequenza trovo più di 31 parole W che presentano lo stesso valore, allora i valori di credibilità dal 31-esimo valore in poi avranno valore pari a zero.



3.3 Report Utilization

Il report di sintesi riporta l'utilizzo dei seguenti componenti:

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	62	0	0	8000	0.78
LUT as Logic	62	0	0	8000	0.78
LUT as Memory	0	0	0	5000	0.00
Slice Registers	65	0	0	16000	0.41
Register as Flip Flop	65	0	0	16000	0.41
Register as Latch	0	0	0	16000	0.00
F7 Muxes	0	0	0	7300	0.00
F8 Muxes	0	0	0	3650	0.00

Figure 3.1: Report post-sintesi fornito da Vivado

È bene osservare come il numero di Latch sia 0. Data la natura prettamente `Behavioral` del codice, tale risultato è stato ottenuto prestando grande attenzione

ogni qual volta si è andati a utillizare costrutti come `if` andando a specificare esplicitamente il valore dei segnali in gioco e inserendo valori di default dove il valore non fosse stato definito esplicitamente e soprattuto grazie all'utilizzo di un unico processo sincronizzato con il segnale di clock.

3.4 Timing report

Dal Timing Report si evince che la differenza tra il tempo di clock (20 *ns*) e il tempo utilizzato per produrre un output è 15.951 *ns*, il tempo massimo per commutare di un segnale sarà quindi 4.049 *ns*.

```
Timing Report

Slack (MET) :             15.951ns  (required time - arrival time)
  Source:                 counter_K_reg[3]/C
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@10.000ns period=20.000ns})
  Destination:            counter_K_reg[0]/CE
                            (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:        3.667ns  (logic 1.817ns (49.550%)  route 1.850ns (50.450%))
  Logic Levels:           4  (CARRY4=2 LUT4=1 LUT6=1)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.081ns = ( 22.081 - 20.000 )
    Source Clock Delay      (SCD):    2.404ns
    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
```

Dal TestBench di prova fornito con K = 14 parole, il tempo di esecuzione è 0.0035501 *ms*.

```
Failure: Simulation Ended! TEST PASSATO
Time: 220030100 ps  Iteration: 1  Process: /project_tb/test_routine  File: D:/Poli/testing/testing.srcs/sim_1/new/tb.vhd
$finish called at time : 220030100 ps : File "D:/Poli/testing/testing.srcs/sim_1/new/tb.vhd" Line 187
```

Creando un appositamente un TestBench con K = 916 parole, il tempo di esecuzione è 0.2200301 *ms*.

```
Failure: Simulation Ended! TEST PASSATO
Time: 220030100 ps  Iteration: 1  Process: /project_tb/test_routine  File: D:/Poli/testing/testing.srcs/sim_1/new/tb.vhd
$finish called at time : 220030100 ps : File "D:/Poli/testing/testing.srcs/sim_1/new/tb.vhd" Line 187
```

Dai TestBench eseguiti è quindi lecito affermare che il tempo di esecuzione di una sequenza è linearmente proporzionale alle sue parole, con una complessità temporale di O(k).

Chapter 4

Conclusioni

La macchina progettata risponde efficacemente alle richieste della specifica, utilizzando meno del 25% del periodo di clock a disposizione del testbench assegnato. Inoltre, il componente ha passato numerosi test sia scritti manualmente per confermarne il funzionamento in casi limite, sia generati in maniera pseudo-casuale per verificarne la sua persistenza nel soddisfare i requisiti.
Nel corso della stesura del componente sono stati utilizzati diversi approcci tra i quali l'utilizzo di più processi per la transizione degli stati, il funzionamento di contatori, registri e l'assegnazione dello stato prossimo. Tramite un processo iterativo in cui abbiamo risolto warning post-sintesi e capito a fondo il motivo dei latch eventualmente generati, con risultato finale l'architettura presentata in questa relazione.
Data la natura della specifica abbiamo trovato l'approccio Behavioral molto intuitivo e facilmente implementabile. Sicuramente nel caso di progetti più complessi e grandi tale approccio non si presterebbe bene dato che non è garantito che generi circuiti logici sintetizzabili e sarebbe altamente consigliato un architettura modulare con una descrizione delle architetture dei vari moduli di tipo Dataflow, Structural o ibrido.