POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

# Progetto di Reti Logiche

## Ingegneria informatica

Autore: **Kevin Ziroldi**

Codice persona: 10764177
Matricola: 982143
Professore: Gianluca Palermo
Anno accademico: 2023/2024

# Indice

# 1 | Introduzione

## 1.1. Descrizione della specifica

La specifica richiede di implementare un modulo HW descritto in VHDL in grado di interfacciarsi con una memoria e di elaborare sequenze di dati.
Ogni sequenza è formata da `i_k` parole, ognuna seguita da un valore di credibilità. Ogni parola ha una dimensione di 1 byte e rappresenta un numero con valore compreso tra 0 e 255. Anche i valori di credibilità sono memorizzati in 1 byte, ma possono assumere valori tra 0 a 31.
Ogni byte in posizione dispari (prima, terza, ...) rappresenta una parola:

- Se la parola è pari a 0, il modulo la sostituisce con l'ultima parola diversa da zero letta, se esiste almeno una parola diversa da zero che precede la parola corrente;
- Se la parola è diversa da zero, il modulo non la modifica.

Ogni byte in posizione pari (seconda, quarta, ...) rappresenta un valore di credibilità:

- Se la parola a cui fa riferimento è pari a zero, la credibilità è pari a quella della parola precedente decrementata di 1, tenendo conto che non può diventare negativa;
- Se la parola a cui fa riferimento è diversa da zero, la credibilità è massima, ovvero pari a 31.

Se una o più parole in testa alla sequenza sono pari a zero, queste non vengono modificate e il loro valore di credibilità viene messo a zero.
Il modulo possiede dei segnali di input, `i_start` e `i_reset`, che lo controllano e possiede un segnale di output, `o_done`, che gli permette di comunicare quando ha terminato l'elaborazione.
Infine, il modulo deve essere in grado di elaborare anche più sequenze una dopo l'altra, anche senza essere resettato dopo una elaborazione.

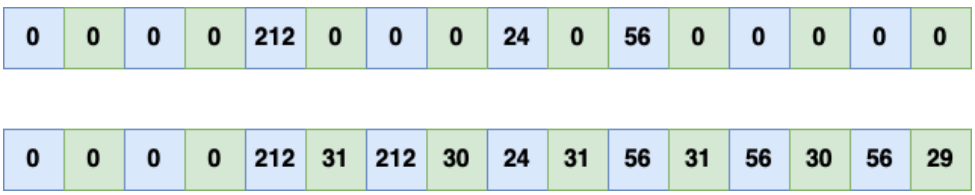


Figure 1.1: Esempio di elaborazione di una sequenza.

## 1.2. Interfaccia del componente

Il componente da realizzare deve avere la seguente interfaccia, descritta in VHDL:

```
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;
```

Segue una descrizione delle porte di ingresso e uscita del componente.

| | **Porte del componente** |
|---|---|
| **i_clk** | Segnale di clock generato dal Test Bench |
| **i_rst** | Segnale asincrono usato per inizializzare il modulo |
| **i_start** | Causa l'inizio di una nuova elaborazione |
| **i_k** | Lunghezza della sequenza da elaborare |
| **i_add** | Indirizzo di partenza della sequenza da elaborare |
| **o_done** | Uscita per comunicare la fine dell'elaborazione |
| **o_mem_addr** | Indirizzo da mandare alla memoria |
| **i_mem_data** | Dato ricevuto dalla memoria dopo una richiesta di lettura |
| **o_mem_data** | Dato da inviare alla memoria dopo una richiesta di scrittura |
| **o_mem_en** | Segnale di enable da mandare alla memoria per utilizzarla |
| **o_mem_we** | Operazione da compiere in memoria (0 = lettura, 1 = scrittura) |

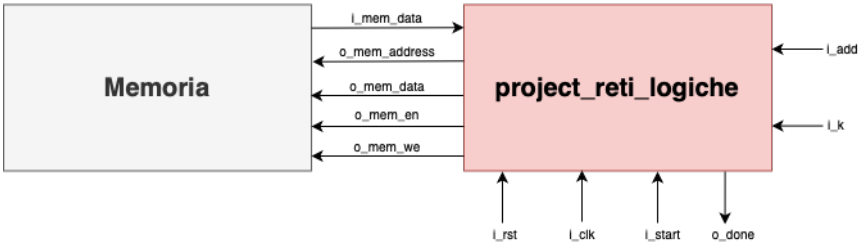


Figure 1.2: Rappresentazione grafica dell'interfaccia del componente.

## 1.3. Descrizione della memoria

Il modulo legge i dati della sequenza da una memoria, elabora la sequenza e sovrascrive la sequenza letta con quella che ha prodotto, secondo le regole di funzionamento precedentemente descritte.
La memoria RAM è implementata nel Test Bench, utilizza indirizzamento al byte e indirizzi di memoria a 16 bit, quindi sono disponibili $2^{16} = 65536$ indirizzi e celle di memoria. Sia le parole che i valori di credibilità hanno dimensione pari a 1 byte e occupano esattamente una cella di memoria. La sequenza è memorizzata a partire dalla cella con indirizzo `i_add` fino alla cella con indirizzo `i_add` + `2i_k` − 1.
Segue la rappresentazione grafica della memoria RAM, con riferimento all'elaborazione nella Figure 1.1.

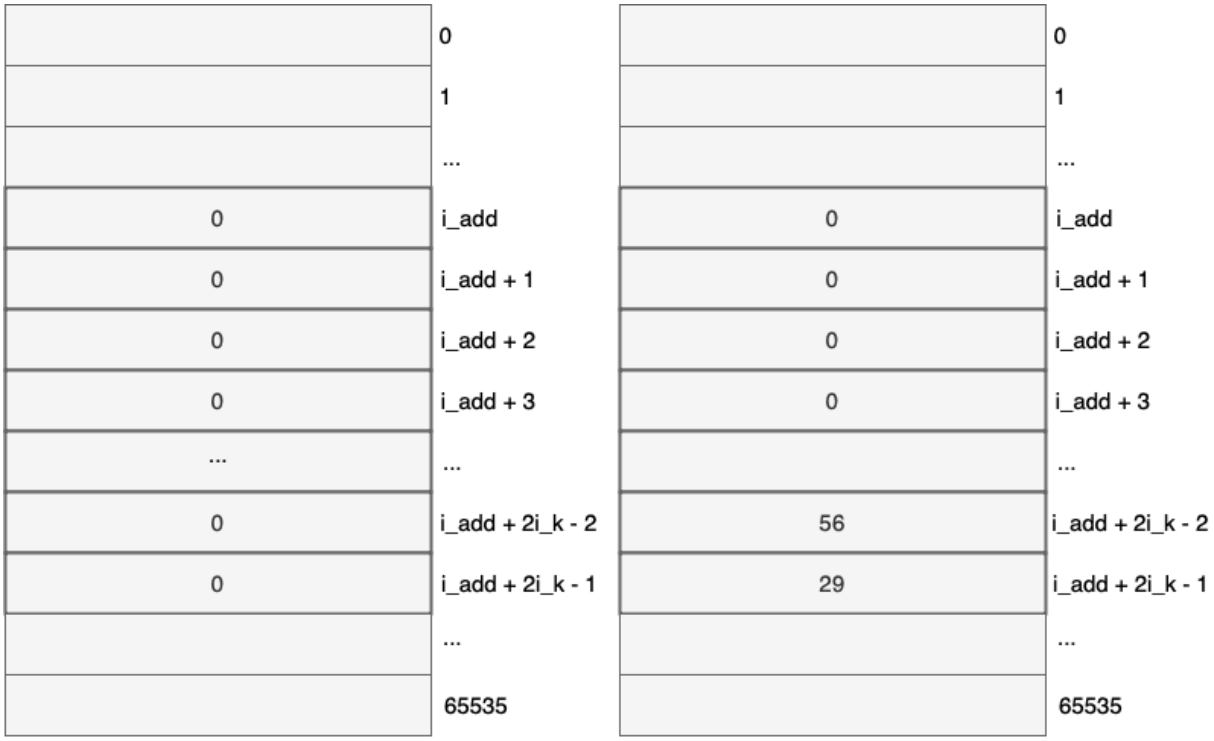


Figure 1.3: Rappresentazione della memoria RAM, prima dell'elaborazione (sulla sinistra) e dopo l'elaborazione (sulla destra).

# 2 | Architettura

## 2.1. Scelte progettuali

Nella progettazione della rete, ho deciso di adottare un approccio modulare, quindi ho descritto il componente `project_reti_logiche` con lo stile strutturale, istanziando dei component per tutti i moduli previsti nella fase di progettazione. In particolare, il modulo è formato da elementi di memoria, come Flip-flop, registri e contatori, e da una macchina a stati di Moore, che controlla il datapath attraverso dei segnali di enable. Mentre, per la descrizione dei componenti che formano il modulo, ho utilizzato l'approccio behavioral.

## 2.2. Schema funzionale

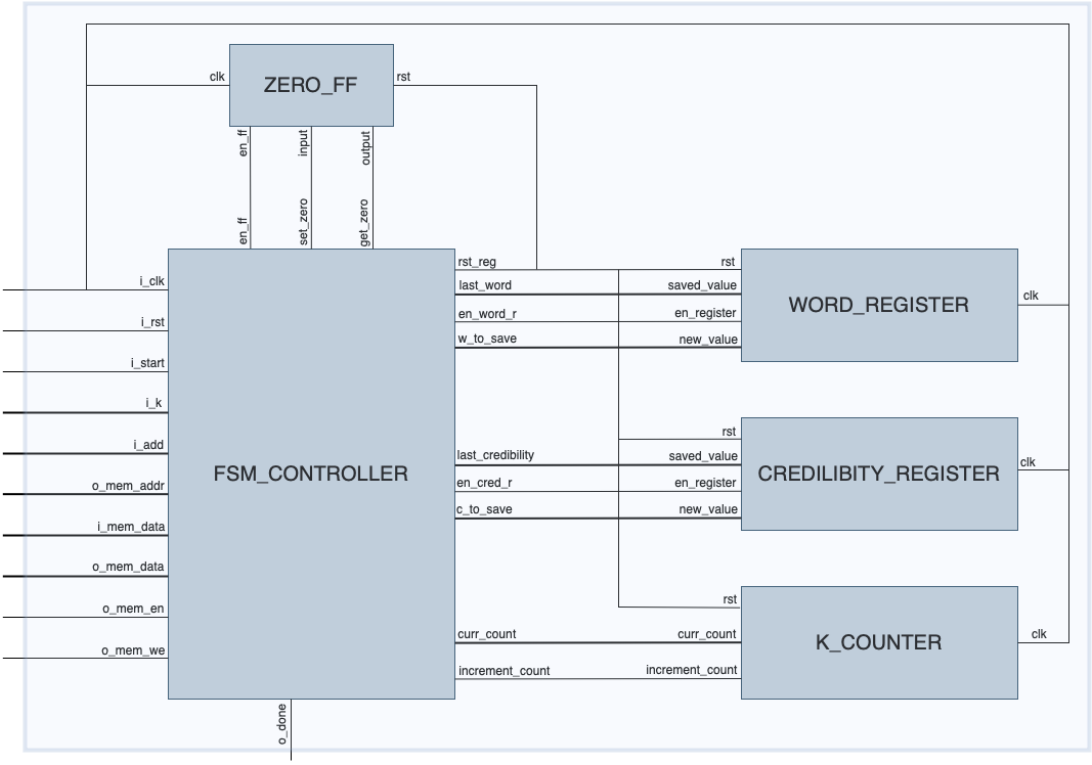


Figure 2.1: Schematico del componente progettato.

Nella tabella Table 2.1 riporto la descrizione del contenuto dei segnali interni utilizzati. Dato che `FSM_CONTROLLER` è il componente che controlla tutti gli altri, tutti i segnali interni collegano `FSM_CONTROLLER` a un altro componente. Ho adottato la convenzione per cui tutti i segnali interni hanno lo stesso nome della porta di `FSM_CONTROLLER` a cui si collegano.

| | **Segnali interni al componente project_reti_logiche** |
|---|---|
| **rst_reg** | Segnale di reset inviato ai componenti |
| **en_ff** | Segnale di enable che attiva `ZERO_FF` |
| **set_zero** | 1 se l'ultima parola letta è uno zero, 0 altrimenti |
| **get_zero** | 1 se l'ultima parola letta è uno zero, 0 altrimenti |
| **en_word_r** | Segnale di enable che attiva `WORD_REGISTER` |
| **last_word** | Parola memorizzata nel `WORD_REGISTER` |
| **w_to_save** | Parola da memorizzare nel `WORD_REGISTER` |
| **en_cred_r** | Segnale di enable che attiva `CREDIBILITY_REGISTER` |
| **last_credibility** | Valore di credibilità memorizzato nel `CREDIBILITY_REGISTER` |
| **c_to_save** | Valore di credibilità da memorizzare nel `CREDIBILITY_REGISTER` |
| **increment_count** | 1 se il contatore deve aumentare il proprio valore, 0 altrimenti |
| **current_count** | Numero di parole lette in memoria |

Table 2.1: Contenuto dei segnali interni al componente project_reti_logiche.

### 2.2.1. ZERO_FF

Il modulo `ZERO_FF` è un Flip-flop D che memorizza un bit, il cui valore permette di sapere se l'ultima parola letta dalla memoria sia uno zero o meno. Questa informazione viene utilizzata dalla FSM per il calcolo della credibilità.
La FSM può inizializzarne il contenuto tramite `rst`, può salvare un nuovo dato con `en_ff` e `input` (su un fronte di salita del `clk`) e può ottenere il dato memorizzato con `output`.

### 2.2.2. WORD_REGISTER

Il modulo `WORD_REGISTER` è un registro a 8 bit, usato per memorizzare l'ultima parola diversa da zero letta in memoria; questo valore viene usato dalla FSM quando deve modificare il valore di una parola nulla in memoria.
Al momento del reset, il suo contenuto viene inizializzato a zero; il contenuto viene aggiornato dalla FSM ogni volta che legge dalla memoria una parola diversa da zero, tramite i segnali `en_register` e `new_value`, su un fronte di salita del `clk`. Infine, la FSM può ottenere il dato memorizzato nel registro tramite `saved_value`.

### 2.2.3. CREDIBILITY_REGISTER

Il modulo `CREDIBILITY_REGISTER` è un registro a 8 bit e viene usato per memorizzare l'ultimo valore di credibilità scritto in memoria; questo valore viene usato dalla FSM nel caso in cui l'ultima parola letta in memoria sia pari a zero per ottenere il nuovo valore di credibilità.
La struttura e il funzionamento dei due registri è uguale, infatti i due registri sono due istanze della stessa entity `register_8bit`.

### 2.2.4. K_COUNTER

Il modulo `K_COUNTER` è un contatore a 10 bit, che memorizza il numero di parole lette in memoria, usato dalla FSM per sapere quando ha terminato l'elaborazione della sequenza. La FSM inizializza il valore del contatore a zero con `rst` e il contenuto viene aumentato di 1 ogni tramite `increment_count`, su un fronte di salita del `clk`, quando la FSM legge un nuovo valore dalla memoria. Infine, la FSM può accedere al contenuto del contatore tramite `curr_count` per sapere se leggere una nuova parola in memoria e continuare l'elaborazione o se terminarla e alzare il segnale `o_done`.

### 2.2.5. FSM_CONTROLLER

`FSM_CONTROLLER` è una FSM di Moore e costituisce il centro di controllo dell'intero modulo; infatti, gestisce i segnali di ingresso e uscita del modulo `project_reti_logiche` e contiene la logica e i segnali per controllare gli altri componenti.
La FSM ha tre processi:

- Un processo `state_reg` si occupa della gestione del reset asincrono e del passaggio tra gli stati in corrispondenza di un fronte di salita del `clk`;
- Un processo `lambda_delta` definisce la funzione di uscita $\lambda$ e la funzione di stato prossimo $\delta$;
- Un processo `elaboration` si occupa di elaborare la sequenza, interagendo con la memoria e con i registri.

La FSM ha 10 stati, dei quali segue una descrizione.

## 2.3. Diagramma degli stati di FSM_CONTROLLER

### 2.3.1. START

È lo stato iniziale, in cui la FSM si trova dopo un segnale di reset, ovvero dopo che `i_rst` = 1. La FSM rimane in questo stato finché `i_start` = 0, quando `i_start` = 1, la FSM passa nello stato `INIT`.

### 2.3.2. INIT

È lo stato in cui la FSM alza il segnale `rst_reg` per inizializzare `ZERO_FF`, `WORD_REGISTER`, `CREDIBILITY_REGISTER` e `K_COUNTER`. Al ciclo di clock successivo, la FSM passa spontaneamente allo stato `CHECK_K`.

### 2.3.3. CHECK_K

In questo stato la FSM controlla, grazie al contatore, se ha esaurito le parole da leggere, ovvero se ha elaborato l'intera sequenza. La condizione viene verificata tenendo conto che il valore del contatore non è ancora stato aggiornato. Se ha terminato la sequenza, passa allo stato `DONE_UP`, altrimenti va in `WAIT_INCREMENT`.
Inoltre, la FSM alza il segnale `increment_count` per aumentare il valore del contatore, dato che sta per leggere una nuova parola (se non ha terminato la sequenza).

### 2.3.4. WAIT_INCREMENT

Se la FSM si trova in questo stato, non ha finito di elaborare la sequenza, ovvero deve leggere ancora una o più parole dalla memoria. Si tratta di uno stato di attesa, in cui la FSM attende che il contatore incrementi il conteggio.

### 2.3.5. READ_W

Il contatore ha incrementato il conteggio, la FSM legge dalla memoria una nuova parola, specificando l'indirizzo da cui leggere tramite `o_mem_addr` e abilitando la memoria in lettura tramite `o_mem_en` e `o_mem_we`.

### 2.3.6. WAIT_DATA

Si tratta di uno stato di attesa, in cui la FSM attende che la RAM renda disponibile il dato richiesto su `i_mem_data`.

### 2.3.7. UPDATE_W

Quando la FSM si trova in questo stato, il dato richiesto è disponibile su `i_mem_data`. La FSM analizza il dato e decide come elaborarlo:

- Se la parola è nulla e tutte quelle precedenti sono nulle (ovvero `last_word` è pari a zero), la FSM non scrive né in memoria né nel `WORD_REGISTER`;
- Se la parola è nulla, ma almeno una di quelle precedenti non lo è, la FSM sostituisce la parola nella memoria con `last_word`, ma non modifica il contenuto del registro;
- Se la parola è diversa da zero, la FSM la salva in `WORD_REGISTER`, ma non modifica il contenuto della memoria.

Infine, la FSM modifica il contenuto di `ZERO_FF` in base al fatto che la parola letta sia zero o meno, tramite `en_ff` e `set_zero`.

### 2.3.8. UPDATE_C

La FSM elabora il valore di credibilità relativo all'ultima parola letta, sfruttando il contenuto di `ZERO_FF` per sapere se questa è nulla o meno:

- Se la parola è diversa da zero, la FSM scrive sia in memoria che in `CREDIBILITY_REGISTER` il valore di credibilità massimo, ovvero 31;
- Se la parola è nulla e l'ultima credibilità, memorizzata in `CREDIBILITY_REGISTER` è nulla, la FSM salva in memoria credibilità zero, ma non modifica il contenuto del registro, dato che la credibilità non può diventare negativa;
- Se la parola è nulla, ma l'ultima credibilità no, la FSM decrementa di 1 il valore di credibilità e lo scrive sia in memoria che nel registro.

### 2.3.9. DONE_UP

In questo stato, la FSM ha terminato l'elaborazione e alza il segnale `o_done` per segnalarlo. La FSM rimane in questo stato finchè `i_start` = 1. Quando `i_start` = 0, passa allo stato `DONE_DOWN`.

### 2.3.10. DONE_DOWN

La FSM abbassa `o_done`. Finchè `i_start` = 0, la FSM rimane in questo stato; se `i_start` viene alzato, la FSM torna nello stato `INIT` per elaborare una nuova sequenza.

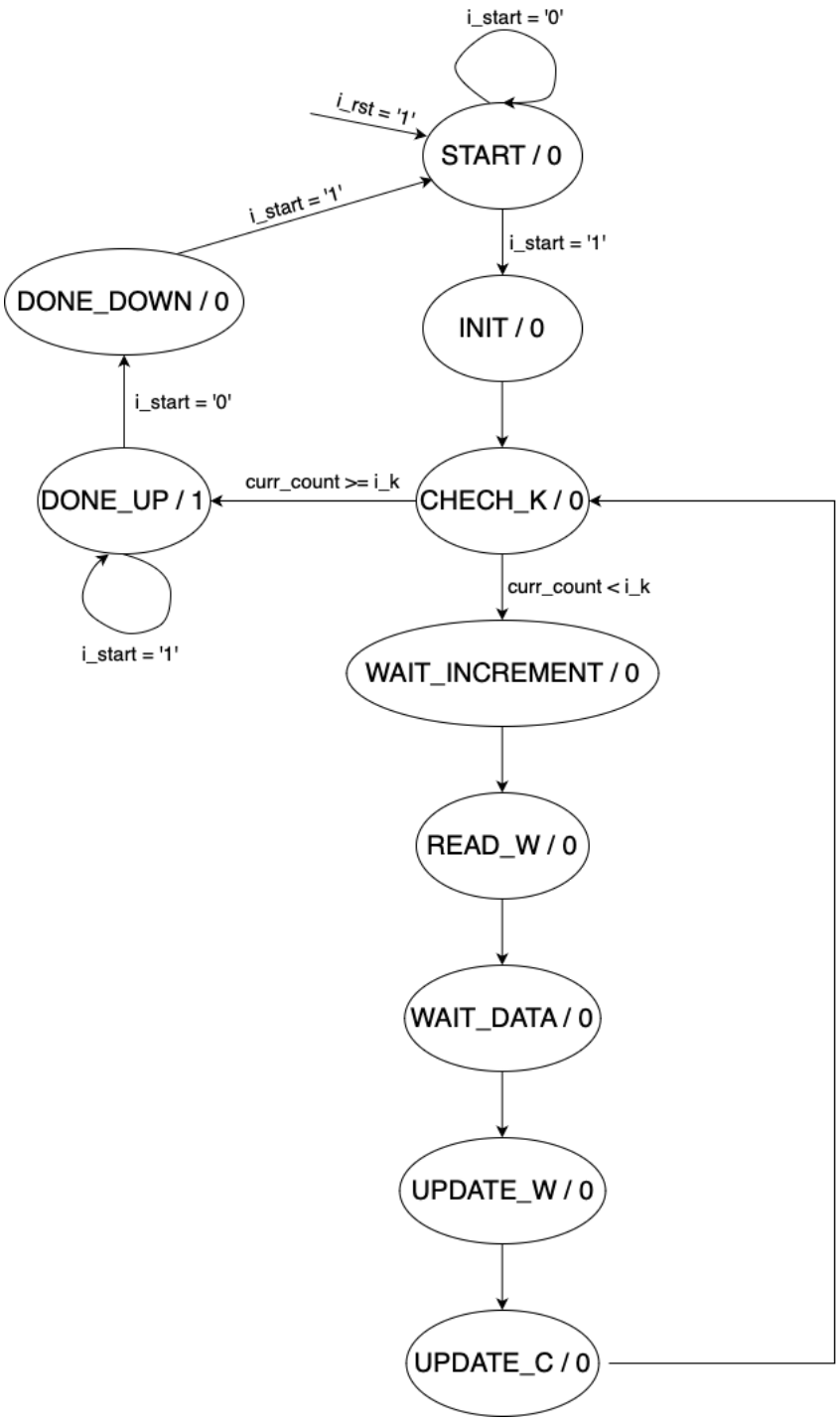


Figure 2.2: Diagramma degli stati di FSM_CONTROLLER.

# 3 | Sintesi

Ho sintetizzato il componente usando come FPGA target quella consigliata, ovvero la Artix-7 FPGA xc7a200tfbg484-1; la fase di sintesi è andata a buon fine, senza generare nessun errore. Vivaho ha generato alcuni warning, che indicano che ci sono dei segnali utilizzati in un processo e che non fanno parte della sua sensitivity list: non ho modificato il contenuto della sensitivity list, poiché non ho reputato necessario l'aggiunta di tali segnali in base al funzionamento del mio modulo.

## 3.1. Utilization Report

Utilizzando il comando `report_utilization` di Vivado, ho ottenuto il seguente risultato:

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice LUTs* | 85 | 0 | 0 | 134600 | 0.06 |
| LUT as Logic | 85 | 0 | 0 | 134600 | 0.06 |
| LUT as Memory | 0 | 0 | 0 | 46200 | 0.00 |
| Slice Registers | 36 | 0 | 0 | 269200 | 0.01 |
| Register as Flip Flop | 36 | 0 | 0 | 269200 | 0.01 |
| 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 |

La prima osservazione su questo report riguarda la percentuale irrisoria di risorse utilizzate rispetto a quelle dispobibili: lo 0,06% delle Slice LUTs e lo 0,01% degli Slice Registers. La seconda osservazione riguarda invece il numero di Latch, che è pari a 0. Nella creazione del modulo ho utilizzato solamente elementi di memoria sincroni, come Flip-Flop D, registri e contatori. Il risultato del report indica che Vivado non ha inferito alcun Latch, ovvero non ha introdotto elementi di memoria asincroni e indesiderati.

## 3.2. Timing Report

Da specifica, il progetto deve funzionare con un periodo di clock di 20 ns. Per verificare questo requisito ho creato un file contenente il seguente constraint sul tempo:

```
create_clock -period 20 -name clock [get_ports i_clk]
```

Successivamente ho verificato che il constraint fosse rispettato sfruttando il comando di Vivado `report_timing`, che ha prodotto il seguente output:

```
Timing Report

Slack (MET) :             16.768ns  (required time - arrival time)
  Source:                 k_counter/count_value_reg[3]/C
                          (rising edge-triggered cell FDCE clocked by clock
                          {rise@0.000ns fall@10.000ns period=20.000ns})
  Destination:            fsm_controller/FSM_onehot_current_state_reg[3]/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:        3.081ns  (logic 1.961ns (63.648%)  route 1.120ns
                          (36.352%))
  Logic Levels:           4  (CARRY4=2 LUT2=1 LUT4=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
```

Dal report possiamo osservare come il constraint sul tempo sia rispettato (MET) e con un margine abbondante, dato che lo Slack Time è pari a 16.768 ns, a fronte di un periodo di clock di 20 ns.

# 4 | Risultato simulazioni

Al fine di testare il corretto funzionamento del modulo richiesto, ho realizzato vari Test Bench che stimolassero il componente nei casi limite.
Per ognuno dei Test Bench creati, ho eseguito Behavioral Simulation, Post-Synthesis Functional Simulation e Post-Synthesis Timing Simulation, passando i test in tutti i casi.
Per questione di brevità, riporto le forme d'onda ottenute solo per alcuni dei Test Bench che ho realizzato e utilizzato.

## 4.1. Test Bench fornito

Il primo Test Bench che ho utilizzato è stato quello fornito insieme alla specifica, esso non testa nessun caso particolare, ma l'ho utilizzato come punto di partenza per verificare il corretto funzionamento del modulo.

## 4.2. Sequenza con vari zero in testa

Ho realizzato un Test Bench per verificare il funzionamento del modulo quando la sequenza ha più parole pari a zero in testa.
Sequenza da elaborare:

```
(0,0,0,0,0,0,0,0,20,0,0,0,200,0,0,0,0,0,0,0)
```

Sequenza elaborata:

```
(0,0,0,0,0,0,0,0,20,31,20,30,200,31,200,30,200,29,200,28)
```

## 4.3. Sequenza con più di 31 parole pari a zero

Ho realizzato un Test Bench per testare il corretto funzionamento del modulo quando la sequenza presenta un numero di parole pari a zero maggiore di 31; in questo caso la credibilità deve essere decrementata fino a zero e poi rimanere pari a zero, senza diminuire ulteriormente.
Sequenza da elaborare:

```
(0,0,0,0,34,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

Sequenza elaborata:

```
(0,0,0,0,34,31,34,30,34,29,34,28,34,27,34,26,34,25,34,24,34,23,
 34,22,34,21,34,20,34,19,34,18,34,17,34,16,34,15,34,14,34,13,
 34,12,34,11,34,10,34,9,34,8,34,7,34,6,34,5,34,4,34,3,34,2,34,1,
 34,0,34,0,34,0,34,0,34,0,34,0,34,0,34,0,34,0,34,0,34,0,34,0)
```

## 4.4. Sequenza di soli zeri

Il Test Bench verifica il caso in cui la sequenza da elaborare sia formata da soli zeri, in questo caso mi aspetto che anche la sequenza di uscita sia formata da parole nulle con credibilità nulla.
Sequenza da elaborare:

```
(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

Sequenza elaborata:

```
(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

## 4.5. Test Bench con i_add = 0

Ho realizzato un Test Bench che testa una sequenza memorizzata in memoria a partire dall'indirizzo `i_add` = 0. L'obiettivo è quello di verificare che il modulo non acceda a indirizzi di memoria non disponibili.
Sequenza da elaborare:

```
(3,0,0,0,10,0,0,0,32,0,17,0,0,0,0,0,0,0,0,0)
```

Sequenza elaborata:

```
(3,31,3,30,10,31,10,30,32,31,17,31,17,30,17,29,17,28,17,27)
```

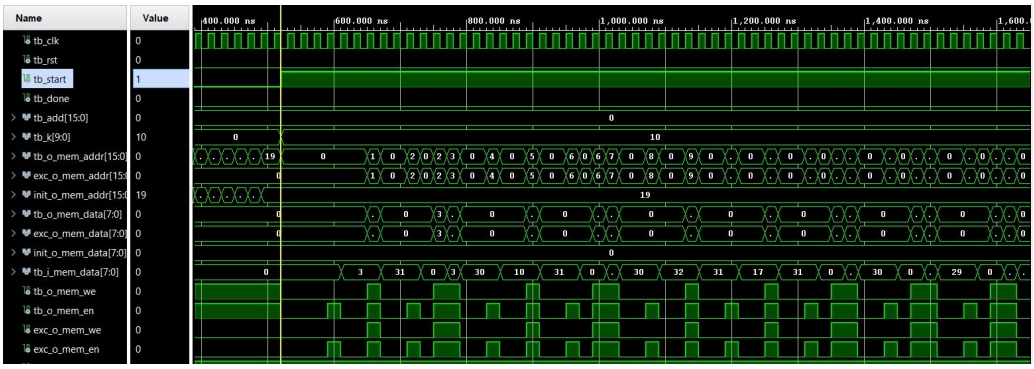


Figure 4.1: Waveform Post-Synthesis Functional Simulation con i_add = 0.

## 4.6. Test Bench che usa l'ultimo indirizzo di memoria

Questo Test Bench testa il caso speculare rispetto a quello precedente, ovvero quello in cui l'ultimo valore della sequenza è memorizzato nell'ultimo indirizzo di memoria disponbile, ovvero l'indirizzo 65535. Anche in questo l'obiettivo è verificare che il modulo non acceda ad indirizzi di memoria non validi.
Sequenza da elaborare:

`(0,0,0,0,10,0,0,0,32,0,17,0,0,0,0,0,0,0,0,0)`

Sequenza elaborata:

`(0,0,0,0,10,31,10,30,32,31,17,31,17,30,17,29,17,28,17,27)`

Nel Test Bench `i_add` = 65516 e `i_k` = 10, l'indirizzo in cui è salvato l'ultimo valore della sequenza è `i_add` + 2`i_k` − 1 = 65535.

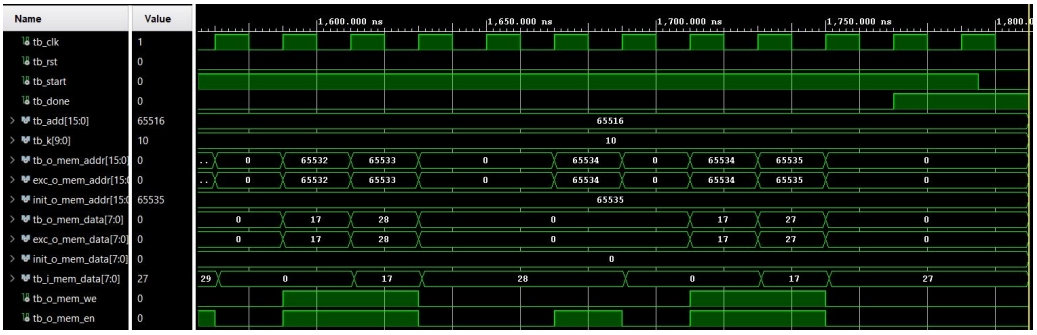


Figure 4.2: Waveform Post-Synthesis Functional Simulation con i_add=65516 e i_k=10.

## 4.7. Test Bench che richiede elaborazioni successive

Come richiesto dalla specifica, il modulo progettato è in grado di elaborare più sequenze consecutive anche senza ricevere un segnale di reset: ho quindi elaborato un Test Bench per verificare il corretto funzionamento del modulo in queste circostanze.

### 4.7.1. Elaborazioni successive con configurazione RAM

Un primo Test Bench testa il caso in cui prima di ogni elaborazione viene caricata una nuova sequenza in memoria, introducendo anche un'attesa di 1 us tra la prima e la seconda elaborazione.

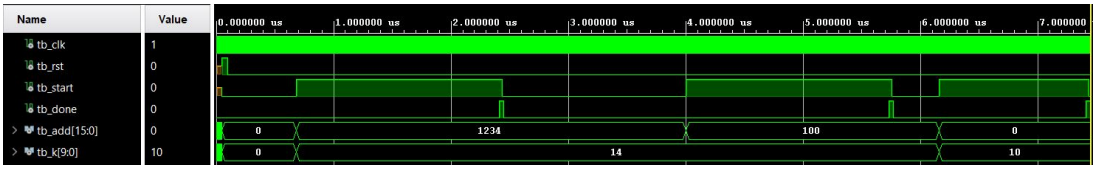


Figure 4.3: Waveform Post-Synthesis Functional Simulation con elaborazioni successive e configurazione RAM.

### 4.7.2. Elaborazioni successive senza configurazione RAM

Un secondo Test Bench testa il caso in cui la RAM viene configurata una sola volta all'inzio, ma non tra prima della seconda configurazione. In questo caso, la prima sequenza elaborata sarà diversa dalla seconda sequenza elaborata, poichè la sequenza di ingresso della seconda elaborazione è la sequenza elaborata risultante dalla prima elaborazione.
Sequenza da elaborare:

```
(128,0,64,0,0,0,0,0,0,0,0,0,0,0,
 100,0,1,0,0,0,5,0,23,0,200,0,0,0)
```

Prima sequenza elaborata:

```
(128,31,64,31,64,30,64,29,64,28,64,27,64,26,
 100,31,1,31,1,30,5,31,23,31,200,31,200,30)
```

Seconda sequenza elaborata:

```
(128,31,64,31,64,31,64,31,64,31,64,31,64,31,
100,31,1,31,1,31,5,31,23,31,200,31,200,31)
```

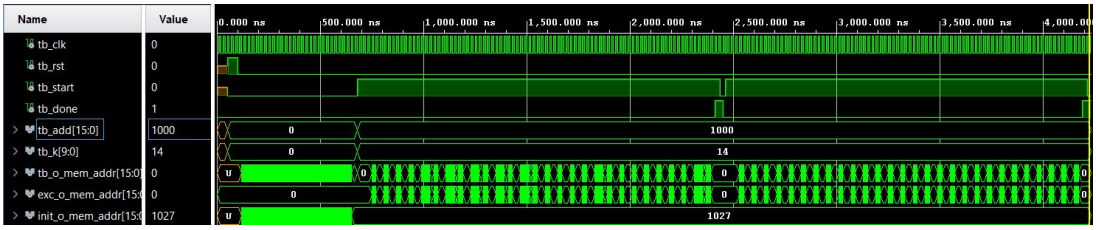


Figure 4.4: Waveform Post-Synthesis Functional Simulation con elaborazioni successive senza configurazione RAM.

## 4.8. Test Bench con i_k = 0

Ho realizzato un Test Bench per verificare il funzionamento del componente nel caso in cui `i_k` = 0, ovvero la sequenza è vuota. In questo caso mi aspetto che il modulo alzi subito il segnale `o_done`, segnalando di aver terminato l'elaborazione.

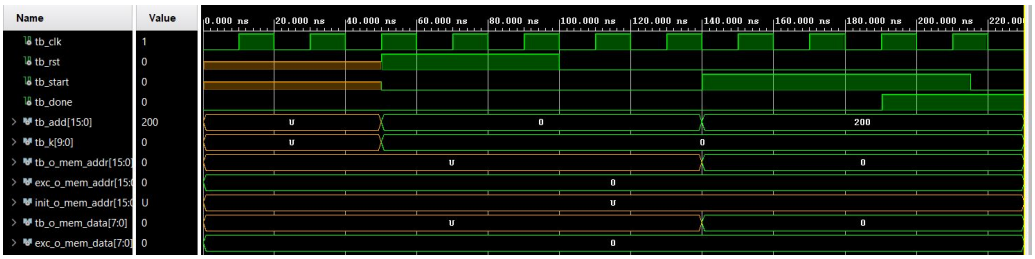


Figure 4.5: Waveform Post-Synthesis Functional Simulation con i_k = 0.

## 4.9. Test Bench con i_k massimo

Ho realizzato un TestBench per testare il corretto funzionamento del modulo nel caso in cui `i_k` assume valore massimo, ovvero `i_k` $= 2^{10} - 1 = 1023$. Per realizzare questo Test Bench, ho creato un piccolo script in Python, in grado di calcolare automaticamente `scenario_input` e `scenario_output` per una sequenza pseudocasuale di lunghezza desiderata.

## 4.10. Altri Test Bench

Infine ho realizzato altri Test Bench meno rilevanti, che, per brevità, elenco di seguito senza entrare nel dettaglio:

- Un Test Bench per ognuno dei cinque esempi presenti sul documento di specifica del progetto;
- Test Bench in cui il valore di credibilità presente in memoria prima dell'elaborazione è diverso da zero per alcune parole;
- Test Bench che invia un segnale di reset asincrono al modulo;
- Test Bench che richiede di elaborare più sequenze sugli stessi indirizzi di memoria;
- Test Bench con `i_k` $= 0$ e in cui `i_add` è l'utlimo indirizzo di memoria disponibile.

# 5 | Conclusione

In conclusione, mi reputo soddisfatto della progettazione e realizzazione del modulo richiesto. L'utilizzo di una tecnica di progettazione che prevede più componenti e più processi, secondo il paradigma *divide et impera*, mi ha permesso di scomporre la complessità del problema e di avere sempre il massimo controllo sul funzionamento del componente, facilitando anche la fase di debug. Il risultato ottenuto è un modulo che sfrutta una percentuale molto piccola delle risorse disponibili e che rispetta i constraint temporali con grande margine, come già evidenziato nell'Utilization Report (Sezione 3.1) e nel Timing Report (Sezione 3.2). Infine, il componente ottenuto è stato testato ed è risultato correttamente funzionante, sia in pre-sintesi che in post-sintesi, nei casi di normale utilizzo, oltre che in tutti i casi limite citati nel Capitolo 4.