POLITECNICO
MILANO 1863

*Corso di laurea triennale in Ingegneria Informatica*

# RELAZIONE PROVA FINALE DI RETI LOGICHE

Relazione di: **Foini Lorenzo**

Codice persona: **10828129**

Matricola: **985821**

**Anno Accademico 2023/2024**

# INDICE

CAPITOLO 1 ……. 2

1.1 INTRODUZIONE ……... 2

1.2 ESEMPIO …… 2

CAPITOLO 2 ……. 3

2.1 INIZIO E FINE DELL'ELABORAZIONE …… 3

CAPITOLO 3 ……. 4

3.1 ARCHITETTURA VISTA DALL'ESTERNO ……... 4

3.2 ARCHITETTURA INTERNA DEL MODULO ……. 6

3.2.1 FSM ……. 7

CAPITOLO 4 ……... 11

4.1 PROCESSO DI SINTESI …….. 11

4.1.1 ANALISI COMANDO "report_utilization" ……. 11

4.1.2 ANALISI COMANDO "report_timing" ……...… 13

4.2 SIMULAZIONE E TEST BENCHES……... 14

4.3 CONCLUSIONI TEST BENCHES ……..… 22

CAPITOLO 5 ……... 23

5.1 CONCLUSIONI …… 23

# CAPITOLO 1

## 1.1 INTRODUZIONE

La “Prova Finale di Reti logiche” dell’anno accademico 2023/2024 richiede la realizzazione di un modulo hardware, descritto in VHDL e utilizzando Vivado, che interagisca con una memoria e che si comporti come di seguito elencato.

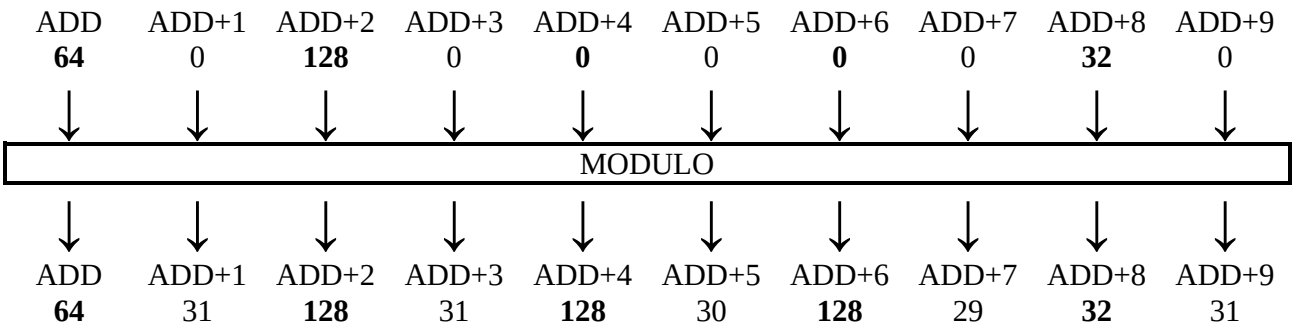
Il componente ha il compito di leggere ed elaborare un messaggio in input contenente una sequenza di parole, in numero K, con valori compresi tra 0 e 255, precisando che il valore 0 è interpretato come “valore non specificato”. Partendo dal primo indirizzo della sequenza, detto ADD, ogni parola viene memorizzata ogni 2 byte fino all’indirizzo ADD+2*(K-1). Eventuali byte mancanti, cioè contenenti il valore 0, verranno completati come segue.

Il modulo progettato realizza il completamento della sequenza letta, copiando l’ultimo valore letto all’indirizzo ADD+2*J (con J $\epsilon$ {0, 1, …, K-1}), se quest’ultimo è diverso da 0, oppure sostituendo lo 0 con l’ultimo valore valido letto precedentemente. Viene poi aggiunto un valore di credibilità all’indirizzo ADD+2*J+1. Tale valore è compreso tra 0 e 31, assumendo il massimo valore ogni qual volta che al byte precedente è stato letto un valore valido. Se invece dovesse essere letto il valore 0 al byte precedente, allora il livello di credibilità viene decrementato di uno rispetto a quello presente a due byte precedenti, non potendo però assumere valori negativi. Alla successiva lettura di un valore valido, il livello di credibilità viene reinizializzato a 31. Nel caso in cui la sequenza inizi con uno 0, allora tale valore resta inalterato, impostando la credibilità a 0. Ciò si ripete fino a quando non viene letto un valore valido nella sequenza.

## 1.2 ESEMPIO

Nel seguente esempio viene mostrata una sequenza in input con K=5 parole (in grassetto nell’esempio), con la prima parola memorizzata all’indirizzo ADD.

Viene poi illustrato come il modulo trasforma la sequenza in quella desiderata, inserendo l’ultimo valore valido al posto degli zeri e i valori di credibilità.



# CAPITOLO 2

## 2.1 INIZIO E FINE DELL'ELABORAZIONE

Per spiegare al meglio quando e in che circostanze inizia e finisce l'elaborazione è necessario introdurre i nomi e il funzionamento di alcuni segnali, i quali verranno illustrati in dettaglio all'interno del capitolo 3.

Il componente è dotato di un segnale di reset asincrono e unico per tutto il sistema e di un segnale di clock, anch'esso unico, ma sincrono. Inoltre, sono presenti altri tre ingressi primari: i_start (1 bit), i_add (16 bit) e i_k (10 bit), indicanti rispettivamente l'inizio dell'elaborazione, l'indirizzo in cui si trova la prima parola della sequenza e la lunghezza della sequenza. Infine, il segnale di uscita o_done (1 bit) segnala la fine dell'elaborazione.

Per il corretto funzionamento del modulo, all'inizio di tutti i test benches, illustrati in dettaglio nel capitolo 4, deve essere dato il reset del componente così da poterlo inizializzare e assegnare o_done = 0. Prima del reset il comportamento del modulo non è specificato.

Successivamente, il reset va a 0 e l'elaborazione inizia quando il segnale di start assume il valore 1, e resta alto fino a quando l'elaborazione non è completata. Tale condizione si verifica quando la scrittura in memoria della sequenza corretta è terminata e il modulo porta il segnale di uscita o_done a 1.

Quando inizia l'elaborazione, i segnali i_add e i_k vengono utilizzati per indicare l'indirizzo da cui parte la sequenza e la sua lunghezza.

Una volta che o_done è alzato a 1, il segnale i_start viene portato a 0, segnalando la fine dell'elaborazione. Dopo questa transizione di i_start, il modulo porta o_done a 0.

Terminata anche questa operazione, una nuova sequenza può essere elaborata quando i_start torna alto. In questo caso non è necessario resettare il componente. Figura 1 mostra come funziona questa sequenza di transizioni.

Qualora si verifichi un reset in un momento qualsiasi dell'elaborazione, il modulo viene reinizializzato e una successiva elaborazione può iniziare quando i_start = 1.

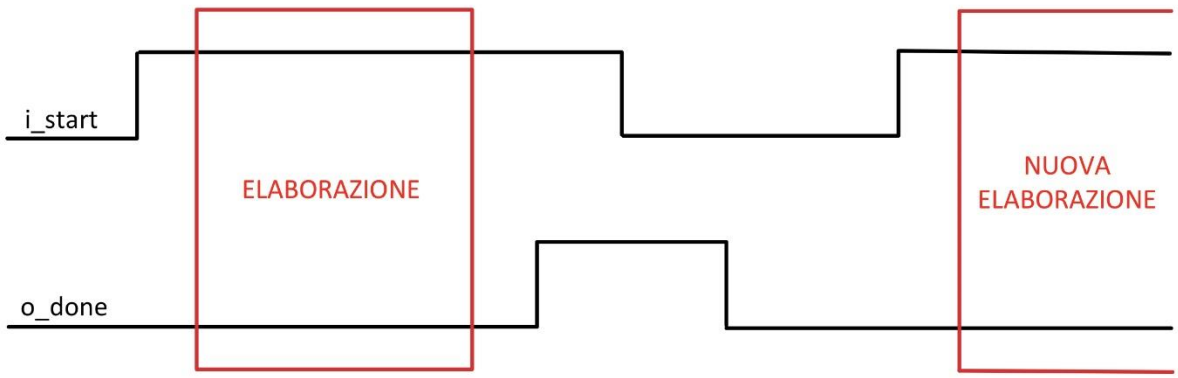



*Figura 1: Comportamento dei segnali di start e done.*

# CAPITOLO 3

## 3.1 ARCHITETTURA VISTA DALL'ESTERNO

Per la corretta simulazione dei test benches, che verranno analizzati in dettaglio all'interno del capitolo 4, il nome del modulo è il seguente: "project_reti_logiche".

Inoltre, sempre per la stessa convezione sopra citata, il modulo contiene un'unica architettura per l'unica entità di cui è composto: una FSM (Finite State Machine).

Il modulo è caratterizzato da sei segnali in input e cinque segnali in output. Essi sono responsabili della comunicazione con la memoria esterna, e di segnalare l'inizio e la fine dell'elaborazione delle sequenze.

I segnali di input sono i seguenti e ognuno ha una funzione esatta:

- i_rst:
  - Segnale da 1 bit;
  - Rappresenta il reset del modulo;
  - Ha il compito di re-inizializzare il modulo.
- i_clk:
  - Segnale da 1 bit;
  - Rappresenta il clock del modulo generato dai test benches;
  - Durata di 20 ns con duty cycle del 50%.
- i_start:
  - Segnale da 1 bit;
  - Indica l'inizio dell'elaborazione della sequenza quando questo viene alzato a 1.
- i_k:
  - Segnale da 10 bit;
  - Indica la lunghezza della sequenza di parole generata dai test benches.
- i_add:
  - Segnale da 16 bit;
  - Indica l'indirizzo da cui parte la sequenza.
- i_mem_data:
  - Segnale da 8 bit;
  - Rappresenta il valore numerico che arriva dalla memoria dopo una richiesta di lettura.

Per i segnali di output:

- o_done:
  - Segnale da 1 bit;
  - Comunica all'esterno la fine dell'elaborazione quando questo viene alzato a 1.
- o_mem_addr:
  - Segnale da 16 bit;
  - Indica l'indirizzo in cui si vuole fare una lettura/scrittura da/in memoria.

- o_mem_data:
  - Segnale da 8 bit;
  - Rappresenta il valore numerico che viene scritto in memoria all'indirizzo contenuto in o_mem_data dopo una richiesta di scrittura.
- o_mem_en:
  - Segnale da 1 bit;
  - Deve essere alzato a 1 quando si vuole interagire con la memoria, sia in lettura sia in scrittura.
- o_mem_we:
  - Segnale da 1 bit;
  - Deve essere alzato a 1 quando si vuole effettuare una scrittura in memoria, altrimenti vale 0 quando si effettua una lettura.

Tutti i segnali sono sincroni, ad eccezione del reset che è asincrono, e sono interpretati sul fronte di salita del clock.

Ricordando che la memoria è già istanziata all'interno dei test benches, Figura 2 mostra come il progetto è visto dall'esterno.

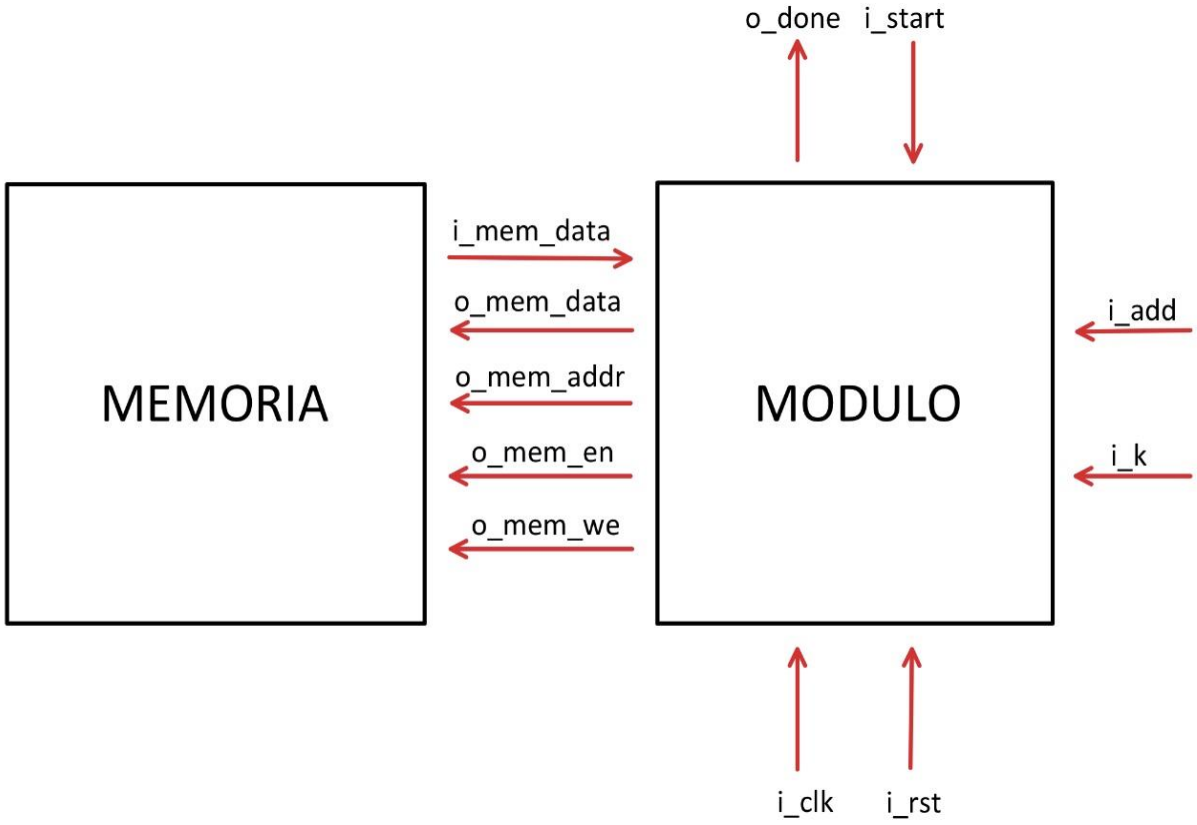



*Figura 2: Vista esterna del progetto.*

## 3.2 ARCHITETTURA INTERNA DEL MODULO

Come già precisato prima, il modulo realizzato ha come unico componente una FSM di Moore, la quale ha il compito di gestire opportunamente tutte le fasi di inizializzazione dei segnali, gestione dell'elaborazione e comunicazione con la memoria.

È stata scelta questa architettura perché la FSM è appunto in grado di gestire tutte le fasi ottimamente, senza generare latch e rispettando il vincolo sullo slack (si consiglia di leggere il capitolo 4 per i risultati sperimentali). Grazie ad essa è stato possibile realizzare un modulo con un'architettura semplice e allo stesso tempo un codice VHDL chiaro e preciso.

Graficamente, Figura 3 mostra come è realizzato internamente il modulo.

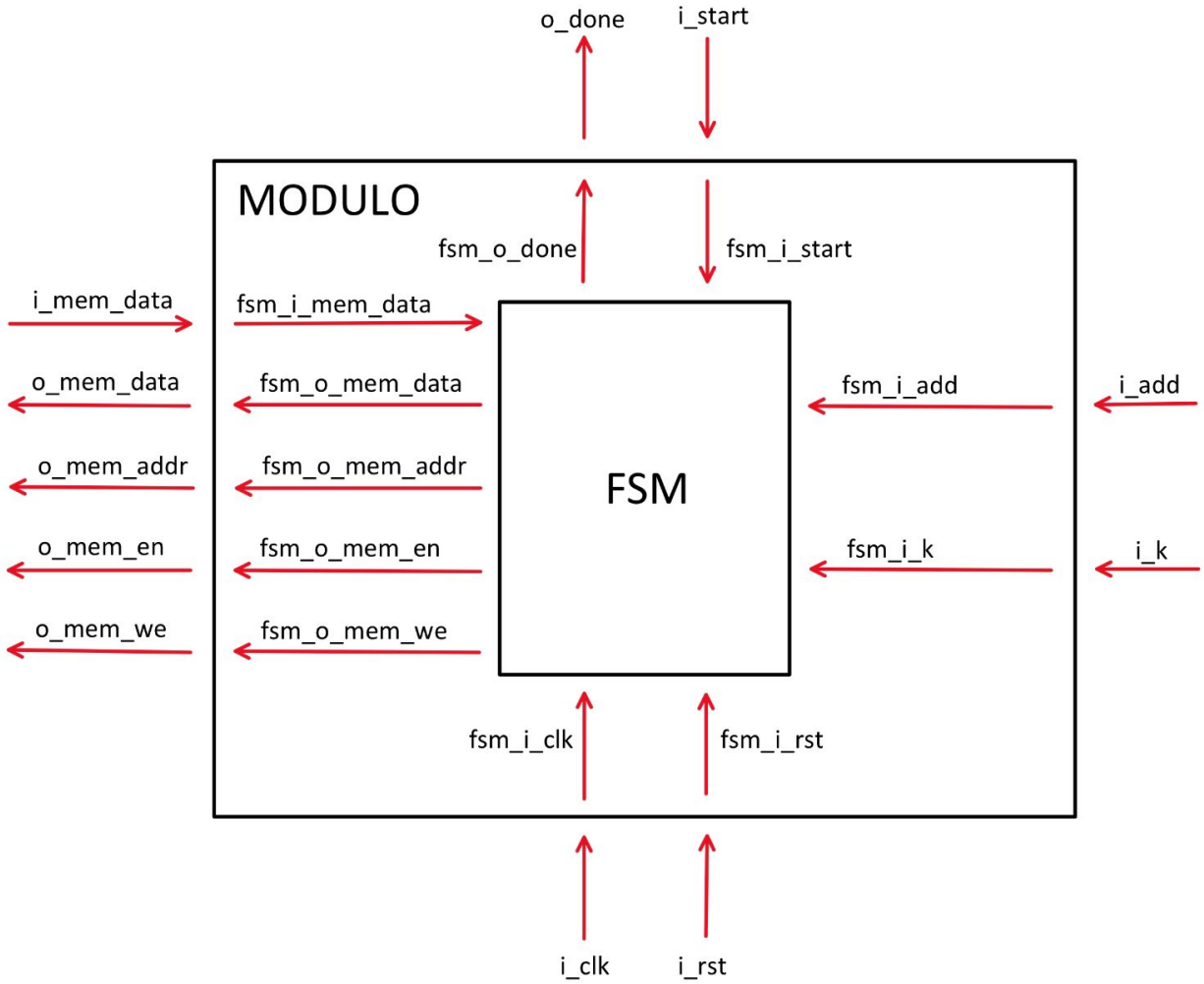



*Figura 3: Vista interna del modulo.*

Entrando nei particolari, viene effettuata una MAP tra i segnali del modulo e quelli della FSM, permettendo così alla FSM di leggere i segnali in input e modificare quelli in output in base alla fase in cui si trova l'elaborazione.

È necessario ora analizzare in dettaglio quali sono gli stati della FSM e cosa accade all'interno di ciascuno di essi.

### 3.2.1 FSM

All'interno della FSM viene dichiarato un segnale chiamato CURRENT_STATE, rappresentante lo stato attuale in cui essa si trova, e quattro variabili:

- last_value:
  - Variabile da 8 bit;
  - Rappresenta l'ultimo valore valido, ossia l'ultima parola diversa da 0, letta dalla sequenza.
- credibility:
  - Variabile da 8 bit;
  - Siccome assume valori compresi tra 0 e 31 basterebbe utilizzare 5 bit, ma così facendo non si riuscirebbe a scrivere nella sequenza elaborata poiché contiene solamente valori rappresentati su 8 bit.
  - Rappresenta l'attuale valore che verrà inserito nel byte successivo a quello contenente la parola.
- curr_addr:
  - Variabile da 16 bit;
  - Rappresenta l'attuale indirizzo che passeremo alla memoria per poter leggere / scrivere.
- done:
  - Variabile da 1 bit;
  - Viene utilizzata all'interno dello stato CHECK_ELABORATION della FSM per poter controllare se l'elaborazione è terminata oppure no;
  - Questa variabile è necessaria perché in un process non è possibile leggere i valori dei segnali in output, tra cui o_done.

Vengono utilizzate le variabili perché la modifica dei loro valori è immediata. Ciò permette di utilizzare la variabile subito dopo la modifica, non dovendo quindi aspettare il successivo rising edge del clock per poter effettivamente vedere l'aggiornamento, come invece accade per i segnali.

I possibili stati della FSM sono i seguenti: INITIAL, CHECK_ELABORATION, READ_VALUE_MEM, CHECK_VALUE_MEM, WRITE_VALUE_MEM, WRITE_CREDIBILITY_MEM, INCREASE_ADDRESS, FINISH_ELABORATION. Successivamente verrà illustrato cosa accade all'interno di ciascuno.

Le operazioni effettuate all'interno della FSM sono definite in un unico process, che sarà quindi responsabile sia della gestione dello stato della FSM sia del settaggio dei valori dei segnali in output e delle variabili che esso introduce. Tale process contiene nella sensitivity list i segnali fsm_i_rst e fsm_i_clk, rappresentanti rispettivamente il reset e il clock. Così facendo, il process viene risvegliato ogni qualvolta che uno dei due segnali cambia.

Poiché il reset è asincrono, e può quindi verificarsi in qualsiasi momento, è necessario controllare quale valore assume ogni qualvolta che il process si risveglia. Nel caso in cui il reset sia

alto vengono re-inizializzati tutti i segnali in output della FSM, re-inizializzate tutte le variabili, tranne curr_addr, e assegnato INITIAL a CURRENT_STATE.

Successivamente, quando reset va a 0 e si verifica un rising edge del clock, allora inizia l'effettiva transizione di stato della FSM, che viene ora illustrata in dettaglio in Figura 4.

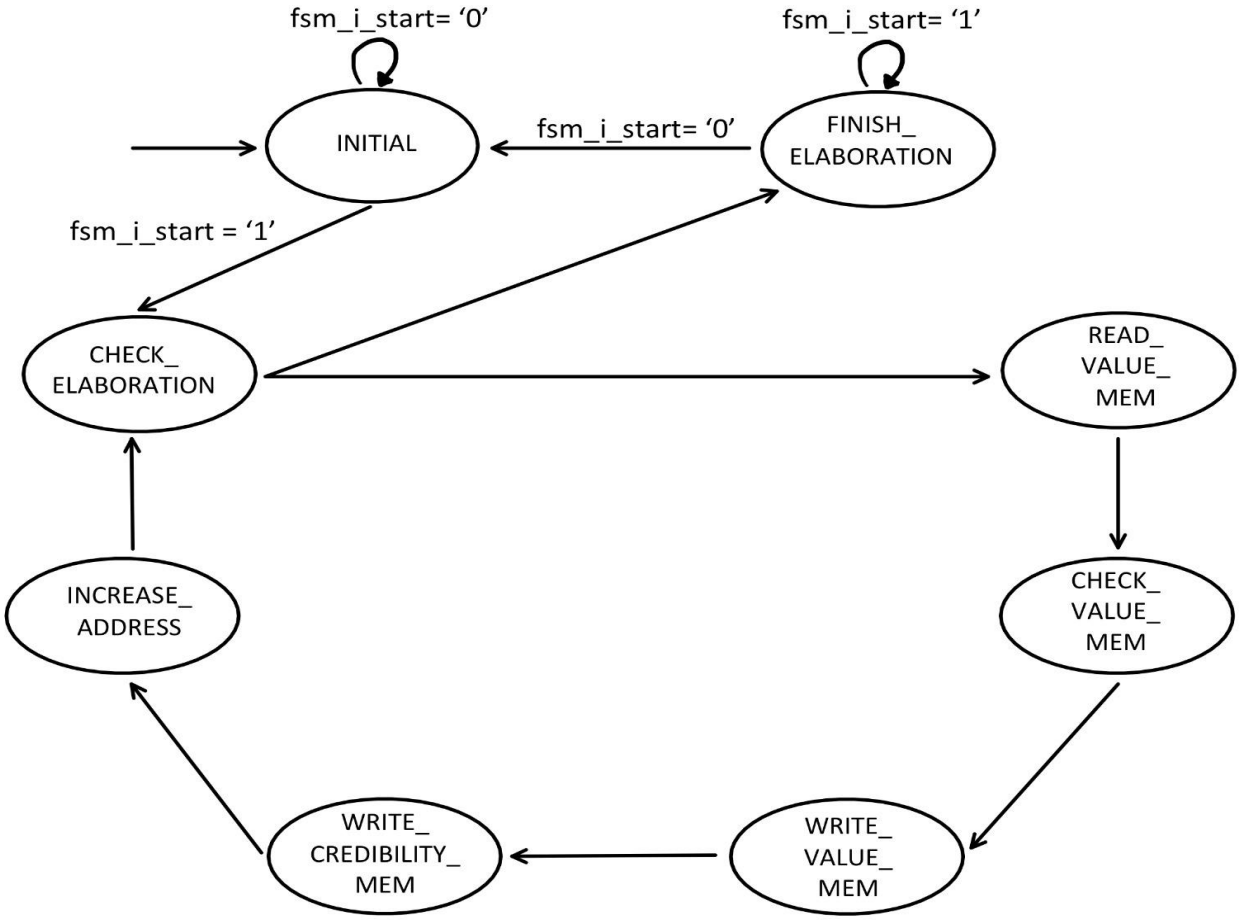



*Figura 4: Transizione di stato della FSM.*

1. INITIAL:

Rappresenta lo stato iniziale della FSM e si rimane in questo stato fino a quando non viene dato il segnale di start della conversione (fsm_i_start = 1). Quando ciò accade, la variabile curr_addr viene "inizializzata" a fsm_i_add. Successivamente si verifica la transizione di stato a CHECK_ELABORATION, che sarà visibile al successivo rising edge del clock.

2. CHECK_ELABORATION:

Questo stato ha il compito di verificare se l'elaborazione della sequenza è terminata oppure è ancora in corso, controllando il valore di fsm_i_k e curr_addr.

L'elaborazione è terminata se: 1) la sequenza contiene 0 parole, rappresentabile da fsm_i_k=0, oppure 2) l'attuale indirizzo che si sta analizzando non fa parte degli indirizzi in cui sono salvate le parole, rappresentabile con curr_addr > fsm_i_add + 2* (fsm_i_k - 1).

Quando si verifica una di queste due condizioni, allora alla variabile done e al segnale fsm_o_done viene assegnato il valore 1, segnalando quindi la fine dell'elaborazione e passando allo stato FINISH_ELABORATION.

In caso contrario, la FSM si prepara alla lettura all'indirizzo curr_addr dalla memoria, alzando quindi a 1 il segnale fsm_o_mem_en e assegnando a fsm_o_mem_addr il valore contenuto in curr_addr. Così facendo, la modifica di questi due segnali è visibile al prossimo rising edge del clock. Infine, si passa allo stato READ_VALUE_MEM.

3. READ_VALUE_MEM:

All'inizio di questo stato sono visibili le modifiche dei due segnali responsabili della lettura da memoria specificati precedentemente. Si attende quindi un ciclo di clock affinché avvenga l'effettiva lettura. Avviene ora la transizione allo stato CHECK_VALUE_MEM.

4. CHECK_VALUE_MEM:

Questo stato controlla il valore letto da memoria disponibile all'interno del segnale fsm_i_mem_data. In base a tale valore, vengono settate le variabili last_value e credibility, che verranno utilizzate per scrivere in memoria i valori desiderati.

Si possono verificare tre casi:

1. fsm_i_mem_data ≠ 0
   ⇨ Il valore letto da memoria è valido e assegnato a last_value.
2. fsm_i_mem_data = 0 e credibility ≠ 0
   ⇨ Il valore letto da memoria non è valido e la credibilità è positiva. A quest'ultima viene assegnato lo stesso valore che conteneva meno 1.
3. fsm_i_mem data = 0 e credibility = 0
   ⇨ Il valore letto da memoria non è valido e la credibilità vale già zero.
   ⇨ Alle variabili non viene effettuata alcuna modifica.

Dopo aver gestito questa casistica, a fsm_o_mem_data viene assegnato il valore di last_value e viene alzato fsm_o_mem_we, così da poter scrivere in memoria all'indirizzo corrente il valore elaborato della parola.

Infine, avviene la transizione allo stato WRITE_VALUE_MEM.

5. WRITE_VALUE_MEM:

All'inizio di questo stato avviene la scrittura in memoria della parola, dopo che questa è stata elaborata in base al suo valore nella sequenza in input.

Si procede poi assegnando alla variabile curr_addr il suo valore successivo, ricordando che tale operazione è immediata, e aggiornando i segnali fsm_o_mem_addr a tale indirizzo e fsm_o_mem_data al valore della credibilità.

Infine, avviene la transizione allo stato WRITE_CREDIBILITY_MEM.

6. WRITE_CREDIBILITY_MEM:

Analogamente a quanto accade nello stato precedente, viene scritto in memoria il valore di credibilità all'indirizzo specificato.

Siccome al prossimo rising edge del clock non è più necessario interagire con la memoria, si può procedere con l'abbassamento dei segnali di enable responsabili di tali operazioni.

Si giunge infine alla transizione verso lo stato successivo.

7. INCREASE_ADDRESS:

L'unica operazione svolta è l'assegnamento dell'indirizzo successivo alla variabile curr_addr, così da poterlo analizzare all'interno dello stato CHECK_ELABORATION, stato in cui la FSM si trova dopo un ciclo di clock.

8. FINISH_ELABORATION:

Ricordando che la FSM giunge in questo stato solo quando ha terminato l'elaborazione della sequenza in input, è possibile ora documentare qual è l'operazione successiva.

Al rising edge che porta a questo stato corrisponde anche l'assegnamento del valore 1 al segnale di uscita fsm_o_done, condizione necessaria affinché si possa abbassare il segnale di input i_start. Le operazioni successive avvengo solamente quando quest'ultimo va a 0, altrimenti si rimane in questo stato.

Vengono ora re-inizializzate le variabili e i segnali di uscita, permettendo al modulo di poter elaborare una nuova sequenza, in caso essa ci sia.

Infine, avviene la transizione allo stato INITIAL, iniziando quindi un nuovo ciclo.

NOTE:

1. Tutte le transizioni allo stato prossimo avvengo al successivo rising edge del clock;
2. Per gli stati in cui non sono state specificate le condizioni per il passaggio allo stato prossimo, la transizione avviene dopo un ciclo di clock.

# CAPITOLO 4

## 4.1 PROCESSO DI SINTESI

Per iniziare, è necessario specificare la FPGA utilizzata in questo progetto. È stata scelta la Artix-7 FPGA xc7a200tfbg484-1 come riferimento, ma qualunque altra FPGA permette la corretta simulazione del progetto.

È necessario ora definire che cosa si intende per sintesi di un modulo hardware: questo processo traduce l'astrazione del comportamento del circuito in una realizzazione fisica e implementabile su un dispositivo, in questo caso sulla FPGA target, qualora le simulazioni siano soddisfacenti.

Una corretta sintesi del componente è cruciale, poiché essa ne determina le prestazioni, il consumo e l'area occupata sulla FPGA. È quindi fondamentale che il codice venga implementato in modo tale da rispettare questi vincoli di efficienza.

Questo processo coinvolge la traduzione delle specifiche in porte logiche, elementi di memoria, connessioni e timing necessari per implementare il comportamento desiderato.

Dopo questa breve introduzione, vengono ora analizzati i report del tool di sintesi e dei dati più rilevanti contenuti in essi. Tali report sono stato generati dalla TCL console di Vivado usando i comandi "report_utilization" e "report_timing".

### 4.1.1 ANALISI COMANDO "report_utilization"

Il comando "report_utilization" è utilizzato per generare un report che fornisce informazioni sull'utilizzo delle risorse di logica programmabile (tra cui LUT, registri, I/O) all'interno del modulo implementato sulla FPGA. Questo report è utile per valutare quanto efficacemente il modulo sfrutta le risorse disponibili sulla FPGA e per identificare eventuali aree di ottimizzazione o problematiche di utilizzo delle risorse.

Vengono ora mostrati e analizzati i dati riguardanti la slice logic, i registri utilizzati in base al tipo e l'utilizzo delle primitive, rappresentati rispettivamente in Figura 5, 6 e 7:

- In Figura 5 si fornisce un riepilogo dell'utilizzo delle risorse di logica programmabile all'interno della FPGA. Si può notare che su un totale di 134600 LUT disponibili, solo 136 sono utilizzate (0.10%). Analogamente, su 269200 registri disponibili, solo 62 sono utilizzati (0.02%). È possibile quindi affermare che il modulo occupa solo una piccola frazione delle risorse disponibili.
  Inoltre, si evidenzia che tutti i registri utilizzati sono configurati come flip-flop, senza nessun latch. Questo è di notevole importanza perché i latch possono causare problemi di temporizzazione e di stabilità. Il fatto che non siano presenti latch indica che il modulo ha una progettazione ben strutturata e priva di potenziali errori di sincronizzazione.

- In Figura 6 si fornisce una suddivisione dettagliata dei registri utilizzati, classificati per tipo. È importante notare che tutti i registri sono utilizzati con l'abilitazione del segnale di clock, con la maggior parte di essi utilizzati per la sincronizzazione sincrona e solo un numero limitato per quella asincrona.

- In Figura 7 si forniscono informazioni dettagliate sull'utilizzo delle primitive nel modulo, come flip-flop e LUT. Queste primitive sono fondamentali per la realizzazione del modulo e la loro corretta configurazione è essenziale per garantire il corretto funzionamento.

1. Slice Logic

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| Slice LUTs* | 136 | 0 | 134600 | 0.10 |
| LUT as Logic | 136 | 0 | 134600 | 0.10 |
| LUT as Memory | 0 | 0 | 46200 | 0.00 |
| Slice Registers | 62 | 0 | 269200 | 0.02 |
| Register as Flip Flop | 62 | 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 5: Slice Logic.*

1.1 Summary of Registers by Type

| Total | Clock Enable | Synchronous | Asynchronous |
|---|---|---|---|
| 0 | _ | - | - |
| 0 | _ | - | Set |
| 0 | _ | - | Reset |
| 0 | _ | Set | - |
| 0 | _ | Reset | - |
| 0 | Yes | - | - |
| 0 | Yes | - | Set |
| 46 | Yes | - | Reset |
| 0 | Yes | Set | - |
| 16 | Yes | Reset | - |

*Figura 6: Registri utilizzati in base al tipo.*

7. Primitives

| Ref Name | Used | Functional Category |
|---|---|---|
| FDCE | 46 | Flop & Latch |
| LUT1 | 39 | LUT |
| IBUF | 37 | IO |
| LUT6 | 36 | LUT |
| OBUF | 27 | IO |
| LUT4 | 22 | LUT |
| LUT3 | 20 | LUT |
| LUT2 | 19 | LUT |
| CARRY4 | 18 | CarryLogic |
| FDRE | 16 | Flop & Latch |
| LUT5 | 14 | LUT |
| BUFG | 1 | Clock |

*Figura 7: Primitive.*

### 4.1.2 ANALISI COMANDO “report_timing”

Il comando "report_timing" è utilizzato per generare un report che fornisce informazioni dettagliate sulle prestazioni temporali del modulo implementato sulla FPGA.

Esso fornisce una valutazione delle prestazioni rispetto ai vincoli temporali specificati, come il periodo di clock richiesto. Queste informazioni sono fondamentali per garantire che il modulo operi correttamente e che soddisfi i requisiti di timing prefissati.

Vengono ora analizzati i dati più rilevanti, rappresentati anche in Figura 8:

- Slack (MET):
  - Indica la differenza temporale tra il tempo richiesto per completare il percorso più critico e il tempo effettivamente impiegato;
  - Uno slack positivo indica che il percorso critico sta operando entro i limiti temporali previsti. Uno slack negativo indica che il tempo richiesto per il percorso supera il tempo disponibile;
  - In questo progetto il valore di slack è 14,175 ns, indicando che il percorso più critico ha un ritardo inferiore al periodo di clock richiesto (20 ns), soddisfacendo quindi il requisito temporale imposto. Figura 9 mostra il calcolo effettivo dello slack.
- Requirement:
  - Indica il tempo massimo consentito per completare le operazioni sul percorso senza violare i vincoli temporali.
- Data Path Delay:
  - Indica il ritardo totale del percorso ed è pari a 5,443 ns;
  - Si divide in ritardo logico (3,265 ns) e ritardo di routing (2,178 ns).
- Logic Levels:
  - Fornisce una visione della complessità del percorso e delle risorse logiche coinvolte;
  - Il percorso coinvolge nove livelli logici, con una combinazione di sei CARRY4, una LUT1, una LUT2 e una LUT6.
- Clock Path Skew:
  - Indica lo sbilanciamento temporale del segnale di clock tra destinazione e sorgente ed è pari a -0,145 ns.
- Clock Uncertainty:
  - Fornisce una stima dell’incertezza associata al segnale di clock utilizzato nel percorso.

```
Slack (MET) :             14.175ns  (required time - arrival time)
  Source:                 fsm_instance/curr_addr_reg[0]/C
                            (rising edge-triggered cell FDRE clocked by clock  {rise@0.000ns fall@10.000ns period=20.000ns})
  Destination:            fsm_instance/fsm_o_mem_addr_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:        5.443ns  (logic 3.265ns (59.985%)  route 2.178ns (40.015%))
  Logic Levels:           9  (CARRY4=6 LUT1=1 LUT2=1 LUT6=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
```

*Figura 8: Output del comando "report_timing".*

```
required time                         22.041
arrival time                          -7.867
-------------------------------------------------
slack                                 14.175
```

*Figura 9: Calcolo dello slack.*

## 4.2 SIMULAZIONE E TEST BENCHES

Una volta completata la sintesi, il modulo può essere ulteriormente validato attraverso simulazioni. Per farlo, è necessario definire diversi test benches per simulare tutte le possibili casistiche che si possono verificare. Occorre anche controllare che le sequenze vengano elaborate correttamente dal modulo, rispettando i vincoli imposti dalla specifica.

Prima di studiare in dettaglio i test benches, è doveroso documentare le seguenti note:

1. I test benches includono la memoria in cui vengono salvate e riscritte le sequenze;
2. Come verrà mostrato successivamente, il modulo è correttamente simulabile in pre-sintesi e post-sintesi, utilizzando la “behavioral simulation” e la “post-synthesis functional simulation”. Non verrà valutata la “post-synthesis timing simulation”. Si precisa che i risultati mostrati di seguito sono estratti dalla “post-synthesis functional simulation” del modulo, poiché una corretta simulazione post-sintesi garantisce che il modulo venga simulato correttamente anche usando la “behavioral simulation”;
3. Tutti i valori visibili nelle prossime Figure sono in base 16.

In totale il modulo è stato simulato su sette diversi test benches, ognuno dei quali, escluso il primo, testa un diverso “corner case” individuato, così da poter infine concludere la corretta implementazione del modulo.

1° TEST BENCH: Esempio fornito

Come già anticipato, questo test bench non mira a verificare il corretto funzionamento del modulo in un “corner case”, ma piuttosto la corretta elaborazione di una sequenza generica e il corretto assegnamento dei segnali in input e in output.

In particolare, esso controlla e interrompe la simulazione segnalando un errore se si dovesse verificare una delle seguenti casistiche:

- o_done = 1 durante il reset del modulo;
- o_done = 1 dopo il reset del modulo, ma prima che il segnale i_start vada a 1;
- o_mem_en = 1 dopo che si è alzato a 1 il segnale o_done;
- Date le corrette sequenze in input e in output, il modulo scrive in memoria un valore o un livello di credibilità diverso da quello corretto.

Tutte queste casistiche non devono verificarsi durante la simulazione, in quanto non consoni con la specifica richiesta.

Vengono ora illustrati i passaggi chiave della simulazione in Figura 10, 11 e 12.

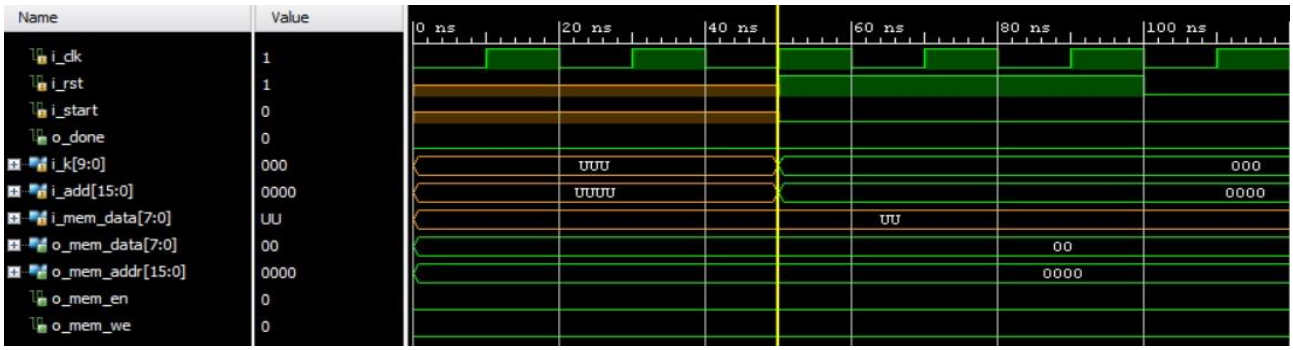



*Figura 10: Reset del modulo.*

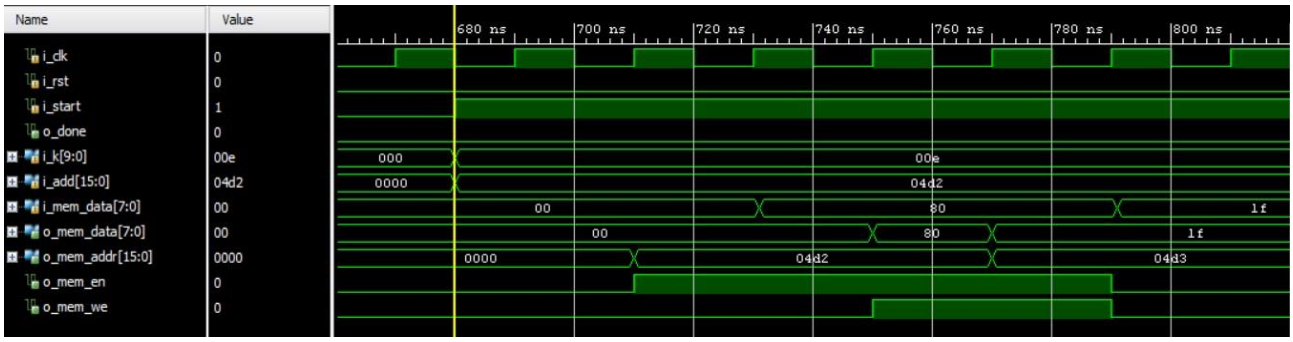



*Figura 11: Inizio elaborazione della sequenza.*

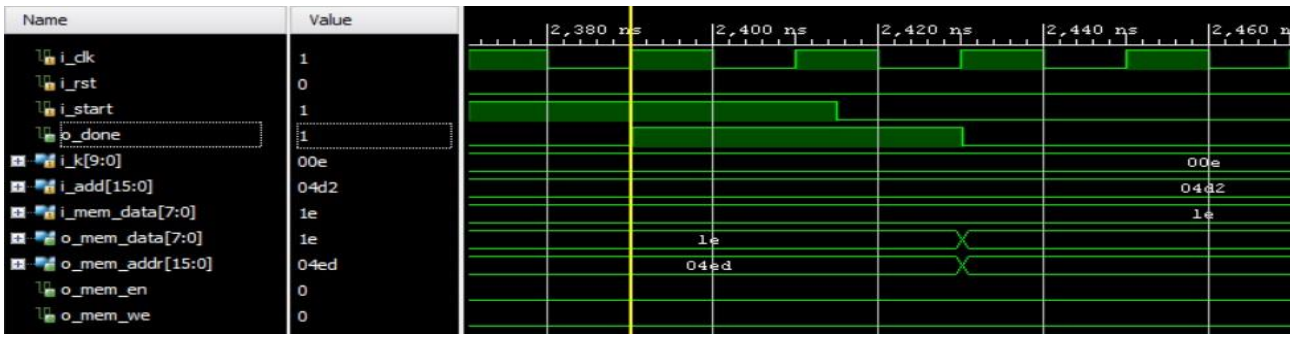



*Figura 12: Elaborazione completata.*

All'inizio della simulazione, il modulo viene correttamente resettato e si attende ora che inizi l'elaborazione. Ciò accade quando il segnale di start viene portato a 1 e inizia quindi l'elaborazione della sequenza, come si può osservare in Figura 11.

Viene poi illustrata la terminazione dell'elaborazione con l'alzamento del segnale di done e il successivo abbassamento del segnale di start, attendendo ora una nuova sequenza se definita.

Viene poi mostrato in Figura 13 il risultato della simulazione, confermando che il modulo elabora correttamente la sequenza rispettando le specifiche e i vincoli.

```
launch_simulation: Time (s): cpu = 00:00:59 ; elapsed = 00:01:00 . Memory (MB): peak = 1678.746 ; gain = 896.363
restart
INFO: [Simtcl 6-17] Simulation restarted
run all
Failure: Simulation Ended! TEST PASSATO (EXAMPLE)
```

*Figura 13: Simulazione finita, test passato.*

2° TEST BENCH: i_k = 0

Potrebbe capire che la sequenza salvata in memoria sia composta da zero parole. Questo test bench verifica questo caso assegnando a i_k il valore 0.

Il reset del modulo avviene come in Figura 10, mentre l'inizio e la fine della elaborazione sono due fasi molto vicine temporalmente, come si può dedurre dalla Figura 14.

Questo accade perché, non appena la FSM si trova nello stato CHECK_ELABORATION, essa transita immediatamente allo stato FINISH_ELABORATION, senza passare attraverso alcuno stato che comunica con la memoria.

Ciò è dovuto al fatto che la condizione i_k = 0 è vera.

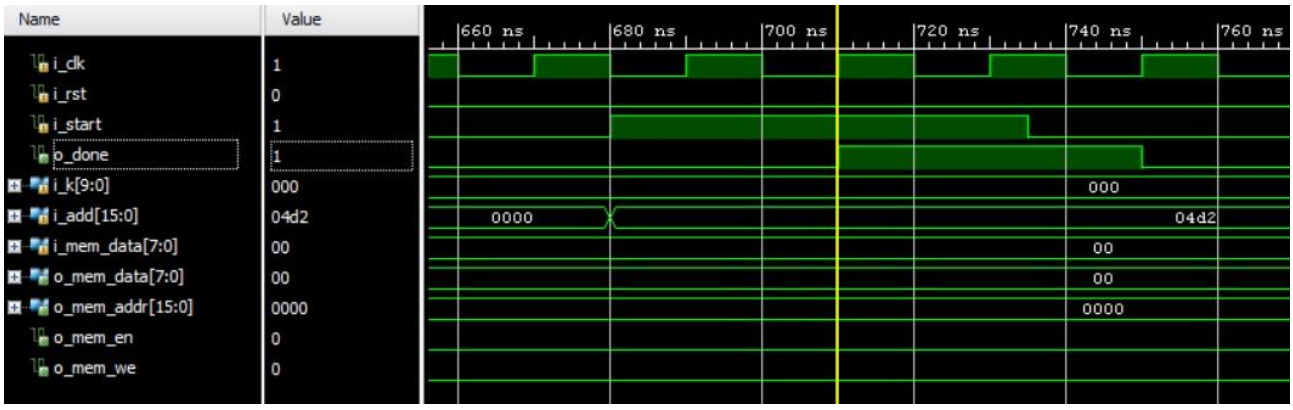



*Figura 14: Inizio e fine elaborazione nel caso in cui i_k sia uguale a 0.*

3° TEST BENCH: i_k = 1023

Il “corner case” opposto a quello precedentemente analizzato si ha quando i_k assume il valore massimo pari a 1023, essendo i_k un vettore da 10 bit. Questo test bench verifica questo caso assegnando a i_k tale valore.

Per generare una sequenza dotata di 1023 parole, è stato utilizzato il codice Python mostrato in Figura 15, il quale stampa la sequenza in input e quella in output dopo essere stata elaborata.

```python
from numpy import random

# Create the input list of random value between 0 and 255
length = 1023
input_list = list()
for i in range(length):
    input_list.append(random.randint(0,255)) # Random value
    input_list.append(0) # Credibility

# Define the correct output list:
output_list = list()
credibility = 0
last_valid_value = 0
for i in range(len(input_list)):
    if i % 2 == 0: # Read value from input_list
        if input_list[i] != 0:
            credibility = 31 # Set credibility level to 31 because I read a valid value
            last_valid_value = input_list[i] # Assign current element to the last valid value
        else:
            if credibility > 0:
                credibility -= 1 # Decrement credibility, if possible
        output_list.append(last_valid_value) # Append valid value to output list
    else: # Read credibility from input list
        output_list.append(credibility) # Append credibility level to output list

print(input_list)  # Display input sequence
print(output_list) # Display output sequence
```

*Figura 15: Codice Python per la generazione di una sequenza randomica di 1023 parole.*

Le tre fasi della simulazione del secondo test bench, si verificano anche durante la simulazione di questo terzo test, con una importante differenza temporale tra l’istante in cui i_start va a 1 e quello in cui o_done va a 1.

Infatti, nella seconda simulazione tale differenza è pari a 30 ns (710 ns - 680 ns), mentre per la terza è pari a 122790,1 ns (163830,1 ns - 41040 ns).

Ciò è ovviamente causato dalla differenza di lunghezza della sequenza letta.

4° TEST BENCH: La sequenza inizia con 0

Un altro “corner case” che necessita di verifica si ha quando la sequenza di parole inizia con uno 0. In questo caso il modulo deve scrivere in memoria il valore 0 e settare a 0 anche la credibilità al byte successivo. Questo vale fino a quando non viene letto un valore valido.

In Figura 16 e 17 viene mostrato come il modulo gestisce correttamente questa casistica, scrivendo in memoria il valore 0 fino a quando non viene letto un valore valido.

Si può infatti notare che i_mem_data e o_mem_data valgono 0 ad inizio elaborazione, e mantengono tale valore fino a quando il modulo legge un valore valido dalla memoria, in questo caso 58. Viene poi scritto 31 al byte successivo, impostando correttamente il valore di credibilità.

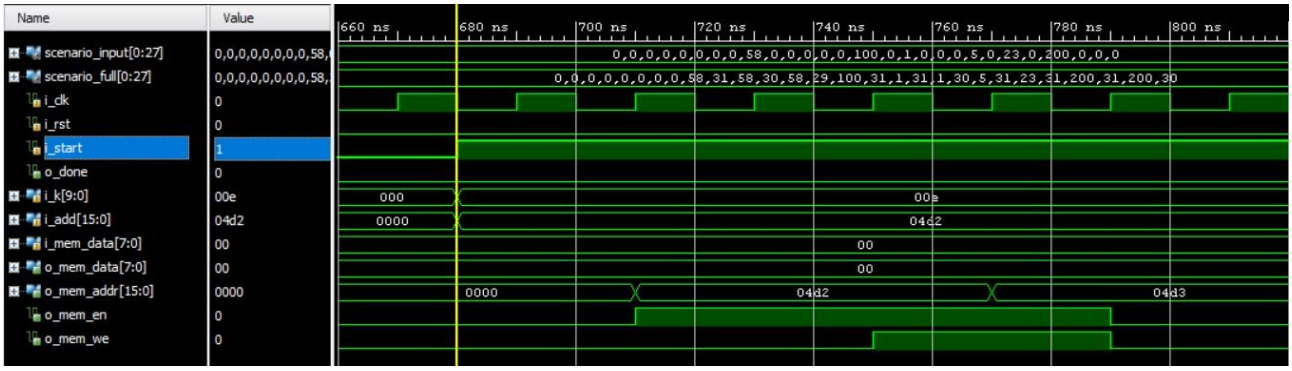



*Figura 16: Il modulo legge 0 dalla memoria e scrive correttamente 0.*

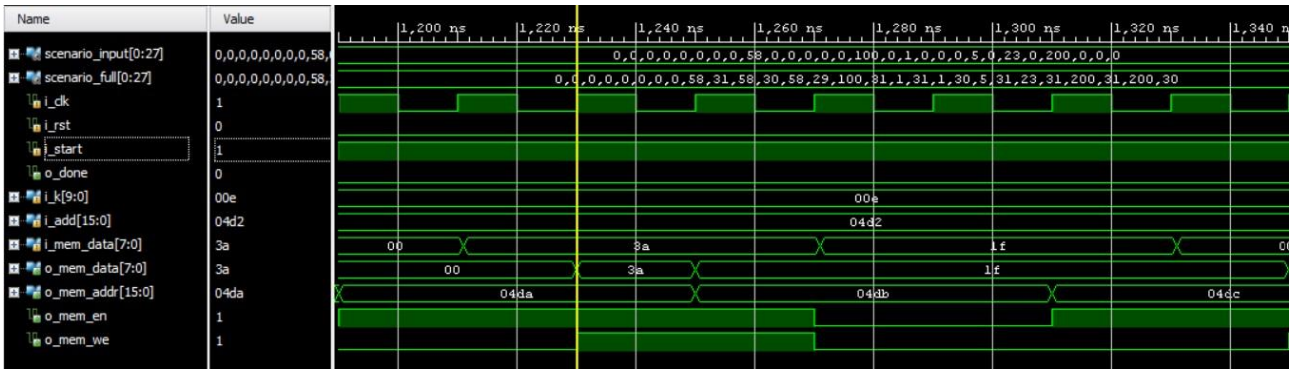



*Figura 17: Il modulo legge un valore valido ed elabora la sequenza correttamente.*

5° TEST BENCH: La sequenza contiene più di trentuno 0 consecutivi.

Come già spiegato nell'introduzione, il valore della credibilità parte da 31 quando viene letto un valore valido dalla sequenza, per poi essere decrementato di uno qualora venga letto uno 0. Potrebbe quindi essere fornita in input una sequenza di parole con più di trentuno 0 consecutivi, portando quindi la credibilità a 0 senza però assumere valori negativi.

È stato quindi definito un test bench con tale caratteristica, in particolare la sequenza è dotata di una sotto-sequenza con trentatré 0 consecutivi.

Figura 18 rappresenta il momento in cui viene letto il valore 100. Successivamente inizia la sequenza di zeri e la loro elaborazione.

In Figura 19, al trentunesimo 0 letto dalla memoria corrisponde la credibilità pari a 0. Per la successiva lettura, che contiene ancora il valore 0, viene correttamente scritto il valore 100, ma la credibilità vale ancora 0 e non assume un valore negativo. La stessa cosa avviene per l'ultimo 0 letto.

Infine, viene letto un valore valido e l'elaborazione procede correttamente, scrivendo tale valore in memoria e impostando la credibilità a 31 nel byte successivo, come mostrato in Figura 20.

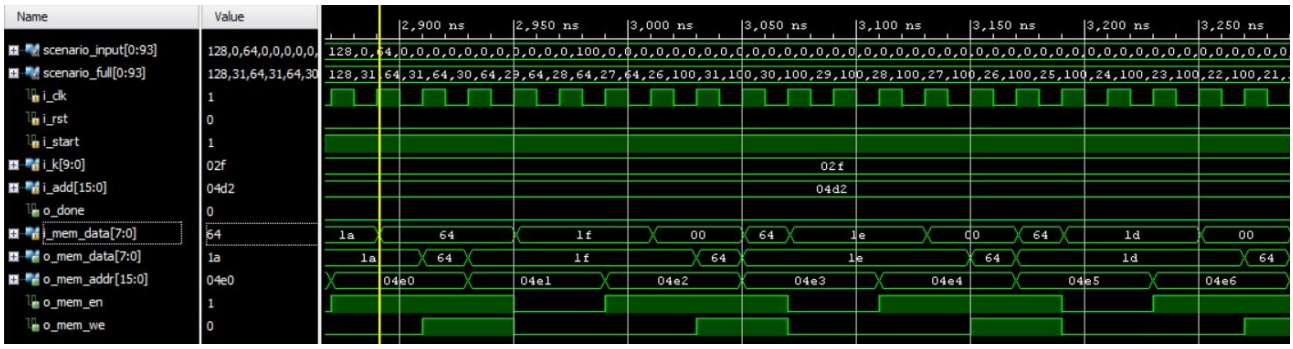



*Figura 18: Inizia la sotto-sequenza di zeri.*

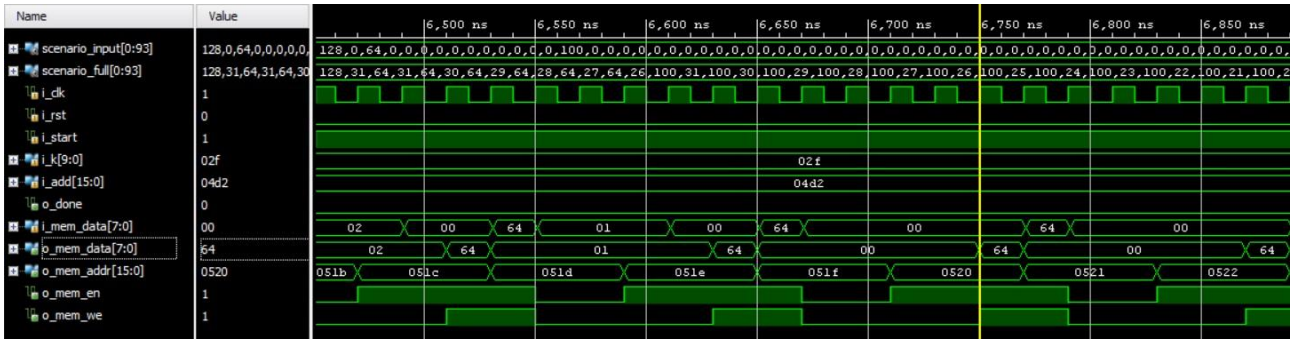



*Figura 19: Al 32-esimo 0 letto corrisponde una credibilità pari a zero, quindi non negativa.*

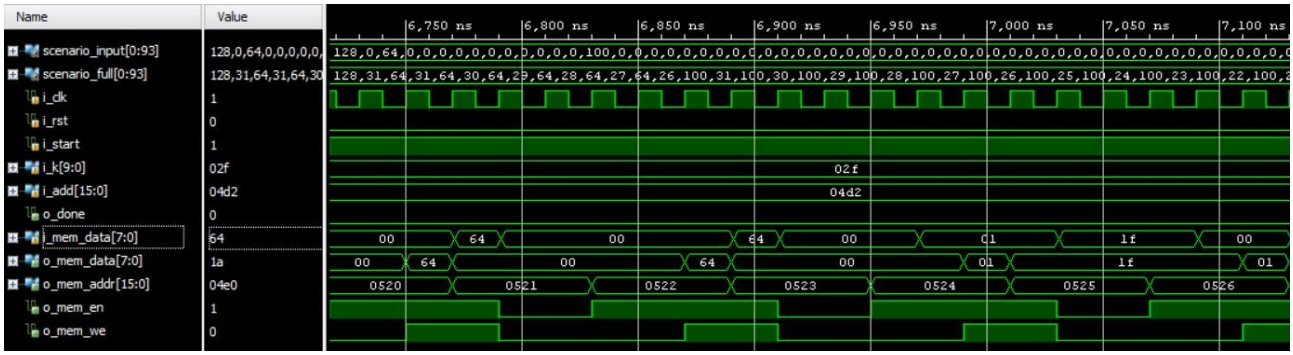



*Figura 20: Dopo la sotto-sequenza di zeri, viene letto il valore 1 e l'elaborazione procede normalmente.*

6° TEST BENCH: Elaborazioni multiple senza reset.

Dopo aver analizzato i casi limite che si possono verificare in una singola sequenza, viene ora analizzata la situazione in cui, una volta terminata l'elaborazione della prima sequenza, il modulo deve analizzarne una seconda, segnalata con l'alzamento del segnale i_start a 1.

In questo caso, non viene alzato il segnale di reset, ma si attende la fine della prima elaborazione prima di iniziare con la seconda.

Questo test bench mira quindi a testare il modulo in questa particolare circostanza.

Come già analizzato nei casi precedenti, viene dato il reset al modulo prima di iniziare la prima elaborazione e si attende l'alzamento del segnale di start.

Figura 21 e 22 mostrano l'inizio e la fine dell'elaborazione della prima sequenza. Figura 23 e 24 mostrano invece l'inizio e la fine dell'elaborazione della seconda, dimostrando che il modulo permette l'elaborazione di più sequenze, come desiderato da specifica.

Osservando attentamente si nota il cambiamento di valore dei segnali i_add e i_k quando inizia l'elaborazione della seconda sequenza, passando rispettivamente da 00e a 00c e da 04d2 a 091e (valori in base 16).

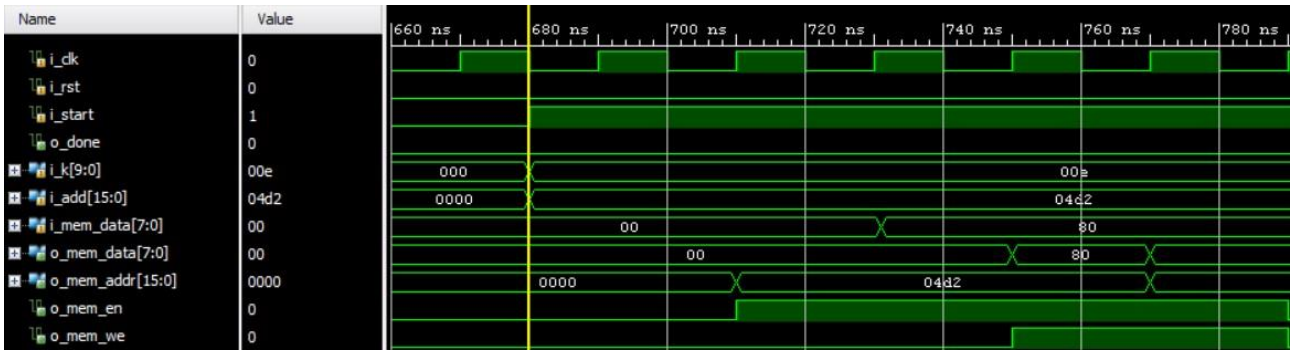



*Figura 21: Inizio elaborazione della prima sequenza.*

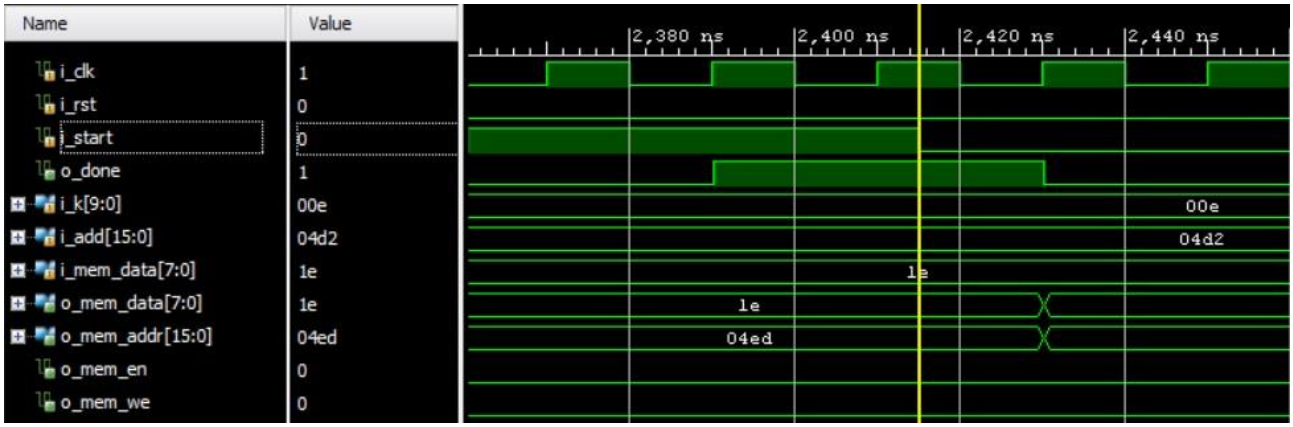



*Figura 22: Fine elaborazione della prima sequenza.*

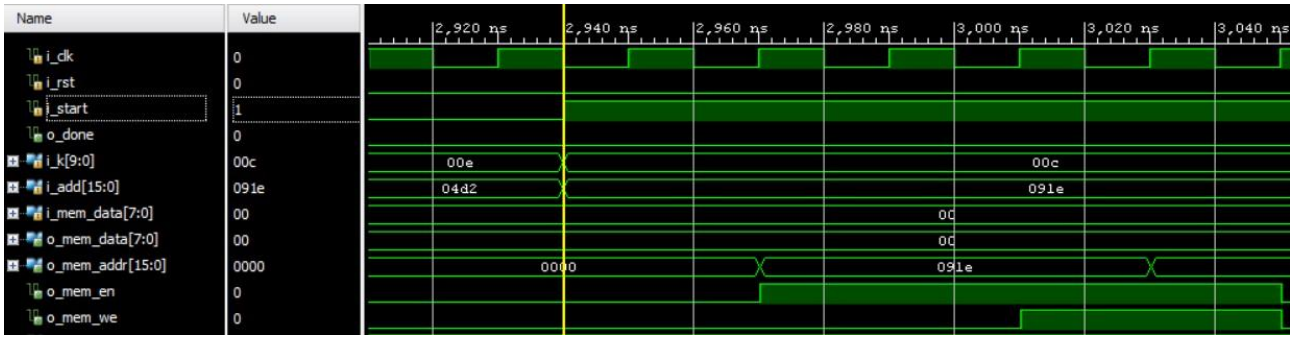



*Figura 23: Inizio elaborazione della seconda sequenza.*

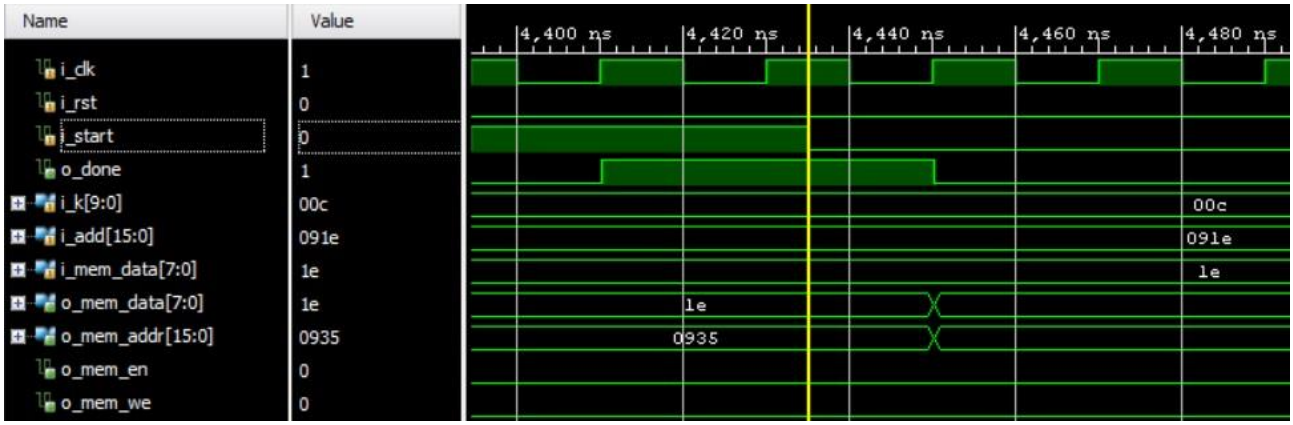



*Figura 24: Fine elaborazione della seconda sequenza.*

7° TEST BENCH: Elaborazioni multiple con reset.

L'ultima situazione che potrebbe verificarsi è quando, durante l'elaborazione di una sequenza, il segnale asincrono i_rst viene innalzato a 1, causando il reset del modulo. Successivamente, il segnale i_start passa a 1 e inizia una seconda elaborazione.

Questo test bench è progettato per verificare se il reset del modulo avviene correttamente e se il modulo elabora correttamente una seconda sequenza in ingresso.

In Figura 25 viene mostrato il momento in cui viene dato il reset durante l'elaborazione della prima sequenza, confermando che il modulo re-inizializza i segnali interessati da questo evento. Osservando attentamente si nota il cambiamento di valore dei segnali i_add e i_k quando inizia l'elaborazione della seconda sequenza, passando rispettivamente da 00e a 00c e da 04d2 a 091e (valori in base 16).

In Figura 26 si dimostra che il modulo si comporta correttamente quando il segnale di start va a 1 una volta terminato il reset. Infine, Figura 27 conferma che il modulo riesce ad elaborare con successo la seconda sequenza.

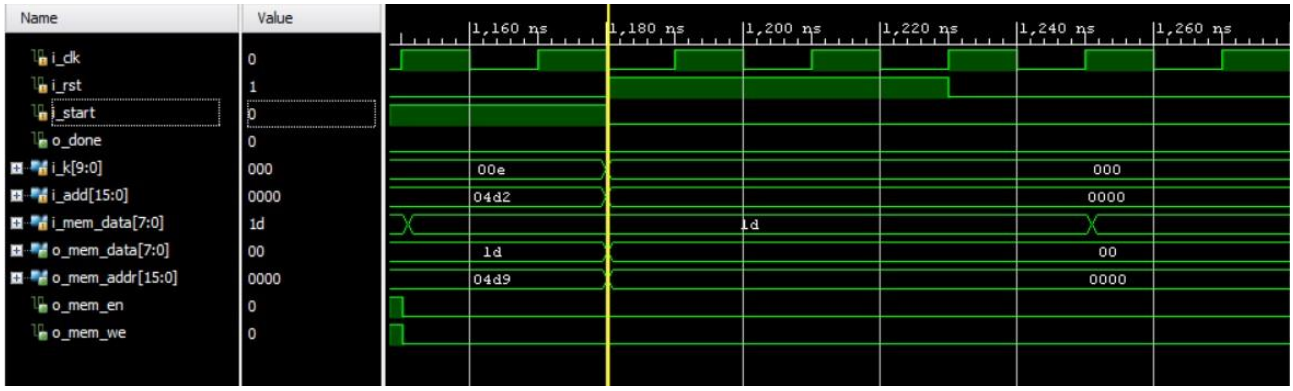



*Figura 25: Segnale di reset va a 1 durante l'elaborazione di una sequenza.*

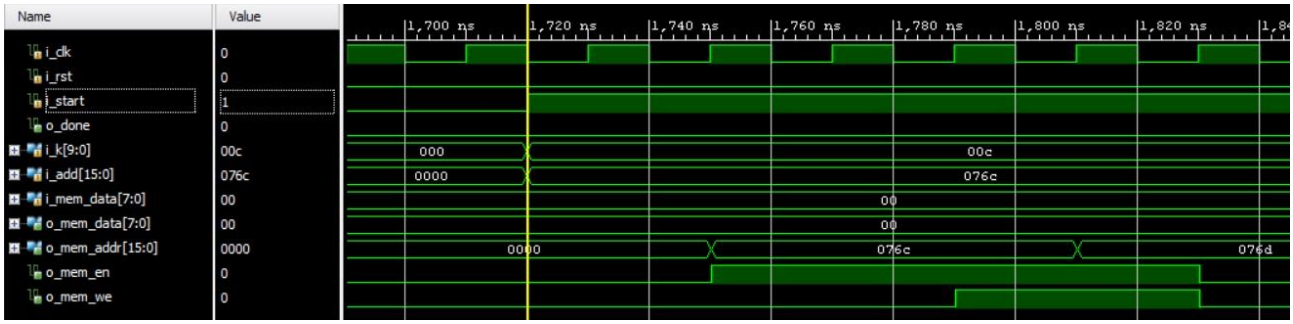



*Figura 26: Inizio di una seconda elaborazione.*

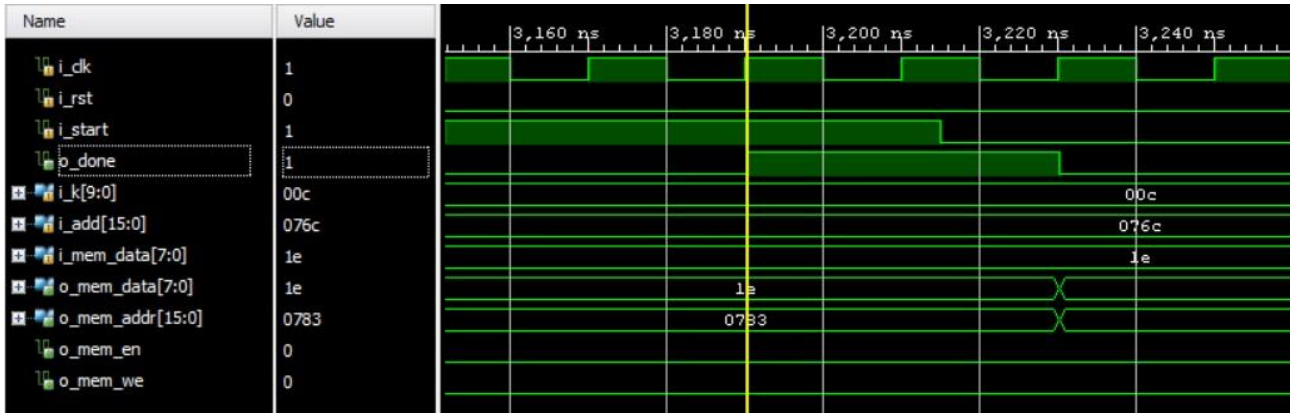



*Figura 27: Il modulo elabora correttamente la seconda sequenza.*

## 4.3 CONCLUSIONI TEST BENCHES

Dopo aver illustrato in dettaglio tutti i test benches utilizzati per la fase di testing, si conclude che il modulo elabora e gestisce opportunamente tutti i “corner case” definiti, confermando quindi la corretta implementazione del modulo rispetto alla specifica.

# CAPITOLO 5

## 5.1 CONCLUSIONI

La “Prova Finale di Reti logiche” è focalizzata sulla progettazione e implementazione di un modulo hardware per completare una sequenza di parole.

Durante la fase di progettazione, è stato sviluppato un modulo efficace ed efficiente, prestando attenzione alla fase di pre-sintesi per garantire la correttezza del codice. Successivamente, è stata condotta un'approfondita fase di post-sintesi per verificare e ottimizzare le prestazioni del circuito, definendo e testando il modulo su diversi test benches.

In conclusione, questo lavoro ha condotto alla realizzazione di un modulo di successo che soddisfa pienamente la specifica.