POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Relazione Prova finale di Reti Logiche

Luca Pozzato - 10794909, Andrea Rossi - 10799170

Academic year:
2023-2024

Abstract: Lo scopo del progetto è la descrizione in VHDL di un componente hardware che, interfacciandosi con una memoria in cui sono presenti alcuni dati, esegue un'analisi su essi e completa i dati mancanti della memoria stessa.

Indice

1. Introduzione

1.1. Specifiche

Il componente hardware da descrivere è connesso ad una memoria e riceve input dall'esterno; in particolare viene fornito un indirizzo della memoria `ADD`, il numero di parole da analizzare `K` ed un segnale di avvio `START`. Quando il modulo riceve il via, analizza la stringa formata dalle prime `K` parole presenti in memoria all'indirizzo `ADD` come viene descritto ora.

La sequenza da processare è costituita da `K` parole `W` il cui valore è compreso tra 0 e 255, dove il valore zero va interpretato come *"valore non specificato"*. Le parole sono memorizzate a partire da un indirizzo `ADD` ogni due byte (e.g. `ADD, ADD+2... ADD+2*(K-1)`). Il byte mancante (agli indirizzi `ADD+1, ADD+3..., ADD+2(K-1)+1`) dovrà essere completato con il valore di *credibilità* `C` della relativa parola `W`. Essa può assumere valori tra 31 e 0: è pari a 31 ogni volta che il valore `W` è *valido* (diverso da zero) mentre viene decrementato rispetto al valore precedente nel caso in cui si incontri uno zero in `W`.
L'obiettivo del componente è processare la stringa e modificarla come segue:

- sostituire le parole con valore 0 con l'ultimo valore *valido*, cioè l'ultimo valore della sequenza letto diverso da zero;
- completare per ogni parola il byte mancante con il suo valore di *credibilità* `C`.

1.2. Esempio

Qui sotto è mostrato un esempio di una stringa prima e dopo l'elaborazione del componente. I numeri sono mostrati in formato decimale per semplicità di lettura.

120	0	**10**	0	**0**	0	**0**	0	**16**	0

Stringa in ingresso con `K` = 5 parole (in **grassetto**)

120	*31*	**10**	*31*	***10***	*30*	***10***	*29*	**16**	*31*

Stringa processata (in *corsivo* le modifiche)

Come è possibile vedere, nelle parole successive ai valori 120, 10 e 16 è stata assegnata credibilità 31, nelle parole con valore 0 è stato assegnato l'ultimo valore letto valido (10) e la credibilità di queste parole è stata decrementata prima a 30, poi a 29.

2. Architettura

2.1. Interfaccia

Il modulo ha interfaccia come mostrato nella Figura 1. Di seguito la descrizione delle varie porte nella Tabella 1. Tutti i segnali sono sincroni e devono essere interpretati sul fronte di salita del clock; fa eccezione il segnale di reset che è asincrono.

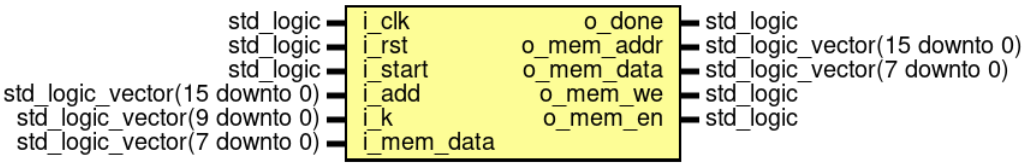

Figura 1: Diagramma del componente

Nome Porta	Tipo	Descrizione
Ingressi Principali		
`i_start`	1 bit	Segnale di inizio elaborazione
`i_add`	16 bit	Indirizzo della memoria da cui inizia la porzione da processare
`i_k`	10 bit	Numero di parole da elaborare
Ingressi Ausiliari		
`i_clk`	1 bit	Segnale di clock unico per tutto il sistema
`i_rst`	1 bit	Segnale di reset asincrono
Uscita Principale		
`o_done`	1 bit	Segnale di terminata elaborazione
Connessioni con la Memoria		
`i_mem_data`	8 bit	Contenuto in ingresso dalla memoria quando letta
`o_mem_en`	1 bit	Segnale di *enable*, deve essere 1 per comunicare con la memoria
`o_mem_we`	1 bit	Segnale di *write enable*, deve essere 1 per scrivere, 0 per leggere
`o_mem_addr`	16 bit	Indirizzo della memoria da cui leggere o in cui scrivere
`o_mem_data`	8 bit	Dato che verrà scritto in memoria

Tabella 1: Descrizione delle porte di interfaccia

2.2. Processi

Il modulo è composto da un unico processo *design*, con elementi della lista di sensibilità i segnali `i_clk` ed `i_rst`. La *FSM* (macchina a stati) è sincronizzata in modo da computare uno stato ad ogni ciclo di clock, mentre, essendo il reset asincrono, è presente nella lista di sensibilità in modo da far terminare l'elaborazione nel momento in cui si verifica un fronte di salita del segnale stesso.
Abbiamo scelto di implementare l'intero componente attraverso un unico processo invece di dividerlo in tre sottoprocessi - cambio di stato, *lambda* (funzione di stato prossimo), *delta* (funzione di uscita) - in quanto la bassa complessità dell'elaborazione rispetto ai tempi disponibili permettevano la costruzione di codice più semplice e leggibile, evitando problemi di sincronizzazione o creazione di latch non desiderati.

2.3. Segnali

Nella Tabella 2 vengono presentati i segnali utilizzati all'interno del modulo.

Nome Segnale	Tipo	Descrizione
`k`	10 bit	Numero di parole rimanenti da elaborare
`addr`	16 bit	Indirizzo da cui leggere o in cui scrivere il prossimo dato
`data`	8 bit	Contenuto dell'ultimo valore di parola `W` valido
`cred`	8 bit	Credibilità `C` da assegnare alla parola corrente
`current_state`	`state_type`	Stato corrente dell'elaborazione

Tabella 2: Segnali interni

2.4. Macchina a stati

Il funzionamento del componente è sintetizzato nella macchina a stati rappresentata in Figura 2. La descrizione di ogni stato è approfondito nella Tabella 3.

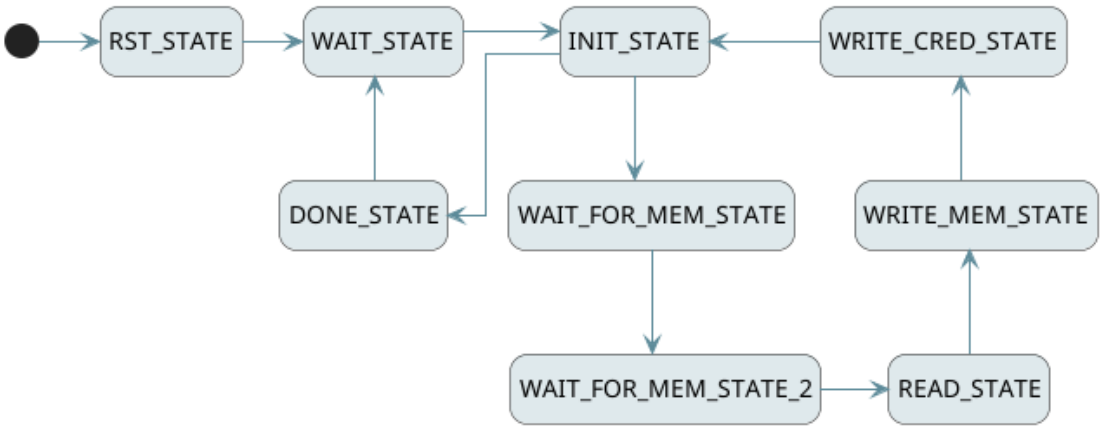

Figura 2: Macchina a stati

Stato	**Descrizione**
`RST_STATE`	Stato di reset, accedibile da ogni stato quando `i_rst` = 1; quando `i_rst`= 0 si passa a `WAIT_STATE`
`WAIT_STATE`	Stato di attesa di una nuova elaborazione; quando `i_start` = 1 vengono inizializzati i segnali `k` e `addr` con i rispettivi segnali in ingresso, mentre a `cred` viene assegnato il valore 31, infine si passa a `INIT_STATE`
`INIT_STATE`	Stato di inizio elaborazione; se `k` $\neq 0$ viene attivato il collegamento con la memoria imponendo `o_mem_en` = 1 e si passa a `WAIT_FOR_MEM_STATE`, altrimenti si impone `o_done` = 1 e si passa a `DONE_STATE`
`WAIT_FOR_MEM_STATE`	Stato di attesa di attivazione della memoria, si passa a `WAIT_FOR_MEM_STATE_2`
`WAIT_FOR_MEM_STATE_2`	Stato di attesa di ricezione del contenuto della memoria, necessario a causa del ritardo di 2 *ns* introdotto dalla memoria tra la ricezione della richiesta di dati e l'invio degli stessi; si passa a `READ_STATE`
`READ_STATE`	Stato di lettura del contenuto della memoria e decisione dei valori in uscita verso la memoria; se è un valore non valido si mantiene in `data` l'ultimo valore valido e si decrementa `cred`, altrimenti si impone il valore corrente a `data` e si resetta `cred`= 31. Viene attivata la scrittura in memoria imponendo `o_mem_we` = 1 e si passa a `WRITE_MEM_STATE`
`WRITE_MEM_STATE`	Stato di scrittura del dato in memoria, in cui viene scritto il valore presente nel segnale `data`, viene aggiornato il segnale `addr` ad `addr` +1 e si passa a `WRITE_CRED_STATE`
`WRITE_CRED_STATE`	Stato di scrittura della credibilità in memoria, in cui viene scritto il valore presente nel segnale `cred`,viene aggiornato il segnale `addr` ad `addr` +1 e si passa a `INIT_STATE`
`DONE_STATE`	Stato di terminazione dell'elaborazione, quando `i_start` viene posto a zero si passa in `WAIT_STATE` ed il componente è pronto ad una nuova elaborazione

Tabella 3: Stati della *FSM*

In ogni stato vengono imposti i valori desiderati ai segnali che saranno utilizzati nello stato successivo: infatti tutti i segnali vengono aggiornati sul fronte di salita del clock, il quale coincide con il cambio di stato. Ad esempio per far sì che durante lo stato di `WRITE_MEM_STATE` venga scritto il dato in memoria è necessario imporre `o_mem_we` = 1 nello stato di `READ_STATE`, in modo che al successivo fronte di salita del clock (quindi al conseguente passaggio di stato) il segnale venga imposto ad uno.

3. Risultati sperimentali

3.1. Report di sintesi

Dopo aver sintetizzato il design del progetto, il circuito è stato simulato utilizzando il test bench fornito. Il circuito si comporta correttamente, superando il test sia con la simulazione comportamentale (*behavioral simulation*) che con la simulazione post-sintesi (*post-synthesis simulation*). Di seguito, sono riportate le tabelle con il timing report e l'utilizzo delle risorse. Nella Tabella 4 è mostrato il timing report del design; analizzandolo, si può notare che il Worst Negative Slack (*WNS*) è di 15,413 ns. Un WNS di 15,413 ns con un periodo di clock di 20 ns indica che il circuito impiega (20 ns - 15,413 ns = 4,587 ns) per completare l'esecuzione di uno stato della FSM del design, dato che la FSM è stata sincronizzata in modo da eseguire uno stato per ogni ciclo di clock, come è possibile notare dalla Figura 3.

Setup	**Hold**	**Pulse Width**
Worst Negative Slack (WNS): 15,413 ns	Worst Hold Slack (WHS): 0,148 ns	Worst Pulse Width Slack (WPWS): 9,500 ns
Total Negative Slack (TNS): 0,000 ns	Total Hold Slack (THS): 0,000 ns	Total Pulse Width Negative Slack (TPWS): 0,000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 119	Total Number of Endpoints: 119	Total Number of Endpoints: 79

Tabella 4: Design Timing Summary

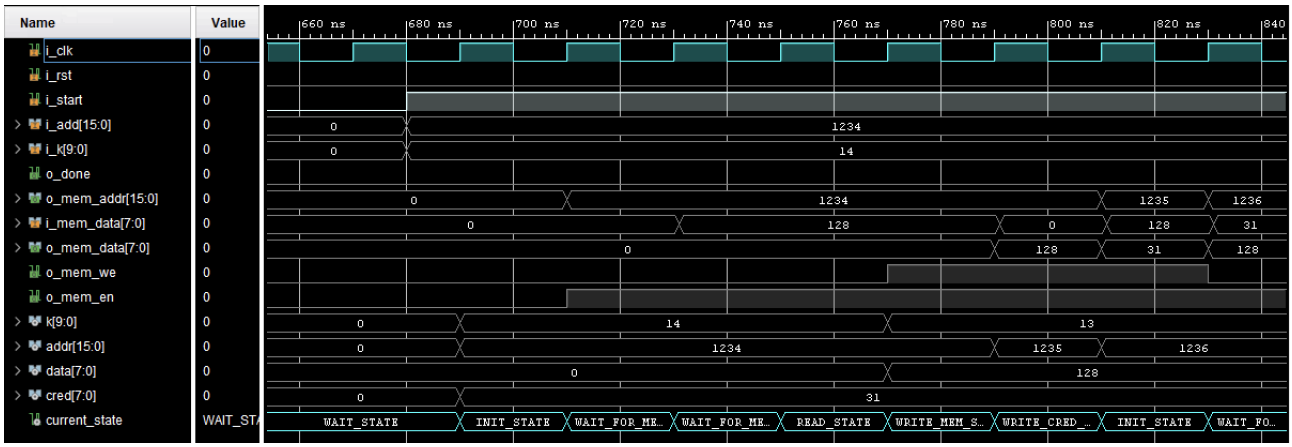

Figura 3: Sincronizzazione della FSM con il clock

Da questo valore è anche possibile calcolare il periodo di clock minimo che il design può supportare. Sapendo che la memoria (RAM) ha un ritardo massimo di 2 ns nelle operazioni di scrittura e lettura, il periodo di clock minimo è dato da:

$$T_{clk} = 20\ ns + 2\ ns - 15,413\ ns = 6,587\ ns \tag{1}$$

e la frequenza di clock massima supportata dal design è:

$$f_{clk} = \frac{1}{T_{clk}} = \frac{1}{6,587\ ns} \approx 151,8\ MHz \tag{2}$$

Dalla Tabella 5 si può invece osservare come il design utilizzi 78 registri di tipo *Flip-Flop* e 86 *LUTs*, il che significa che il design non presenta alcun tipo di *Latch*. Questo evita comportamenti imprevedibili e indesiderati, assicurando inoltre che il design sia completamente sincrono. Per evitare la creazione di Latch e sincronizzare correttamente il design con il periodo di clock, è stato necessario implementare la FSM in un singolo processo invece di dividerla in più processi.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	86	0	134600	0,06
LUT as Logic	86	0	134600	0,06
LUT as Memory	0	0	46200	0,00
Slice Registers	78	0	269200	0,03
Register as Flip Flop	78	0	269200	0,03
Register as Latch	0	0	269200	0,00
F7 Muxes	0	0	67300	0,00
F8 Muxes	0	0	33650	0,00

Tabella 5: Resource Utilization Summary

3.2. Simulazioni

Per assicurarsi che il design fosse corretto e privo di errori, sono stati simulati vari test bench con l'obiettivo di verificare diversi corner cases. Inoltre, è stato sviluppato un codice in Python per generare automaticamente una serie di test bench, inizializzando la RAM con messaggi composti da $\texttt{K} \in [1, 100]$ parole e assegnando a ciascuna parola un valore $\texttt{W} \in [0, 255]$, come da specifica. I test bench generati da Python sono stati successivamente simulati con GHDL, uno strumento utilizzato per la simulazione e verifica dei design VHDL.

3.2.1. Tesh bench fornito

In questo test bench viene illustrato il funzionamento del design con una sequenza definita come *"normale"*, in cui viene mostrato il corretto utilizzo di tutti i segnali per interfacciarsi con la RAM. Nella Figura 4 e nella Figura 5 è possibile osservare i segnali di inizio e fine elaborazione, che vengono attivati correttamente durante l'esecuzione del design. In particolare, il segnale di inizio elaborazione (`i_start`), attivato una volta che il segnale di reset (`i_rst`) è disattivato, avvia correttamente l'elaborazione del design, come si può notare dal cambiamento degli stati; in dettaglio, la FSM passa da `WAIT_STATE` a `INIT_STATE`, procedendo con la lettura della RAM. Una volta che il design ha terminato l'elaborazione della sequenza, il segnale di terminazione dell'elaborazione (`o_done`) viene attivato e, infine, il segnale `i_start` viene disattivato, rendendo il design pronto per una nuova elaborazione.

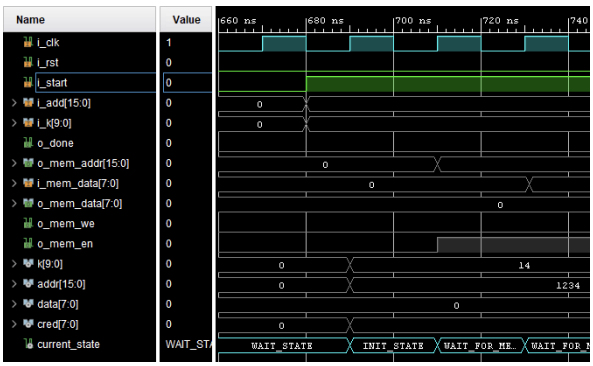

Figura 4: Segnali di inizio elaborazione

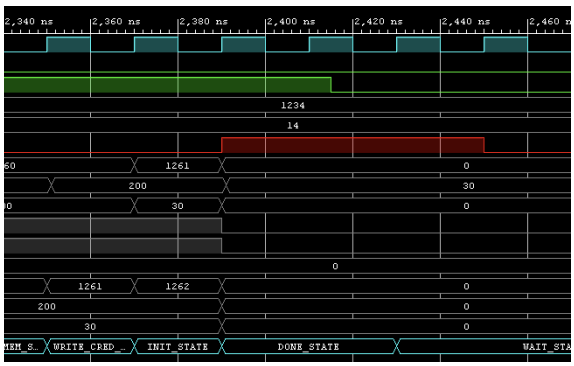

Figura 5: Segnali di fine elaborazione

3.2.2. Sequenza di lunghezza nulla

In questo test bench viene illustrato il comportamento del design quando la sequenza è vuota, ovvero quando il segnale che indica la lunghezza della sequenza (`i_k`) è pari a 0. Come mostrato nella Figura 6, il design avvia correttamente l'elaborazione non appena il segnale di reset viene disattivato e quello di start attivato. Tuttavia, appena rileva che $\texttt{k} = 0$ leggendo dalla RAM, il design conclude immediatamente l'elaborazione, attivando il segnale `o_done`, che viene poi disattivato quando `i_start` è disattivato. La FSM, una volta attivato `i_start`,

passa da WAIT_STATE a INIT_STATE; da quest'ultimo stato, procede a DONE_STATE se k = 0, altrimenti passa a WAIT_FOR_MEM_STATE, come si può osservare nella figura.

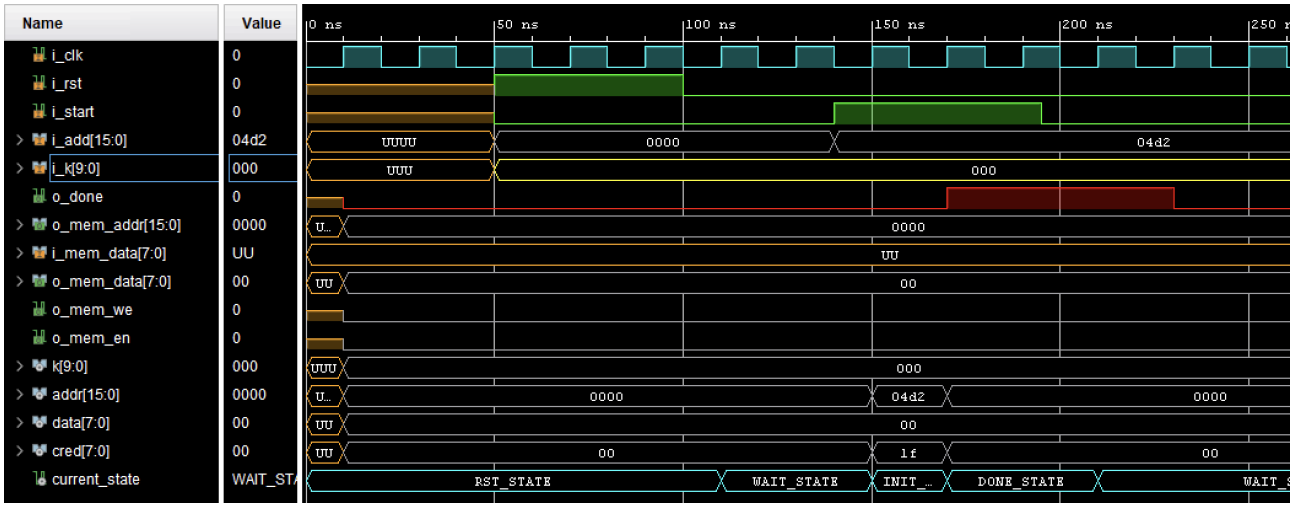

Figura 6: Sequenza di lunghezza nulla

3.2.3. Credibilità che raggiunge 0

Questo test bench illustra il comportamento del design quando la credibilità viene decrementata ogni volta che si incontra uno zero in W, finché non si rileva un valore diverso da 0, momento in cui la credibilità viene reinizializzata a 31 per poi essere nuovamente decrementata se si incontra un altro zero. Secondo la specifica, una volta che la credibilità raggiunge 0, il design deve mantenere tale valore fino a che non si rileva un valore in W diverso da 0 o fino al termine della sequenza. Come previsto, alla fine della sequenza, il design deve attivare il segnale di fine elaborazione. Nella Figura 7 e nella Figura 8 è possibile osservare come il design si comporti correttamente, continuando a decrementare la credibilità fino a 0 e mantenendola a 0, dato che non si verificano valori diversi da 0 in W.

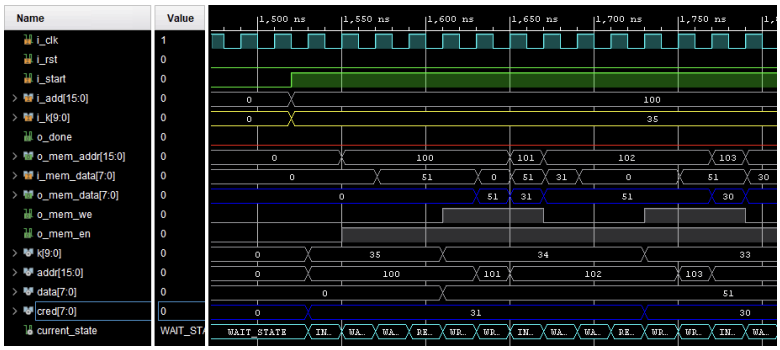

Figura 7: Segnali di inizio elaborazione

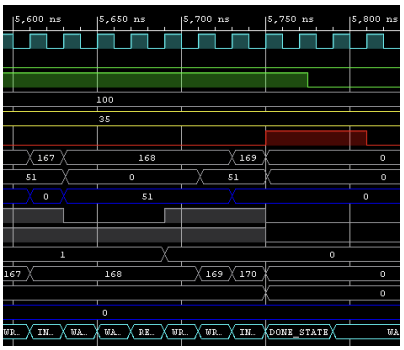

Figura 8: Segnali di fine elaborazione

3.2.4. Test bench con sequenza nulla iniziale

In questo test bench si è voluto verificare il funzionamento del design quando la sequenza di parole W inizia con un valore pari a 0. In questo scenario, il design deve avviare l'elaborazione della sequenza mantenendo il valore W nullo e impostando e mantenendo la credibilità a 0, fino a quando non si legge dalla RAM un valore di W diverso da 0. A quel punto, il design si comporta come mostrato nei test bench precedenti. Nella Figura 9 e nella Figura 10 si verifica che il design opera in modo appropriato: in particolare, si può notare come sia la credibilità che il valore di W rimangano a 0. Questo è evidenziato anche dal segnale o_mem_data (segnale che indica il valore da scrivere nella RAM all'indirizzo o_mem_addr), che resta a 0 fino a quando non viene letto dalla RAM un valore diverso da 0; nel caso del test bench, questo valore è 2 e la credibilità viene poi posta a 31.

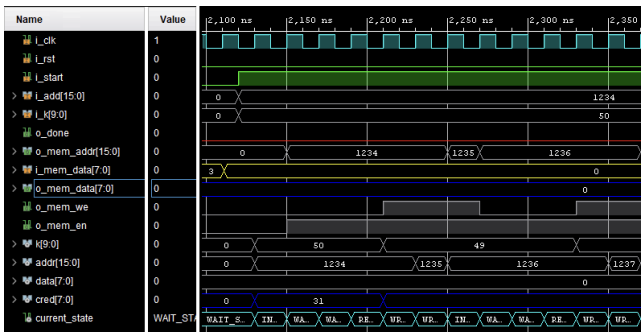

Figura 9: Segnali di inizio elaborazione

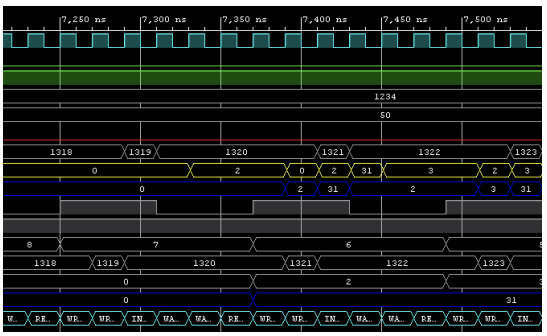

Figura 10: Fine elaborazione di una sequenza di 0

3.2.5. Sequenze multiple

In questo test bench si verifica come il design si comporta nell'elaborazione di sequenze multiple. Dalla specifica, il design, una volta completata l'elaborazione della prima sequenza, deve attivare il segnale di fine elaborazione (`o_done`). Quando `o_done` viene attivato, il segnale `i_start` viene disattivato, seguito dalla disattivazione del segnale di fine elaborazione, il che indica che il modulo è pronto per una nuova elaborazione. Quando `i_start` viene attivato una seconda volta, il modulo avvia la seconda elaborazione. Secondo la specifica, la seconda elaborazione non deve attendere il reset del modulo; questo è verificabile osservando come il modulo, una volta terminata l'elaborazione, passi da `DONE_STATE` a `WAIT_STATE`, e successivamente, con la riattivazione di `i_start`, la FSM passi da `WAIT_STATE` a `INIT_STATE`, procedendo con l'elaborazione della nuova sequenza.

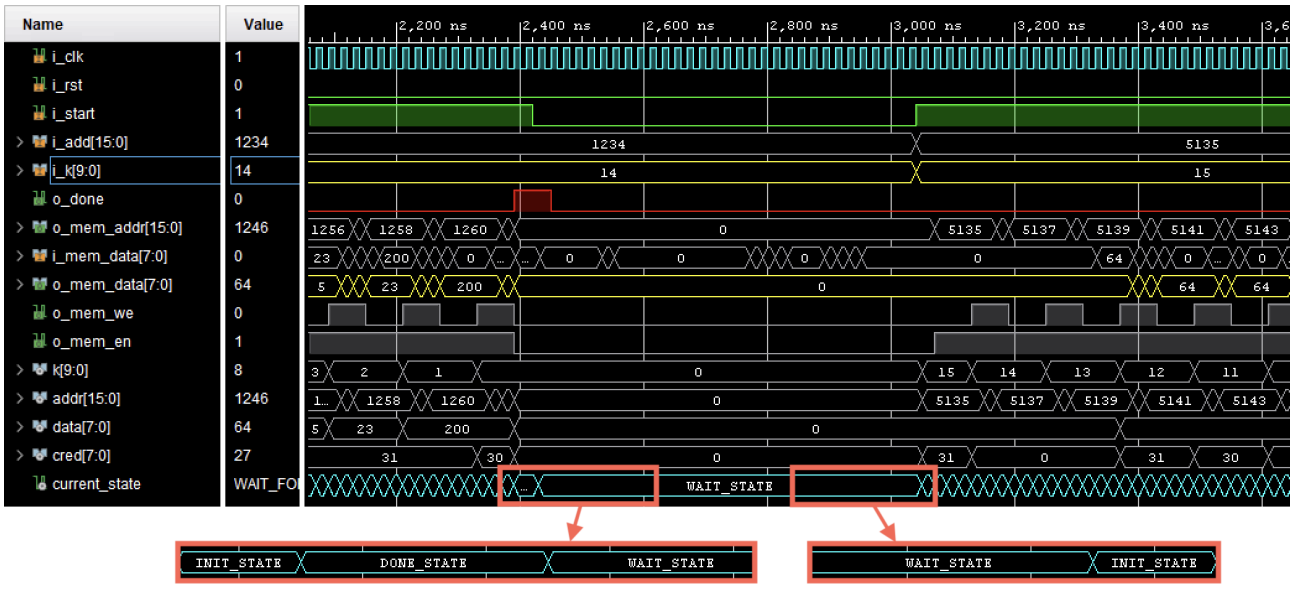

Figura 11: Sequenze multiple

3.2.6. Segnale di reset

In questo test bench si verifica il corretto funzionamento del design in presenza del segnale di reset asincrono. Come si può notare dalla Figura 12, nel test bench viene attivato il segnale di reset `i_rst` anche se non sincronizzato con il periodo di clock. A seguito dell'attivazione del reset, il modulo viene re-inizializzato, disattivando correttamente tutti i segnali necessari. Tra questi segnali, vi sono quelli che si interfacciano con la RAM, come `o_mem_data`, `o_mem_we` e `o_mem_en`, oltre ai segnali interni al modulo, come `k`, `addr`, `data` e `cred`. Il corretto funzionamento del design è evidente anche dagli stati della FSM implementata nel modulo. La FSM passa dallo stato `WRITE_CRED_STATE` a `RST_STATE` non appena il segnale di reset viene portato a 1. Successivamente, appena il segnale di reset viene disattivato e il clock è attivo, la FSM passa da `RST_STATE` a `WAIT_STATE`, indicando che il modulo è pronto per una nuova elaborazione.

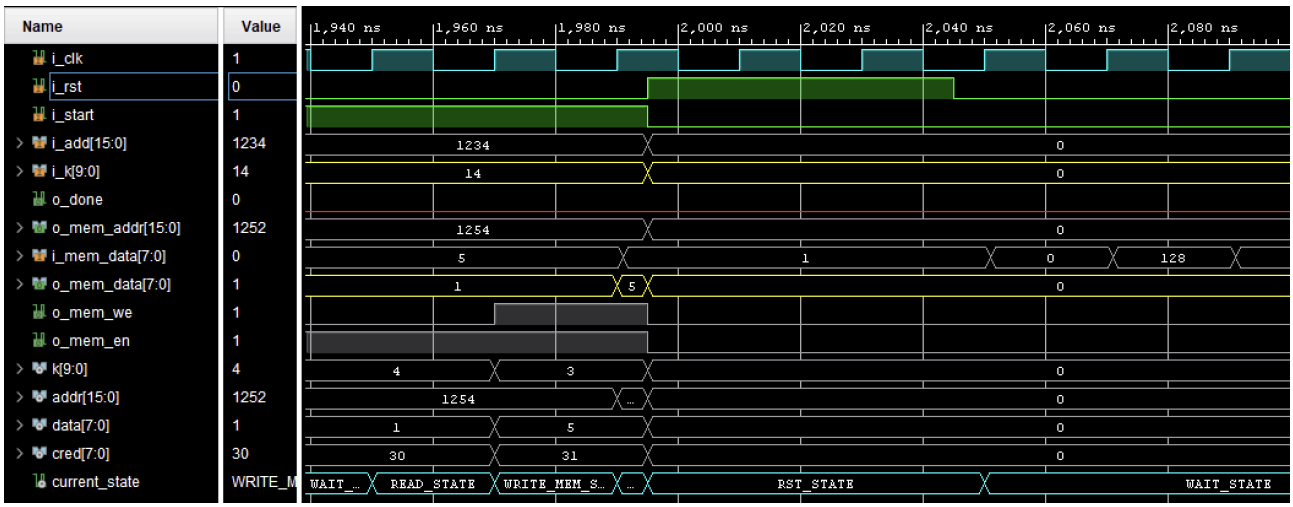

Figura 12: Segnale di reset

3.2.7. Segnale di start nello stato di reset

In questo test bench si verifica il corner case in cui il segnale di inizio elaborazione rimane attivo per tutta la durata dell'attivazione del segnale di reset. Secondo la specifica, il sistema funziona correttamente quando, dopo che il segnale di reset viene disattivato, il modulo avvia l'elaborazione al momento in cui il segnale di start viene portato a 1. Tuttavia, non è specificato il comportamento nel caso in cui il segnale di start non venga mai abbassato, anche dopo l'attivazione del reset. Abbiamo quindi gestito questo scenario con la continuazione dell'elaborazione precedente, senza aspettare che il segnale di start venga abbassato e riattivato, come si può osservare nella Figura 13. Questa scelta è stata fatta immaginando il modulo in un contesto realistico. Ad esempio, se il modulo fosse responsabile di monitorare la corretta lettura di un dispositivo di rilevamento del battito cardiaco (con valori compresi tra 0 e 255) e dovesse segnalare un tecnico quando la credibilità del valore scende a 0, in caso di un reset del modulo senza che la periferica invii un nuovo segnale di start, il sistema continuerebbe a svolgere correttamente il suo compito.

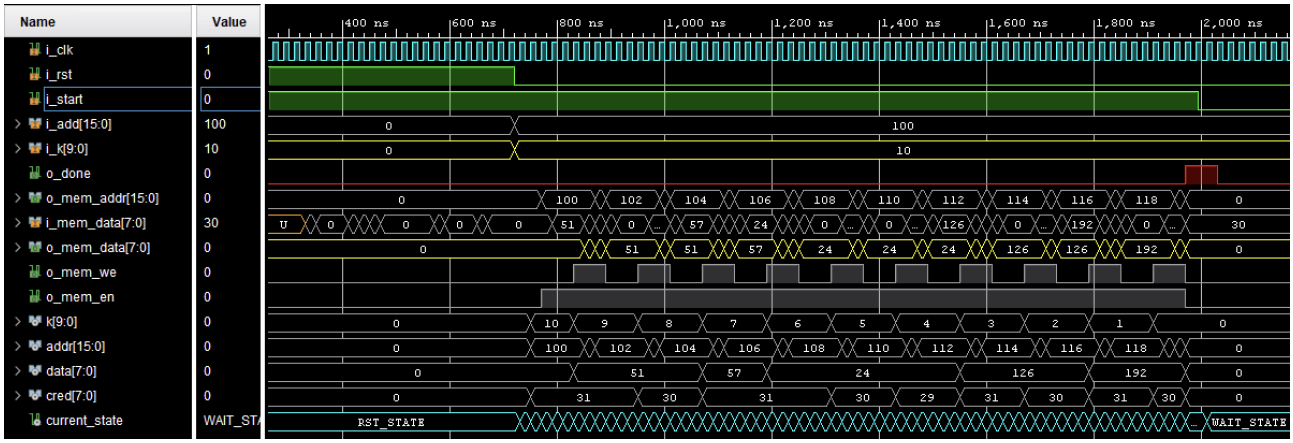

Figura 13: Segnale di start nello stato di reset

3.2.8. Altri test bench

Oltre ai test bench mostrati precedentemente, il modulo è stato testato anche con sequenze di lunghezze variabili, da valori bassi fino a lunghezze di circa $K \approx 1000$. Il modulo è stato inoltre sottoposto a test con più di due sequenze e con segnali di reset casuali durante l'elaborazione.

4. Conclusioni

Il design sintetizzato funziona correttamente sia nei test di simulazione comportamentale che in quelli post-sintesi (sia *timing* che *functional*), dimostrando come il design rispetti pienamente le specifiche, compresi i vincoli riguardanti il periodo di clock, che risultano soddisfatti, come analizzato nel paragrafo dedicato al report di sintesi. Per garantire una sincronizzazione corretta ed efficiente, il modulo è stato implementato utilizzando un singolo processo; data la semplicità del progetto, questo approccio si è rivelato sufficiente. Tuttavia, se fossero stati richiesti vincoli sul periodo di clock più restrittivi, sarebbe stato necessario adottare un approccio diverso. Inoltre, si sarebbe potuto analizzare la FSM per identificare eventuali ridondanze, ottimizzando l'uso dei registri e rendendo il modulo più compatto da questo punto di vista.