Prova Finale (Progetto di Reti Logiche)

Prof. Gianluca Palermo - Anno 2022/2023

Tocalli Nicolò (Codice Persona 10716843 - Matricola 957355)

Stasi Michelangelo (Codice Persona 10787871 - Matricola 957140)

Indice

---

1. Introduzione

1.1 Scopo del progetto

Lo scopo del progetto è quello di realizzare un componente hardware descritto in VHDL che possa interfacciarsi con una memoria.
Il componente da sintetizzare riceve in input informazioni riguardanti un indirizzo di memoria e un canale di output tra i 4 disponibili e dopo avere prelevato il contenuto della cella specificata, lo mostra sul canale di output selezionato.

1.2 Interfaccia del componente e funzionamento

Il componente presenta cinque segnali di input e sei segnali di output primari:

`i_clk` è il segnale di clock in ingresso al componente sequenziale che sincronizza sul fronte di salita lettura e scrittura dei segnali e dei registri.
`i_rst` è il segnale di reset asincrono che se asserito re-inizializza il componente e azzera i valori dei registri e delle uscite.
Il segnale `i_w` a 1 bit viene letto sequenzialmente su ogni fronte di salita del clock ma viene interpretato solo durante il periodo nel quale `i_start` è alto.
I primi due bit di `i_w` validi codificano una delle 4 uscite (00 →OUT0, 01→OUT1 , 10→OUT2 , 11→OUT3) mentre i restanti N bit (eventualmente estesi a sinistra a 0) formano l'indirizzo di memoria a 16 bit passato alla memoria tramite il segnale `o_mem_addr.`
`i_mem_data` è il segnale a 8 bit con cui il componente ottiene il dato letto all'indirizzo di memoria specificato.
`o_z0, o_z1, o_z2, o_z3` sono i canali di uscita a 8 bit dove vengono mostrati i dati, mentre `o_done` è un segnale a 1 bit che indica quando il componente sta mostrando i risultati sulle uscite e quando invece tutti gli 8 bit delle 4 uscite sono posti a 0.

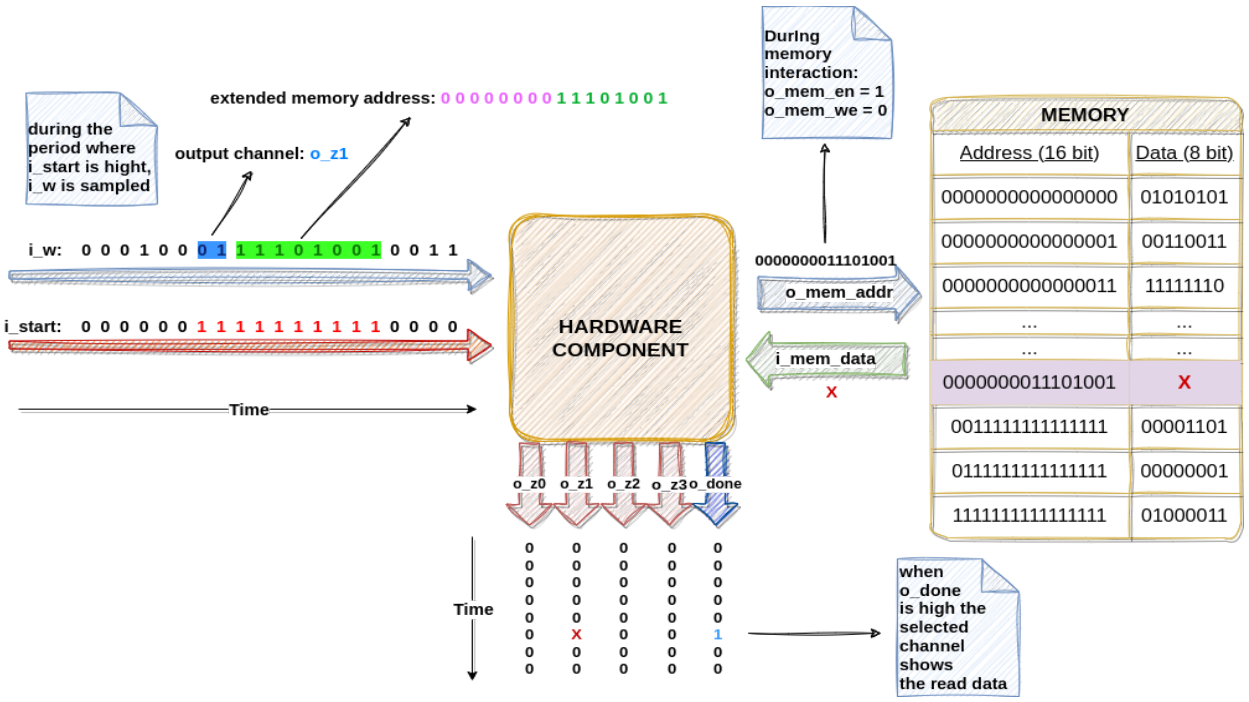


Figura 1 : disegno esplicativo del funzionamento del componente

2 . Architettura

Il componente possiede due processi :
- **il primo processo** è di tipo puramente sequenziale e gestisce l'aggiornamento dei registri di memoria sul fronte di salita del clock e soprattutto gestisce il comportamento del componente quando riceve un segnale `i_rst` asincrono.
- **il secondo processo** invece regola gli stati della macchina a stati finiti, si occupa quindi dell'elaborazione della sequenza di ingresso e della lettura in memoria del dato associato;

Qui di seguito una descrizione dettagliata dei due processi.

2.1 Processo sequenziale

Questo processo contiene nella lista di sensitività i segnali `i_rst` e `i_clk`. In particolare, gestisce il reset asincrono, portando tutti i registri e i segnali al valore iniziale di reset. Inoltre, ad ogni fronte di salita del clock vengono aggiornati i segnali necessari per il corretto funzionamento del componente.

2.2 Processo della FSM

Questo è il processo principale del componente, il quale gestisce la lettura del segnale di ingresso `i_w` e associa ad ogni bit della sequenza un registro dedicato. È composto da 8 stati elencati di seguito. Il diagramma è rappresentato in Figura 2.

2.2.1 START

È lo stato iniziale della macchina, lo stato di "RESET". Il suo scopo è di attendere che il segnale `i_start` diventi alto, e di inizializzare alcuni segnali del componente, in modo tale da avere un corretto funzionamento.
Essendo uno stato di "RESET", quando si riceverà un segnale `i_rst` alto, la macchina tornerà in questo stato.

2.2.2 READ_CHANNEL

Quando il segnale `i_start` assume valore alto, inizia la lettura del segnale `i_w` campionato sul fronte di salita del `i_clock`. In particolare abbiamo utilizzato un segnale `enable` a 1 bit che ci permette di distinguere il primo bit di ingresso dal secondo : infatti, una volta ricevuto il primo bit, dopo averlo salvato adeguatamente in un registro a 2 bit per memorizzare il canale attuale, `enable` cambia valore e quando viene ricevuto il secondo bit, esso viene salvato nella posizione corretta del registro di memoria che codifica il canale di output.
Una volta letto il secondo bit, si passa allo stato adibito alla lettura dell'indirizzo.

2.2.3 READ_ADDRESS

Sapendo che `i_start` può essere alto per un intervallo compreso tra i 2 e i 18 cicli di clock, qui, a differenza dello stato precedente, viene gestito il caso in cui `i_start` passi a '0' prima di 18 cicli : in tal caso `curr_addr` conterrà il valore finale dell'indirizzo, senza il bisogno eventuale di estendere a sinistra con degli zeri in quanto il processo di modifica da noi applicato lascia a 0 tutti i bit più significativi di `curr_addr` che non sono direttamente modificati nello stato corrente.

Nei cicli in cui i_start è alto il registro che contiene l'indirizzo attualmente letto viene moltiplicato per 2 (e quindi sommato a se stesso) e poi viene sommato con il segnale i_w_curr che contiene l'input i_w campionato sul fronte di salita del clock. In questo modo l'indirizzo parziale letto nel ciclo precedente viene traslato a sinistra di una posizione e il nuovo valore di i_w viene aggiunto in posizione meno significativa.
Inoltre abbiamo introdotto un segnale curr_pos che tiene traccia della posizione dell'attuale i_w all'interno della sequenza di 16 bit, in modo tale da attivare ad ogni ciclo di lettura il processo (siccome il segnale è stato aggiunto nella lista di sensitività e viene modificato a prescindere dall'input ricevuto), anche nel caso in cui il nuovo valore di i_w letto sia 0 e quindi il registro contenente l'indirizzo parziale non cambi.

2.2.4 READ_MEM

In questo stato il segnale o_mem_addr assume il valore di curr_addr ottenuto nello stato precedente e viene mandato in input alla memoria; inoltre, affinché la comunicazione avvenga, sono modificati i segnali o_mem_we e o_mem_en, che, da specifica, sono rispettivamente i segnali di write enable (posto a 0 siccome vogliamo leggere un dato) e di enable (mandato alla memoria per poter comunicare).

2.2.5 WAIT_MEM

Questo è uno stato di attesa che dura un ciclo di clock affinché la memoria possa modificare il segnale di input i_mem_data contenente il valore della cella di memoria all'indirizzo passato nello stato precedente.
Inoltre vengono portati ai valori di default i segnali di enable della memoria.

2.2.6 WRITE_MEM

In questo stato il registro che contiene il valore dell' uscita principale codificata da curr_channel viene aggiornato al valore appena letto dalla memoria mentre le altre 3 uscite vengono aggiornate in base ai valori precedentemente assunti nei processi di lettura passati.

2.2.7 DONE

Stato di attesa in cui viene inizializzato il prossimo valore di o_done.

2.2.8 LAST

In questo stato o_done assume valore alto e contemporaneamente le uscite mostrano i contenuti dei loro rispettivi registri per 1 ciclo di clock.

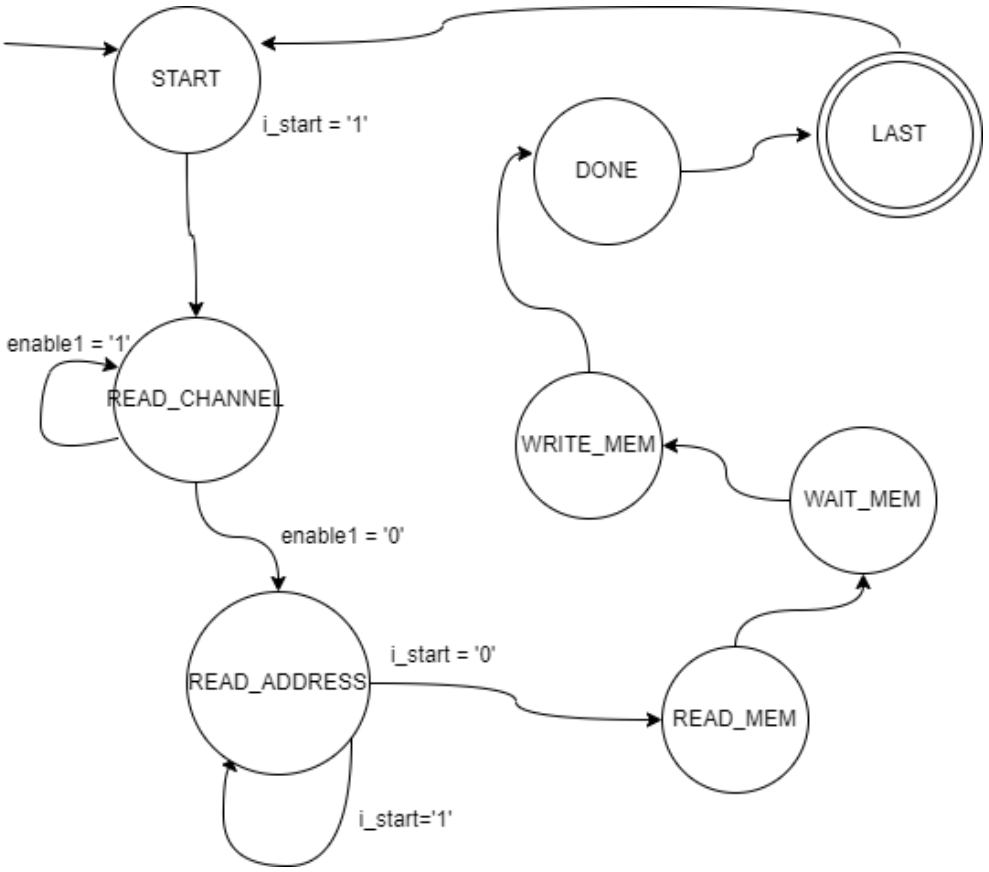


Figura 2 : Diagramma degli stati della FSM con alcune condizioni principali

2.3 Scelte progettuali

Nel processo sequenziale abbiamo deciso di gestire alcuni casi particolari in base ad un segnale contenente lo stato corrente, ad esempio:

- quando lo stato attuale è WRITE_MEM, il registro associato al canale dato dalla sequenza viene aggiornato con il valore di input `i_mem_data` dato dalla memoria;
- quando lo stato attuale è DONE e lo stato prossimo è LAST, tutte le uscite vengono poste al valore contenuto nel registro corrispondente; in ogni altro caso, esse sono poste al valore di default.

Un'altra scelta progettuale è stata non inserire tutti i segnali che vengono letti nella lista di sensitività, poichè ciò portava ad un eccesso di chiamate del processo e alla perdita di sincronia.

Sfruttando l'assegnamento sequenziale nei processi, abbiamo assegnato ad alcuni segnali prossimi il loro valore attuale all'inizio del processo, così, nel caso essi siano modificati

nell'algoritmo, l'ultima modifica viene salvata, mentre se non ci sono modifiche, il segnale torna a contenere il valore precedente. Questo ci ha permesso di evitare di avere segnali a cui veniva assegnato il valore 'X' (unknown) nelle simulazioni.
Unico segnale che non ha seguito questa logica è stato il segnale contenente l'indirizzo della memoria ottenuto dalla sequenza, siccome il valore prossimo veniva aggiornato con il valore corrente senza attendere che il valore corrente fosse aggiornato sul fronte di salita del clock.

L'uso della variabile adder_exit è stato essenziale per salvare temporaneamente i risultati delle operazioni aritmetiche contenute in READ_ADDRESS e aggiornare i segnali necessari a fine processo.

La scelta di implementare alcuni stati di attesa deriva dalla possibilità di avere al più 20 cicli di clock tra `i_start` = 0 e `o_done` = 1 : avere degli stati di attesa ci ha assicurato di non perdere alcuni riferimenti e valori e ci ha facilitato la gestione di sincronizzazione tra il segnale `o_done` = 1 le uscite asserite ai valori contenuti nei corrispondenti registri.

Infine, i segnali contenenti l'indirizzo di memoria letto dalla sequenza in ingresso sono stati dichiarati come "unsigned", così da evitare problemi di overflow nello stato READ_ADDRESS in cui sono coinvolte operazioni aritmetiche.

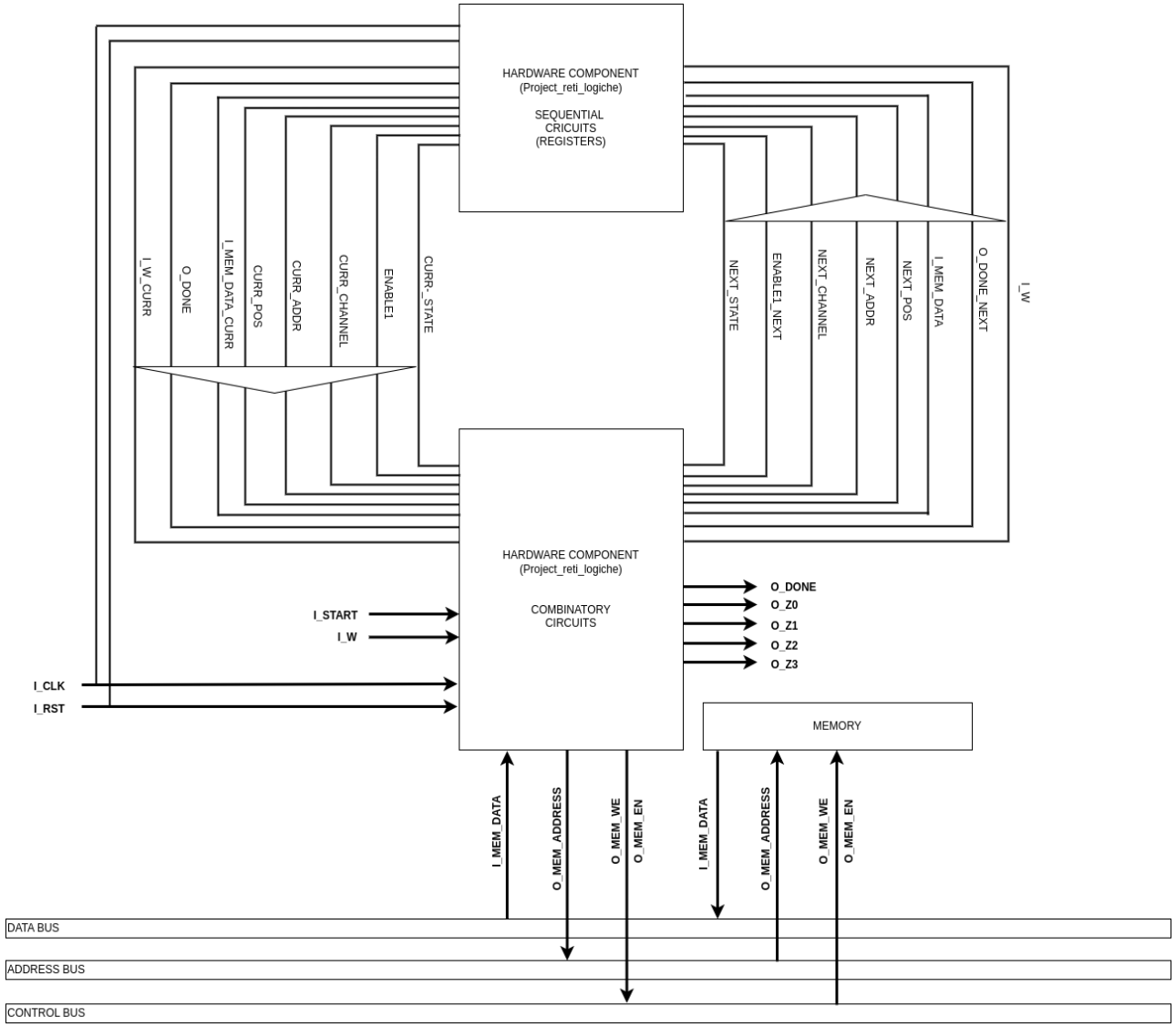


Figura 3: lo schema del componente sequenziale

3.Risultati dei test

Il componente viene sintetizzato correttamente e le simulazioni Behavioral e Functional dei test forniti vengono eseguiti correttamente.
I test dei casi limite sono elencati nel paragrafo 3.2.

3.1 Report di sintesi

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	80	0	134600	0.06
LUT as Logic	80	0	134600	0.06
LUT as Memory	0	0	46200	0.00
Slice Registers	112	0	269200	0.04
Register as Flip Flop	96	0	269200	0.04
Register as Latch	16	0	269200	<0.01
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

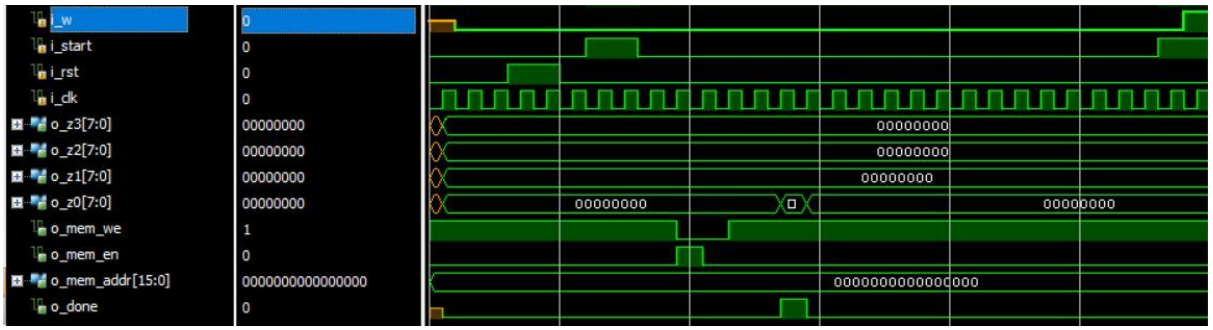
Dopo aver effettuato la sintesi, la logica del componente, come si può notare, è costituita principalmente da Flip Flop e 16 Latch ; questi ultimi sono dovuti ai segnali `curr_addr` e `next_addr` che non vengono assegnati all'inizio del processo della FSM.

3.2 Simulazioni

Oltre a sottoporre il componente ai Testbench forniti dai docenti, abbiamo analizzato dei casi limite che potessero testare la robustezza del progetto.

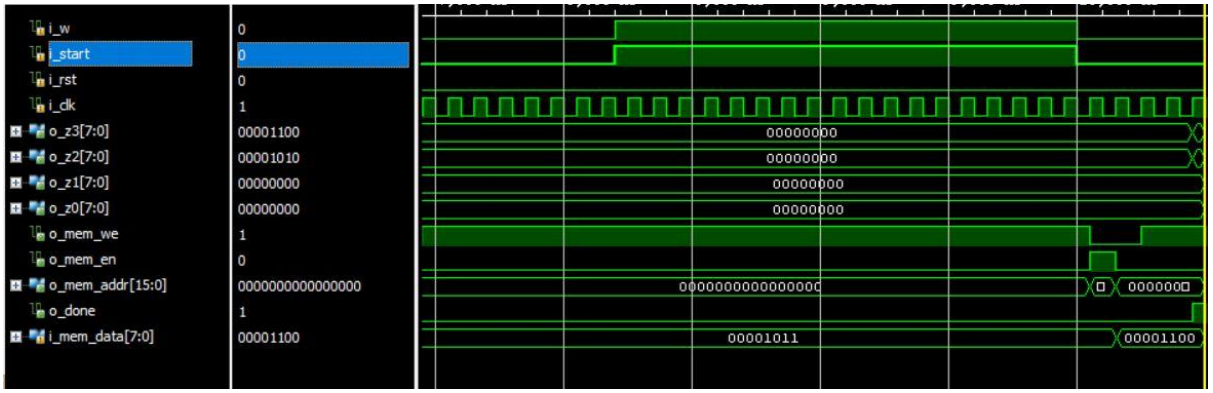
I casi analizzati sono diversi; siamo partiti dall'analisi di sequenze di `i_w` particolari per verificare la corretta lettura e creazione del canale e dell'indirizzo di memoria, come ad esempio:

1. una sequenza di 18 '0':



Inoltre qui abbiamo testato anche il comportamento con il segnale `i_start` alto per 2 soli cicli di clock, dove l'indirizzo non viene modificato e viene esteso con 16 zeri.

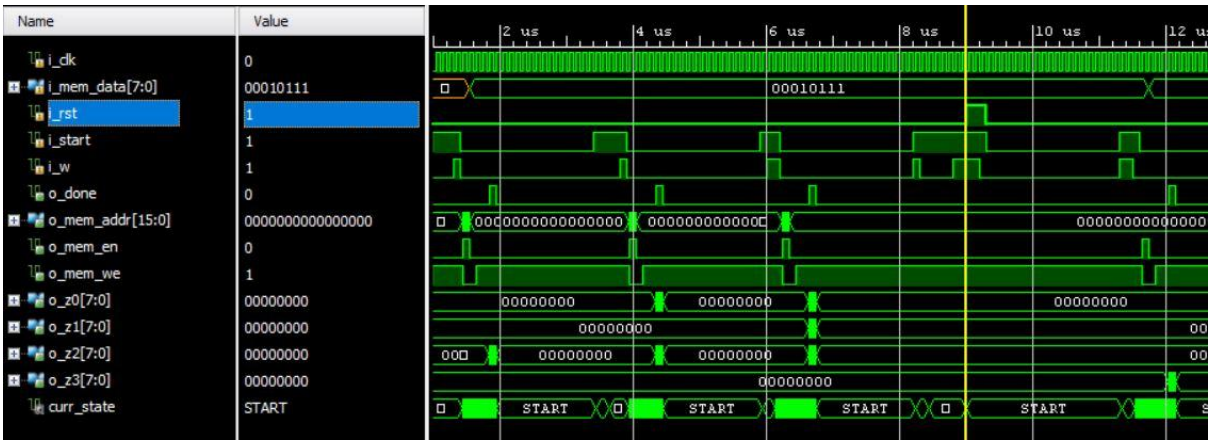
2. una sequenza di solo ‘1’, che testa attivamente il nostro algoritmo di composizione dell’indirizzo per tutti e 16 i cicli di clock;



Infine abbiamo verificato che in caso in cui l’indirizzo letto sia parziale, esso venga esteso correttamente.

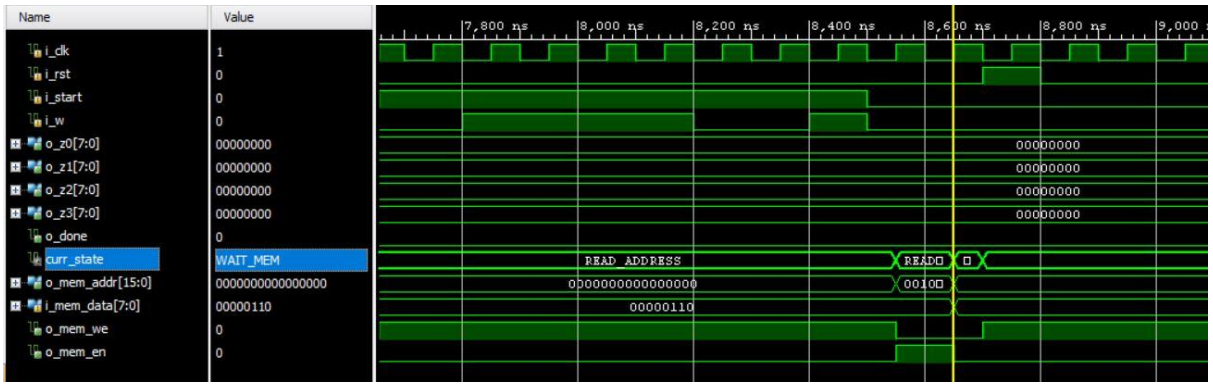
Abbiamo testato poi il funzionamento del componente alla ricezione di un `i_rst` alto in diverse situazioni :

3. `i_rst` = ’1’ con `i_start` = ‘1’ : il componente torna nello stato START e non continua l’elaborazione dell’indirizzo parziale letto, resettando tutti i segnali.

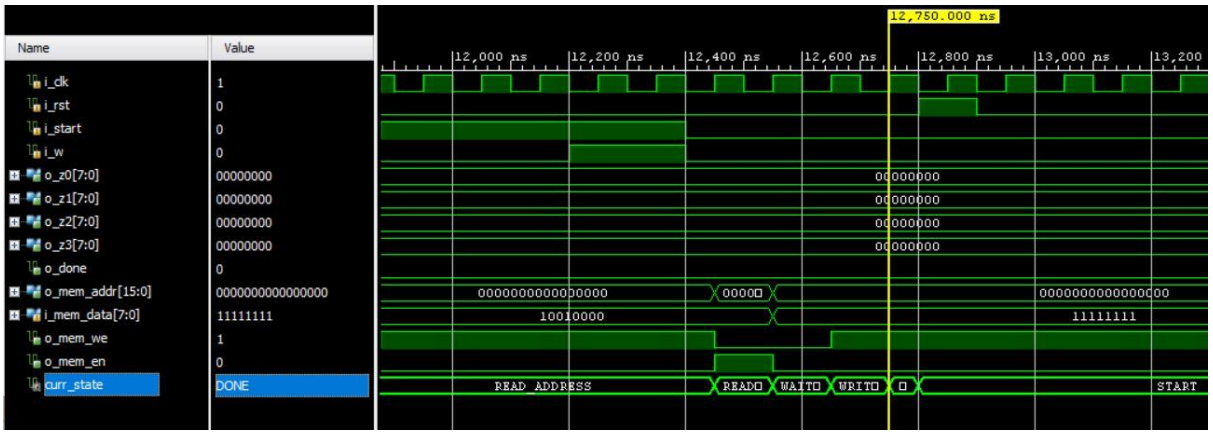


4. `i_rst` = ‘1’ con `i_start` =’0’ prima della salita di `o_done` : siamo nel mezzo della fase di elaborazione del dato, in cui il componente comunica con la memoria. Il reset porta la macchina nello stato START, senza mostrare

le uscite, essendo il reset prima della salita del segnale `o_done`. I segnali vengono resettati.



5. `i_rst` = '1' con `i_start` = '0' in corrispondenza del rising edge di `o_done` : il reset diventa alto appena il segnale `curr_state` passa da DONE a LAST, stato in cui `o_done` dovrebbe rimanere alto per un ciclo di clock. Tuttavia il reset porta `curr_state` a START, e di conseguenza `o_done` rimane basso. Le uscite non vengono mostrate e i segnali vengono resettati.



Abbiamo anche testato il caso in cui `i_rst` = '1' con `i_start` = '0' e `o_done` = '1' : `o_done` torna a '0' senza aspettare la fine del ciclo di clock, le uscite, analogamente, sono in chiaro solo nell'intervallo di tempo in cui `o_done` rimane alto. Il componente torna nello stato START e i segnali vengono resettati come da specifica.
Infine abbiamo sottoposto il componente a dei test randomici non limite, generati da uno script python, per testare ulteriormente le funzionalità attese e la robustezza del progetto.

4. Conclusioni

Il componente si comporta come da specifica in pre e post sintesi, e supera sia i Testbench forniti dal docente che quelli da noi ideati e implementati per stressarlo in situazioni limite.

Le difficoltà inizialmente riscontrate nello sviluppo di questo progetto sono state sia durante la stesura del codice vhdl, dove ci è voluto un po' di tempo per familiarizzare con la logica di programmazione hardware, evitando di implementare tecniche tipiche della programmazione software, sia nella fase di testing post-sintesi, dove abbiamo incontrato alcuni errori che ci hanno portato a modificare parte del codice e del modello.