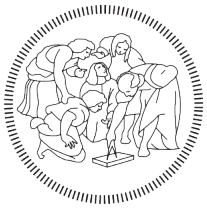


POLITECNICO
MILANO 1863

Progetto Reti Logiche

Christian Biffi 10787158
Michele Cavicchioli 10706553

Anno accademico 2022/2023

Indice

1 Introduzione

1.1 Scopo del progetto

Il progetto consiste nel creare un sistema in grado di leggere dalla memoria un dato e riportarlo sull'uscita selezionata tra le quattro disponibili, ricevendo in maniera sequenziale sia la codifica corrispondente all'uscita su cui mostrare il dato, sia l'indirizzo di memoria da cui andare a leggere il dato.

1.2 Specifiche generali

Il sistema leggerà l'input tramite il segnale `i_w` solamente quando anche il segnale `i_start` sarà alto. La specifica ci assicura che il segnale `i_start` rimarrà alto per un minimo di 2 cicli di clock, fino ad un massimo di 18 cicli di clock.

Siccome abbiamo quattro possibili uscite (Z0, Z1, Z2, Z3) ci bastano 2 bit per codificare la scelta dell'uscita. I primi due bit letti tramite `i_w` rappresentano l'uscita selezionata, da lì in poi finchè `i_start` rimane alto, per un massimo di 16 cicli di clock, tutti i bit andranno a comporre l'indirizzo di memoria da cui leggere il dato.

Nel momento in cui la computazione è terminata bisogna rendere alto il segnale `o_done` e mantenerlo per un ciclo di clock. Contemporaneamente dobbiamo mostrare sull'uscita selezionata il dato letto dalla memoria.
Nel caso di letture successive, ogni volta che il segnale `o_done` diventa alto bisogna mostrare su ogni uscita l'ultimo valore letto dalla memoria corrispondente.
Durante la lettura e la computazione invece, ovvero quando `o_done` rimane basso, sulle uscite andrà mostrato il valore 0.

È inoltre presente il segnale `i_rst` che, nel caso in cui sia alto, resetta il sistema allo stato iniziale, eliminando quindi anche gli eventuali dati letti in precedenza e salvati nelle uscite.

1.3 Interfaccia del componente

```
entity project_reti_logiche is
    port (
        i_clk : in std_logic;
        i_rst : in std_logic;
        i_start : in std_logic;
        i_w : in std_logic;

        o_z0 : out std_logic_vector(7 downto 0);
        o_z1 : out std_logic_vector(7 downto 0);
        o_z2 : out std_logic_vector(7 downto 0);
        o_z3 : out std_logic_vector(7 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_we : out std_logic;
        o_mem_en : out std_logic
    );
end project_reti_logiche;
```

`i_clk`	È il segnale di `CLOCK` dato dal testbench
`i_rst`	È il segnale di `RESET` che riporta allo stato iniziale
`i_start`	È il segnale di `START` generato dal testbench per indicare l'inizio della sequenza di dati da elaborare
`i_w`	È il segnale di ingresso della sequenza dei dati che il componente dovrà elaborare
`o_z0`	È il vettore di segnali relativo all'uscita di *Z0*
`o_z1`	È il vettore di segnali relativo all'uscita di *Z1*
`o_z2`	È il vettore di segnali relativo all'uscita di *Z2*
`o_z3`	È il vettore di segnali relativo all'uscita di *Z3*
`o_done`	È il segnale che indica quando la computazione è terminata e i dati letti dalla memoria sono disponibili sulle uscite
`o_mem_addr`	È il vettore di segnali che manda l'indirizzo alla memoria
`i_mem_Data`	È il vettore di segnali letti dall'indirizzo di memoria
`o_mem_we`	È il segnale di `WRITE ENABLE` da mandare alla memoria per poter abilitare la scrittura. In caso di lettura deve essere posto a 0
`o_mem_en`	È il segnale di `ENABLE` da mandare alla memoria per poter comunicare

1.4 Esempio

Possiamo vedere il funzionamento con il seguente esempio:

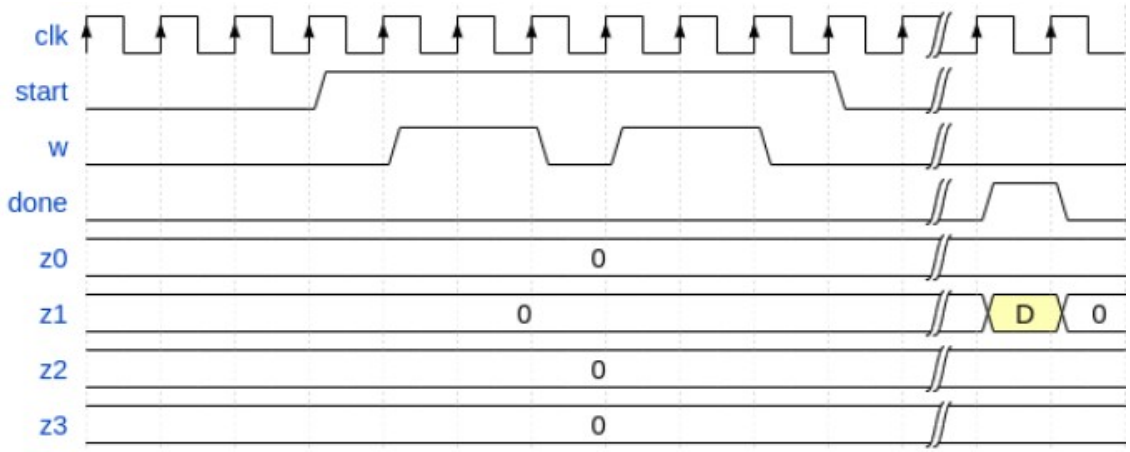


Figure 1: Schema di esempio del funzionamento

Quando il segnale `start` inizia ad essere alto, i primi due bit di `w` rappresentano l'uscita selezionata, in questo esempio l'uscita Z1. Tutti i restanti bit di `w` letti fino a che `start` rimane alto vanno a comporre l'indirizzo di memoria, ovvero `10110` che shiftato su 16 bit diventa `0000000000010110`. Da questo indirizzo andiamo a leggere il dato D che verrà mostrato sull'uscita Z1 quando `done` sarà alto. Successivamente dopo un ciclo di clock, `done` deve ritornare a 0, così come tutte le uscite.

2 Architettura

L'architettura che abbiamo implementato per il progetto è suddivisa in tre componenti principali:

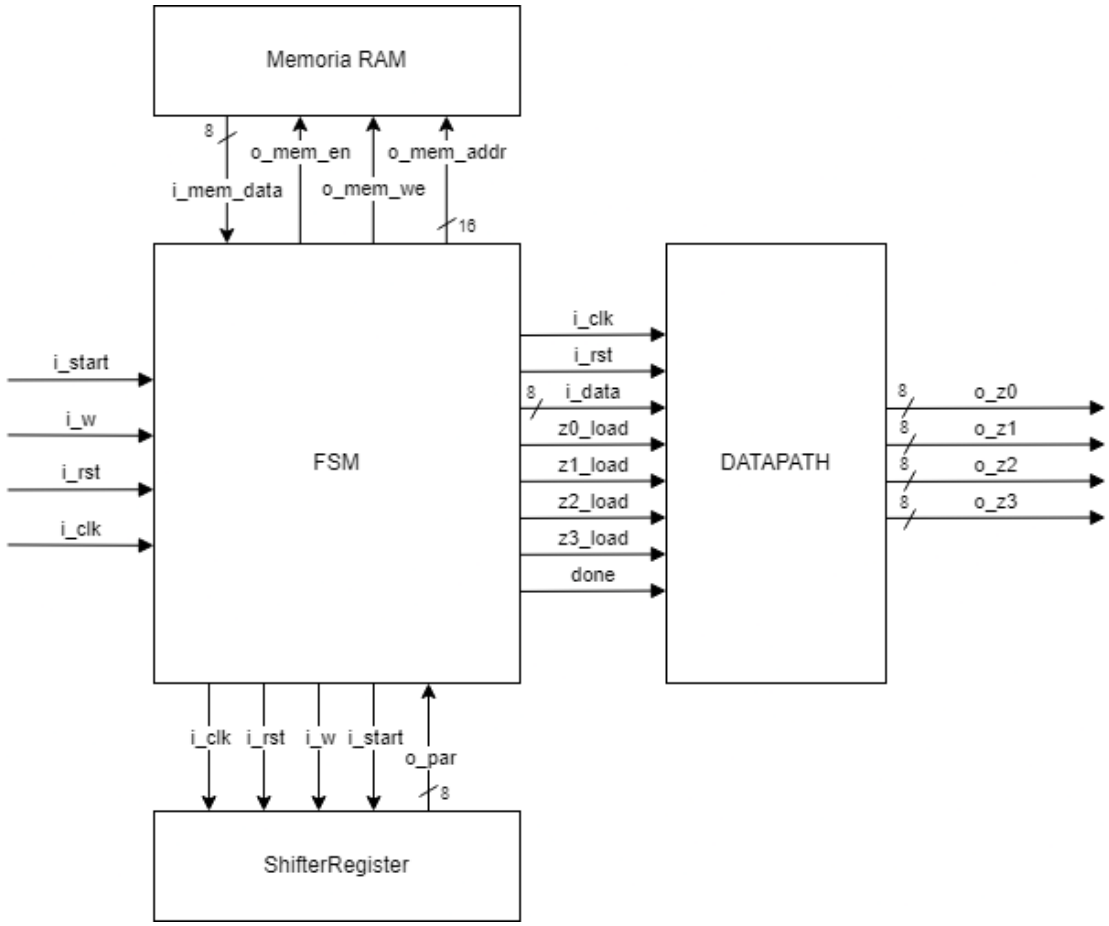


Figure 2: Schema dell'architettura interna del componente

2.1 Entità

2.1.1 Generale

Questa è l'entità principale del progetto, essa presenta le entrate e le uscite essenziali per l'uso del componente. Inoltre, contiene i processi per la gestione della macchina a stati finiti che a sua volta gestisce i segnali per il funzionamento dello shifter e della parte combinatoria.

Di seguito riportiamo i segnali interni utilizzati da questo componente:

`S`	È l'enumerazione degli stati della FSM
`curr_state`	È il segnale per rappresentare lo stato corrente della FSM
`s_rst`	È il segnale di reset dello *ShifterRegister*
`s_addr`	È il vettore di segnali relativo all'indirizzo di memoria per l'interrogazione alla RAM. Corrisponde a *o_par* di *ShifterRegister*
`z0_load`	È il segnale di load per il registro relativo all'uscita *Z0*
`z1_load`	È il segnale di load per il registro relativo all'uscita *Z1*
`z2_load`	È il segnale di load per il registro relativo all'uscita *Z2*
`z3_load`	È il segnale di load per il registro relativo all'uscita *Z3*
`d_sel`	È il vettore di segnali che rappresenta la selezione del canale di uscita
`d0_load`	È il segnale di load per scrivere sul LSB di *d_sel*
`d1_load`	È il segnale di load per scrivere sul MSB di *d_sel*
`done`	È il segnale che notifica che le uscite sono aggiornate

2.1.2 ShifterRegister

L'entità *ShifterRegister* la usiamo per implementare uno shift register sinistro. Questo shifter ci serve per gestire l'indirizzo di memoria che verrà poi usato per eseguire l'interrogazione alla memoria RAM. La sua implementazione è molto semplice, è composta da un singolo processo che ad ogni fronte di salita del clock verifica se il segnale di reset è attivo e in tal caso il registro viene resettato a 0, altrimenti, se il segnale `i_start` è alto si shifta il registro verso sinistra inserendo poi il bit in ingresso nel bit meno significativo (LSB).

```
entity ShifterRegister
    is port (
        i_clk : in std_logic;
        i_rst: in std_logic;
        i_w : in std_logic;
        i_start: in std_logic;
        o_par : out std_logic_vector(15 downto 0)
    );
end ShifterRegister;
```

`temp`	È il vettore di segnali usato temporaneamente per effettuare lo shift

2.1.3 Datapath

L'entità *Datapath* ha l'obiettivo di descrivere e implementare la parte combinatoria del progetto. Questa sezione gestisce l'output prendendo dalla memoria RAM il dato in uscita e inserendolo nel registro appropriato. I registri per il salvataggio dei dati correnti prevedono un segnale di *load* che ci permette di decidere quando vogliamo cambiare il dato al loro interno, tali segnali sono gestiti dalla FSM. Infine per occuparci dell'uscita abbiamo un processo dedicato che, se il segnale *done* è alto, imposta le uscite Z_i al valore caricato nel registro corrispondente, altrimenti 0.

```
entity datapath
    is port (
        i_clk : in std_logic;
        i_rst : in std_logic;
        done : in std_logic;
        i_data : in std_logic_vector(7 downto 0);
        z0_load : in std_logic;
        z1_load : in std_logic;
        z2_load : in std_logic;
        z3_load : in std_logic;
```

```
            o_z0 : out std_logic_vector(7 downto 0);
            o_z1 : out std_logic_vector(7 downto 0);
            o_z2 : out std_logic_vector(7 downto 0);
            o_z3 : out std_logic_vector(7 downto 0);
            o_done : out std_logic
        );
    end datapath;
```

Z0	È il vettore di segnali per rappresentare il valore più recente salvato sul canale *Z0*
Z1	È il vettore di segnali per rappresentare il valore più recente salvato sul canale *Z1*
Z2	È il vettore di segnali per rappresentare il valore più recente salvato sul canale *Z2*
Z3	È il vettore di segnali per rappresentare il valore più recente salvato sul canale *Z3*

2.2 Macchina a stati finiti

La Macchina a Stati Finiti si occupa di gestire lo stato della computazione all'interno del nostro sistema. Viene suddivisa in sei stati differenti, come è possibile vedere nel seguente schema:

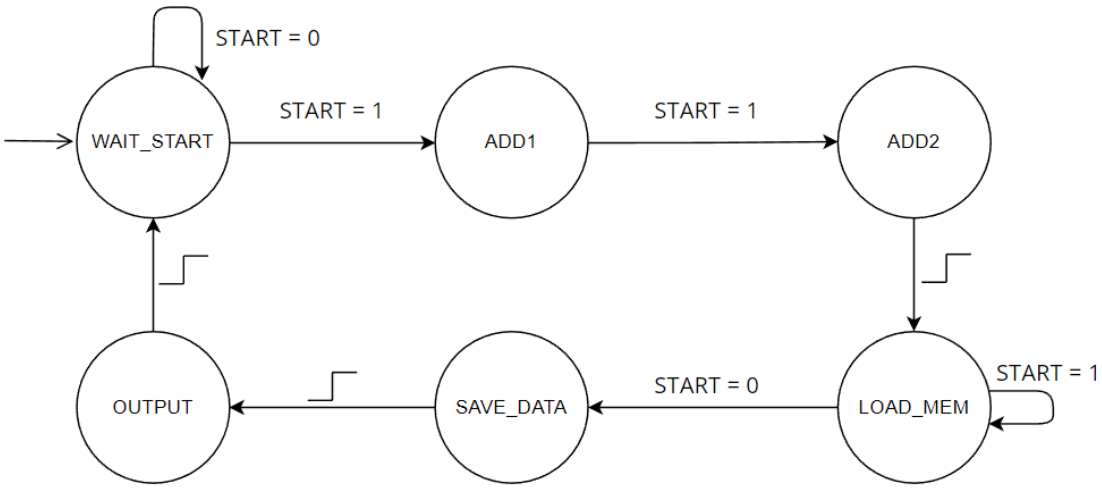


Figure 3: Schema della macchina a stati finiti

2.2.1 *WAIT_START*

Stato iniziale della FSM, viene usato come stato di attesa per la salita del segnale `i_start`, che identifica l'inizio della sequenza di input valida. Inoltre torniamo in questo stato ogni volta che il segnale `i_rst` è alto.

2.2.2 *ADD1*

Il ruolo di questo stato è quello di leggere il primo bit del canale di uscita selezionato.

2.2.3 *ADD2*

Questo stato legge il secondo e ultimo bit di selezione del canale di uscita.

2.2.4 *LOAD_DATA*

In questo stato ci occupiamo di gestire l'input dell'indirizzo di memoria bit per bit, utilizzando lo *ShiftRegister* per salvare correttamente l'input.

2.2.5 *SAVE_DATA*

Lo stato ha il compito di andare ad interrogare la memoria e gestire la sua risposta. In particolare andiamo a salvare il dato preso dalla memoria all'interno del registro di uscita selezionato.

2.2.6 *OUTPUT*

Questo è l'ultimo stato della nostra FSM, si occupa di alzare il segnale `o_done` e quindi di mostrare sulle uscite i dati precedentemente salvati. Da questo stato si ritorna allo stato iniziale *WAIT_START*, in modo da essere pronti per leggere una nuova sequenza.

3 Risultati

3.1 Sintesi

Effettuando la sintesi del componente si ottengono i seguenti risultati:

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	17	0	0	134600	0.01
LUT as Logic	17	0	0	134600	0.01
LUT as Memory	0	0	0	46200	0.00
Slice Registers	89	0	0	269200	0.03
Register as Flip Flop	89	0	0	269200	0.03
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

Nello sviluppo del componente abbiamo posto particolare attenzione a non inferire dei latch, i quali avrebbero portato ad un comportamento anomalo e non voluto.
Lo schema della sintesi è il seguente:

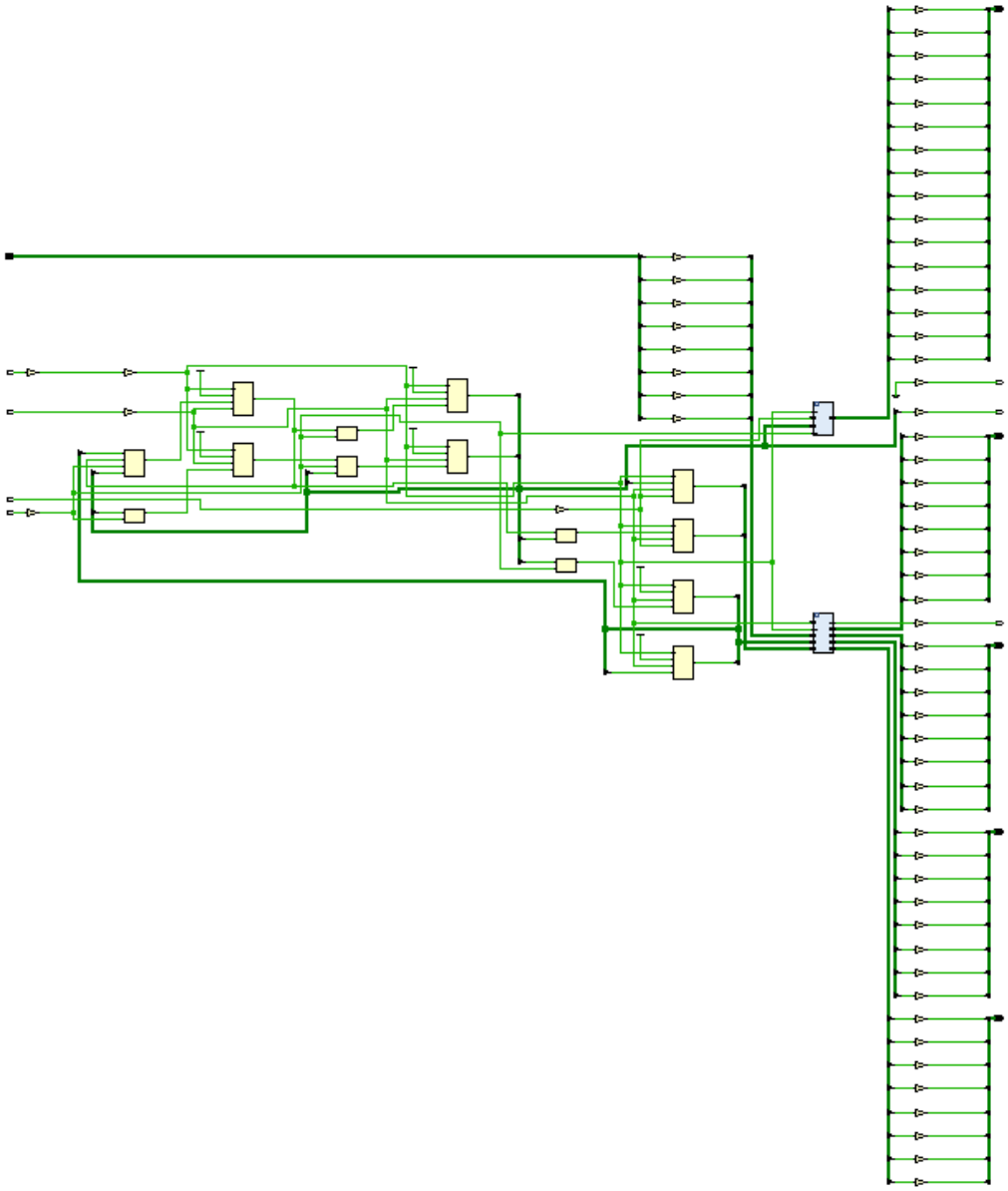


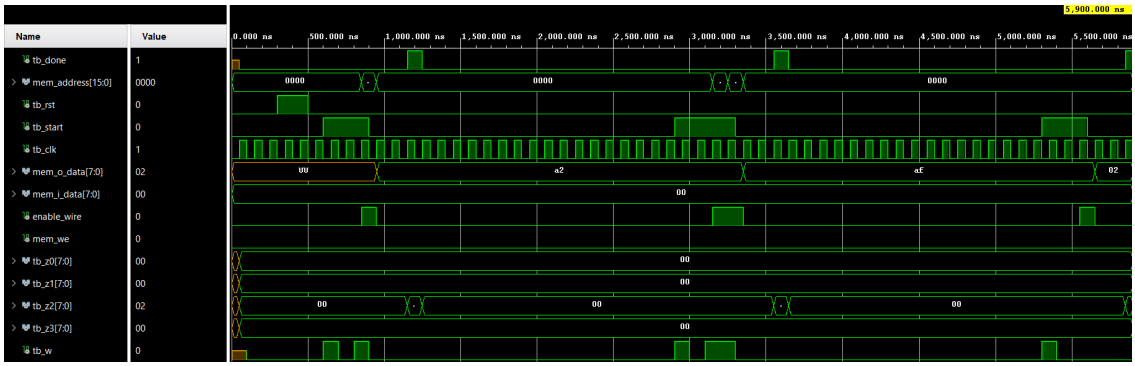
Figure 4: Schema di sintesi del componente

3.2 Simulazioni

Oltre ad utilizzare il testbench fornito dai docenti, abbiamo creato diversi testbench che simulassero i casi limite del componente in modo da verificare il corretto funzionamento. Il componente sviluppato passa correttamente i test sia in *Behavioral Simulation* che in *Post-Synthesis Functional Simulation*.

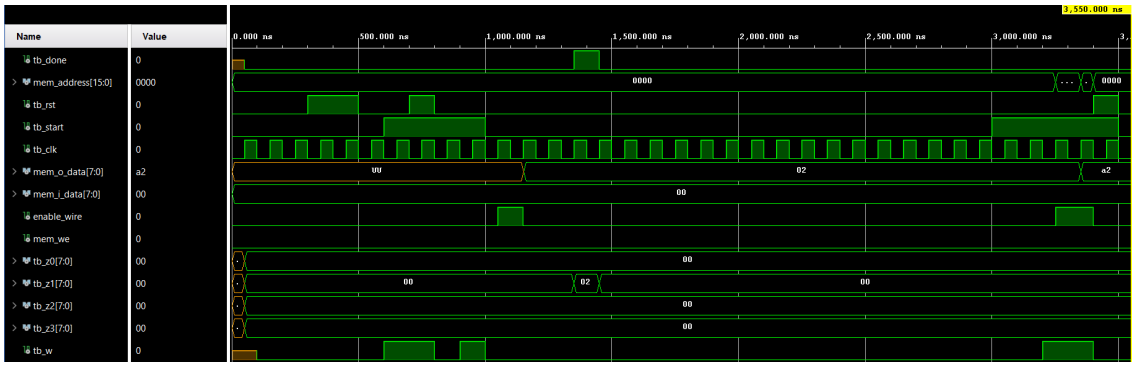
3.3 Test 1: Sovrascrittura di un registro

L'obiettivo di questo testbench è quello di testare il corretto funzionamento nel caso in cui si vada a sovrascrivere un registro ripetutamente con dati differenti.



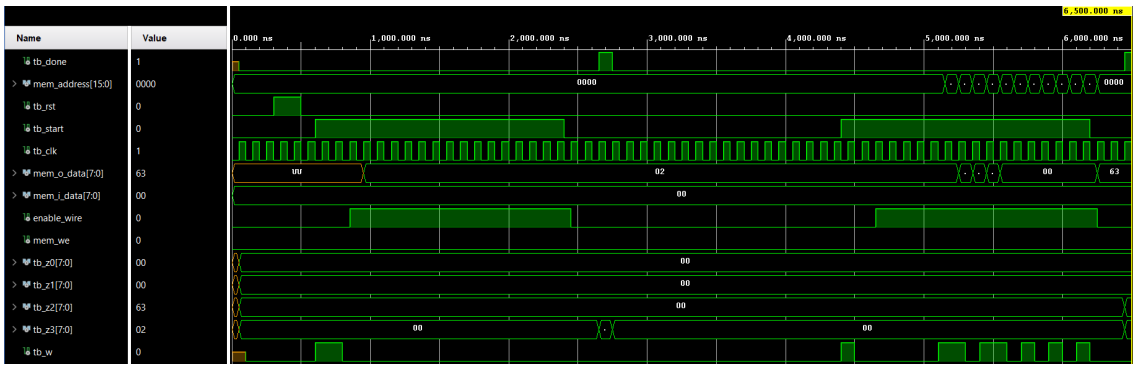
3.4 Test 2: Segnale di reset durante la lettura

Qui andiamo a testare il caso in cui il segnale di reset sia alto mentre si stanno leggendo dei dati, ovvero quando `i_start` è alto. Andiamo sia a verificare che subito dopo il reset il resto dei dati vengano letti correttamente, sia che subito dopo il reset le uscite siano riportate a 0.



3.5 Test 3: Lettura input di lunghezza massima

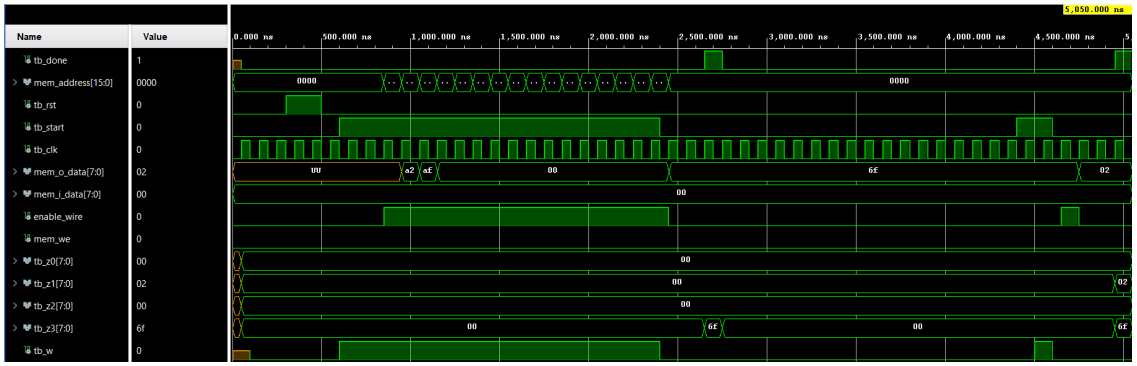
Con questo testbench abbiamo testato il caso in cui `i_start` rimanga alto per il numero massimo di cicli di clock, passando quindi indirizzi di memoria di lunghezza massima. In particolare poi abbiamo verificato che funzionasse correttamente sia passando come bit di memoria tutti 0, sia con un indirizzo di memoria generico.



3.6 Test 4: Casi limite input

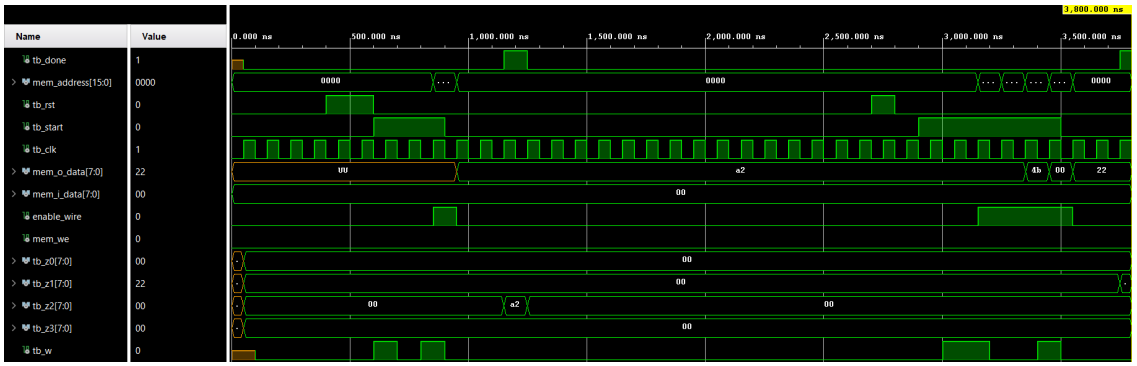
In questo test andiamo a verificare il corretto funzionamento nel caso in cui vengano passati tutti 1, sia come uscita che come indirizzo di memoria e come secondo test anche il corretto funzionamento

nel caso in cui venga solo selezionata l'uscita senza indicare nessun indirizzo di memoria da cui leggere il dato.



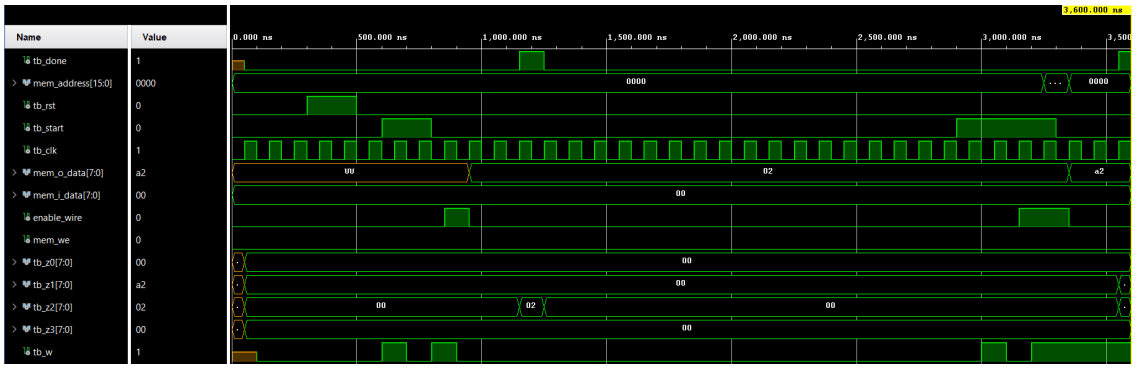
3.7 Test 5: Reset con start consecutivo

Con questo test verifichiamo il corretto funzionamento del componente nel caso in cui venga alzato il segnale di start consecutivamente ad un reset. Come secondo caso di test valutiamo invece un semplice reset per verificare che tutti i registri vengano resettati correttamente.



3.8 Test 6: Corretta gestione degli input con start basso

Il seguente testbench ha il compito di verificare che nel caso in cui start è 0 qualsiasi tipo di input su *i_w* venga ignorato dal componente.



4 Conclusione

Siamo partiti inizialmente da una macchina a stati finiti con 8 stati diversi e con diverse modifiche al codice siamo riusciti ad ottimizzarlo riducendo gli stati a 6. Inoltre abbiamo cercato di mantenere il codice chiaro e pulito dividendolo in componenti diversi, ognuno con uno scopo preciso.
Per concludere riteniamo che il componente sviluppato soddisfi a pieno la specifica, superando i diversi test proposti in *Behavioral Simulation* e in *Post-Synthesis Functional Simulation*.