Anno Accademico: 2022-2023

POLITECNICO
MILANO 1863

Relazione Prova Finale (Progetto di Reti Logiche)

PROF. FORNACIARI WILLIAM

ALESSANDRO CONTI CODICE PERSONA: 10710583 MATRICOLA: 955525

FEDERICO DE INTRONA CODICE PERSONA: 10796946 MATRICOLA: 960696

Indice

1 Introduzione

1.1 Scopo del Progetto

Lo scopo del progetto è l'implementazione di un componente hardware, descritto mediante il linguaggio di programmazione VHDL, in grado di interfacciarsi con una memoria per leggere i dati e mandarli in uno dei canali di uscita possibili.

1.2 Specifiche Generali

Il modulo hardware da progettare ha in ingresso, oltre ai segnali di *CLOCK* e *RESET*, due segnali denominati *W* e *START* e cinque uscite *Z0*, *Z1*, *Z2*, *Z3* e *DONE*.

Il segnale d'ingresso *W* è un ingresso primario seriale formato da una sequenza di bit così organizzata:

i primi due formano l'intestazione, cioè su quale canale (Z) indirizzare il valore da mostrare,
i restanti N bit formano l'indirizzo della memoria da cui leggere il dato da mostrare in uscita.

L'ingresso primario seriale W è valido fin tanto che il segnale d'ingresso *START* è alto e termina la validità quando quest'ultimo è basso.

Dopo che il segnale *START* è ritornato basso (cioè è passato da 1 a 0), il componente richiede alla memoria il valore che è salvato all'indirizzo ricevuto in ingresso; successivamente il dato viene letto e mandato nel canale d'uscita (Z) selezionato tramite i primi due bit dell'ingresso primario *W*.

1.3 Interfaccia del Componente

Il componente ha la seguente interfaccia:

```
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;
```

in particolare:

- *i_clk* è il segnale di CLOCK generato dal Test Bench;
- *i_rst* è il segnale di RESET generato dal Test Bench;
- *i_start* è segnale di START generato dal Test Bench;
- *i_w* è il segnale di W generato dal Test Bench;
- *o_z0* è il canale di uscita 0;
- *o_z1* è il canale di uscita 1;
- *o_z2* è il canale di uscita 2;
- *o_z3* è il canale di uscita 3;
- *o_done* è il segnale di uscita che comunica la fine dell'elaborazione;
- *o_mem_addr* è il segnale che arriva alla memoria per comunicare l'indirizzo;
- *i_mem_data* è il segnale che arriva al componente per comunicare il valore dell'indirizzo di memoria interessato
- *o_mem_we* è il segnale di WRITE ENABLE da mandare alla memoria per poterci scrivere;
- *o_mem_en* è il segnale di ENABLE da mandare alla memoria per poter comunicare con essa in scrittura e in lettura.

1.4 Dati in Ingresso e Descrizione della Memoria

Gli ingressi primari sono *W* e *START* e hanno dimensione di 1 bit, mentre quattro delle cinque uscite, *Z0, Z1, Z2, Z3*, hanno dimensione di 8 bit e l'ultima, *DONE*, di 1 bit.

I dati letti dalla sequenza d'ingresso W hanno dimensione variabile.

L'insieme dei bit, che arrivano dalla sequenza *W*, è compreso tra 2 e 18; questo è dovuto al fatto che l'ingresso primario seriale ha una determinata organizzazione.
I primi due bit servono da intestazione, cioè definiscono in quale canale d'uscita mandare il dato ottenuto dalla memoria. Gli altri N-2 bit permettono di costruire un indirizzo di memoria.

L'indirizzo della memoria è composto da 16 bit, mentre i dati che vi sono contenuti sono di 8.

Se dall'ingresso *W* arriva un indirizzo più corto dei 16 bit richiesti, vengono aggiunti, sul bit più significativo, tanti 0 fino a raggiungere la dimensione corretta.

2 Scelte Progettuali

2.1 Descrizione Generale

Il nostro componente funziona mediante l'algoritmo di una *Finite State Machine* (FSM) la quale descrive in ogni stato le azioni da compiere.

Inizialmente la FSM aspetta che *START* diventi alto, a questo punto leggerà i primi due bit dalla sequenza in ingresso da W, che indicheranno a quale canale d'uscita mandare il dato letto dalla memoria; fatto ciò continuerà a leggere, sempre dall'ingresso *W*, l'indirizzo di memoria in qui è salvato il dato.

Quando il segnale di *START* si abbassa la FSM permette di mandare l'indirizzo alla memoria; fatto ciò si aspetta un ciclo di clock per permettere alla memoria di elaborare la richiesta.

Subito dopo si leggerà il dato che la memoria ha elaborato e lo si scriverà nel canale selezionato precedentemente, alzando il segnale di uscita *DONE* a 1.

2.2 Scelte Implementative

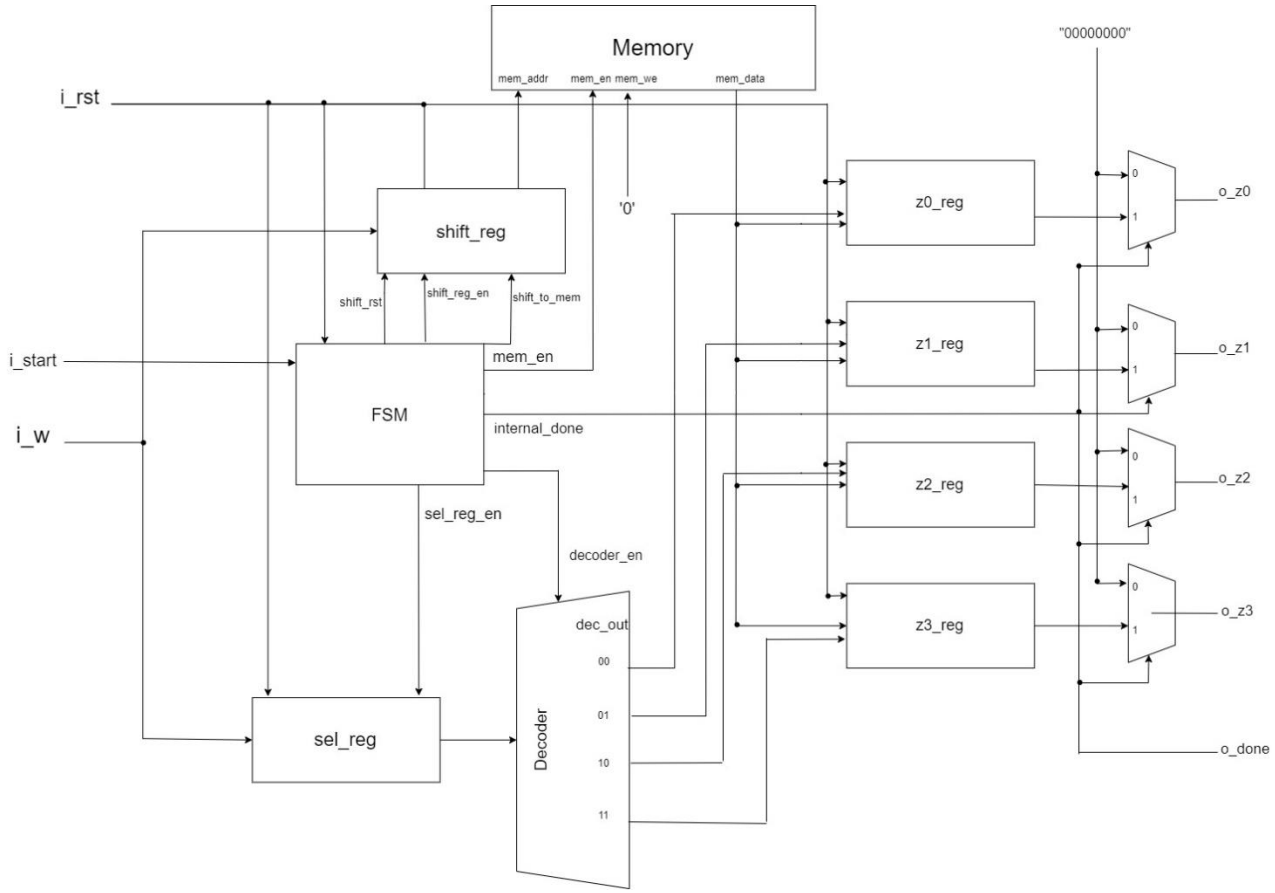

Figura 1 Disegno del componente

Il componente hardware, da noi implementato, è formato da processi che descrivono la logica sequenziale e da altri che si occupano della gestione della FSM, il cambio di stato e l'output da mostrare.

I primi processi, quelli che gestiscono la logica sequenziale, servono per permettere il salvataggio e l'utilizzo dei dati da parte dei registri; gli altri, che si occupano di governare la FSM, determinano lo stato prossimo e i valori dei segnali interni al modulo hardware, partendo dallo stato corrente e da i segnali in ingresso.

I registri interni utilizzati sono:

- **sel_reg**: è uno *shift-register* e serve per salvare i primi due bit della sequenza di *W*, ha come segnale interno il solo *sel_reg_en* che serve per decidere quando salvare i bit del canale di output.
- **shift_reg**: è uno *shift-register* e viene utilizzato per salvare i bit dell'indirizzo di memoria interessato, presi dalla sequenza di *W* e ha come segnali interni:
  - *shift_rst* che viene utilizzato per resettare l'indirizzo di memoria interessato dopo ogni fine lettura dalla memoria;

- *shift_reg_en* che viene utilizzato per decidere quando salvare i bit dell'indirizzo di memoria interessato;
- *shift_to_mem* che viene utilizzato per decidere quando inviare l'indirizzo di interesse alla memoria.

- **z0_reg**, **z1_reg**, **z2_reg**, **z3_reg**: sono *parallel-register* e servono per salvare i dati letti dalla memoria; hanno come segnale interno il solo *dec_out* che viene utilizzato per abilitare la scrittura sul registro in uscita.

Si è deciso di utilizzare un decoder per selezionare correttamente il canale d'uscita a cui mandare il dato letto dalla memoria. Viene utilizzato il segnare di *decoder_en* per abilitare la scelta; in ingresso troviamo i primi due bit letti dalla sequenza valida di *W*, salvati precedentemente nel registro *sel_reg*.

2.3 Stati Della FSM

La nostra FSM è una *macchina di Moore*:

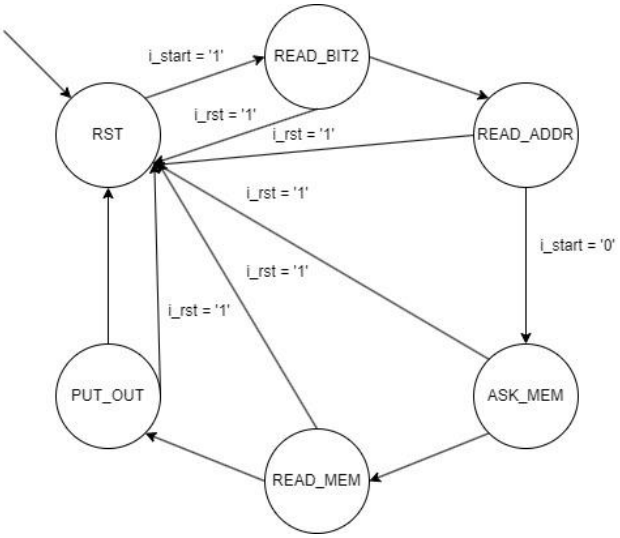

Figura 2 Grafico della FSM (senza i valori di output)

La macchina ideata e implementata da noi è formata da 6 stati; di seguito viene riportato una breve descrizione per ognuno di essi:

1. *RST*: è lo stato iniziale, aspetta che START passi da 0 a 1, e salva il primo bit dell'intestazione; serve anche a resettare il componente, cioè portarlo nella condizione iniziale;
2. *READ_BIT2*: è lo stato che legge il secondo bit d'intestazione;
3. *READ_ADDR*: è lo stato che legge da W tutti i bit dell'indirizzo di memoria d'interesse; si resta in questo stato fintanto che il segnale *START* è alto;
4. *ASK_MEM*: è lo stato che manda l'indirizzo salvato alla memoria; in questo stato ci si arriva quando l'ingresso START passa da 1 a 0;
5. *READ_MEM*: è lo stato che permette di salvare il valore ritornato dalla memoria nel giusto canale d'uscita dal componente hardware;
6. *PUT_OUT*: è lo stato che mostra le uscite di tutti i canali e assegna a DONE il valore 1 per un solo ciclo di clock.

Dato che usiamo una *Macchina di Moore*, i valori di output cambiano in base allo stato in cui si trova e i valori sono i seguenti:

```
internal_done <= '0';
```

```
o_done <= '0';
o_mem_we <= '0';
o_mem_en <= '0';
shift_rst <= '0';
shift_reg_en <= '0';
shift_to_mem <= '0';
sel_reg_en <= '0';
decoder_en <= '0';
case curr_state is
      when RST =>  sel_reg_en <= '1';
         shift_rst <= '1';
      when READ_BIT2 => sel_reg_en <= '1';
      when READ_ADDR => shift_reg_en <= '1';
      when ASK_MEM => shift_to_mem <= '1';
         o_mem_en <= '1';
      when READ_MEM => decoder_en <= '1';
      when PUT_OUT => o_done <= '1';
      internal_done <= '1';
end case;
```

3 Testing e Risultati

Per verificare il corretto funzionamento del componente sintetizzato, dopo averlo testato con il test banch d'esempio, abbiamo definito altri 11 test, in modo da verificare particolari eventi di comportamento e massimizzare tutti i possibili casi limite.

Di seguito sono riportati tutti gli 11 test; per quelli più significativi, evidenziamo il corretto funzionamento dell'esecuzione mostrando le immagini dell'andamento dei segnali.

Test 0. Indirizzo di memoria è specificato ma non è né vuoto né pieno (tb_example_23_agg).

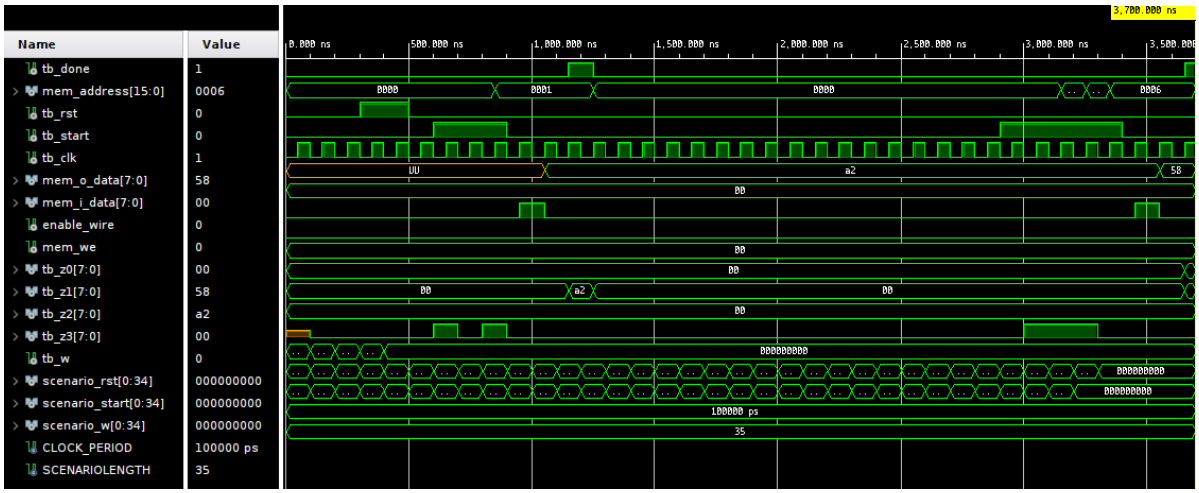

Figura 3 tb_example_23_agg

Test 1. **Indirizzo di memoria vuoto** (START=1 per soli 2 cicli di clock): il test verifica che il componente funzioni quando non viene specificato l'indirizzo di memoria da cui leggere il dato.

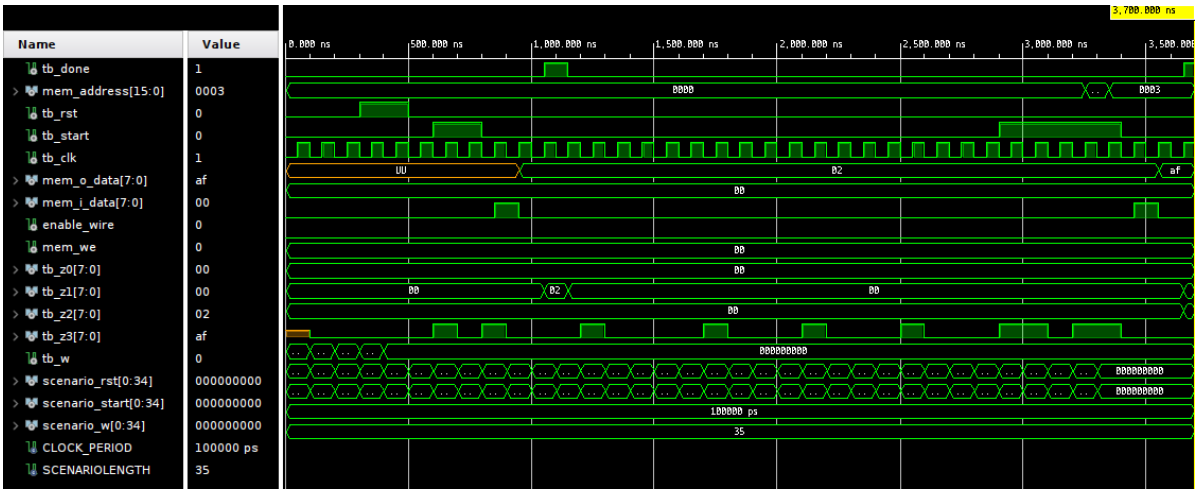

Figura 4 indirizzo_di_memoria_vuoto

Test 2. **Indirizzo di memoria pieno** (START=1 per 18 cicli di clock): il test verifica che il componente funzioni quando viene specificato completamente l'indirizzo di memoria da cui leggere il dato.

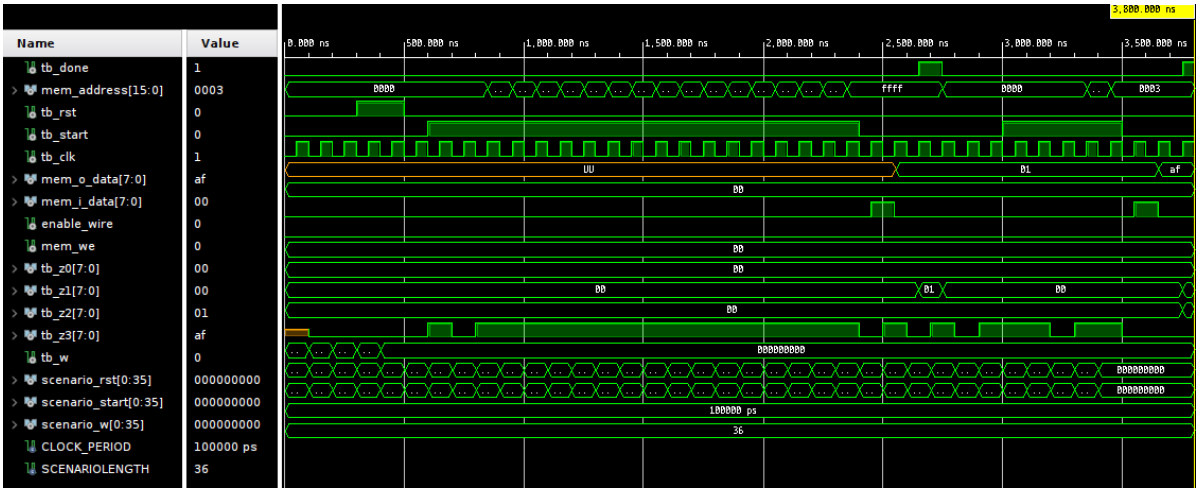

Figura 5 indirizzo_di_memoria_pieno

Test 3. **Scrive su tutte le uscite**: il test verifica che il componente scriva correttamente su tutte le uscite.

Test 4. **Scrive su tutte le uscite e le sovrascrive correttamente**: il test verifica che il componente scriva correttamente su tutte le uscite anche quando queste vengono sovrascritte.

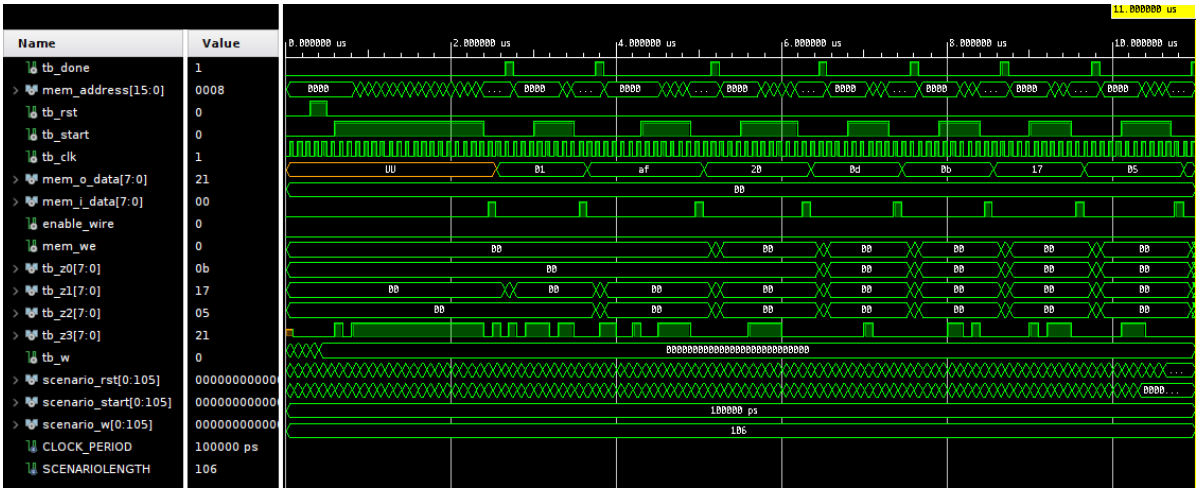

Figura 6 scrive_su_tutte_le_uscite_e_le_sovrascrive

Test 5. **Reset quando start è alto** (RESET=1 quando START=1): il test verifica che il componente si resetti correttamente anche quando START è alto (viene garantito che se START resta alto ci siano almeno altri due cicli di clock).

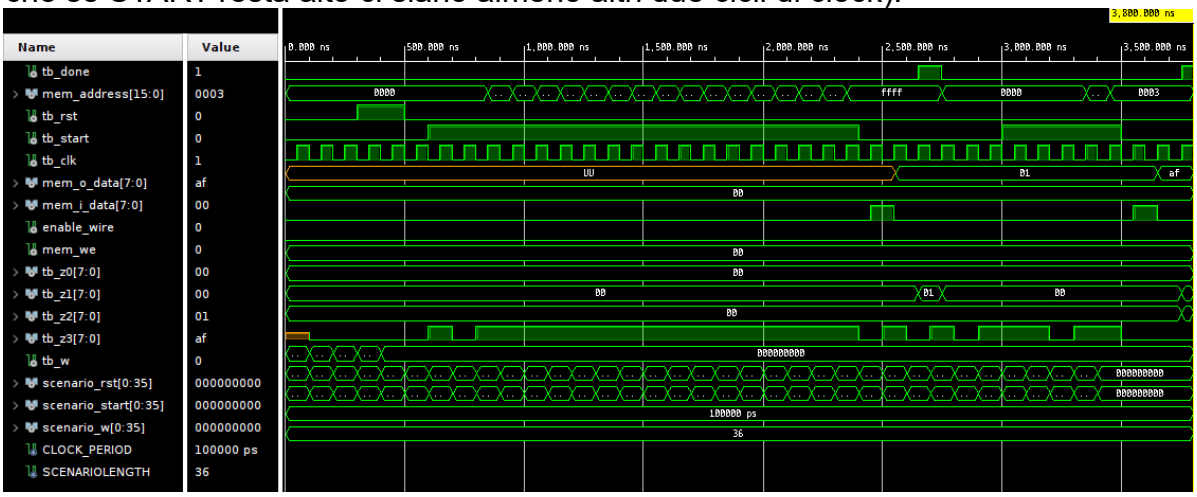

Figura 7 reset_quando_start_alto

Reset per ogni stato della FSM: i test verificano che il componente, in qualunque stato della FSM si trovi, venga resettato correttamente.

- **Test 6.** Per *RST*
- **Test 7.** Per *READ_BIT2*
- **Test 8.** Per *READ_ADDR*
- **Test 9.** Per *ASK_MEM*
- **Test 10.** Per *READ_MEM*
- **Test 11.** Per *PUT_OUT*

Oltre ai test mirati alla ricerca dei casi limite, sopra citati, abbiamo simulato diversi test randomici per testare ulteriormente il componente. Utilizziamo uno *script in python* per generare i test e poi i risultati li ricopiamo in una nuova *simulation sources*.

4 Conclusioni

4.1 Risultati della Sintesi

Il componente sintetizzato supera correttamente tutti i test utilizzati nelle 3 simulazioni: *Behavioural*, *Post-Synthesis Functional* e *Post-Synthesis Timing*.

Di seguito riportiamo un confronto tra i tempi di simulazione, in *behavioural*, dei due *corner case* che portano la macchina verso la più breve e la più lunga simulazione:

- Tempo di simulazione con l'indirizzo di memoria vuoto: **3700ns**
- Tempo di simulazione con l'indirizzo di memoria pieno: **3800ns**

Per avere un parametro che quantifichi, rispetto al tempo totale a disposizione, quale sia il tempo del *path* peggiore si considera il *Worst Negative Slack* riportato nella tabella (fig. 8) *Design Timing Summary* del componente sintetizzato.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	97.009 ns	Worst Hold Slack (WHS):	0.146 ns	Worst Pulse Width Slack (WPWS):	49.500 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:	84	Total Number of Endpoints:	84	Total Number of Endpoints:	54

Figura 8 Design Timing Summary

4.2 Ottimizzazioni

Le ottimizzazioni sono:

- la *riduzione del numero degli stati che compongono la FSM*; dopo l'ottimizzazione la maggior parte degli stati sono "Stati Ponte", cioè vengono utilizzati solamente per fa passare un ciclo di clock;
- l'utilizzo dei registri serie-parallelo al posto dei registri serie-serie in maniera che si riduca il tempo di lettura; facendo ciò i bit che compongono il valore salvato nel registro vengono letti in un unico ciclo di clock;