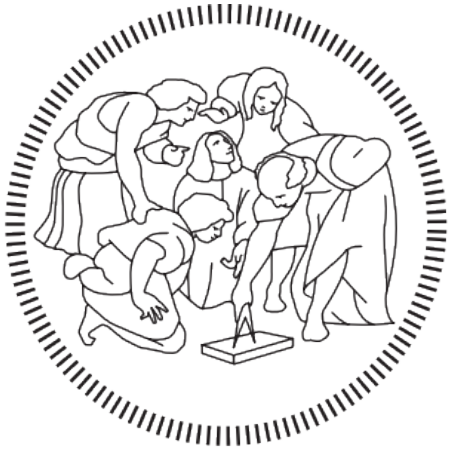


POLITECNICO
MILANO 1863

Prova Finale di Reti Logiche
Anno Accademico 2022/2023

Benelle Francesco
Codice Persona: 10727489 - Matricola: 959528

Cavallotti Alberto
Codice Persona: 10721275 - Matricola: 956304

Indice

1 Introduzione

1.1 Scopo del Progetto e Specifiche

Lo scopo del progetto consiste nel, fornito un input seriale, rappresentare su uno dei quattro canali di uscita il dato contenuto all'indirizzo di memoria fornito in input.
Per lo svolgimento del progetto è fornito un input seriale di dimensione compresa tra i 2 e i 18 bit. I primi due bit, sempre forniti, rappresentano su quale delle quattro uscite (Z0, Z1, Z2, Z3) deve essere rappresentato il dato in questione. Gli altri bit, forniti in una quantità compresa tra 0 e 16 bit, rappresentano l'indirizzo RAM dal quale prendere il dato da mostrare sul canale di output specificato.
La specifica fornitaci per questo progetto riguarda il segnale *i_start*, infatti quest'ultimo rimane alto per almeno 2 cicli di clock e non può rimanere alto per più di 18 cicli di clock e permettere di leggere l'input fornito da *i_w*. Inoltre il segnale di reset (*i_rst*) è un segnale che può diventare 1 in qualsiasi momento a prescindere dal valore di *i_start* o dal segnale di clock.

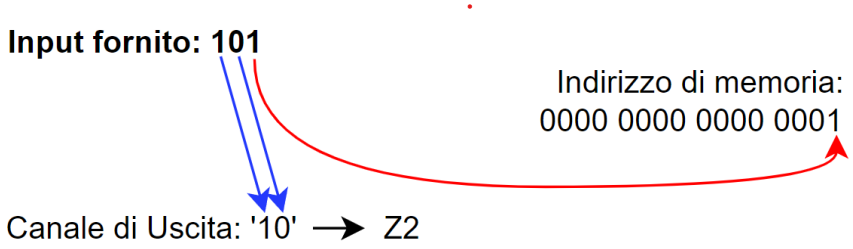


Figure 1: Esempio lettura input

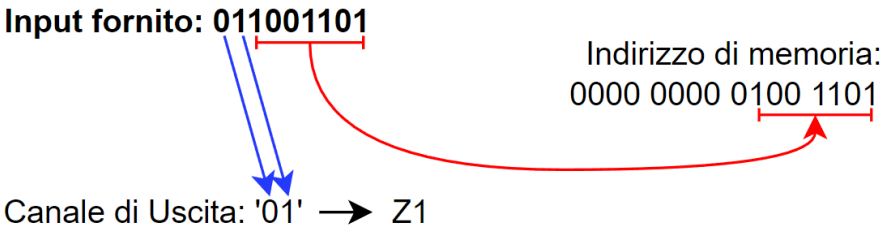


Figure 2: Esempio lettura input

1.2 Interfaccia del Componente

L'interfaccia data per eseguire il progetto è:

```
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;
```

Figure 3: Entity del progetto

In particolare:

- Segnali di **INPUT**:
  - *i_clk* è il segnale di *CLOCK* fornito dal TestBench.
  - *i_rst* è il segnale di *RESET* fornito dal TestBench.
  - *i_start* è il segnale di *START* fornito dal TestBench che regola la lettura dell'input.
  - *i_w* è il segnale che rappresenta l'input seriale da leggere.
  - *i_mem_data* è il segnale (vettore) che contiene il dato che rappresenta nella memoria il segnale passato in input.

- Segnali di **OUTPUT**:
  - *o_z0* è il segnale (vettore) dell'uscita Z0 in cui verrà mostrato il dato in output.
  - *o_z1* è il segnale (vettore) dell'uscita Z1 in cui verrà mostrato il dato in output.
  - *o_z2* è il segnale (vettore) dell'uscita Z2 in cui verrà mostrato il dato in output.
  - *o_z3* è il segnale (vettore) dell'uscita Z3 in cui verrà mostrato il dato in output.
  - *o_done* è il segnale che notifica la fine dell'elaborazione dell'input.
  - *o_mem_we* è il segnale di *WRITE_ENABLE* da mandare alla memoria per poter scrivere.
  - *o_mem_en* è il segnale di *ENABLE* da mandare alla memoria per poter comunicare.

2 Architettura

2.1 Scelte Progettuali

Dato che l'input è composto da un numero variabile di bit, abbiamo deciso di scomporlo in più segnali. I primi due bit sono sempre forniti e quindi saranno salvati in due appositi registri, fino a quando non torneranno utili per decidere su quale dei registri di uscita salvare il contenuto della memoria.

Gli altri 16 bit, che indicano la cella di memoria dalla quale prelevare il dato, se presenti sono salvati in un segnale vettoriale da 16 bit, che viene inizializzato in modo da contenere solo zeri prima di iniziare la lettura dell'input.

L'input è seriale e viene fornito partendo dal bit più significativo, fino al meno significativo, perciò si è reso necessario salvare ogni bit singolarmente, dopo aver eseguito un'operazione di *shift left* dell'intero vettore. In partenza il vettore è occupato da 16 bit inizializzati a zero, in modo che, se sono forniti in input bit in numero minore di 16, le cifre più significative del registro vengano occupate da zeri, in modo da non modificare il valore dell'indirizzo fornito.

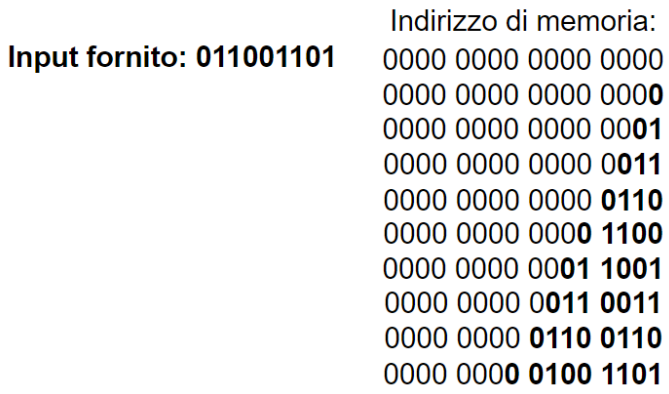


Figure 4: Esempio esecuzione dell'operazione di shift

Successivamente, una volta ottenuto il valore della memoria, questo verrà salvato in uno dei quattro registri appositi, ognuno corrispondente a ciascuna uscita. Questa operazione è necessaria in quanto il valore di uscita va mostrato per un solo ciclo di clock e per i restanti cicli le uscite devono essere fisse a zero, quindi si rende necessario salvare in dei registri i valori da mostrare in seguito.

2.2 Segnali Principali

Alcuni dei principali segnali utilizzati per lo svolgimento del progetto sono i seguenti:

- *bit_first*: contiene il primo bit ricevuto in input, necessario per stabilire quale delle 4 uscite Z scegliere.
- *bit_second*: contiene il secondo bit ricevuto in input, necessario per stabilire quale delle 4 uscite Z scegliere.
- *address*: segnale vettoriale da 16 bit, contiene i bit che rappresentano l'indirizzo di memoria da cui leggere.
- *reg_z0, reg_z1, reg_z2, reg_z3*: registri vettoriali da 8 bit, contenenti i bit da mostrare in output sulle rispettive uscite quando *o_done* è 1.
- il restante dei segnali sono utilizzati per attivare i sottoprocessi ai differenti stati della FSM.

2.3 Macchina a Stati Finiti

Per la realizzazione del progetto abbiamo scelto di utilizzare una macchina a stati finiti di Moore la cui caratteristica principale è il fatto che le uscite sono in funzione solo degli stati correnti e non anche degli stati d'ingresso.

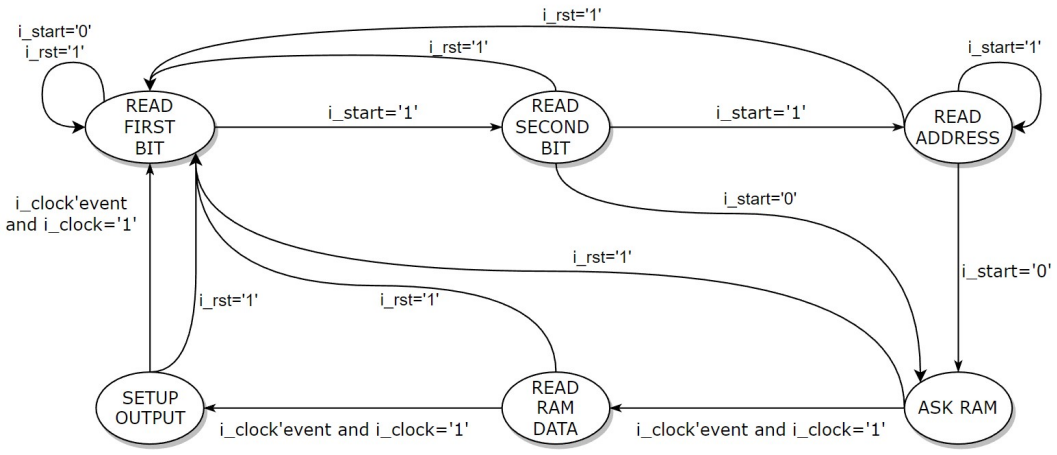


Figure 5: Rappresentazione della macchina a stati finiti

Sono specificati di seguito gli stati utilizzati:

Tabella degli Stati	
Stato	Funzionamento
READ_FIRST_BIT	Stato iniziale, quando il segnale di start è alto legge il primo bit del segnale sequenziale *i_w* in input relativo all'uscita. Inoltre è lo stato a cui torna la FSM quando si incontra un segnale di reset.
READ_SECOND_BIT	Secondo stato, legge il secondo bit del segnale sequenziale *i_w* in input relativo all'uscita.
READ_RAM_ADDRESS	Stato che si occupa della lettura dell'input rappresentante l'indirizzo da inviare alla memoria.
ASK_RAM	Stato che aggiorna il segnale di output *o_mem_addr* e aziona il senale *o_mem_en*, in modo da permettere la comunicazione tra la entity e la memoria.
READ_RAM_DATA	Stato per la lettura del dato in ingresso sul canale *i_mem_data*, mentre la FSM si trova in questo stato viene anche copiato il dato ricevuto sul registro di uscita corretto in base ai dati letti durante i primi due stati della macchina.
SETUP_O	Stato usato per porre *o_done* a 1, in modo da notificare la fine dell'elaborazione, mentre i canali di output Z0, Z1, Z2 e Z3 passano da 00000000 a rappresentare i valori contenuti negli appositi registri di uscita.

Il funzionamento della macchina a stati finiti è garantito dalla presenza dei due segnali *curr_state* e *next_state*: il primo rappresenta lo stato presente al ciclo di clock in cui ci si trova, e a seconda di questo e dei segnali in input come *i_start* vengono decisi gli stati futuri (*next_state*), in modo che al ciclo di clock successivo *curr_state* si aggiorni al valore di *next_state*.

All'interno del progetto ci sono due diversi processi per la gestione e il funzionamento della macchina a stati: una funzione per gestire i segnali che attivano i processi di supporto, l'altra serve ad aggiornare il valore di *next_state*.

2.4 Gestione dei Processi Principali

Per semplificare la comprensione del progetto, abbiamo deciso di effettuare una divisione in sotto-processi:

- Processo per aggiornare il primo bit: quando il segnale corrispondente viene attivato nella FSM, il bit corrente nell'ingresso *i_w* viene salvato nel primo registro.
- Processo per aggiornare il secondo bit: quando il segnale corrispondente viene attivato nella FSM, il bit corrente nell'ingresso *i_w* viene salvato nel secondo registro.
- Processo per aggiornare l'indirizzo: processo che rimane attivo per tutti i restanti cicli di clock in cui *i_start* è a 1. Si occupa di eseguire l'operazione di *shift left* del registro da 16 bit, dopodichè salva il valore del bit corrente nell'ingresso *i_w* nella sedicesima posizione del vettore. Il registro non viene resettato solamente quando *i_rst* va a 1, ma anche quando il valore della cella di memoria è stato letto, in quanto avere ancora dei bit del vettore ad 1 potrebbe compromettere il corretto salvataggio dell'indirizzo la volta successiva che la macchina a stati attiva il processo.
- Processo per l'invio dell'indirizzo di memoria: si occupa di copiare il valore del registro di indirizzo al segnale vettoriale di output *o_mem_addr*.
- Processo per salvare il valore che arriva dalla memoria in un apposito registro.
- Processo per copiare il valore arrivato dalla memoria: salva il valore nel registro di uscita specifico in base ai due bit salvati all'inizio dell'esecuzione del programma.

2.5 Gestione dei Reset

Ognuno dei processi gestisce l'arrivo di un segnale di reset al suo interno: quando *i_rst* va a 1, in qualunque stato della FSM ci si trovi vengono resettati e portati a zero tutti i registri. Inoltre, indipendentemente da quale sia il *curr_state*, si porta *next_state* allo stato READ_FIRST_BIT. Quest'ultima operazione è eseguita nel processo denominato *aggiorna_stato_fsm*.

2.6 Risultato finale Architettura

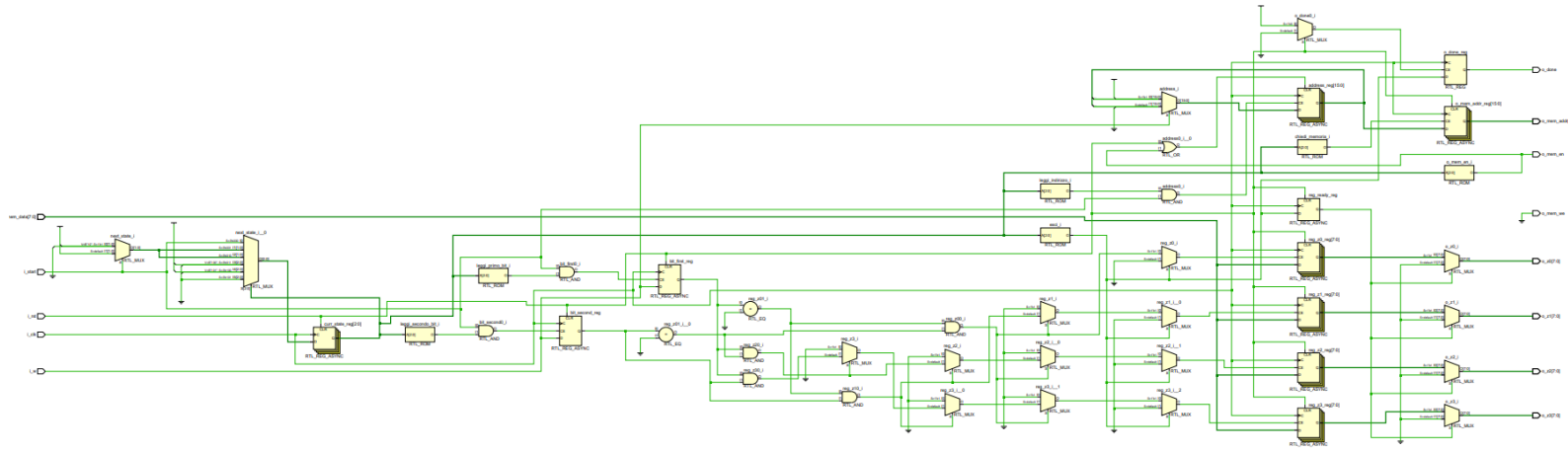


Figure 6: Struttura finale del progetto generata da Vivado, che tiene in conto dei registri e dei segnali principali del componente

3 Risultati Sperimentali

Sono riportati in seguito il Report di Sintesi e il Timing Report generati da Vivado al momento dell'esecuzione della sintesi.

3.1 Report Utilization

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	28	0	0	134600	0.02
LUT as Logic	28	0	0	134600	0.02
LUT as Memory	0	0	0	46200	0.00
Slice Registers	74	0	0	269200	0.03
Register as Flip Flop	74	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

Il primo report rappresenta un riassunto riguardante i Flip-Flop utilizzati nella sintesi (74 in questo caso); si può notere che non è presente alcun latch nel nostro progetto.

3.2 Timing Report

```
Timing Report

Slack (MET) :             97.501ns  (required time - arrival time)
  Source:                 FSM_onehot_curr_state_reg[4]/C
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@5.000ns period=100.000ns})
  Destination:            address_reg[0]/CLR
                            (recovery check against rising-edge clock clock  {rise@0.000ns fall@5.000ns period=100.000ns})
  Path Group:             **async_default**
  Path Type:              Recovery (Max at Slow Process Corner)
  Requirement:            100.000ns  (clock rise@100.000ns - clock rise@0.000ns)
  Data Path Delay:        1.910ns  (logic 0.751ns (39.319%)  route 1.159ns (60.681%))
  Logic Levels:           1  (LUT2=1)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.100ns = ( 102.100 - 100.000 )
    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
```

Dal Timing Report si evince che la differenza tra il tempo di clock (100 ns) e il tempo effettivamente utilizzato per produrre un output è di 97.501ns,

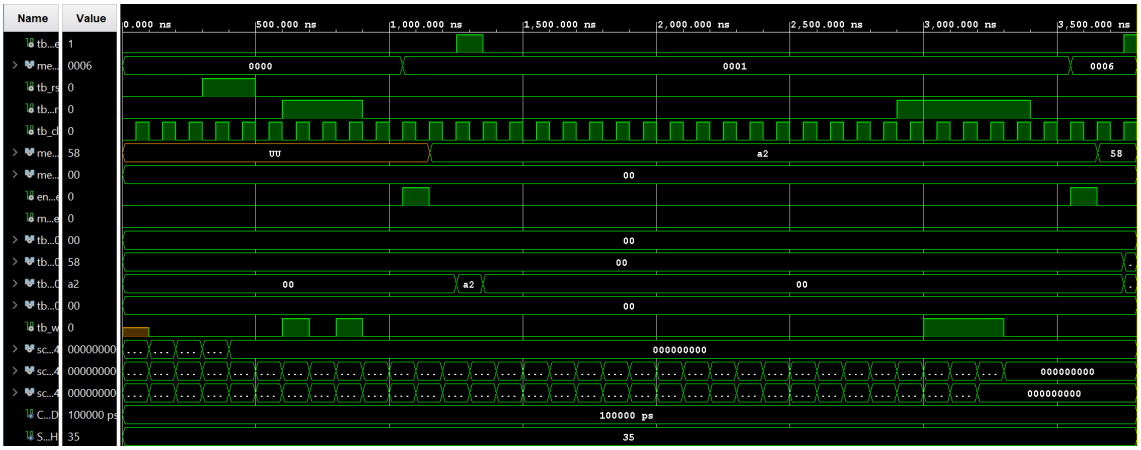
questo significa che il segnale che impiega più tempo a commutare lo fa in $2.499ns$ ben al di sotto del tempo massimo definito nelle specifiche.

3.3 Testing

Dopo aver superato sia in *Behavioural* sia in *Post-Synthesis* il TestBench fornitoci di prova, abbiamo testato il nostro progetto per vedere se superasse i casi limite:

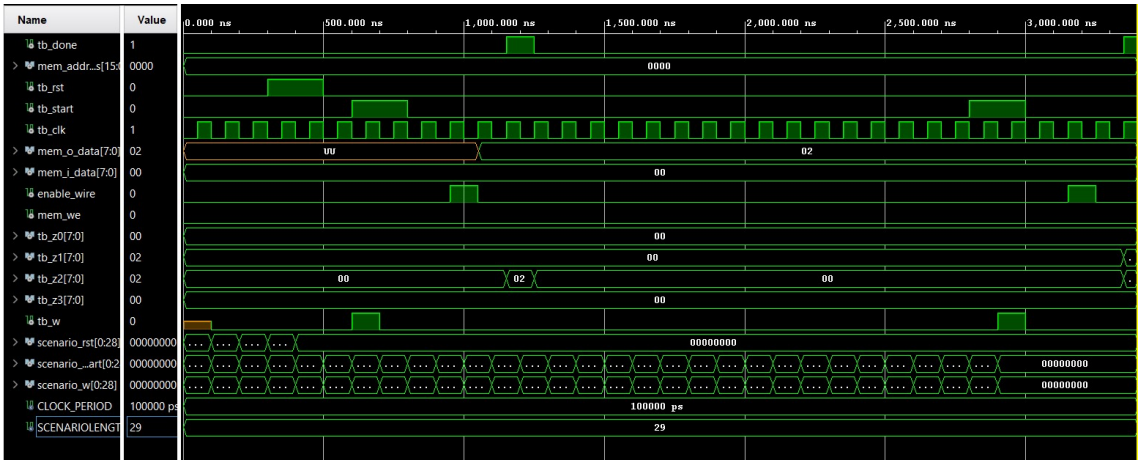
1. Caso base (TestBench fornito tramite WeBeep): si verifica un reset iniziale (prima di qualsiasi altra operazione) e due richieste di raccolta di dati che non appartengono ai casi limite riportati sotto.

```
CONSTANT SCENARIOLENGTH : INTEGER := 35; -- 5 + 3 + 20 + 7   (RST) + (CH2-MEM[1]) + 20 CYCLES + (CH1-MEM[6])
SIGNAL scenario_rst : unsigned(0 TO SCENARIOLENGTH - 1)      := "00110" & "000" & "00000000000000000000" & "0000000";
SIGNAL scenario_start : unsigned(0 TO SCENARIOLENGTH - 1)    := "00000" & "111" & "00000000000000000000" & "1111100";
SIGNAL scenario_w : unsigned(0 TO SCENARIOLENGTH - 1)        := "00000" & "101" & "00000000000000000000" & "0111000";
```



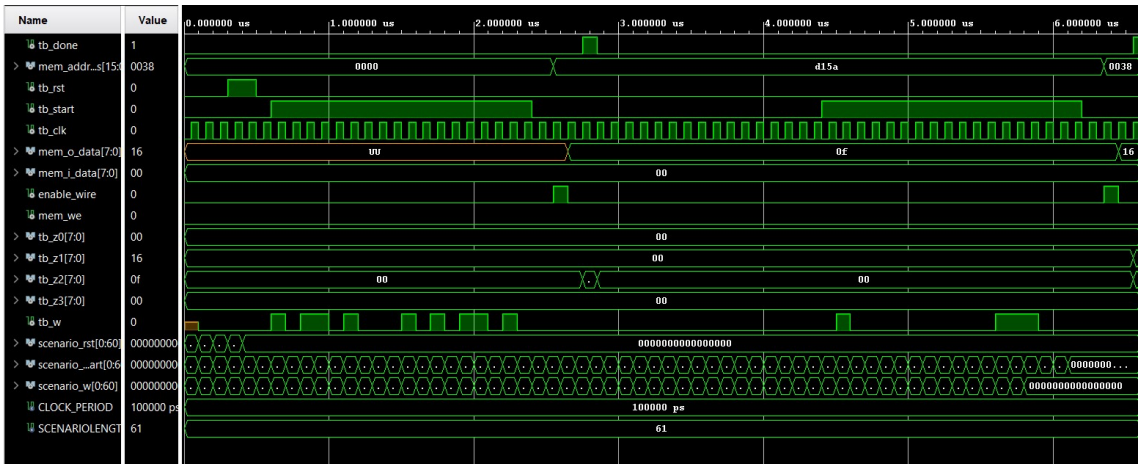
2. Caso in cui *i_start* rimanga alto solo per 2 cicli di clock (il minimo tempo definito dalle specifiche).

```
CONSTANT SCENARIOLENGTH : INTEGER := 29;
SIGNAL scenario_rst : unsigned(0 TO SCENARIOLENGTH - 1)     := "00110" & "00" & "00000000000000000000" & "00";
SIGNAL scenario_start : unsigned(0 TO SCENARIOLENGTH - 1)   := "00000" & "11" & "00000000000000000000" & "11";
SIGNAL scenario_w : unsigned(0 TO SCENARIOLENGTH - 1)       := "00000" & "10" & "00000000000000000000" & "01";
```



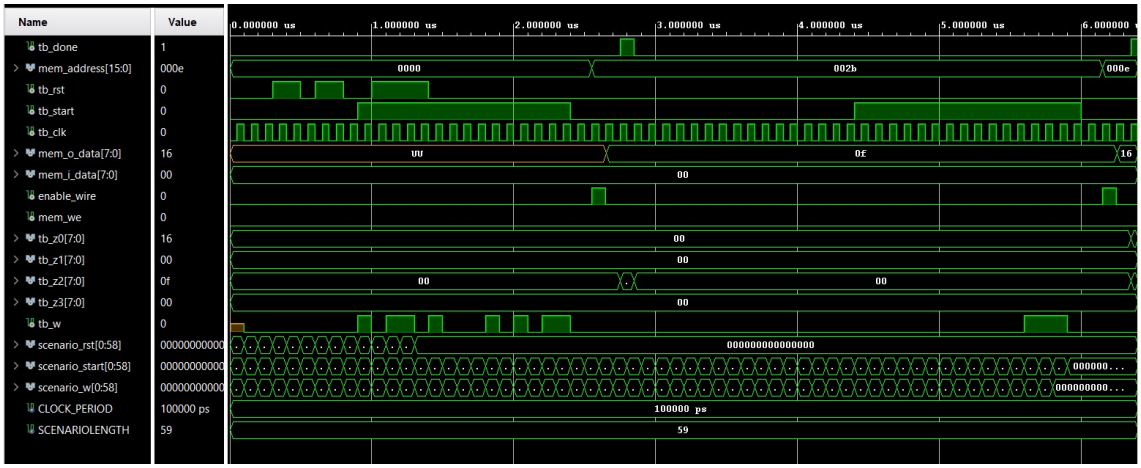
3. Caso in cui *i_start* rimanga alzato per 18 cicli di clock (il massimo tempo definito dalle specifiche).

```
CONSTANT SCENARIOLENGTH : INTEGER := 61;
SIGNAL scenario_rst : unsigned(0 TO SCENARIOLENGTH - 1)     := "00110" & "00000000000000000" & "00000000000000000000" & "00000000000000000";
SIGNAL scenario_start : unsigned(0 TO SCENARIOLENGTH - 1)   := "00000" & "11111111111111111" & "00000000000000000000" & "11111111111111111";
SIGNAL scenario_w : unsigned(0 TO SCENARIOLENGTH - 1)       := "00000" & "10110100010101101O" & "00000000000000000000" & "01000000000111000";
```



4. Caso in cui *i_rst* si alzi quando *i_start* è a 1 e quindi mentre si sta leggendo l'input.

```
CONSTANT SCENARIOLENGTH : INTEGER := 59;
SIGNAL scenario_rst : unsigned(0 TO SCENARIOLENGTH - 1)     := "00110110" & "011110000000000" & "00000000000000000000" & "00000000000000";
SIGNAL scenario_start : unsigned(0 TO SCENARIOLENGTH - 1)   := "00000000" & "111111111111111" & "00000000000000000000" & "111111111111111";
SIGNAL scenario_w : unsigned(0 TO SCENARIOLENGTH - 1)       := "00000000" & "101101000101011" & "00000000000000000000" & "00000000000001110";
```

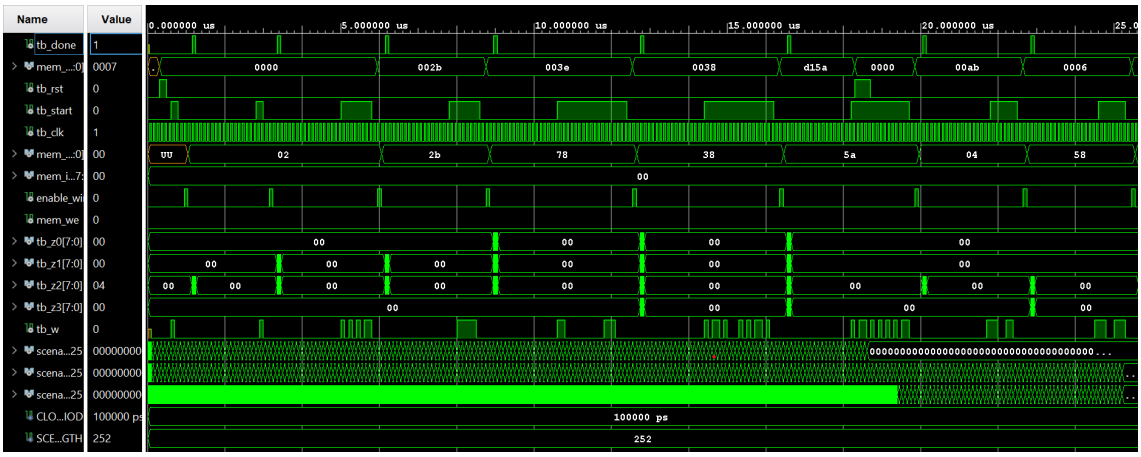


5. Caso in cui succedono tutti i casi precedenti all'interno dello stesso TestBench.

```
CONSTANT SCENARIOLENGTH : INTEGER := 252;
SIGNAL scenario_rst : unsigned(0 TO SCENARIOLENGTH - 1)     := "00110" & "00" & "0000000000000000000" & "00" & "0000000000000000000" & "00000000" & "0000000000000000000"
                                                             & "00000000" & "0000000000000000000" & "0000000000000000000" & "0000000000000000000" & "0000000000000000000"
                                                             & "0000000000000000000" & "01111000000000" & "0000000000000000000" & "00000000" & "0000000000000000000"
                                                             & "00000000";

SIGNAL scenario_start : unsigned(0 TO SCENARIOLENGTH - 1)   := "00000" & "11" & "0000000000000000000" & "11" & "0000000000000000000" & "11111111" & "0000000000000000000"
                                                             & "11111111" & "0000000000000000000" & "1111111111111111" & "0000000000000000000" & "1111111111111111"
                                                             & "0000000000000000000" & "11111111111111" & "0000000000000000000" & "01111111" & "0000000000000000000"
                                                             & "01111111";

SIGNAL scenario_w : unsigned(0 TO SCENARIOLENGTH - 1)       := "00000" & "10" & "0000000000000000000" & "01" & "0000000000000000000" & "10101011" & "0000000000000000000"
                                                             & "00111110" & "0000000000000000000" & "1100000000111000" & "0000000000000000000" & "1011010001011010"
                                                             & "0000000000000000000" & "101101010101011" & "0000000000000000000" & "11100110" & "0000000000000000000"
                                                             & "11100111";
```



Tutti i casi elencati prima sono stati superati sia in *Behavioural* sia in *Post-Synthesis* nei tempi prestabiliti dalle specifiche.

4 Conclusioni

Questo progetto è stata una novità dal momento che in nessun corso ci è stato chiesto di implementare da zero a livello fisico come funzionano i componenti che utilizziamo o comunque non con una struttura simile.
Utilizzare Vivado ci ha posto nuove sfide da affrontare e ci ha insegnato a pensare in modo differente e a cambiare i nostri schemi mentali, dovendo per la prima volta programmare un componente hardware anzichè uno software, partendo solamente da una specifica.