

Travaini Alessandro



PROVA FINALE

PROGETTO DI RETI LOGICHE

POLITECNICO DI MILANO
INGEGNERIA INFORMATICA

Codice persona: 10742196

Matricola: 955634

Docente: Palermo Gianluca

Anno accademico: 2022-2023

Sommario

| | | |
|-----|---|----|
| 1 | Introduzione | 2 |
| 1.1 | Specifiche generali..... | 2 |
| 1.2 | Interfaccia del componente | 3 |
| 1.3 | Descrizione della RAM | 3 |
| 2 | Architettura | 4 |
| 2.1 | Tabella dei segnali e delle variabili interne | 4 |
| 2.2 | Scelte progettuali | 5 |
| 2.3 | Descrizione stati FSM | 6 |
| 3 | Risultati sperimentali..... | 7 |
| 3.1 | Report di sintesi..... | 9 |
| 3.2 | Simulazioni..... | 10 |
| 3.3 | Simulazioni custom..... | 11 |
| 4 | Conclusioni | 12 |

1 Introduzione

Il sistema descritto nel seguito è un particolare dispositivo di conversione seriale parallelo. Il dispositivo riceve indicazioni di una locazione di memoria, il cui contenuto viene presentato su uno dei quattro canali d'uscita disponibili. Le informazioni circa l'indirizzo di memoria e l'uscita da utilizzare vengono fornite in modo codificato sull'ingresso seriale.

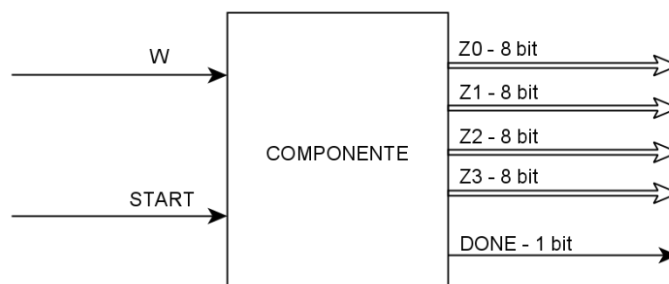


Figura 1: Diagramma ad alto livello del componente

1.1 Specifiche generali

Nello specifico, il componente riceve in ingresso (W) una sequenza di bit sincrona con il clock di sistema che viene considerata valida quando il segnale di abilitazione (START) vale "1". Il segnale di ingresso è formato in generale da due bit che identificano l'uscita su cui scrivere e da una serie di massimo sedici bit che identificano l'indirizzo di RAM dal quale recuperare il dato. La codifica seriale dei dati in ingresso presuppone che il primo dato ricevuto sia l'MSB del selettore di uscita seguito dagli altri, come indicato in figura 2:

| Selettore uscita – 2 bit | | Indirizzo RAM – massimo 16 bit | |
|--------------------------|-----|--------------------------------|-----|
| MSB | LSB | MSB | LSB |

Figura 2: Divisione del dato seriale in ingresso

A seguito della ricezione del dato dalla RAM, il componente lo scrive prima su un registro dell'uscita corretta, poi copia il contenuto di tutti i registri sulle uscite. Questo passo è necessario per specifica, dato che, per ogni sequenza d'ingresso valida, il componente deve restituire sull'uscita selezionata il nuovo dato, mentre su tutte le altre il dato scritto precedentemente. Le uscite sono normalmente a "0" ad eccezione del singolo ciclo di clock in cui sono presentati i dati; contemporaneamente viene portato a "1" il segnale che indica che il componente ha terminato l'elaborazione (DONE).

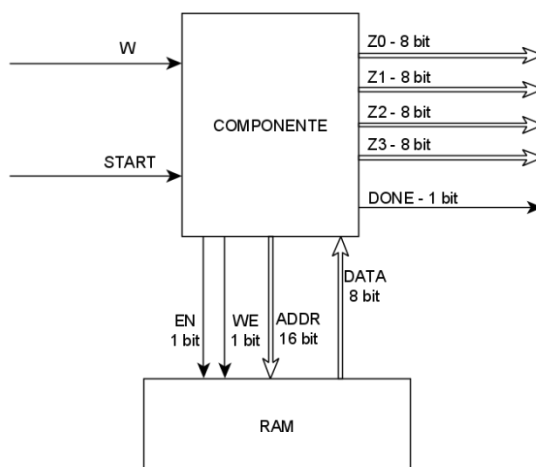


Figura 3: Diagramma con interazione della RAM

1.2 Interfaccia del componente

Il componente è descritto in linguaggio VHDL con 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 in ingresso generato dal Test Bench;
- *i_rst* è il segnale di RESET che inizializza la macchina per ricevere il primo segnale di START;
- *i_start* è il segnale di START generato dal Test Bench. Questo segnale abilita il componente ad accettare lo stream d'ingresso *i_w*;
- *i_w* è il segnale W precedentemente descritto e generato dal Test Bench;
- *o_z0*, *o_z1*, *o_z2*, *o_z3* sono i quattro canali di uscita;
- *o_done* è il segnale di uscita che comunica la fine dell'elaborazione;
- *o_mem_addr* è il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- *i_mem_data* è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
- *o_mem_en* è il segnale di ENABLE da dover mandare alla memoria per poter comunicare sia in lettura sia in scrittura;
- *o_mem_we* è il segnale di WRITE ENABLE da dover mandare alla memoria per poter scriverci. Per leggere da memoria esso deve essere "0". Nel caso del progetto rimarrà sempre a "0".

1.3 Descrizione della RAM

La memoria viene istanziata all'interno dei test bench a cui è sottoposto il componente. La struttura della memoria e il suo protocollo seguono le linee guida proposte nella [documentazione di Xilinx](#).

Nel caso del progetto la memoria ha 65536 posizioni indirizzabili per mezzo di una parola a 16 bit; ciascuna posizione può contenere un dato di 8 bit.

2 Architettura

2.1 Tabella dei segnali e delle variabili interne

| NOME | TIPO | VALORE INIZIALE | NOTE |
|----------------|-------------------------------|--------------------|--|
| curr_state | state_type | U | Memorizza lo stato della FSM. |
| outputSelector | std_logic_vector(1 downto 0) | 00 | Memorizza l'uscita da scrivere con il dato nuovo. |
| ramAddress | std_logic_vector(15 downto 0) | 0000000000000000 | Memorizza l'indirizzo della memoria dal quale recuperare il dato. |
| i | integer | 0 | Indice necessario per il funzionamento del registro a scorrimento. |
| numFiller | integer | 0 | Tiene conto di quanti zeri sono stati inseriti come "padding" nel registro a scorrimento. |
| shift_counter | integer | 0 | Tiene conto di quanti valori sono stati inseriti nel registro a scorrimento. |
| myStart | boolean | false | Indicatore per iniziare a inserire "padding" nel registro a scorrimento quando il segnale <i>i_start</i> diventa "0", ma il registro non è ancora pieno. |
| myInput | std_logic | 0 | Valore di "padding" per il registro. Durante l'esecuzione rimarrà sempre a "0". |
| shift_register | std_logic_vector(17 downto 0) | 000000000000000000 | Memorizza i valori per il registro a scorrimento usato per leggere l'ingresso seriale. I 18 bit verranno poi divisi in 2 bit per il selettore di uscita e 16 bit di indirizzo della memoria. |
| data_read | std_logic_vector(7 downto 0) | 00000000 | Memorizza il dato proveniente dalla RAM. |
| latchOutput_0 | std_logic_vector(7 downto 0) | 00000000 | Memorizza il dato scritto sull'uscita <i>o_z0</i> . |
| latchOutput_1 | std_logic_vector(7 downto 0) | 00000000 | Memorizza il dato scritto sull'uscita <i>o_z1</i> . |
| latchOutput_2 | std_logic_vector(7 downto 0) | 00000000 | Memorizza il dato scritto sull'uscita <i>o_z2</i> . |
| latchOutput_3 | std_logic_vector(7 downto 0) | 00000000 | Memorizza il dato scritto sull'uscita <i>o_z3</i> . |

Figura 4: Tabella dei segnali e variabili e loro valori

2.2 Scelte progettuali

Nella realizzazione del componente si è deciso di implementare un'architettura di tipo *behavioral*, utilizzando un singolo process attivato dai segnali di clock, reset, start e mem_data. All'interno di tale process è stata creata la macchina a stati finiti che regola l'avanzamento del componente.

Il componente svolge principalmente tre azioni:

- Lettura degli ingressi e trasformazione del dato da seriale a parallelo;
- Accesso alla RAM;
- Scrittura delle uscite;

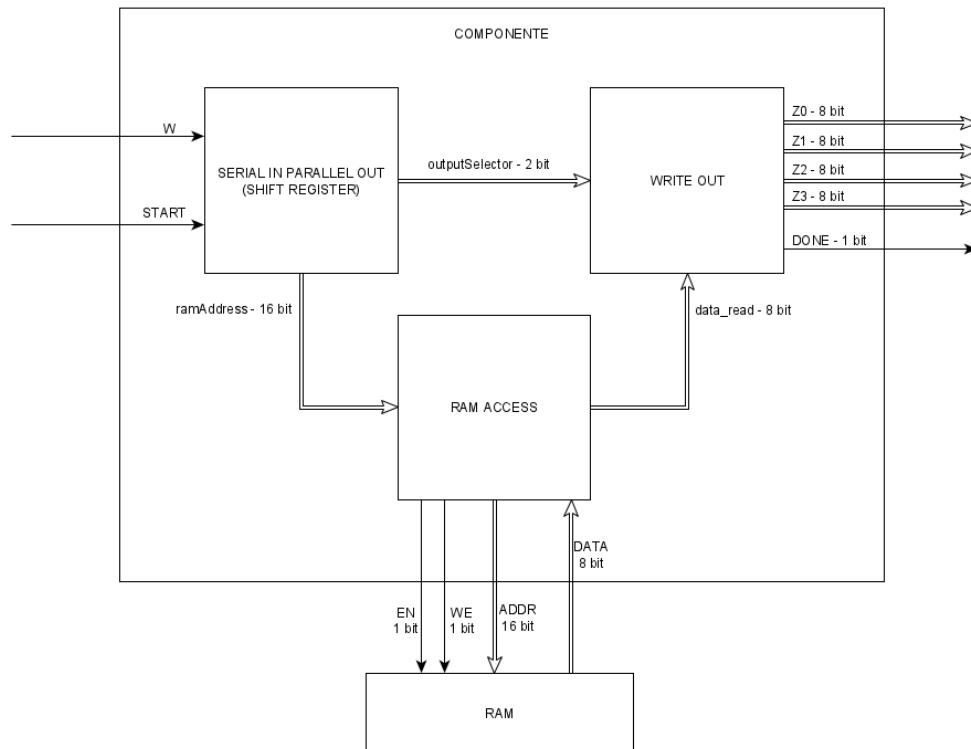


Figura 5: Visualizzazione delle azioni del componente.

Ad ognuna di queste azioni sono stati assegnati uno o più stati della FSM, rappresentata nella figura seguente:



Figura 6: Diagramma della macchina a stati finiti

2.3 Descrizione stati FSM

Gli stati scelti per il funzionamento del componente sono cinque:

- **SERIAL_IN_PARALLEL_OUT:** questo stato implementa il registro a scorrimento. Nel caso in cui i bit dell'indirizzo non siano sedici, si è deciso di inserire zeri come "padding" per riempire il registro: ciò è stato possibile con le variabili myStart e myInput. La prima infatti diventa 1 allo stesso momento in cui i_start diventa 1, in modo tale da sovrascrivere i_start quando questo diventa 0 e il registro non è ancora pieno. Nel registro normalmente viene scritto l'input i_w, mentre myInput è inserito solo quando i_start diventa 0. Quando shift_counter, che tiene conto di quante posizioni sono state scritte nel registro, è uguale a 18, il registro è pieno e il suo contenuto è diviso secondo le adeguate regole e copiato nei vettori outputSelector e ramAddress. Poiché nel registro a scorrimento il bit più recente è in posizione 0 (LSB), bisogna prestare attenzione nel caso si siano usati bit di "padding", perché altrimenti verrebbe copiato in ramAddress un indirizzo di memoria sbagliato. Per ovviare a ciò si usa la variabile numFiller, che tiene conto di quanti zeri sono stati inseriti come "padding": per copiare l'indirizzo di memoria in ramAddress infatti si ignorano tutti i primi n bit (con n = numFiller).
- **FETCH_RAM:** questo stato implementa la richiesta del dato alla RAM. La RAM viene attivata (o_mem_en = 1), ne viene disabilitata la scrittura (o_mem_we = 0) e viene inviato l'indirizzo corretto per recuperare il dato.
- **READ_RAM_DATA:** in questo stato si attende la risposta della memoria. La RAM viene disattivata (o_mem_en = 0).
- **OUTPUT_DATA:** in questo stato vengono scritte le uscite. Vengono prima scritti i registri delle uscite in base al valore del selettore dell'uscita outputSelector. Tali registri sono necessari per poter scrivere alle richieste successive i dati precedenti. In seguito, tutte le uscite del componente sono scritte con i valori dei propri registri. L'uscita o_done viene messa a 1 come da specifica.
- **JOB_DONE:** in questo stato si resettano tutte le uscite del componente portandole al valore di default, 0. Inoltre, vengono resettati i vettori outputSelector e ramAddress in modo tale da prepararli per la prossima richiesta.

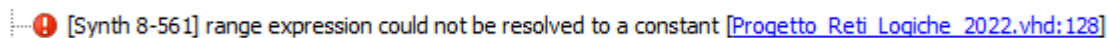
3 Risultati sperimentali

Il componente è stato implementato in origine utilizzando la seguente porzione di codice per calcolare l'indirizzo di accesso alla RAM:

```
for i in 0 to (15 - numFiller) loop
    ramAddress(i) <= shift_register(i + numFiller);
end loop;
```

Con questa implementazione sono stati superati tutti i test bench in simulazione behavioral pre-sintesi.

Successivamente si è tentato di effettuare la sintesi del componente, ma quest'ultima è fallita a causa del seguente errore:



[Synth 8-561] range expression could not be resolved to a constant [\[Progetto Reti Logiche 2022.vhd:128\]](#)

Figura 7: Messaggio di errore di sintesi

L'origine di questo errore era legata al modo con cui è stato implementato il ciclo for visto in precedenza. Poiché la lunghezza del ciclo non è costante, ma cambia a *runtime* a seguito della variazione di numFiller, il sintetizzatore di Vivado non riesce a preallocare le risorse hardware in modo corretto, generando l'errore.

Il problema è stato risolto nella revisione successiva, sostituendo il ciclo for con uno switch case sul valore di numFiller e creando ramAddress tramite una concatenazione di zeri iniziali e la corretta parte del registro a scorrimento.

```
case(numFiller) is
  when 0 =>
    ramAddress <= shift_register(15 downto 0);
  when 1 =>
    ramAddress <= "0" & shift_register(15 downto 1);
  when 2 =>
    ramAddress <= "00" & shift_register(15 downto 2);
  when 3 =>
    ramAddress <= "000" & shift_register(15 downto 3);
  when 4 =>
    ramAddress <= "0000" & shift_register(15 downto 4);
  when 5 =>
    ramAddress <= "00000" & shift_register(15 downto 5);
  when 6 =>
    ramAddress <= "000000" & shift_register(15 downto 6);
  when 7 =>
    ramAddress <= "0000000" & shift_register(15 downto 7);
  when 8 =>
    ramAddress <= "00000000" & shift_register(15 downto 8);
  when 9 =>
    ramAddress <= "000000000" & shift_register(15 downto 9);
  when 10 =>
    ramAddress <= "0000000000" & shift_register(15 downto 10);
  when 11 =>
    ramAddress <= "00000000000" & shift_register(15 downto 11);
  when 12 =>
    ramAddress <= "000000000000" & shift_register(15 downto 12);
  when 13 =>
    ramAddress <= "0000000000000" & shift_register(15 downto 13);
  when 14 =>
    ramAddress <= "00000000000000" & shift_register(15 downto 14);
  when 15 =>
    ramAddress <= "000000000000000" & shift_register(15 downto 15);
  when 16 =>
    ramAddress <= "0000000000000000";
  when others =>
    ramAddress <= "0000000000000000";
end case;
```

A seguito di questo accorgimento, il componente risulta correttamente sintetizzabile senza errori.

3.1 Report di sintesi

Dal log di sintesi:

Synthesis finished with 0 errors, 0 critical warnings and 0 warnings.

| Resource | Estimation | Available | Utilization % |
|----------|------------|-----------|---------------|
| LUT | 239 | 134600 | 0.18 |
| FF | 189 | 269200 | 0.07 |
| IO | 63 | 285 | 22.11 |
| BUFG | 1 | 32 | 3.13 |

Figura 8: Report di sintesi

Il diagramma del componente è il seguente:

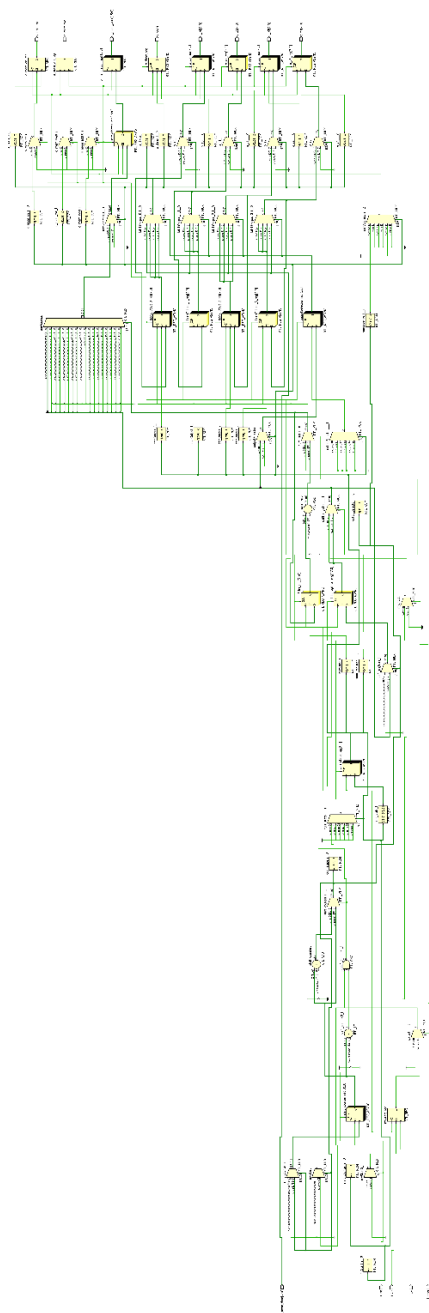


Figura 9: Schematico del componente

3.2 Simulazioni

Il componente supera correttamente tutti i test bench proposti dal docente in simulazione behavioral pre-sintesi, come da tabella seguente:

| N. | Test bench | Simulazione pre-sintesi | Simulazione post-sintesi |
|----|---------------------|-------------------------|--------------------------|
| 1 | tb_1.vhd | TEST PASSATO | TEST FALLITO |
| 2 | tb_2.vhd | TEST PASSATO | TEST PASSATO |
| 3 | tb_3.vhd | TEST PASSATO | TEST FALLITO |
| 4 | tb_4.vhd | TEST PASSATO | TEST PASSATO |
| 5 | tb_5.vhd | TEST PASSATO | TEST FALLITO |
| 6 | tb_6.vhd | TEST PASSATO | TEST PASSATO |
| 7 | tb_7.vhd | TEST PASSATO | TEST PASSATO |
| 8 | tb_example23.vhd | TEST PASSATO | TEST PASSATO |
| 9 | tb_reset_custom.vhd | TEST PASSATO | TEST PASSATO |

In simulazione post sintesi, sia functional che timing, il componente supera tutti i test bench eccetto tb_1, tb_3 e tb_5. In questi casi il componente recupera correttamente i dati dalla memoria, li scrive sull'uscita selezionata, ma il test bench fallisce nel controllare le uscite nel momento giusto, nonostante tutti i segnali necessari siano presenti. Come si può vedere dalle immagini seguenti, a seguito di una comparazione manuale, i grafici d'onda della simulazione behavioral e functional coincidono sia nei dati presentati, sia negli istanti in cui questi ultimi vengono scritti sulle uscite. Il momento evidenziato è quando la simulazione functional post sintesi fallisce; tuttavia, come si può notare i valori dei segnali sono congruenti.



Figura 10: Confronto tra la simulazione behavioral (sopra) e functional (sotto) del testbench tb_1

3.3 Simulazioni custom

Al fine di valutare alcuni casi limite, soprattutto per quanto riguarda il segnale di reset, si è deciso di scrivere e simulare un test bench fatto *ad hoc*.

Tale test bench contiene infatti dei segnali di reset nei punti principali del funzionamento del modulo:

- Come da specifica è presente un reset prima che il segnale START vada ad '1';
- Il secondo reset avviene mentre il segnale START è uguale a '1', dopo che alcuni valori sono già stati inseriti nel registro a scorrimento. Il comportamento atteso è che non venga attivata la RAM né venga scritto qualcosa sulle uscite.
- Il terzo reset accade quando START commuta a '0', ovvero quando il registro a scorrimento sta inserendo zeri di "padding". Anche in questo caso il comportamento atteso è analogo a quello di cui sopra.
- Il quarto reset infine viene mandato in concomitanza con la scrittura delle uscite. In quest'ultimo caso il comportamento atteso è il seguente: il componente recupera il dato dalla memoria, ma non lo scrive sulle uscite.

Per semplicità, visto che non è oggetto del test, viene richiesto sempre lo stesso dato che deve essere scritto sempre sulla stessa uscita (o_z2).

Il componente passa senza errori il test sia in simulazione behavioral, sia in simulazione functional.

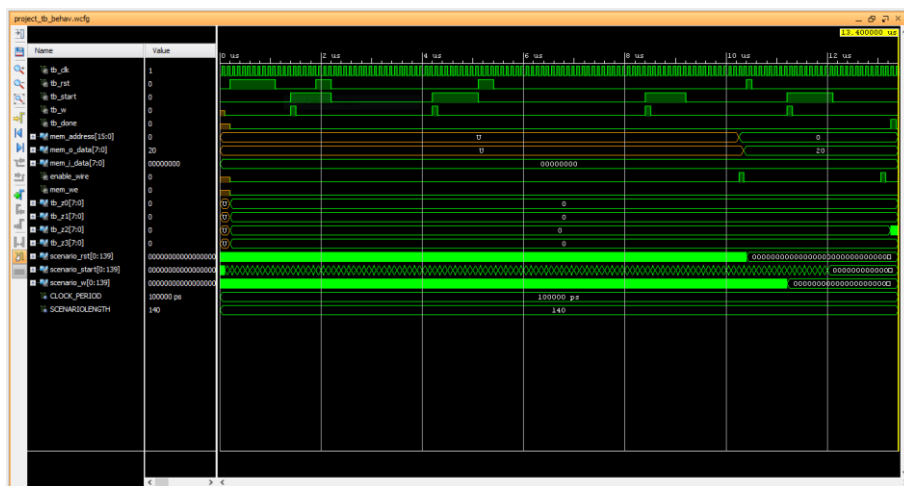


Figura 11: Simulazione behavioral

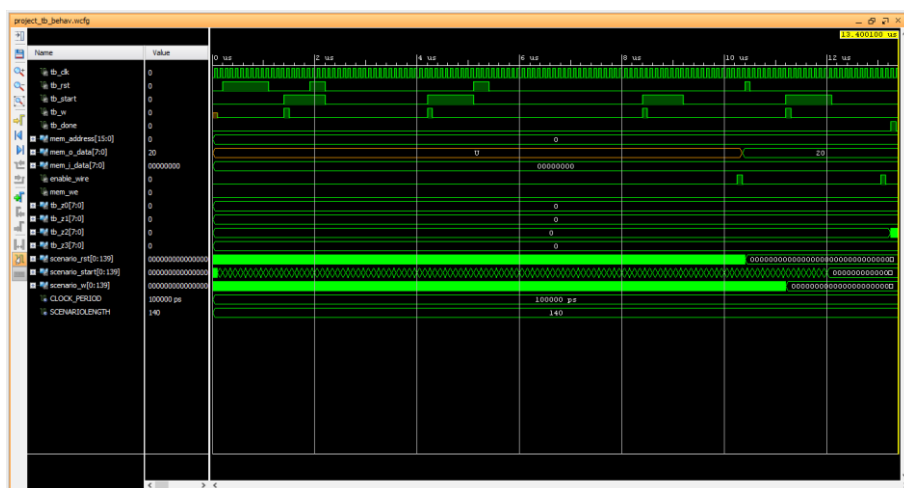


Figura 12: Simulazione functional post sintesi

4 Conclusioni

Il modulo hardware descritto è in grado di leggere un ingresso seriale, trasformarlo in parallelo e tramite regole ben specificate riesce ad accedere ad un indirizzo della RAM, recuperare un dato e scriverlo su un'apposita uscita delle quattro disponibili.

Una possibile applicazione di questo modulo hardware nel mondo reale può essere come traduttore seriale-parallelo "avanzato" per un componente che riceve input in modo solo parallelo, interno ad un comune calcolatore dove la maggior parte della comunicazione avviene via seriale; in tal caso il componente accettore della traduzione, può confrontare un nuovo dato proveniente dalla memoria (ad esempio sull'uscita o_z0) con uno storico di tre dati precedenti presenti sulle altre uscite.