Progetto di Reti Logiche 2023

Studente: Angelo Antona Codice Persona: 10665838 Matricola: 911540

Sommario

1. INTRODUZIONE

In questo capitolo si riassume la specifica del progetto attraverso la descrizione dell'interfaccia esterna e del comportamento atteso del componente che si vuole implementare.

1.1. Interfaccia esterna

Il sistema in analisi comunica con l'esterno attraverso i seguenti input/output:

- **i_clk:** è il segnale clock di sistema (1 bit);
- **i_rst:** è il segnale di reset asincrono, che se impostato $= '1'$ permette di reimpostare l'intero sistema al suo stato iniziale (1 bit);
- **i_w:** rappresenta un ingresso seriale (1 bit) tramite il quale il sistema riceverà una stringa binaria la cui funzione verrà specificata in seguito;
- **i_start:** è un segnale generato dal testBench (1 bit) che ha una duplice funzione:
    a. indica l'istante in cui deve iniziare l'elaborazione, ovvero sul primo $rising\_edge(i\_clk)$ in cui $i\_start = '1'$;
    b. Specifica per quale intervallo di tempo bisogna considerare i bit presenti su i_w come significativi, ovvero per tutto il tempo in cui $i\_start = '1'$.

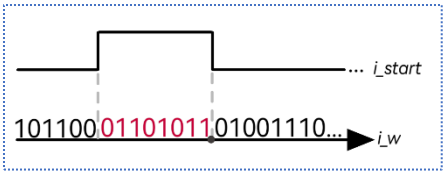


Figura 1: Esempio bit i_w validi (in rosso).

- **o_mem_en, o_mem_we:** rappresentano rispettivamente il segnale per abilitare la comunicazione con la memoria e per scrivere in memoria (1 bit). A seconda dei valori dei due segnali si ha che se:
    - o_mem_en & $o\_mem\_we = \text{“}0-\text{”} \rightarrow comunicazione\ disabilitata.$
    - o_mem_en & $o\_mem\_we = \text{“}10\text{”} \rightarrow lettura\ da\ memoria.$
    - o_mem_en & $o\_mem\_we = \text{”}11\text{”} \rightarrow scrittura\ su\ memoria.$

  (dove "&" è l'operatore di concatenamento di VHDL e "–" indica la condizione di "don't care").
- **o_mem_addr:** è il segnale tramite cui il sistema comunica alla memoria l'indirizzo dal quale desidera leggere (16 bit);
- **i_mem_data:** è il segnale tramite cui la memoria ci restituisce il dato presente nell'indirizzo specificato da o_mem_addr (8 bit);
- **o_Z_0, o_Z_1, o_Z_2, o_Z_3:** sono i 4 segnali di "output principale" del sistema (8 bit). Il loro funzionamento verrà descritto nel paragrafo successivo;
- **o_done:** è il segnale che il sistema porrà $= '1'$ per indicare la fine dell'elaborazione (1 bit).

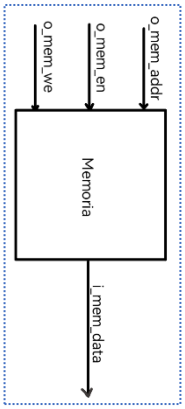


Figura 2: Interfaccia della memoria.

1.2. Comportamento del sistema

La specifica impone che il funzionamento del componente segua le fasi descritte sotto.

ITERAZIONE 1: descriviamo il funzionamento della prima iterazione di funzionamento:

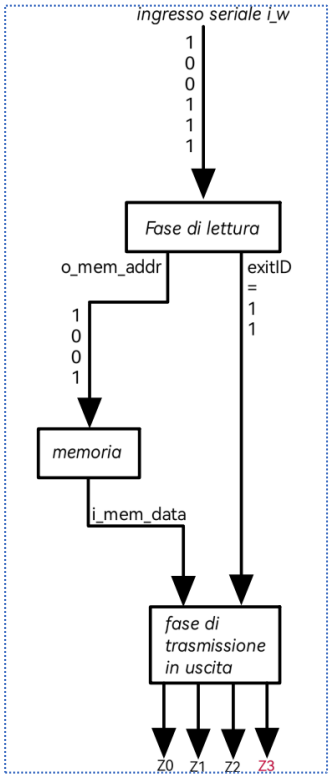
- **IDLE:** All'accensione del dispositivo, quest'ultimo riceve un segnale di reset ($i\_reset = '1'$) che imposta tutti i componenti allo stato di default e rimane successivamente in una fase di attesa, che permane fintanto che non si avrà $i\_start = '1'$. Tale fase di attesa è caratterizzata da:
    - Memoria disabilitata (o_mem_en & $o\_mem\_we = \text{“}0-\text{”}$);
    - Uscite o_Z_i tutte nulle (o_Z_0, o_Z_1, o_Z_2, o_Z_3 <= $(others$ => '0')).
- **LETTURA SERIALE:** Sul primo $rising\_edge(i\_clk)$ in cui $i\_start = '1'$, il sistema dovrà iniziare a leggere i bit sul segnale i_w e dovrà continuare fintanto che $i\_start = '1'$. I bit ricevuti attraverso i_w costituiranno, uno dopo l'altro, una *stringa binaria seriale* composta come segue:
    a. I primi 2 bit della stringa rappresentano in base binaria l'identificativo k di una delle 4 uscite o_Z_k (es. "10" indica l'uscita o_Z_2);
    b. Gli n bit successivi (dove $0 \leq n \leq 16$) rappresentano l'indirizzo (non espanso in segno) da trasmettere successivamente alla memoria attraverso il segnale o_mem_addr.
- **LETTURA DA MEMORIA:** Quando il segnale i_start ritornerà al valore '0' avremo finito di leggere l'ingresso seriale i_w e potremo procedere con la fase di lettura dalla memoria, articolata nei seguenti passaggi:
    a. Se necessario, espandere il segno dell'indirizzo precedentemente letto, così da ottenere un indirizzo di memoria a 16 bit;

b. Trasmettere tale indirizzo alla memoria attraverso il segnale o_mem_addr.

Una volta che la memoria avrà ricevuto l'indirizzo attraverso o_mem_addr, questa ci restituirà un dato tramite il segnale i_mem_data, e potremo così procedere al passo successivo dell'esecuzione.

- **TRASMISSIONE IN USCITA:** Il sistema dovrà a questo punto trasmettere per un solo ciclo di clock il dato i_mem_data sull'uscita o_Z_k. Durante lo stesso ciclo di clock (e soltanto durante quest'ultimo) è necessario che il sistema imposti $o\_done = '1'$ (le altre uscite rimarranno nulle);
- **IDLE:** Il nostro componente avrà così concluso il primo ciclo di elaborazione. A questo punto si dovranno nuovamente "mascherare" a 0 le uscite e si ritornerà in fase di $idle$, rimanendoci fintanto che i_start non tornerà $= '1'$.

Figura 3: Esempio di un flusso di elaborazione.



ITERAZIONI SUCCESSIVE: Le iterazioni di computazione successive non differiscono dalla prima se non per la *FASE DI TRASMISSIONE IN USCITA*. Comprendiamo il come ciò debba avvenire analizzando la trasmissione in uscita della seconda iterazione di esecuzione:

- **TRASMISSIONE IN USCITA:** come per la prima iterazione, il sistema dovrà per un solo ciclo di clock porre $o\_done = '1'$ e trasmettere il dato i_mem_data sull'uscita o_Z_h (dove h è l'identificativo letto nella *FASE DI LETTURA DA MEMORIA* della seconda iterazione). Ricordando che k è l'identificativo letto nella prima iterazione, si avranno due casi possibili:
  a. Se $h = k$ →il dato dell'iterazione precedente verrà sovrascritto e il sistema trasmetterà i_mem_data sull'uscita $o\_Z_k = o\_Z_h$;
  b. $Se\ h \neq k$ →l'uscita o_Z_k manterrà il suo valore precedente mentre l'uscita o_Z_h trasmetterà il dato i_mem_data corrente.

  Le altre uscite rimarranno nulle.

In sintesi, nella fase di *TRASMISSIONE IN USCITA,* tutte le uscite mantengono il valore che avevano assunto nella medesima fase delle iterazioni precedenti, a meno che:

- Il valore sull'uscita venga sovrascritto (si riceve un identificativo già letto in un'iterazione precedente);
- Venga ricevuto un reset asincrono ($i\_reset = '1'$) che reimposti il sistema allo stato di default (si riparte in tal caso dall'iterazione 1).

2. ARCHITETTURA

2.1. Struttura concettuale dell'architettura

Lo schema dell'implementazione che ho deciso di adottare per il progetto è il seguente:

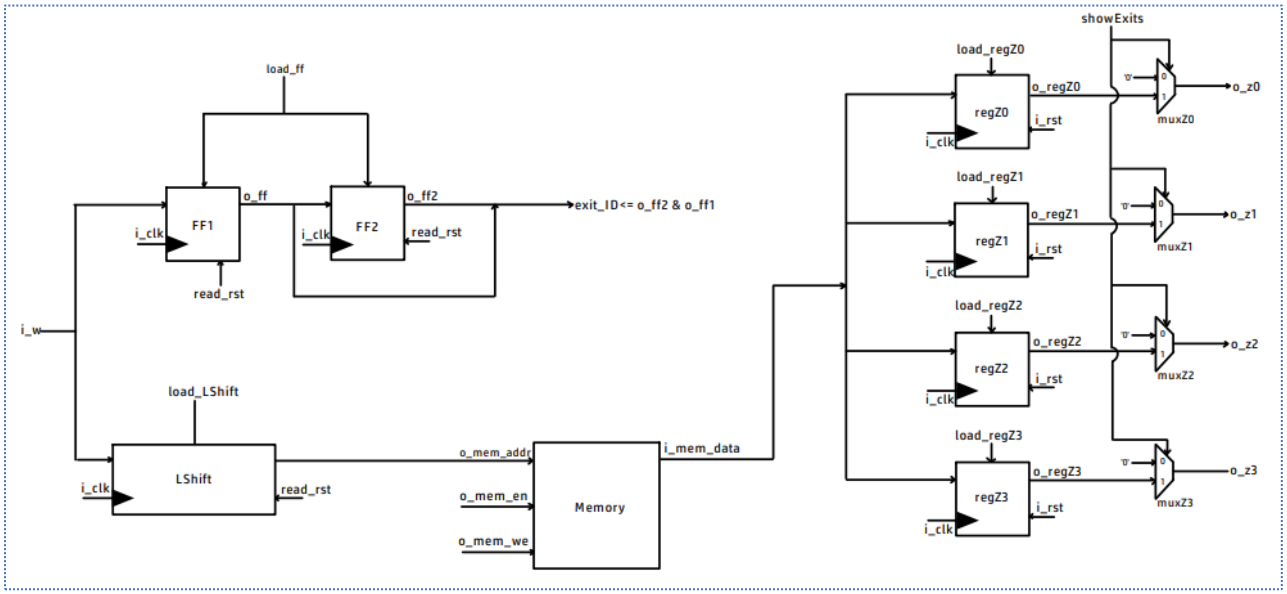


Figura 4: Schema dell'implementazione del sistema.

Analizziamone i vari moduli nel dettaglio.

GESTIONE DELLA LETTURA → La gestione della lettura dei bit in arrivo da i_w è affidata ai seguenti moduli:

- ***Modulo di lettura dell'identificativo di uscita:*** l'idea è quella di avere due flip-flop connessi tra loro che si attivano contemporaneamente e soltanto per i primi due cicli di lettura da i_w. In questo modo:
  - Nel primo ciclo di lettura, $FF2$ memorizzerà lo '0' proveniente da $FF1$ (che essendo all'inizio della computazione, conterrà il suo valore di default) mentre $FF1$ memorizzerà il $bit\#1$ proveniente da i_w;
  - Nel secondo ciclo di lettura, $FF2$ memorizzerà il $bit\#1$ in uscita da $FF1$ mentre quest'ultimo memorizzerà il $bit\#2$ proveniente da i_w.

  Concatenando i valori di uscita dei due flip-flop si otterrà quindi l'identificativo della porta su cui scrivere il dato che otterremo nei passaggi successivi. Chiameremo tale identificativo $exit\_ID$.

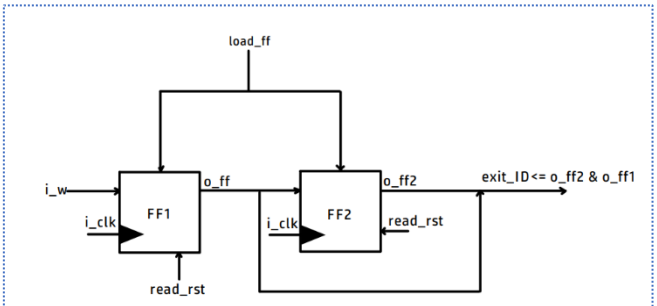


Figura 5: Modulo di lettura ID uscita.

  Nel codice, l'implementazione di tale modulo è effettuata attraverso l'uso dei segnali interni $\boldsymbol{o\_ff}$ e $\boldsymbol{exit\_ID}$, rappresentanti proprio la concatenazione tra i due flip-flop. La gestione di tali segnali è affidata alla macchina a stati, che verrà descritta successivamente.

- ***Modulo di lettura dell'indirizzo:*** per leggere i bit successivi al primo, la scelta più conveniente si è rivelata essere un Left-Shifter, poiché tramite l'adozione di quest'ultimo si rende indipendente il numero di cicli post-lettura dal numero di bit di indirizzo letti tramite i_w. Nello specifico, il funzionamento del modulo è il seguente:
  - All'inizio di ogni iterazione, lo Shifter conterrà 16 bit a 0;
  - Dopo che il sistema avrà letto i bit #1 e #2, si porrà $load_LShift = $ '1', in modo da attivare la lettura dello Shift Register;
  - Da tale istante (e fintanto che $i_start = $ '1) ogni bit letto verrà posizionato nella cella più a destra, e tutte le altre verranno shiftate a sinistra. In questo modo alla fine della lettura si otterrà l'indirizzo di memoria già espanso in segno.

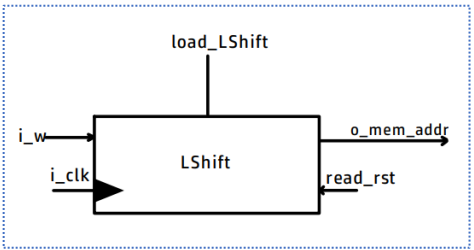


Figura 6: Modulo di lettura mem_addr.

Il modulo appena descritto è implementato tramite l'adozione del segnale interno $\boldsymbol{LShift}$, il cui comportamento è gestito dalla macchina a stati e verrà descritto in seguito.

NOTA: i moduli che compongono la gestione della lettura vanno resettati all'inizio di ogni iterazione di computazione. Per tale motivo, nella rappresentazione grafica, il loro segnale di reset è diverso da quello di sistema. Il valore di $read\_rst$ andrà impostato ad '1' sia quando si comincia un nuovo ciclo di esecuzione, sia quando arriva un reset asincrono dall'esterno. Anche il reset dei moduli di lettura sarà gestito dalla macchina a stati.

GESTIONE DELLE USCITE→ Ho scelto di affidare la gestione delle uscite al seguente insieme di componenti:

- ***Registri e Mux di uscita:*** i registri di uscita permettono di memorizzare i dati estratti dalla memoria, in modo che questi ultimi possano essere ancora disponibili nelle iterazioni di computazione successive alla loro ricezione. I Mux permettono invece di mascherare il valore delle uscite o_Z_i.
  Il funzionamento del modulo è il seguente:
  - Dopo aver completato la fase di lettura da i_w ed aver estratto il dato dalla memoria, la macchina a stati attiverà $load\_regZ_K$, dove k è l'identificativo di uscita letto precedentemente. In questo modo il dato verrà salvato nel registro apposito e rimarrà disponibile fintanto che tale registro non verrà sovrascritto/resettato;
  - Di norma il selettore dei Mux ($showExits$) sarà posto a '0'. Ogni volta che si vorranno mostrare i dati contenuti nei registri $regZ_i$, basterà porre $showExits <=$ '1'. In tal modo l'uscita dei Mux passerà da $others =>$ '0' al valore effettivo delle uscite dei registri.

  I Mux e i Registri del modulo di gestione delle uscite sono implementati attraverso delle component, che descriveremo più nel dettaglio in seguito.

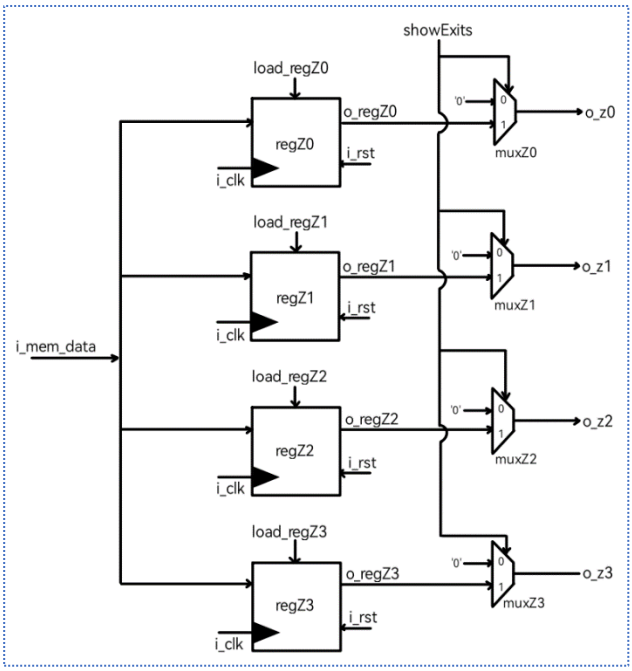


Figura 7: Modulo di gestione delle uscite.

2.2. Implementazione concreta dell'architettura

Nel paragrafo seguente si descriverà nel dettaglio il come la struttura concettuale è stata implementata in VHDL.

2.2.1. Segnali interni e component: I segnali interni che è stato necessario dichiarare sono:

```
--Flip flop per lettura ID_uscita.
signal o_ff: STD_LOGIC;
signal exit_ID: STD_LOGIC_VECTOR (1 downto 0);
--LeftShift per lettura indirizzo.
signal LShift: STD_LOGIC_VECTOR(15 downto 0);
--Registri regZ0, regZ1, regZ2 e regZ3 usati per memorizzare i dati da mostrare quando o_done='1'.
signal load_regZ0, load_regZ1, load_regZ2, load_regZ3 : STD_LOGIC;
signal o_regZ0, o_regZ1, o_regZ2, o_regZ3 : STD_LOGIC_VECTOR (7 downto 0);
--Multiplexer che maschera l'uscita.
signal showExits: STD_LOGIC;
--Segnali per macchina a stati.
type S is (reset,readID,read_addr,putInReg,done,idle);
signal state : S;
```

Le component usate sono invece:

```
--Registri di uscita.
REGZ0: Reg PORT MAP (DIN=>i_mem_data, Clock=> i_clk, Reset=> i_rst, Load=>load_regZ0, Q=>o_regZ0);
REGZ1: Reg PORT MAP (DIN=>i_mem_data, Clock=> i_clk, Reset=> i_rst, Load=>load_regZ1, Q=>o_regZ1);
REGZ2: Reg PORT MAP (DIN=>i_mem_data, Clock=> i_clk, Reset=> i_rst, Load=>load_regZ2, Q=>o_regZ2);
REGZ3: Reg PORT MAP (DIN=>i_mem_data, Clock=> i_clk, Reset=> i_rst, Load=>load_regZ3, Q=>o_regZ3);
--Multiplexer di uscita muxZ0, muxZ1, muxZ2, muxZ3.
MUXZ0: mux2to1 PORT MAP (A=>"00000000",B=>o_regZ0,SEL=>showExits,Z=> o_z0);
MUXZ1: mux2to1 PORT MAP (A=>"00000000",B=>o_regZ1,SEL=>showExits,Z=> o_z1);
MUXZ2: mux2to1 PORT MAP (A=>"00000000",B=>o_regZ2,SEL=>showExits,Z=> o_z2);
MUXZ3: mux2to1 PORT MAP (A=>"00000000",B=>o_regZ3,SEL=>showExits,Z=> o_z3);
```

Sono stati anche effettuati i seguenti assegnamenti tra i segnali interni e quelli della $entity$:

```
--Collego i segnali interni ai corrispondenti segnali dell'interfaccia esterna.
o_mem_addr<=LShift;
o_done<=showExits;
```

In questo modo l'uscita dello $shifter$ rappresenterà l'indirizzo da porre alla memoria, mentre cambiando il valore di $showExits$ si maschereranno/mostreranno le uscite e si cambierà di conseguenza il valore di o_done.

2.2.2. Macchina a stati: le interazioni tra segnali e component sono governate dalla **FSM**. Descriviamone il funzionamento:

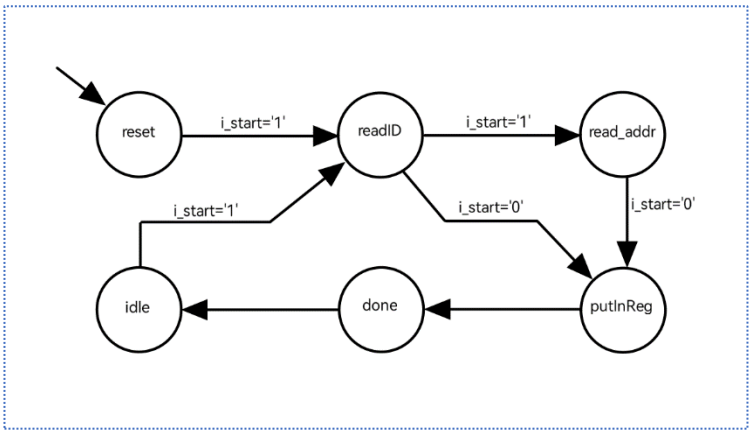


Figura 8: diagramma degli stati.

- $\boldsymbol{reset}$ → è lo stato di inizio. Qui tutti i segnali vengono posti ai loro valori di default nell'istante iniziale di esecuzione. Si uscirà da tale stato solo se $i\_start = '1'$ e vi si ritornerà ogni qual volta arriverà un reset asincrono, qualsiasi sia lo stato corrente in cui ci si trova.
- $\boldsymbol{readID}$ → è lo stato di lettura dell'identificativo dell'uscita. In questo stato:
    1. Verrà memorizzato in o_ff il primo bit in ingresso da i_w;
    2. Verrà ottenuto l'$exit\_ID$ attraverso il concatenamento: `exit_ID <= o_ff & i_w;`.

  Una volta letto l'ID dell'uscita, si avranno due possibilità:
    1. $i\_start = '1'$: in tal caso si proseguirà con la lettura dei bit di indirizzo, passando allo stato $read\_addr$;
    2. $i\_start = '0'$: questo è il caso in cui l'indirizzo di memoria è composto da soli '0'. In tal caso non sarà necessario passare alla fase di lettura di indirizzo poiché $LShift$ viene resettato ad ogni inizio iterazione e quindi sarà già composto da soli zeri. Si può in questo caso direttamente passare alla fase di lettura da memoria (stato $putInReg$).
- $\boldsymbol{read\_addr}$ → in questo stato vengono letti gli n bit di indirizzo (con $1 \leq n \leq 16$). Si resterà in tale stato fintanto che $i\_start = '1'$. Quando invece tale segnale passerà a 0, si andrà nello stato $putInReg$.
  Lo Shifter è implementato, come detto precedentemente, attraverso il segnale $LShift$. In particolare:
    - Il primo bit di indirizzo verrà posto nella cella più a destra del vettore $LShift$;
    - Dal secondo bit in poi, si effettuerà il concatenamento: `LShift<=LShift(14 downto 0)&i_w;`.

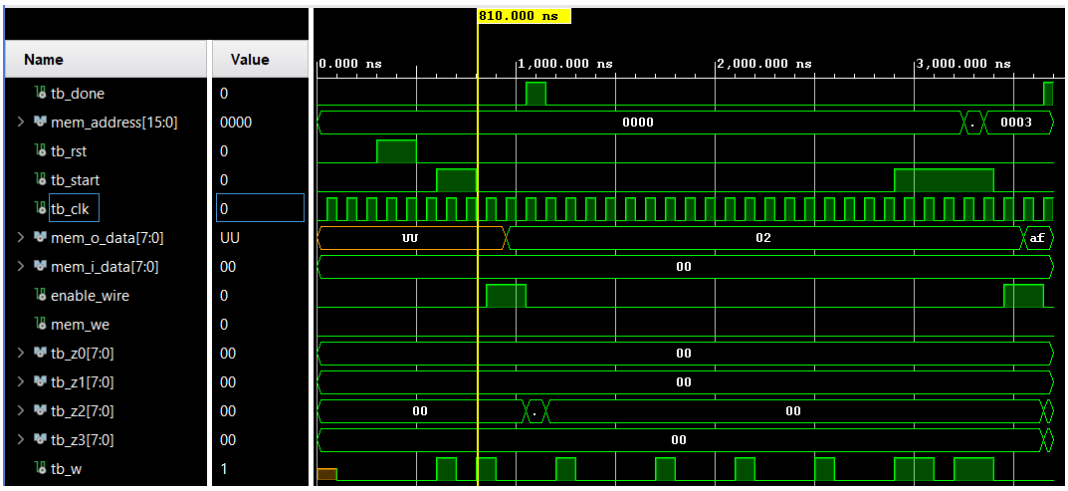
  In tal modo si otterrà l'indirizzo già espanso in segno.
- $\boldsymbol{putInReg}$ → è lo stato di lettura da memoria. Qui il nostro sistema attiverà dapprima la comunicazione con la memoria ($o\_mem\_en = '1'$) così da leggere i_mem_data e memorizzerà poi il dato letto nel registro indicato dall'$exit\_ID$ ponendo $= '1'$ il corrispondente $load\_regZ_i$. Si passerà poi incondizionatamente allo stato $done$.
- $\boldsymbol{done}$ → come si evince dal nome, $done$ è il segnale di fine computazione. Qui la FSM imposterà $showExits = '1'$, mostrando così le uscite dei registri $regZ_i$ (in tal modo anche o_done sarà $= '1'$). Si passerà poi incondizionatamente allo stato di $idle$.
- $\boldsymbol{idle}$ → in tale stato si attenderà che i_start ritorni $= '1'$ per ricominciare una nuova computazione (passaggio a $readID$). Questo stato è molto simile a $reset$, ma differisce con quest'ultimo per il fatto che in $idle$ vengono resettati solo i moduli di lettura, mentre quello di gestione delle uscite non subisce modifiche.

3. RISULTATI SPERIMENTALI

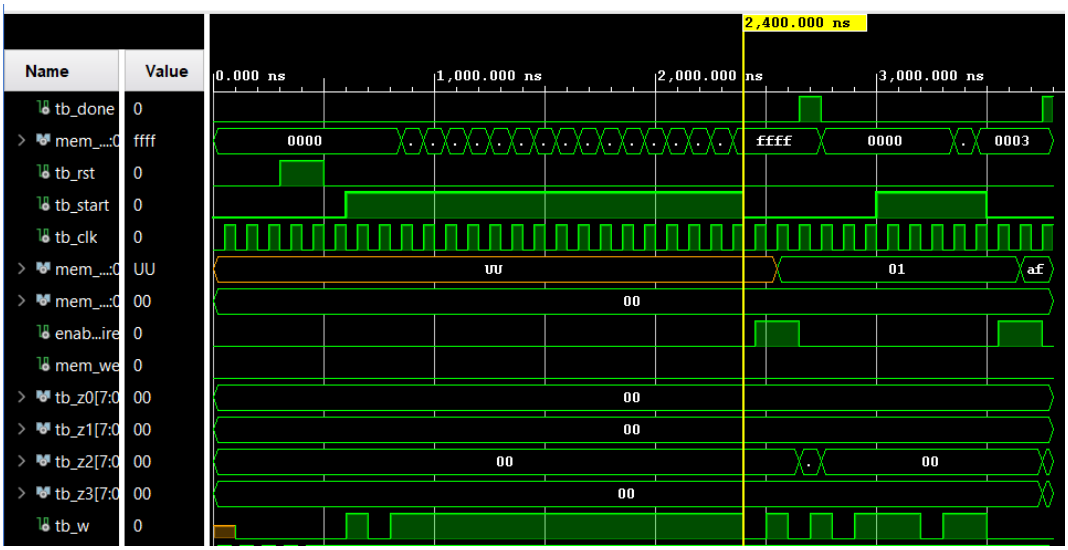
3.1. Simulazioni

Al fine di verificare la correttezza dell'implementazione, sono stati effettuati diversi test che ne provassero il corretto funzionamento nei possibili edge-cases. Di seguito se ne riportano gli esiti:

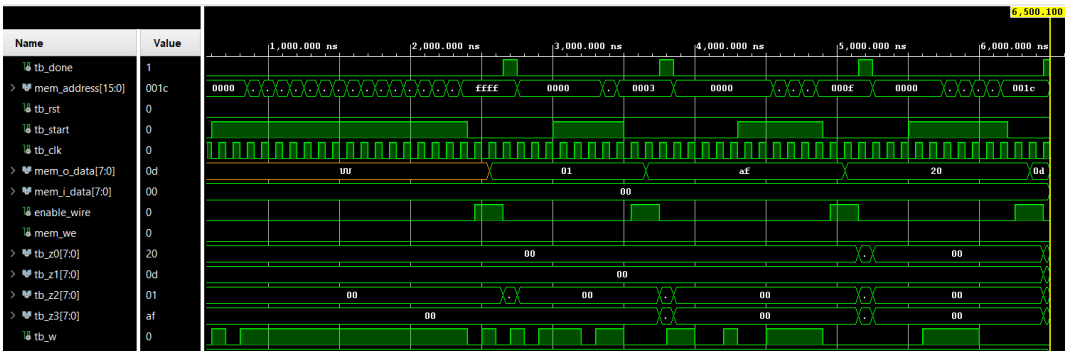
1. Caso in cui la stringa seriale è lunga solo 2 bit (lettura di 0 bit di indirizzo di memoria):



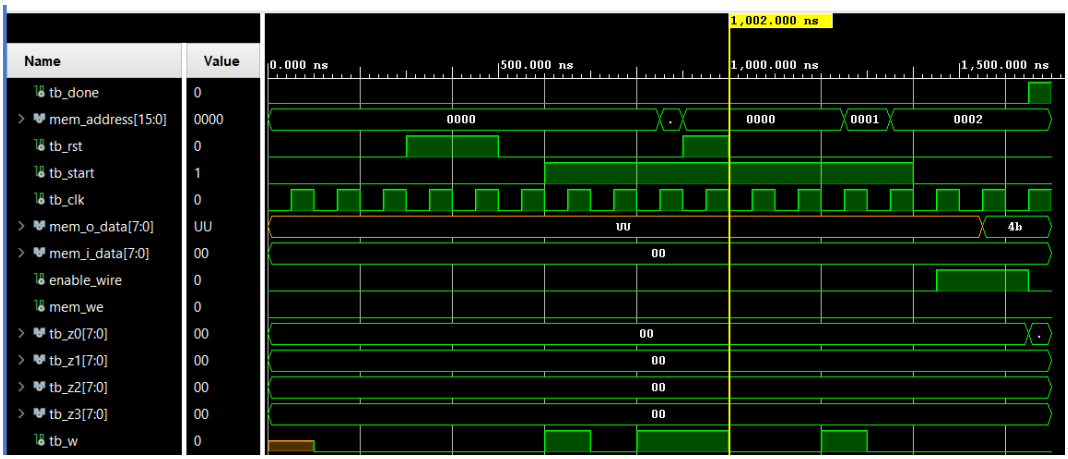
2. Caso in cui la stringa seriale è lunga 18 bit (lettura di esattamente 16 bit di indirizzo di memoria):



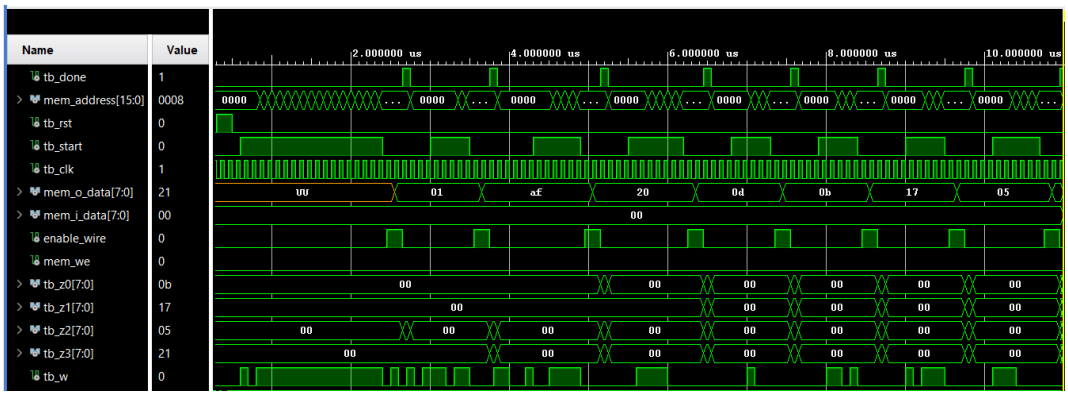
3. Scrittura su tutti i registri $regZ_i$ di uscita:



4. Reset asincrono del sistema: Ho effettuato il reset in diversi istanti di esecuzione del sistema (pre-lettura, post-lettura, …). La figura di seguito riporta uno dei test citati:



5. Sovrascrittura di ognuno dei registri $regZ_i$ di uscita:



3.2. Report di sintesi

Sono inoltre stati eseguiti i necessari controlli sul componente sintetizzato, verificando che non fossero stati involontariamente inferiti latch e che il periodo di clock massimo concesso (100ns) fosse compatibile col periodo di esecuzione del sistema.

Entrambe le verifiche hanno restituito esito positivo.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	40	0	0	134600	0.03
LUT as Logic	40	0	0	134600	0.03
LUT as Memory	0	0	0	46200	0.00
Slice Registers	60	0	0	269200	0.02
Register as Flip Flop	60	0	0	269200	0.02
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

Figura 9: report di utilizzo.

```
Slack (MET) :             97.530ns  (required time - arrival time)
  Source:                 FSM_sequential_state_reg[1]/C
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@5.000ns period=100.000ns})
  Destination:            LShift_reg[0]/CE
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@5.000ns period=100.000ns})
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            100.000ns  (clock rise@100.000ns - clock rise@0.000ns)
  Data Path Delay:        2.088ns  (logic 0.751ns (35.967%)  route 1.337ns (64.033%))
```

Figura 10: report temporale.

4. CONCLUSIONI

Il sistema ha passato tutti i test a cui è stato sottoposto sia in pre-sintesi che in post-sintesi (functional e timing simulation).

La principale criticità riscontrata durante lo sviluppo dell'implementazione è stata la sincronizzazione tra i segnali di $load$ delle varie componenti (inizialmente avevo implementato anche i $flip\ flop$ e lo $shifter$ tramite delle component esterne). Tale contesto portava lo $shifter$ a leggere 1 bit in meno/effettuare 1 shift in più, a seconda del ciclo di clock nel quale si poneva $load_LShifter = $ '1'. Dopo qualche giorno di lavoro ho deciso di rappresentare lo $shifter$ e i due $flip - flop$ di lettura tramite i segnali interni descritti nei precedenti capitoli della relazione. Ho deciso inoltre di unificare i processi rappresentanti la macchina a stati in uno soltanto.

In tal modo mi è stato possibile risolvere i problemi di sincronia, col vantaggio aggiuntivo di poter avere "sott'occhio", in maniera condensata e sintetica, sia la selezione degli stati che ciò che si effettua in ognuno di essi.

L'altro lato della medaglia consiste nel fatto che, a causa di tale implementazione della FSM, non ho potuto usare le "assegnazioni di default" a inizio processo poiché, dato che l'unico processo costituente la macchina a stati si triggera ad ogni fronte del segnale i_clk, i valori di default sarebbero stati impostati per ogni $falling\_edge(i\_clk)$. Tuttavia, l'aver dovuto specificare esplicitamente il valore dei pochi segnali per ognuno degli stati non si è rivelato un problema dato il numero limitato degli stessi.

Ho ritenuto quindi in conclusione, che i vantaggi di tale approccio superassero il singolo svantaggio (per questo specifico caso) e ho deciso di mantenere l'implementazione descritta.