Prova Finale (Progetto di Reti Logiche)
Prof. Salice Fabio - Anno 2022/2023

Chiara Fossà 10724941 957087
Marina Mastroleo 10766089 961107

1 Indice

POLITECNICO
MILANO 1863

2 Introduzione

2.1 Scopo del progetto

Questo progetto consiste nel realizzare un componente hardware descritto in VHDL, in grado di interfacciarsi con la memoria. Tramite un ingresso seriale da 1 bit, vengono date come input delle indicazioni riguardanti quale canale di uscita selezionare e l'indirizzo di memoria a cui bisogna accedere. In uscita ha quattro canali da 8 bit in parallelo. La memoria, fornita nella specifica, restituisce il contenuto dell'indirizzo che le è stato passato. Il dato ottenuto dalla memoria deve essere preso e indirizzato verso il canale di uscita corretto.

2.2 Specifiche generali

Analizzando la struttura più nello specifico:

- il modulo da implementare deve avere **due ingressi primari** da **1 bit**:

  1. *i_w* che fornisce **2 bit di intestazione** che identificano il canale di uscita, seguiti dai **bit di indirizzo** della memoria che possono variare **da 0 fino a 16**. L'indirizzo di memoria deve essere sempre di **16 bit**, quindi se dovesse succedere che la sua lunghezza è inferiore a 16 bit, l'indirizzo viene esteso con 0 sui bit più significativi.

  2. *i_start* stabilisce quando la sequenza di *i_w* è valida. Quest'ultimo rimane alto per **almeno 2 cicli di clock** e **massimo per 18 cicli di clock** (2 bit del canale + 16 bit al massimo per indirizzare la memoria).

- vi sono poi **cinque uscite primarie**:

- **quattro** canali da **8 bit** che trasmetteranno il dato preso dalla memoria (*o_z0*, *o_z1*, *o_z2*, *o_z3*) e **una** da **1 bit** (*o_done*).

- Quando il segnale *o_done* è basso **tutti i canali di uscita devono essere a zero**. I valori sui canali di uscita sono visibili solo quando il valore di *o_done* è alto. Questi valori, ad eccezione di quello sul cui canale viene mandato il nuovo dato preso dalla memoria, rimangono inalterati. Contemporaneamente alla scrittura del messaggio sul canale, il segnale *o_done* passa da 0 a 1 e rimane attivo per un solo ciclo di clock. Il risultato è prodotto in meno di 20 cicli di clock (da quando *i_start* torna a 0).

- Quando viene dato il segnale di reset *i_rst*, il modulo viene reinizializzato. Prima del primo *i_start*=1 verrà sempre dato *i_rst*. Tutte le volte in cui il segnale *i_start* diventerà nuovamente alto invece, non si dovrà attendere il reset del modulo.

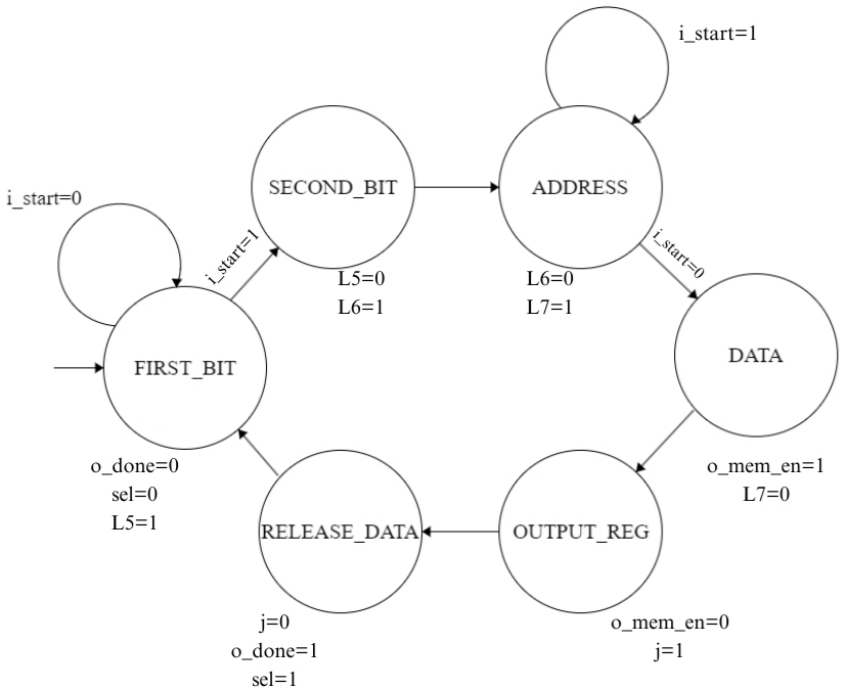


Figura 1: **FSM** del modulo

2.3 Esempio di funzionamento

Un caso di esempio per comprendere meglio il funzionamento del componente hardware spiegato sopra può essere il seguente:

Come mostrato nella Figura 2, quando il segnale *i_start* è alto, vengono elaborati i dati forniti sul segnale *i_w*. I primi 2 bit definiscono il canale di uscita ('10' → l'uscita sarà *o_z2*) e i restanti definiscono l'indirizzo di memoria ('110' → l'indirizzo sarà '0006').

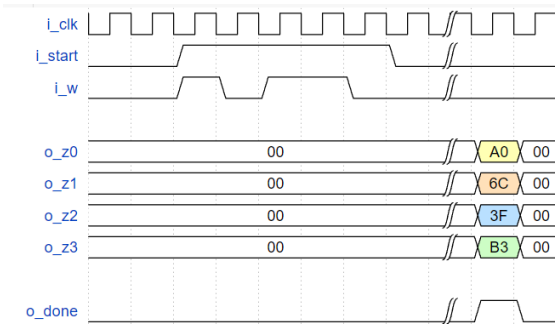


Figura 2: Waveform di esempio

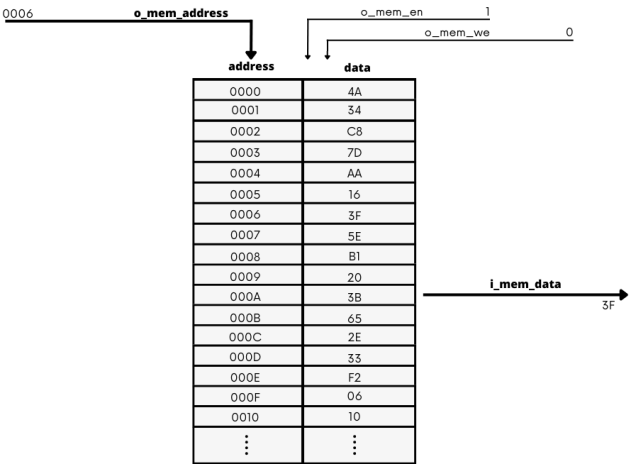


Figura 3: Esempio di memoria

La memoria, rappresentata in Figura 3, restituisce il dato (3F) contenuto all'indirizzo '0006'. Si può notare che contemporaneamente alla scrittura del messaggio '3F' sul canale, il segnale *o_done* diventa alto e rimane attivo per un solo ciclo di clock.

2.4 Interfaccia del componente

Il componente da descrivere possiede 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 perchè sia pronta per ricevere i dati.
- *i_start* è il segnale di START generato dal Test Bench.
- *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 da 8 bit.
- *o_done* è il segnale di uscita che comunica la fine dell'elaborazione e la presenza di dati sui canali d'uscita.
- *o_mem_addr* è il segnale di uscita che manda l'indirizzo alla memoria.
- *i_mem_data* è il segnale 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 che in scrittura) di attivarsi.
- *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 (in questa implementazione WRITE ENABLE sarà sempre a 0 perchè non dobbiamo scrivere in memoria).

3 Architettura

3.1 Datapath

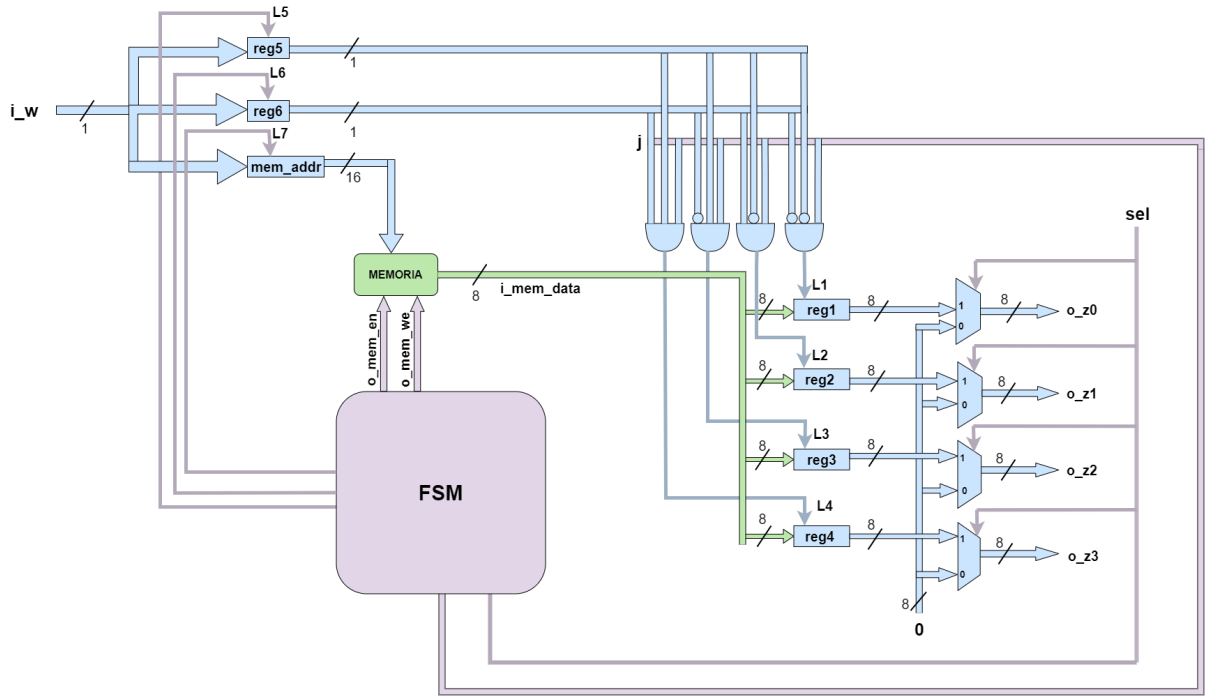


Figura 4: Datapath realizzato

Nel datapath che abbiamo realizzato è possibile notare la presenza dei seguenti componenti:

- *i_w* è il segnale di ingresso primario seriale da 1 bit.
- Questo segnale è collegato a tre registri: **reg5** , **reg6** e **mem_addr**, ciascuno con un suo segnale di selezione (rispettivamente **L5**, **L6**, **L7**) che, se alti, permettono di scrivere nel registro associato. Il primo bit che identifica l'uscita viene salvato in **reg5**, il secondo bit in **reg6**, mentre in **mem_addr** (un registro a scorrimento SIPO) vengono salvati i 16 bit che rappresentano l'indirizzo di memoria dove bisogna andare a prendere il dato.
- E' anche presente il componente che rappresenta la memoria. Il segnale *o_mem_we* vale sempre 0 poichè non ci viene mai chiesto di scrivere in memoria. Invece, il segnale *o_mem_en* viene posto a 1 dalla **FSM** non appena *i_start* si abbassa, cioè quando termina la lettura dei bit di *i_w*. Alzando *o_mem_en* diventa dunque possibile captare il dato presente nell'indirizzo di memoria che le passiamo tramite il canale da 16 bit, il quale connette il registro **mem_addr** alla memoria. Il dato presente negli indirizzi di memoria è sempre da 8 bit e viene indirizzato lungo il canale *i_mem_data*.
- Sono poi presenti 4 altri registri denominati **reg1**, **reg2**, **reg3**, **reg4**, ciascuno con un proprio segnale di selezione (rispettivamente **L1**, **L2**, **L3** ed **L4**). Per poter capire in quale registro salvare il dato preso da memoria , cioè per capire quale dei segnali tra **L1**, **L2**, **L3**, **L4** attivare, sono utilizzate delle porte logiche **AND** che svolgono la funzione AND tra i primi due bit che identificano il canale d'uscita e il segnale "**j**". Quest'ultimo viene posto ad 1 dalla **FSM** solo quando si è conclusa l'operazione di

estrarre il dato dalla memoria per segnalare ai registri la presenza di un nuovo dato su *i_mem_data*. Non appena si inizia a mandare il dato sul canale d'uscita corretto, j viene riportato a zero dalla **FSM**.

- Infine sono presenti quattro multiplexer che hanno due ingressi, un'uscita e un segnale di selezione "**sel**" il cui valore viene settato nella **FSM**. Gli ingressi di ciascun multiplexer sono il segnale relativo al registro a cui è associato (**reg1**, **reg2**, **reg3**, **reg4**) e un segnale fisso a 0000 0000. L'uscita di ciascun multiplexer corrisponde invece ad uno tra i quattro canali d'uscita (*o_z0*, *o_z1*, *o_z2*, *o_z3*). Tramite il segnale sel, che svolge la funzione di indirizzamento, è possibile mandare il dato preso da memoria sul canale d'uscita corretto.

3.2 Stati della FSM

Gli stati che compongono la **FSM** che abbiamo realizzato (Figura 1) sono i seguenti:

1. FIRST_BIT

Stato iniziale in cui si attende che il segnale *i_start* venga alzato. Quando *i_start* viene alzato, viene letto il primo bit che andrà a definire il canale e registrato in **reg5** grazie al segnale **L5** che viene posto uguale ad 1.

2. SECOND_BIT

Con *i_start* ancora a 1 viene letto il secondo bit che andrà a definire il canale e registrato in **reg6** grazie al segnale **L6** che viene posto uguale ad 1.

3. ADDRESS

Finchè *i_start* è tenuto ad 1, vengono letti i bit che definiscono l'indirizzo e vengono registrati nel registro SIPO a scorrimento **mem_addr** grazie al segnale **L7** che viene posto pari ad 1.

4. DATA

Quando *i_start* viene abbassato, viene inviato l'indirizzo contenuto in **mem_addr** alla memoria, la quale lo leggerà e restituirà il dato richiesto.

5. OUTPUT_REG

Il dato restituito dalla memoria viene memorizzato nel registro **reg1**, **reg2**, **reg3** o **reg4** a seconda dei valori contenuti in **reg5** e **reg6** e in base al segnale "j", il quale comunica ai registri che contengono i dati dei canali che è presente un nuovo dato che deve essere registrato da uno di loro.

6. RELEASE_DATA

o_done, a cui è assegnato il valore del segnale *sel*, viene alzato e sui canali sono trasmessi i dati contenuti in **reg1**, **reg2**, **reg3** e **reg4**.

3.3 Scelte progettuali

Il componente hardware descritto in VHDL è dotato di 5 processi. Ciascuno viene eseguito solo quando cambiano dei segnali nella sensitivity list. Abbiamo deciso di usare 3 processi per rappresentare la FSM (una macchina di Moore) per ridurre la sensitivity list di ciascun processo e dunque avere una simulazione più veloce. Invece gli altri 2 processi rappresentano il funzionamento del datapath.

1. Il primo processo rimanda allo stato iniziale **FIRST_BIT** quando il segnale *i_rst* viene alzato, altrimenti lo stato corrente prende il valore dello stato successivo.

2. Il secondo processo specifica la funzione dello stato successivo in base allo stato corrente e a *i_start*, che è l'ingresso primario della **FSM**. Questo processo è combinatorio. Quando la **FSM** si trova nello stato iniziale **FIRST_BIT**, se il segnale *i_start* è alto va nel prossimo stato (**SECOND_BIT**), altrimenti rimane nello stato iniziale. Quando la FSM si trova nello stato **SECOND_BIT** va nello stato successivo (**ADDRESS**). Quando la FSM si trova nello stato **ADDRESS** vi rimane se *i_start*=1, altrimenti va nello stato successivo (**DATA**). Se invece la **FSM** si trova negli ultimi tre stati, va semplicemente nello stato successivo.

3. Il terzo processo specifica la funzione d'uscita, la quale dipende solo dallo stato corrente. Anche questo processo è combinatorio. Abbiamo dato un valore di default a tutti i segnali. Quando la **FSM** si trova nello stato iniziale **FIRST_BIT** assegna **L5**=1, quando è nello stato **SECOND_BIT** assegna L6=1 e riporta L5=0. Quando è nello stato **ADDRESS** assegna **L7**=1 e riporta **L6**=0. Quando è in **DATA** riporta **L7**=0 e assegna *o_mem_en*=1. Quando è in **OUTPUT_REG** riporta *o_mem_en*=0 e assegna *j*=1. Infine quando è in **RELEASE_DATA** riporta *j*=0 e assegna *sel*=1.

4. Il quarto processo azzera tutti e quattro i canali di uscita se *o_done*=0, altrimenti mette nei canali di uscita il valore contenuto nei registri **reg1**, **reg2**, **reg3** e **reg4**. Se si verifica il reset del modulo, azzera i registri **reg1**, **reg2**, **reg3** e **reg4**. Infine mette nel registro corretto tra **reg1**, **reg2**, **reg3** o **reg4** il dato estratto dalla memoria a seconda di quale segnale tra **L1**, **L2**, **L3**, **L4** vale 1. Questi ultimi quattro segnali vengono calcolati tramite delle porte logiche **AND**.

5. Il quinto processo azzera i tre registri **reg5**, **reg6** e **mem_addr** se si verifica il reset del modulo. Se *o_done*=1 azzera il registro **mem_addr**, che è collegato alla memoria tramite il canale *o_mem_addr*. Infine assegna il valore di *i_w* nel registro corretto tra **reg5**, **reg6**, **mem_addr** a seconda di quale segnale tra **L5**, **L6**, **L7** viene attivato dalla **FSM**.

4 Risultati sperimentali

4.1 Report di sintesi

Abbiamo verificato che il dispositivo è perfettamente sintetizzabile, in quanto la sintesi è andata a buon fine. Abbiamo usato una FPGA xc7a200tfbg484-1.

Utilization report

Digitando il comando "*report_utilization*" sulla Tcl Console, abbiamo potuto vedere che il nostro componente è stato sintetizzato in 39 celle dell' FPGA, sono stati usati 56 Flip Flop e non si sono verificati Latch.

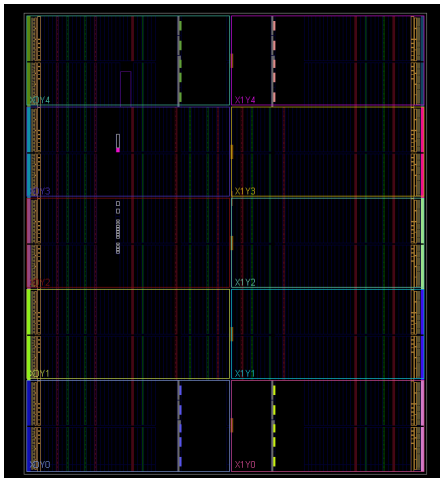


Figura 5: Componente sintetizzato

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	39	0	0	134600	0.03
LUT as Logic	39	0	0	134600	0.03
LUT as Memory	0	0	0	46200	0.00
Slice Registers	56	0	0	269200	0.02
Register as Flip Flop	56	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 6: Tcl Console: "report_utilization"

Timing report

Digitando il comando "*report_timing*" sulla Tcl Console, abbiamo potuto vedere che lo slack, ovvero il tempo che avanza al nostro componente è pari a 96.991 ns

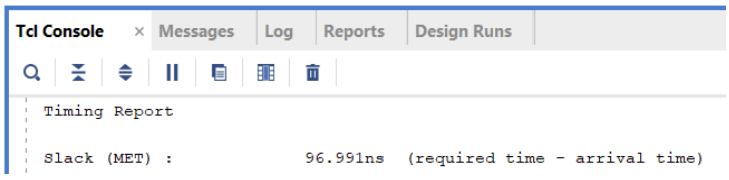
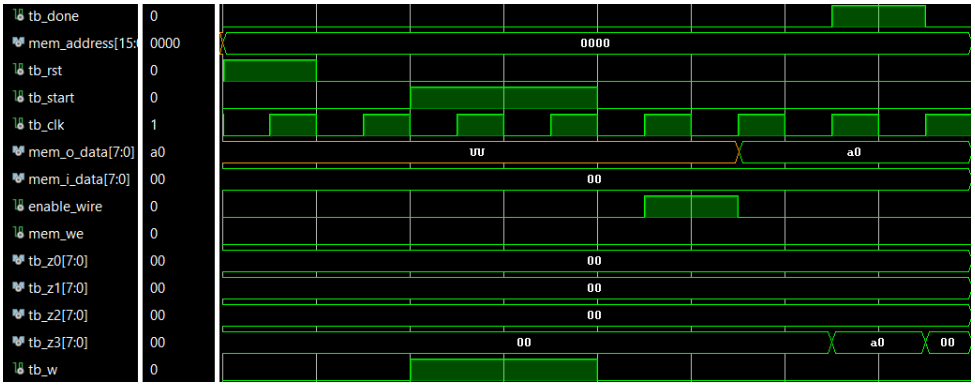


Figura 7: Tcl Console: "report_timing"

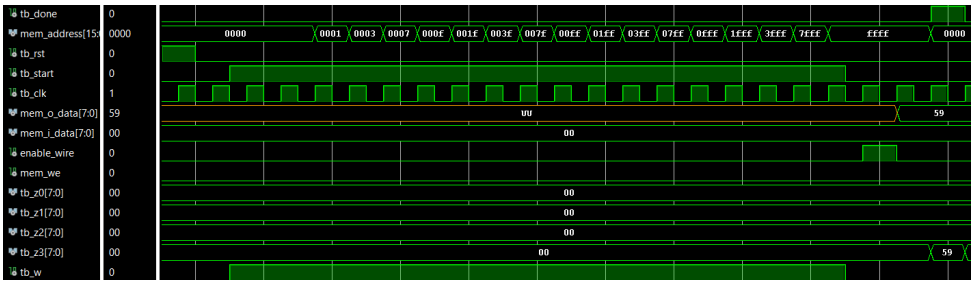
4.2 Simulazioni

Tutti i Test Bench che ci sono stati forniti sono andati a buon fine superando la Behavioral Simulation e la Post-Syntesis Functional Simulation. Abbiamo creato inoltre i seguenti Test Bench per verificare dei particolari casi limite:

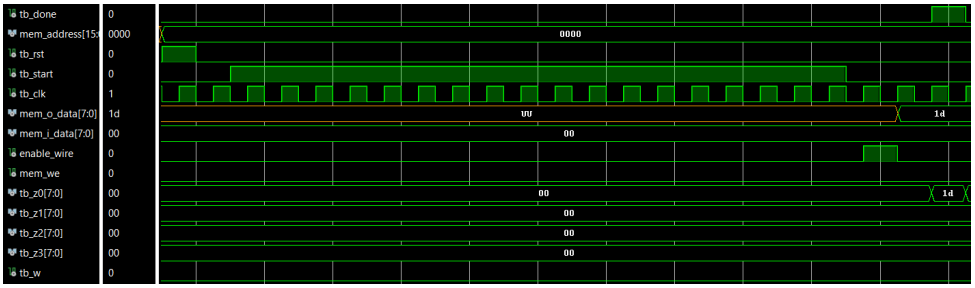
- **Senza indirizzo (solo 2 bit sul segnale i_w):**
  Vengono inviati sul segnale i_w il numero minimo di bit, ovvero solo i 2 bit che determinano il canale d'uscita. Questo test verifica che non inviando sul segnale i_w nessun bit per l'indirizzo, la memoria andrà a leggere il dato presente all'indirizzo 0000.



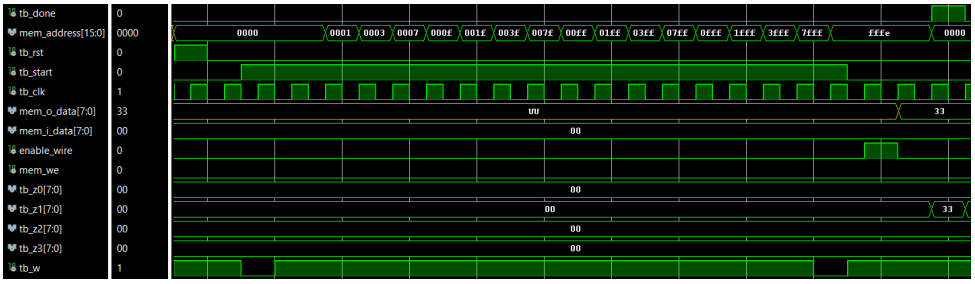
- **Numero massimo di bit (18 bit sul segnale i_w) e tutti 1:**
  Vengono inviati sul segnale i_w il numero massimo di bit (2 che determinano il canale d'uscita più i 16 che determinano l'indirizzo). Inoltre questi 18 bit sono tutti a 1. Questo test mostra che inviando sul segnale i_w il numero massimo di bit e che inviando l'indirizzo massimo, questo viene processato correttamente dallo shift register, che la memoria lo riceve correttamente e che il dato viene indirizzato sul canale corretto.



- **Tutti 0:**
  Al contrario del testbench precedente questo test verifica che, inviando un indirizzo della lunghezza massima che rappresenta l'indirizzo minimo (18 bit tutti a 0), questo viene processato correttamente dallo shift register, la memoria lo riceve correttamente e il dato viene indirizzato sul canale corretto.

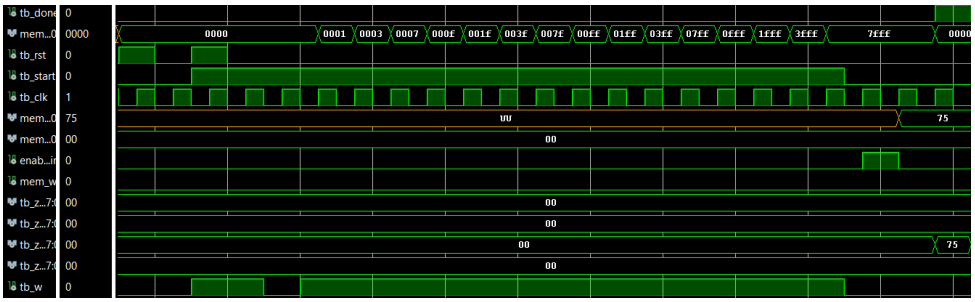


- **Non legge quando non deve:**
  In questo test vengono inviati dei dati sul segnale i_w quando il segnale i_start è basso. Verifica quindi che, quando il segnale i_start=0, i dati passati sul segnale i_w non vengono letti.

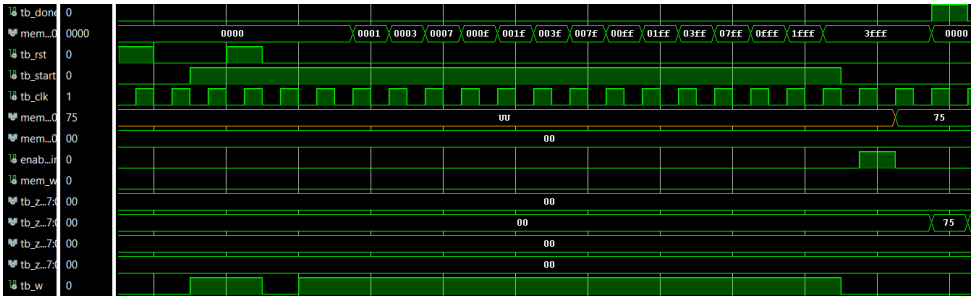


- **Reset in ogni stato:**
  Abbiamo verificato con dei Test Bench appositi il verificarsi del reset in ciascuno stato della FSM per controllare che il modulo venga reinizializzato perfettamente.

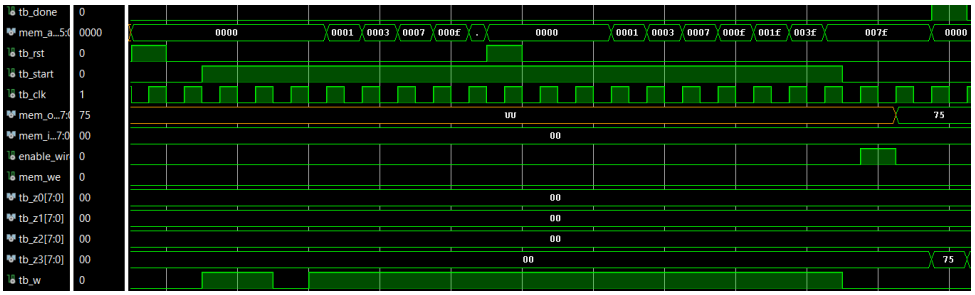
  Nello stato "FIRST_BIT":



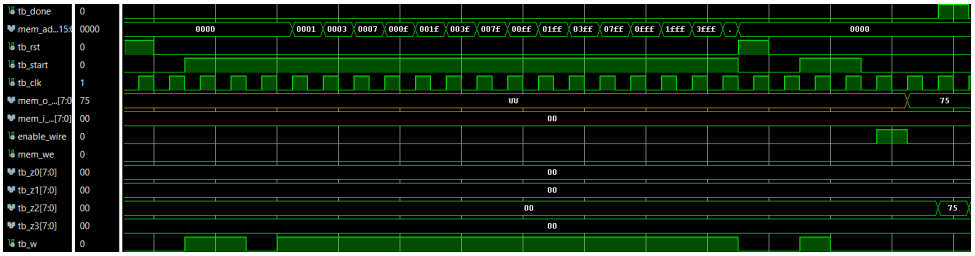
  Nello stato "SECOND_BIT":



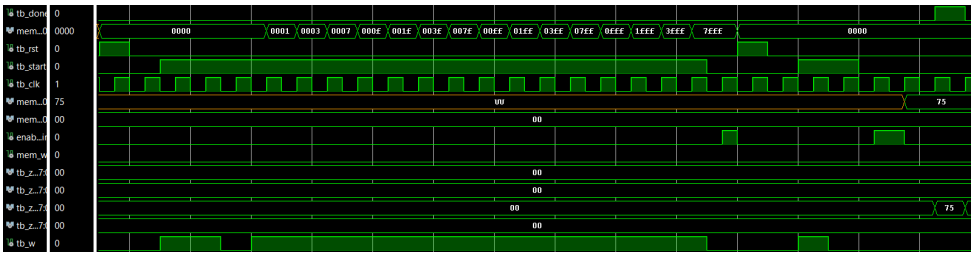
Nello stato “ADDRESS”:



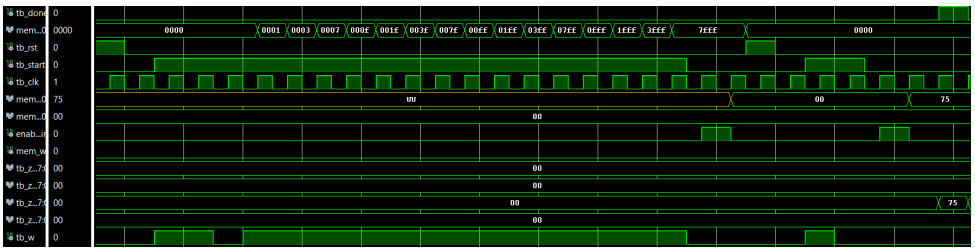
Nello stato “DATA”:



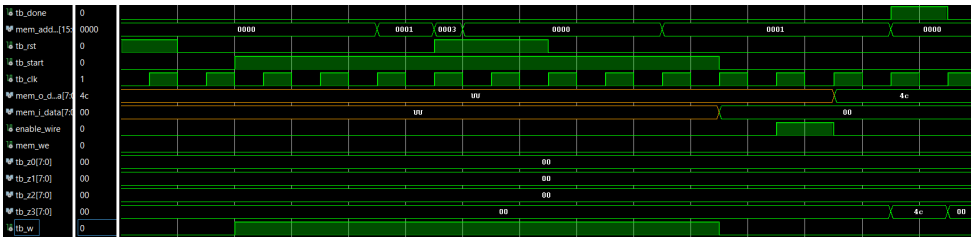
Nello stato “OUTPUT_REG”:



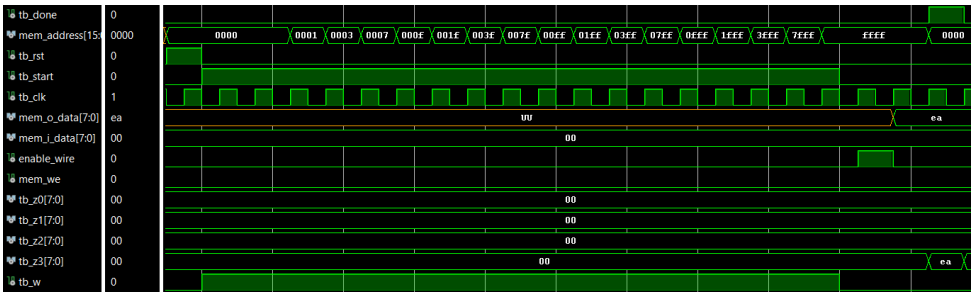
Nello stato “RELEASE_DATA”:



- **Reset a metà di un ciclo di clock:**
  In questo test abbiamo inviato un reset a metà di un ciclo di clock per verificare che il componente reinizializzi il modulo e che riprenda correttamente la lettura.



- **Segnale i_start consecutivo al reset:**
  In questo test abbiamo verificato il caso in cui il segnale i_start viene alzato subito dopo quello di reset.



4.3 Ottimizzazioni

Abbiamo attuato le seguenti ottimizzazioni delle nostre scelte implementative: inizialmente abbiamo identificato i vari processi. Per memorizzare correttamente l'indirizzo avevamo ipotizzato di usare un contatore per poter capire di quanti bit fare il padding dell'indirizzo, ma poi abbiamo deciso di utilizzare un registro a scorrimento SIPO inizialmente settato con 16 bit a 0, che va ad aggiungere in coda il nuovo bit ricevuto facendo scorrere il registro. Avevamo valutato l'ipotesi di usare un demultiplexer per instradare correttamente il dato uscente dalla memoria nel registro d'uscita corretto, ma poi abbiamo notato che potevamo ottenere lo stesso risultato sfruttando solamente i segnali di load dei registri per fare in modo che solo il registro corretto memorizzi il dato.

5 Conclusioni

Abbiamo iniziato disegnando il datapath, che è stato soggetto a diverse modifiche al fine di migliorare e ottimizzare sempre di più il componente. In seguito, siamo passate alla realizzazione della macchina a stati finiti e alla stesura del codice. Abbiamo analizzato il codice risolvendo i problemi che presentava la stesura iniziale. I Test Bench forniti ci sono stati d'aiuto per riflettere sul comportamento di ciò che stavamo realizzando. Abbiamo ulteriormente testato il funzionamento del codice creando 12 Test Bench che verificano dei casi limite e hanno dato esito positivo tutti al primo tentativo, è stata una grande soddisfazione. Svolgere questo progetto in coppia, è stato utile perchè ci ha permesso di condividere le nostre idee, generandone di più innovative rispetto a quelle che avremmo potuto avere individualmente e per vedere il progetto da punti di vista diversi. Riteniamo che questo progetto sia stata un'esperienza non solo interessante, ma anche molto formativa perché abbiamo potuto comprendere il funzionamento di un linguaggio che non è sequenziale, come quelli che abbiamo abbiamo studiato finora, ma abbiamo potuto imparare per la prima volta come scrivere del codice concorrente, in cui i vari processi funzionano contemporaneamente, andando quindi a realizzare un circuito elettronico.