POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Prova Finale di Reti Logiche

A.A. 2022/2023

Report

Simone Grandi (cod. persona 10741772)

Francesco Antonio Gangi (cod. persona 10793520)

Docente: Fabio Salice

Introduzione

La specifica della "Prova Finale (Progetto di Reti Logiche)" per l'Anno Accademico 2022/2023 chiede di implementare un modulo HW (descritto in VHDL) che si interfacci con una memoria e che rispetti le indicazioni riportate nella seguente specifica.

Ad elevato livello di astrazione, il sistema riceve indicazioni circa una locazione di memoria, il cui contenuto deve essere indirizzato verso un canale di uscita fra i quattro disponibili. Le indicazioni circa il canale da utilizzare e l'indirizzo di memoria a cui accedere vengono forniti mediante un ingresso seriale da un bit, mentre le uscite del sistema, ovvero i succitati canali, forniscono tutti i bit della parola di memoria in parallelo.

Il modulo da implementare ha due ingressi primari da 1 bit (W e START) e 5 uscite primarie. Le uscite sono le seguenti: quattro da 8 bit (Z0, Z1, Z2, Z3) e una da 1 bit (DONE). Inoltre, il modulo ha un segnale di clock CLK, unico per tutto il sistema e un segnale di reset RESET anch'esso unico.

Traduzione ed elaborazione della specifica

I bit sull'ingresso seriale W sono validi solamente quando START è uguale a 1, che rimane alto per un minimo di 2 cicli di clock e un massimo di 18. Supponiamo che START sia a 1 per 10 cicli, quindi leggiamo dall'ingresso W una sequenza di 10 bit, per esempio "0111011001".

I primi 2 bit (MSB) della sequenza ci indicano, convertendo in decimale, su quale dei quattro canali di uscita verrà propagato il dato. In questo caso, con "01111011001", l'uscita sarà propagata sul canale Z1.

I restanti bit servono per raggiungere l'indirizzo da 16 bit della memoria dove è localizzato il dato che dobbiamo propagare su Z1. Nel nostro esempio, escludendo i bit di intestazione, la sequenza in ingresso è più corta di 16 bit ed è pertanto necessario effettuare 0-padding per estendere l'indirizzo, aggiungendo in testa alla sequenza tanti zeri quanti ne servono per arrivare a 16 bit. Nel nostro esempio: 11011001 → 0000000011011001.

A questo punto il dato da 8 bit prelevato dalla memoria viene restituito al componente e viene inoltrato sul registro del canale concordato (Z1).

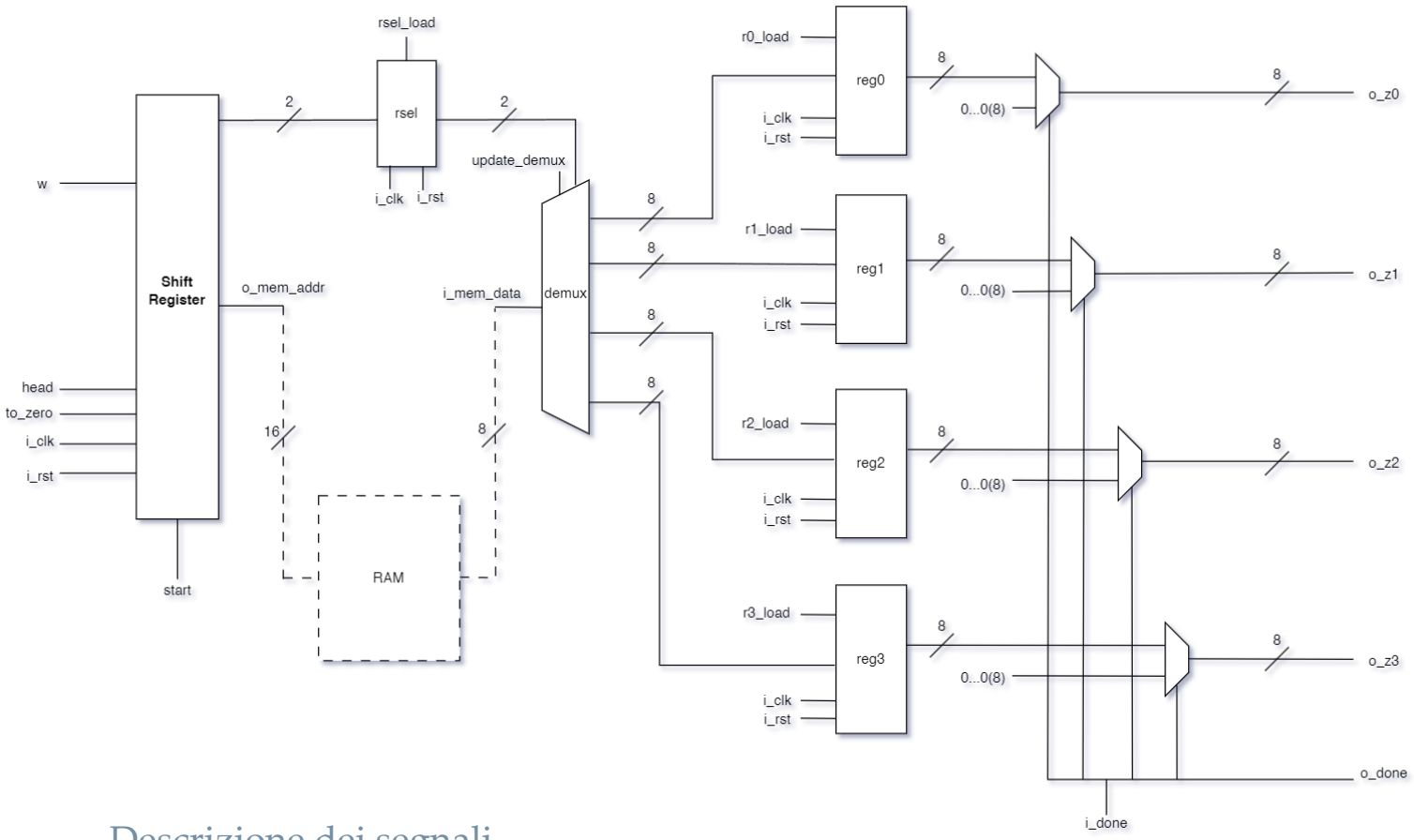
Fintanto che DONE è a 0, tutti i canali mostrano in uscita "00000000". Quando invece il segnale DONE è settato a 1 (per la durata di un singolo ciclo di clock) il canale selezionato mostrerà il dato prelevato dalla memoria. I restanti canali mostreranno l'ultimo dato propagato su di loro nelle letture precedenti; quindi, ogni canale tiene memoria dell'ultimo dato inviatogli. Se i canali non sono mai stati utilizzati mostreranno "00000000". Nel nostro caso, quando DONE vale 1, Z1 mostra in uscita gli 8 bit estratti dall'indirizzo di memoria "0000000011011001". Per cui, ogni qualvolta DONE commuterà a 1, verrà mostrato quel dato, fintanto che il canale Z1 non venga riutilizzato per propagare un nuovo valore in uscita.

Architettura

L'architettura implementata consta di un datapath che racchiude in esso tutte le componenti implementate, pilotate da alcune variabili interne. Queste variabili vengono a loro volta gestite da una macchina a stati.

Datapath

Di seguito è riportato lo schema del datapath:



Descrizione dei segnali

- *head:* è un segnale a 1 bit di supporto alla lettura del segnale W, e serve per separare i 2 bit di intestazione dai successivi bit di indirizzo. Esso è a 0 per i primi 2 cicli di clock di lettura della sequenza di ingresso, per poi andare a 1 successivamente "comunicando" la fine della lettura dell'header.
- *to_zero:* è un segnale a 1 bit utilizzato per azzerare interamente lo shift register. Alla fine di ogni elaborazione di una sequenza di ingresso, dopo che il dato viene propagato in uscita, il segnale va a 1 cosicché i vettori che ricevono l'input vengano azzerati per prepararsi a una nuova lettura.
- *update_demux:* è un segnale a 1 bit che comanda l'inserimento in ingresso al demux del nuovo dato letto da memoria.
- *i_done:* è un selettore a 1 bit che controlla l'uscita dei 4 multiplexer.

- *rsel_load:* è un selettore a 1 bit che controlla il caricamento del dato in ingresso al registro *rsel*.
- *r0_load:* è un selettore a 1 bit che controlla il caricamento del dato in ingresso al registro *reg0*.
- *r1_load:* è un selettore a 1 bit che controlla il caricamento del dato in ingresso al registro *reg1*.
- *r2_load:* è un selettore a 1 bit che controlla il caricamento del dato in ingresso al registro *reg2*.
- *r3_load:* è un selettore a 1 bit che controlla il caricamento del dato in ingresso al registro *reg3*.

Scelte implementative

Shift Register

Dal momento che l'ingresso W è seriale abbiamo optato per uno shift register SIPO (serial input-parallel output) a 18 bit per rendere l'ingresso parallelo, e pensato per inviare i primi 2 bit letti al selettore del demultiplexer e i restanti 16 alla memoria (il padding viene effettuato dallo shift register). Dato che il prelievo da memoria non è immediato, utilizziamo un registro *rsel* per salvare temporaneamente i bit di selezione del demultiplexer. Nella nostra implementazione VHDL abbiamo deciso di separare i 2 bit di intestazione dai 16 bit di indirizzo utilizzando due diversi **std_logic_vector**, denominati *addr* (da 16 bit, per l'indirizzo) e *sel_demux* (da 2 bit, per l'header). Il popolamento di questi ultimi è comandato dal segnale *head* descritto in precedenza. In particolare, il vettore *addr* è inizializzato a una sequenza di zeri che shiftano a sinistra di una posizione ogni qual volta in ingresso leggo un nuovo valore da inserire. Il risultato finale, qualora la sequenza fosse lunga meno di 16 bit, sarebbe un vettore contenente l'indirizzo con 0-padding automaticamente effettuato.

Demultiplexer

La scelta del demux è motivata dalla necessità di scegliere un canale di uscita specifico per l'invio del dato estratto dalla memoria. Solo dopo che il dato è stato prelevato ed è pronto ad essere inviato il segnale *update_demux* commuta a 1 attivando il demultiplexer, che riceve in ingresso il valore.

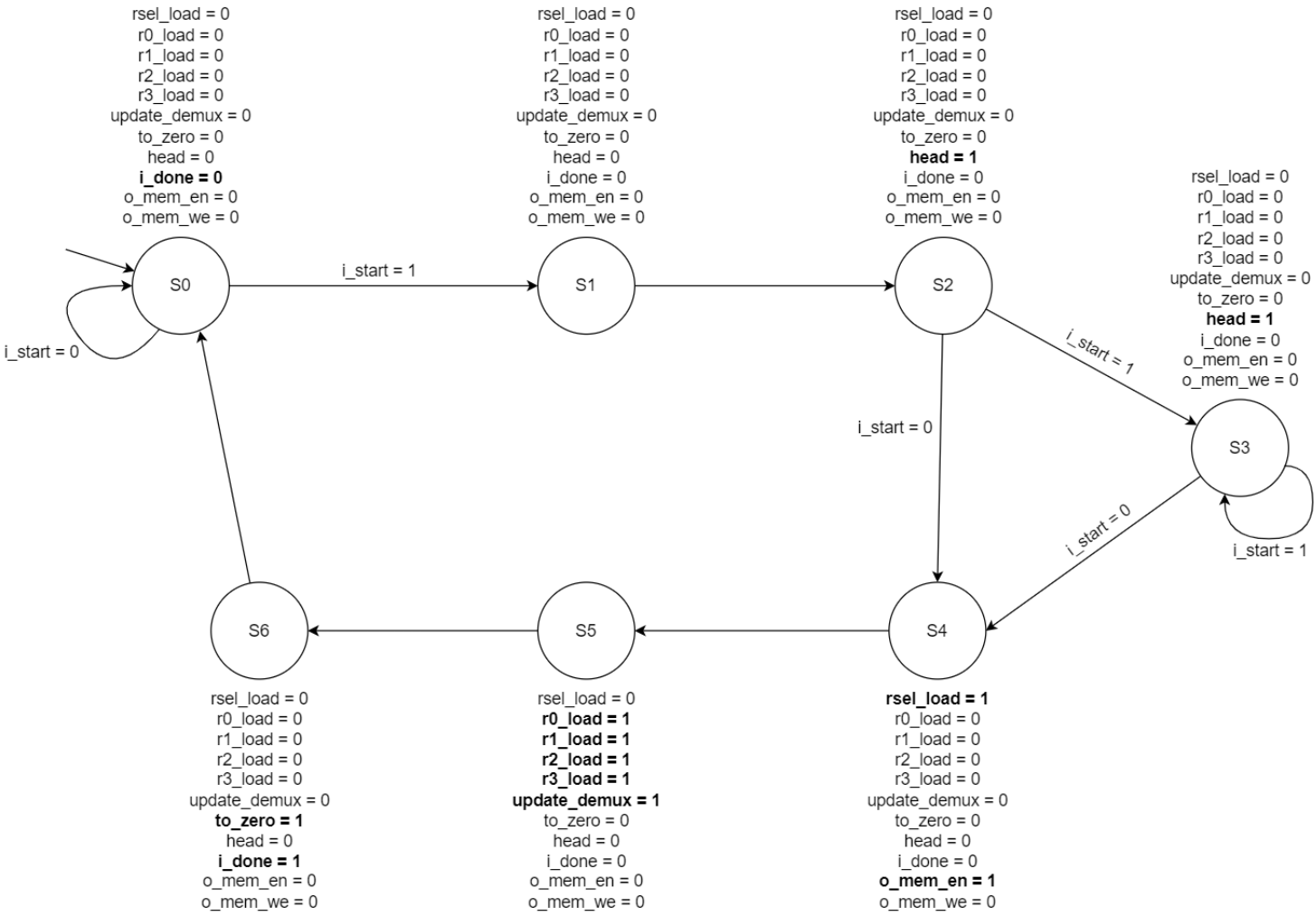
Registri/multiplexer

Dato che, da specifica, quando il segnale DONE è a 1 le uscite devono tenere memoria dei dati precedentemente inviati (se non modificate durante l'ultima elaborazione) per poterli propagare in uscita, abbiamo deciso di utilizzare una serie di registri. Il segnale DONE opera da selettore nei 4 multiplexer posti in corrispondenza delle uscite di questi registri in modo che, quando esso è a 1 viene propagato in uscita il dato salvato in ogni registro, mentre quando è a 0 viene propagata una sequenza di zeri.

Macchina a stati finiti

Per gestire correttamente il datapath implementato, abbiamo ideato una macchina a stati finiti che comandi le variabili interne del nostro componente per poter effettuare le varie operazioni di lettura, elaborazione e propagazione dei dati.

Di seguito lo schema della macchina a stati finiti:



Descrizione degli stati

Condizione di reset: quando il reset è alto vengono azzerati tutti i registri e i segnali del datapath e viene impostato come stato prossimo lo stato *S0*.

- **S0**: è uno stato che funziona da "sala d'attesa" e rimane in se stesso fintanto che START è a 0. Quando quest'ultimo passa a 1 è possibile iniziare a leggere l'input seriale, passando così allo stato successivo *S1*. In particolare, se arrivo da *S7* in *S0* pongo *i_done* a 0, dato che da specifica deve rimanere alto per un singolo ciclo di clock. È inoltre impostato *head* a 0 in modo che il successivo bit letto sia salvato nel vettore dei bit di selezione del canale di uscita.
- **S1**: viene letto il primo bit sull'ingresso *i_w* e viene salvato nel LSB del vettore dei 2 bit di selezione. Il segnale *head* rimane basso in modo tale da poter aggiungere il secondo bit di selezione allo stesso vettore.
- **S2**: viene letto il secondo bit di selezione e tramite l'operazione di shifting dello shift register il bit precedentemente letto passa nel MSB, mentre quello appena letto nel LSB. A questo punto è completata la lettura dei bit di selezione quindi *head* viene impostato a 1 in modo che il successivo bit di ingresso, se presente, venga salvato in un secondo vettore da 16 bit che si occupa di immagazzinare l'indirizzo di memoria. Dato che potrebbe non esserci un bit di ingresso successivo e quindi avere un indirizzo di memoria fatto di soli zeri, lo stato *S2* ha come possibile stato prossimo lo stato *S4* a cui arriva se START torna basso e quindi l'input valido da acquisire è terminato. Altrimenti, se START rimane alto, lo stato prossimo sarà *S3* in modo tale da poter continuare ad acquisire l'input.
- **S3**: iteriamo su questo stato fintanto che START non torna basso e quindi la sequenza d'ingresso è stata completamente acquisita. Durante la permanenza in *S3* viene sempre impostato *head* a 1 in modo che i bit vengano inseriti sempre nel vettore da 16 bit dell'indirizzo di memoria, che shifta a sinistra di una posizione ogni volta che ne viene letto e salvato uno nuovo. Quindi nel momento in cui START torna basso avrò nel vettore da 16 bit il mio indirizzo che sarà già 0-padded e potrò passare allo stato prossimo *S4*.
- **S4**: mettiamo *rsel_load* a 1 in modo che il registro *rsel* possa immagazzinare il vettore dei bit di selezione. Poniamo inoltre *o_mem_en* a 1 così da abilitare in lettura la memoria esterna, andando quindi a prelevare il dato contenuto all'indirizzo di memoria specificato dal vettore di 16 bit popolato poco prima.
- **S5**: *update_demux* viene impostato ad 1 in modo che il demultiplexer possa ricevere *i_mem_data* per poi inoltrarlo al canale indicato dal selettore. Inoltre, i segnali di load dei 4 registri vengono messi ad 1 in modo da poter salvare al loro interno il dato in arrivo dal demultiplexer, nel caso il loro canale fosse quello selezionato.
- **S6**: il nuovo dato viene immagazzinato nel registro del canale selezionato; quindi viene messo *i_done* a 1 in modo che le uscite possano essere rese visibili. Viene inoltre impostato *to_zero* a 1 per far sì che il vettore *sel_demux* dei 2 bit di selezione e il vettore *addr* dei 16 bit di indirizzo vengano azzerati per essere pronti a una nuova lettura.

Risultati sperimentali

Sintesi

Nella nostra implementazione, il componente descritto è stato sintetizzato correttamente tramite l'utilizzo di 55 flip-flop, ed è stato necessario l'utilizzo di 30 lookup tables.

Di seguito è mostrato il report da console Vivado.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	30	0	134600	0.02
LUT as Logic	30	0	134600	0.02
LUT as Memory	0	0	46200	0.00
Slice Registers	55	0	269200	0.02
Register as Flip Flop	55	0	269200	0.02
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

È possibile, inoltre, osservare l'assenza di latch, ulteriore sintomo di una buona sintesi in quanto conferma il totale e unico utilizzo di dispositivi sincroni come i flip-flop.

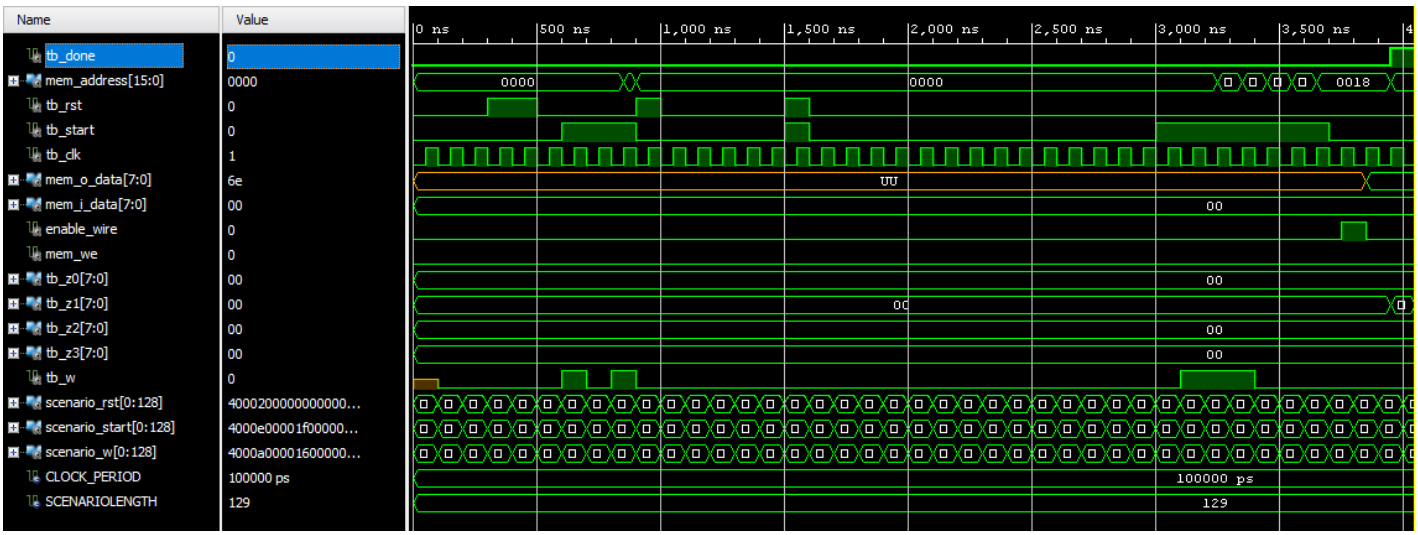
Simulazioni

Per valutare la qualità, la correttezza e l'affidabilità del componente sintetizzato ci siamo serviti di una serie di casi di test con l'intento di "stressare" l'implementazione nei suoi possibili casi limite.

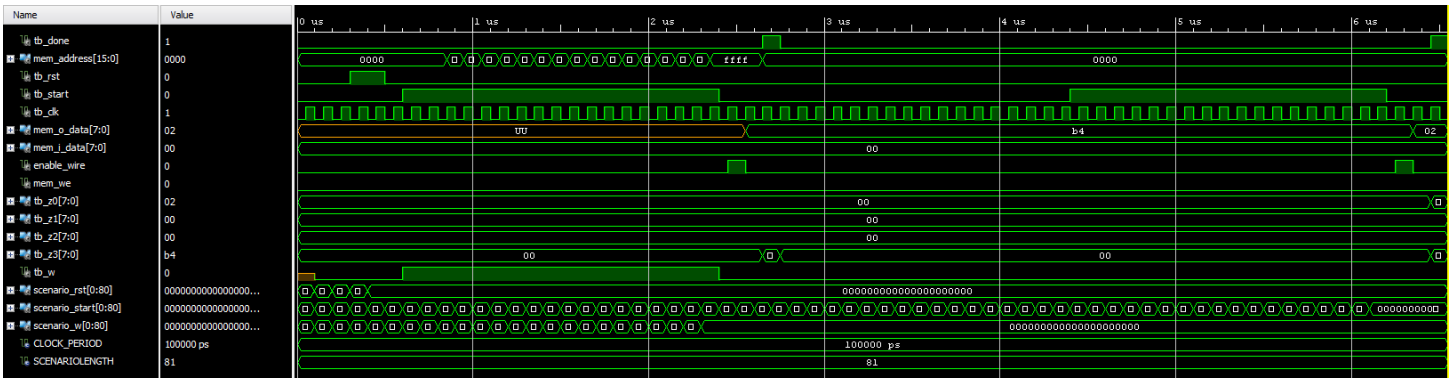
Abbiamo effettuato gli stessi test sia in fase di pre-sintesi che in post-sintesi. Durante la fase di pre-sintesi, le simulazioni sono state necessarie affinché riuscissimo a correggere alcuni errori di tipo implementativo nel nostro codice VHDL. Solo dopo aver risolto questi problemi, siamo passati alla sintesi. Quindi, servendoci dei medesimi test, abbiamo effettuato le simulazioni anche in post-sintesi per assicurarci della correttezza e funzionalità del componente sintetizzato.

Nella pagina seguente troviamo alcuni dei test effettuati.

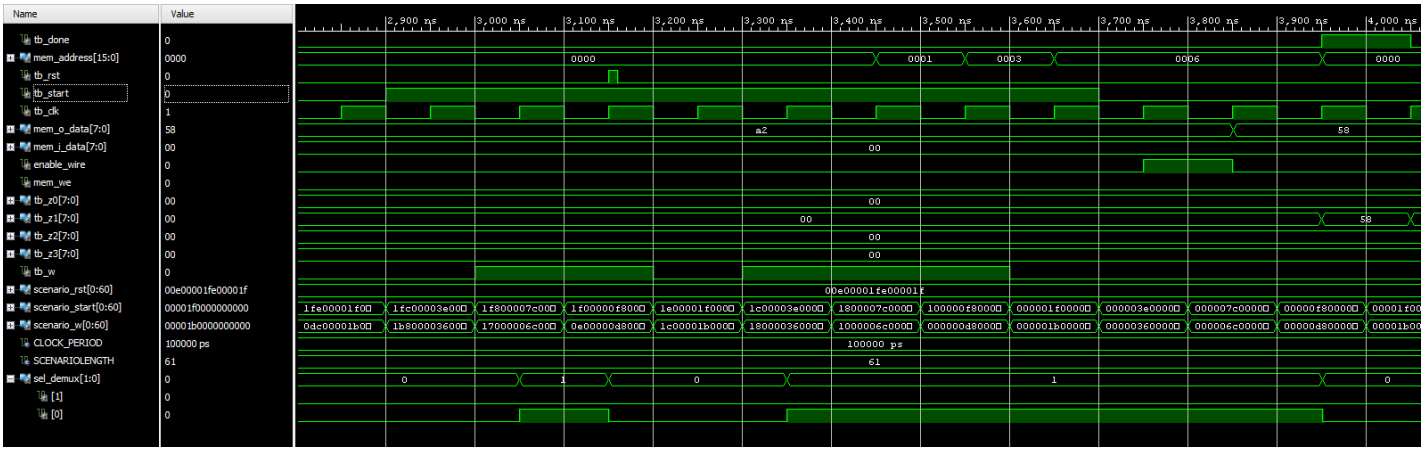
- **<u>Reset multipli</u>**: questo test ci è stato utile per verificare il corretto funzionamento del reset da noi implementato. Ci interessava chiaramente che, anche durante una sequenza di ingresso valida, qualora si fosse verificata una condizione di reset, i vari segnali venissero azzerati correttamente.



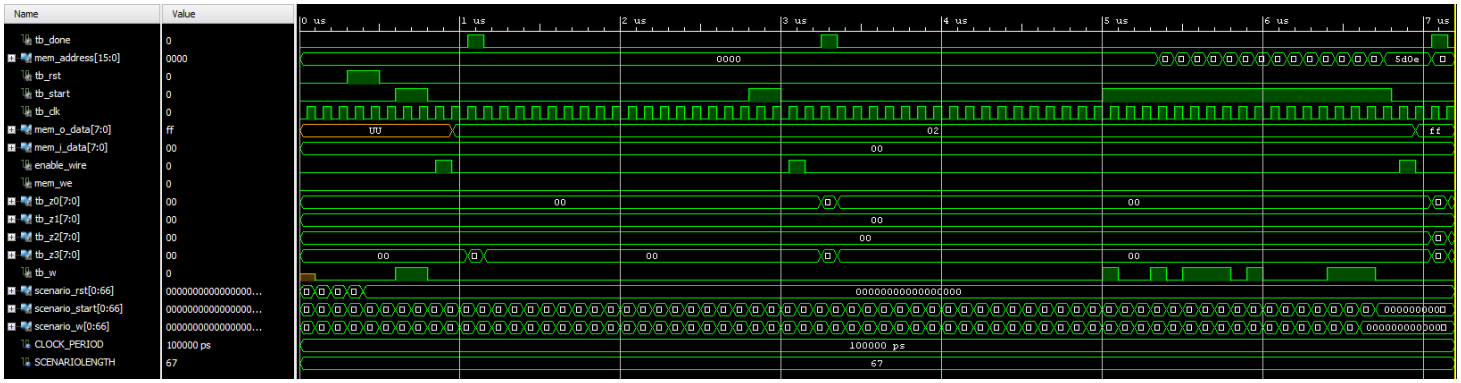
- **<u>Sequenza da diciotto '0'/sequenza da diciotto '1'</u>**: questo test ci ha permesso di verificare dei casi limite in cui venga inserita una sequenza di ingresso <u>intera</u> (2 bit + 16 bit) prima con soli '1' e poi con soli '0'. Il comportamento è quello atteso.



- **<u>Reset asincrono</u>**: questo test invia un segnale di reset (RESET=1) durante una sequenza di ingresso valida (START=1) e, come sperato, azzera correttamente tutti i segnali. In particolare, durante questa fase di lettura, il reset va a 1 durante la lettura del secondo bit di intestazione che, da nostra implementazione, viene inserito nel vettore *sel_demux* (ultimo segnale in foto). Com'è possibile notare, appena dopo il reset, *sel_demux* torna a 0 dimostrando così il corretto funzionamento del componente sintetizzato.



- **<u>Due sequenze con sola intestazione/una sequenza da diciotto bit</u>**: in questo caso di test abbiamo verificato altri casi limite di sequenze in ingresso. In particolare, prima verifichiamo l'input "11" che corrisponderà all'uscita sul canale Z3; successivamente con l'input "00" verifichiamo che venga propagato il dato sul canale Z0. Dopo le due sequenze da 2 bit viene letta una terza sequenza, questa volta da 18 bit. Anche in questo caso il risultato è quello atteso.



Conclusioni

I risultati ottenuti dalla progettazione e implementazione del componente sono stati soddisfacenti. Riuscendo correttamente a superare in pre-sintesi e in post-sintesi tutti i test a cui esso è stato sottoposto, il nostro modello si è rivelato convincente e funzionante.

Le scelte implementative sono state ben chiare fin dall'inizio, dalla scelta dello shift register per la lettura dell'ingresso seriale, fino al demultiplexer impiegato per la corretta selezione del canale di uscita e la successiva propagazione del dato.

Infine, con il supporto di alcuni segnali da noi inseriti e una macchina a stati finiti semplice e funzionale, siamo riusciti a trovare una quadra che permettesse correttamente di gestire tutte le varie fasi da compiere per la lettura, elaborazione e propagazione dei vari dati.