Relazione Progetto Reti Logiche

Allegra Chiavacci (10733258), Alessandro Cavallo (10734387)

A.A. 2022-23

1 Introduzione

Il progetto assegnatoci consiste nell'implementazione di un componente hardware che si interfacci con una memoria. Attraverso un input seriale, il componente riceve indicazioni riguardo un indirizzo di memoria, il cui contenuto deve essere scritto su uno dei quattro canali output disponibili (tutti i bit vengono forniti in parallelo). I primi due bit di input identificano il canale d'uscita. Per la stesura del progetto si è ambito a realizzare un'architettura semplice e funzionale, che rispettasse le specifiche proposte in tempi inferiori a quelli richiesti. L'obiettivo è stato quello di combinare vari componenti architetturali studiati durante il corso di Reti Logiche, quali un contatore, un demultiplexer, dei registri e una FSM.

1.1 Funzionamento in sintesi

Di seguito viene descritto sinteticamente il funzionamento del componente, che verrà poi presentato in dettaglio nella sezione 2 (Architettura) tramite l'opportuna FSM:

1. RESET ed attesa del primo segnale di START;
2. Nel caso di segnale di RESET alto, vengono inizializzati a 0 i quattro buffer corrisponenti a ciascun canale di uscita (8 bit);
3. Reset del contatore predisposto al conteggio dei bit in ingresso, il segnale di ENABLE viene posto alto e quello di WRITE ENABLE basso.
4. Quando il segnale di START viene posto alto, avviene la lettura dei primi 2 bit di input e il loro salvataggio in un apposito registro, che permette di identificare il corretto canale di uscita;
5. Si procede alla lettura degli N bit ($0 \leq N \leq 16$) e al loro salvataggio in un registro di supporto, fintanto che il segnale di START rimane alto;
6. Viene generato l'indirizzo di 16 bit, con un opportuno shifting sulla base del numero N di bit ricevuti;
7. Una volta applicata l'opportuna estensione dell'indirizzo, si attende la ricezione del contenuto della locazione di memoria, il quale verrà poi salvato in un registro (buffer) associato al canale di uscita corretto.
8. Il segnale di DONE viene posto alto e su ciascuno dei canali di uscita si leggono parallelamente gli 8 bit contenuti nei buffer corrispondenti. Il canale associato al messaggio in questione cambierà valore, mentre gli altri mostreranno l'ultimo valore associato a quel canale di uscita derivante dai messaggi precedenti.
9. Il segnale di DONE rimane alto per un ciclo di clock, dopo di che torna basso, in attesa di un segnale di RESET o di START, tornando quindi alla fase iniziale.

2 Architettura

2.1 Macchina a Stati

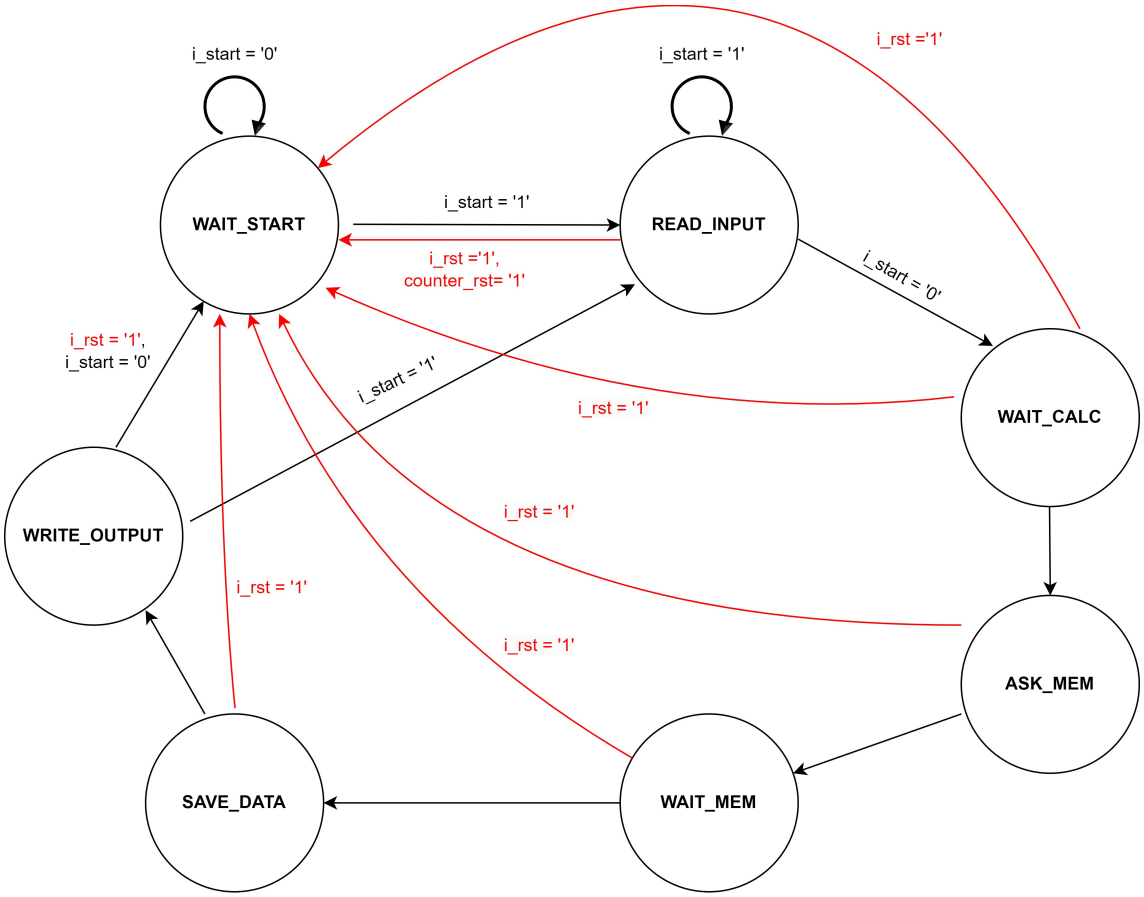

Figura 1: Stati della FSM

Con la FSM si vuole rappresentare il comportamento del componente in un modo schematico, descrivendo la successione degli stati e i segnali utilizzati:

- **WAIT_START**: Stato iniziale e finale della FSM. Si attende che il segnale START vada alto;
- **READ_INPUT**: Si permane in questo stato finchè START rimane alto. Vengono salvati i bit di ingresso: i primi 2 nel registro indicante il numero del canale di uscita ed i successivi N nel registro di supporto, prima del calcolo dell'indirizzo effettivo. Un caso che viene controllato, ma secondo la specifica del progetto, non dovrebbe verificarsi, è quello di un overflow del contatore (5 bit incrementer), che riporterebbe la macchina nello stato di WAIT_START;
- **WAIT_CALC**: Stato intermedio che permette di gestire eventuali ritardi relativi al contatore;
- **ASK_MEM**: Generazione dell'indirizzo di memoria con opportuno shifting;
- **WAIT_MEM**: Stato intermedio che permette il trasferimento del dato dalla memoria;
- **SAVE_DATA**: Salvataggio del dato nell'opportuno registro (buffer), sulla base del numero salvato nel registro indicante il numero del canale di uscita;
- **WRITE_OUTPUT**: Il segnale di DONE viene posto alto e sui canali d'uscita viene scritto il valore del buffer corrispondente.

Il segnale di DONE torna basso dopo un ciclo di clock e la FSM può evolvere in due modi diversi:

1. Dallo stato WRITE_OUTPUT passa allo stato READ_INPUT, nel caso in cui il segnale START venga posto alto;
2. Dallo stato WRITE_OUTPUT passa allo stato WAIT_START, in caso di RESET oppure in attesa di un nuovo segnale di START.

In VHDL abbiamo utilizzato due process per descrivere l'avanzamento della FSM:

- Un process per descrivere la parte sequenziale della FSM, ovvero per determinare lo stato in cui evolve la FSM in funzione del current_state e di alcuni segnali in ingresso, quali i_start e i_rst, oppure di segnali generati da componenti del circuito come counter_ovf. L'aggiornamento degli stati avviene sul fronte di salita del clock oppure in caso di segnale di RESET alto.
- Un process per descrivere la parte combinatoria della FSM, ovvero la gestione di registri e segnali interni al componente, quali counter_rst, generate_addr o save_data_sign, sulla base dello stato corrente della FSM. L'utilità dei segnali appena citati verrà esplicitata nelle sezioni successive.

2.2 Gestione dell'input

Abbiamo deciso di gestire i 2+N bit di input con il supporto di un contatore (5 bit incrementer), un demultiplexer a 18 uscite e due registri, rispettivamente di 2 e 16 bit.

Il contatore è stato implementato come un componente esterno puramente combinatorio e la sua architettura è stata sviluppata seguendo la logica del sommatore a full adder sequenziali, come studiato durante il corso di Reti Logiche. L'incremento del contatore viene gestito all'interno di un process sequenziale dedicato, sensibile al segnale di clock ed al segnale i_start. L'obiettivo è quello di determinare il numero di bit che vengono passati in input, dato che verrà utilizzato successivamente per la generazione dell'indirizzo. Il segnale di counter_ovf viene utilizzato per un'eventuale gestione di una situazione di overflow, anche se la specifica del progetto la esclude, in quanto i bit in ingresso saranno al massimo 18 (2+16) ed il componente è in grado di contare fino a 31. Il contatore viene resettato a "00000" in caso di RESET del sistema oppure successivamente alla ricezione del contenuto della locazione di memoria.

Il demultiplexer viene utilizzato per fare in modo che ogni bit in ingresso venga salvato nella corretta posizione del registro corrispondente:

- i primi 2 bit vengono salvati dal più al meno significativo nel registro regnumoutput, indicante il numero del canale di uscita scelto.
- i successivi N bit vengono salvati all'interno di un registro di 16 bit regaddress dal più al meno significativo, non curandosi per il momento di un'eventuale necessità di estendere l'indirizzo con degli 0.

2.3 Estensione dell'indirizzo

Quando il segnale di start viene posto basso, si procede con l'operazione di shifting e la conseguente generazione dell'indirizzo, ponendo il segnale generate_address = '1'. L'estensione dell'indirizzo viene gestita all'interno di un process mediante un loop, che tiene conto del valore salvato nel contatore.

2.4 Gestione del dato

Il contenuto della locazione di memoria all'indirizzo o_mem_address, generato nel modo spiegato sopra, viene passato dalla memoria al componente attraverso il segnale i_mem_data. Abbiamo racchiuso questa operazione all'interno di un altro process ed abbiamo utilizzato un demultiplexer con le seguenti caratteristiche:

- segnale d'ingresso: i_mem_data;
- segnale di controllo: contenuto del registro regnumoutput, citato nella sezione 2.2, indicante il numero del canale d'uscita scelto;

- 4 possibili uscite, corrispondenti a 4 registri da 8 bit ciascuno: registri chiamati regbuffer_00, regbuffer_01, regbuffer_02, regbuffer_03, con il numero che identifica il canale d'uscita.

In questo modo viene sovrascritto il solo regbuffer corrispondente all'uscita scelta, mentre gli altri mantengono il valore ottenuto dai messaggi precedenti. I quattro regbuffer vengono inizializzati a "00000000" in seguito ad un segnale di RESET. A questo punto il segnale o_done viene posto alto e sui canali d'uscita viene prodotto il contenuto dei regbuffer corrispondenti (e.g. sul canale o_z0 viene scritto il contenuto di regbuffer_00). Quando il segnale o_done è basso, ogni canale d'uscita è a zero.

2.5 Rappresentazione dell'Architettura

Di seguito la rappresentazione dell'architettura compresa dei segnali di controllo dei vari registri.

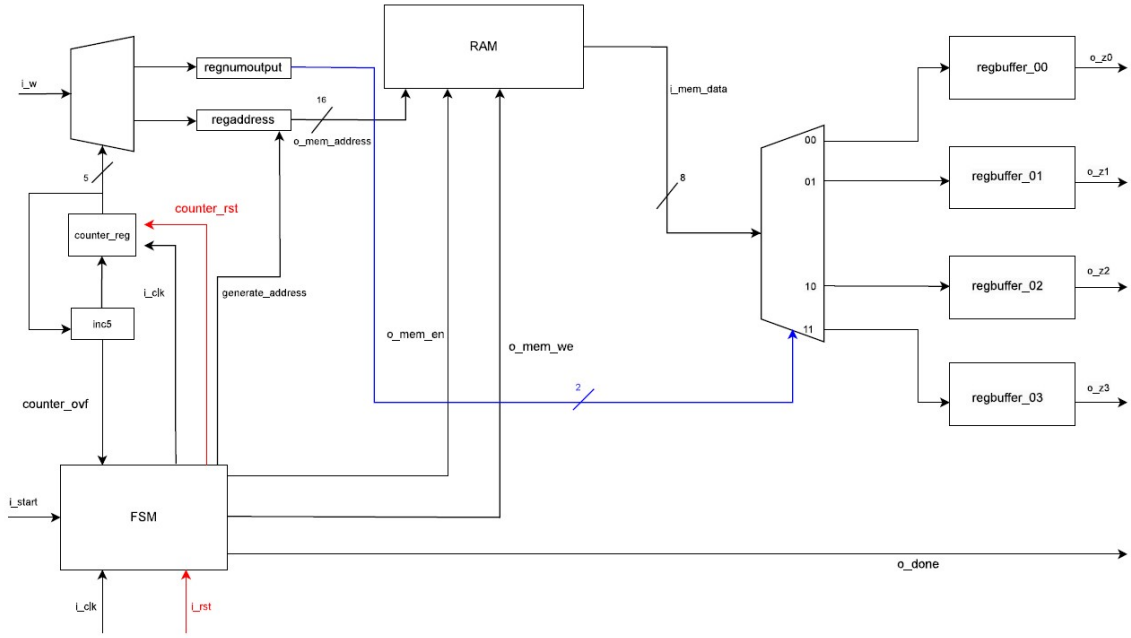

Figura 2: Rappresentazione Architettura

3 Risultati sperimentali

3.1 Sintesi

3.1.1 Report Utilization

Attraverso il comando report_utilization su Vivado, è stato possibile analizzare l'area occupata dal componente e il tipo di risorse utilizzate.

Risorse	Utilizzo	Utilizzo %
LUTs	91	0.07
Flip Flops	78	0.03

Tabella 1: Report Utilization

Non è stata rilevata la presenza di Latch in post sintesi ed è stato gestito il ritardo dei vari componenti.

3.1.2 Timing Report

Tramite il Timing Report si può analizzare la velocità del componente durante un ciclo di clock. Tenendo conto di un constraint di 100 ns, di seguito i risultati:

Slack	Data Path Delay	Requirement
96.948 ns	2.670 ns	100.000 ns

Tabella 2: Timing Report in Post Sintesi

Ritardo	logic	logic %	route	route %
2.670 ns	0.901 ns	33.745%	1.769 ns	66.255%

Tabella 3: Analisi del Data Path Delay

Dopo questa analisi siamo in grado di capire quanto siamo vicini/lontani dal ciclo di clock previsto dal constraint. Il Data Path Delay invece indica l'intervallo di tempo che intercorre tra l'arrivo del fronte di salita del clock e la commutazione delle porte logiche del componente. Nel tempo definito Slack tutti i segnali sono in attesa. Per calcolare la frequenza del componente hardware possiamo utilizzare la formula $f_{max} = 1/(T_{ck} - Slack)$, tenendo conto di un periodo di clock pari a 100 ns e lo Slack pari al valore riportato in tabella (96.948ns). Ne risulta una $f_{max} \simeq 327.65MHz$

3.2 Simulazioni

Una volta progettato il componente, lo abbiamo sottoposto ad alcuni testbench, tra cui uno fornito. Segue l'analisi del Test Bench fornito dal docente e due dei Test Bench creati da noi, con l'obiettivo di testare il corretto funzionamento sia in pre-sintesi che in post-sintesi del componente. Proponiamo una serie di casi particolari, che avrebbero potuto generare comportamenti inaspettati:

- Input di 2 + N bit con N=0;
- Input di 2 + N bit con N=16;
- Reset dopo ricezione di un input, ma prima che DONE venga posto alto
- Sovrascrittura di un canale di uscita, (stesso canale di uscita del messaggio appena precedente)

Tutti i Test Bench che abbiamo eseguito, sono stati superati sia in Behavioral sia in post-sintesi.

3.2.1 TestBench 0 (fornito)

Nel Test Bench in questione viene controllato che i segnali o_mem_en = '1' e o_mem_we = '0'. Il primo segnale alto permette l'accesso in memoria, mentre il secondo viene posto basso, dato che è sempre necessario leggere e mai scrivere in memoria. Il segnale di reset viene posto alto prima del primo START e viene passato un messaggio che richiede la scrittura del contenuto nella locazione di memoria all'indirizzo '1' (esteso su 16 bit con 0) sul canale d'uscita 2.

Il secondo messaggio prevede la scrittura sul canale d'uscita 3 del contenuto dell'indirizzo "11000" (esteso su 16 bit con 0). Nel Test Bench viene controllato che venga scritto sul canale d'uscita il corretto valore contenuto nella cella di memoria indicata.

Segue un diagramma esplicativo:

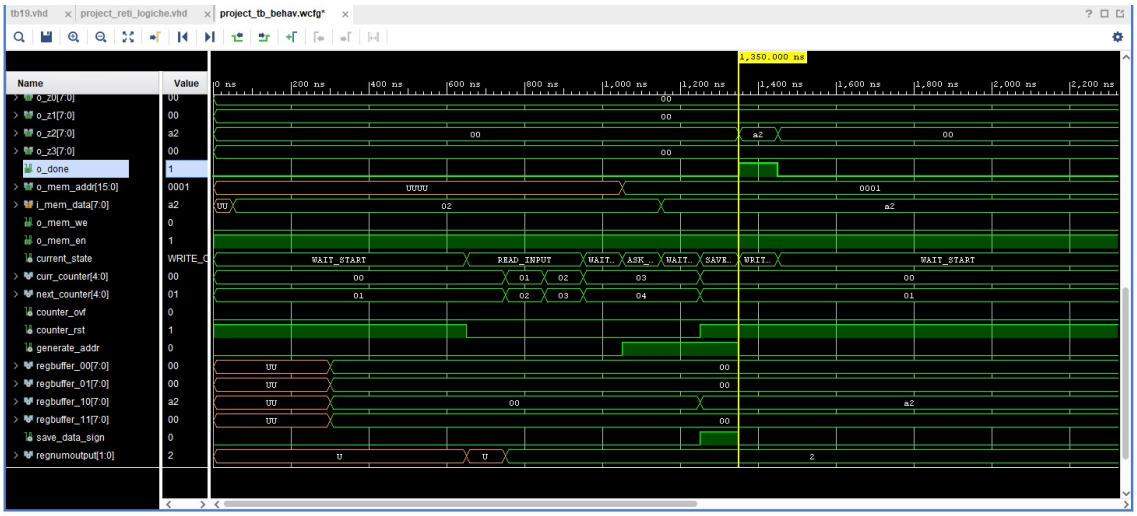

Figura 3: Test Bench 0

3.2.2 TestBench 1 (non fornito)

Abbiamo scritto questo Test Bench per testare tre dei casi elencati nella lista iniziale:

- Input di 2 + N bit con N=0;
- Input di 2 + N bit con N=16;
- Sovrascrittura di un canale di uscita (stesso canale in due messaggi che si susseguono).

Il segnale di reset viene posto alto prima del primo START e viene passato un messaggio di 2 + N bit con N=0 (input = "10"). Questo implica di dover scrivere in uscita il contenuto dell'indirizzo "0000000000000000" sul canale 2.

Il secondo messaggio prevede la scrittura sullo stesso canale d'uscita del contenuto dell'indirizzo "1111111111111111".

Nel Test Bench viene controllato che venga scritto sul canale d'uscita il corretto valore contenuto nella cella di memoria indicata, sovrascrivendo correttamente il regbuffer_02.

Il contenuto degli indirizzi di memoria è stato inventato ed unito ai dati del Test Bench 0 fornito dal docente.

Segue un diagramma esplicativo:

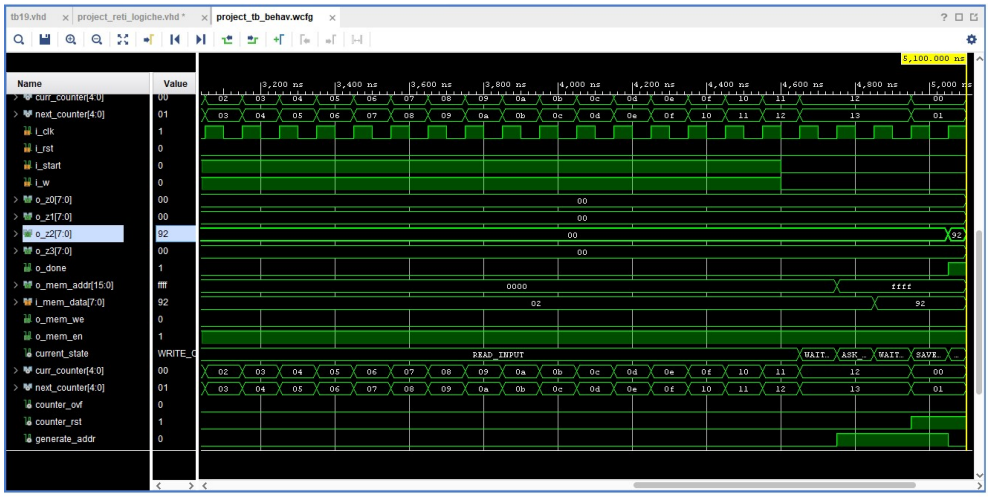

Figura 4: Test Bench 1

3.2.3 TestBench 2 (non fornito)

Questo Test Bench ha il compito di testare il reset in due momenti diversi della computazione:

- un primo reset subito dopo il passaggio del segnale di start da 1 a 0 (prima di o_done=1)
- un secondo reset dopo o_done=1;

Il segnale di reset viene posto alto prima del primo START e viene passato un messaggio che richiede la scrittura del contenuto dell'indirizzo "10" (esteso su 16 bit con 0) sul canale di uscita 2.

Un secondo messaggio richiede la scrittura del contenuto dell'indirizzo "101010010" sul canale di uscita 1. Appena il segnale i_start è posto basso, viene fatto il reset del componente.

Dopo il terzo messaggio, richiedente la scrittura del contenuto dell'indirizzo "10001011011" sul canale di uscita 1, è stato possibile testare il corretto funzionamento del reset, controllando che sul canale 2 venisse scritto "00000000" nel momento di o_done = '0'.

Un altro reset è stato fatto in seguito al quarto messaggio, dopo che il segnale di o_done = '1'.

Anche in questo caso abbiamo aggiunto il contenuto di alcune locazioni di memoria all'interno della specifica della memoria nel Test Bench.

Segue un diagramma esplicativo:

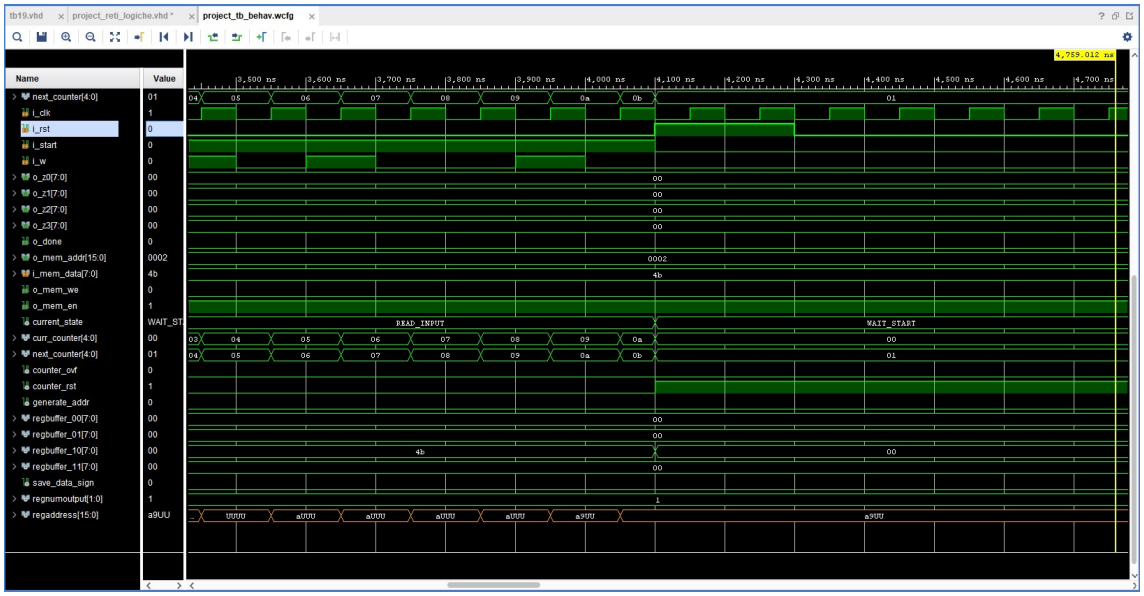

Figura 5: Test Bench 2

4 Conclusioni

Questo progetto ci ha permesso di imparare ad utilizzare il linguaggio VHDL e il tool Vivado. E' stata un'occasione per mettere in pratica quanto studiato durante il primo semestre nel corso di Reti Logiche. E' stato necessario, non solo pensare a sviluppare un programma funzionante dal punto di vista applicativo e che rispettasse la specifica, ma anche soffermarsi ad analizzare i componenti hardware necessari per la sua realizzazione. Per far questo, abbiamo iniziato realizzando graficamente il circuito e la relativa FSM, per poi procedere alla scrittura del codice. Con questa relazione vogliamo presentare un componente funzionante sia in fase Behavioral sia in Post-Sintesi senza la creazione di latch.