POLITECNICO
MILANO 1863

Prova Finale
Progetto di Reti Logiche

Prof. Gianluca Palermo - Anno 2022/23

Nome	Codice Persona	Matricola
Melanie Tonarelli	10787497	960072
Niccolò Sobrero	10718657	955080

Indice

1. Introduzione

1.1 Scopo del progetto

Lo scopo del progetto è di implementare un componente hardware, descritto in VDHL, in grado di estrarre, in momenti predefiniti, indirizzi di memoria da un flusso seriale di bit ed indirizzarne i contenuti su quattro canali di uscita.
I messaggi letti da memoria vengono salvati, eventualmente sovrascrivendone altri, e mostrati ogni qualvolta che viene finita una lettura da memoria.

1.2 Specifiche generali

Dopo l'istante iniziale, corrispondente al reset del sistema, si legge dall'ingresso primario seriale `W` una sequenza di bit in corrispondenza del segnale `START` alto (`START=1`) e sul fronte di salita del clock.
La sequenza di bit valida è composta da almeno 2 bit fino ad un massimo di 18 bit e viene organizzata nel seguente modo:

- i primi due bit della sequenza rappresentano uno dei 4 canali d'uscita (identificati con `zX`, dove `X` rappresenta un numero compreso tra 0 e 3). In particolare: `00` identifica `z0`, `01` identifica `z1`, `10` identifica `z2` e `11` identifica `z3`.
- il resto della sequenza, di lunghezza tra 0 e 16 bit, rappresenta l'indirizzo di memoria a cui leggere il messaggio composto da 8 bit. Il primo bit letto è quello più significativo, mentre l'ultimo è quello meno significativo. Se il numero di bit letti è inferiore a 16, l'indirizzo viene esteso aggiungendo 0 sui bit più significativi. Ad esempio:

  (N=9) `10111001` —› `0000000010111001`

  (N=0) `0` —› `0000000000000000`

  (N=16) `1010010011111001` —› `1010010011111001`

Il messaggio letto all'indirizzo di memoria identificato viene indirizzato sul canale d'uscita indicato dai primi due bit validi, di cui abbiamo discusso poc'anzi.
Il segnale di `DONE` viene alzato per un solo ciclo di clock contemporaneamente a quando il messaggio viene scritto sul canale; in particolare, quando `DONE=1`, le uscite `z0`, `z1`, `z2` e `z3` mostrano gli stessi messaggi letti in precedenza, ad eccezione del canale aggiornato nelle modalità appena descritte. Invece `DONE=0` implica che sulle uscite `z0`, `z1`, `z2` e `z3` vengano mostrati tutti zeri.
Inoltre, un eventuale reset successivo all'istante iniziale reinizializza il modulo e pertanto vengono eliminati i messaggi letti fino a quell'istante.

Viene garantito che:

- il segnale `START` è alto per almeno 2 cicli di clock e per massimo 18 cicli di clock;
- il segnale di `START` è basso fino a che il segnale `DONE` non è tornato a 0.

1.3 Interfaccia del componente

Il componente da descrivere ha un'interfaccia così definita:

```
entity project_reti_logiche is
port (
    i_clk          : in std_logic;
    i_start        : in std_logic;
    i_rst          : in std_logic;
    i_data         : in std_logic_vector(7 downto 0);
    o_address      : out std_logic_vector(15 downto 0);
    o_done         : out std_logic;
    o_en           : out std_logic;
    o_we           : out std_logic;
    o_data         : out std_logic_vector(7 downto 0)
);
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 pronta per ricevere il primo segnale di `START`;
- `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` e `o_z3` sono i quattro canali d'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 che in scrittura);
- `o_mem_we` è il segnale di `WRITE ENABLE` da dover mandare alla memoria per poterci scrivere. Per leggere da memoria esso deve essere 0.

1.4 Dati e descrizione della memoria

Il componente hardware implementato si interfaccia con una memoria RAM di tipo 'single-port block RAM write-first mode', ossia una memoria sincrona ad accesso sequenziale che prioritizza la scrittura rispetto alla lettura.
L'indirizzamento è al byte ed avviene grazie all'utilizzo del segnale `o_mem_address` formato da 16 bit, mentre i dati contenuti in memoria hanno ciascuno una dimensione di 8 bit.

2. Design

2.1 Stati della FSM

La macchina a stati finiti implementata è costituita da 6 stati, come mostrato nella figura sottostante (figura 1).
Qui di seguito è fornita una breve descrizione per ciascuno stato.

2.1.1 IDLE

Stato iniziale in cui si attende il segnale `i_start`. Ogni qualvolta che viene alzato il segnale `i_rst`, si torna in questo stato.
In particolare, alla prima occorrenza di `i_start` alto, dato che ci troviamo in questo stato, viene salvato il primo bit (ossia quello più significativo) dell'indirizzo del canale di uscita su un apposito registro.

2.1.2 GET_REG

Stato in cui viene memorizzato il secondo bit (ossia quello meno significativo) dell'indirizzo del canale di uscita su un registro dedicato.

2.1.3 GET_MEM

Stato in cui viene salvato su un registro a scorrimento (di cui discutiamo nel paragrafo 2.2) l'indirizzo di memoria a cui andare a leggere.
Si rimane in questo stato finché `i_start` resta alto, in quanto non è possibile acquisire tutti i bit dell'indirizzo di memoria in simultanea, essendo `i_start` un ingresso seriale.

2.1.4 WAIT_RAM

Stato in cui si attende la risposta della memoria in seguito alla richiesta di un dato.

2.1.5 WRITE_REG

Stato in cui viene memorizzato il dato letto da memoria nel registro associato al canale d'uscita indicato precedentemente.

2.1.6 DONE_STATE

Stato di terminazione in cui si settano i segnali d'uscita in modo che al ciclo di clock successivo (ossia una volta che si è ritornati allo stato `IDLE`) vengano mostrati il segnale di `o_done` alto e i contenuti dei registri sui rispettivi canali d'uscita.
Questo è uno stato di ritardo aggiunto per sincronizzare la lettura da memoria e la scrittura delle uscite.

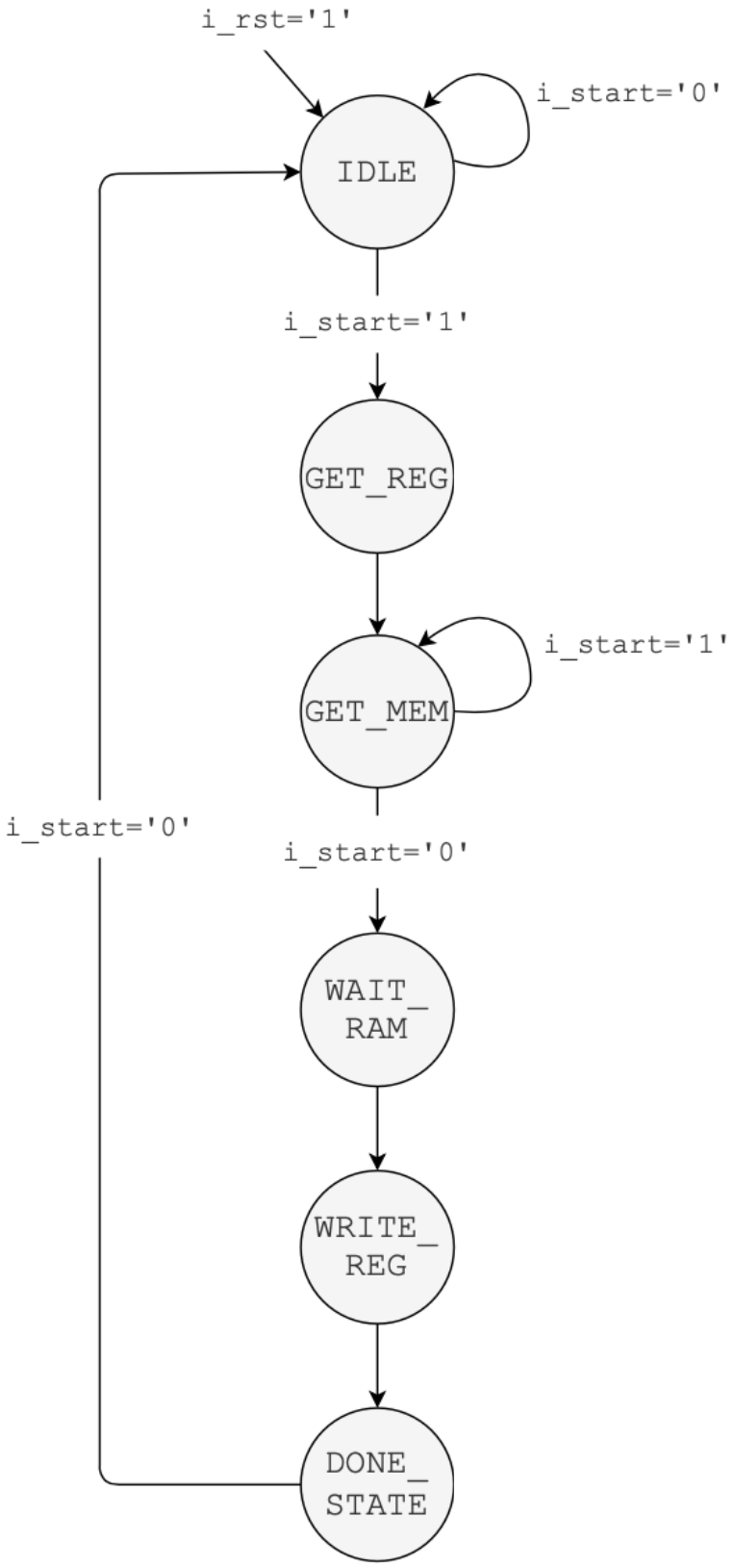

Figura 1: FSM del modulo

2.2 Scelte progettuali

Il modulo è costituito da 3 processi:

1. Il primo rappresenta la parte sequenziale della macchina e serve per gestire l'RT (register transfer) e quindi come vengono manipolati i registri.
2. Il secondo rappresenta l'evoluzione del sistema e determina lo stato prossimo della FSM analizzando i segnali in ingresso e lo stato corrente.
3. Il terzo determina le uscite prossime della FSM, ossia i valori delle uscite che verranno mostrato al ciclo di clock successivo rispetto allo stato in cui ci si trova.

L'indirizzo del canale d'uscita, identificato dai primi due bit di `i_w` in corrispondenza di `i_start` alto, sono salvati in due registri distinti `ch1` e `ch0`, che mantengono rispettivamente il bit più significativo e quello meno significativo.

Per acquisire e memorizzare l'indirizzo di memoria dal segnale seriale `i_w`, utilizziamo un registro a scorrimento di tipo SIPO (serial input-parallel output) a 16 bit inizializzati ogni volta tutti a `0`, in modo che l'estensione dell'indirizzo di memoria con bit pari a `0` sulla parte più significativa sia automatica.
In questo modo, ogni qualvolta che viene letto un bit da `i_w`, esso viene collocato nella posizione meno significativa dell'indirizzo facendo scorrere gli altri bit presenti a sinistra di una posizione. Tale componente viene implementato sfruttando il segnale d'appoggio `mem_address` e l'operazione di concatenazione di stringhe.

Vengono poi utilizzati altri quattro registri, ossia i segnali `reg0`, `reg1`, `reg2` e `reg3` (che vengono poi tradotti in registri dal sintetizzatore), per memorizzare i messaggi letti da memoria e mostrare il loro contenuto nel giusto istante nei segnali d'uscita `o_z0`, `o_z1`, `o_z2` e `o_z3`.

Tra i segnali d'uscita effettivi del modulo e le uscite della FSM implementata abbiamo inserito un banco di registri in modo da isolare le uscite effettive del modulo dal resto del circuito ed evitare il fenomeno di glitch (ossia un picco breve ed improvviso non periodico) del segnale `o_done` e sincronizzare tutte le uscite senza ritardi di propagazione.

3. Risultati sperimentali

3.1 Sintesi

Il dispositivo progettato è perfettamente sintetizzabile e la sua sintesi schematica è mostrata nella figura sottostante (figura 2).

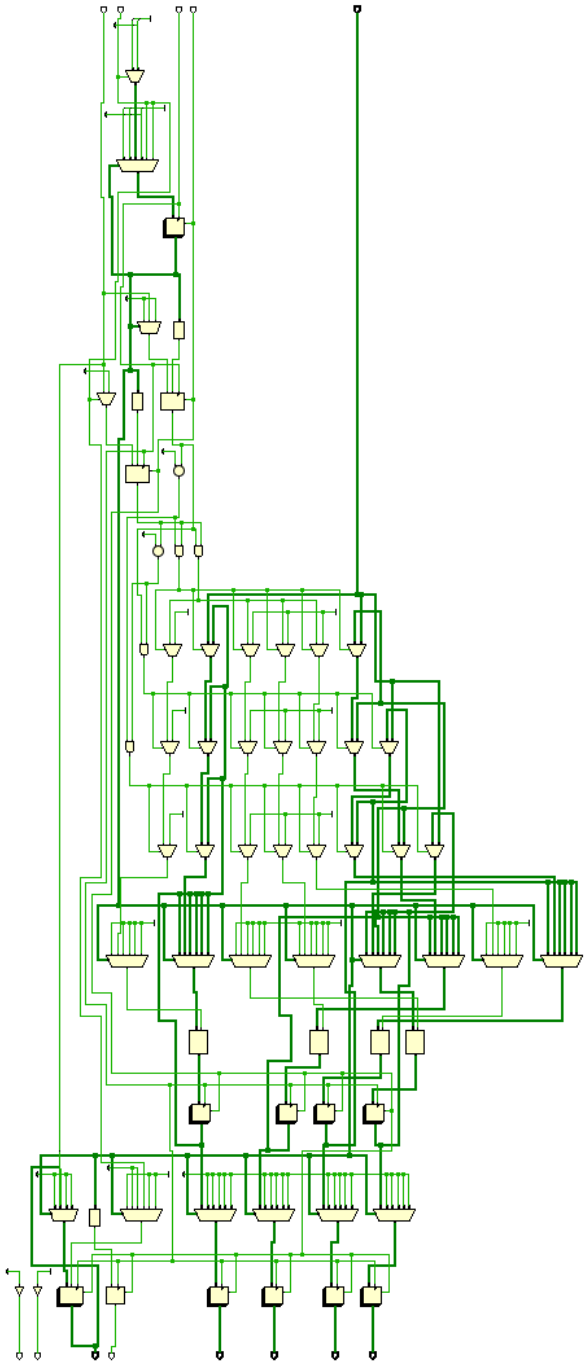

Figura 2: Schematico di sintesi del modulo

Analizziamo ora i reports di sintesi.

- Dall'analisi del `report_utilization` (comando fornito da Vivado), vengono fornite le informazioni osservabili nella tabella sotto (figura 3).

Resource	Utilization	Available	Utilization %
LUT	88	134600	0.07
FF	86	269200	0.03
IO	63	285	22.11

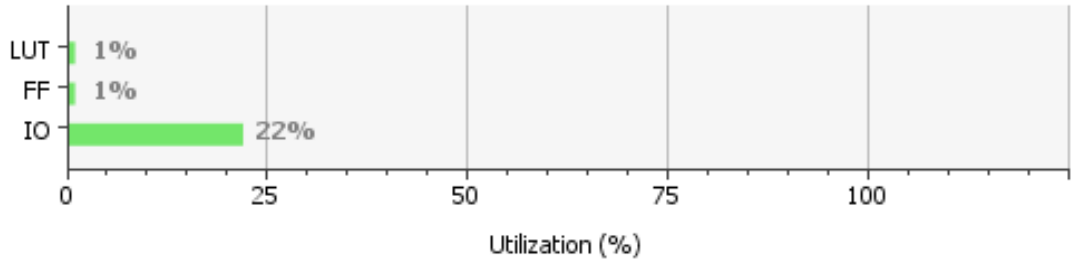

Figura 3: Utilization Report

In particolare, notiamo che sono stati utilizzati ben 86 Flip-Flop e 88 Look-Up Table. Inoltre, vi è un basso grado di occupazione della FPGA rispetto alla sua disponibilità.

- Un dato importante, che riguarda il tempo di esecuzione, è il Worst Negative Slack che ci dice quanto tempo rimane al completamento del ciclo di clock nel peggiore dei paths. Con un ciclo di clock di 100ns abbiamo i seguenti risultati (figura 4):

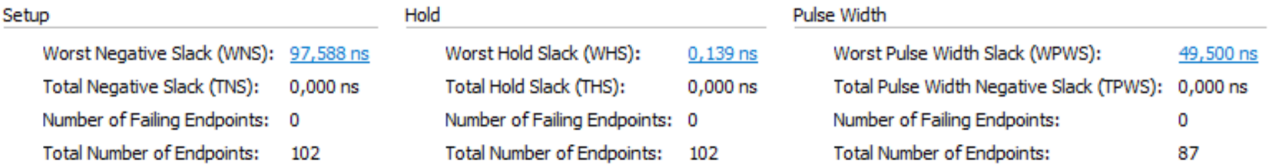

Figura 4: Design Timing Summary

Il comando `report_timing_summary` mostra che il Worst Negative Slack (WNS) e il Worst Hold Slack (WHS) sono maggiori di zero, quindi il vincolo $T_{clock} \leq 100ns$ viene rispettato correttamente.

3.2 Simulazioni

Di seguito è fornita una breve descrizione dei test utilizzati e per quelli più significativi viene anche mostrato il corretto funzionamento grazie allo screenshot dell'andamento dei segnali durante la simulazione. Si noti che per alcuni screenshot è stato fatto uno zoom sul tratto di simulazione su cui si porge l'attenzione, per questo viene omessa la prima porzione di waveform in cui il segnale di `i_rst` è alto (ossia l'inizializzazione del modulo).

- **Sequenza Minima**: si testa il caso in cui il segnale di `i_start` è alto per soli due cicli di clock consecutivi. In particolare, la simulazione in figura 5 mostra che, in corrispondenza di i_start alto, `i_w` vale `00` e pertanto viene scritto il messaggio contenuto all'indirizzo di memoria `0…0` sul segnale d'uscita `o_z0`.

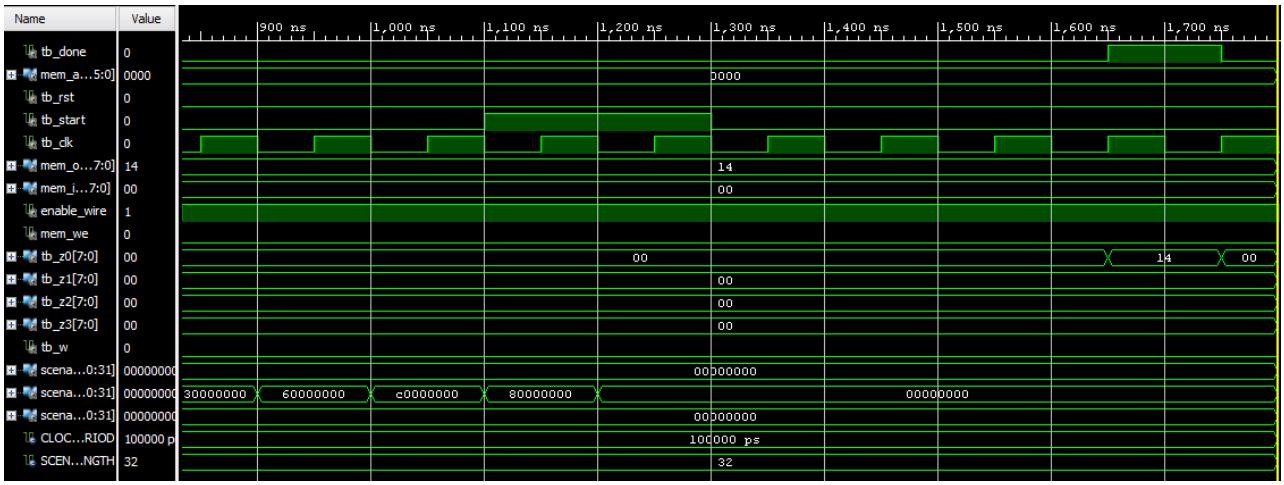

Figura 5: Waveform del test con `i_start` alto per due soli cicli di clock consecutivi

- **Sequenza Massima**: si testa il caso in cui il segnale di `i_start` è alto per 18 cicli di clock consecutivi. Mostriamo il seguente esempio in figura 6: i bit validi di `i_w` sono tutti pari ad `1`, quindi verrà mostrato il contenuto dell'indirizzo di memoria `1…1` sul canale d'uscita `o_z3`.

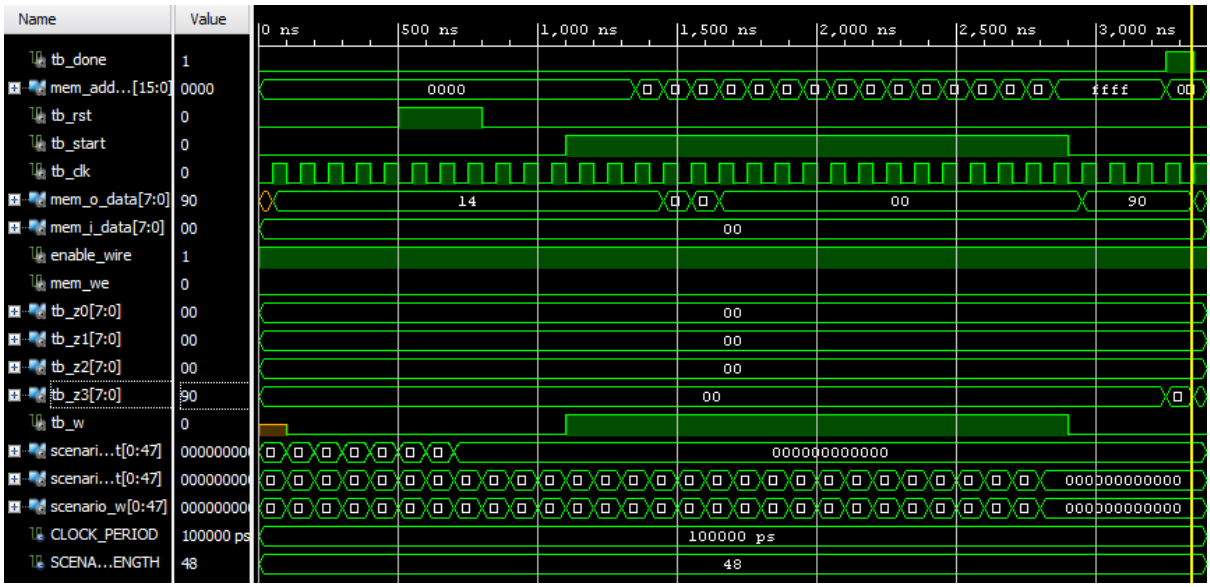

Figura 6: Waveform del test con `i_start` alto per 18 cicli di clock consecutivi

Abbiamo inoltre testato anche il caso in cui i 16 bit meno significativi tra quelli validi di `i_w` (ossia i bit che vanno a formare l'indirizzo di memoria a cui leggere il messaggio poi da mostrare in uscita) siano randomici, ossia né tutti pari a `0` né tutti pari a `1`.

- **Reset Asincrono**: il test verifica che il trigger asincrono del segnale di reset non comprometta la computazione e che questa ricominci facendo tornare la macchina allo stato iniziale `IDLE`.

  Mostriamo i risultati nei seguenti due casi
  - `i_reset` diventa alto mentre `i_start` è basso (figura 7.1):

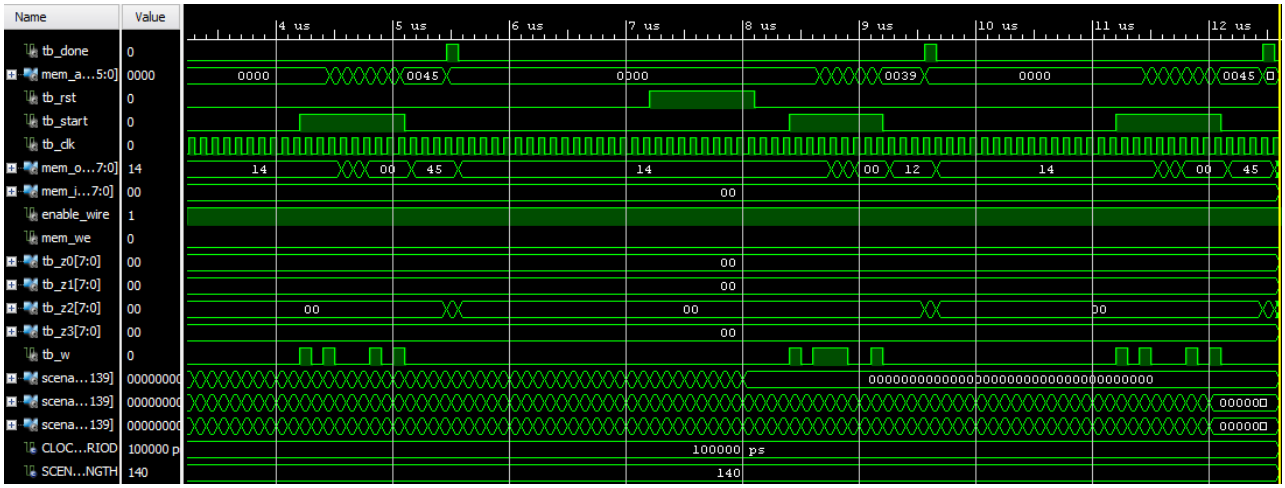

Figura 7.1: Waveform del Reset Asincrono

  - `i_reset` diventa alto mentre anche `i_start` è alto (figura 7.2): in tal caso si vuole verificare che i dati letti dopo `i_start=1` e prima di `i_rst=1` non vengano memorizzati per poi essere riportati sui canali d'uscita.

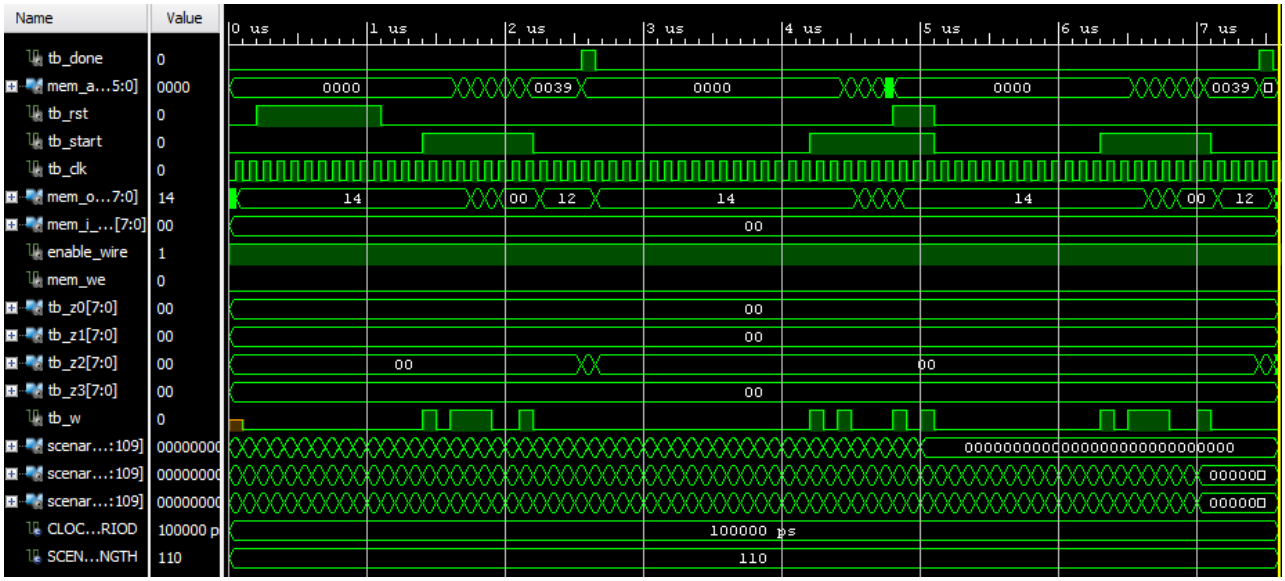

Figura 7.2: Waveform del Reset Asincrono

- **Doppia Computazione**: si vuole testare il caso in cui il segnale `i_start` si alzi subito dopo il segnale di `i_done` alto. Distinguiamo i due seguenti casi:
  - `i_start` diventa alto dopo un ciclo di clock da quando `i_done` torna a zero: mostriamo il risultato di questa simulazione nella figura 8.1.

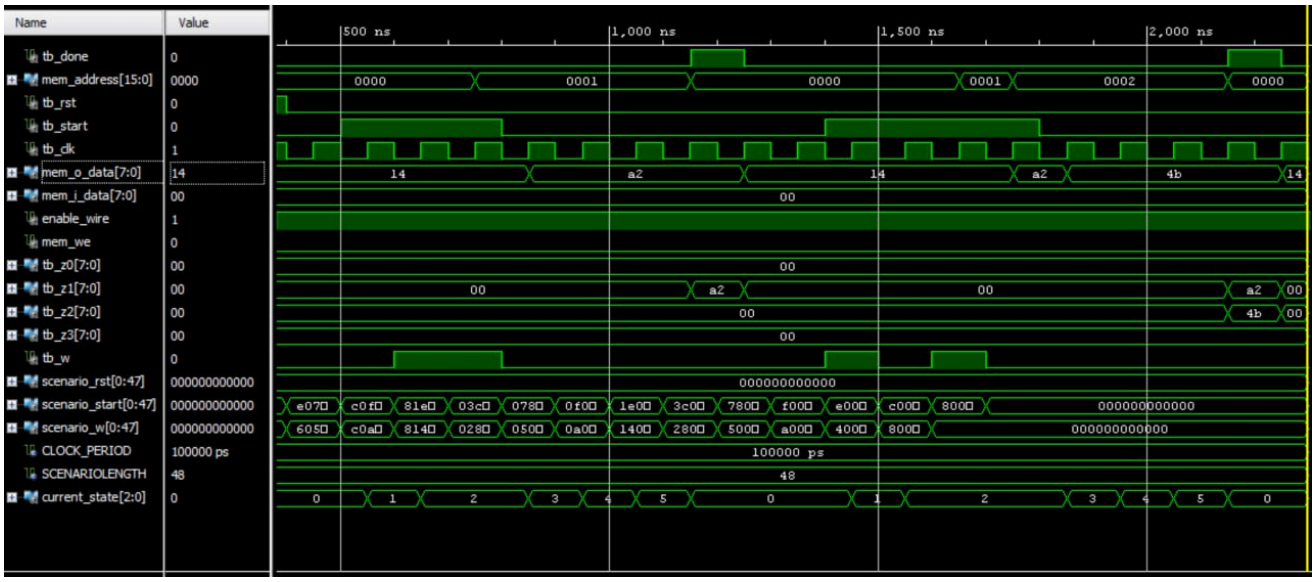

Figura 8.1: Waveform della Doppia Computazione

  - `i_start` diventa alto subito dopo che `i_done` si abbassa: mostriamo il risultato di questa simulazione nella figura 8.2.

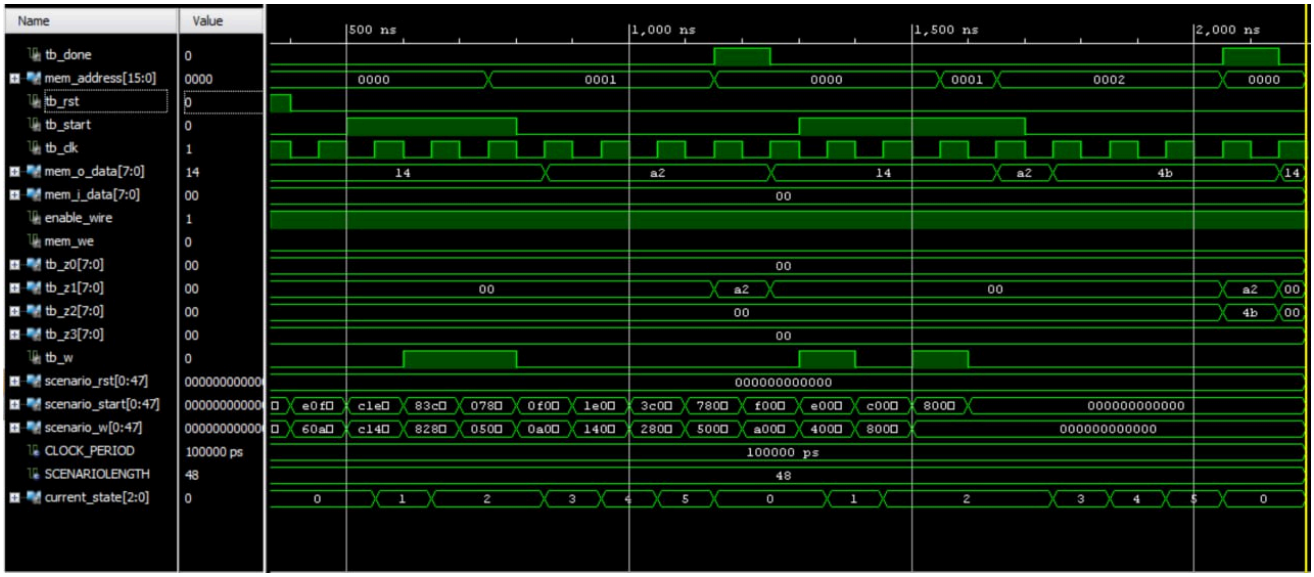

Figura 8.2: Waveform della Doppia Computazione

- **Test Randomici**: abbiamo simulato tutti i Test Bench proposti come esempio e generato altri Test Bench randomici, di cui non riportiamo il waveform per motivi di spazio. Tutti i test vengono superati correttamente.

4. Conclusioni

Il componente sintetizzato supera correttamente tutti i test specificati nelle tre simulazioni: Behavioral, Post-Synthesis Functional e Post-Synthesis Timing.
Le ottimizzazioni attuate si focalizzano principalmente sulla riduzione degli stati della FSM e sulla giusta sincronizzazione tra i vari segnali d'uscita.

L'inizio del nostro lavoro si è focalizzato sulla costruzione della macchina a stati finiti e del datapath, che hanno reso più agevole ed ordinata l'implementazione del codice. Successivamente, abbiamo iniziato la fase di debug e corretto gli errori trovati.
In un primo momento il modulo, seppur funzionante nella simulazione post-synthesis functional, presentava dei ritardi di propagazione delle uscite e fenomeni di glitch del segnale `o_done` nella simulazione post-synthesis timing. Pertanto abbiamo aggregato alcuni stati intermedi in modo da poter aggiungere uno stato di ritardo senza aumentare eccessivamente il numero totale degli stati della FSM.

Il modulo così ottenuto rispetta tutte le specifiche richieste e presenta tutte le ottimizzazioni e le accortezze di cui abbiamo discusso nei paragrafi precedenti.