Prova Finale (Progetto di Reti Logiche)

Prof. Fabio Salice - Anno 2019/2020

Lorenzo Mainetti (Codice Persona 10622242 - Matricola 889554)

Indice

1 Introduzione 2
1.1 Scopo del progetto 2
1.2 Specifiche generali 2
1.3 Interfaccia del componente 3
1.4 Dati e descrizione memoria 4
2 Architettura 5
2.1 Stati della macchina 5
2.1.1 IDLE state 5
2.1.2 READ state 5
2.1.3 CODE state 5
2.1.4 WRITE state 5
2.1.5 DONE state 6
2.2 Scelte progettuali 6
3 Risultati sperimentali 7
3.1 Registri sintetizzati 7
3.2 Report di utilizzo 8
3.3 Tempi di esecuzione 8
4 Simulazioni 9
5 Conclusioni 11
5.1 Ottimizzazioni 11

1 Introduzione

1.1 Scopo del progetto

Partendo dal metodo di codifica a bassa dissipazione denominato “Working Zone”, si vuole progettare ed implementare un componente hardware (encoder), descritto in VHDL, che ricevuti in ingresso l’indirizzo da codificare e gli indirizzi base delle working zone, calcoli la codifica secondo la specifica, individuando l’eventuale appartenenza ad una particolare working zone.

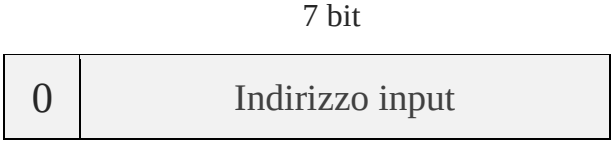
1.2 Specifiche generali

Le Working Zone sono definite come un intervallo di indirizzi di dimensione fissa (Dwz) caratterizzate da un numero identificativo (Nwz). La codifica dell’indirizzo in ingresso è composta da (1+n+m) bit: il bit più significativo, se a ‘1’, indica l’appartenenza ad una generica wz; i successivi n bit specificano il numero della wz di appartenenza, in binario naturale; infine gli ultimi m bit rappresentano l’offset rispetto all’indirizzo base, in codifica one-hot (con n = $\log_2$(Nwz) e m = Dwn).
Se invece l’indirizzo in ingresso non appartiene ad alcuna wz il primo bit viene posto a ‘0’ e i rimanenti bit riportano l’indirizzo così com’è.

Nella specifica assegnata sono definite 8 working zone, ognuna composta da 4 indirizzi, l’input è a 8 bit con il bit più significativo a ‘0’ e 7 bit di indirizzo. Anche la codifica è a 8 bit e si presenta nei seguenti modi:

- se l’indirizzo in input appartiene a una working zone

- se l’indirizzo in input non appartiene a una working zone (in questo caso l’output è uguale all’input



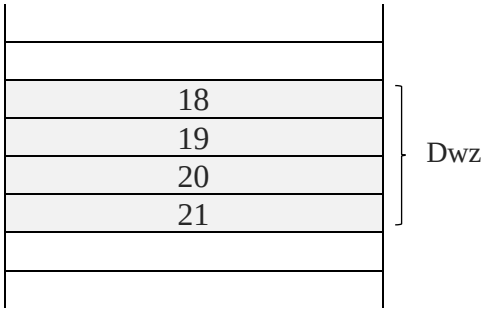


Figura 1: esempio working zone

Prendendo come esempio la working zone mostrata in figura (e supponendo sia la prima):

- Se l’indirizzo in ingresso è 20
  Codifica = “1 000 0100”
- Se l’indirizzo in ingresso è 34
  Codifica = “0 0100010”

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_start è il segnale di START generato dal test bench;
- i_rst è il segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START;
- i_data è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
- o_address è il segnale (vettore) di uscita che permette di indirizzare la memoria in lettura e in scrittura (ad esempio ponendo o_address a 8, a i_data sarà assegnato il contenuto dell'ottava cella della RAM);
- o_done è il segnale di uscita che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria;
- o_en è il segnale di ENABLE da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- o_we è il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter scriverci. Per leggere da memoria esso deve essere 0;
- o_data è il segnale (vettore) di uscita dal componente verso la memoria.

1.4 Dati e descrizione memoria

La RAM, con cui il componente deve interfacciarsi e che viene simulata dal test bench, è composta da 2^16 parole da 8 bit. Gli indirizzi di memoria sono su 16 bit e di interesse sono solo i primi 9 indirizzi:

- Gli indirizzi dallo 0 al 7 sono usati per memorizzare gli indirizzi base delle working zone
- L'indirizzo 8 è usato per memorizzare l'indirizzo da codificare
- L'indirizzo 9 è usato per scrivere la codifica dell'indirizzo

Indirizzo 0	Indirizzo base wz0
Indirizzo 1	Indirizzo base wz1
Indirizzo 2	Indirizzo base wz2
Indirizzo 3	Indirizzo base wz3
Indirizzo 4	Indirizzo base wz4
Indirizzo 5	Indirizzo base wz5
Indirizzo 6	Indirizzo base wz6
Indirizzo 7	Indirizzo base wz7
Indirizzo 8	Indirizzo da codificare
Indirizzo 9	Codifica

Figura 2: Rappresentazione indirizzi significativi della memoria

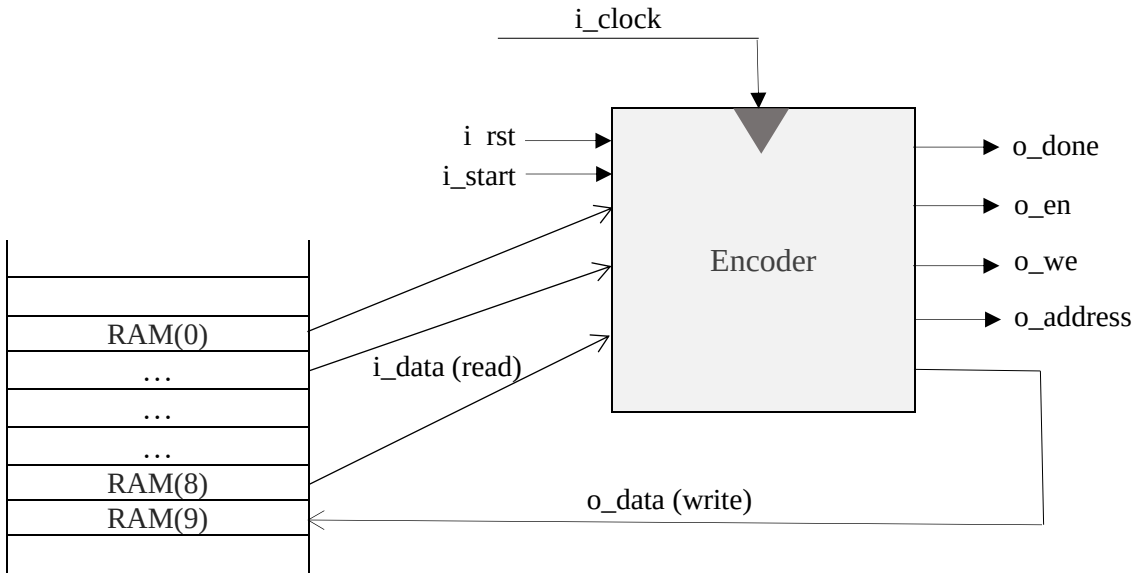


Figura 3: Rappresentazione interfacciamento componente con memoria

2 Architettura

La specifica prevede l'adozione di un meccanismo di full interlocking. Quando il segnale i_start in ingresso viene portato a '1', il componente sviluppato inizia l'elaborazione spostandosi dallo stato IDLE al primo stato della computazione. Una volta terminata la computazione, dopo avere scritto il risultato in memoria, il componente alza il segnale o_done. Il test bench risponde abbassando i_start e, successivamente, il componente riporta a '0' o_done; il componente ritorna quindi nello stato IDLE, in attesa che il segnale i_start torni alto.

Il componente dispone inoltre di un segnale i_rst che, insieme agli altri appena elencati, suggerisce di definire una macchina a stati finiti. Nelle seguenti sezioni troviamo infatti sia la descrizione della FSM (parte combinatoria), sia la descrizione della parte sequenziale della macchina che permette la gestione dei registri utilizzati.

2.1 Stati della macchina

La macchina è costituta da soli 5 stati, si è infatti cercato di minimizzarne il numero. Di seguito viene fornita una breve descrizione di ciascuno stato.

2.1.1 IDLE state

È lo stato iniziale, in cui si attende che il test bench alzi il segnale i_start, quindi si richiede l'indirizzo da codificare e si prosegue con lo stato successivo. In caso venga alzato il segnale i_rst, si torna in questo stato.

2.1.2 READ state

In questo stato viene letto l'indirizzo da codificare e si richiede l'indirizzo base delle working zone corrente. Si passa poi allo stato successivo.

2.1.3 CODE state

Stato in cui, dopo aver letto l'indirizzo della working zone corrente e calcolati gli offset possibili, vengono effettuati i confronti con l'indirizzo da codificare. Se c'è corrispondenza tra l'indirizzo da codificare e uno degli indirizzi della working zone, si richiede la scrittura in memoria della codifica (oppure dell'indirizzo inalterato se sono state lette tutte le wz senza individuare corrispondenze) e si passa allo stato successivo. Altrimenti si torna nello stato di READ.

2.1.4 WRITE state

Stato in cui viene portata a termine la scrittura in memoria e quindi viene posto o_done a '1'. Si passa dunque all'ultimo stato.

2.1.5 DONE state

In questo stato si attende che il test bench abbassi i_start. Una volta conclusa l'attesa, si procede abbassando o_done e tornando nello stato IDLE.

2.2 Scelte progettuali

La principale scelta progettuale effettuata è stata quella di descrivere il componente con un unico modulo composto da due processi:

1. Il primo rappresenta la parte sequenziale della macchina e serve per gestire il register transfer e quindi come vengono manipolati i registri. Si è scelto di utilizzare un reset sincrono.

2. Il secondo rappresenta invece la parte combinatoria della FSM, che analizza i segnali in ingresso e lo stato corrente per determinare il prossimo stato in cui evolverà il sistema.

La codifica viene valutata in modo molto semplice avvalendosi del costrutto if-then-else. Viene poi utilizzata la ADD per incrementare l'indirizzo di memoria da richiedere e il numero della working zone corrente e il CAST per assegnare un intero a un registro.

Inizialmente si era pensato ed implementato in modo funzionante un approccio che leggesse e salvasse le working zone dopo ogni reset, sapendo che tra due segnali di reset possono essere letti e codificati più indirizzi. Nel trade-off tempo-area si dava quindi maggior risalto al tempo di esecuzione nel caso di molte codifiche. Non potendo però fare assunzioni sul numero di codifiche con working zone invariate, si è deciso di pensare ad una soluzione che rimanesse più generale e questa volta ponesse in risalto l'area.

Nell'implementazione fornita si è quindi preferito mantenere memorizzate meno informazioni possibili, utilizzando così un minor numero di FF e LUT e di conseguenza un'area ridotta. Gli indirizzi delle working zone vengono salvati solamente per il confronto con l'indirizzo da codificare corrente e sovrascritte con l'analisi delle working zone successive. Questo approccio offre oltre a costi minori, una maggiore scalabilità in quanto, se si dovesse aumentare il numero di working zone da analizzare, non sarebbero necessarie modifiche al codice se non quelle di aumentare il range di interi necessari a rappresentare il numero della wz e la condizione per scrivere, nel caso non sia riscontrata appartenenza ad alcuna working zone.

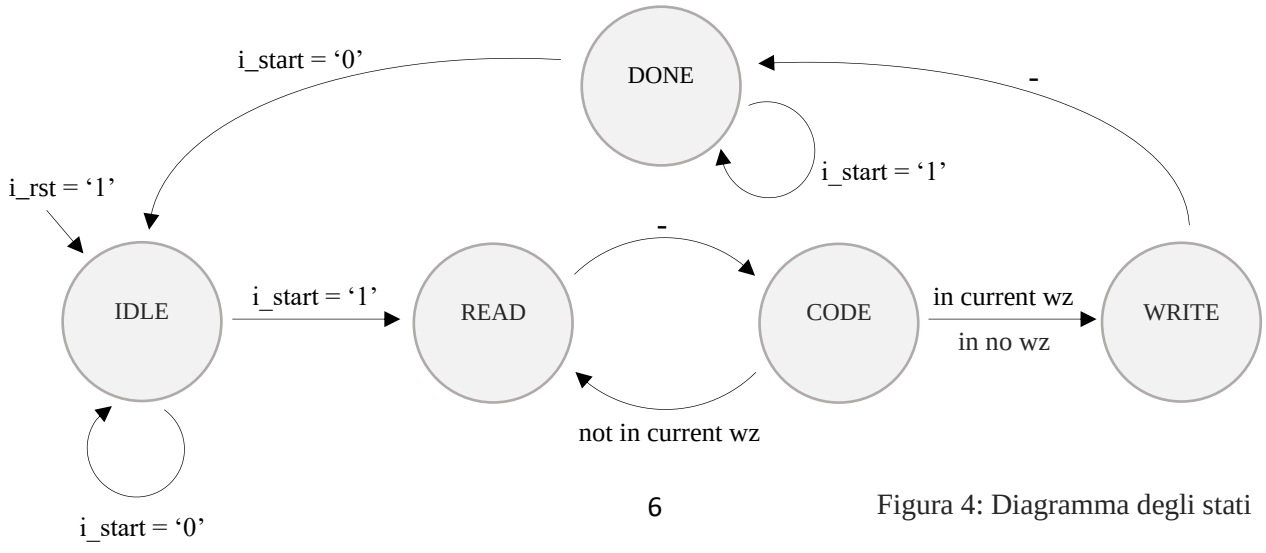


Figura 4: Diagramma degli stati

3 Risultati sperimentali

Come anticipato, viste le scelte progettuali compiute, l'aspettativa della sintesi è un componente che presenta un'area relativamente ridotta e un tempo di esecuzione, che può variare molto in base agli input. Di seguito viene riportata la schematic del componente sintetizzato.

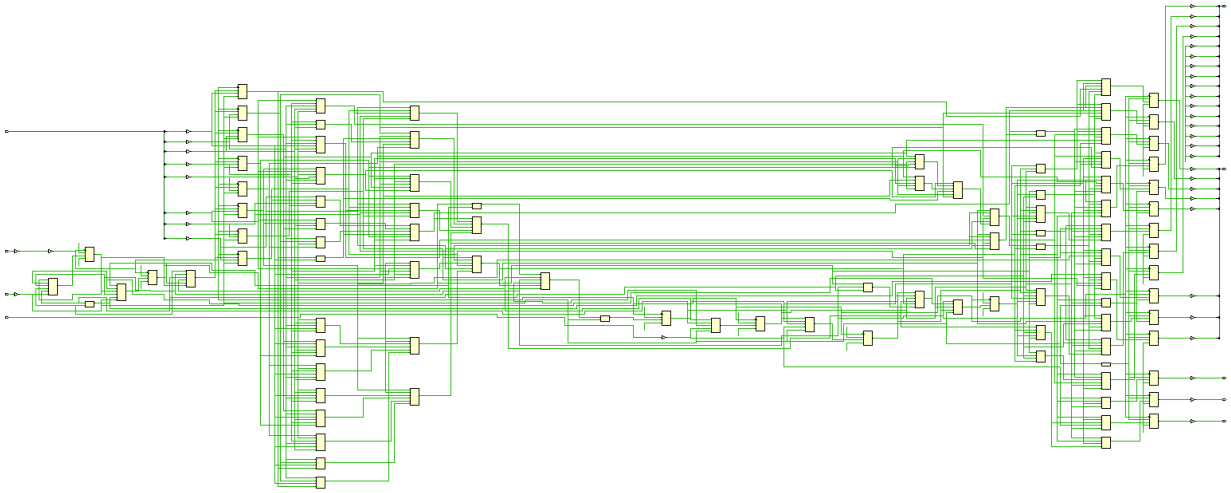


Figura 5: Vivado schematic del componente sintetizzato

3.1 Registri sintetizzati

Analizzando il "Vivado Synthesis Report" in primo luogo si nota che i 5 stati vengono codificati in binario naturale, utilizzando quindi tre bit. Si può successivamente osservare che i registri vengono sintetizzati come descritto nel codice. In particolare viene segnalata la creazione di 7 registri (per un totale di 29 Flip Flop D utilizzati).

Numero bit	Numero registri	Contenuto
16	1	Indirizzo di memoria (o_address)
8	2	Indirizzo in ingresso da codificare; indirizzo in uscita codificato
3	1	contatore a 3 bit
1	3	Bit di done; bit di scrittura; bit di lettura

3.2 Report di utilizzo

Proseguendo nell'analisi del report è interessante riportare i dati di utilizzo che, come anticipato mostrano il numero ridotto di elementi del design sintetizzato.

```
Report Cell Usage:
+------+-----+------+
|      |Cell |Count |
+------+-----+------+
|1     |BUFG |     1|
|2     |LUT1 |     1|
|3     |LUT2 |     7|
|4     |LUT3 |     5|
|5     |LUT4 |     8|
|6     |LUT5 |    12|
|7     |LUT6 |    36|
|8     |FDRE |    29|
|9     |IBUF |    11|
|10    |OBUF |    27|
+------+-----+------+

Report Instance Areas:
+------+---------+-------+------+
|      |Instance |Module |Cells |
+------+---------+-------+------+
|1     |top      |       |   137|
+------+---------+-------+------+
```

Figura 6: Estratto del Vivado Synthesis Report

Risorsa	**Utilizzo**	**Disponibilità**	**Utilizzo in %**
Look Up Table	59	134600	0.04%
Flip Flop	29	269200	0.01%

Figura 7: Tabella contenente dati di utilizzo

3.3 Tempi di esecuzione

Per analizzare le tempistiche del componente sintetizzato si possono sfruttare due casi semplici, ma che ne mettono in luce l'alta variabilità e la dipendenza dall'indirizzo da codificare in ingresso.

Caso ottimo: l'indirizzo da codificare si trova nella prima working zone. Il tempo di esecuzione sarà minimo; infatti l'indirizzo verrà subito codificato e poi scritto in memoria, senza la necessità di leggere altre working zone al di fuori della prima. Tmin = **900,10 ns** (1000 ns in behavioral).

Caso pessimo: l'indirizzo da codificare non si trova in alcuna working zone. Il tempo di esecuzione sarà massimo; infatti sarà necessario leggere tutte le working zone e solo alla fine, dopo averlo confrontato con tutte, concludere che non appartiene a nessuna di queste e quindi scriverlo inalterato in memoria. Tmax = **2300,10 ns** (2400 ns in behavioral).

4 Simulazioni

Per verificare e validare il corretto funzionamento del componente, esso è stato sottoposto a numerosi test bench, partendo da quelli forniti come esempio e formulandone successivamente altri con l'obiettivo di testare i casi critici e i casi limite. Di seguito vengono brevemente descritti i test utilizzati e per quelli più significativi viene anche mostrato il corretto funzionamento grazie allo screenshot dell'andamento dei segnali durante la simulazione. Da specifica tutti i test sono stati eseguiti con periodo di clock di 100 ns, sono però supportate correttamente anche frequenze di clock molto maggiori (periodi di clock fino a 3 ns).

Test bench base:

1. **Appartenenza working zone**: il test verifica che l'indirizzo da codificare assegnato appartenga alla corretta wz e che quindi la codifica sia corretta.

2. **Non appartenenza working zone**: il test verifica che l'indirizzo da codificare assegnato non appartenga ad alcuna wz e che quindi venga scritto in memoria inalterato.

Test bench di verifica dei casi limite:

1. **Confini working zone**: il test verifica il corretto funzionamento in caso di wz con indirizzo base 0 oppure 124, rispettivamente valori minimo e massimo rappresentabili e l'indirizzo da codificare appartenente a questa wz limite.

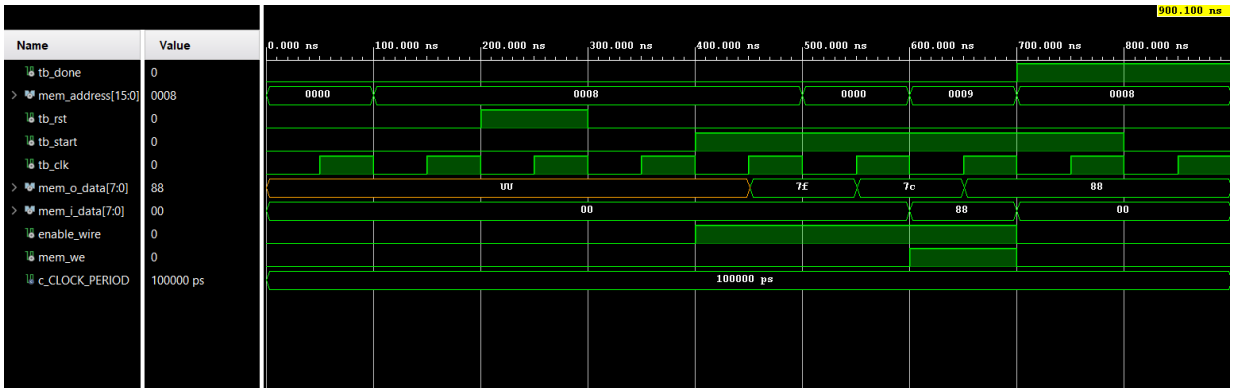


Figura 8: Simulazione con indirizzo base della prima working zone uguale a 124

2. **Confini indirizzo in ingresso**: il test verifica il corretto funzionamento nel caso l'indirizzo da codificare valga 0 oppure 127, rispettivamente valori minimo e massimo rappresentabili.

3. **Working zone adiacenti**: il test verifica il corretto funzionamento in caso di wz consecutive (l'indirizzo base di una wz è il successivo del quarto indirizzo di un'altra wz).

Test bench per verificare il corretto funzionamento dei segnali:

1. **Reset asincrono**: il test verifica che il trigger asincrono del segnale di reset non comprometta la computazione e che questa riinizi facendo ritornare la macchina nello stato iniziale IDLE.

2. **Multi start**: il test verifica la corretta sincronizzazione dei segnali i_start, i_rst, o_done simulando due volte la memoria, mantenendo inalterate le working zone e cambiando l'indirizzo da codificare.

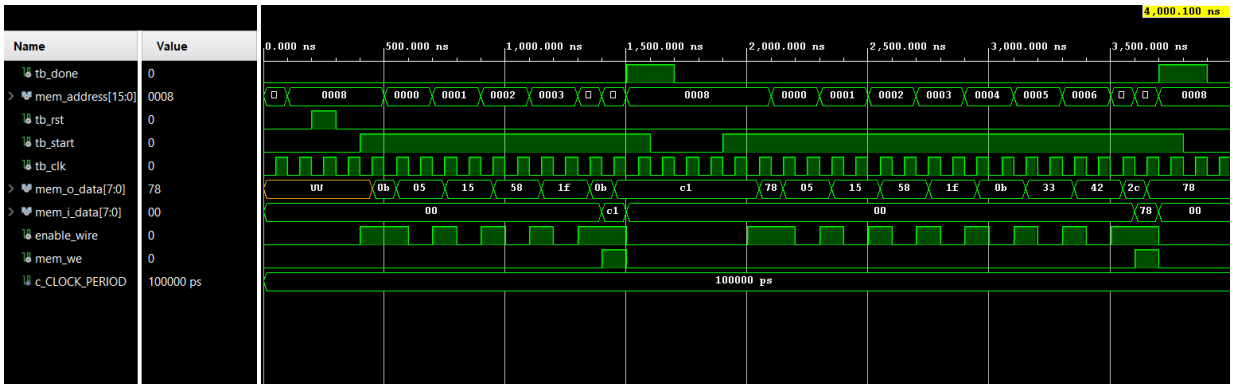


Figura 9: Simulazione con doppio segnale i_start e doppia computazione

3. **Multi reset**: il test verifica la corretta sincronizzazione dei segnali i_start, i_rst, o_done, simulando due volte la memoria e cambiando i dati in ingresso (indirizzi base wz e volendo anche indirizzo da codificare).

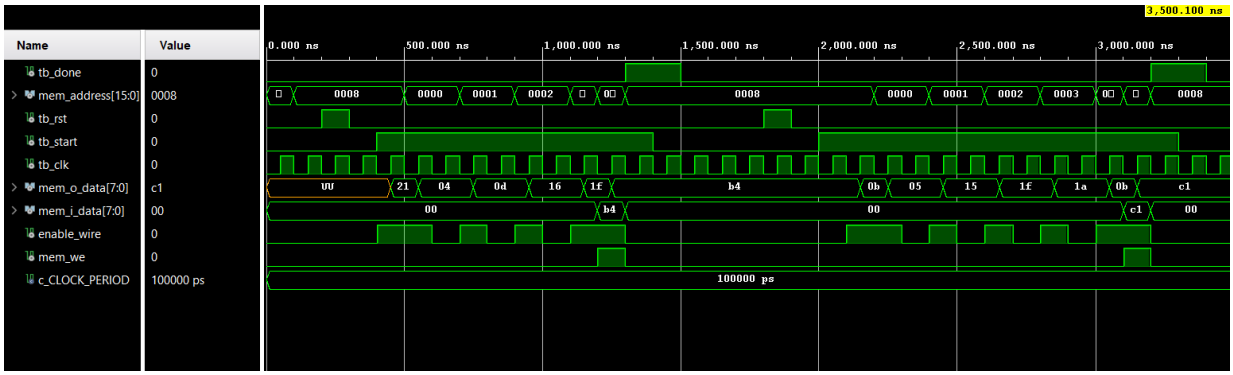


Figura 10: Simulazione con doppio segnale i_rst

Oltre ai test descritti, sono stati simulati anche un gran numero di test randomici per testare ulteriormente il componente. In particolare si è sfruttato un test bench che legge da file esterno i valori contenuti nelle prime 8 celle della RAM e verifica se per ogni set di ingressi il test viene passato, scrivendo su due file esterni i test passati e quelli non passati.

Il componente ha passato correttamente tutti i test a cui è stato sottoposto, sia in *Behavioral Simulation* che in *Post-Synthesis Functional Simulation* (nonostante non sia richiesto da specifica, è supportata anche la fase di implementazione e viene svolta correttamente la *Post-Implementation Functional Simulation*).

5 Conclusioni

Riassumendo si è progettato e implementato un design con queste caratteristiche:

- Funzionante in pre e post sintesi
- Ottimizzato per avere il numero minimo di stati
- Ottimizzato per avere un'area ridotta
- Utilizzo di 59 LUT e 29 FF

5.1 Ottimizzazioni

Le ottimizzazioni attuate in fase di progettazione, come precedentemente menzionato, sono la minimizzazione del numero di stati della FSM e la riduzione dell'area utilizzata, con particolare attenzione al numero di FF e di LUT. La prima è stata raggiunta per fasi successive, raffinando e combinando più stati, come ad esempio uno stato inizialmente dedicato alla lettura dell'indirizzo da codificare, poi unito allo stato di READ. La seconda, in parte conseguenza della prima, ha inoltre riguardato l'adozione di alcuni oggetti *variable* e l'eliminazione di *signal* superflui.

Alcune ottimizzazioni ancora possibili potrebbero essere; un'ulteriore riduzione dell'area occupata, ad esempio sostituendo nel codice alcuni *signal* con corrispondenti *variable*. E probabilmente una più efficiente gestione dei cicli di clock, con conseguente aumento della velocità di esecuzione.