POLITECNICO
MILANO 1863

PROVA FINALE DI RETI LOGICHE [054441]

Studente: Davide Giacomini
Codice Persona: 10567357
Matricola: 889237

SOMMARIO

1. INTRODUZIONE

La seguente documentazione ha l'obiettivo di motivare il lavoro svolto per la Prova Finale di Reti Logiche con riferimento all'anno accademico 2019/2020.

La specifica fa riferimento ad un metodo di codifica degli indirizzi a bassa dissipazione di potenza denominato "Working Zone", pensato ed usato per il Bus Indirizzi.

Nel caso in cui l'indirizzo trasmesso appartenga a certi intervalli, detti appunto working-zone, viene codificato un nuovo indirizzo. La codifica fa riferimento al numero della working-zone e all'offset dell'indirizzo trasmesso rispetto all'indirizzo base dell'intervallo. Nel caso in cui l'indirizzo non appartenga a nessuna working-zone, il valore codificato sarà identico al valore dell'indirizzo. Per distinguere i due casi, alla codifica è necessario aggiungere un bit (il più significativo) che, se posto ad uno, denota l'appartenenza ad una working-zone.

La specifica prevede la presenza di 8 working-zone, mentre la dimensione della working-zone è di 4 indirizzi, incluso quello base. Gli indirizzi base delle working-zone vengono salvati nelle prime 8 celle di memoria, dall'indirizzo 0 all'indirizzo 7 compresi, cifre che corrispondo anche al numero della working-zone. Ad esempio, il valore salvato all'indirizzo 3 della memoria corrisponde all'indirizzo base della working-zone numero 3. La specifica prevede l'utilizzo anche dell'indirizzo 8 e 9 della memoria. Il primo contiene il valore dell'indirizzo da codificare, il secondo invece conterrà il valore codificato secondo le regole spiegate precedentemente.

Solitamente le working-zone sono le aree di memoria maggiormente utilizzate, poiché la codifica che viene impiegata (l'offset, ad esempio, è codificato in one-hot) è a bassa dissipazione di potenza. Di conseguenza anche il metodo Working Zone risulta, nel suo complesso, a bassa dissipazione di potenza.

Di seguito viene riportato prima un esempio in cui l'indirizzo non appartiene ad alcuna working-zone, poi un esempio in cui invece c'è corrispondenza.
Si noti come nel primo caso il valore codificato in output è identico al valore dell'indirizzo da codificare. Nel secondo caso invece il valore è stato codificato con le regole dettate sopra.

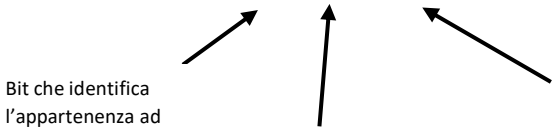
CASO 1 CON VALORE NON PRESENTE IN NESSUNA WORKING-ZONE

Indirizzo Memoria	**Valore**	**Commento**
0	4	// Indirizzo Base WZ 0
1	13	// Indirizzo Base WZ 1
2	22	// Indirizzo Base WZ 2
3	31	// Indirizzo Base WZ 3
4	37	// Indirizzo Base WZ 4
5	45	// Indirizzo Base WZ 5
6	77	// Indirizzo Base WZ 6
7	91	// Indirizzo Base WZ 7
8	42	// ADDR da codificare
9	42	// Valore codificato con in OUTPUT

CASO 2 CON VALORE PRESENTE IN UNA WORKING-ZONE

Indirizzo Memoria	**Valore**	**Commento**
0	4	// Indirizzo Base WZ 0
1	13	// Indirizzo Base WZ 1
2	22	// Indirizzo Base WZ 2
3	31	// Indirizzo Base WZ 3
4	37	// Indirizzo Base WZ 4
5	45	// Indirizzo Base WZ 5
6	77	// Indirizzo Base WZ 6
7	91	// Indirizzo Base WZ 7
8	33	// ADDR da codificare
9	180	// Valore codificato con in OUTPUT

(1 - 011 - 0100)



Bit che identifica l'appartenenza ad una working-zone

Numero di working-zone, codificato in binario, che corrisponde all'indirizzo della memoria dove è salvato l'indirizzo base della working-zone: 3.

L'indirizzo da codificare è 33, e dato che l'indirizzo base della working-zone 3 è 31 ->

WZ_OFFSET = 33-31 = 2 (in one-hot)

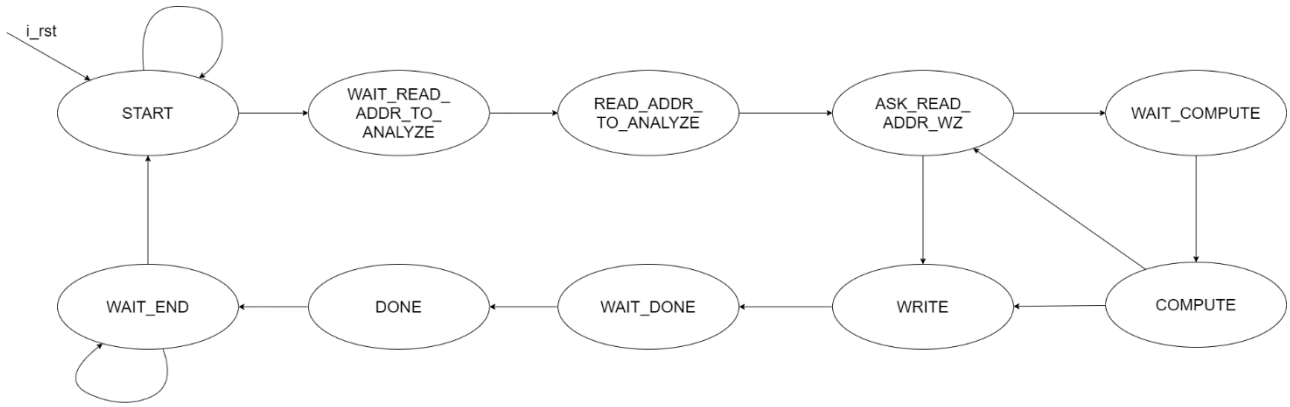
2. ARCHITETTURA

MACCHINA A STATI FINITI

Dopo una prima analisi della specifica si può osservare come ci siano due possibili approcci da poter seguire. Una prima possibilità è salvarsi gli indirizzi base di tutte le working-zone e l'indirizzo da codificare, per poi confrontare quest'ultimo in parallelo agli 8 indirizzi delle working-zone. Un secondo approccio, che è quello per cui ho optato, prevede un criterio sequenziale e la possibilità di non salvarsi gli indirizzi delle working-zone.

Nell'algoritmo pensato, l'indirizzo viene inizialmente salvato per poi essere riutilizzato ciclicamente fino a quando non viene caricata da memoria una working-zone alla quale appartiene. In quel caso, il ciclo interrompe e viene direttamente scritto in memoria il risultato. Se l'indirizzo non appartiene a nessuna working-zone, il ciclo continua fino a quando non sono state analizzate tutte le working-zone, per poi scrivere in memoria il risultato finale (che sarà uguale all'indirizzo, in questo caso).

Di seguito viene mostrato uno schema della macchina a stati, con la relativa descrizione di ogni stato. Durante la realizzazione dello schema si è notato che è necessario inserire uno stato di attesa ogniqualvolta si vuole leggere da memoria o scriverci.



Il segnale "i_rst" è il segnale di RESET che viene generato dal testbench. Nel processo è stata utilizzata la variabile count_WZ, che tiene il conto del numero di working-zone analizzate. Inoltre, per comodità, è stata creata una variabile chiamata "address_converted", in cui viene salvato l'indirizzo da scrivere in memoria. Di seguito è presente una spiegazione dettagliata riguardo al comportamento degli stati:

- START ➔ Quando viene alzato lo start, viene comunicato alla memoria che si vuole leggere e da quale indirizzo si vuole leggere. Se lo start è basso non succede niente ed al ciclo successivo il componente si troverà ancora in START.
- WAIT_ADDRESS_TO_ANALYZE ➔ Qui viene atteso un ciclo prima di salvare l'indirizzo.
- READ_ADDR_TO_ANALYZE ➔ In questo stato viene salvato l'indirizzo che poi verrà usato nel ciclo di analisi delle working-zone.

- ASK_READ_ADDR_WZ ➔ In questo stato count_WZ è fondamentale per decidere lo stato successivo. Se sono stati analizzati tutti gli indirizzi, lo stato successivo sarà direttamente WRITE, altrimenti si procede con la prossima working-zone e si passa a WAIT_COMPUTE.
- WAIT_COMPUTE ➔ Qua viene atteso un ciclo prima di usare l'indirizzo in ingresso dalla memoria (che sarà l'indirizzo base della working-zone che si vuole analizzare).
- COMPUTE ➔ In questo stato avviene l'effettiva analisi degli indirizzi. Viene confrontato l'indirizzo base della working-zone con l'indirizzo da codificare, salvato precedentemente. Se l'indirizzo appartiene alla working-zone si passerà allo stato WRITE, anche se non si fossero ancora analizzate tutte le working-zone. In caso contrario si torna allo stato ASK_READ_ADDR_WZ per procedere col ciclo.
- WRITE ➔ Viene scritto in memoria il valore finale.
- WAIT_DONE ➔ A questo punto si attende che la scrittura in memoria faccia il suo corso in attesa di entrare nello stato DONE.
- DONE ➔ Qua si comunica alla memoria di aver finito con la computazione e si alza il segnale di done.
- WAIT_END ➔ In questo stato si attende che il segnale di start venga abbassato. Quando risulta abbassato, viene resettato il componente e si torna allo stato di START, in attesa di un nuovo trigger. Altrimenti, si rimane in questo stato ad attendere.

GESTIONE PROBLEMATICHE

Ci sono state tre principali problematiche da affrontare:

1. Come confrontare l'indirizzo con le working-zone.
2. Come gestire il segnale di reset.
3. Come gestire i tempi della memoria.

CASO 1:

Il confronto tra indirizzi è effettuato nello stato COMPUTE.

In questo stato viene analizzato il segnale "i_data", che corrisponde all'indirizzo base della working-zone che si vuole confrontare, senza la necessità di salvare il suo valore. La variabile "address_to_analyze" è l'indirizzo che era stato salvato nello stato READ_ADDR_ TO_ANALYZE e ora viene utilizzata per il confronto.

Se address_to_analyze < i_data allora non viene fatto nessun altro controllo perché risulta già all'esterno della working-zone che si sta analizzando. In caso contrario si effettua la differenza tra i due valori e, poiché il risultato è l'offset, si controlla se l'indirizzo da codificare sia nel range dei possibili offset. In caso positivo, viene costruito il nuovo indirizzo e si passa direttamente a WRITE.

Ho scelto questo algoritmo perché nei casi non pessimi non c'è bisogno di analizzare ogni working-zone, ma ci si può fermare prima. Nell'altro caso, invece, bisogna comunque salvarsi tutte le working-zone e solo dopo si può sfruttare il parallelismo. Sebbene nel secondo caso la vera e propria computazione venga fatta in parallelo, per numeri così piccoli ho ritenuto ragionevole non preoccuparmi di questo aspetto.

CASO 2:

Il segnale di reset, essendo asincrono, viene gestito senza tenere conto dello stato attuale del componente. In caso di reset, vengono resettate tutte le variabili e lo stato successivo viene posto a START.

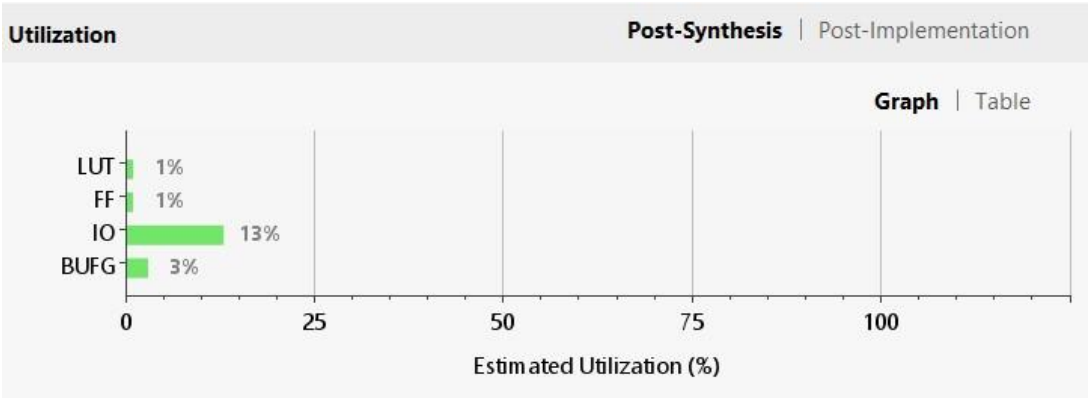
```
begin;
 if (i_rst = '1') then
          next_state <= START;
          count_WZ <= 0;
          o_done <= '0';
          o_en <= '0';
          o_we <= '0';
          o_address <= (others => '0');
          o_data <= (others => '0');
          address_to_analyze <= (others => '0');
          address_converted <= (others => '0');
 elsif (rising_edge(i_clk)) then

      -- solo se si entra nell'elsif lo stato del componente viene considerato

 end if;
end process;
```

CASO 3:

I tempi della memoria vengono gestiti con gli stati denominati WAIT_*. L'effetto che si ha con questi stati è la semplice attesa di un ciclo di clock prima di procedere l'esecuzione.

3. REPORT DI SINTESI

Non essendo richieste elevate prestazioni, non sono state ottimizzate in termini di LUT e FF, ma solo il numero di cicli che è necessario fare in caso l'indirizzo appartenga ad una working-zone. Di seguito vengono illustrate le percentuali post-sintesi stimate di utilizzo delle LUT, dei FF e dei blocchi IO. Successivamente si può anche osservare una tabella numerica più dettagliata.



Utilization — **Post-Synthesis** | Post-Implementation

Graph | **Table**

Resource	Estimation	Available	Utilization %
LUT	86	134600	0.06
FF	84	269200	0.03
IO	38	285	13.33
BUFG	1	32	3.13

Sono stati sintetizzati tutti e dieci gli stati con una codifica one-hot.

Il tempo di simulazione post-sintesi impiegato con il testbench ufficiale no_wz è risultato: "Time: 3650100 ps".
Per l'altro testbench invece il tempo di simulazione risulta: "Time: 2450100 ps".
Già da questi due risultati si nota come l'appartenenza alla working-zone può far migliorare i tempi di risposta anche di oltre il 30%. Le tempistiche saranno analizzate più in dettaglio nel prossimo capitolo.

4. TESTBENCH

Al fine di valutare il corretto funzionamento del componente, sono stati usati i due testbench ufficiali e altri testbench alternativi utili a coprire i seguenti casi:

- Reset asincroni durante l'esecuzione.
- I 4 casi di offset 0, 1, 2 o 3, al fine di coprire tutto il codice.
- I casi limite in cui l'indirizzo da codificare fa parte della prima working-zone e dell'ultima. Serve per testare che il contatore count_WZ lavori bene.
- Una grande varietà di casi generati in modo casuale, in cui rientrano anche i casi in cui le working-zone non siano salvate in ordine crescente all'interno della memoria.

Notare che ci sarebbe la necessità di testare due ulteriori casi: il primo si verifica quando l'indirizzo risulta minore dell'indirizzo base di una working-zone, il secondo invece quando risulta maggiore dell'ultimo indirizzo di una working-zone. Dato che questi due casi vengono già verificati da uno dei due testbench ufficiali, non è stato necessario crearne di alternativi.

E' stata effettuata, per ogni testbench, prima una simulazione comportamentale al fine di valutare i valori delle variabili durante la computazione, poi una simulazione post-sintesi, al fine di valutare attentamente, tramite la waveform, che i segnali di ingresso e di uscita siano quelli attesi. Di seguito vengono riportate le waveform delle simulazioni post-sintesi.

TESTBENCH UFFICIALI

I testbench forniti sono due. Il primo di cui illustrerò l'esito verifica il caso in cui l'indirizzo non faccia parte di alcuna working-zone. La memoria di questo testbench è composta da questi indirizzi:

```
signal RAM: ram_type := (     0 => std_logic_vector(to_unsigned( 4 , 8)),
                              1 => std_logic_vector(to_unsigned( 13 , 8)),
                              2 => std_logic_vector(to_unsigned( 22 , 8)),
                              3 => std_logic_vector(to_unsigned( 31 , 8)),
                              4 => std_logic_vector(to_unsigned( 37 , 8)),
                              5 => std_logic_vector(to_unsigned( 45 , 8)),
                              6 => std_logic_vector(to_unsigned( 77 , 8)),
                              7 => std_logic_vector(to_unsigned( 91 , 8)),
                              8 => std_logic_vector(to_unsigned( 42 , 8)),   -- indirizzo da codificare
        others => (others =>'0'));
```

La waveform è riportata nella pagina successiva, in figura 1.

Come da aspettative, si nota che dalla memoria sono stati passati tutti e 8 gli indirizzi base delle working-zone. Infatti, il segnale mem_address, in azzurro, mostra che ogni 3 cicli di clock (in bianco) viene incrementato l'indirizzo della memoria dal quale leggere il valore. Vengono letti tutti e 9 gli indirizzi (da 0 a 8) perché non c'è appartenenza a nessuna working-zone. Questo è infatti il caso pessimo, quando tutte le working-zone devono essere testate.

Notare inoltre che il segnale di done (primo segnale in magenta) viene alzato con un ritardo di un ciclo di clock da quando la memoria legge il valore finale codificato, cioè $2a_{(HEX)} = 42_{(DEC)}$. Questo è il frutto dell'attesa implementata con lo stato di wait.

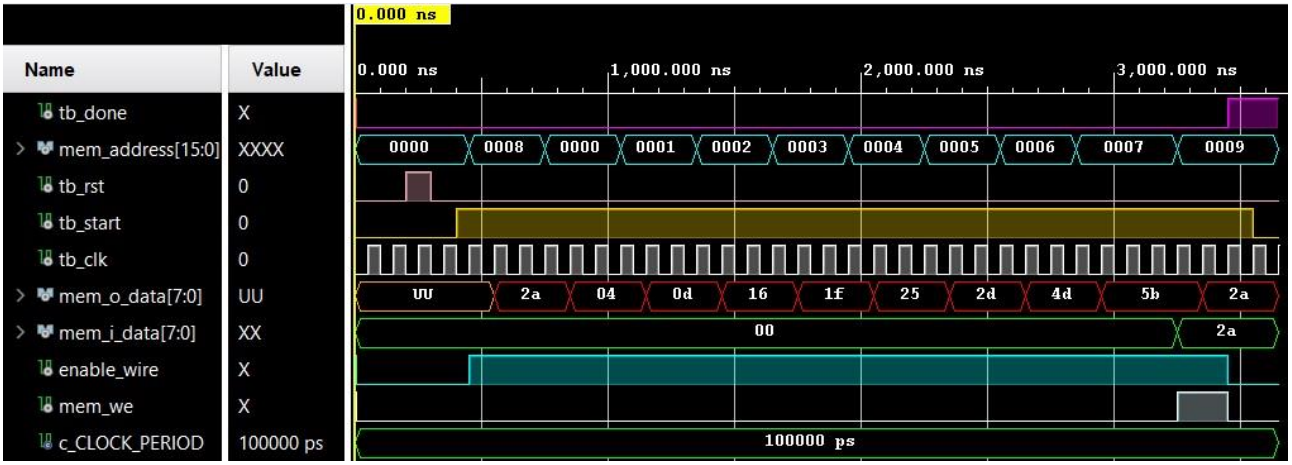


Figura 1 -- waveform testbench ufficiale, no wz

Il secondo dei due testbench ufficiali verifica il funzionamento nel caso in cui l'indirizzo faccia parte di una working-zone. In particolare, in questo test, l'indirizzo fa parte della quarta working-zone e ha un offset uguale a 2. Ci si aspetta quindi che vengano analizzate le prime 4 working-zone e che in seguito venga scritto subito in memoria il risultato, senza analizzarne altre. La memoria, in questo testbench, è composta da questi indirizzi:

```
signal RAM: ram_type := (     0 => std_logic_vector(to_unsigned( 4 , 8)),
                              1 => std_logic_vector(to_unsigned( 13 , 8)),
                              2 => std_logic_vector(to_unsigned( 22 , 8)),
                              3 => std_logic_vector(to_unsigned( 31 , 8)),        -- indirizzo base della 4ª wz
                              4 => std_logic_vector(to_unsigned( 37 , 8)),
                              5 => std_logic_vector(to_unsigned( 45 , 8)),
                              6 => std_logic_vector(to_unsigned( 77 , 8)),
                              7 => std_logic_vector(to_unsigned( 91 , 8)),
                              8 => std_logic_vector(to_unsigned( 33 , 8)),        -- indirizzo da codificare
                others => (others =>'0'));
```

Ci si aspetta che l'indirizzo codificato sarà così composto:

- Il primo bit sarà posto a 1.
- Il numero della working-zone, codificato in binario, dovrà essere $3_{(DEC)} = 011_{(BIN)}$.
- L'offset, codificato in one-hot, dovrà essere $2_{(DEC)} = 0100_{(one\text{-}hot)}$.

La waveform è riportata in figura 2, nella pagina successiva.

L'indirizzo codificato risulta $b4_{(HEX)} = 180_{(DEC)} = 1\ 011\ 0100_{(BIN)}$, che è quello che ci si aspettava. Notare inoltre che la simulazione è durata 1 µs in meno rispetto al primo testbench, poiché non si sono analizzate tutte le working-zone. Questo fatto è facilmente visibile in mem_address, in azzurro, dove a 1900ns si passa direttamente dall'indirizzo 3 all'indirizzo 9, saltando quindi le working-zone dalla posizione 4 alla 7 della memoria.

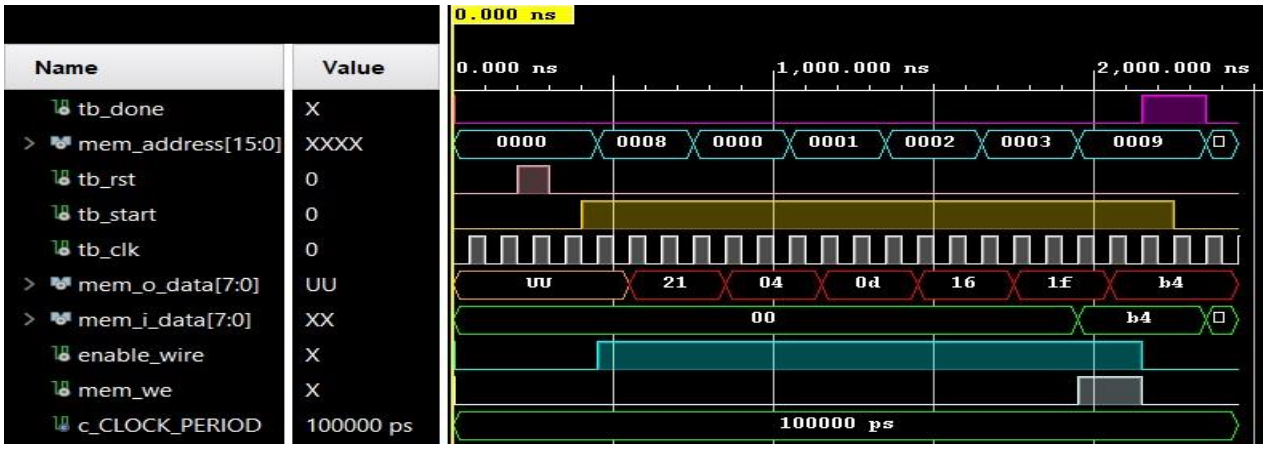


Figura 2 -- waveform testbench ufficiale, in wz

TESTBENCH ALTERNATIVI

Di seguito non riporterò la waveform per ogni testbench alternativo, ma mi limiterò a descrivere i test effettuati.

Per testare il reset ho modificato il testbench ufficiale alzando il segnale di reset in due casi: prima che venga scritto il segnale in memoria e dopo averlo fatto. In entrambi i casi il test è andato a buon fine. Di seguito mostro un estratto del processo del testbench. Notare che col primo reset, aspettando 50ns in più, ho anche testato la sensibilità all'asincronismo.

```
begin
    wait for 100 ns;
    report "prova";
    wait for c_CLOCK_PERIOD;
    tb_rst <= '1';
    wait for c_CLOCK_PERIOD;
    tb_rst <= '0';
    wait for c_CLOCK_PERIOD;
    tb_start <= '1';
    wait for 20*c_CLOCK_PERIOD;
    wait for 50ns;
    tb_rst <= '1';
    wait for c_CLOCK_PERIOD;
    tb_rst <= '0';
    wait until mem_we = '1';
    wait for c_CLOCK_PERIOD;
    tb_rst <= '1';
    wait for c_CLOCK_PERIOD;
    tb_rst <= '0';
    wait until tb_done = '1';
    wait for c_CLOCK_PERIOD;
    tb_start <= '0';
    wait until tb_done = '0';
    wait for 100 ns;

    -- qua ho testato che l'indirizzo fosse corretto

end process test;
```

Per gli altri tre punti descritti all'inizio di questo capitolo si è utilizzato un testbench generale con un numero elevato di casistiche casuali. Per testare anche i casi limite sono stati modificati manualmente alcuni casi.

Sono stati passati tutti i testbench con successo, verificando quindi la correttezza del componente.

5. CONCLUSIONI

Queste pagine hanno avuto l'obiettivo di descrivere la realizzazione del componente, dimostrandone la correttezza ed esponendo i ragionamenti che sono stati seguiti nel corso della progettazione.

Dato che non erano richiesti limiti di prestazione, si è preferito concentrarsi sulla correttezza, pur con probabilmente qualche attesa di troppo, ma con l'obiettivo di evitare inconsistenza dovuta ad eventuali ritardi dei segnali.

Sono state riportate solo le waveform dei testbench ufficiali, poiché riportarle tutte sarebbe risultata un'inutile ripetizione, ma con la speranza di dimostrare appieno la correttezza del componente.

Infine, si è voluto anche riportare qualche estratto di codice nelle sue parti più significative, per migliorare la comprensione del progetto.