Prova finale (Progetto di Reti Logiche)
Prof. Gianluca Palermo – Anno 2019/2020

Scritto da Edoardo Saputelli (Codice persona 10615291 – Matricola 889967)

Indice

1. Introduzione

1.1 Scopo del progetto

Lo scopo del progetto consiste nell'implementazione di un componente hardware, descritto nel linguaggio VHDL, che permetta la realizzazione del metodo di codifica degli indirizzi denominato "*Working zone*".

Al tal fine, nella specifica sono fornite sia la descrizione HW della memoria di tipo RAM, sia l'interfaccia del componente stesso.

1.2 Specifiche generali

Il metodo Working Zone è stato ideato per la codifica di indirizzi trasmessi lungo il Bus Indirizzi, al fine di trasformare il loro valore in modo dipendente dalla WZ alla quale essi eventualmente appartengono.

Ciascuna WZ è definita come un intervallo di indirizzi di dimensione fissa e si identifica con il corrispondente indirizzo base.

Nella realizzazione del metodo di codifica, la versione da implementare definisce 4 indirizzi come dimensione della WZ e 8 bit come dimensione di ciascun indirizzo. Inoltre il numero di WZ da considerare sarà 8.

Dato in ingresso un indirizzo a 7 bit, il risultato dell'esecuzione del circuito dovrà essere di 8 bit, codificato in base alla presenza dell'input in una delle Working Zone fornite.

Infatti, l'eventuale corrispondenza con una di esse comporterà, nell'indirizzo codificato, la variazione del bit iniziale, denominato WZ_BIT, il quale sarà posto uguale a 0 in caso di mancata corrispondenza con le 8 WZ, o ad 1, in caso contrario.

Dopodiché, nella prima situazione descritta, il completamento dell'indirizzo da scrivere in memoria sarà semplicemente lo stesso input ricevuto all'inizio dell'esecuzione, comportando un risultato della forma WZ_BIT (1 bit) & INDIRIZZO (7 bit).

D'altra parte, nella situazione opposta, il componente andrà a scrivere in memoria la codifica in base alla WZ con cui è avvenuto il match (definita in binario), specificando anche l'offset corrispondente (codificato invece come one-hot) portando a una codifica con la seguente struttura: WZ_BIT (1 bit) & WZ_NUM (3 bit) & WZ_OFFSET (4 bit).

1.3 Interfaccia del Componente

Il componente da descrivere ha un'interfaccia definita nel seguente modo:

```
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 TestBench;
- `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 manda l'indirizzo alla memoria;
- `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 memoria svolge un ruolo fondamentale nella realizzazione del metodo di codifica, in quanto non solo contiene i dati a cui deve accedere il componente implementato durante l'esecuzione, ma è anche il luogo dove scrivere l'indirizzo una volta codificato.

Infatti, nelle posizioni che vanno da 0 a 7, la memoria contiene gli indirizzi base delle corrispettive Working Zone, mentre nella posizione 8 il componente troverà l'indirizzo da codificare che, a processo terminato, dovrà essere scritto nella posizione 9.

Indirizzo base WZ 0	Indirizzo 0
Indirizzo base WZ 1	Indirizzo 1
Indirizzo base WZ 2	Indirizzo 2
Indirizzo base WZ 3	Indirizzo 3
Indirizzo base WZ 4	Indirizzo 4
Indirizzo base WZ 5	Indirizzo 5
Indirizzo base WZ 6	Indirizzo 6
Indirizzo base WZ 7	Indirizzo 7
Indirizzo da codificare	Indirizzo 8
Indirizzo codificato	Indirizzo 9

2. Architettura

Il modulo implementato partirà nell'elaborazione quando il segnale `START` verrà portato a 1.

Quindi, verrà chiesto alla memoria l'indirizzo in posizione 8, che è l'indirizzo da codificare, in modo da salvarlo, per poi procedere con il confronto di ogni elemento di ciascuna Working Zone con l'indirizzo salvato.

Nel caso in cui venga trovata una corrispondenza, si passerà dallo stato della WZ corrente allo stato finale, altrimenti a quest'ultimo si arriverà solo dopo aver testato ciascuna WZ. In entrambi i casi questa transizione di stato vedrà la scrittura del risultato nella posizione 9 della memoria.

Infine, lo stato finale vedrà il passaggio del segnale `DONE` da 0 a 1 e, solo dopo questa operazione, si provvederà ad abbassare il segnale `START`, per poi tornare in `IDLE` in modo tale da essere pronto a cominciare una nuova elaborazione.

2.1 Stati della macchina

La macchina costruita è costituita da 10 stati. Qui di seguito è fornita una breve descrizione per ciascuno di essi.

2.1.1 Stato `IDLE`

È lo stato iniziale, dove il componente aspetta che il segnale di `START` passi da 0 a 1 per avviare l'esecuzione.

Quando questo avviene, viene attivato il segnale di `ENABLE` per attivare lo scambio di informazioni tra il mio componente e la memoria.

Infine, dopo aver chiesto alla RAM l'indirizzo in posizione 8, avviene il passaggio nello stato `SALV`.

2.1.2 Stato `SALV`

Lo stato `SALV` è stato introdotto puramente per evitare un errore nel salvataggio dell'indirizzo, causato dalla quantità non sufficiente di tempo che intercorreva tra gli stati `IDLE` e `PRESET`.

2.1.3 Stato `PRESET`

In questo stato, il componente implementato procede con il salvataggio dell'indirizzo da codificare, in modo da poterlo poi confrontare con ogni elemento di ciascuna Working Zone.

Dopodiché, la macchina a stati provvederà al passaggio allo stato `C_WZ0`, per iniziare i confronti.

2.1.4 Stato `C_WZ0`

È lo stato dove iniziano tutti i confronti.

Se l'indirizzo salvato è uguale al dato che arriva dalla memoria, vuol dire che l'indirizzo da codificare è la base della WZ presa in considerazione (in questo caso la numero 0).

In caso contrario si testano anche, uno per uno, i tre elementi successivi, dove, se si trova il confronto, si passa allo stato `FINE`, scrivendo in memoria il risultato in base all'indirizzo che ha causato la corrispondenza. Altrimenti il prossimo stato sarà `C_WZ1`.

2.1.5 Stato `C_WZ1/C_WZ2/C_WZ3/C_WZ4/C_WZ5/C_WZ6`

Agiscono analogamente allo stato `C_WZ0`, in base alla corrispondente Working Zone.

2.1.6 Stato `C_WZ7`

In questo stato si testa la presenza dell'indirizzo da codificare nell'ultima WZ.

Se il confronto va a buon fine, il processo è identico a quello dei precedenti stati implementati per cercare la corrispondenza.

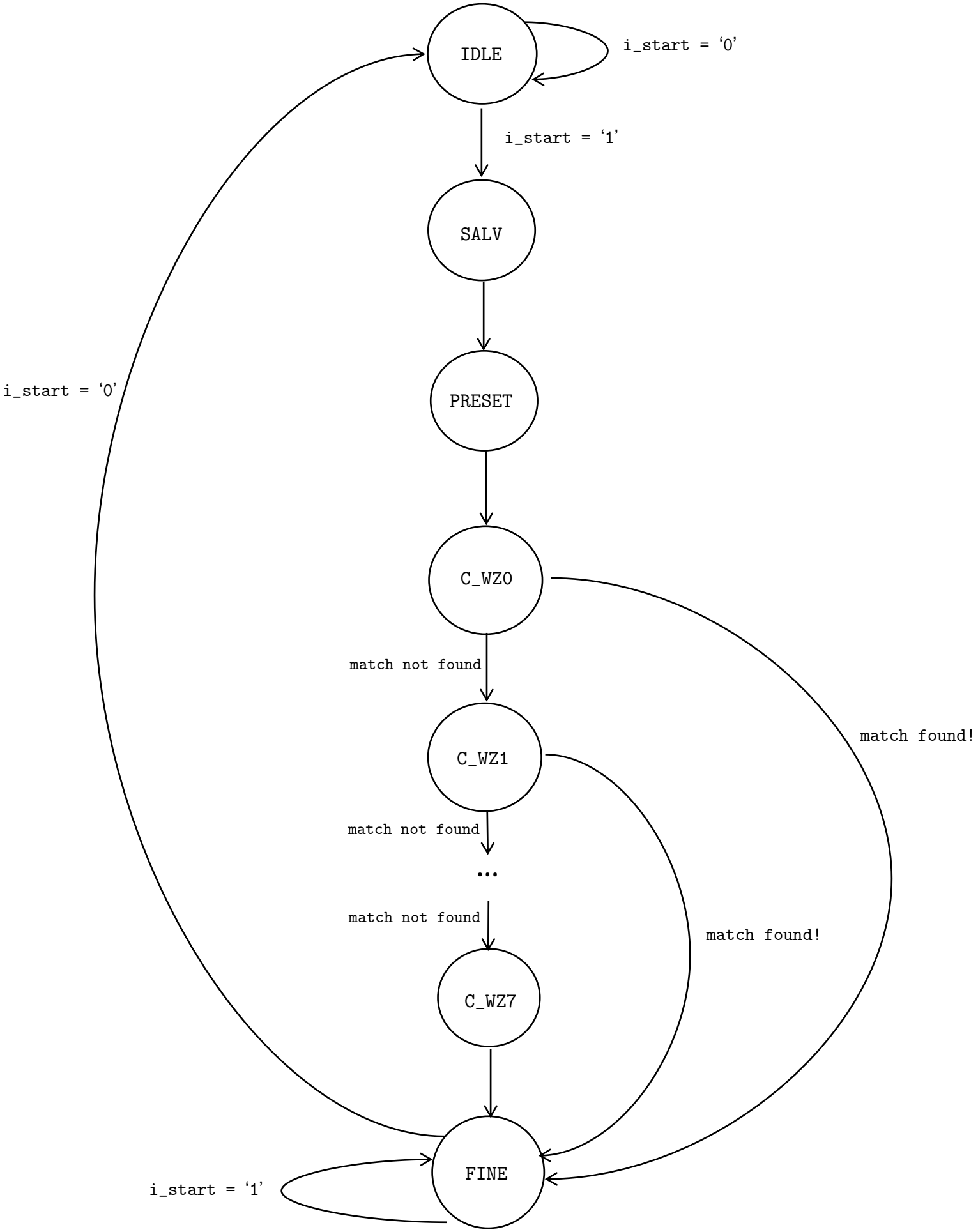
In caso contrario invece, si passerà ugualmente allo stato `FINE`, ma scrivendo in memoria il risultato non come appartenente a una Working Zone ma nel formato WZ_BIT & ADDR, con WZ_BIT = 0, a significare che l'indirizzo da codificare non è presente in nessuna WZ.

2.1.7 Stato `FINE`

Questo è lo stato finale. Qui il modulo implementato si aspetterà il passaggio del segnale `DONE` da 0 a 1 e si attenderà l'abbassamento del segnale `START`.

Una volta avvenuto quest'ultimo, si procederà con il ritorno in `IDLE` per mettersi nella condizione di iniziare, se necessario, una nuova elaborazione.

2.2 Rappresentazione grafica della macchina a stati



3. Risultati sperimentali

In modo da verificare il corretto funzionamento del componente implementato, oltre ai testbench di esempio forniti dal personale docente, ho deciso di implementare altri casi di test, affinché il codice scritto venisse collaudato in diverse ulteriori situazioni, comprendenti, in particolar modo, alcuni casi limite.

Di seguito è fornita una breve descrizione dei test implementati, associata, nei casi più rilevanti, allo screenshot della waveform, in modo da mostrare l'andamento dei segnali durante la simulazione.

3.1 Cambio delle Working Zone dopo un segnale di RESET

Dal momento che, successivamente a un segnale di RESET, gli indirizzi base delle WZ possono cambiare, è bene testare questo caso.

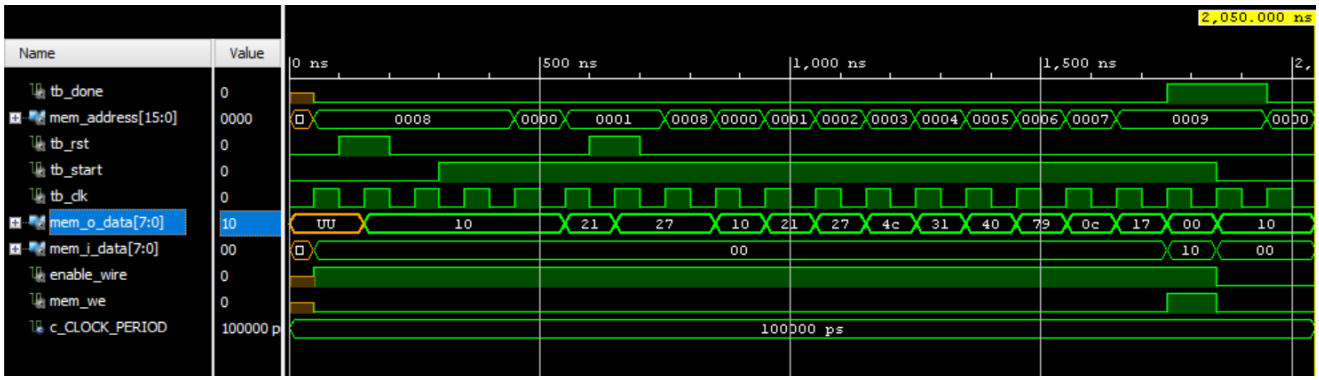
È importante però far notare che nella versione del componente da me implementata, ciò non creerà nessun problema.

Infatti, ho deciso di interrogare sistematicamente la RAM in modo da ottenere gli indirizzi base delle WZ durante ciascuna esecuzione: questo porterà il componente a non distinguere in un caso a parte l'eventuale presenza di un segnale di RESET al termine di un'elaborazione e quindi, un eventuale cambiamento degli indirizzi base delle WZ.

3.2 Segnale asincrono di RESET

È bene gestire correttamente l'arrivo di un segnale asincrono di RESET, facendo ricominciare al componente l'elaborazione dallo stato iniziale.

Nello screenshot è possibile osservare la waveform del test che prevede un segnale asincrono di RESET dopo 3 cicli di clock.

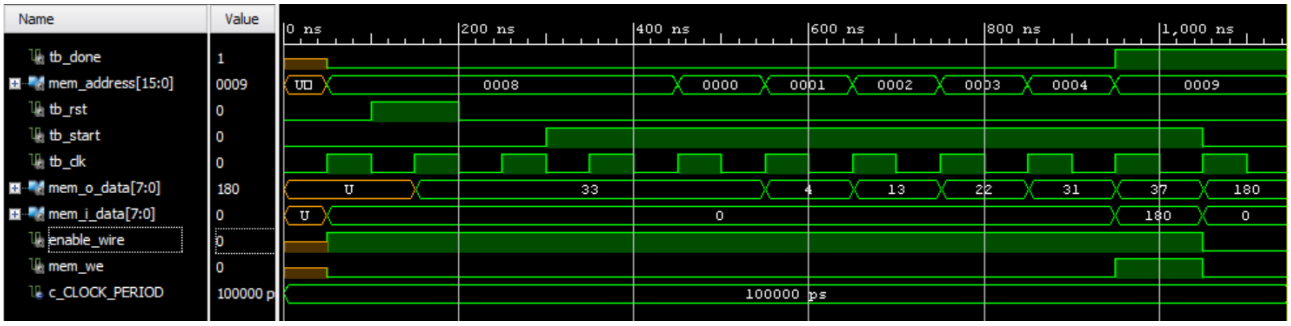


3.3 Arrivo del segnale START durante l'elaborazione

Un problema derivante dalla presenza di questo caso limite è evitato facendo in modo che il componente controlli l'arrivo del segnale di START solamente nello stato iniziale.

Perciò, nell'eventualità, si evita che venga ricominciata l'elaborazione a causa del nuovo segnale di START.

La waveform è tratta da un test in cui il segnale di START viene portato a 1 dopo 2 e 4 cicli di clock, oltre ovviamente quando serve da trigger per l'inizio dell'elaborazione.



3.4 Test multipli in sequenza

Questo caso è risolto permettendo al componente, nel momento in cui si trova nello stato finale, di tornare nello stato iniziale, dove aspetterà l'arrivo di un nuovo segnale di START, per cominciare una nuova esecuzione.

4. Testbench utilizzate

In questo paragrafo ho raccolto i casi di test forniti al componente sintetizzato in modo da verificarne la correttezza. Sono associate ad alcuni casi limite descritti nel paragrafo precedente.

4.1. Singolo START (esecuzione standard)

```
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 c_CLOCK_PERIOD;
wait until tb_done = '1';
wait for c_CLOCK_PERIOD;
tb_start <= '0';
wait until tb_done = '0';
```

4.2 Segnale asincrono di RESET (dopo 3 cicli di clock)

```
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 c_CLOCK_PERIOD;
wait for c_CLOCK_PERIOD;
wait for c_CLOCK_PERIOD;
tb_rst <= '1';
wait for c_CLOCK_PERIOD;
tb_rst <= '0';
wait for c_CLOCK_PERIOD;
wait until tb_done = '1';
wait for c_CLOCK_PERIOD;
tb_start <= '0';
wait until tb_done = '0';
wait for c_CLOCK_PERIOD;
```

4.3 Arrivo del segnale START durante l'elaborazione (dopo 2 e 4 cicli di clock)

```
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 c_CLOCK_PERIOD;
wait for c_CLOCK_PERIOD;
tb_start <= '1';
wait for c_CLOCK_PERIOD;
wait for c_CLOCK_PERIOD;
tb_start <= '1';
wait until tb_done = '1';
wait for c_CLOCK_PERIOD;
tb_start <= '0';
wait until tb_done = '0';
```

5. Conclusioni

Il componente sintetizzato supera correttamente tutti i test specificati nelle due simulazioni richieste: *Behavioral* e *Post-Synthesis Functional*.

Il mio approccio all'implementazione ha dato particolare importanza alla linearità del codice: infatti, leggendolo, è semplice comprendere le scelte effettuate, distinguendo con facilità la sezione dedita al confronto con le Working Zone da quella iniziale, prevista per l'inizializzazione della struttura, o ancora da quella finale, che regola, invece, l'ultima parte dell'elaborazione, fino al ritorno nello stato di START.