Progetto di Reti Logiche

Working-Zone encoding system

Prof Gianluca Palermo

Anno Accademico 2019/2020

Stefano Vanerio (Codice Persona 10583521 – Matricola 890404)

Mark Federico Zampedroni (Codice Persona 10608097 – Matricola 888340)

Indice

1 Introduzione . . . 3
- 1.1 Obiettivo del progetto . . . 3
- 1.2 Specifica generale . . . 3
  - 1.2.1 Working Zone . . . 3
  - 1.2.2 Caso in esame . . . 3
- 1.3 Descrizione utilizzo memoria . . . 4
- 1.4 Esempio . . . 5

2 Architettura . . . 6
- 2.1 Interfaccia del componente . . . 6
- 2.2 Architettura interna . . . 7
  - 2.2.1 Processo di caricamento . . . 7
  - 2.2.2 Processo Macchina a Stati . . . 7
    - 2.2.2.1 Stato IDLE . . . 8
    - 2.2.2.2 Stato CHECK . . . 8
    - 2.2.2.3 Stato DONE . . . 8

3 Risultati sperimentali . . . 9
- 3.1 Sintesi . . . 9
- 3.2 Simulazioni . . . 9
- 3.3 Test effettuati . . . 10
  - 3.3.1 Test sui casi limite . . . 10
  - 3.3.2 Test sui segnali . . . 11
  - 3.3.3 Test sui casi particolari . . . 12
  - 3.3.4 Test generati casualmente . . . 12

4 Conclusioni . . . 12

1 Introduzione

1.1 Obiettivo del progetto

Lo scopo del progetto è descrivere in VHDL una componente hardware che implementi il metodo di codifica *Working Zone*. Originariamente pensato per il *Bus Indirizzi*, è un metodo per la codifica di indirizzi basato sulla presenza di intervalli di valori nelle aree di memoria, chiamati Working Zone (WZ). Il componente userà l'indirizzo delle WZ e quello da codificare per elaborare il risultato che sarà poi trasmesso in output.

1.2 Specifica generale

1.2.1 Working Zone

Si progetti un componente per la codifica di un indirizzo nel caso appartenga a determinati range. Ognuno di questi range, denominato Working Zone, è definito come un intervallo di indirizzi di dimensione fissa (`Dwz`) che parte da un indirizzo base. Solitamente, all'interno dello schema di codifica esistono molteplici WZ (`Nwz`), le quali sono complete e non si sovrappongono.

Il *processo di encoding* per un indirizzo `ADDR` avviene in uno dei due seguenti modi:

Nel caso `ADDR` non appartenga a nessuna WZ, esso viene trasmesso con la sola aggiunta di un bit (`WZ_BIT`) nella posizione più significativa e posto a 0.

Nel caso `ADDR` appartenga all'intervallo di una delle WZ, il `WZ_BIT` viene posto ad 1 ed i rimanenti bit dell'indirizzo codificato rappresentano due parti:

- Il numero della WZ al quale `ADDR` appartiene, `WZ_NUM`, riportato in codifica binaria.
- L'offset rispetto all'indirizzo di base della WZ (`WZ_OFFSET`), indicato tramite codifica one-hot. (Nella quale è posto a 1 solamente il bit in posizione corrispondente al valore decimale che indica la distanza dall'indirizzo base, mentre gli altri sono posti a 0).

1.2.2 Caso in esame

Gli indirizzi, contestualmente al progetto, sono ad 8 bit; a causa del `WZ_BIT` i valori codificabili correttamente risultano da 00000000 a 01111111, rispettivamente da 0 a 127 in decimale, rendendo solo 7 bit realmente significativi.

Lo schema di codifica ha `Nwz` $= 8$ e `Dwz` $= 4$, con le WZ caratterizzate sempre dall'indirizzo di base, anch'esso ad 8 bit. Queste direttive per `Nwz` e `Dwz` portano ad avere `WZ_NUM` di 3 bit e `WZ_OFFSET` di 4 bit.

1.3 Descrizione utilizzo memoria

Il componente progettato, per ricevere i valori dell'indirizzo da codificare e degli indirizzi base delle WZ, comunica, attraverso appositi segnali, con una memoria RAM. È necessario quindi che il componente gestisca sia la parte di lettura da memoria, per poter ricevere gli indirizzi base delle diverse Working Zone, sia la parte di scrittura, in quanto una volta che l'indirizzo viene codificato dev'essere inserito nella memoria stessa all'indirizzo dedicato.

La RAM usata è già implementata e viene fornita con in memoria i valori necessari. I suoi indirizzi sono a 16 bit, quindi dispone dell'accesso a 65535 parole, ognuna di 8 bit.

Le WZ e l'indirizzo da codificare saranno salvati sempre nelle stesse posizioni, le restanti parole sono inizializzate a 0.

0	Indirizzo **prima** WZ
1	Indirizzo **seconda** WZ
2	Indirizzo **terza** WZ
3	Indirizzo **quarta** WZ
4	Indirizzo **quinta** WZ
5	Indirizzo **sesta** WZ
6	Indirizzo **settima** WZ
7	Indirizzo **ottava** WZ
8	Indirizzo **da codificare**
9	Indirizzo **codificato**

Figura 1: Indirizzi memorizzati nella memoria RAM

Facendo riferimento alla *Figura 1* il contenuto della memoria RAM è il seguente:

- Dalla posizione 0 alla posizione 7 sono inseriti gli **indirizzi base** delle Working Zone.
- La posizione numero 8 contiene l'**indirizzo da codificare**.
- Nella posizione numero 9 invece dev'essere scritto l'**indirizzo codificato**.

Tutte le altre posizioni non sono rilevanti e nella RAM fornita rimarranno sempre con il valore di default.

1.4 Esempio

0	0000 0000	(0)
1	0000 1111	(15)
2	0101 0111	(87)
3	0000 0100	(4)
4	0011 0000	(48)
5	0100 0100	(68)
6	0111 0111	(119)
7	0011 1111	(63)
8	0000 0111	(7)
9	0000 0000	(0)

Figura 2: Esempio di indirizzi inizializzati nella RAM

In questo esempio alla posizione 8 viene fornito come indirizzo da codificare `ADDR` il valore 00000111, in decimale corrispondente a 7. Una volta confrontato con i valori numerici delle WZ, contenuti all'interno delle posizioni di memoria 0-7, si riscontra di essere nel primo caso del *processo di encoding* (descritto nel paragrafo 1.2.1). L'indirizzo codificato risulta quindi:

(1)	(2)	(3)
1	011	1000

Figura 3: Indirizzo codificato

1. `WZ_BIT` viene posto a 1.
2. `WZ_NUM` indica la posizione in cui è salvata la WZ contenente `ADDR`, in questo caso la terza.
3. `WZ_OFFSET` in codifica one-hot che è l'offset dall'indirizzo base della WZ.

L'indirizzo codificato viene quindi scritto nella posizione 9 della memoria RAM.

Nell'eventualità in cui con le stesse WZ in *Figura 2* si dovesse codificare un indirizzo non contenuto in alcuna queste, come per esempio 00011000, in decimale 24, verrebbe salvato in memoria con la sola aggiunta del `WZ_BIT` a 0 in testa. In questo caso la codifica risulterebbe:

(1)	(2)
0	0001 1000

Figura 4: Indirizzo codificato

1. `WZ_BIT` che viene posto a 0.
2. `ADDR`, valore dell'indirizzo non codificato.

2 Architettura

2.1 Interfaccia del componente

Il componente da descrivere ha un'interfaccia definita nel seguente modo (in VHDL):

```
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;
```

I segnali con prefisso `i` sono ricevuti dalla RAM, quelli con `o` sono mandati alla RAM.
In particolare:

- **`i_clk`**: è il segnale di CLOCK inviato dal test bench;
- **`i_start`**: è il segnale di START inviato dal test bench, quando `i_start` è basso può cambiare `ADDR`, ma non le WZ;
- **`i_rst`**: è il segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START, con `i_rst` alto le WZ e `ADDR` possono cambiare;
- **`i_data`**: è il segnale (vettore ad 8 bit) che arriva dalla memoria RAM in seguito ad una richiesta di lettura;
- **`o_address`**: è il segnale (vettore a 16 bit) di uscita che manda l'indirizzo alla memoria 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 ad 8 bit) di uscita dal componente verso la memoria.

2.2 Architettura interna

Il componente progettato è diviso logicamente in tre processi. Il primo di questi, che chiameremo Loader, è sequenziale; si occupa delle richieste alla memoria RAM. I rimanenti due, uno sequenziale e uno combinatorio, invece, strutturano una macchina a stati.
Nei paragrafi seguenti vengono analizzati più nel dettaglio.

2.2.1 Processo di Caricamento (Loader)

Per riuscire ad eseguire la codifica con un costo temporale ridotto abbiamo affiancato alla macchina a stati un processo che funziona in parallelo. Tale processo è specializzato nella creazione delle richieste di lettura alla memoria RAM, e da qui la scelta di assegnarli il nome Loader.

Il Loader inizia con l'invio delle richieste sulla prima salita del ciclo di clock dopo che `i_start` viene alzato ad '1', e continua fino a quando l'FSM non fa richiesta di scrittura. Internamente funziona con un contatore che parte da 8 e decresce, richiedendo quindi prima l'indirizzo da codificare e poi gli indirizzi base delle Working Zone. Le risposte alle richieste di lettura vengono trasmesse dalla RAM su `i_data`, utilizzata esclusivamente dalla FSM.

2.2.2 Processo Macchina a Stati (FSM)

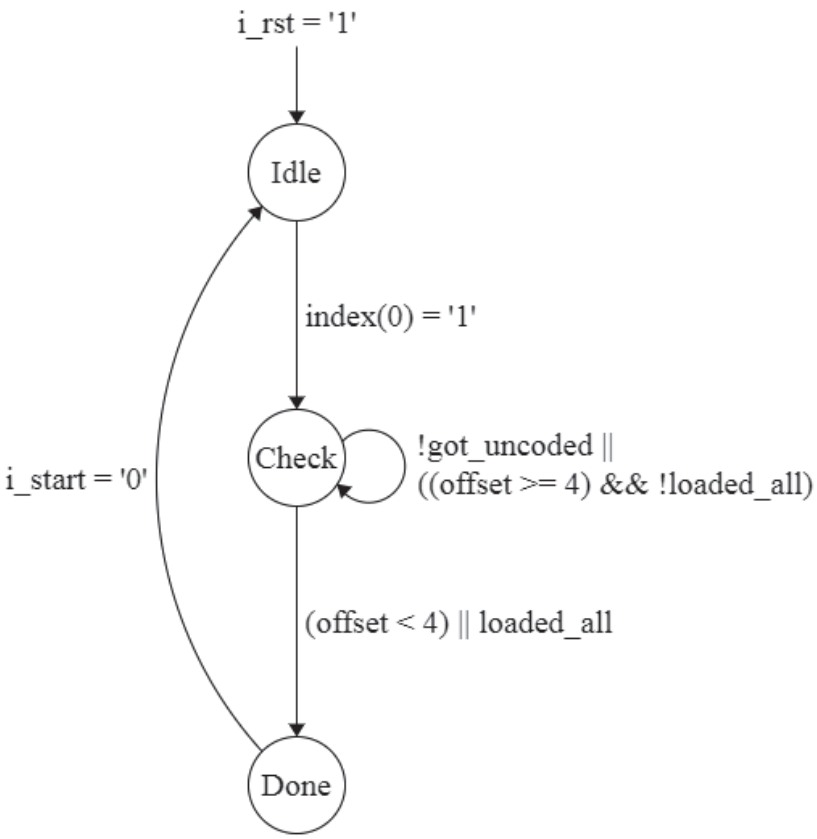


Figura 5: Diagramma stati FSM

L'FSM si occupa dei confronti con le WZ e della scrittura dell'indirizzo opportunamente codificato.
È implementata con l'utilizzo di due process, uno per la logica dei registri e uno per la logica degli stati. Questo per garantire che Vivado riconosca e sintetizzi una FSM.

Di seguito si riportano i segnali rappresentati sulle transizioni in *Figura 5* (diagramma stati):

- **`Index(3 downto 0)`** : indica a quale punto del caricamento delle Working Zone si è giunti; partendo dal valore 8 il Loader lo riduce di 1 ad ogni ciclo di clock.

- **`got_uncoded`** : registro flag utilizzato per indicare che è stato letto `ADDR`, salvato in `uncoded_addr`.

- **`loaded_all`** : registro flag utilizzato per indicare che tutte le Working Zone sono state caricate. Gestito dall'FSM in base al valore di `Index`.

- **`offset`** : variabile usata dall'FSM, non è un registro ma fa da "handler" per la distanza dall'indirizzo base della WZ letta all'indirizzo da codificare `ADDR`. Ad ogni ciclo di clock vale: `offset = 128 + (ADDR - i_data)`. Se `offset` < 4 allora `ADDR` è nel range della WZ.

2.2.2.1 Stato IDLE

Stato iniziale, in IDLE la macchina rimane in attesa che il bit meno significativo di `Index` diventi '1', quindi passa in CHECK. Questo avviene a seguito della prima richiesta alla RAM da parte del Loader, in cui `Index` da 1000 scende a 0111; Si può tornare in IDLE solamente in concomitanza di un segnale di reset o al termine dell'esecuzione.

2.2.2.2 Stato CHECK

Stato con l'implementazione logica più corposa, si occupa di:

1) Memorizzare l'indirizzo da codificare nell'opportuna variabile `uncoded_addr` e segnalare l'avvenuta lettura tramite la flag `got_uncoded`.
2) Eseguire la differenza tra `uncoded_addr` e `i_data` (WZ in entrata), che ad ogni ciclo di clock risulterà nel valore `offset`.
3) Scrivere in RAM alla posizione 9 l'indirizzo codificato e alzare il segnale `o_done`.

Riamane nello stato di CHECK fin quando non pone il segnale `o_done` ad '1', quindi passa allo stato DONE.

2.2.2.3 Stato DONE

Attende che il segnale `i_start` venga riabbassato per poter tornare allo stato IDLE. Con `i_start` basso può cambiare il contenuto della RAM come descritto nel paragrafo 2.1.

3 Risultati Sperimentali

Per la sintesi del codice abbiamo provato sia il software di sintesi di Vivado 2019.2, sia quello di Vivado 2016.4. In entrambi i casi è stata eseguita con successo, nonostante si possano notare delle piccole differenze sulla quantità di componenti generate, a favore della versione 2019.2.

Di seguito useremo i dati raccolti con quest'ultima.

3.1 Sintesi

Leggendo il report di sintesi confermiamo che Vivado ha correttamente riconosciuto la struttura tipica di una FSM, allocando 2 FF i cui valori combinati indicano lo stato. Si nota che gli unici avvisi sono info, non si riscontrano warning o errori.

Dalla sezione "*Report Cell Usage*" vediamo che in totale sono state sintetizzate 108 celle, compresi i buffer necessari su input ed output, esclusi i quali risultano 70 componenti non indotte dall'interfaccia.

Si riporta il *Design Runs* dopo la sintesi, in cui si legge il numero di LUT e FF sintetizzati.

LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
32	31	0.0	0	0	2/29/20, 12:24 AM	00:00:30	Vivado Synthesis Defaults (Vivado Synthesis 2019)

Figura 6: Design Runs. Vivado 2019.2

3.2 Simulazioni

Tutti i test che verranno poi elencati sono stati eseguiti sia in simulazione *Behavioral* che in *Post-Synthesis Functional* e *Post-Synthesis Timing*. Questo ci conferisce un buon margine di sicurezza nell'affermare che il componente funzioni correttamente sia in pre-sintesi che in post-sintesi.

Inseriti i *time constraints* (100ns di clk) Vivado ha generato il seguente *Report Timing Summary*:

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	94,146 ns	Worst Hold Slack (WHS):	0,139 ns	Worst Pulse Width Slack (WPWS):	49,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	40	Total Number of Endpoints:	40	Total Number of Endpoints:	32

All user specified timing constraints are met.

Figura 7: Report Timing Summary. Vivado 2019.2

Dal *WNS* ci si può fare una idea del tempo richiesto dal componente sintetizzato per propagare i segnali lungo il suo percorso più lungo.

Abbiamo anche individuato (in *Behavioral*) i due casi che portano alla simulazione più breve e più lunga. Trascorrono, dal momento in cui `i_start` sale al momento in cui si pone `o_done` alto:

- **450ns** nel caso più veloce, in cui `ADDR` sia nella prima WZ controllata.
- **1150ns** nel caso più lento, in cui `ADDR` non sia in alcuna WZ.

3.3 Test effettuati

Per accertarci che il componente funzioni correttamente abbiamo ideato una serie di *test bench* (tb) che portano ad avere casi di input particolari, tra cui i *corner case*. In base all'obiettivo i test si possono dividere in quattro categorie:

- *I test sui casi limite:* lo scopo è confermare che anche con input che portano ai corner case la codifica risulti corretta. Si riporta anche l'andamento dei segnali.

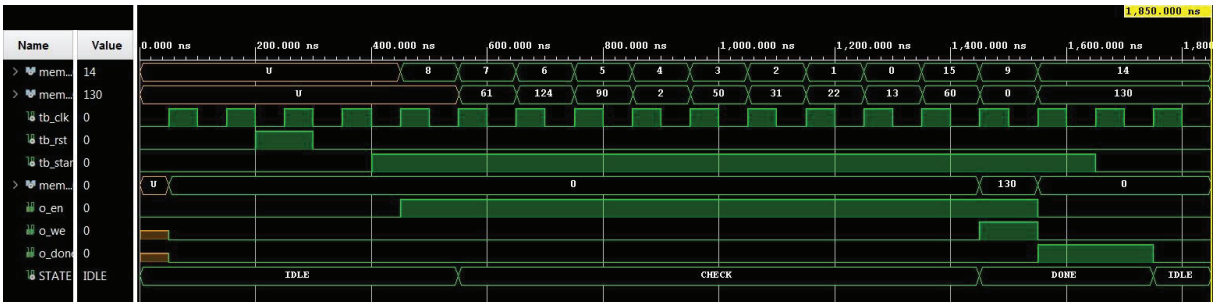
- *I test sui segnali:* mettono alla prova la corretta interazione del componente con i segnali in entrata. In alcuni si riporta anche l'andamento dei segnali.

- *I test sui casi particolari:* la RAM è inizializzata con valori che potrebbero portare in situazioni non ricorrenti e difficilmente generabili casualmente.

- *Test generati casualmente:* per garantire ulteriormente la robustezza del componente lo abbiamo sottoposto a 1.200.000 casi di test generati casualmente.

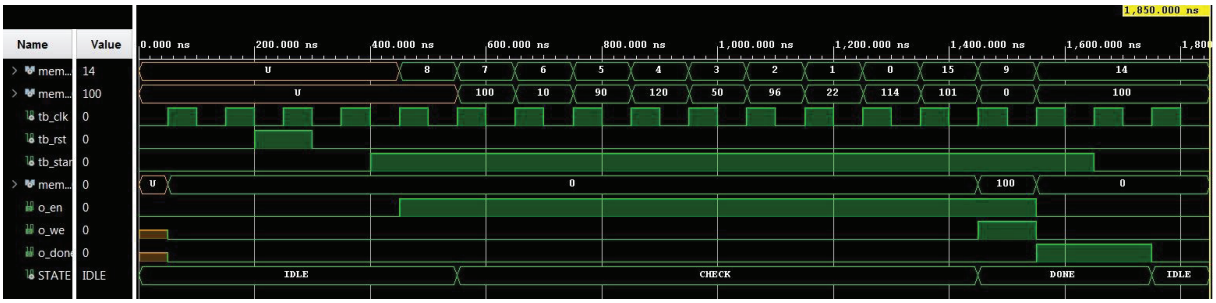
3.3.1 Test sui casi limite

1. **Test con `ADDR` appartenente all'intervallo di valori della WZ 0** (prima in ordine numerico ed ultima letta). Verifica la corretta codifica nel caso la WZ venga trovata nel momento in cui la flag `loaded_all` è portata ad '1'.



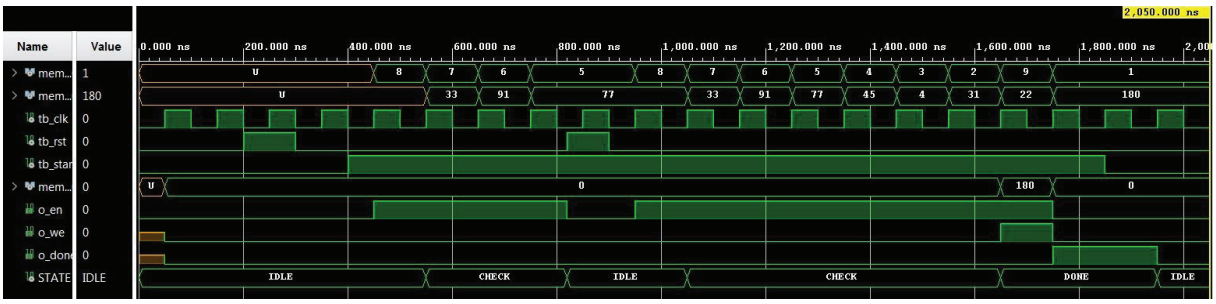
2. **Test con `ADDR` appartenente all'intervallo di valori della WZ 7** (ultima in ordine numerico e prima letta). Verifica la corretta codifica nel caso in cui si trovi il risultato il ciclo di clock dopo aver messo la flag `got_uncoded` ad '1'.

3. **Test con `ADDR` fuori dagli intervalli di tutte le WZ**. Verifica la corretta codifica nel caso non sia trovata alcuna WZ che lo contenga e la flag `loaded_all` valga '1'.

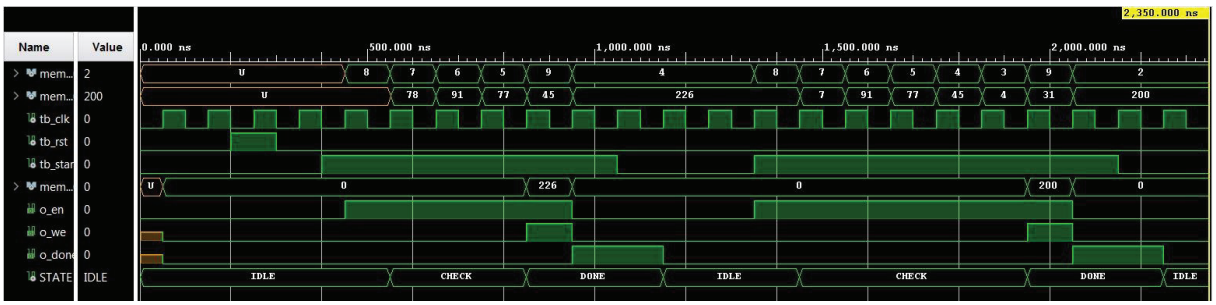


3.3.2 Test sui segnali

1. **Test con reset asincrono** in ritardo di 20ns rispetto al fronte di discesa del clock. Verifica che anche nel caso di reset in mezzo al ciclo di clock il componente funzioni correttamente e riporti ai valori di default tutti i segnali.



2. **Test con doppia esecuzione** e indirizzo da codificare cambiato dopo il primo `o_done` ad '1'. Verifica che quando viene portato `i_start` a '0' i segnali si resettino correttamente.



3. **Test con reset asincrono mezzo ciclo di clock dopo la richiesta di scrittura** da parte della FSM. Analogo al primo test sul reset ma più specifico. Serve per verificare che non venga erroneamente alzato il segnale di `o_done` al fronte di salita del clock successivo.

3.3.3 Test sui casi particolari

1. **Test con `ADDR` di valore 0** e WZ ordinate in modo che si leggano (in sequenza) gli indirizzi base 4, 124 e 0. Questo test serve a verificare che la variabile offset assuma i valori previsti, riconoscendo correttamente a che WZ appartiene `ADDR`.

2. **Test con `ADDR` di valore 127** e WZ ordinate in modo che si leggono (in sequenza) gli indirizzi base 0, 120 e 124. Intento analogo al test precedente.

3. **Test con `ADDR` fuori da tutte le WZ ma esattamente in mezzo a due intervalli coperti**. `ADDR` di valore 30, una WZ con indirizzo base 26 e una con indirizzo base 31. Verifica che il componente implementi la corretta `Dwz`.

3.3.4 Test generati casualmente

Utilizzando Python 2.7 e le funzioni di lettura/scrittura fornite dal package TEXTIO di VHDL abbiamo creato un test bench che può eseguire in una sola simulazione un numero potenzialmente illimitato di test. Il tb a fine simulazione crea un report che riporta su due blocchi di testo, rispettivamente, i test passati ed i test non passati, in modo da evidenziare per la fase di debugging eventuali anomalie o errori nella codifica.

Volendo coprire il maggior numero di casi possibili abbiamo eseguito 1.200.000 test, generati in set da 300 mila, in cui sono inclusi in modo misto: test con una sola richiesta di codifica, test con una seconda richiesta di codifica dopo la prima e `ADDR` cambiato (WZ uguali), test con reset asincrono. Tutti i test hanno avuto un esito positivo.

L'esecuzione dei 1.200.000 test ha impiegato pochi minuti in *Behavioral,* quasi mezz'ora in *Post-Synthesis Functional* e circa un'ora in *Post-Synthesis Timing*.

Il codice in Python ed il test bench si possono trovare al seguente link (share privato).

4 Conclusioni

Inizialmente avevamo preso in considerazione di dividere il progetto in tre moduli, Loader, FSM e una componente di memoria dove salvare le WZ che venivano lette. Questo avrebbe permesso, nel caso di richieste di codifica successive e senza reset, di abbassare drasticamente il tempo richiesto per ottenere i risultati. Siamo però giunti alla conclusione che sia una soluzione poco scalabile ed eccessivamente dispendiosa in termini di area. Inoltre, il ritardo causato da tutti i confronti (in un solo ciclo di clock) con WZ precedentemente salvate farebbe da collo di bottiglia, abbassando la massima (possibile) frequenza di funzionamento.

Abbiamo quindi optato per una soluzione più bilanciata: eliminare il modulo della memoria interna e unire i due vecchi moduli Loader e FSM, per agevolare l'utilizzo dei registri interni. Una volta scelta la struttura l'intento su cui ci siamo concentrati è stato quello di tenere al minimo il numero di cicli di clock tra il segnale di start e l'avvenuta codifica, senza perderne nell'attesa delle risposte dalla RAM o nella scrittura.