Prova Finale di Reti Logiche

Leonardo Gargani

A.A. 2019/2020

Indice

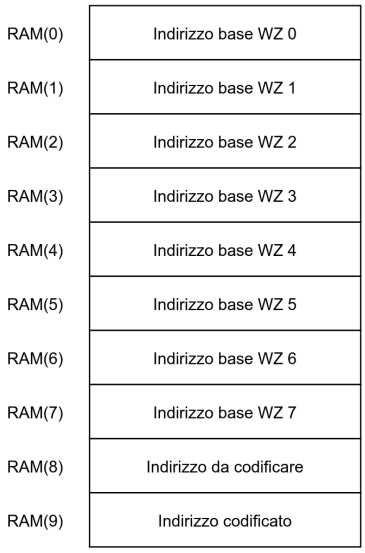
1 Introduzione

Lo scopo di questo progetto è stato la realizzazione di un componente hardware tramite una descrizione in linguaggio VHDL. In particolare, il funzionamento di questo componente si ispira al metodo di codifica a bassa dissipazione di potenza denominato "Working Zone".

1.1 Specifiche di progetto

Il componente da realizzare ha il compito di leggere un indirizzo da una memoria (RAM), codificarlo in modo opportuno e riscrivere nella stessa memoria (in un'altra posizione) il risultato della codifica.

Si introduce adesso il concetto di Working Zone (WZ): una WZ è semplicemente un intervallo di indirizzi di dimensione fissa, che parte da un cosiddetto indirizzo base.
All'interno di una memoria vi possono esistere molteplici WZ: è questo il nostro caso.



La RAM con cui dobbiamo comunicare ha le seguenti caratteristiche:

- gli indirizzi di memoria hanno una lunghezza di 16 bit;
- le parole salvate in memoria hanno una lunghezza di di 8 bit;
- le posizioni 0 - 7 contengono gli indirizzi base di 8 WZ;
- la posizione 8 contiene l'indirizzo da codificare, lungo 7 bit ma memorizzato su 8 bit (il primo bit è sempre a "0");
- la posizione 9 è quella in cui dobbiamo andare a scrivere il valore codificato.

L'algoritmo di codifica prevede due situazioni:

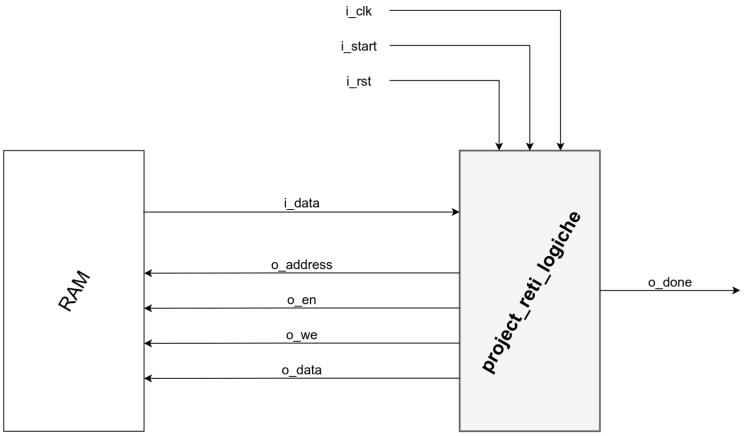
- *caso (a)* : l'indirizzo da codificare non appartiene a nessuna WZ;
- *caso (b)* : l'indirizzo da codificare appartiene ad una WZ.

In base a quale di questi due casi incontro, in posizione 9 della memoria dovrò quindi scrivere:

- *caso (a)* : un bit "1" e concatenare ad esso semplicemente i 7 bit dell'indirizzo;
- *caso (b)* : un bit "0" e concatenare ad esso prima 3 bit contenenti la codifica binaria della WZ di appartenenza, poi 4 bit contenenti la codifica one-hot dell'offset dell'indirizzo da codificare rispetto all'indirizzo base di tale WZ.

1.2 Interfaccia del componente

Il componente interagisce con la memoria e con eventuali componenti circostanti attraverso alcuni segnali in ingresso ed altri in uscita, come rappresentato in figura:



Vediamo il significato e la tipologia di ognuno di essi:

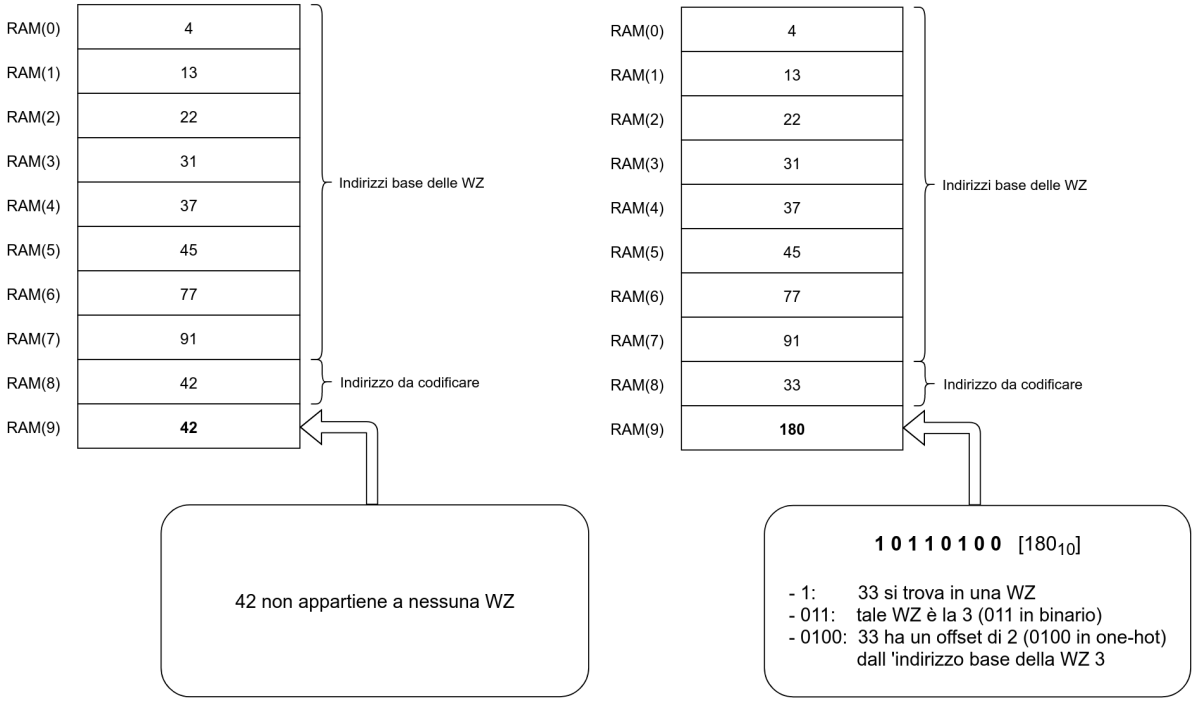
i_clock	Segnale (1 bit) di CLOCK in ingresso generato dal Test Bench.
i_start	Segnale (1 bit) di START in ingresso generato dal Test Bench.
i_rst	Segnale (1 bit) di RESET in ingresso che inizializza la macchina, pronta per ricevere il primo START.
i_data	Segnale (8 bit) di DATA in ingresso che arriva dalla memoria in seguito ad una richiesta di lettura.
o_address	Segnale (16 bit) di ADDRESS in uscita che manda alla memoria l'indirizzo a cui si vuole accedere in lettura o scrittura.
o_en	Segnale (1 bit) di ENABLE in uscita da mandare alla memoria per poter comunicare sia in lettura che in scrittura.
o_we	Segnale (1 bit) di WRITE ENABLE in uscita da mandare alla memoria per poterci scrivere (per poterci leggere deve invece essere a "0").
o_data	Segnale (8 bit) di DATA in uscita utilizzato per inviare alla memoria il valore che si vuole scrivere.
o_done	Segnale (1 bit) di DONE in uscita che comunica la fine dell'elaborazione e la scrittura del dato in memoria.

Le operazioni da eseguire devono rispettare le seguenti indicazioni:

- il modulo inizia l'elaborazione quando START viene alzato a "1" (e rimane a tale valore) ;
- gli accessi in lettura alla memoria devono avvenire alzando ENABLE e WRITE ENABLE a "1";
- l'accesso in scrittura alla memoria deve avvenire alzando ENABLE a "1" e abbassando WRITE ENABLE a "0";
- terminata la computazione e la scrittura in memoria, deve essere alzato DONE a "1";
- START viene riportato a "0", e il modulo si mette in attesa che questi venga rialzato a "1".

1.3 Esempi di funzionamento

Vengono riportate, a titolo di esempio, due possibili situazioni che è possibile incontrare.



Nella prima, a sinistra, l'indirizzo da codificare (42) non appartiene a nessuna WZ: esso non rientra infatti in alcun intervallo di ampiezza 4 a partire da uno degli indirizzi contenuti in RAM(0) - RAM(7).

Nella seconda, a destra, l'indirizzo da codificare (33) appartiene invece alla WZ 3: infatti, questa comprende gli indirizzi da 31 a 34.

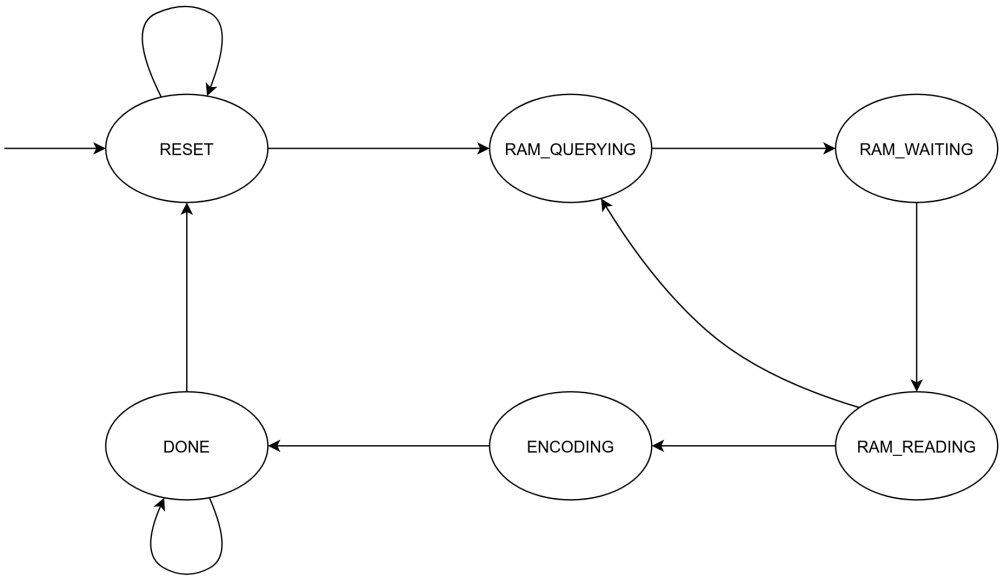
Si presti attenzione al fatto che in questi due esempi la configurazione della memoria resta sempre la stessa, ovvero gli indirizzi base delle WZ sono i soliti. Tuttavia ciò non è necessariamente vero osservando la RAM in due qualsiasi momenti: è infatti garantito che tali valori restino uguali fintanto che non venga ricevuto un segnale di RESET, dopodiché questi possono cambiare.

2 Architettura

Per prima cosa, la progettazione del componente è partita con la ricerca di una macchina a stati finiti, tale che non richiedesse un alto numero di stati, ma che allo stesso tempo risultasse semplice e di chiara interpretazione.

2.1 Macchina a stati finiti

In questo progetto si è individuato il miglior compromesso in un algoritmo interpretabile da una FSM (Finite-State Machine) a 6 stati strutturata come segue.



RESET	Stato di partenza in cui si attende il segnale di START, alla ricezione del quale la FSM si sposta su RAM_QUERYING.
RAM_QUERYING	Stato in cui viene inviato alla memoria l'indirizzo della cella a cui si intende accedere, dopodiché la FSM si sposta su RAM_WAITING.
RAM_WAITING	Stato in cui si attende che la memoria invii il contenuto dell'indirizzo di memoria richiesto in RAM_QUERYING; il prossimo stato è RAM_READING.
RAM_READING	Stato che, ricevendo il dato dalla memoria all'indirizzo richiesto: • se questo è il valore da codificare, lo memorizza in un registro e la FSM torna su RAM_QUERYING richiedendo l'indirizzo precedente; • se questo è l'indirizzo base di una WZ a cui appartiene l'indirizzo da codificare, allora la FSM si sposta su ENCODING; • se questo è l'indirizzo base di una WZ a cui non appartiene l'indirizzo da codificare, allora la FSM si sposta su ENCODING nel caso in cui la memoria sia stata già tutta letta, altrimenti torna su RAM_QUERYING richiedendo l'indirizzo precedente.
ENCODING	Stato in cui viene svolto l'encoding, distinguendo se è stata individuata una WZ a cui appartiene il dato da codificare oppure no, dopodiché viene inviato alla memoria il valore codificato e la FSM si sposta su RAM_WAITING.
DONE	Stato in cui la FSM rimane per almeno due cicli di clock, fintanto che non riceva un segnale di RESET che la riporta sullo stato RESET.

2.2 Implementazione VHDL

2.2.1 Processi

Per quanto riguarda la descrizione in VHDL, è stato scelto di utilizzare due `process`:

- *registers_handler*, che gestisce l'evoluzione degli stati settandoli al loro valore di deafult nel caso di un RESET, altrimenti al loro valore *next* (ogni segnale ha il proprio *next*);
- *combinatory_logic*, che gestisce la logica combinatoria svolgendo determinate operazioni in base allo stato in cui si trova la FSM.

Il motivo che ha spinto verso la scelta di due `process` è stato quello di voler separare la logica della macchina a stati da quella di aggiornamento dei segnali, in modo da migliorarne la manutenibilità ed un'eventuale espansione futura con l'aggiunta di nuovi stati.

2.2.2 Segnali

A supporto dei segnali già previsti da specifica, sono stati introdotti anche nuovi segnali interni (per comodità e brevità di spiegazione vengono omessi i corrispondenti segnali *next*):

- *state*, di tipo `state_type` (da me creato), che contiene lo stato corrente in cui si trova la FSM;
- *ram_position*, di tipo `integer`, che contiene la posizione dell'ultimo indirizzo di memoria che è stato acceduto;
- *wz_found*, di tipo `std_logic`, che serve per tenere traccia del fatto che sia stata trovata o meno una WZ di appartenenza al dato da codificare;
- *value_to_encode*, di tipo `std_logic_vector`, che contiene l'indirizzo da codificare;
- *oh_offset*, di tipo `std_logic_vector`, che contiene l'offset dell'indirizzo da codificare rispetto all'indirizzo base della WZ a cui appartiene;
- *done_raised*, di tipo `std_logic`, che serve per tenere traccia del fatto che sia stato alzato o meno il segnale di DONE nel ciclo di clock precedente.

In questo modo sono stati resi sempre disponibili valori che spesso era necessario avere a portata di mano, senza doverli ricalcolare ogni volta e utilizzandoli in stati differenti da quelli in cui sono stati ricevuti.

2.2.3 Costanti

Sono state inoltre introdotte anche le seguenti costanti:

- *NWZ* (valore: 8), ovvero il numero delle WZ presenti in memoria;
- *DWZ* (valore: 4), ovvero la dimensione di ciascuna WZ;
- *RAM_ADDR_BITS* (valore: 16), ovvero i bit di indirizzamento della memoria;
- *RAM_VAL_BITS* (valore: 8), ovvero la lunghezza in bit di ciascun dato contenuto in memoria.

Grazie all'uso di queste costanti il componente è stato reso facilmente scalabile. Esso può essere infatti immediatamente adattato a memorie più grandi, WZ diverse in numero e dimensione, dati di differente lunghezza, il tutto cambiando solo il valore della costante all'inizio del file, e non dovendo modificare altro all'interno del codice.

3 Risultati sperimentali

La FSM ottenuta è risultata correttamente funzionante utilizzando tutti i Test Bench generati, nelle simulazioni *Behavioral*, *Post-Synthesis Functional* e *Post-Synthesis Timing*.

3.1 Report di sintesi

Per quanto riguarda la sintesi, il componente è risultato sintetizzabile con 38 FF (filp-flop) e 38 LUT (lookup table):

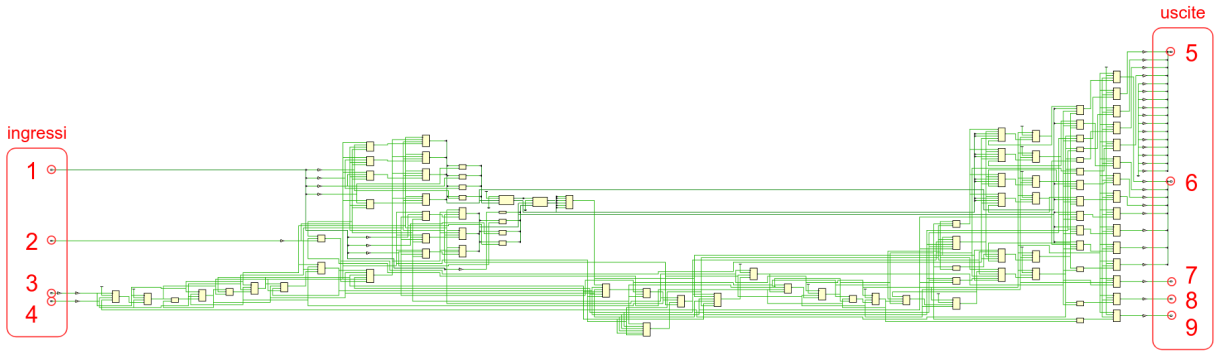
Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF
synth_1	constrs_1	synth_design Complete!								38	38
impl_1	constrs_1	route_design Complete!	NA	NA	NA	NA	NA	2.993	0	38	38

Tali valori corrispondono ad un utilizzo complessivo, sull'intera FPGA, di circa lo 0.03% dei LUT disponibili e lo 0.01% dei FF disponibili.

Si osservi come la scelta di non salvare gli indirizzi base delle WZ abbia permesso, come previsto, di avere percentuali molto basse in termini di area occupata. Ovviamente in questo caso si parla di numeri talmente piccoli che anche tale salvataggio non comporterebbe sostanziali differenze. Tuttavia, progettando il componente in ottica di renderlo facilmente scalabile ed espandibile, questo accorgimento permette di risparmiare area sull'FPGA che in un futuro potrebbe rivelarsi utile.

3.2 Schematic

Uno schema, generato da Vivado, della netlist post-sintesi mostra come effettivamente il componente riceva in ingresso 4 segnali e ne produca 5 in uscita:



1 : *i_data* 2 : *i_start* 3 : *i_clock* 4 : *i_rst*	5 : *o_address* 6 : *o_data* 7 : *o_en* 8 : *o_we* 9 : *o_done*

4 Simulazioni

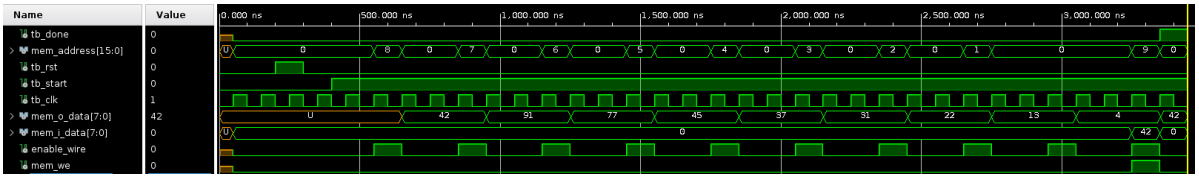
Per le simulazioni (sia in pre-sintesi che in post-sintesi) sono stati utilizzati, oltre a quelli forniti con la specifica, anche dei Test Bench scritti appositamente per sottoporre il componente a casi limite e verificarne il corretto funzionamento in ogni condizione.

4.1 Test Bench 0 (fornito con la specifica)

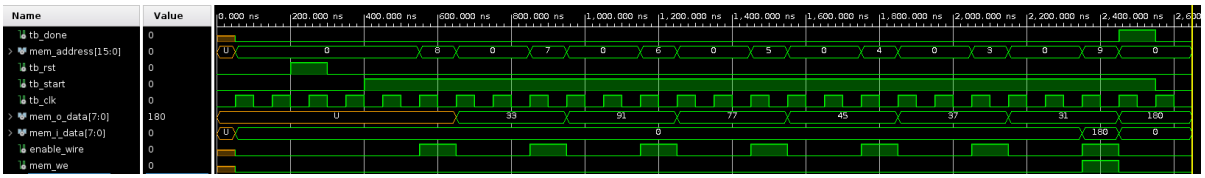
Questo Test Bench, fornito insieme alla specifica, testa semplicemente una singola computazione.

In particolare, abbiamo due casi:

- il dato da codificare appartiene ad una WZ...

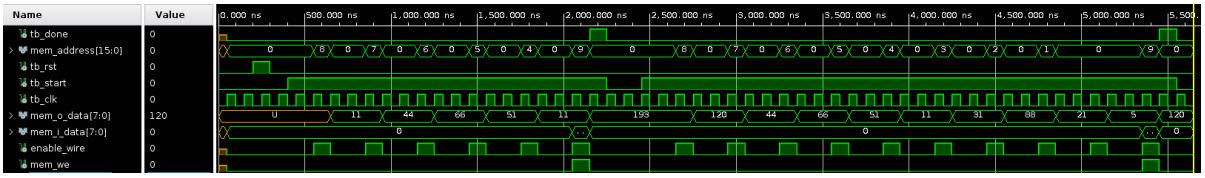


- il dato da codificare non appartiene ad alcuna WZ...



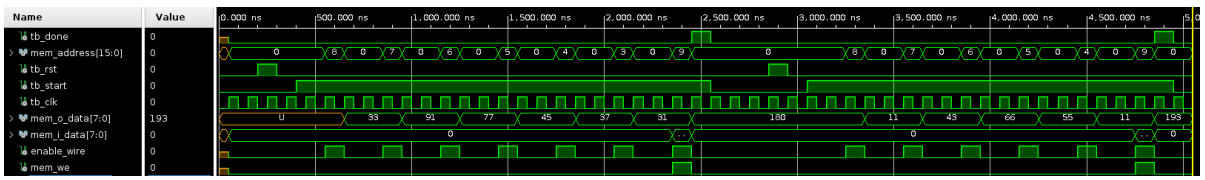
4.2 Test Bench 1 (esecuzioni successive)

Tramite questo Test Bench viene controllato il corretto funzionamento del componente nel caso di due computazioni successive (il segnale START viene alzato una seconda volta, dopo che DONE è stato alzato per un ciclo di clock).



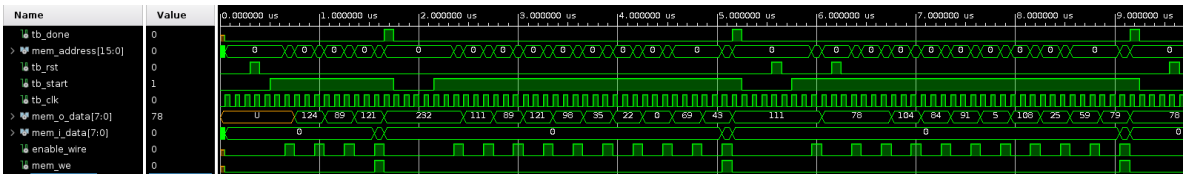
4.3 Test Bench 2 (RESET sincrono)

Tramite questo Test Bench viene controllato il corretto funzionamento del componente nel caso in cui venga ricevuto un segnale di RESET sincrono al termine di una prima computazione.



4.4 Test Bench 3 (esecuzioni successive e RESET asincrono)

Tramite questo Test Bench viene controllato il corretto funzionamento del componente nel caso in cui venga ricevuto un segnale di RESET, questa volta asincrono, durante una computazione. Tale segnale viene interpretato come previsto, e la FSM ricomincia l'interrogazione della memoria da capo.



4.5 Altre casistiche base testate

Oltre ai suddetti casi base, sono stati testati singolarmente anche i casi limite seguenti:

- indirizzo da codificare pari a 0 o a 127;
- indirizzo base della WZ di appartenza pari a 0 o a 124;
- offset dall'indirizzo base della WZ pari a 0 o a 3;
- posizione in memoria della WZ di appartenenza pari a 0 o a 7.

4.6 Test Bench finale

È stato infine creato un Test Bench che comprendesse tutte le casistiche precedenti e le sottoponesse al componente combinandole in ordine casuale e utilizzando dati sempre differenti. In questo modo è stato quindi possibile simulare quello che potrebbe essere un funzionamento reale del componente stressando anche tutti i casi più particolari che si possano potenzialmente incontrare.
È stato constatato un esito corretto della codifica sia in pre-sintesi che in post-sintesi.

5 Conclusione

L'obiettivo di questo progetto è stato quello di realizzare un componente hardware che seguisse le specifiche, ed è stato raggiunto tramite una FSM a 6 stati inizializzata sullo stato di RESET.
Per rendere più funzionale il componente, in particolare per favorire la sua scalabilità, si sono utilizzate delle costanti a livello di programmazione.
Inoltre, per la sintesi il componente ha richiesto un utilizzo di 38 FF e 38 LUT, privilegiando una bassa area occupata rispetto ad una più alta efficienza temporale in un'ottica di un'espansione futura.
Il corretto funzionamento è stato verificato sottoponendo il componente a una serie di Test Bench che sono stati superati tutti positivamente.