Prova finale (Reti Logiche)

Prof. Salice Fabio - Anno 2019-20

Hamza Haddaoui [10583761]

14 maggio 2020

Indice

1 Introduzione

1.1 Scopo del progetto

Lo scopo della prova, é la progettazione di un componente hardware in VHDL, che realizzi una simulazione di codifica a bassa dissipazione di potenza, denominata **WZE** (*Working Zone Encoding*).
Il componente si occupa di codificare gli indirizzi di input, in base ad un algoritmo che ne verifica l'appartenenza alle working-zone specificate.

Working zone sono intervalli di indirizzi di memoria, aventi dimensione fissa che rappresentano le aree di memoria più frequentemente accedute.
Tenendone traccia, è possibile adottare l'algoritmo di WZE, per ridurre la potenza dissipata in fase di fetch dei dati.

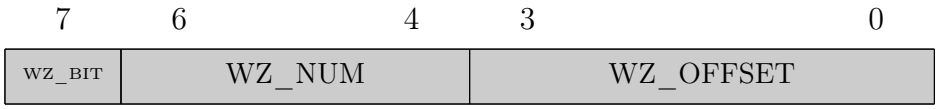
1.2 Specifiche

Il componente VHDL sintetizzato rappresenta una versione semplificata, ma fedele, del reale funzionamento di un working-zone encoder.
L'indirizzo da codificare è di 7 bit, mentre gli indirizzi in output sono di 8 bit.
Le working-zone presenti sono 8, ciascuna contenente 4 indirizzi di memoria (dunque 32 indirizzi candidati alla conversione).

La codifica differisce a seconda che l'indirizzo sia contenuto o meno all'interno di una della working-zone:

- Input $\notin$ working-zones :
  - WZ_BIT$\rightarrow = 0$ (encoding non attivo)
  - IN_ADDRESS $\rightarrow$ indirizzo ricevuto come input

- Input $\in$ working-zones :
  - WZ_BIT $\rightarrow = 1$ (Encoding attivo)
  - WZ_NUM $\rightarrow$ il numero di working-zone a cui appartiene l'indirizzo
  - WZ_OFFSET $\rightarrow$ offset rispetto ad indirizzo base working-zone (one-hot)



1.3 Interfaccia 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;
```

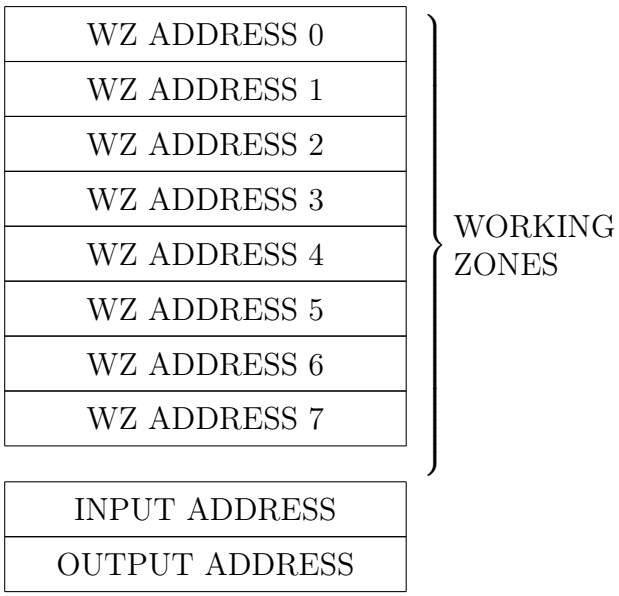
In particolare:

- `i_clk` é il segnale di CLOCK in ingresso generato dal test bench;
- `i_start` é il segnale di START generato dal test bench;
- `i_rst` é il segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START;
- `i_data` é il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
- `o_address` é il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- `o_done` é il segnale di uscita che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria;
- `o_en` é il segnale di ENABLE da dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- `o_we` é il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter scriverci. Per leggere da memoria esso deve essere 0;
- `o_data` é il segnale (vettore) di uscita dal componente verso la memoria.

1.4 Descrizione memoria

I dati, ciascuno di dimensione 8 bit, sono memorizzati in una memoria con indirizzamento al byte:

- Gli indirizzi da 0 al 7 sono usati per memorizzare le working-zone;
- L'indirizzo 8 è utilizzato per memorizzare l'input;
- L'indirizzo 9 è utilizzato per scrivere l'output codificato.



2 Architettura

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. La test bench risponde abbassando i_start e, successivamente, il componente riporta a 0 o_done; il componente ritorna nello stato IDLE, in attesa che il segnale i_start torni alto.
Il componente dispone inoltre di un segnale i_rst che permette la reinizializzazione del componente.
Nelle seguenti sezioni troviamo la descrizione della FSM che sintetizza il funzionamento del componente.

2.1 Stati della macchina

Il componente sintetizzato è stato descritto mediante FSM. La macchina costruita é composta da 7 stati. Di seguito é fornita una breve descrizione per ciascuno di questi.

2.1.1 `IDLE` state

Stato iniziale in cui si attende il segnale di `i_start`. In caso venga alzato il segnale `i_rst`, si torna in questo stato.

2.1.2 `FETCH_DATA` state

Stato in cui viene richiesto un dato alla RAM. I dati sono letti sequenzialmente a partire dall'indirizzo 0 all'indirizzo 8, e memorizzati in un buffer di memoria interno.

2.1.3 `WAIT_RAM` state

Stato in cui si attende la risposta dalla memoria in seguito alla richiesta di un dato.

2.1.4 `COMPUTE_OFFSET` state

Stato nel quale si perviene quando è terminata la fase di fetch dati dalla RAM. L'indirizzo di input viene sottratto all'indirizzo di working-zone, per calcolare l'offset.

2.1.5 `CHECK_ADDRESS` state

Viene verificato l'offset tra input e working-zone; nel caso in cui l'offset sia compreso tra 0 e 3 (vincolo di appartenenza alla working-zone), viene calcolato l'output. Altrimenti, in caso vi siano ancora working-zone da verificare si ritorna al `COMPUTE_OFFSET` state, oppure si procede alla fase di `WRITE_OUT` .

2.1.6 `WRITE_OUT` state

In questo stato viene scritto l'output sulla RAM, all'indirizzo 9.

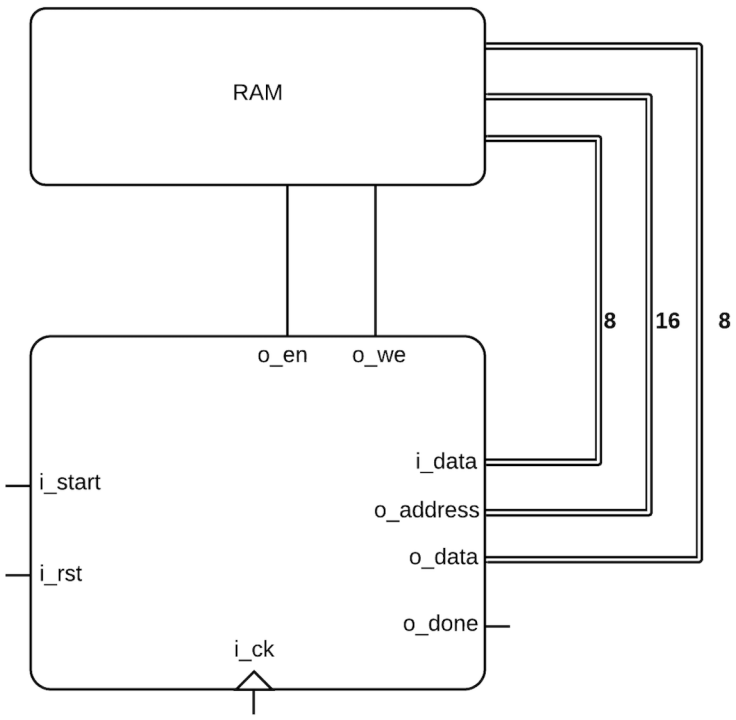
2.1.7 `DONE` state

Stato in cui si attende che la memoria abbassi `i_start` per poter resettare il bit `o_done` e tornare nello stato IDLE.

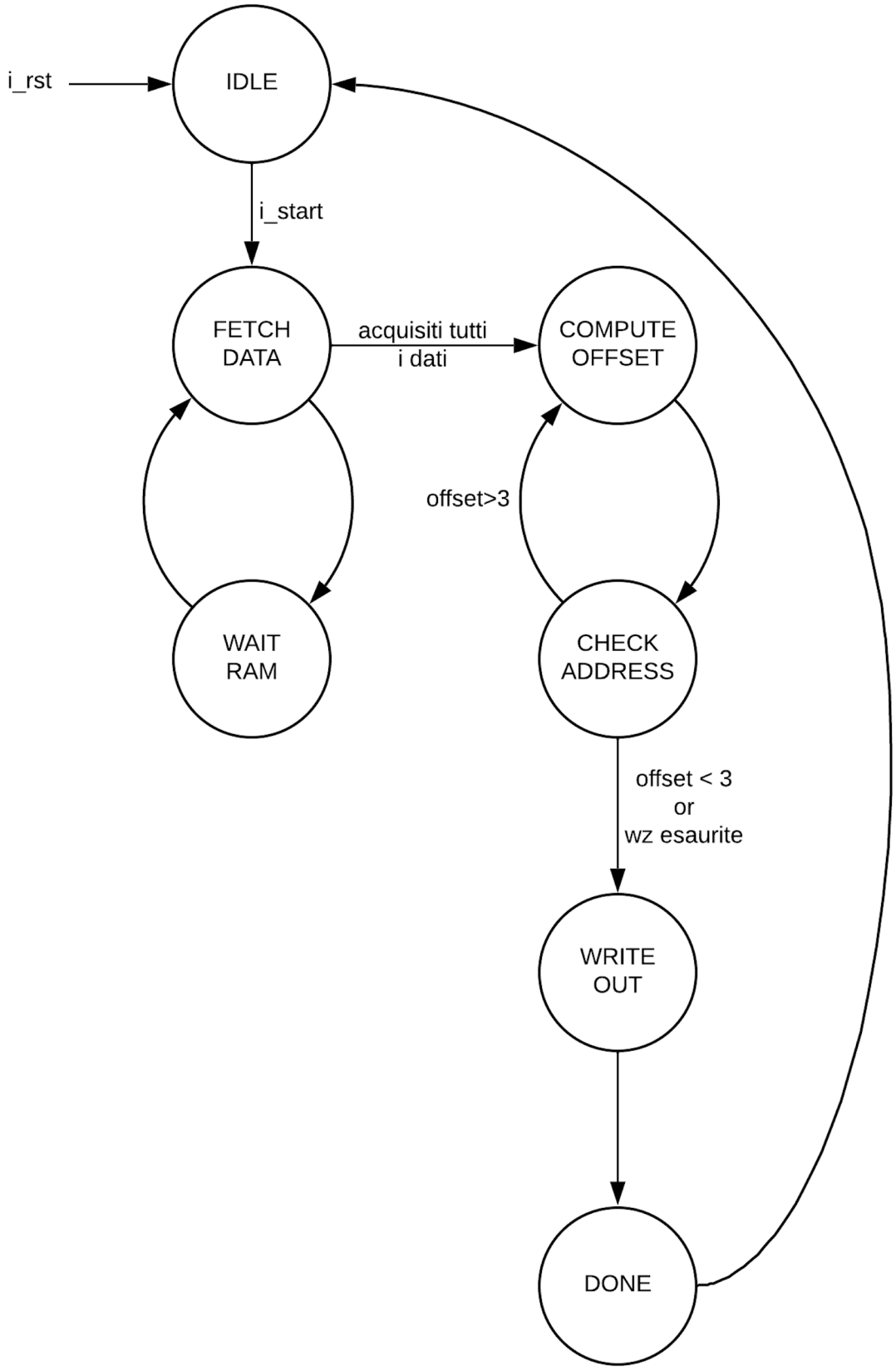
2.2 Scelte progettuali

La principale scelta progettuale effettuata é stata quella di descrivere il componente con un solo processo per rendere più semplice la comprensione e la manutenzione del codice.
Inoltre ho deciso di utilizzare un approccio che prevedesse la memorizzazione di tutte le informazioni necessarie, prima di passare alla computazione; tutti i dati sono salvati dal componente, in maniera che tra una conversione e l'altra, l'unico dato da ricaricare dalla RAM fosse l'indirizzo di input, considerando inviariate le working-zone.
Ulteriore precisazione, è quella che le working-zone vengono ricaricate dal componente solamente in presenza di un segnale di reset.

2.3 Diagramma componente



2.4 Diagramma degli stati



3 Risultati sperimentali

La sintesi del componente ha evidenziato un utilizzo oneroso di componenti, quali LUT e FF (173, 156), probabilmente a causa dell'alto numero di informazioni memorizzate.
Tuttavia, l'utilizzo di queste risorse non impatta assolutamente sulle prestazioni del componente, in quanto una maggior complessità spaziale, garantisce una minor complessità temporale.
Avendo a disposizione una FPGA intera, considerando che l'occupazione percentuale di LUT e FF è 0.13% e 0.06%, ho pensato fosse più sensato concentrarsi su una miglior robustezza del codice piuttosto che su ottimizzazioni di poco valore.

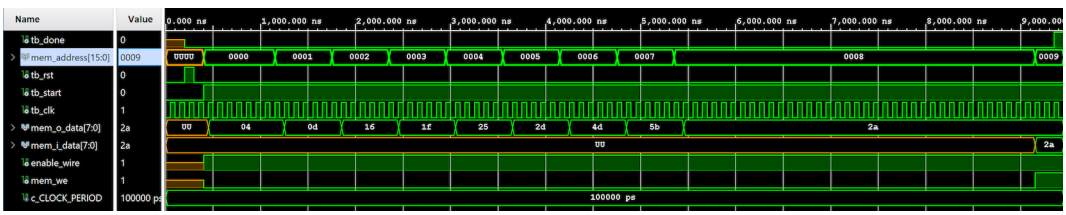
Il componente è testato e funzionante fino a 100MHz.

4 Simulazioni

Una volta creato il design, esso va anche testato. Per fare ciò ho creato dei test bench appositi al fine di testare il componente sia nei normali casi di utilizzo e sia nei casi limite. L'obiettivo é stato quello di testare tutte le funzionalità e le istruzione del codice. Qui sono riportati i test bench più significativi.

4.1 Test bench 1 (Indirizzo $\notin$ working-zone)

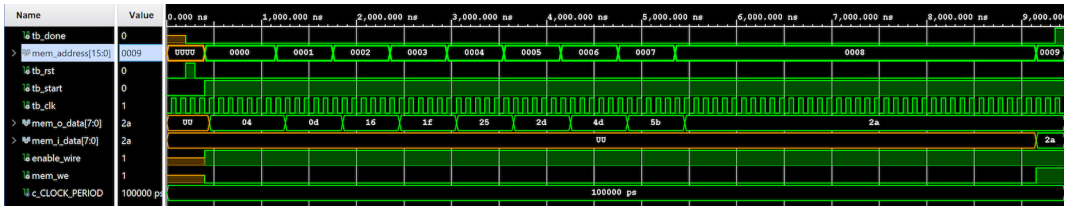
Indirizzo di memoria	Valore	Contenuto
0	4	WZ 0
1	13	WZ 1
2	22	WZ 2
3	31	WZ 3
4	37	WZ 4
5	45	WZ 5
6	77	WZ 6
7	91	WZ 7
8	42	Input address



Il test verifica che l'input non appartenga a nessuna working-zone.

4.2 Test bench 2 (Indirizzo ∈ working-zone)

Indirizzo di memoria	Valore	Contenuto
0	4	WZ 0
1	13	WZ 1
2	22	WZ 2
3	31	WZ 3
4	37	WZ 4
5	45	WZ 5
6	77	WZ 6
7	91	WZ 7
8	33	Input address



Il test verifica che l'input appartenga alla working zone 3. Il valore codificato in uscita è 1-011-0100.
Cambiando il valore di input, sono stati testati anche tutte le altre working-zone, a diversi offset (0-1-2-3).

4.3 Test bench 3 (Start / Multi-start)

Il test verifica che in caso di più segnali di start consecutivi, il componente ricarichi solamente l'indirizzo di input e processi il dato correttamente.

4.4 Test bench 4 (Reset / Multi-reset)

Il test verifica che in caso di reset, il componente abbandoni qualsiasi operazione e re-inizializzi tutto da zero. Viene inoltre verificato che in caso di stress sul reset, il componente esegua le operazioni correttamente.

4.5 Test bench 5 (Offset al limite)

Il test verifica che in caso di indirizzi in input, limitrofi alle working zone (offset = -1 oppure offset = 4) il componente si comporti correttamente

5 Conclusioni

Il progetto è funzionante sia in pre che in post sintesi.
Non sono state effettuate molte ottimizzazioni, se non quella sul numero degli stati, che risultano il minor numeri di stati possibili per effettuare la conversione degli indirizzi.
Inizialmente i process erano due, ma questo portava ad alcune instabilità interne che generavano problemi, dunque mi sono portato su un singolo process che risulta più stabile e meno soggetto ad errori.
Il componente è ottimizzato in modo che ogni lettura della RAM venga eseguita solo se strettamente necessaria (non viene letto ciò che non serve) e durante l'elaborazione venga sfruttata al massimo la RAM (a ogni ciclo di clock una lettura o una scrittura).