- PROVA FINALE -

PROGETTO RETI LOGICHE

Prof. Fabio Salice

A.A. 2019/2020

Gianmarco Naro

Matricola: 888331, Codice Persona: 10610374

Indice

1 Introduzione

1.1 Scopo del progetto

Progettare un componente hardware, realizzato in VHDL, che, preso in input un indirizzo, garantisca la trasmissione codificata di quest'ultimo, utilizzando un approccio che si basa sul metodo di codifica Working Zone.

Quest'ultimo, in sintesi, è un metodo utilizzato per trasformare il valore di un indirizzo quando questo viene trasmesso, se e solo se appartiene a certi intervalli stabili, denominati, per l'appunto, working zone.

1.2 Specifica

Dato in ingresso un indirizzo da codificare (ADDR) e gli 8 indirizzi base delle working zone (WZ), bisogna trasmettere l'indirizzo codificato in maniera opportuna. L'indirizzo da codificare è di 7 bit e definisce tutti gli indirizzi compresi tra 0 e 127 (inclusi), mentre le working zone stabilite sono 8 e hanno una dimensione fissata di 4 indirizzi, incluso quello base, ed ogni working zone è sempre completa, ovvero possiede sempre tutti e 4 gli indirizzi. Inoltre, le working zone possono essere adiacenti fra di loro, oppure essere adiacenti ai limiti della memoria, ma non possono intersecarsi fra di loro.

Di conseguenza, l'indirizzo codificato sarà formato da 8 bit e la sua struttura dipende dal fatto che l'indirizzo da codificare appartiene o meno ad una working zone. Per comporre la codifica verranno usati i seguenti campi:

- **WZ_BIT:** Specifica se l'indirizzo da codificare appartiene o meno ad una working zone. *[1 bit]*
- **WZ_NUM:** Specifica, in codifica binaria, a quale working zone appartiene l'indirizzo. *[3 bit]*
- **WZ_OFFSET:** Specifica, in codifica one hot, l'offset di ADDR rispetto all'indirizzo base della working zone corrispondente. *[4 bit]*

Si possono presentare due casi:

1. **L'indirizzo da trasmettere non appartiene a nessuna Working Zone**

   In questo caso viene utilizzato solo WZ_BIT, il quale assume valore '0' specificando, quindi, che ADDR non appartiene a nessuna working zone.

   L'indirizzo codificato si otterrà concatendando WZ_BIT e ADDR in questo ordine. Segue un esempio.

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

   OUTPUT: 42 (0-0101010)

2. **L'indirizzo da trasmettere appartiene ad una Working Zone**

   In questo caso WZ_BIT assume valore '1' specificando, quindi, che ADDR appartiene ad una working zone. Successivamente viene specificato in WZ_NUM il numero della working zone a cui ADDR appartiere e in WZ_OFFSET il valore dell'offset rispetto all'indirizzo base della working zone precedentemente selezionata.

   L'indirizzo codificato si otterrà concatenando WZ_BIT, WZ_NUM, WZ_OFFSET in quest'ordine. Segue un esempio.

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

   OUTPUT: 180 (1-011-0100)

Come possiamo notare dai precedenti esempi, tutti i dati, di dimensione 8 bit, sono memorizzati in una memoria con indirizzamento al Byte. In particolare, le posizioni in memoria da '0' a '7' sono usate per memorizzare gli indirizzi base delle 8 working zone, mentre la posizione '8' conterrà al suo interno l'indirizzo di memoria da codificare. Infine la posizione '9' verrà utilizzata per scrivere in memoria il valore dell'indirizzo codificato alla fine della computazione.

1.3 Interfaccia del componente

L'interfaccia del componente, come da specifica, è mostrata in figura [1].

```
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;
```

Figure 1: Interfaccia

Andando ad analizzare singolarmente:

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

2 Architettura

A livello implementativo, vista la complessità ridotta del problema, la scelta è ricaduta su una FSM con l'utilizzo di un singolo processo, questo per evitare problemi di inferring latch, per gestire in maniera ottimale la sensitivity list e per rendere tutto il processo sincrono al fronte di salita del clock.

La computazione comincia con la lettura delle working zone dalla memoria, le quali vengono salvate per evitare la loro successiva rilettura, e, successivamente, dopo che il segnale `i_start` viene asserito ad '1', comincia la lettura dell'indirizzo da codificare e la sua successiva codifica. Il segnale `i_start` avrà sempre lo stesso valore fino a quando il segnale `o_done` non verrà asserito anche'esso ad '1', ovvero alla fine della computazione. Il segnale `o_done` deve essere '1' fino a quando il segnale `i_start` non viene riportato a '0' e, inoltre, un nuovo segnale `i_start` non può essere riasserito fin tanto che `o_done` non è riportato a '0'.

In aggiunta, il componente possiede un segnale `i_rst`, il quale, se posto ad '1', inizializza la macchina a stati finiti, pronta a ricevere un nuovo segnale `i_start`, ricominciando la letture delle working zone.

2.1 Macchina a stati finiti

La macchina a stati finiti utilizzata è composta da 8 stati. Di seguito, per ognuno di essi, viene fornita una breve spiegazione del loro ruolo durante la computazione.

2.1.1 `READ ADDRESS` state

In questo stato viene assserito ad '1' il segnale di lettura `o_en` e con l'aiuto di un *counter*, che indica l'indirizzo di memoria da prelevare, viene estratto e salvato temporaneamente il contenuto di `o_address` e viene stabilito se quest'ultimo sia un indirizzo di una working zone o, contrariamente, l'indirizzo da codificare.

Quando è necessario leggere, ma soprattutto salvare degli indirizzi dalla memoria, lo stato successivo sarà `WAIT DATA`, ma nel caso in cui il *counter* asserisca che gli indirizzi da leggere dalla memoria siano terminati e `i_start` sia '1', verrà salvato l'indirizzo da codificare e lo stato successivo sarà `WZ CHECK`. Contrariamente, se `i_start` è '0' e la lettura delle working zone è terminata, si attende il segnale `i_start` a '1' per proseguire la computazione.

Inoltre, nel caso in cui venga posto il segnale `i_rst` a '1', a tutte le variabili e tutti i segnali verrà assegnato il loro valore iniziale in modo da poter ricominciare la computazione, partendo proprio da questo stato. In particolare, `o_done` e gi altri segnali di lettura/scrittura della memoria vengono asseriti a '0', così come tutte le variabili.

2.1.2 WAIT DATA state

In questo stato si attende che il contenuto di o_address venga caricato correttamente in i_data, per poi passare allo stato SAVE ADDRESS.

2.1.3 SAVE ADDRESS state

In questo stato viene prelevato il contento caricato precedentemente in i_data. L'indirizzo appena prelevato viene salvato in un array di vector dove i primi 8 vettori contengono gli indirizzi delle working zone, mentre l'ultimo vettore, posto in nona posizione, contiene l'indirizzo da codificare. Nel processo viene utilizzato un *index* per scorrere correttamente l'array in questione, per poter inserire o prelevare gli indirizzi.

Inoltre, prima di passare allo stato successivo, ovvero READ ADDRESS, vengono incrementati i valori di *counter* e di *index* in modo da poter progredire nella lettura degli indirizzi.

2.1.4 WZ CHECK state

In questo stato viene controllato se l'indirizzo da codificare appartiene o meno ad una working zone. In caso affermativo vengono calcolati e salvati i valori di WZ_NUM e WZ_OFFSET ed infine viene posto WZ_BIT a '1'. Al contrario, se l'indirizzo da codificare non appartiene a nessuna working zone, viene solamente posto WZ_BIT a '0'. Lo stato successivo sarà COMPOSE RESULT.

2.1.5 COMPOSE RESULT state

In questo stato, utilizzando i dati calcolati nello stato precedente, viene creato l'indirizzo codificato concatenando i valori corretti, per poi passare allo stato successivo, ovvero WRITE OUT.

2.1.6 WRITE OUT state

In questo stato viene asserito ad '1' o_we, in modo da poter inserire l'indirizzo codificato in o_data, per poi passare allo stato successivo, ovvero WAIT WRITING.

2.1.7 WAIT WRITING state

In questo stato si attende che l'indirizzo codificato venga scritto correttamente in memoria, per poi passare allo stato successivo, ovvero DONE.

2.1.8 DONE state

In questo stato viene asserito o_done ad '1', però, dal momento in cui i_start viene posto a '0', o_done verrà asserito nuovamente a '0' in modo da poter ricominciare la computazione, ritornando così allo stato READ ADDRESS. In particolare il counter assume il valore '8', in modo da poter imediatamente leggere nuovo un indirizzo da codificare.

Di seguito è riportata la tabella della codifica degli stati della macchina a stati finiti.

Stato	Codifica
READ ADDRESS	000
WAIT DATA	001
SAVE ADDRESS	010
WZ CHECK	011
COMPOSE RESULT	100
WRITE OUT	101
WAIT WRITING	110
DONE	111

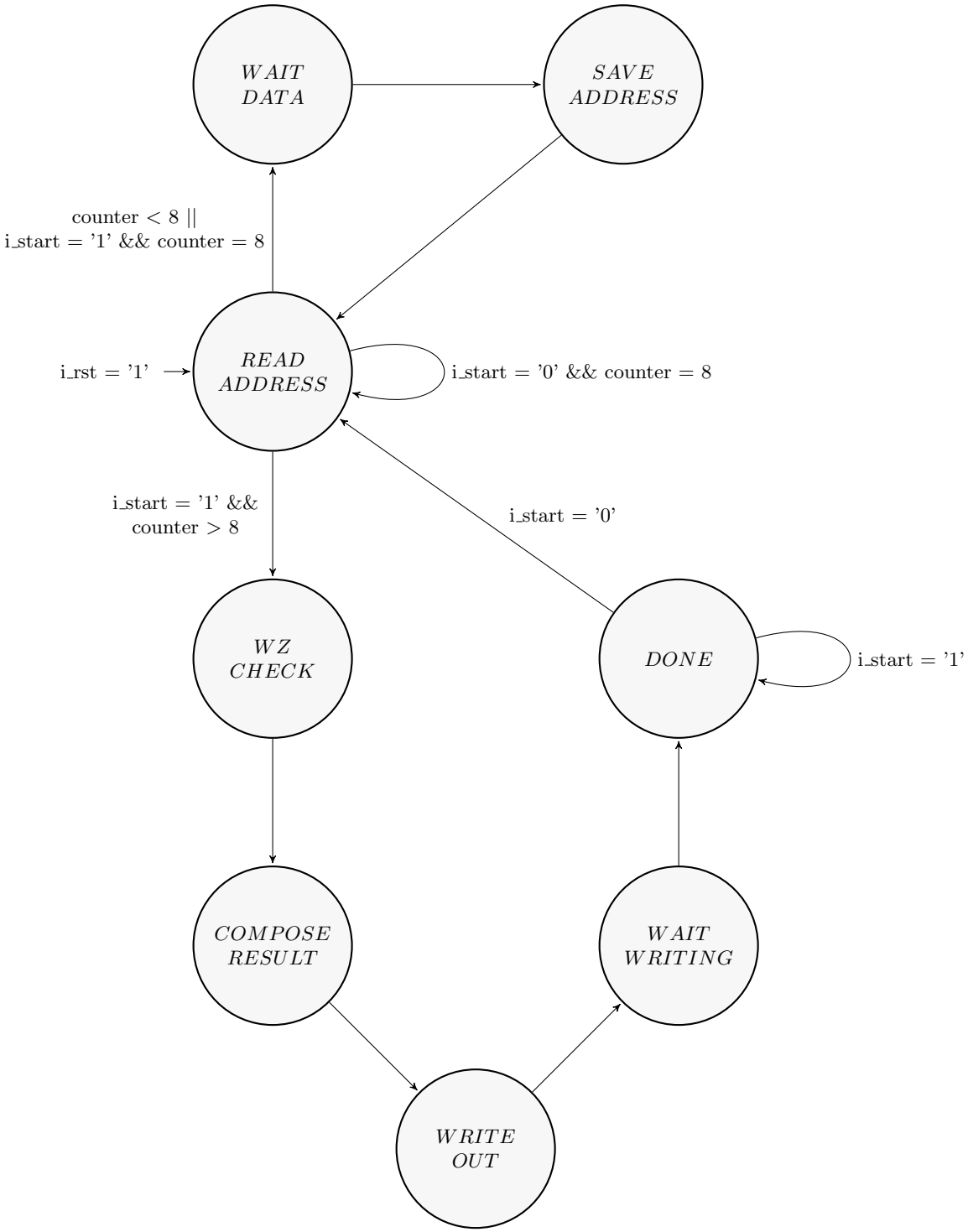
WAIT DATA
SAVE ADDRESS
counter < 8 ||
i_start = '1' && counter = 8
READ ADDRESS
i_rst = '1'
i_start = '0' && counter = 8
i_start = '1' &&
counter > 8
i_start = '0'
WZ CHECK
DONE
i_start = '1'
COMPOSE RESULT
WAIT WRITING
WRITE OUT

Figure 2: FSM

3 Risultati sperimentali

Per verificare il corretto funzionamento del componente, oltre al test bench d'esempio, sono stati svolti diversi altri test in modo da verificare la corretta copertura di tutti possibili i casi, tra cui i casi limite. Nel complesso, sono stati svolti due tipologie di test:

- **Test casuali**
- **Test specifici**

Per quanto riguarda i test casuali, sono stati generati dei test bench in maniera casuale, in modo da poter visionare, in generale, il corretto comportamento del componente.

Dopo aver appurato il corretto funzionamento del componente, sono stati generati dei test bench più specifici, atti a controllare i casi limite degni di nota. Infatti, questi test specifici, sono stati utilizzati per visionare soprattutto l'andamento della maggior parte dei segnali durante la computazione e per controllare se, effettivamente, quest'ultimi si comportassero in maniera consona.

Di seguito sono riportati i test mirati più significativi che sono stati utilizzati per il controllo dei casi limite.

Inoltre, durante la sintesi del componente, si presenta un warning che evidenzia come `i_data[7]` non sia mai utilizzato. Questo è corretto in quanto gli indirizzi da leggere siano di 7 bit, rappresentati quindi da un vettore che va da '0' a '6'.

I test bench che sono risultati fondamentali sono 4:

1. **Reset asincrono**

   Questo test si è rivelato importante in quanto è stato utilizzato per controllare come il componente reagisce ad una serie di reset asincroni e se l'insieme dei segnali si comporta in maniera corretta durante tutta la computazione.

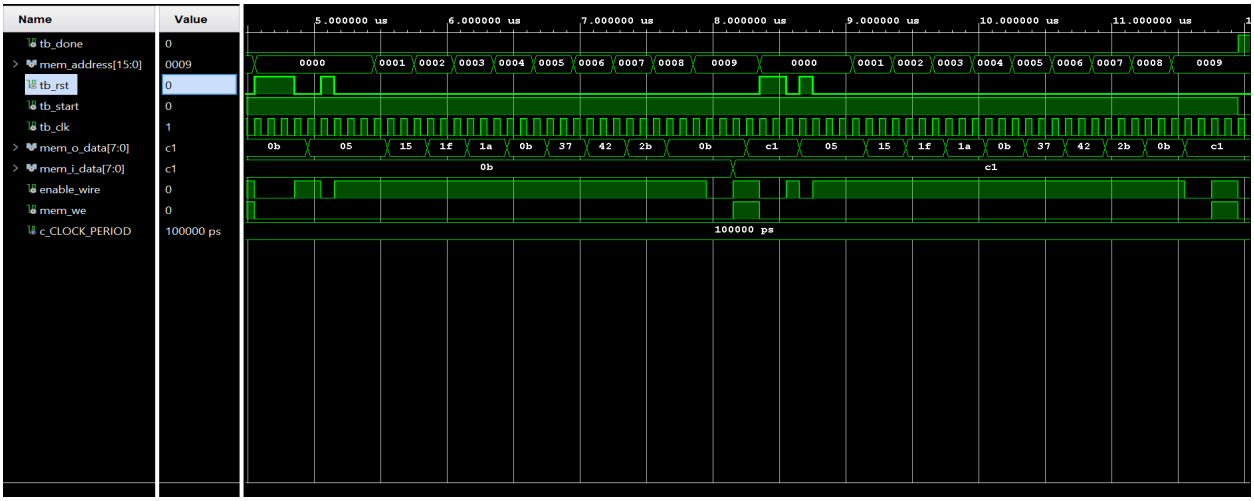

Figure 3: Waveform estratta dal test reset asincrono

2. **Start senza reset**

   Questo test si è rivelato importante in quanto è stato utilizzato per controllare il corretto funzionamento del segnale di `start` e di `done` e come il componente reagisce nel momento in cui bisogna avviare una nuova computazione senza reset. Infatti, se il `reset` non viene azionato, la computazione comincia nuovamente, senza però, reinizializzare le variabili e i segnali, tranne *counter*, il quale viene reinizializzato a '8' nello stato `DONE`.

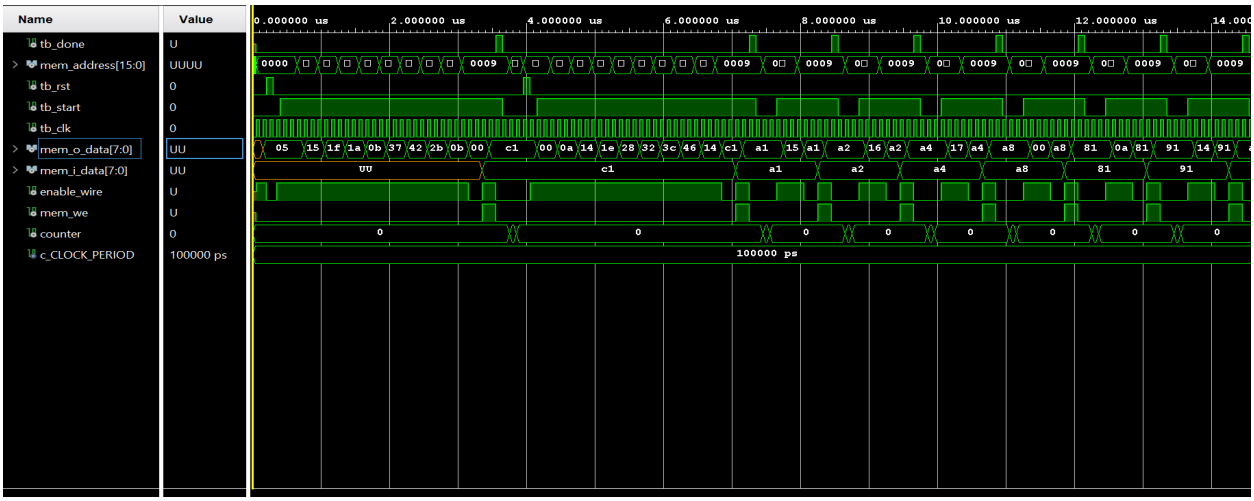

Figure 4: Waveform estratta dal test con start senza reset

3. **WZ adiacenti e WZ ai limiti della memoria**

   Questo test si è rivelato importante in quanto è stato utilizzato per controllare come il componente reagisce quando le working zone sono adiacenti fra di loro, o quando si trovano ai limite della memoria. Inoltre, il suddetto test è stato utile per capire se le working zone vengano individuate e salvate correttamente.

   Un altro controllo che questo test effettua riguarda gli offset delle working zone. Infatti viene verificato che tutti gli offset siano raggiungibili e codificati correttamente dal componente.

   Ancora, il test verifica la corretta conversione del dato in ingresso, visionando, in particolare, che il dato da inserire in memoria sia stato composto nella maniera corretta, concatenando i giusti bit.

4. **Overflow**

   Questo test si è rilevato importante in quanto è stato utilizzato per controllare che tutti le variabili assumano i valori previsti, non andando così in overflow.

4 Conclusioni

Il componente realizzato ha dimostrato di superare correttamente, con tutti i test, la simulazione *Behavioural*, la simulazione *Post-Syntesis Functional* e la simulazione *Post-Syntesis Timing*. La scelta di salvare le working zone all'inizio della computazione si è rilevata ottima dal punto di vista temporale, a discapito, però, dell'area di memoria, la quale sarà maggiore rispetto al caso in cui le working zone non vengono salvate. Perciò, si può concludere che il componente è in grado di risolvere il problema richiesto.