Prova Finale di Reti Logiche

Matteo Pancini[1]

Anno Accademico 2021/2022

[1]Codice Persona: 10656944, Matricola: 932095, Docente: Gianluca Palermo

Indice

1 Introduzione **2**
1.1 Codificatore 1/2 2
1.2 Interfaccia di memoria 4
1.3 Riepilogo specifica 4

2 Architettura **6**
2.1 Schema del componente 6
2.2 Implementazione 7
2.2.1 Stati della macchina 8
2.2.2 Variabili utilizzate 9
2.2.3 Schematico 9
2.3 Sintesi 10

3 Risultati sperimentali **12**
3.1 Test bench 1 12
3.2 Test bench aggiuntivi 13

4 Conclusione **15**

Capitolo 1

Introduzione

Il progetto di reti logiche di quest'anno consiste nell'implementazione in VHDL di un modulo hardware che opera una convoluzione 1/2 su uno stream continuo di bit , ossia dati n bit in ingresso si genera uno stream di $2n$ bit in uscita.

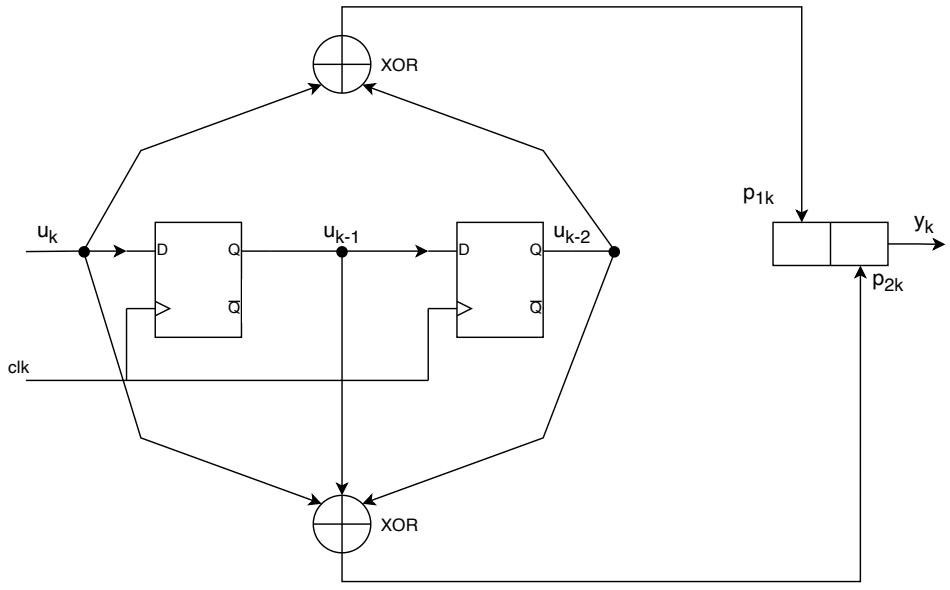
Analizziamo adesso singolarmente le due parti che andranno a costituire il nostro modulo:

1. il codificatore 1/2
2. l'interfaccia di memoria

1.1 Codificatore 1/2

Il core del progetto riguarda proprio la conversione dello stream di bit in ingresso.

Il codificatore è un modulo di questo tipo:

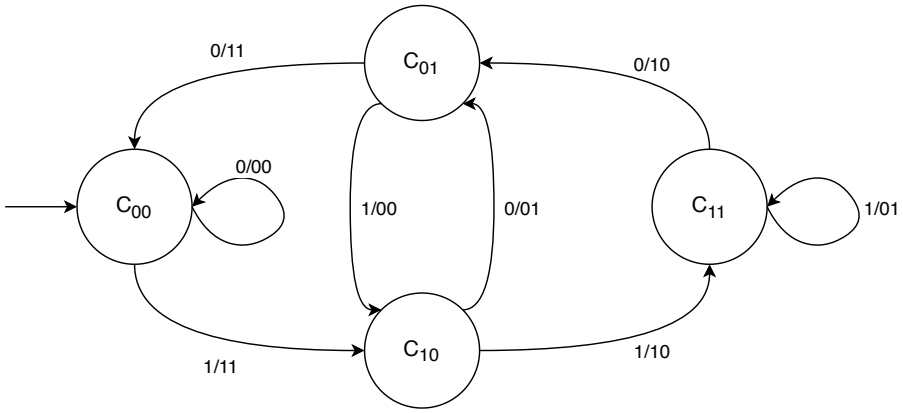


In particolare il codificatore segue il seguente algoritmo: dato un istante k, definito da un impulso del segnale di clock (clk), e dato il bit u_k ricevuto in ingresso, con i relativi bit u_{k-1} e u_{k-2} ricevuti nei due istanti precedenti, la macchina genera un uscita y_k.

L'uscita y_k è costituita da due bit:

- $p_{1k} = u_k \oplus u_{k-2}$
- $p_{2k} = u_k \oplus u_{k-1} \oplus u_{k-2}$

Questo algoritmo, può essere descritto in maniera più semplice e intuitiva mediante la seguente macchina a stati finiti:



Per quanto riguarda la nomenclatura degli stati, delle transizioni e delle uscite, con riferimento al codificatore descritto poco sopra, possiamo dire che:

- il nome di ogni stato contiene i bit ricevuti agli istanti k e $k-1$ della macchina, che corrispondono dunque ai bit u_k e u_{k-1}
- i bit in uscita corrispondono rispettivamente a p_{1k} e p_{2k} e generano dunque l'uscita y_k
- ogni transizione ha dunque la forma $u_k/p_{1k}p_{2k}$

Osservazione. Si osservi che la prima volta che entriamo nella fase di codifica si assume che i bit u_{k-1} e u_{k-2} siano 0 e quindi che lo stato di reset corrisponda allo stato C_{00} della macchina.

Esempio. Sfruttiamo un esempio per chiarire meglio il funzionamento dell'algoritmo di conversione mediante l'uso della macchina a stati.

Vogliamo convertire il numero 163.

$$163 \xrightarrow{bin} 10100011$$

La macchina opererà i seguenti passaggi:

$$\boxed{C_{00}} \xrightarrow{1/11} \boxed{C_{10}} \xrightarrow{0/01} \boxed{C_{01}} \xrightarrow{1/00} \boxed{C_{10}} \xrightarrow{0/01} \boxed{C_{01}} \xrightarrow{0/11} \boxed{C_{00}} \xrightarrow{0/00} \boxed{C_{00}} \xrightarrow{1/11} \boxed{C_{10}} \xrightarrow{1/10} \boxed{C_{11}}$$

Dunque lo stream in uscita ottenuto, dopo uno split delle due singole parole da scrivere, sarà:

$$1101000111001110 \rightarrow \begin{array}{l} 11010001 \xrightarrow{dec} 209 \\ 11001110 \xrightarrow{dec} 206 \end{array}$$

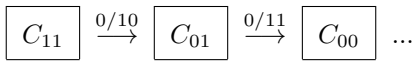
Nel funzionamento completo del modulo bisogna tenere in considerazione il fatto che, per la pura conversione, lo stream è continuo, dunque il convertitore per parole successive di memoria non dovrà ripartire ogni volta dallo stato iniziale C_{00} (a meno di eventuali reset), ma dovrà proseguire nella conversione a partire dall'ultimo stato in cui si trovava nella conversione precedente.

Esempio. Utilizziamo un esempio per rendere più chiaro questo fatto.

Ipotizziamo di leggere i due valori interi: 163 (10100011) e 47 (00101111).

Come visto nell'esempio precedente, una volta letto il valore 163, al termine dell'operazione di convoluzione il convertitore si troverà nello stato C_{11}.

Dopo la scrittura della conversione di 163, il modulo leggerà il valore 47, dunque ripartirà direttamente dallo stato C_{11} ed effettuerà la nuova convoluzione:



Osservazione. Se il convolutore si fosse invece, erroneamente resettato sullo stato C_{00}, leggendo 0 avrebbe restituito la coppia di bit 00 e chiaramente il risultato dell'intera convoluzione sarebbe stato errato.

1.2 Interfaccia di memoria

Il nostro modulo dovrà interfacciarsi con una memoria, con indirizzamento al byte. In particolare dovremo:

- leggere una parola (1 byte) dalla memoria
- effettuare la convoluzione
- scrivere in memoria la conversione ottenuta, quindi 2 parole (2 byte)

reiterando questo processo per ogni parola letta in memoria.

La specifica ci fornisce alcuni dettagli implementativi aggiuntivi quali:

- all'indirizzo 0 della memoria troviamo il numero di parole che dobbiamo convertire
- a partire dall'indirizzo 1 della memoria troviamo le parole da convertire (numero massimo di parole pari a 255)
- ogni parola convertita deve essere scritta a partire dall'indirizzo 1000 della memoria

La memoria avrà dunque una struttura generica di questo tipo:

INDIRIZZO	VALORE	
0	n	//Numero di byte da leggere ($n_{\max} = 255$)
1	x_1	//Byte da codificare
[...]		
n	x_n	
[...]		
1000	y'_1	//Primo byte della codifica di x_1
1001	y''_1	//Secondo byte della codifica di x_1
[...]		
$1000 + 2n$	y''_n	

1.3 Riepilogo specifica

Possiamo dunque riepilogare le operazioni che il nostro modulo deve svolgere secondo il seguente schema:

1. Lettura del numero di byte da leggere dalla memoria
2. Lettura di un singolo byte
3. Operazione di convoluzione 1/2 del byte letto

4. Scrittura in memoria del byte convertito

5. Se restano altri byte da leggere si ritorna al punto (2.) altrimenti si termina l'esecuzione

Tutto questo processo è scandito da un clock con periodo di almeno 100 ns e l'inizio della prima codifica verrà sempre anticipato da un segnale di reset.

Il componente deve inoltre essere in grado di gestire in qualsiasi momento eventuali ulteriori segnali di reset che comportano l'istantanea sospensione di ogni operazione del modulo e il suo reset a condizioni iniziali.

L'intero progetto è stato svolto attraverso il software Vivado 2021.2, utilizzando come device un FPGA della famiglia Kintex-7 (xc7k70tfbv676-1).

Capitolo 2

Architettura

2.1 Schema del componente

Dal punto di vista architetturale, il nostro componente ha la seguente interfaccia (descritta in VHDL):

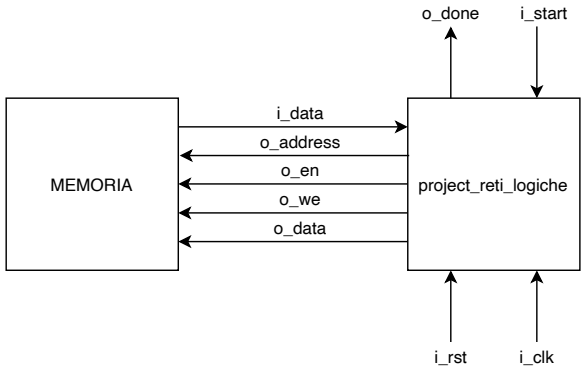
```
entity project_reti_logiche is
    Port (
        i_clk : in std_logic;
        i_rst : in std_logic;
        i_start : in std_logic;
        i_data :  in std_logic_vector(7 downto 0);
        o_address : out std_logic_vector(15 downto 0);
        o_done : out std_logic;
        o_en : out std_logic;
        o_we : out std_logic;
        o_data : out std_logic_vector (7 downto 0)
        );
```

in cui:

- `i_clk` = segnale di clock che caratterizza il funzionamento del componente
- `i_rst` = segnale di reset usato per:
  - inizializzare la macchina prima di ricevere il primo start
  - sospendere operazioni correnti e inizializzare il componente
- `i_start` = segnale di start delle varie operazioni
- `i_data` = segnale (vettore) che ci permette di leggere dalla memoria
- `o_address` = segnale (vettore) che contiene l'indirizzo che richiediamo alla memoria, su cui vogliamo operare
- `o_done` = segnale di uscita che indica la terminazione delle operazioni da parte del componente
- `o_en` = segnale di enable, che deve essere settato a 1 per poter interagire con la memoria (sia in lettura sia in scrittura)
- `o_we` = segnale di write enable, che deve essere:
  - 0 per poter leggere dalla memoria
  - 1 per poter scrivere in memoria

  Osservazione. Chiaramente se `o_en` = 0, il segnale `o_we` è praticamente inutile.
- `o_data` = segnale (vettore) che esce dal componente (con i dati elaborati dal nostro modulo) e opera in memoria

A livello schematico si deve dunque far funzionare una struttura di questo tipo:



2.2 Implementazione

Sfruttando una memoria già istanziata e sintetizzata nei test bench dobbiamo dunque progettare il componente.

Dal momento che la specifica non impone alcuna scelta progettuale sul componente si è scelto di sviluppare il modulo sottoforma di un unico modulo architetturale, costituito da una macchina a stati che svolge tutte le operazioni del processo considerato.

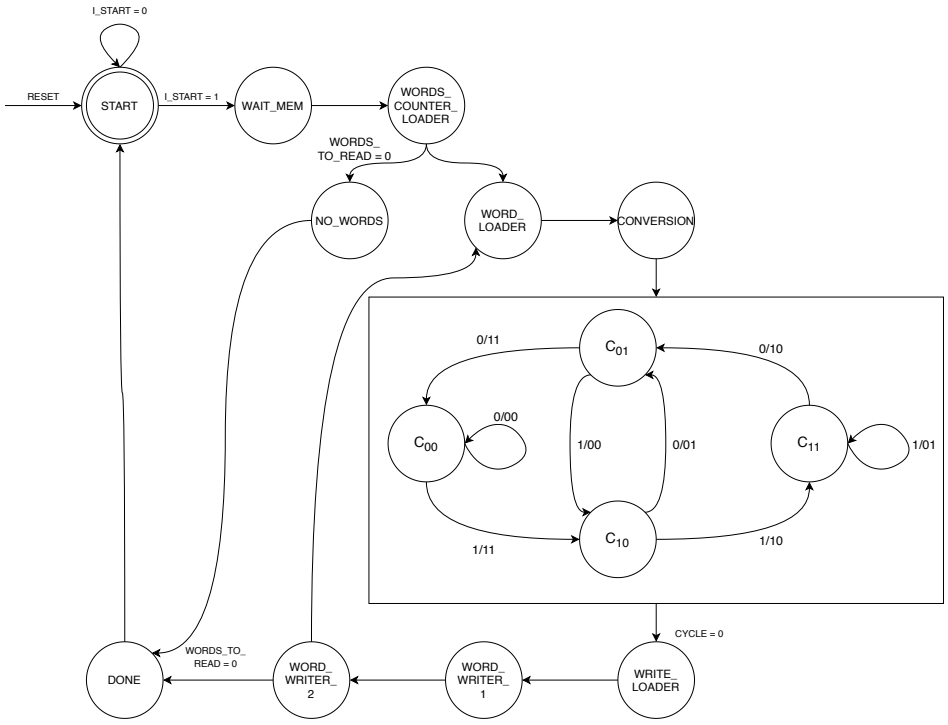
La scelta è stata supportata dalla sua estrema semplicità e chiarezza, che rendono il modulo facilmente manutenibile. Inoltre questa scelta evita ripetizioni nel codice e ne permette un facile controllo di esecuzione sia in pre-sintesi, sia in post-sintesi.

Osservazione. Lo stesso risultato si sarebbe potuto ottenere mediante un'architettura modulare, che separasse le funzionalità di interfaccia con la memoria e di codifica.

Tuttavia, proprio per la descrizione del codificatore, esso non necessita di chissà quali condizioni di scalabilità che la modularità potrebbe garantire.

Per questo motivo si è optato per un unico modulo, garantendo maggiore intuitività e pulizia di tutto il processo.

Il funzionamento del componente è stato descritto attraverso la seguente macchina a stati:



Osservazione. Si noti che nel disegno della macchina si è utilizzata una white box per la parte riguardante la conversione, con annesse transizioni di stato generiche.

Questo è frutto del fatto che: alla prima iterazione, la macchina dopo `conversion` andrà certamente in `C00`, a partire dalla seconda iterazione ciò non è più detto perchè seguirà il segnale `new_word_start_state`. Lo stesso ragionamento vale per l'uscita: non possiamo infatti mai sapere, a priori, in quale stato di conversione si fermerà la codifica.

2.2.1 Stati della macchina

Passiamo adesso alla descrizione del funzionamento di ogni stato della macchina.

- START = stato iniziale della macchina, in cui si posiziona all'inizio di ogni computazione (sia che la precedente sia andata a buon fine, sia che sia stato forzato un segnale di reset).
- WAIT_MEM = stato di attesa di un ciclo di clock, mediante il quale si permette alla RAM di dare al nostro componente il contenuto dell'indirizzo di memoria precedentemente impostato (o aggiornato).
- WORDS_COUNTER_LOADER = stato di caricamento del numero di parole di memoria che dovranno essere lette durante l'esecuzione, salvandolo nella variabile `words_to_read`. Si assegna inoltre il nuovo indirizzo della RAM in modo da poter iniziare subito, a partire dal prossimo ciclo di clock, la lettura delle parole da convertire.
- WORD_LOADER = stato di caricamento della nuova parola di memoria da convertire, che da ora in poi sarà contenuta in `i_data`.
- CONVERSION = stato iniziale della conversione di una parola. Viene utilizzato per inizializzare le variabili `cycle` e `i_to_write`, indici utili per la conversione.
- C00 = stato di conversione che corrisponde ad aver ricevuto nei due istanti precedenti rispettivamente i bit 0 $(t-1)$ e 0 $(t-2)$. Coincide anche con lo stato di reset di ogni nuova conversione complessiva.
- C01 = stato di conversione che corrisponde ad aver ricevuto nei due istanti precedenti rispettivamente i bit 0 $(t-1)$ e 1 $(t-2)$.
- C10 = stato di conversione che corrisponde ad aver ricevuto nei due istanti precedenti rispettivamente i bit 1 $(t-1)$ e 0 $(t-2)$.
- C11 = stato di conversione che corrisponde ad aver ricevuto nei due istanti precedenti rispettivamente i bit 1 $(t-1)$ e 1 $(t-2)$.
- WRITE_LOADER = stato di caricamento della scrittura. Permette di:
  - alzare il segnale di abilitazione della scrittura (`o_we`)
  - preparare il primo byte da scrivere in memoria
  - incrementare i valori delle variabili `address_in` e `address_out` in modo che siano pronte per le prossime letture/scritture
- WORD_WRITER_1 = stato di scrittura del primo byte dopo la conversione. Si effettuano inoltre le seguenti operazioni:
  - si aggiorna il segnale `o_data` per la scrittura del secondo byte al prossimo ciclo di clock
  - si decrementa il numero di parole rimaste da leggere in modo tale da effettuare il controllo al prossimo ciclo di clock
  - tramite la variabile `address_out`, precedentemente incrementata, si aggiorna l'indirizzo di memoria su cui scrivere
- WORD_WRITER_2 = stato di scrittura del secondo byte dopo la conversione. La scrittura termina, dunque si abbassa il segnale `o_we`. A questo punto si apre uno statement condizionale:
  - se abbiamo finito di leggere tutte la parole richieste, allora l'esecuzione termina (si alza il segnale di terminazione `o_done` e si abbassa il segnale di elaborazione `o_en`)

  - se rimangono ancora parole da leggere si aggiorna l'indirizzo di memoria in modo da fargli leggere una nuova parola al prossimo ciclo di clock e si incrementa anche la variabile `address_out` in modo tale da prepararla per la successiva scrittura

- NO_WORDS = stato utile per risolvere in maniera veloce ed efficacie il caso in cui all'indirizzo 0 indichi di leggere 0 byte. Proprio come accade, in condizione di normale funzionamento, allo stato WORD_WRITER_2, si alza il il segnale di terminazione `o_done` e si abbassa il segnale di elaborazione `o_en`.

- DONE = stato di terminazione del processo, in cui si aspetta che il segnale di start (`i_start`) venga abbassato in modo tale da abbassare il segnale di terminazione `o_done` e di riportare la macchina allo stato iniziale (START).

I segnali utilizzati per le transizioni della macchina sono:

- `current_state` = segnale che segue, durante tutto il processo, lo stato corrente della macchina.

- `new_word_start_state` = segnale che, dopo ogni convoluzione 1/2, contiene l'ultimo stato toccato dalla conversione. Questo stato deve essere in qualche modo «memorizzato» perchè l'insieme delle parole in ingresso viene trattato come uno stream unico quindi, per ogni nuova parola letta, bisogna iniziare la sua conversione a partire dall'ultimo stato incontrato durante la conversione precedente.

  Osservazione. Questo dettaglio implementativo è stato descritto, con esempio annesso, anche nella trattazione specifica del codificatore 1/2.

2.2.2 Variabili utilizzate

Per garantire un buon funzionamento della macchina e semplificarne i processi si è fatto uso di alcune variabili, di cui discuteremo adesso la funzionalità. Si è cercato di renderle, comunque, il più autoesplicative possibili in modo da garantire manutenibilità e comprensibilità del codice non solo in fase di sviluppo ma anche in fase di lettura e analisi.

- `words_to_read` (integer range 0 to 255) = variabile che contiene il numero di parole da leggere in memoria in un determinato processo di conversione. Il dominio è ottenuto dalla specifica richiesta, cioè che si possa leggere un numero massimo di byte pari a 255.

- `word_out` (std_logic_vector (15 downto 0)) = variabile utile nella fase di convoluzione 1/2 in quanto contiene proprio l'output di ogni transizione di stato, dunque la conversione da scrivere in memoria.

- `address_in` (std_logic_vector (15 downto 0)) = variabile che contiene durante tutto il funzionamento della macchina l'indirizzo (sempre aggiornato) da cui, a ogni iterazione, bisogna leggere in memoria.

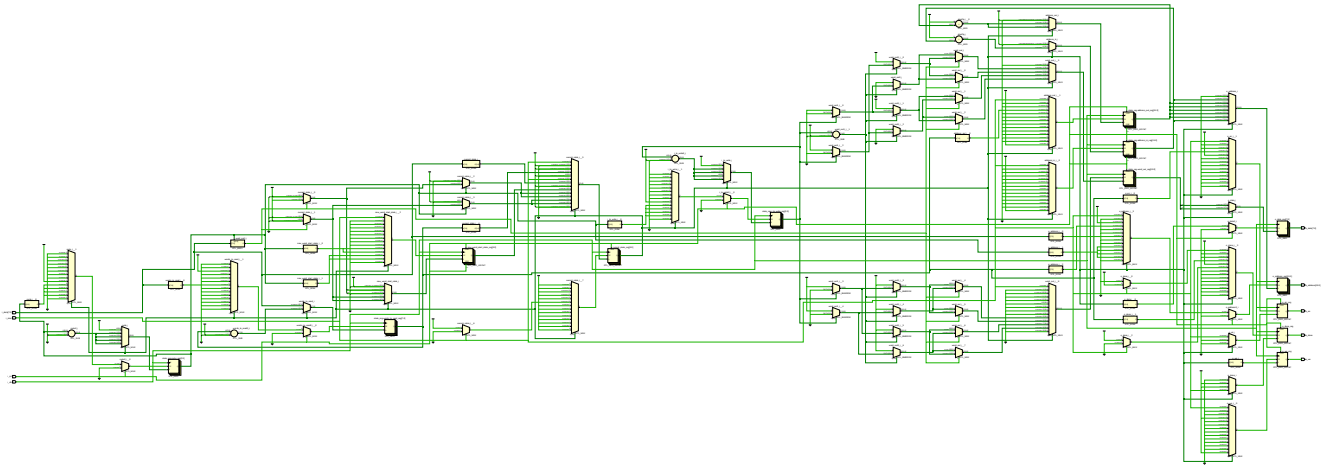
- `address_out` (std_logic_vector (15 downto 0)) = variabile che contiene durante tutto il funzionamento della macchina l'indirizzo (sempre aggiornato) in cui, a ogni iterazione, bisogna scrivere in memoria.

- `cycle` (integer range 0 to 7) = variabile utile nella fase di convoluzione 1/2 in quanto funge da indice della parola in input (`i_data`) che vogliamo convertire.

- `i_to_write` (integer range 0 to 15) = variabile utile nella fase di convoluzione 1/2 in quanto funghe da indice della parola in output (`word_out`) che poi dovremo scrivere in memoria.

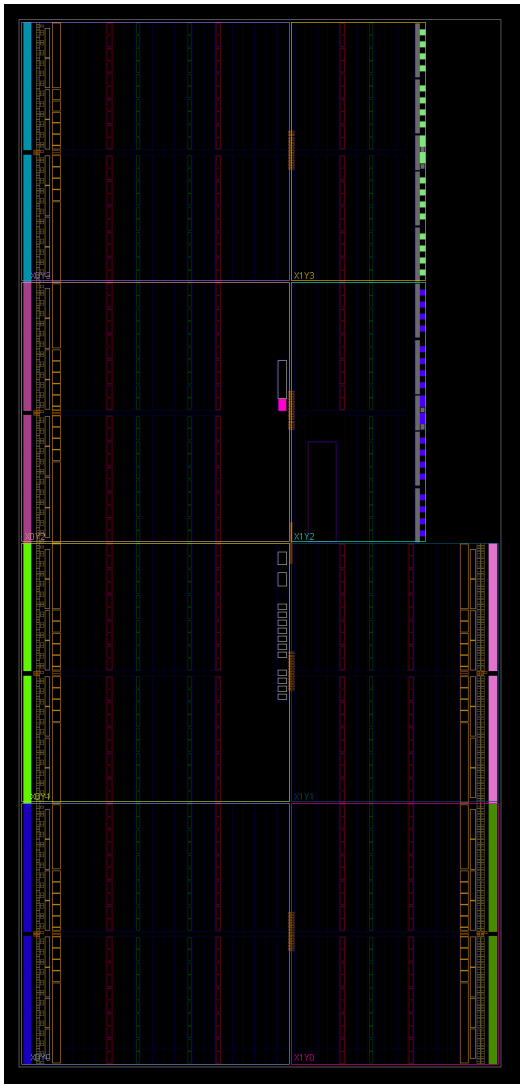
2.2.3 Schematico

L'RTL analysis del codice VHDL ha generato il seguente schematico:



2.3 Sintesi

Il device generato in fase di sintesi è il seguente:



In particolare la sintesi, in termini di area occupata, riporta i seguenti dati:

- LUT: 135 (0.33% del totale)
- FF: 97 (0.12% del totale)

Non essendoci, nella specifica, particolari richieste di ottimizzazione dello spazio occupato si è lasciato questo aspetto totalmente alla responsabilità del tool di sintesi.

Osservazione. Si noti che, in fase di scrittura del codice, si è fatta particolare attenzione al fine di evitare l'insorgere di latch.

Capitolo 3

Risultati sperimentali

Una volta generato il design del nostro componente, questo è stato testato mediante simulazioni sia behavioral sia in post sintesi.

Oltre al test bench fornito sono stati generati due ulteriori test bench, ottenuti a partire dagli esempi forniti dalla specifica. Questi primi tre test sono stati utilizzati in fase di sviluppo per verificare la correttezza delle operazioni svolte dalla macchina.

Successivamente si sono sfruttati ulteriori test aggiuntivi che andavano a controllare il buon funzionamento anche in casi limite.

Vediamo di seguito i risultati sperimentali ottenuti.

3.1 Test bench 1

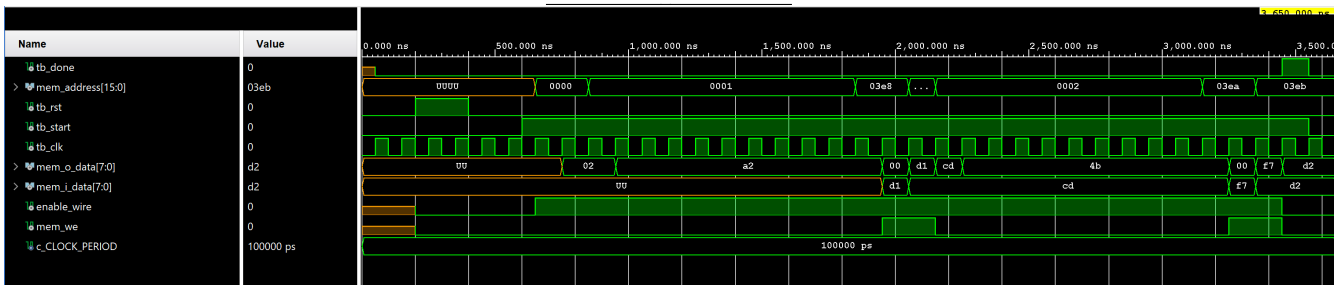
Analizziamo adesso, a titolo di esempio, il test bench 1 fornito dalla specifica.

I dati forniti in memoria sono i seguenti:

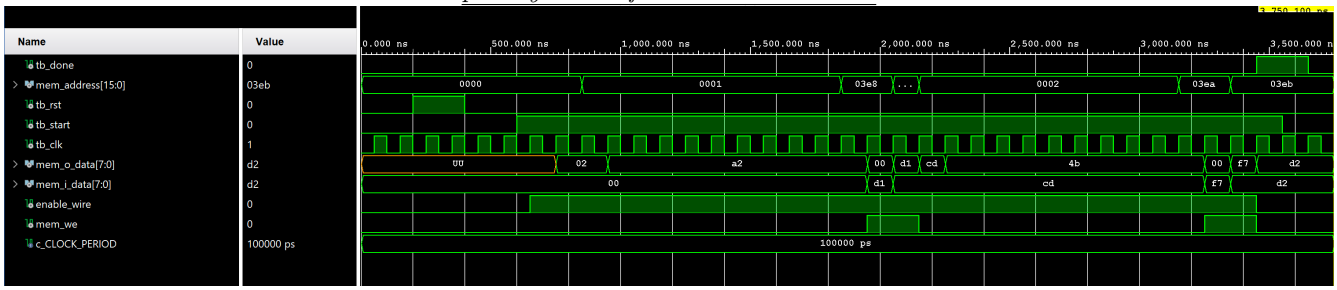
INDIRIZZO	VALORE	
0	2	
1	162	//bin 10100010, hex $a2$
2	75	//bin 01001011, hex $4b$
[...]		
1000	209	//bin 11010001, hex $d1$
1001	205	//bin 11001101, hex cd
1002	247	//bin 11110111, hex $f7$
1003	210	//bin 11010010, hex $d2$

Riportiamo di seguito le waveform in simulazione behavioral e in post sintesi.

behavioral simulation



post-synthesis functional simulation



Osservazione. Si può notare il fatto che i don't care siano scomparsi in post-sintesi, sostituiti da valori reali.

Per evitare di risultare inutilmente prolissi, dal momento che gli altri due esempi forniti nella specifica non aggiungono informazione alla trattazione, si evita di riportare i testbench 2 e 3.

3.2 Test bench aggiuntivi

Al fine di testare in maniera completa il modulo si sono generati alcuni test bench generici e non.

Di seguito alcune specifiche del componente che sono state testate:

- sequenze con 0 byte da leggere (lunghezza minima)
- sequenze con 255 byte da leggere (lunghezza massima)
- sequenze con presenza di reset
- sequenze con assenza di reset (tranne quello iniziale)

I test sono stati condotti facendo variare anche il periodo di clock, dimostrando una buona esecuzione del componente non solo a 100 ns, come richiesto, ma anche a 1 ns sia in behavioral, sia in post-sintesi.

Di particolare interesse è il test condotto su 10000 test case sequenziali, tutti con una lettura e successiva codifica di 255 byte.

Il test in questione è stato eseguito sia senza alcun reset al termine di ogni elaborazione (giocando quindi solo con il segnale di `i_start` e di `o_done`), sia con reset. Chiaramente in tal modo si verificano anche i comportamenti del modulo in presenza di eventuali reset intermedi.

Non si riporta la waveform completa per evidenti motivi di spazio.

Seguono i tempi di esecuzione del test considerando un periodo di clock di 100 ns.

Test senza reset:

- behavioral simulation: 1 669 126 750 ns
- post-synthesis functional simulation: 1 671 126 650 ns

Test con reset:

- behavioral simulation: 1 671 126 550 ns
- post-synthesis functional simulation: 1 673 126 450 ns

Osservazione. Si noti che il componente è leggermente più efficiente in assenza di reset, in quanto ciò permette di evitargli l' operazione di setup del segnale `o_done`. La differenza è minima in quanto, per il funzionamento stesso della specifica, il «reset» di segnali e variabili deve comunque essere fatto a ogni nuova iterazione.

Capitolo 4

Conclusione

In conclusione si ritiene che l'architettura progettata rispetti la specifica richiesta, dal momento che il componente ha risposto correttamente ai test (manuali e casuali) a cui è stato sottoposto, senza errori o warning sia in pre-sintesi che in post-sintesi.

Il codice risulta semplice e intuitivo alla lettura, il che permette una facile manutenzione e comprensione.

La macchina è basata su un unico processo che, proprio per la sua specificità di utilizzo in termini di codifica, viene risvegliato soltanto quando richiesto.