Prova Finale di Reti Logiche

Montanari Tommaso e Negri Riccardo

15 Maggio 2022

POLITECNICO
MILANO 1863

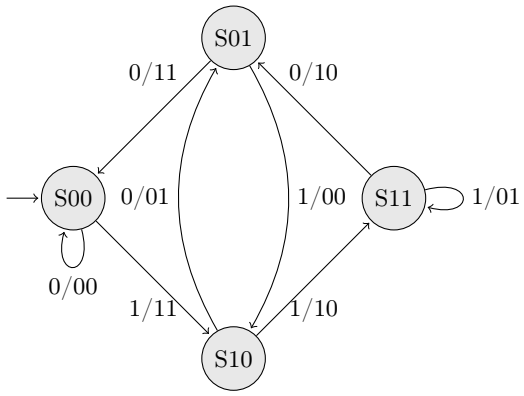
Docente:	Salice Fabio		
Studente 1:	Montanari Tommaso	10661941	932673
Studente 2:	Negri Riccardo	10729927	936820

Indice

1 Introduzione

La prova prevede l'implementazione in VHDL di un componente che opera su una memoria. Il componente deve per prima cosa leggere il primo byte dalla memoria che identifica il numero di parole che sono state fornite come input. Questa informazione è importante per capire quando la macchina deve terminare la lettura. Dopodiché ogni parola successiva viene tradotta in due parole che vengono scritte progressivamente in un'altra parte della memoria a partire dall'indirizzo 1000. Le parole di memoria sono interpretate dal codificatore (rappresentato qui sotto) come un flusso continuo di bit e l'uscita è un flusso di lunghezza doppia. Visto che i byte vengono tradotti come un unico flusso, l'output di una certa parola dipende anche da quelle precedenti e non possono essere processate separatamente.

1.1 Codificatore convoluzionale



1.2 Esempio

Dato lo stato iniziale della memoria riportato di seguito il componente legge la prima parola che rappresenta la lunghezza dell'input (W=2).

Indirizzo	Valore	Codifica binaria
0	2	0000 0010
1	35	0010 0011
2	161	1010 0001

Successivamente legge la parola presente all'indirizzo 1 e la elabora secondo il funzionamento del codificatore convoluzionale. Essendo la prima parola letta, il codificatore parte dallo stato S00. Di seguito sono riportati gli stream in input e output dal codificatore:

Stream	Codifica binaria	Parole corrispondenti
Input	0010 0011	35
Output	0000 1101 1100 1110	13 206

A seguito di questa codifica il codificatore convoluzionale si trova nello stato S11. Dopo avere scritto in memoria le due parole, il componente legge la seconda, e ultima, parola. Viene quindi eseguita la codifica continuando dallo stato S11. Di seguito sono riportati gli stream in input e output dal codificatore:

Stream	Codifica binaria	Parole corrispondenti
Input	1010 0001	161
Output	0110 0001 1100 0011	97 e 195

Le ulteriori due parole vengo quindi scritte in memoria e, una volta concluso il processo, la memoria appare come rappresentato di seguito.

Indirizzo	Valore	Codifica binaria
1000	13	0000 1101
1001	206	1100 1110
1002	97	0110 0001
1003	195	1100 0011

1.3 Ipotesi progettuali

- Per lo sviluppo del componente si è utilizzata una scheda Artix-7 FPGA xc7a200tfbg484-1.
- La quantità di numeri in ingresso (W) è contenuta in una parola da un byte quindi anche il numero massimo di parole da tradurre è 255.
- Dato che l'input occupa al massimo 256 byte è possibile scrivere in memoria a partire dai byte successivi. Da specifiche si scrive sempre a partire dall'indirizzo 1000 che quindi sicuramente non conterrà l'input.

2 Architettura

L'implementazione è stata realizzata in un solo modulo per semplicità, si possono distinguere due parti all'interno del componente che cooperano per raggiungere il risultato. È presente una macchina che simula il funzionamento dell'automa a stati finiti fornito nella specifica, esso rappresenta il codificatore convoluzionale responsabile della traduzione uno a due dei bit. Inoltre è presente una seconda macchina principale che si occupa della lettura e scrittura dei flussi di bit e tratta la prima come una scatola nera. Per ciascuna macchina esiste un segnale distinto che contiene il suo stato.

2.1 Interfaccia

- **i_clk:** Segnale di clock.
- **i_rst:** Segnale di reset, deve riportare la macchina allo stato iniziale.
- **i_start:** Segnale di start generato dal test bench.
- **i_data:** Vettore di 8 bit che contiene la parola di memoria che è stata letta.
- **o_address:** Vettore di 16 bit che contiene l'indirizzo della parola da scrivere o da leggere.
- **o_done:** Segnale di uscita che comunica la fine della elaborazione.
- **o_en:** Se settato a 1 richiede l'operazione di lettura o scrittura.
- **o_we:** Se settato a 1 l'operazione richiesta da o_en è scrittura altrimenti è lettura.
- **o_data:** Vettore di 8 bit che contiene la parola di memoria da scrivere.

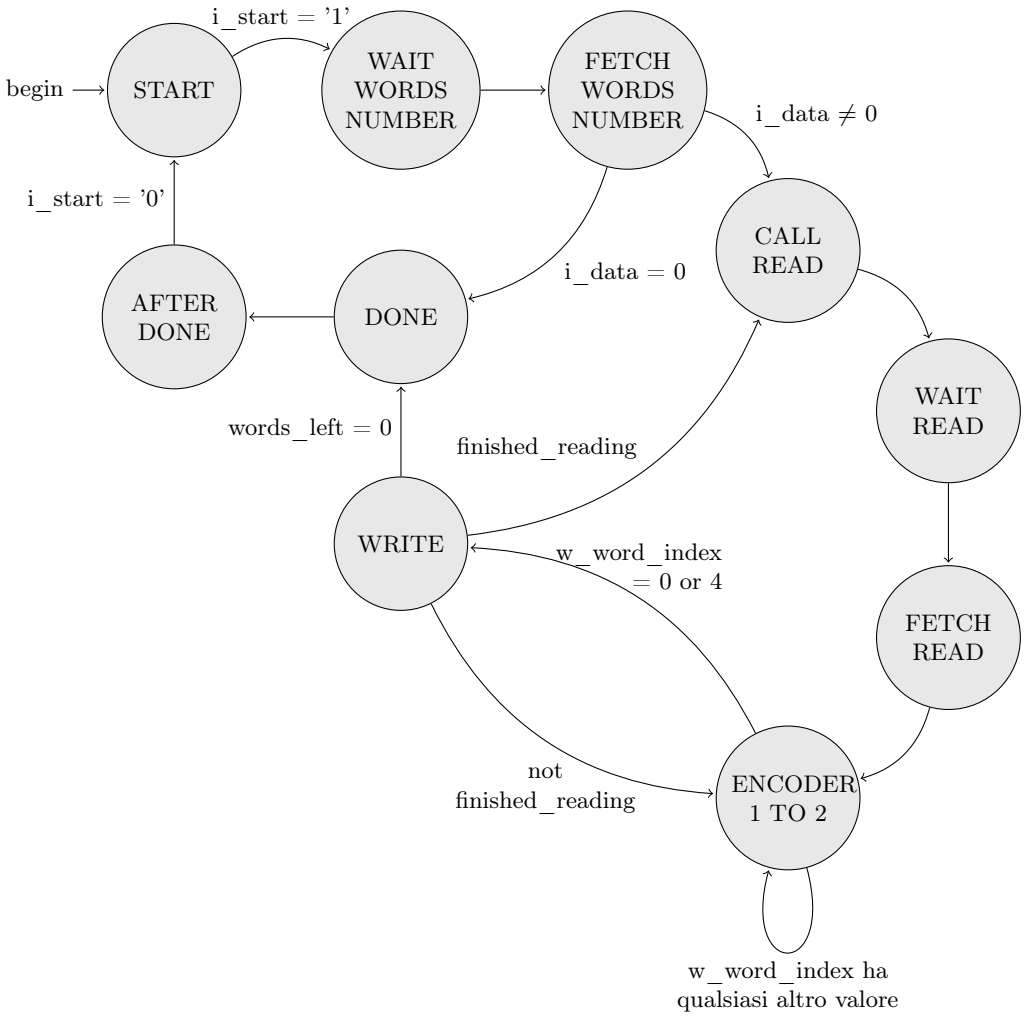
2.2 Descrizione ad alto livello

Questa è la descrizione a parole delle fasi che l'esecuzione segue. All'interno del componente ciascuna di queste diverse fasi corrisponde ad uno stato (o a più stati).

1. Inizio:
    (a) Legge il primo byte di memoria contenente la lunghezza dell'input e salva il valore in un segnale.
    (b) Inizializza l'indirizzo di lettura al secondo byte e quello di scrittura al byte 1000.
2. Traduzione:
    (a) Legge un byte e sposta il punto di lettura.
    (b) Simula un passo dell'automa del codificatore usando in ingresso un bit dal byte appena letto e aggiungendo due bit al segnale di output. (Ripetuto 4 volte)
    (c) Tutti e 8 i bit del segnale di output sono stati impostati, viene quindi salvato in una parola di memoria e si incrementa l'indirizzo di scrittura.
    (d) Se sono rimasti bit da tradurre nel byte letto, torna al punto 2.b
    (e) Altrimenti, se sono rimasti byte da leggere, torna al punto 2.a
    (f) Altrimenti termina.

Ogni operazione di lettura nella memoria si scompone in più stati: richiesta della lettura, attesa della fine dell'operazione, utilizzo del dato letto o salvataggio in un segnale.

2.3 Diagramma degli stati



- **START:** Lo stato in cui la macchina è all'inizio dell'esecuzione. In presenza del segnale i_start = '1' imposta i valori iniziali, fa la richiesta di lettura all'indirizzo 0 e passa allo stato successivo.
- **WAIT_WORDS_NUMBER:** Riporta a 0 il segnale o_en.
- **FETCH_WORDS_NUMBER:** Dopo la fine della lettura salva il numero di byte in input nel segnale words_left. Nel caso particolare in cui ci sono 0 byte in ingresso passa direttamente allo stato DONE, altrimenti incomincia con la lettura del primo byte.

- **CALL_READ:** Richiede la lettura e imposta l'indirizzo corretto.
- **WAIT_READ:** Riporta a 0 il segnale o_en.
- **FETCH_READ:** Dopo che la lettura è terminata salva il valore letto in w_word.
- **ENCODER_1_TO_2:** In questo stato viene simulato il codificatore che effettua la traduzione, lo stato di tale automa è salvato in un segnale separato chiamato internal_state. Ogni 4 bit tradotti z_word, ovvero la parola da un byte in uscita, è piena e quindi si passa alla scrittura di tale parola in memoria.
- **WRITE:** Scrive z_word in memoria. Dopodiché, se rimangono dei bit da leggere in w_word, torna a ENCODER_1_TO_2. Se la parola è finita ma ci sono altre parole in input da leggere torna a CALL READER, se tutte le parole sono state tradotte allora passa a DONE.
- **DONE:** Porta il segnale o_done a 1.
- **AFTER_DONE:** Prima di tornare allo stato START si assicura che il segnale i_start che era stato dato all'inizio sia stato riportato a 0 e in tal caso riporta a 0 il segnale o_done.

3 Risultati sperimentali

3.1 Sintesi

Il componente sviluppato è sintetizzabile ed implementabile.

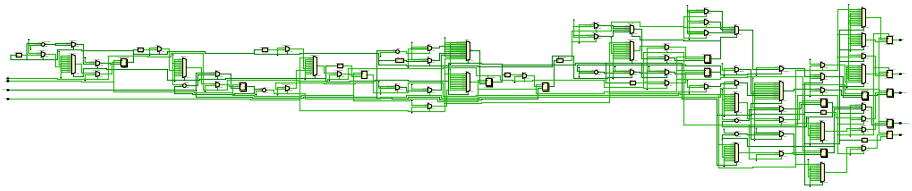


Figura 1: Schema post-sintesi

Nella seguente tabella sono indicati i componenti utilizzati nella sintesi.

Site Type	Used	Available	Utilization%
LUT as Logic	117	134600	0.09
Register as Flip Flop	95	269200	0.04
F7 Muxes	1	67300	<0.01

3.2 Simulazioni

Al fine di verificare il corretto funzionamento del componente sono stati eseguiti diversi test bench in simulazioni Behavioural, Post-synthesis functional e Post-synthesis timing. Di seguito sono riportati i test significativi effettuati con relativa spiegazione, tempi ottenuti e grafico d'onda.

3.2.1 Flussi successivi

Tramite questo test si verifica la correttezza nel caso di codifica di più flussi uno dopo l'altro. Nel test bench usato vengono codificati tre flussi (senza reset dopo ogni flusso).

Behavioural 21850 ns

Post-synthesis functional 22350100 ps

Post-synthesis timing 22353714 ps

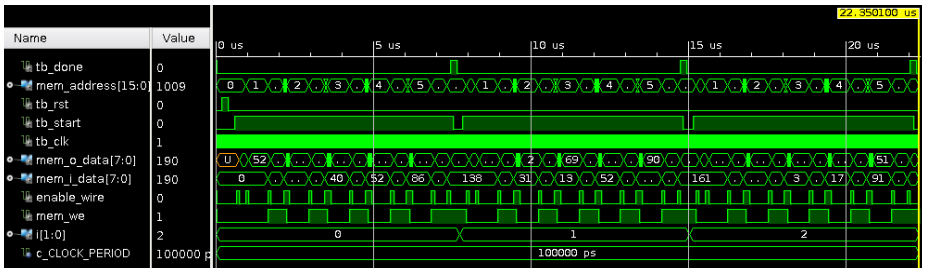


Figura 2: Post-synthesis functional simulation waveform

3.2.2 Reset asincrono

Tramite questo test si verifica il corretto funzionamento del reset asincrono.

Behavioural 10650 ns

Post-synthesis functional 10750100 ps

Post-synthesis timing 10753714 ps

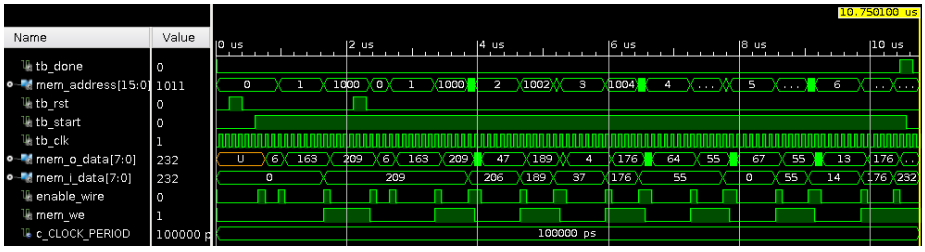


Figura 3: Post-synthesis functional simulation waveform

3.2.3 Sequenza di lunghezza massima

Tramite questo test si verifica il corretto funzionamento nel caso di sequenza di ingresso di lunghezza massima: 255 byte.

Behavioural 332450 ns

Post-synthesis functional 332550100 ps

Post-synthesis timing 332553714 ps

In questo caso non si riporta grafico della forma d'onda perché non è possibile distinguere i valori che i segnali assumono nella vista "fit".

3.2.4 Sequenza di lunghezza nulla

Tramite questo test si verifica il corretto funzionamento nel caso di sequenza di ingresso di lunghezza minima: 0 byte.

Behavioural 950 ns

Post-synthesis functional 1050100 ps

Post-synthesis timing 1053714 ps

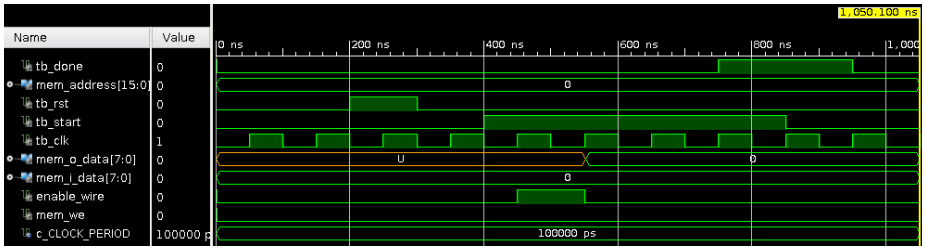


Figura 4: Post-synthesis functional simulation waveform

3.2.5 Test con reset

Test bench, con reset dopo ogni test, che effettua 1000 test (generati casualmente con uno script Python) per testare ulteriormente il componente.

Behavioural 172743250 ns

Post-synthesis functional 172943150100 ps

Post-synthesis timing 172943153714 ps

In questo caso non si riporta grafico della forma d'onda perché non è possibile distinguere i valori che i segnali assumono nella vista "fit".

3.2.6 Test senza reset

Test bench, senza reset dopo ogni test, che effettua 1000 test (generati casualmente con uno script Python) per testare ulteriormente il componente.

Behavioural 172543450 ns

Post-synthesis functional 172743350100 ps

Post-synthesis timing 172743353714 ps

In questo caso non si riporta grafico della forma d'onda perché non è possibile distinguere i valori che i segnali assumono nella vista "fit".

4 Conclusioni

A seguito di estensivo testing effettuato tramite test bench scritti sia manualmente che generati casualmente, si ritiene che l'architettura progettata rispetti le specifiche. Reputiamo inoltre il componente adeguatamente robusto perché ha soddisfatto le specifiche anche in casi limite.

Nella realizzazione del componente in VHDL si è utilizzato un singolo process per avere una maggiore chiarezza, un minore utilizzo di segnali e avere un codice molto leggibile.

Il design è risultato sintetizzabile ed implementabile. La soluzione adottata sfrutta quasi interamente LUT e Flip Flop fatta eccezione per un F7 Muxes, non fa quindi uso di Latch.