Politecnico di Milano

Laurea Triennale in Ingegneria Informatica

Prova Finale Progetto di Reti Logiche

Professore:
William Fornaciari

Studente:
Paolo Cerutti

ANNO ACCADEMICO 2021/2022

Indice

1 Introduzione

La specifica della *Prova Finale (Progetto di Reti Logiche)* 2021/2022 richiede di implementare un modulo Hardware descritto in VHDL che si interfacci con una memoria con l'obiettivo di rappresentare un codificatore convoluzionale con tasso di trasmissione $\frac{1}{2}$.

1.1 Specifica del progetto

Il modulo da implementare deve leggere la sequenza da codificare da una memoria, applicare la convoluzione parola per parola e successivamente salvare in memoria la sequenza codificata. In particolare, dato un ordine, dovrà:

1. Leggere la quantità di parole da codificare dalla memoria;
2. Leggere la sequenza di parole da codificare una alla volta dalla memoria, ogni singola parola di memoria è un Byte;
3. Trasformare la sequenza di Byte in un flusso di Bit (Serializzazione);
4. Applicare al flusso di Bit il codice convoluzionale;
5. Generare in uscita un flusso di Bit;
6. Trasformare il flusso di Bit in Byte (Parallelizzazione);
7. Salvare in memoria il risultato.

Il tasso di trasmissione $\frac{1}{2}$ indica che un Bit viene codificato con due Bit; infatti, il flusso in uscita è ottenuto come concatenamento alternato dei due Bit di uscita.

Il convolutore è una macchina sequenziale sincrona con un clock globale e un segnale di reset avente il diagramma degli stati come in Figura 1.1 e che ha nel suo 00 lo stato iniziale.

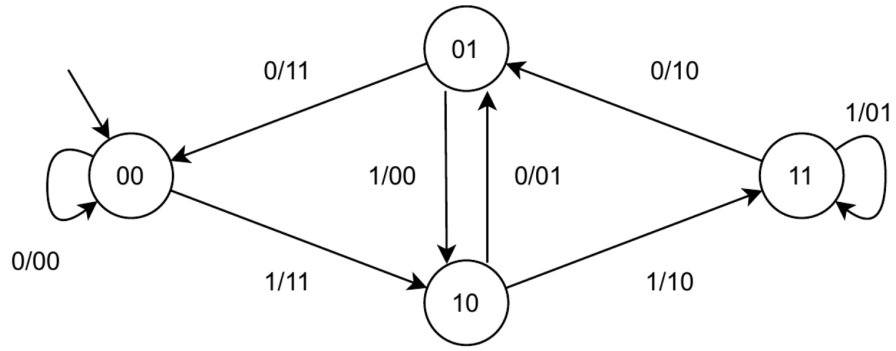


Figura 1.1: Macchina a stati del codificatore convoluzionale

1.2 Dati e dettagli

- Il modulo partirà nell'elaborazione quando un segnale `START` in ingresso verrà portato a 1.
- Il segnale di `START` rimarrà alto fino a che il segnale di `DONE` non verrà portato a 1.
- Al termine della computazione, il modulo da progettare deve portare a 1 il segnale `DONE` che notifica la fine dell'elaborazione e deve rimanere alto finché il segnale di `START` non è riportato a 0.
- Un nuovo segnale `START` non può essere dato finché `DONE` non è stato riportato a 0. Se a questo punto viene rialzato il segnale di `START`, il modulo dovrà ripartire con la fase di codifica.
- Il modulo deve essere dunque progettato per poter codificare più flussi uno dopo l'altro. Ad ogni nuova elaborazione, il convolutore viene portato nel suo stato di iniziale 00.
- Il modulo deve essere progettato considerando che in una seconda elaborazione non dovrà attendere il `RESET` del modulo ma solo la terminazione dell'elaborazione.

1.3 Struttura memoria

La quantità di parole da codificare è memorizzata nell'indirizzo 0; il primo Byte (quindi la prima parola) della sequenza è memorizzato all'indirizzo 1. Le parole codificate in uscita devono essere memorizzate a partire dall'indirizzo 1000. La dimensione massima della sequenza di ingresso è 255 Byte.

1.4 Interfaccia componente

Il componente da descrivere ha un'interfaccia così definita:

```
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)
    );
end project_reti_logiche;
```

In particolare:

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

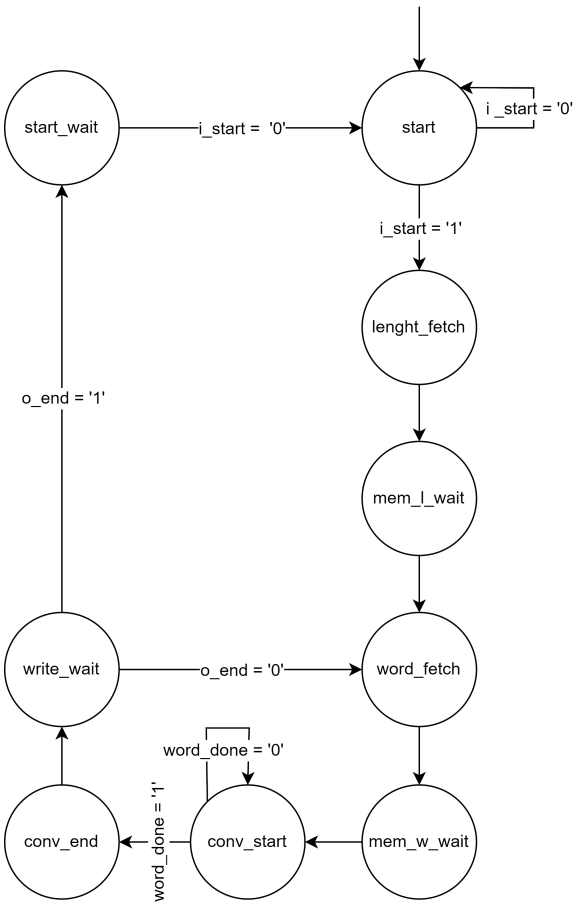
2 Architettura

2.1 Approccio problema

Si è deciso di approcciare il problema con una macchina a stati finiti. Trattandosi di un problema di complessità ridotta si è optato per una soluzione mono modulare con un singolo process che elabora i vari stati.

2.2 Macchina a stati

Di seguito una breve spiegazione degli stati della macchina e i segnali di controllo:



- `start`: Lo stato di inizio, è anche lo stato di reset. In questo stato si aspetta che il segnale `i_start` venga alzato a 1.
- `length_fetch`: Lo stato in cui vengono alzati i segnali per poter leggere dalla memoria, in particolare la lunghezza della parola.
- `mem_l_wait`: Lo stato di attesa per permettere alla memoria di caricare il dato.
- `word_fetch`: Lo stato in cui vengono alzati i segnali per poter leggere dalla memoria, in particolare la parola.
- `mem_w_wait`: Lo stato di attesa per permettere alla memoria di caricare il dato.
- `conv_start`: Lo stato in cui viene eseguita la serializzazione e la codifica della parola. La macchina rimane in questo stato fino a quando `word_done` viene posto a 1.

- `conv_end`: Lo stato in cui la codifica è conclusa e inizia il processo di scrittura in memoria. (Primi Byte)
- `write_wait`: Lo stato in cui termina il processo di scrittura in memoria (Secondo Byte).
- `start_wait`: Lo stato di arrivo se la macchina ha codificato tutte le parole.

2.3 Algoritmi implementati

Di seguito una breve spiegazione della logica implementata per poter affrontare i problemi di serializzazione e parallelizzazione e di codifica delle parole.

Serializzazione

La serializzazione della parola si basa sull'utilizzo di shift. Infatti, quando necessario, viene salvato in un `std_logic` il bit nella posizione più significativa della parola da codificare (che è salvata in un `std_logic_vector`); dopodiché al vettore viene effettuato uno shift una posizione verso sinistra, così che sia pronto per il prossimo ciclo.

Codifica della parola

Supponendo che il bit da codificare sia chiamato msg, i bit in output siano chiamati rispettivamente x_1 e x_0[1] e la bitmask dello stato corrente della macchina del convolutore siano rispettivamente m_1 e m_0[2] allora è possibile constatare che:

$$x_1 = msg \oplus m_0$$

$$x_0 = msg \oplus m_1 \oplus m_0.$$

È anche possibile calcolare il prossimo stato della macchina del convolutore come:

$$m_1 = msg$$

$$m_0 = x_1$$

Parallelizzazione

Dopo la codifica della parola si ha in uscita un flusso di bit. Anche la parallelizzazione del flusso si basa sull'utilizzo di shift, i bit in uscita (`std_logic`) vengono salvati in un `std_logic_vector` in posizione 1 e 0 rispettivamente, dopo di che al vettore viene effettuato uno shift di una posizione verso sinistra, così che sia pronto per il prossimo ciclo.

[1] x_1 è il bit più significativo rispetto a x_0
[2] m_1 è il bit più significativo rispetto a m_0

2.4 Registri Interni

Di seguito una breve descrizione dei segnali interni per la gestione della logica.

```
-- Counters
signal bit_counter : std_logic_vector(7 downto 0);
signal read_address : std_logic_vector(15 downto 0);
signal words_read : std_logic_vector(7 downto 0);
-- Flags signals
signal word_done : std_logic;
signal o_end : std_logic;
signal enable : std_logic;
-- Useful signals
signal words_to_read : std_logic_vector(7 downto 0);
signal word : std_logic_vector (7 downto 0);
signal message : std_logic;
signal merged : std_logic_vector (15 downto 0);
signal state : std_logic_vector (1 downto 0);
```

Counters

- `bit_counter`: Tiene conto di quanti bit sono già stati codificati;
- `read_address`: Tiene conto fino a che punto è stato scritto in memoria;
- `words_read`: Tiene conto di quante parole sono state codificate.

Flags

- `word_done`: Viene settato a 1 se la parola è stata completamente codificata;
- `o_end`: Viene settato a 1 se tutte le parole sono state codificate;
- `enable`: Viene settato a 1 quando inizia la codifica della parola.

Useful signals

- `words_to_read`: Tiene conto di quante parole devono essere codificate;
- `word`: Contiene la parola da codificare;
- `message`: Contiene il bit della parola da codificare (utilizzato per: 2.3);
- `merged`: Contiene il risultato della codifica della parola (utilizzato per: 2.3);
- `state`: Rappresenta lo stato della macchina convoluzionale (utilizzato per: 2.3).

3 Risultati Sperimentali

La sintesi del componente viene eseguita correttamente e tutti i test vengono superati. Segue un piccolo approfondimento dei report e dei test bench.

3.1 Report Utilization, Report Timing

Com'è possibile osservare in Figura 3.1 entrambi i report mostrano che il componente è stato implementato correttamente, non vengono inferiti latch e l'utilizzo del clock è ben al di sotto dei constraints.

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	114	0	134600	0.08
LUT as Logic	114	0	134600	0.08
LUT as Memory	0	0	46200	0.00
Slice Registers	101	0	269200	0.04
Register as Flip Flop	101	0	269200	0.04
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Setup

Worst Negative Slack (WNS):	96,304 ns
Total Negative Slack (TNS):	0,000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	184

All user specified timing constraints are met.

Figura 3.1: Il risultato dei report di Vivado

In particolare il componente è composto da 101 Flip Flop, 0 latch e vengono utilizzati soli 4 nanosecondi a ogni ciclo di clock per completare le operazioni sui 100 disponibili.

3.2 Test Benches

Per verificare l'effettivo funzionamento del componente, esso è stato testato con numerosi test bench e con qualche test mirato sui casi critici nonostante la struttura del progetto non presenta molti corner case che non sono già richiesti all'interno della specifica. Qui di seguito si fornisce una breve descrizione dei casi più importanti.

0 Byte da leggere

Il caso in cui i Byte da leggere sono 0 il componente si comporta correttamente, esso non fa partire la convoluzione e termina il processo.

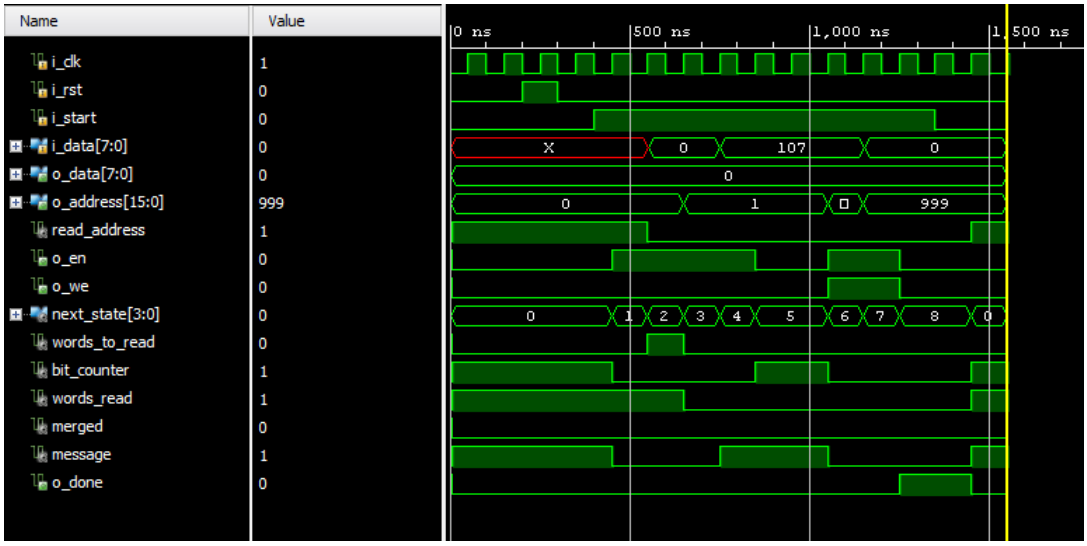


Figura 3.2: Lo schema dei segnali nel caso in cui non ci siano parole da leggere

255 Byte da leggere

Il caso in cui i Byte da leggere sono il massimo richiesto il componente non presenta criticità ed esegue la convoluzione correttamente.

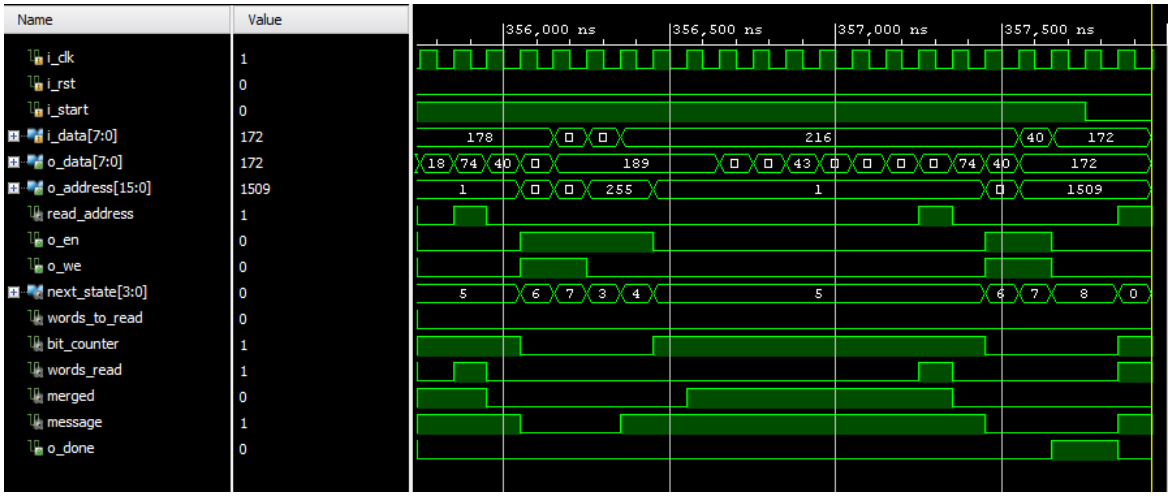


Figura 3.3: Lo schema dei segnali nel caso in cui la sequenza di parole da leggere è 255

Due o più codifiche senza reset intermedio

Nel caso in cui si vogliono eseguire più convoluzioni consecutive è necessario tenere conto che il componente deve essere pronto senza ricevere in ingresso un segnale di reset. Il componente realizzato rispetta la specifica e non ha problemi a gestire convoluzioni multiple.

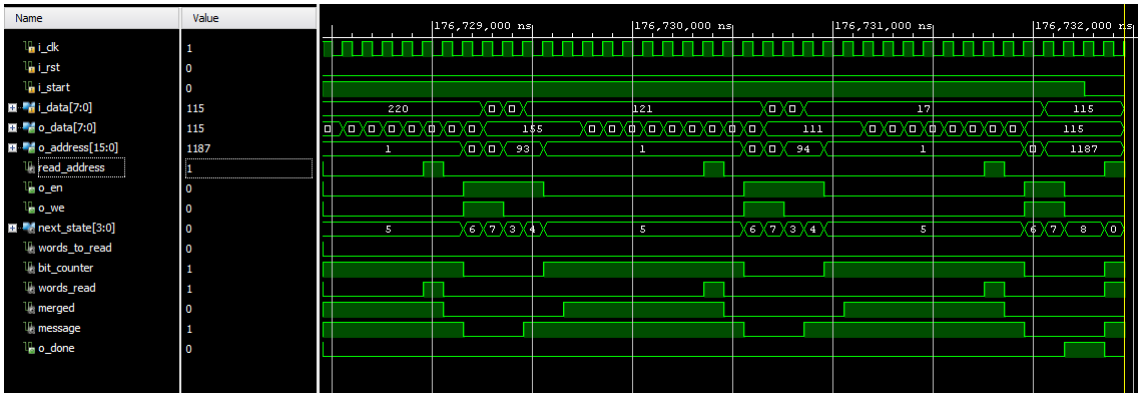


Figura 3.4: Lo schema dei segnali in un caso di codifiche multiple

3.3 Conclusioni e Scelte Progettuali

Si è giunti quindi alla conclusione che il componente realizzato rispetta pienamente le specifiche ed è stato accuratamente testato.

Si è deciso di utilizzare un registro per memorizzare l'indirizzo in cui leggere, e di non memorizzare quello in cui scrivere poiché viene ricavato in fase di esecuzione. Questo permette di avere un architettura più chiara e di risparmiare sull'utilizzo di qualche registro.