Prova Finale di Reti Logiche

Viola Renne

Aprile 2022

Matricola:	932160
Codice Persona:	10681612
Docente:	Gianluca Palermo

Indice

1 Introduzione

L'obiettivo del progetto è quello di implementare un modulo hardware in VHDL che applica ad un flusso di parole il codice convoluzionale $\frac{1}{2}$ (ovvero ogni bit di una parola viene codificato con due bit). Dato il numero di parole che compongono il flusso, al componente viene richiesto di:

1. accedere ad un indirizzo della memoria RAM per recuperare le parole;
2. serializzare le parole, generando un flusso in ingresso continuo da 1 bit;
3. applicare sul flusso il codice convoluzionale $\frac{1}{2}$ fornito dalla specifica (riportato successivamente).
   Questa operazione genererà un flusso in uscita ottenuto dal concatenamento dei due bit in output;
4. parallelizzare, su 8 bit, il flusso continuo in uscita e scriverlo in memoria RAM.

L'implementazione deve essere in grado di gestire un segnale di reset. Per l'implementazione si è scelto di supporre il reset asincrono rispetto al segnale di clock.

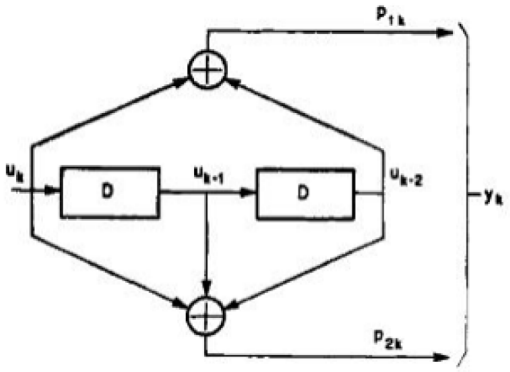

Figura 1: Codificatore convoluzionale

1.1 Esempio

Si riporta di seguito un esempio di codifica con il codificatore convoluzionale. Nella figura è riportata una memoria con indirizzamento a 16 bit. La lettura parte all'indirizzo 0, il quale contiene la lunghezza della sequenza in ingresso (nell'esempio 2 parole). Le parole che compongono il flusso di ingresso vengono lette a partire dall'indirizzo 1, mentre le parole che compongono il flusso di uscita si scrivono a partire dall'indirizzo 1000.
Sequenza di parole in ingresso: 01101001 10110101
Sequenza di parole in uscita: 00111010 00011111 10100010 10000100

Al termine della computazione la RAM è la seguente:

Indirizzo	Valore
0	2
1	105
2	181
[...]	
1000	58
1001	31
1002	162
1003	132

La computazione avvenuta è la seguente:
u_k è l'input dato al codificatore convoluzionale p_{1k} e p_{2k} sono rispettivamente il primo e il secondo output

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
u_k	0	1	1	0	1	0	0	1	1	0	1	1	0	1	0	1
p_{1k}	0	1	1	1	0	0	1	1	1	1	0	1	1	0	0	0
p_{2k}	0	1	0	0	0	1	1	1	0	0	0	0	0	0	1	0

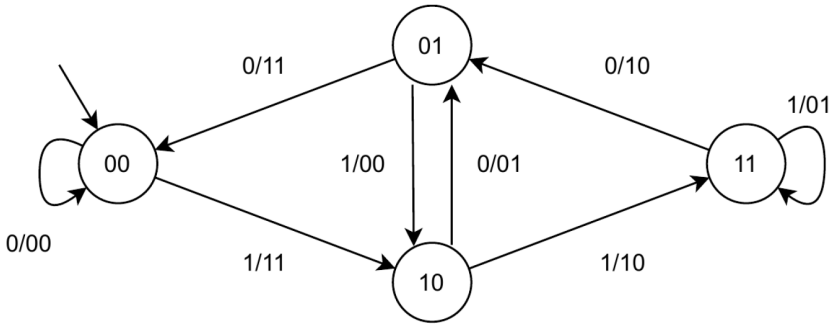

Figura 2: Macchina a Stati Finiti del codificatore convoluzionale

2 Architettura

L'architettura è stata progettata in maniera modulare, in modo da specializzare i singoli componenti creati. Di seguito si trova una spiegazione dei vari moduli.

2.1 Codificatore convoluzionale

Il codificatore convoluzionale è il modulo più interno che implementa la macchina a stati finiti in figura 2. Questo modulo è stato realizzato con specifica behavioural mediante tre processi: **state_reg**, **delta0** e **lambda0**. Il processo **state_reg** è il processo sequenziale che ha il compito di cambiare lo stato interno sul fronte di salita del clock. Il processo **delta0** è un processo combinatorio che calcola lo stato valido per il prossimo fronte di salita. Il processo **lambda0** è un processo combinatorio che calcola le uscite. Questo modulo riceve in ingresso quattro segnali e ha in uscita due segnali. Il segnale di ingresso `u` rappresenta la sequenzializzazione delle parole, mentre i due output che compongono il flusso in uscita sono `p1` e `p2`. In ingresso sono presenti anche i segnali di `stop` e di `reset`, Il segnale di `stop` serve a mantenere lo stato attuale della FSM in attesa che nel registro `i_word` venga caricata la parola successiva da computare. Il segnale di `reset` serve per resettare il codificatore convoluzionale nel momento in cui è terminata la codifica di un flusso e si deve cominciare la codifica del flusso successivo.

2.2 Datapath

Il datapath è l'insieme dei registri necessari per l'esecuzione delle operazioni. Di seguito vi è la lista dei vari registri e della loro funzione:

- `i_word`: nel momento in cui viene letta una parola, ad `i_word` viene assegnato il valore presente in `i_data`. Per eseguire un assegnamento al registro `i_word` viene posto il segnale `i_word_load` a '1'.
- `input`: si tratta del segnale di input del codificatore convoluzionale. E' composto da un singolo bit che viene selezionato attraverso un multiplexer con segnale di selezione `i_sel`.
- `output`: è il registro che mantiene il concatenamento di `output_1` e `output_2`, ovvero le uscite del codificatore convoluzionale.
- `o_word`: si tratta di un registro di 16 bit che contiene la concatenazione delle due parole che verranno scritte in memoria. Il suo contenuto è ottenuto a partire dal risultato del codificatore convoluzionale a cui viene dato in input la sequenzializzazione dalla parola nel registro `i_word`. Per poter parallelizzare il suo contenuto a 8 bit, viene utilizzato un multiplexer con segnale di selezione `d_sel`. Per `d_sel` = '1', a `o_data` vengono assegnati gli 8 bit più significativi di `o_word`, al contrario, per `d_sel` = '0', a `o_data` vengono assegnati gli 8 bit meno significativi.

- `end_addr`: contiene il valore della memoria RAM all'indirizzo 0 che rappresenta sia l'indirizzo al quale si trova l'ultima parola da leggere sia il numero di parole da leggere.

- `read_addr`: contiene l'indirizzo di memoria al quale si è arrivati con la lettura. Nel momento in cui si deve eseguire un'operazione di lettura, ad `o_addr` verrà assegnato il valore di `read_addr` attraverso il multiplexer con segnale di selezione `oa_sel`. Al termine dell'operazione di lettura, il valore di `read_addr` verrà incrementato di 1. Il valore di `read_addr` viene resettato ponendo a '0' il valore di `re_sel` e ponendo a '1' il valore di `read_addr_load`. Il valore di `read_addr` viene comparato al valore di `end_addr` e se questi due valori sono uguali, il segnale `o_end` viene posto a '1'.

- `write_addr`: questo segnale è analogo al segnale `read_addr` per l'operazione di scrittura. Per assegnare a `o_addr` il valore di `write_addr` per effettuare un'operazione di lettura, il segnale `oa_sel` viene posto a '1'. Al termine dell'operazione di lettura, il valore del registro viene incrementato di 1 attraverso il multiplexer con segnale di selezione `wr_sel` e ponendo il segnale di caricamento `write_addr_load` pari a '1'.

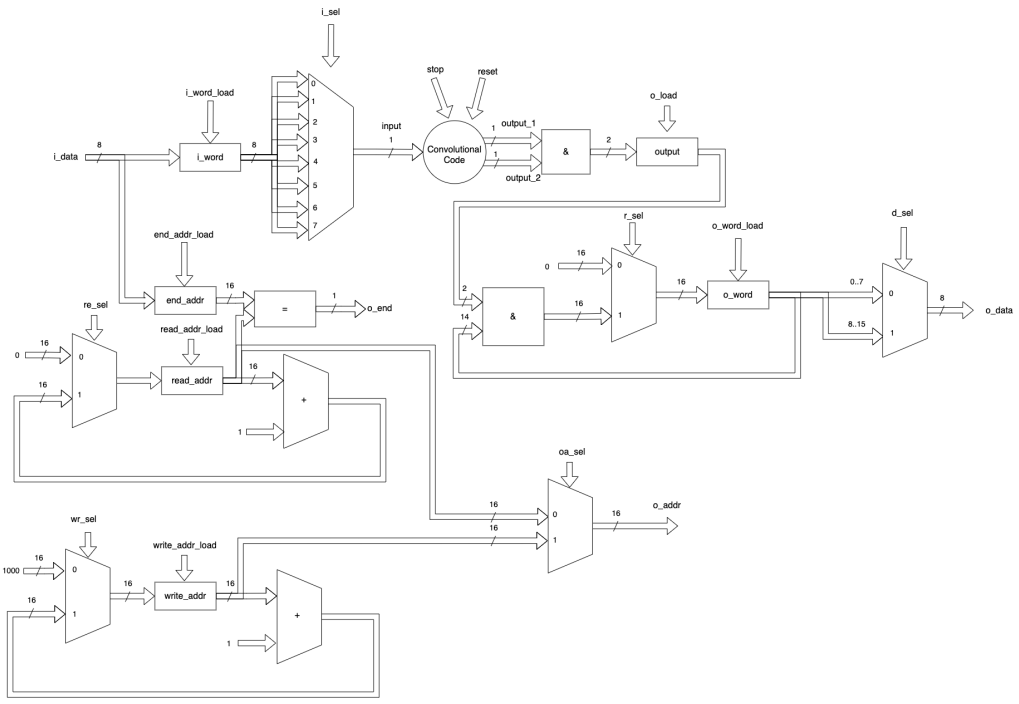

Figura 3: Datapath

2.3 Macchina a Stati Finiti

La FSM mantiene lo stato attuale della computazione e stabilisce il valore dei segnali di selezione e di caricamento del datapath e i segnali di output del componente project_reti_logiche. La FSM è stata realizzata con specifica behavioural mediante tre processi: **STATE_OUTPUT**, **DELTA1**, **LAMBDA1**.
STATE_OUTPUT è il processo sequenziale che ha il compito di asserire le uscite (`o_we` e `o_en` e i segnali di selezione `oa_sel` e `i_sel`) e di cambiare lo stato interno sul fronte di salita del clock. **DELTA1** e **LAMBDA1** sono invece i processi combinatori che calcolano rispettivamente lo stato e le uscite. Il suo schema in termini di diagramma degli stati ha 12 stati ed è rappresentato in figura 4. Segue una descrizione degli stati formali della FSM:

- `IDLING`: stato di idle in cui si posiziona la FSM al reset della computazione. La macchina aspetta che venga asserito il segnale `i_start`.
- `SET_READ_ADDR`: stato in cui vengono preparate le uscite per leggere l'indirizzo dell'ultima parola da leggere dalla RAM.
- `WAITING_ADDR`: stato in cui si aspetta un ciclo di clock per consentire alla RAM di asserire le sue uscite
- `READ_ADDR`: stato in cui si legge il dato dalla RAM e viene salvato l'indirizzo di fine lettura in `end_addr`
- `SET_READ_DATA`: stato in cui vengono preparate le uscite per leggere la parola all'indirizzo `read_addr` se `o_end` è '0'. Nel caso in cui `o_end` fosse pari a '1', si termina la computazione di questo flusso di parole e si passa allo stato di `END_WRITE`.
- `WAITING_DATA`: stato in cui si aspetta un ciclo di clock per consentire alla RAM di asserire le sue uscite
- `READ_DATA`: stato in cui si legge la parola dalla RAM e viene salvata in `i_word`.
- `GENERATE_WORD`: stato in cui si attende la generazione delle parole di uscita attraverso il codificatore convoluzionale.
- `END_GENERATION`: stato in cui termina la generazione delle parole e si leggono gli ultimi due output.
- `WRITE_FIRST_WORD`: stato in cui vengono preparate le uscite per scrivere la prima parola nella RAM.
- `WRITE_SECOND_WORD`: stato in cui vengono preparate le uscite per scrivere la seconda parola nella RAM.
- `END_WRITE`: stato in cui si asserisce `o_done` a '1' e si aspetta che i_start venga portato a '0' per riportarsi allo stato di `IDLING`.

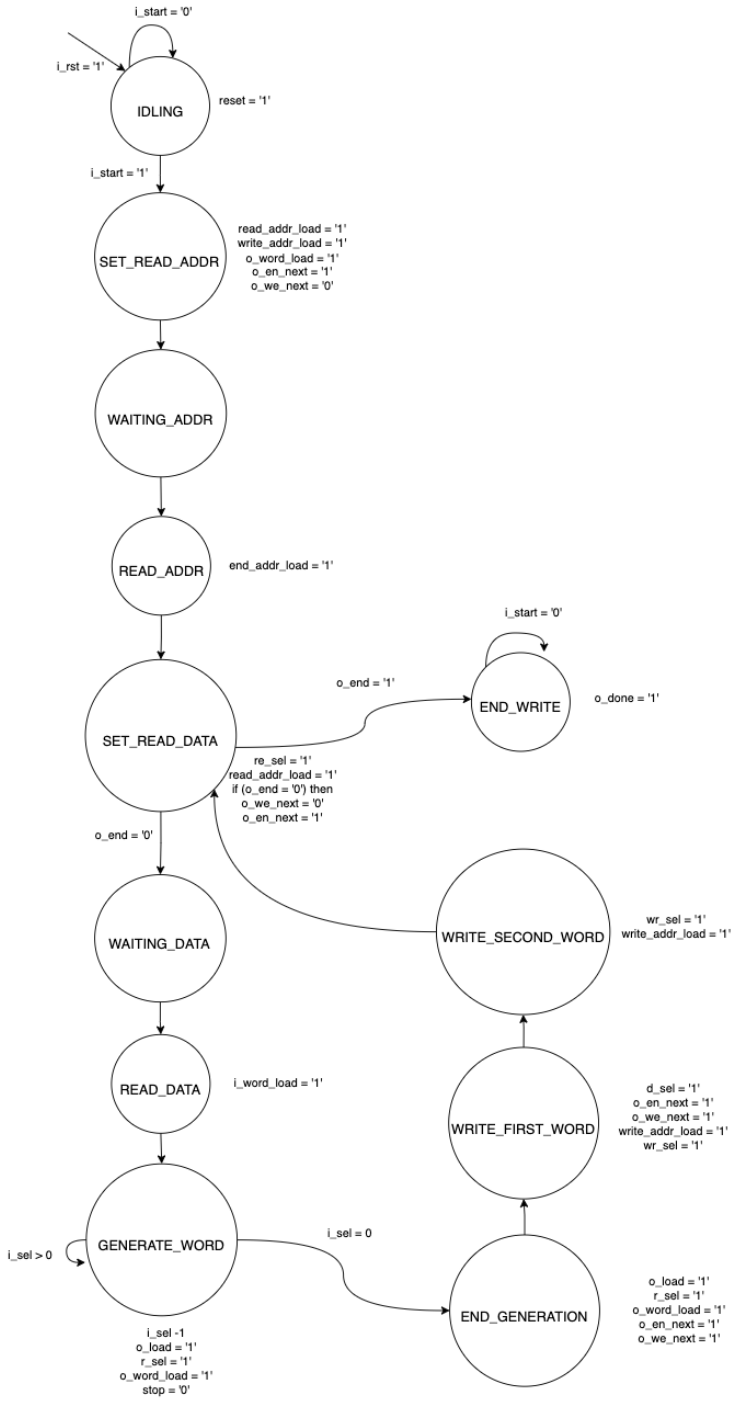

Figura 4: Macchina a Stati Finiti

3 Risultati sperimentali

3.1 Report di sintesi

La sintesi è stata effettuata con software Vivado 2021.2. Dal punto di vista dell'area la sintesi riporta il seguente utilizzo dei componenti:

- LUT: 98
- FF: 77
- F7 Mutex: 1

Si è fatta particolare cautela nella scrittura del codice per evitare utilizzo di latch.
Lo slack ottenuto è di 96.746 ns.

3.2 Simulazioni

Di seguito vengono riportare le simulazioni effettuate per verificare la correttezza dell'implementazione. Per tutti i test riportati è stata effettuata la simulazione behavioural e successivamente la simulazione functional, entrambe con successo.

3.2.1 Test Bench 1

Inizialmente sono stati utilizzati il test di prova fornito dal docente e un test simile descritto nella sezione esempio 1.1.

3.2.2 Test Bench 2

Questo test controlla che la computazione sia corretta nel caso in cui il numero di parole da leggere sia pari a zero, ovvero la sequenza di parole da computare abbia lunghezza nulla. In questo caso RAM(0) = "00000000".

3.2.3 Test Bench 3

Questo test controlla che la computazione sia corretta nel caso in cui la sequenza di parole da computare sia massima (ovvero pari a 255 parole). In questo caso RAM(0) = "11111111".

3.2.4 Test Bench 4

Questo test controlla che la computazione sia corretta nel caso in cui si richieda la codifica di più flussi uno dopo l'altro. In particolare sono stati fatti test su 2, 3 e 4 flussi successivi sulla stessa RAM.

3.2.5 Test Bench 5

Questo test controlla che la computazione sia corretta nel caso in cui si richieda la codifica di più flussi uno dopo l'altro. In particolare sono stati fatti test su 3 flussi successivi andando a modificare i valori nella RAM.

3.2.6 Test Bench 6

Questo test controlla che non avvengano errori nel caso in cui durante la computazione venga asserito il segnale di reset e venga fatta ripartire la computazione.

3.2.7 Test Bench 7

Sono stati generati casualmente vari test per coprire una varietà di casi.

4 Conclusioni

Si ritiene che l'architettura progettata rispetti le specifiche. Ciò è stato verificato mediate testing sia casuale che con test benches scritti manualmente. L'architettura inoltre presenta delle prestazioni considerevoli in quanto è possibile abbassare il ciclo di clock fino a un ordine di grandezza in meno rispetto a quello dato nella specifica, infatti lo slack è di 96.746 ns. Questo risulta un vantaggio considerando anche l'applicazione reale del codice convoluzionale, il quale viene utilizzato nelle telecomunicazioni per la correzione d'errore. Infatti, l'obiettivo è quello di ottenere un trasferimento di dati affidabile in numerose applicazioni come la telefonia mobile e la radio.
La scelta di un'architettura modulare e facilmente espandibile è un vantaggio in quanto consente di riutilizzare più volte lo stesso identico componente.