POLITECNICO
DI MILANO

Progetto finale di Reti Logiche

Andrea Ortenzi - 10660322

Marzo 2022

Contenuti

1 Introduzione

L'obiettivo del progetto è la realizzazione di un componente hardware che faccia da codificatore covoluzionale con tasso di trasmissione 1/2 e lavori con un periodo di clock di almeno 100 ns. Il convolutore deve leggere e scrivere i byte usando una RAM indirizzata al byte e deve poter gestire un segnale di reset.
Procedimento della macchina da realizzare:

1. Attendere che la sequenza sia pronta, quindi che il segnale di start si alzi
2. Leggere il byte 0 che contiene in notazione unsigned int il numero di byte (N) della sequenza da codificare.
3. Leggere gli N byte salvati in memoria partendo dall'indirizzo 1
4. Salvare gli N*2 byte generati dal codificatore a partire dall'indirizzo 1000
5. Alzare il bit done fino a quando il bit di start è a 1
6. Ripetere dal punto 1 per ogni sequenza da processare

1.1 Rappresentazione del convolutore come macchina a stati finiti

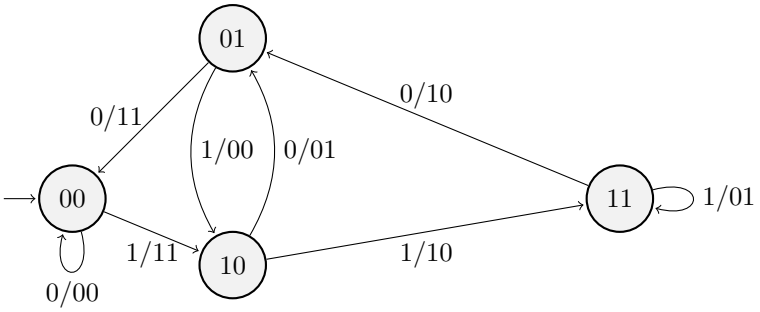


Figure 1: FSM codificatore covoluzionale

1.2 Esempio memoria a fine esecuzione

Indirizzo	Valore	Commento
0	2	Byte lunghezza sequenza di ingresso
1	162	Primo Byte sequenza da codificare
2	75	
	...	
1000	209	Primo Byte sequenza di uscita
1001	205	
1002	247	
1003	210	

Table 1: Memoria

2 Architettura

Sono state sviluppate due architetture, seriale e parallela ottimizzata.
La seriale non modifica il codificatore, prende quindi il byte dalla memoria e codifica un bit alla volta.
Per ottenere questo risultato si è utilizzato un segnale che identifica il bit da codificare durante quel ciclo di clock appartenente al byte precedentemente letto.
Il componente sviluppato è composto da un singolo processo che implementa una macchina a stati. Innestato in questa macchina a stati è presente il convolutore, implementato anch'esso come fsm.
Questa architettura non è ulteriormente approfondita perchè si sceglie di descrivere la versione ottimizzata. La macchina a stati è tuttavia rappresentata per visualizzare il confronto tra le due.

2.1 Rappresentazione componente seriale

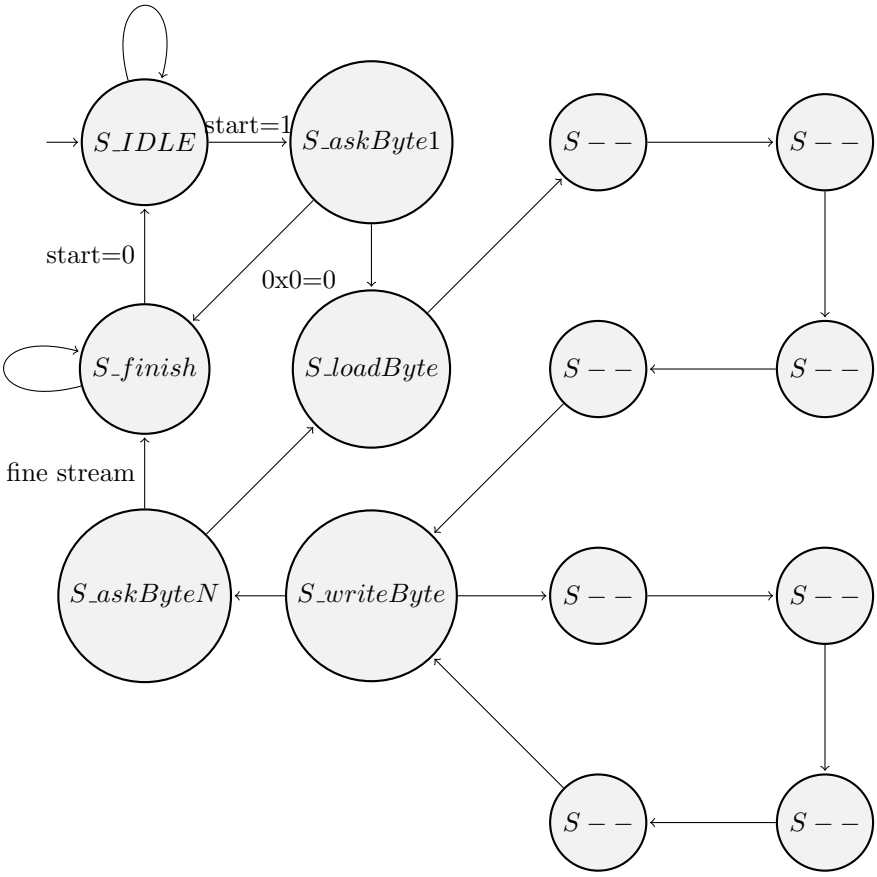
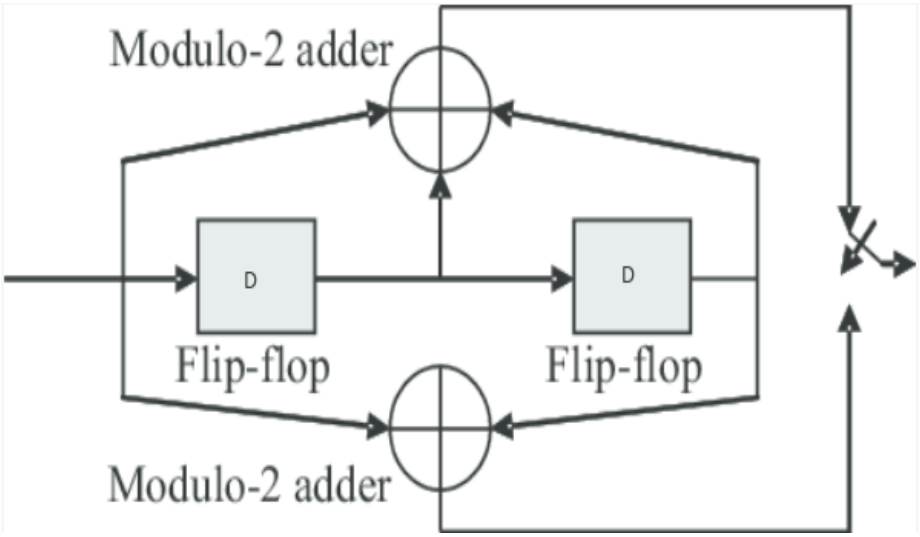


Figure 2: Componente sviluppato

*rappresento con S– un generico stato del convolutore (S00,S01,S10,S11)

2.2 Ottimizzazione architettura

Dopo uno studio del funzionamento del convolutore si osserva che, complicando leggermente la fase di encoding, risulta possibile codificare un intero byte in un unico ciclo di clock. La FSM rappresentata in figura 1 è infatti una rappresentazione del seguente schema logico:



Lo stato del codificatore è dovuto ad una sorta di shift register dato dai due FFD, quindi è possibile evitare di calcolare tutte le transizioni della sua FSM e impostarlo direttamente uguale agli ultimi due bit che questo processa.
Per trovare i due byte di uscita è stato utilizzato un maggior numero di porte xor in parallelo.

Si è quindi valutata la possibilità di sfruttare maggiormente la memoria incrementando gli accessi. Questa infatti ha un unico punto di accesso sia per la scrittura che per la lettura che però non è utilizzato durante l'encoding.
Si sceglie quindi di utilizzare quel ciclo di clock per leggere il byte successivo e farne l'encoding durante la scrittura dei byte generati dal primo, creando così una sorta di pipeline.

2.3 Pipeline

byte	clk 1	clk 2	clk 3	clk 4	clk 5	clk 6
byte dispari	Read	Encode	Write	Write	#	#
byte pari	#	Read	Encode	#	Write	Write

Table 2: Pipeline

* con # si indicano gli stalli.

2.4 Rappresentazione componente ottimizzato

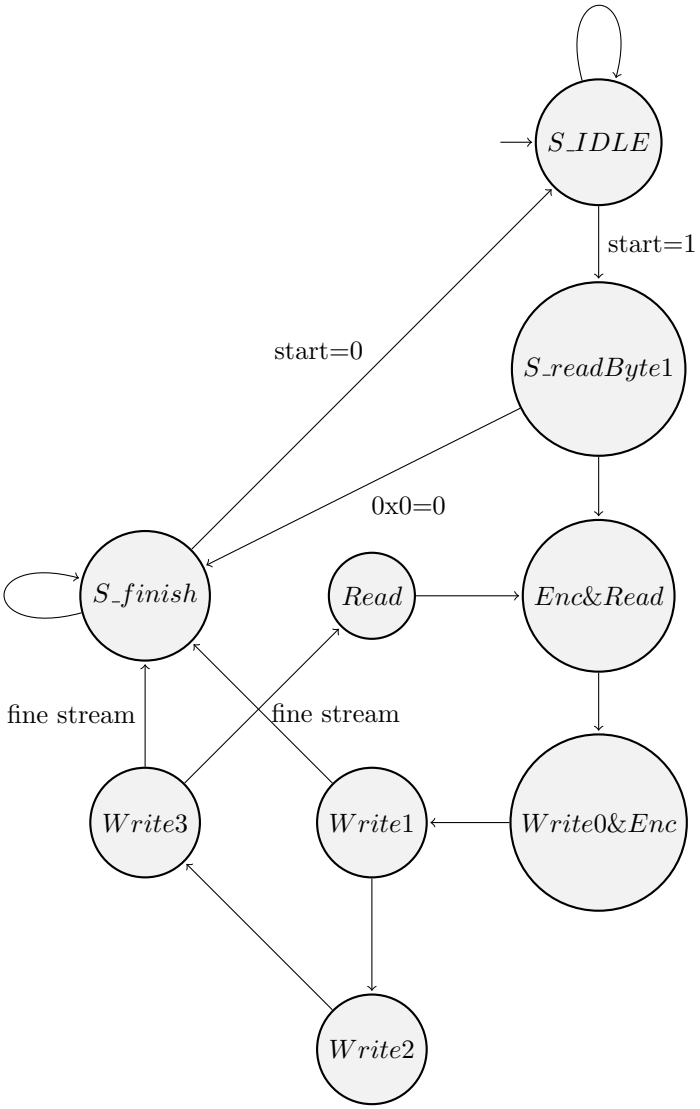


Figure 3: Componente ottimizzato

2.5 Descrizione codice VHDL componente

Per implementare l'ottimizzazione descritta (2.2) si è scelto di usare un solo processo con gli stati mostrati in figura 3. Si è preferito gestire il parallelismo della "pipeline" come singoli stati che eseguono due operazioni piuttosto che

due processi eseguiti in parallelo. In tal modo si evitano i latch, gli stalli e altre complicazioni per l'accesso alla RAM.

Durante lo sviluppo della versione ottimizzata del componente si è prestata particolare attenzione alle risorse utilizzate, cercando di minimizzare le variabili e i segnali utilizzati sia per quantità che per dimensione. Si riporta qui di seguito alcuni di questi interventi:

- E' stato eliminato il segnale che funzionava da contatore byte letti, il quale andava incrementato ogni volta e confrontato con il numero di byte da leggere. Per svolgere questa funzione si è scelto, invece, di usare un segnale che contenesse l'indirizzo dell'ultima cella di memoria da leggere. In tal modo non è necessario modificarne il valore durante la codifica e viene confrontato con il segnale *readAddress* che è già utilizzato per sapere quale byte leggere.
- La dimensione dei segnali *readAddress* e *writeAddress* è stata ridotta perchè il loro valore è limitato superiormente e non necessitano di 2 byte per la memorizzazione.
  In questa fase va prestata particolare attenzione ad eventuali overflow come riportato nella documentazione del codice.

2.5.1 Stati del processo del componente:

In questa descrizione non si descrive quello che è deducibile dalla rappresentazione della macchina a stati.
Inoltre si precisa che con il termine leggere da memoria ci si riferisce al processo di richiesta di un byte alla memoria tramite segnale enable, l'effettiva ricezione avverrà successivamente.

- **S_IDLE:** è lo stato da cui si esce solo quando la sequenza di byte da processare è pronta. Quando è pronto per iniziare legge il byte 0 dalla memoria.
- **S_readByte1:** viene salvata in *lastAddressToRead* la lunghezza dello stream di byte da processare, se questa non è nulla prosegue l'encoding leggendo il primo byte della sequenza.
- **PR:** viene processato il byte dispari. In questa fase il primo byte prodotto viene già scritto in *o_data* pronto per essere scritto in memoria, mentre il secondo viene salvato in *byteToWrite1*. In parallelo viene letto il byte pari se non è finita la sequenza.
- **WP:** si scrive il byte in *o_data* in memoria e, nel caso non fosse finito lo stream, si processa il byte pari generando altri due byte: *byteToWrite2* e *byteToWrite3*.
- **W1,W2,W3:** si scrivono i byte generati dal codificatore.

- **R:** si leggono dalla memoria i byte dispari.
- **S_finish:** si alza il bit done a 1 e si attendende che la memoria abbassi lo start.

3 Risultati sperimentali

3.1 Report di sintesi

Si è scelta per la sintesi una scheda Artix-7 AC701 Evaluation Platform (xc7a200tfbg676-2).
I risultati ottenuti in termini di componenti utilizzati sono riportati nella seguente tabella:

	Componente seriale		Componente ottimizzato	
Risorse	Stima utilizzo	% Utilizzata	Stima utilizzo	% Utilizzata
LUT	201	0.15	100	0.07%
FFF	129	0.05	86	0.03%
IO	38	9.50	38	9.50%
BUFG	1	3.13	1	3.13%

Table 3: Stima delle risorse necessarie

3.2 Schema RTL

Viene qui riportato lo schema RTL e non quello dell'implementazione perchè visivamente più immediato e perchè non era richiesta la sintesi e l'implementazione del componente su una scheda specifica.

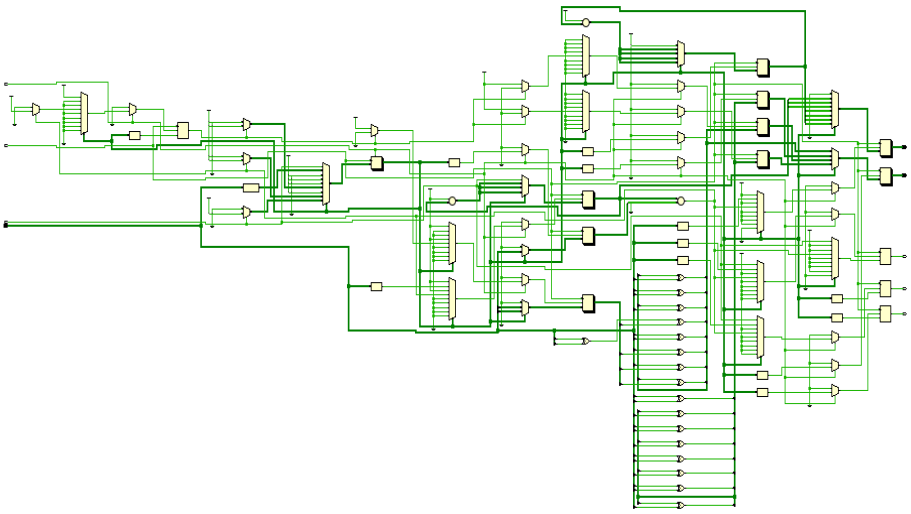


Figure 4: Schematic RTL componente ottimizzato

3.3 Simulazioni effettuate

1. Per vedere se il componente è corretto nei casi base si usano dei test bench brevi e senza segnale di reset iniziale.
   Questi testano il componente con una singola sequenza di byte e non controllano alcun caso limite.
   Sono molto utili per poter analizzare i problemi dal waveform viewer di Vivado.

   - **tb_esempio_1.vhd**
   - **tb_esempio_2.vhd**
   - **tb_esempio_3.vhd**

2. Per testare se il componente è in grado di gestire più letture consecutive senza reset si possono usare i seguenti test bench:

   - **tb_doppio_uguale.vhd:** Fa processare due volte gli stessi byte.
   - **tb_re_encode.vhd:** Fa processare tre sequenze di byte lunghe uguali
   - **tb_tre_bis.vhd:** Fa processare tre sequenze di byte diverse

3. Per verificare se il segnale di reset funziona correttamente si eseguono:

   - **tb_reset.vhd:** Codifica una singola sequenza di byte
   - **tb_tre_reset.vhd:** Codifica più sequenze di byte

4. Si verificano i casi limite sulla lunghezza della sequenza di byte da processare tramite altri due test bench.
   La lunghezza della sequenza è specificata nella RAM usando 8bit in notazione unsigned, quindi i casi limite da verificare sono 255 e 0.

   - **tb_seq_max.vhd:** Controlla una sequenza di 255 byte
   - **tb_seq_min.vhd:** Controlla una sequenza di 0 byte

4 Conclusioni

Si conclude che il componente sviluppato supera in maniera molto soddisfacente tutti i test bench sia in behavioral simulation che in post-sintesi con un periodo di clock di almeno 100ns, come richiesto dalla specifica.
Con l'ottimizzazione effettuata si è passati da 13 cicli di clock per codificare 1 byte, a 6 cicli di clock per processarne 2, raggiungendo così la massima velocità ottenibile con una memoria a singolo accesso e un periodo di clock fissato di 100ns. Modificando leggermente il test bench *tb_seq_max.vhd* si è potuto confrontare il numero di cicli di clock necessari al componente ottimizzato con quelli del seriale osservando un miglioramento di circa 4 volte (3062 contro 767 cicli di clock).