Prova Finale di Reti Logiche

Demis Selva

A.A. 2021-22

Matricola:	934172
Codice Persona:	10669443
Docente:	Gianluca Palermo

Indice

1 Requisiti del Progetto

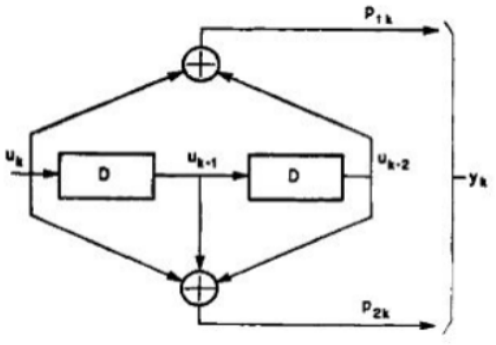


Figura 1: Codificatore convoluzionale con tasso di trasmissione $\frac{1}{2}$

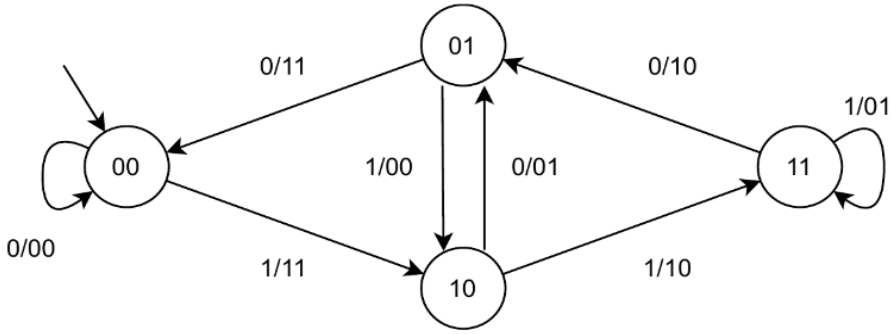
La specifica del progetto è ispirata a un **codificatore convoluzionale**, in particolare a uno con tasso di trasmissione $\frac{1}{2}$. I codici convoluzionali si utilizzano in numerose applicazioni allo scopo di ottenere un trasferimento di dati affidabile e sono spesso implementati in concatenazione. Al componente viene richiesto di:

1. Accedere al primo indirizzo della RAM per leggere la quantità di parole W da codificare (indirizzo 0), sapendo che la dimensione massima della sequenza di ingresso è 255 byte.
2. Accedere al successivo indirizzo per leggere la parole (da 8bit -1Byte-) che forma il flusso U.
3. Applicare la codifica convoluzionale $\frac{1}{2}$, generando quindi 2 parole in uscita nel flusso Z.
4. Scrivere i 2 risultati Z nella memoria RAM (partendo dall'indirizzo 1000).
5. Ripetere i passaggi dal 2 in poi, fino alla completa codifica di tutte le W parole di input in 2*W parole di output.

L'implementazione deve essere in grado di gestire un unico segnale di Reset iniziale. Per l'implementazione si è quindi scelto di supporre lo stato di Reset accessibile solo con un segnale esplicito di RESET, per poi usare uno stato IDLE di appoggio per reimpostare i valori iniziali prima delle (eventuali) successive codifiche. Il modulo deve anche essere in grado di codificare più flussi intervallati dal segnale di START.
L'implementazione è stata sintetizzata con target Artix-7 FPGA xc7a200tfbg484-1.
Viene riportata (nella pagina succerssiva) per riferimento la FSM data dalla specifica:

1.1 Esempio

Viene riportato l'esempio proposto nella specifica in quanto chiaro ed esaustivo sul funzionamento del modulo.
Un esempio di funzionamento è il seguente dove il primo bit a sinistra (il più significativo del BYTE) è il primo bit seriale da processare:



- BYTE IN INGRESSO = 10100010 (viene serializzata come 1 al tempo t, 0 al tempo t+1, 1 al tempo t+2, 0 al tempo t+3, 0 al tempo t+4, 0 al tempo t+5, 1 la tempo t+6 e 0 al tempo t+7) Applicando l'algoritmo convoluzionale si ottiene la seguente serie di coppie di bit:

```
T   0 1 2 3 4 5 6 7
Uk  1 0 1 0 0 0 1 0
P1k 1 0 0 0 1 0 1 0
P2k 1 1 0 1 1 0 1 1
```

Il concatenamento dei valori Pk1 e Pk2 per produrre Z segue il seguente schema: Pk1 al tempo t, Pk2 al tempo t, Pk1 al tempo t+1 Pk2 al tempo t+1, Pk1 al tempo t+2 Pk2 al tempo t+2, ... cioè Z: 1 1 0 1 0 0 0 1 1 1 0 0 1 1 0 1 - BYTE IN USCITA = 11010001 e 11001101 NOTA: ogni byte di ingresso W ne genera due in uscita (Z).

1.2 Ipotesi Progettuali

Si sono supposti veri i seguenti fatti:

1. La RAM mantiene il collegamento almeno fino al segnale di DONE, senza essere scollegata o disattivata durante la codifica.
2. Il programma è sintetizzato in modo da poter codificare più parole e/o flussi di parole mantenendo i vincoli di RESET imposti dalla specifica.
3. Durante la codifica potrebbe essere alzato di nuovo il segnale di RESET, in questo caso il modulo torna allo stato iniziale.

2 Architettura

È stata progettata un'architettura modulare per avere più componenti specifici e facili da adattare o modificare per eventuali variazioni future al codice. In questo modo le funzionalità di lettura, applicazione dell'algoritmo e scrittura sono tra loro separate.

2.1 Descrizione ad Alto Livello

Ad alto livello l'implementazione si comporta in questo modo:

1. Aspetta un segnale di RESET e imposta gli indirizzi e i segnali con i valori iniziali (es.: read a 0, write a 1000)
2. Appena riceve il segnale di START, legge il primo indirizzo e salva il numero delle parole
3. Legge il successivo indirizzo che contiene la parola da codificare
4. Inizia a processare la parola tenendo un contatore del numero di bit completati
5. Segue la FSM proposta dalla specifica e dopo aver processato tutti gli 8 bit tiene da parte la concatenazione degli output
6. Spezza in due l'output ottenuto, generando le 2 parole codificate
7. Scrive le parole in memoria una dopo l'altra, incrementando l'indirizzo di scrittura dopo ognuna
8. Incrementa il contatore delle parole completate e lo confronta con il numero W:
   (a) Se il contatore è minore, riparte dal punto 3;
   (b) Altrimenti, la codifica è terminata, si passa al punto 9.
9. Infine finalizza il processo alzando DONE e disabilitando la RAM, pronto per un eventuale nuovo segnale di START con cui riparte dal punto 2.

Per gestire questo algoritmo si è scelta un'implementazione costituita da una macchina a stati finiti, che sarà anche l'unico componente.

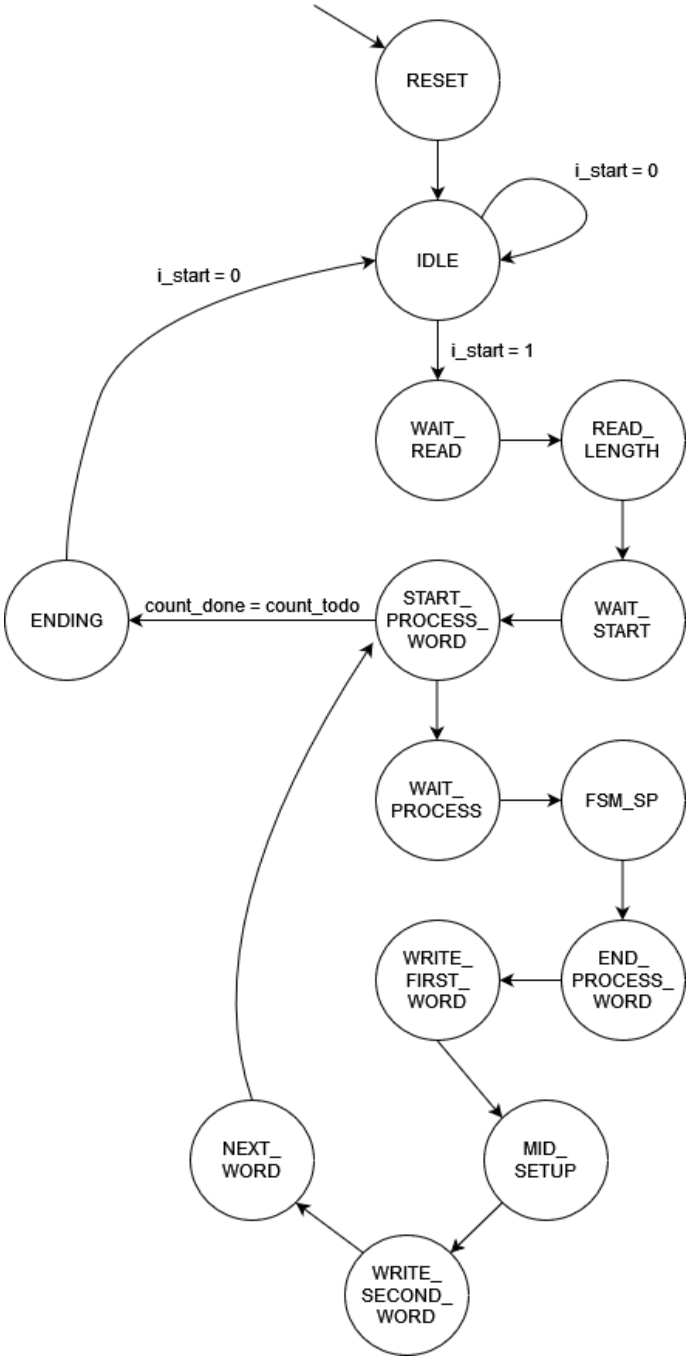
2.2 Macchina a Stati Finiti

La **Macchina a Stati Finiti** (FSM) è stata realizzata con specifica *Behavioural*. Segue una descrizione degli stati:

- `RESET`: Stato iniziale in cui si inizializzano tutti i valori a quelli iniziali della specifica (es.: read a 0, write a 1000).
- `IDLE`: Stato di riferimento per ogni inizio di codifica. La macchina aspetta il segnale alto di `i_start`, per poi reimpostare di nuovo i valori base e attivare la RAM ($o\_\ en$ -> 1).
- `WAIT_READ`: Stato in cui si aspetta che la RAM sia pronta per essere letta.
- `READ_LENGTH`: Stato in cui si legge il numero W di parole, salvato in $count\_\ todo$, e si sposta il riferimento di read all'indirizzo successivo.
- `WAIT_START`: Stato in cui si imposta l'indirizzo su quello di lettura (successivo).
- `START_PROCESS_WORD`: Stato in cui inizia la codifica: si fa prima un controllo per verificare se sono già state codificate tutte le parole: in caso affermativo si alza DONE e si passa alla finalizzazione; in caso negativo si procede.

- `WAIT_PROCESS`: Stato in cui viene letto U e si inizializza a 0 il numero dei bit processati fino a quel punto (*count_state*). Viene incrementato anche il riferimento read. Entra nella FSM data in specifica, all'ultimo stato incontrato (default S0 -> 00).
- `S0`, `S1`, `S2`, `S3`: Stati equivalenti alla FSM presente nella specifica (S0 = 00, S1 = 01, S2 = 10, S3 = 11). In ognuno si verifica il numero di bit processati, se sono già stati affrontati tutti, lo stato viene salvato come *last_state* da cui si riprenderà per le parole successive. Se invece non sono già stati processati 8 bit, si riempie sequenzialmente (2 bit alla volta) p_k, che alla fine conterrà entrambe le parole di output.
- `END_PROCESS_WORD`: Stato in cui si separa p_k nelle due parole di output. Si abilita la RAM per la scrittura e l'indirizzo di riferimento diventa quello di write
- `WRITE_FIRST_WORD`: Stato in cui si scrive la prima parola codificata, viene incrementato l'indirizzo per la scrittura.
- `MID_SETUP`: Stato di transizione in cui si imposta il successivo indirizzo per la scrittura.
- `WRITE_SECOND_WORD`: Stato in cui si scrive la seconda parola codificata, viene incrementato l'indirizzo per la scrittura.
- `NEXT_WORD`: Stato in cui si disabilita la scrittura su RAM, si imposta l'indirizzo di riferimento a quello di read e vengono resettati gli output. Si torna allo stato `START_PROCESS_WORD`.
- `ENDING`: Stato in cui si finalizza il processo, si aspetta che START venga riportato basso e si torna in `IDLE`.

Nella figura seguente è riportato l'automa implementato. Per semplicità e per evitare ripetizioni viene identificato come `FSM_SP` uno stato che comprende tutti quelli dell'automa della specifica (S0,S1,S2,S3)


RESET
IDLE
i_start = 0
i_start = 0
i_start = 1
WAIT_
READ
READ_
LENGTH
ENDING
count_done = count_todo
START_
PROCESS_
WORD
WAIT_
START
WAIT_
PROCESS
FSM_SP
WRITE_
FIRST_
WORD
END_
PROCESS_
WORD
NEXT_
WORD
MID_
SETUP
WRITE_
SECOND_
WORD

2.3 Schema dell'Implementazione

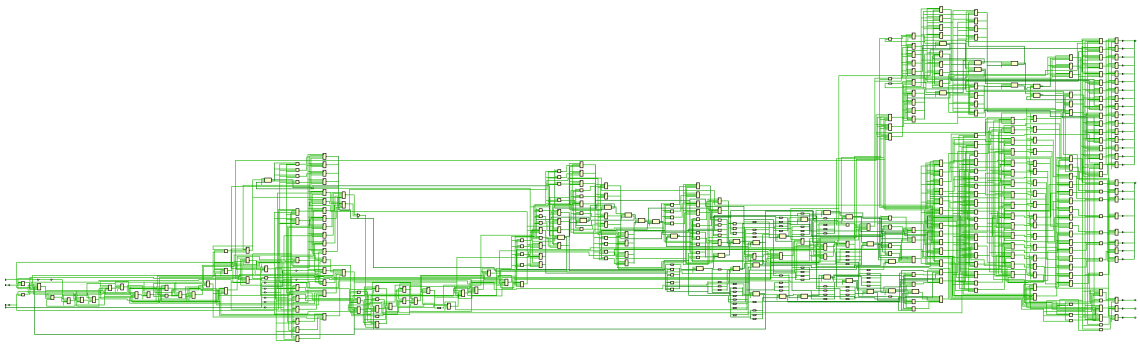


Figura 2: Schematic dell'implementazione

3 Risultati Sperimentali

3.1 Report di Sintesi

Il report di sintesi riporta il seguente utilizzo dei componenti:

- LUT: 213 (0.16%)
- FF: 156 (0.06%)
- IO: 38 (13.33%)

Si è prestata molta attenzione nella scrittura del codice per non utilizzare alcun Latch.

3.2 Simulazioni

I test effettuati erano mirati a presentare tutte le possibili situazioni critiche. Alcuni dei testbench usati sono:

- Un grande numero di test generati casualmente tramite un generatore in python, con lunghezza di input variabile.
- Caso specifico con lunghezza input = 0.
- Caso specifico con lunghezza input = 255.
- Caso specifico con più codifiche in successione (multistart).
- 3 Singoli test separati con lunghezza massima, minima e intermedia(casuale).

- Altri test casuali con periodo di clock variabile: 150ns, 100ns, 50ns, 10ns.

Durante la loro esecuzione sono state evidenziate diverse problematiche nel codice iniziale, che è stato man mano ritoccato fino a quando tutti i test funzionavano correttamente.

Per tutti i test è stata effettuata la simulazione behavioural e in seguito la simulazione functional post-synthesis, tutte con successo.

I risultati rilevanti sono riportati nella sezione 3.3.

3.3 Risultato dei Test Bench

Durante la fase di testing, come specificato prima, si è anche provato a modificare il periodo di clock, confermando che l'implementazione rispetta le specifiche funzionando a 100 ns. In particolare i test sono stati superati anche con periodi di clock molto minori, fino a 10 ns compreso (non sono stati testati periodi inferiori visto che la specifica temporale è ampiamente rispettata). Nel caso con periodo di clock = 10 ns, il report_timing menzionava uno slack di 3.338 ns.

Di seguito i risultati temporali legati a 3 casi di test (clock 100 ns)

Batteria da 5000 test con parametri casuali (Functional Post-Synthesis):	1 013 954 650 100 ps
Input Length 0, 2 test (Functional Post-Synthesis):	2 050 100 ps
Input Length 255, 2 test (Functional Post-Synthesis):	409 150 100 ps

Come si può notare dai risultati, la durata media dell'esecuzione nella batteria iniziale è (per il singolo test) 202 790 930 ps, mentre la durata media per test del terzo è 204 575 050 ps. Possiamo dedurre che il tempo per la codifica totale è mediamente stabile intorno al primo valore (avendo un gran numero di test con lunghezza variabile), con minimi di 1 025 050 ps (length 0) e massimi di 204 575 050 ps.

4 Conclusioni

Si può quindi affermare che il modulo progettato rispetta le specifiche, visti i risultati corretti in ognuno dei numerosi test casuali e specifici. La scelta di usare un'unica FSM ha anche permesso di non utilizzare Latch che avrebbero rischiato di portare a cicli infiniti.

Per quanto riguarda il design, si è scelto di utilizzare una singola FSM che include anche la FSM a 4 stati già presente nella specifica.