Prova finale di Reti Logiche

Luca Maestri

Virginia Longo

A. A. 2021-22

Matricole: 937019 – 937860

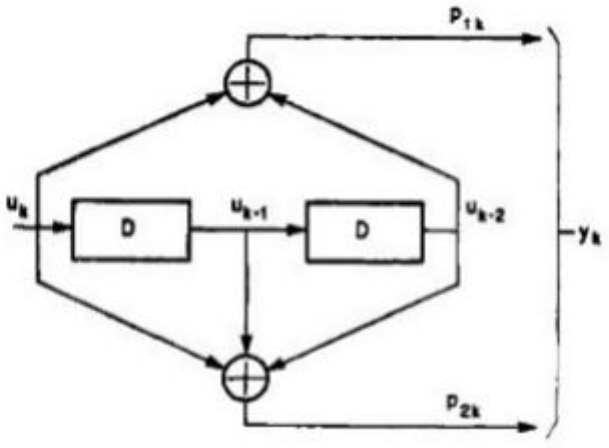
Codici Persona: 10702329 – 10685389

Docente: Fabio Salice

Requisiti del Progetto

La specifica della Prova finale (Progetto di Reti Logiche) 2022 è ispirata al metodo del convolutore, ovvero una macchina asincrona che riceve in ingresso una parola e ne produce due in uscita. Infatti il nome completo di tale metodo è: codificatore convoluzionale con tasso di trasmissione un mezzo.

Nel nostro caso, per ogni bit della parola in ingresso viene applicato il seguente procedimento:



In particolare viene richiesto:

1. Accedere alla RAM per leggere il numero di parole che il convolutore deve processare.
2. Leggere ogni parola della sequenza data.
3. Per ogni parola, più precisamente per ogni bit di essa, applicare il convolutore.
4. Scrivere le due parole ottenute in Memoria partendo dall'indirizzo 1000.

Inoltre, l'implementazione deve essere in grado di gestire un segnale di Reset. Tale segnale è asincrono, proprio come richiesto nelle specifiche.

Ipotesi di progetto

Si sono supposti veri i seguenti fatti:

- Il massimo numero di parole delle sequenza è 255.
- Ci possono essere zero parole.
- Il programma può rileggere la stessa sequenza più volte seguendo i criteri del reset.

Esempi

Un esempio di convolutore è il seguente:

Input in ingresso: 10100010

Output in uscita:11010001,11001101

Le stringhe ottenute con il seguente flusso temporale:

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

Più precisamente, troviamo i seguenti risultati ai seguenti indirizzi:

RAM:

0 : 00000001

1: 10100010

…

1000: 11010001

1001: 11001101

Implementazione

L'architettura è stata progettata in maniera modulare, in modo da specializzare i singoli componenti creati e separare le funzionalità di calcolo della codifica degli indirizzi con la gestione della macchina a stati finiti.

Descrizione ad alto livello

Pensando più ad alto livello, l'implementazione segue i seguenti passi:

1 Legge dall' indirizzo zero il numero di parole e lo salva
2 Controlla il numero di parole:
    - Se è zero , l'algoritmo termina.
    - Se è maggiore di zero, si prosegue al passo 3
3 Viene settato il contatore per determinare quante convoluzione andranno fatte.
4 Viene letta la parola da "processare"
5 Viene realizzata la convoluzione (tramite una FSM), inoltre viene anche preparato il prossimo bit da processare
6 Vengono salvati i due bit prodotti da convoluzione.
7 Si ripete dal punto 5 per un totale di 8 volte (una convoluzione per ogni bit), dopodichè si passa al punto 8.
8 Viene salvata la prima parola prodotta a partire dall'indirizzo 1000
9 Viene salvata la seconda parola prodotta a seguire dall'indirizzo dell'ultima parola salvata.
10 Viene fatto un controllo sul numero di parole
    - Se è maggiore di zero, si riprende dal punto 4
    - Se è uguale a zero si passa al punto 11
11 L'algoritmo termina finchè non viene richiesto (tramite un reset o un segnale di start) di ripartire dal punto 1.

Per gestire questo algoritmo si è usata una macchina a stati finiti, che rappresenta il top-level component e gestisce l'intero processo. Si noti come all'interno di tale macchina ne è presente un'altra che gestisce solo la parte ci convoluzione.

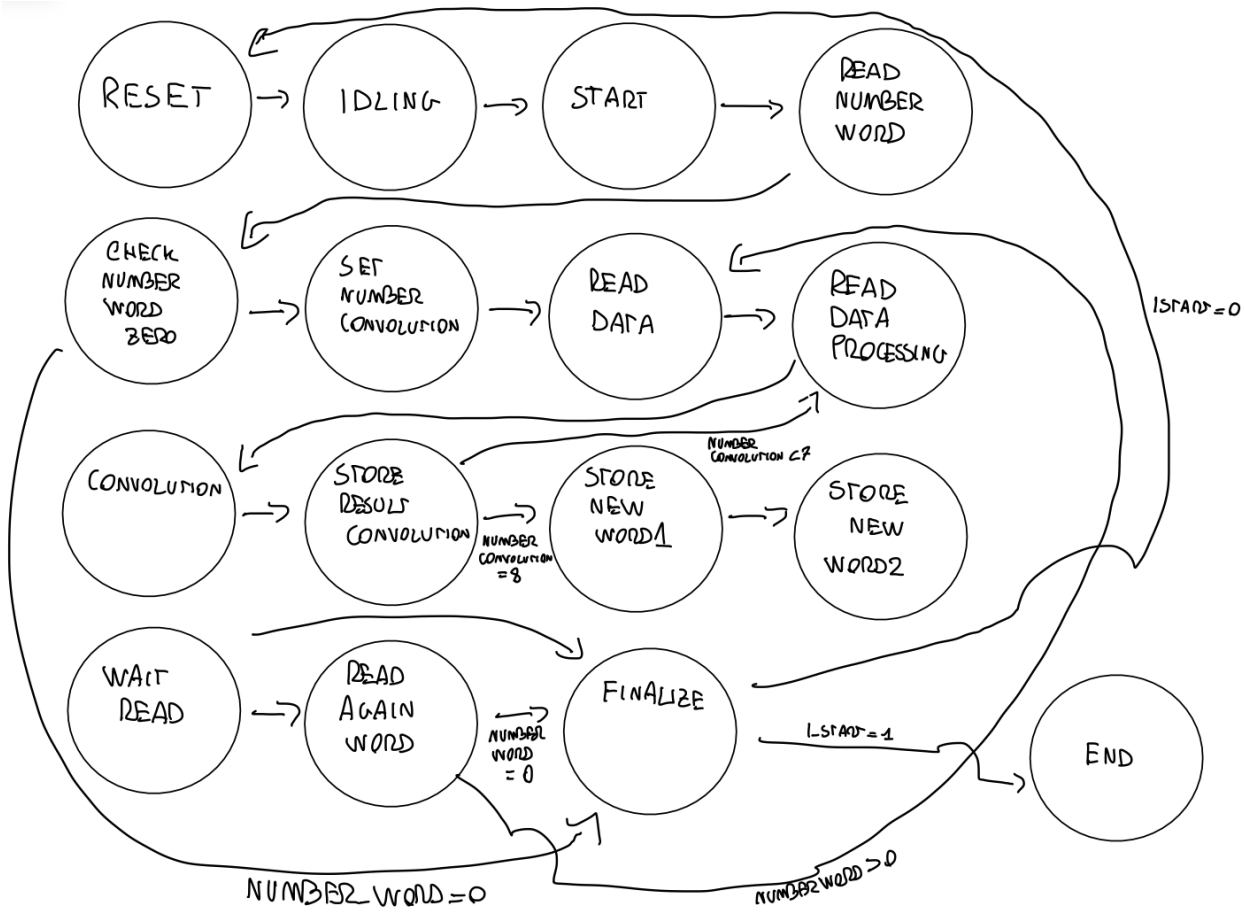
Macchina a stati finiti

E' stata realizzata con specifica Behavioural. A seguire troviamo una descrizione degli stati di tale FSM:

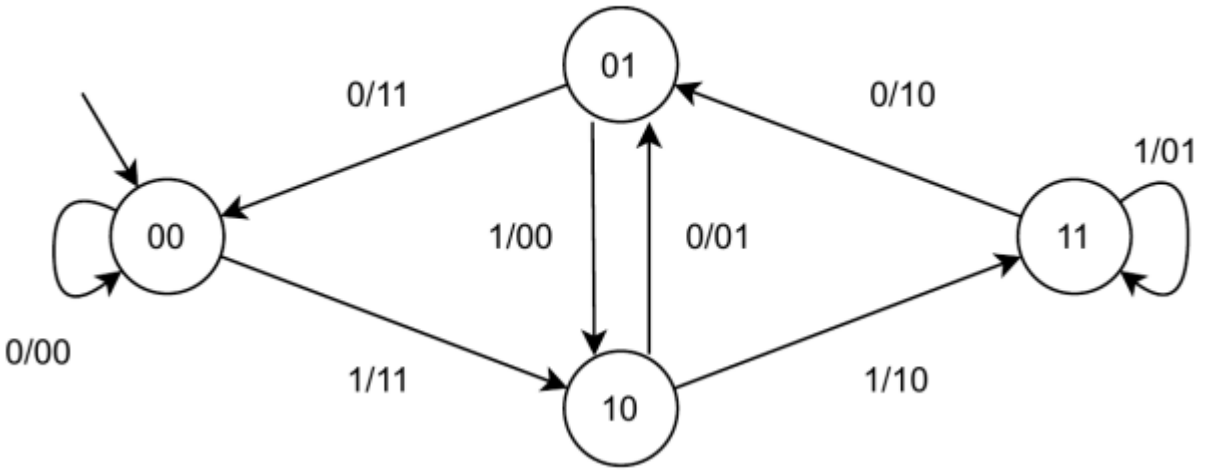
- RESET: stato di idle in cui si posizione la FSM al reset della computazione attendendo che vengo asserito il segnale i_start.
- IDLING: stato di intramezzo che consente alla macchina portesi avviare.
- START: stato di inizio, viene preparata la memoria in lettura

- READ_NUMBER_WORD: stato in cui viene letto il numero di parole della sequenza
- CHECK_NUMBER_WORD: stato in cui viene confrontato il numero di parole da leggere, se è zero, si passa direttamente allo stato finalize, altrimenti si prosegue.
- SET_NUMBER_CONVULUTION: stato in cui viene resettato il numero di convoluzioni
- READ_DATA_AGAIN_ stato che riprepara la memoria a leggere una nuova parola
- READ_DATA: stato in cui viene letta la parola di cui fare la convoluzione
- READ_DATA_PROCESSING: stato che serve a permettere che la lettura della parola sia ultimata(questo è dovuto ai ritardi).
- CONVULUTION: stato in cui viene preparato il prossimo bit tramite una moltiplicazione per due della parola letta (in origine si era pensato ad uno switch, ma per velocizzare l'esecuzione si è optato per questa seconda opzione). Inoltre viene applicata la convoluzione al bit in prima posizione della parola.
- STORE_RESULT_CONVULUTION: stato in cui in base al numero di convoluzioni vengono sovrascritti due bit della parola (parola1 se stiamo convolulando i primi 4 bit della parola, parola2 altrimenti).
- STORE_NEW_WORD_1: stato in cui viene salvata la prima parola prodotta
- STORE_NEW_WORD_2: stato in cui viene salvata la seconda parola prodotta
- WAIT_READ: stato in cui viene controllato se ci sono ancora parole da leggere.
- FINALIZE: stato in cui se i_start è alto, allo alza anche o_done, altrimenti pone o_done basso e si rimette nello stato di reset.

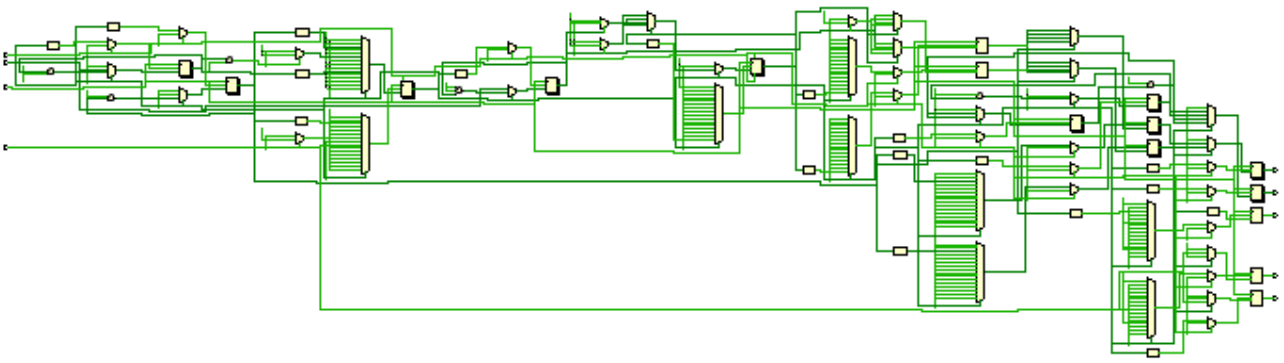
Disegno della FSM



Disegno della FSM del convolulatore (già fornita)



Schema dell'implementazione



Test benches

I test effettuati hanno cercato di effettuare transizioni critiche oppure di verificare possibili configurazione di memoria estreme. Sono stati creati dei test bench che coprissero questi casi:

- Varie parole generate casualmente
- Un test con 255 parole
- Un test con 0 parole
- Rilettura della memoria più volte
- Utilizzo del reset

Oltre a questi test si sono testati anche alcuni forniti dai docenti, superandoli sia in behavioural sia in post sintesi (timing e functional).

Risultati sperimentali

Dal punto di vista dell'area la sintesi riporta il seguente utilizzo dei componenti:

- LUT: 142
- FF: 105

Durante la scrittura del codice si è fatta molta attenzione per evitare utilizzo di Latch.

Risultato dei test bench

Lavorando sul testing si sono provati vari periodi di clock. Il progetto rispetta le specifiche, passando i test con almeno un clock di 100 ns.

Si riportano i risultati di tempo relativi ai test che sono stati più ardui da superare:

- Tb_doppio_uguale (Functional Post-Synthesis) : 15550100 ps
- Tb_reset (Functional Post-Synthesis) : 16850100 ps
- Tb_seq_max (Functional Post-Synthesis) : 587550100 ps

Si noti come l'ultimo caso richieda molto più tempo degli altri. Tuttavia l'aumento non è così significativo se si pensa che vengono processate almeno 200 parole in più. Ciò fa comprendere come l'architettura funzioni in modo più efficiente quando ci sono casi di reset.

Inoltre per poter rispettare il clock, il caso con 255 parole ha costretto a dei cambiamenti nell'architettura, in particolare nella scrittura del codice, al fine di poter velocizzare il processo:

- La prima versione prevedeva che al convolutore fosse passato l'i-esimo bit della parola tramite il seguente case:

```
case counter_convulution is
        when 0 =>
           bit_process <= word(7);
        when 1 =>
           bit_process <= word(6);
        when 2 =>
           bit_process <= word(5);
        when 3 =>
           bit_process <= word(4);
        when 4 =>
           bit_process <= word(3);
        when 5 =>
           bit_process <= word(2);
        when 6 =>
           bit_process <= word(1);
         when 7 =>
           bit_process <= word(0);
        when others =>
        end case;
```

  Si evidenzia come si sfruttasse il contatore del numero di convoluzioni per decidere l'iesimo bit.
  Tuttavia tale stato era molto lento, per cui si è optato per un ragionamento diverso:
  prendere sempre il bit in posizione 7 e moltiplicare per 2 la parola letta, in questo modo tutti i bit della parola finiranno in tale posizione.
  Si e scoperto che questo algoritmo ha velocizzato il tempo di esecuzione in modo significativo.

Conclusioni

Si ritiene che l'architettura progettata rispetti anzitutto le specifiche, fatto che è stato verificato sia tramite i test forniti sia tramite test benches manualmente scritti.

Sfruttando la FSM, l'architettura evita la creazione di possibili latch, evitando così possibili cicli infiniti.

Per la creazione dell'architettura è stato fondamentale accettare che la prima soluzione funzionante non è per forza la migliore. Si può sempre trovare un algoritmo migliore.