# Prova Finale - Progetto di Reti Logiche

Nicolas Benatti (10668784), Francesco Barisani (10667413)

Docente: William Fornaciari
A.A. 2021/2022

POLITECNICO
MILANO 1863

# Indice


**1 Introduzione** **2**
- 1.1 Scopo del progetto . . . 2
- 1.2 Schema di alto livello . . . 2
- 1.3 Algoritmo di convoluzione . . . 2
- 1.4 Interfaccia VHDL del componente . . . 3
- 1.5 Descrizione della memoria . . . 3
- 1.6 Esempio di esecuzione e contenuto della memoria . . . 4

**2 Architettura** **5**
- 2.1 Datapath . . . 5
- 2.2 Codificatore convoluzionale . . . 5
- 2.3 Program Counter . . . 6
- 2.4 `R0`, `R1`, `R2`: registri a caricamento parallelo . . . 6
- 2.5 `R3`, `R4`: registri parallelo-serie con enable . . . 6
- 2.6 `R5`: registro serie-parallelo . . . 7
- 2.7 Macchina a stati . . . 7

**3 Risultati sperimentali** **8**
- 3.1 Sintesi . . . 8
  - 3.1.1 Report di sintesi . . . 8
  - 3.1.2 Report dei tempi . . . 8
- 3.2 Simulazioni e Test . . . 9
  - 3.2.1 Testbench #1 (fornito dal docente) . . . 9
  - 3.2.2 Testbench #2 (conversioni multiple) . . . 9
  - 3.2.3 Testbench #3 (reset durante l'elaborazione) . . . 9
  - 3.2.4 Testbench #4 ($W = 0$) . . . 9
  - 3.2.5 Testbench #5 ($W = 255$) . . . 9
  - 3.2.6 Testbench #6 (vincoli tra segnali) . . . 9

**4 Conclusioni** **10**
- 4.1 Scelte di progetto . . . 10
- 4.2 Valutazioni di carattere generale . . . 10

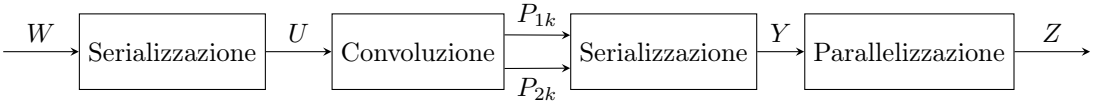
# 1 Introduzione

## 1.1 Scopo del progetto

È richiesto di implementare un componente hardware che esegue l'operazione di codifica convoluzionale di una sequenza di Byte, interfacciandosi con una memoria nella quale sono memorizzati i dati in ingresso e nella quale verrà scritto il risultato prodotto in uscita.
I codici convoluzionali sono una tecnica di correzione d'errore ampiamente usata nel campo delle telecomunicazioni: dalle comunicazionni cellulari (GSM, 3G, ...), trasmissioni satellitari (DVB-S) e reti WLAN (IEE802.11).
Nel dominio di competenza del nostro progetto, tale codificatore convoluzionale dovrà leggere dalla memoria una sequenza di $W$ Byte, produrre una sequenza d'uscita di $Z$ Byte e scriverla in memoria. La cardinalità $W$ è assegnata in modo arbitrario e viene letta dalla memoria, mentre il numero di Byte d'uscita è legato a $W$ dal tasso di trasmissione del codice utilizzato (vedi Sezione 1.3).

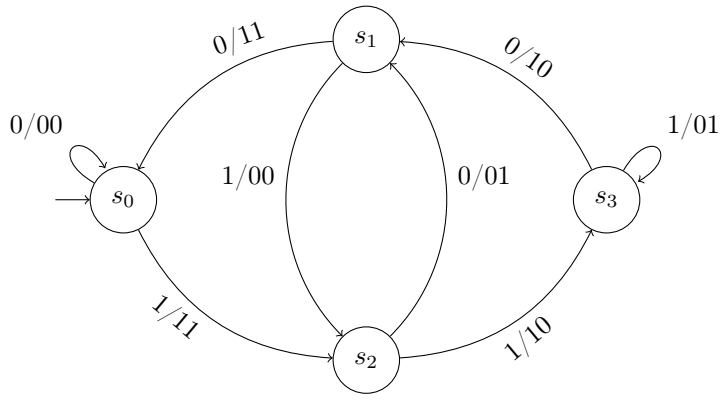
## 1.2 Schema di alto livello

Di seguito riportiamo un block diagram che mostra ad alto livello gli stadi dell'elaborazione:




## 1.3 Algoritmo di convoluzione

Verrà impiegato un codificatore convoluzionale con rapporto di trasmissione $\frac{1}{2}$. L'algoritmo che collega logicamente un dato ingresso all'uscita prodotta da tale codificatore è descritto dalla seguente macchina a stati:



Ogni transizione della macchina è annotata come $u_k/P_{1k}, P_{2k}$
Per ogni bit in ingresso, ne vengono prodotti 2 in uscita; di conseguenza il numero di Byte d'uscita sarà $Z = 2W$.

## 1.4 Interfaccia VHDL del componente

```
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;
```

Di seguito una breve descrizione dei segnali:

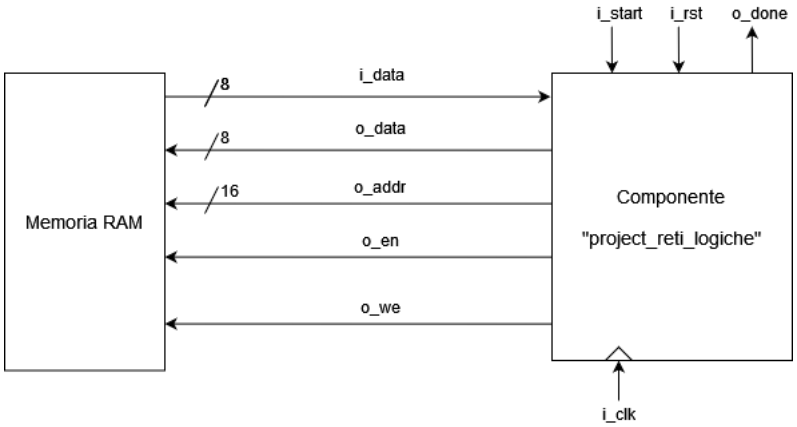
- `i_clk`: Segnale di clock di sistema.
- `i_rst`: Segnale di reset di sistema.
- `i_start`: Segnale che stabilisce l'inizio dell'elaborazione da parte del componente.
- `i_data`: Byte letto dalla memoria.
- `o_address`: Indirizzo di scrittura, in uscita dal componente.
- `o_done`: Segnala la fine dell'elaborazione.
- `o_en`: Segnale di abilitazione operazioni in memoria.
- `o_we`: Abilitazione della scrittura in memoria.
- `o_data`: Byte d'uscita da scrivere in memoria.

## 1.5 Descrizione della memoria

La memoria in uso è indirizzata al Byte.
Inoltre, si tratta di una memoria sincrona, di conseguenza il dato sarà disponibile 1 ciclo di clock dopo aver depositato l'indirizzo sul bus (indirizzi riportati in decimale per semplicità).

| Address [16 bit] | Value [8 bit] |
|---|---|
| 0 | $W$ (lunghezza sequenza in ingresso) |
| 1 ... $W$ | i-esima parola in ingresso |
| [...] | |
| 1000 ... (1000+$Z$-1) | parole di uscita |

Di seguito mostriamo come il componente comunica con la memoria:




## 1.6 Esempio di esecuzione e contenuto della memoria

Esempio con sequenza di lunghezza 3:

$W$: 01110000 10100100 00101101
$Z$: 00111001 10110000 11010001 11110111 00001101 00101000

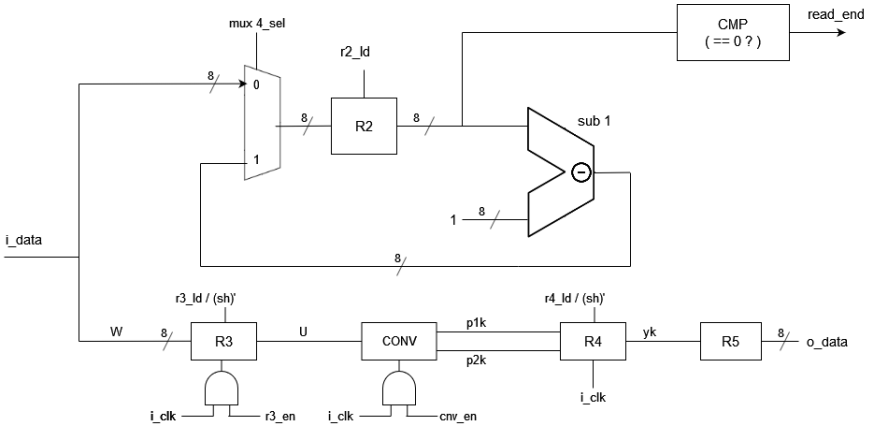
Per semplicità, riportiamo nella tabella valori ed indirizzi in decimale, anche se nella memoria vengono effettivamente memorizzati come valori binari `unsigned` su 8 bit.

**Contenuto della memoria dopo l'esecuzione**

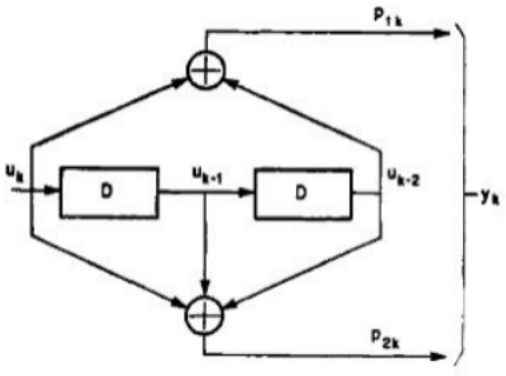
| Indirizzo | Valore | Semantica |
|---|---|---|
| 0 | 3 | lunghezza sequenza in ingresso ($W$) |
| 1 | 112 | primo byte sequenza da codificare |
| 2 | 164 | |
| 3 | 45 | |
| [...] | [...] | |
| 1000 | 57 | primo byte sequenza di uscita |
| 1001 | 176 | |
| 1002 | 209 | |
| 1003 | 247 | |
| 1004 | 13 | |
| 1005 | 40 | |

# 2 Architettura

## 2.1 Datapath



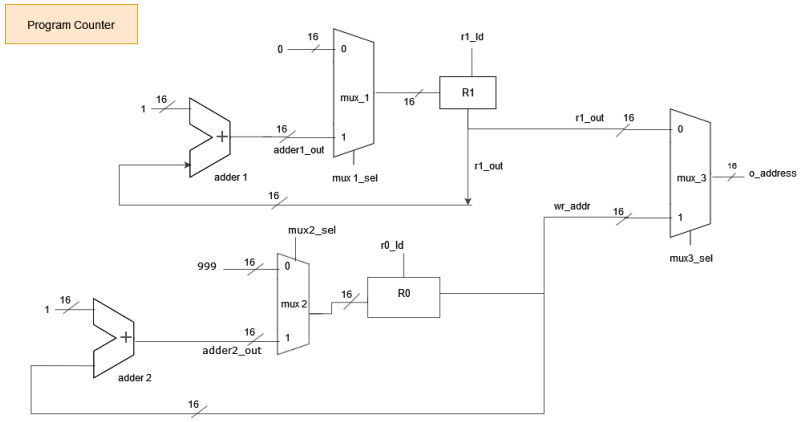

## 2.2 Codificatore convoluzionale




Il componente prende in ingresso il bit del flusso serializzato $u_k$ e produce in uscita i 2 bit $P_{1k}$ e $P_{2k}$ (vedi sezione 1.2). È composto da 2 registri da 1 bit e 2 sommatori.

## 2.3 Program Counter

Il program counter è composto da 2 registri separati, uno per memorizzare l'indirizzo corrente di lettura e uno per la scrittura, che possono essere fatti avanzare indipendentemente tramite segnali di `load` separati.




## 2.4 `R0`, `R1`, `R2`: registri a caricamento parallelo

Sono registri "standard", la cui funzione è esclusivamente quella di memorizzare dati provenienti dalla memoria o da sommatori, in particolare:

- `R0` memorizza l'indirizzo di scrittura.
- `R1` memorizza l'indirizzo di lettura.
- `R2` memorizza il primo Byte letto, ossia il valore di $W$, per tenere traccia di quanti Byte rimangono da leggere tramite un sottrattore in retroazione (vedi Sezione 2.1).

## 2.5 `R3`, `R4`: registri parallelo-serie con enable

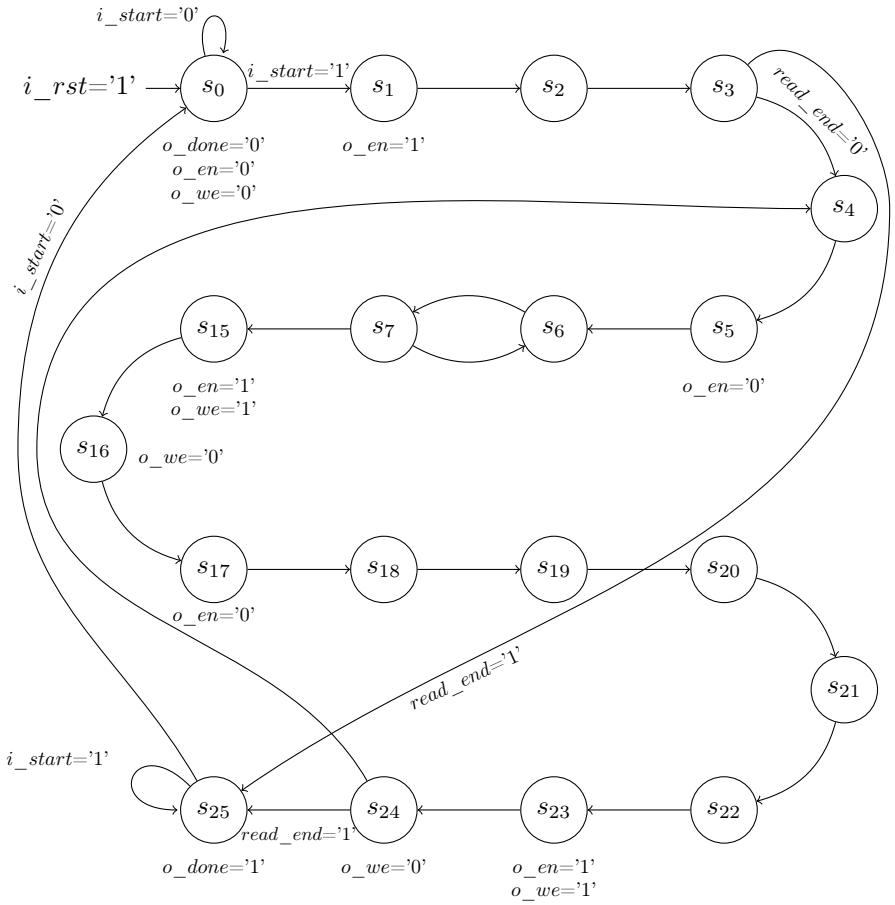
Questi registri si occupano, rispettivamente, di serializzare il Byte del flusso $W$ in ingresso (producendo il flusso $U$) e di serializzare l'uscita del convolutore producendo il flusso $Y$.
`R3` è inoltre dotato di un segnale di enable che permette di mantenere l'uscita stabile mentre `R4` serializza l'uscita del convolutore.
Tali registri sono stati modificati rispetto alla realizzazione circuitale vista a lezione, per consentire di produrre in uscita i bit dal più al meno significativo (con ripetizione dell'LSB). Nel datapath, il sistema di enable è rappresentato tramite *gating* del segnale di clock.

## 2.6 R5: registro serie-parallelo

Questo registro si occupa di "raccogliere" i bit del flusso $y_k$ per produrre i Byte di uscita $Z$. La macchina a stati si occuperà di abilitare la scrittura in memoria non appena un Byte sarà pronto in uscita al registro.

## 2.7 Macchina a stati

Nel seguente *pallogramma* sono riportati solo i segnali di interfaccia verso l'esterno, nei momenti in cui il loro valore varia rispetto allo stato precedente.



Di seguito una breve descrizione degli stati:

- $s_0$: stato di reset.
- $s_6$,$s_7$: produzione del primo Byte d'uscita
- $s_{15}$...$s_{16}$: scrittura del primo Byte avanzamento dell'indirizzo di scrittura.
- $s_{17}$...$s_{23}$: produzione del secondo Byte d'uscita.
- $s_{24}$: scrittura del secondo Byte d'uscita.
- $s_{25}$: comunicazione di fine elaborazione.

# 3 Risultati sperimentali

## 3.1 Sintesi

### 3.1.1 Report di sintesi

Di seguito è riportato il risultato del report di sintesi:

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice LUTs* | 75 | 0 | 0 | 134600 | 0.06 |
| LUT as Logic | 75 | 0 | 0 | 134600 | 0.06 |
| LUT as Memory | 0 | 0 | 0 | 46200 | 0.00 |
| Slice Registers | 67 | 0 | 0 | 269200 | 0.02 |
| Register as Flip Flop | 67 | 0 | 0 | 269200 | 0.02 |
| Register as Latch | 0 | 0 | 0 | 269200 | 0.00 |
| F7 Muxes | 0 | 0 | 0 | 67300 | 0.00 |
| F8 Muxes | 0 | 0 | 0 | 33650 | 0.00 |

Il componente è correttamente sintetizzabile ed è composto da 75 Look-up Tables (`LUT`) (memorie ROM) e 67 Flip-Flop (`FF`). Nessuna ROM è realmente adibita alla memorizzazione di dati, ma piuttosto alla codifica di funzioni logiche.

### 3.1.2 Report dei tempi

Le voci di maggior interesse di questo report di sintesi sono lo *Slack* e il *Data Path Delay*, nel nostro caso:

```
Slack (MET) : 96.807ns
Data Path Delay : 3.075ns  (logic 1.853ns (60.260%)  route 1.222ns (39.740%))
```

Su un periodo di clock $T_{clk} = 100ns$.
Sulla base di questi dati, possiamo calcolare la massima frequenza alla quale può funzionare il nostro componente, a causa del ritardo introdotto dalla logica del datapath:

$$f_{max} = \frac{1}{T_{delay}} = \frac{1}{3.075ns} = 325.2MHz$$

## 3.2 Simulazioni e Test

### 3.2.1 Testbench #1 (fornito dal docente)

Questo è il testbench più semplice di nostro interesse: dopo aver resettato il componente alzando il segnale `i_rst`, si avvia la conversione alzando `i_start`, per poi attendere il momento in cui il componente alzerà il segnale `o_done`. Infine, si attende che il componente abbassi il segnale `o_done`, dato che, da specifica, non è possibile cominciare un nuovo processo di conversione fino a quando ciò non accade.

### 3.2.2 Testbench #2 (conversioni multiple)

In questo testbench abbiamo messo alla prova la capacità del componente di effettuare più di una conversione di fila all'altra. Per attuare tale test è stato sufficiente ripetere 2 volte la descrizione VHDL del testbench precedente, senza però mandare un segnale di reset tra una conversione e l'altra.

### 3.2.3 Testbench #3 (reset durante l'elaborazione)

Per valutare l'efficacia del segnale di reset, abbiamo testato il componente a fronte di un reset nel mezzo della fase di elaborazione. L'elaborazione viene correttamente abortita e quella successiva va a buon fine.

### 3.2.4 Testbench #4 ($W = 0$)

Con questo test abbiamo verificato il comportamento del componente nel caso limite in cui non ci sia alcun Byte da convertire. Come previsto, il componente reagisce in maniera adeguata, ovvero alzando il segnale di terminazione `o_done` non appena il valore di $W$ arriva in `R2`.

### 3.2.5 Testbench #5 ($W = 255$)

Il caso limite duale a quello dell'assenza di Byte in ingresso è quello dove il numero di Byte da elaborare è il massimo possibile.

### 3.2.6 Testbench #6 (vincoli tra segnali)

In questo testbench testiamo il vincolo temporale tra il segnale `i_start` e `o_done`, come riportano le specifiche:

> «Il segnale DONE deve rimanere alto fino a che il segnale di START non è riportato a 0. Un nuovo segnale start non può essere dato fin tanto che DONE non è stato riportato a zero. Se a questo punto viene rialzato il segnale di START, il modulo dovrà ripartire con la fase di codifica.»

Per essere sicuri di non rompere mai tale vincolo, la nostra implementazione abbassa `o_done` non appena viene abbassato `i_start` (l'effettivo abbassamento

di `o_done` avverrà in realtà al primo fronte di salita del clock successivo all'abbassamento di `i_start`).
In tutti gli altri testbench viene esplicitamente atteso il momento in cui `o_done` viene alzato dal componente per poi poter abbassare `i_start`, qui invece teniamo `i_start` alto anche dopo il termine dell'elaborazione e, come desiderato, `o_done` rimane alto fino a quando non viene abbassato `i_start`, anche se l'elaborazione è già terminata da tempo.

# 4 Conclusioni

## 4.1 Scelte di progetto

Di seguito riportiamo alcuni accorgimenti specifici adottati a livello di design e implementazione del datapath e della FSM:

- Nella macchina a stati è stato fatto volutamente *unwinding* dei cicli di scrittura dei 2 Byte di $Z$ per ogni lettura di un Byte di $W$: essendo il numero di iterazioni è fissato, si è preferito eliminare l'utilizzo di contatori, anche se a discapito di una FSM più ridondante. Non sono state riscontrate significative differenze nel report dei tempi.
- Il registro contenente l'indirizzo di scrittura è effettivamente inizializzato al valore $999_{(10)}$ anziché $1000_{(10)}$: questo rende la transizione $s_{24} \rightarrow s_4$, che riprende il ciclo di elaborazione di un nuovo Byte in ingresso, più semplice. Viene quindi eseguita una somma "a vuoto" durante la $1^a$ iterazione, che fa sì che il primo indirizzo nel quale si scrive sia effettivamente $1000_{(10)}$.

## 4.2 Valutazioni di carattere generale

A fronte di quanto già diffusamente commentato ed articolato nelle precedenti sezioni del report, sentiamo di poter affermare di aver compiuto tutti i passi necessari ad un buon svolgimento del progetto. In particolare riteniamo di aver avuto una concreta occasione per estendere quanto appreso a livello teorico durante il corso di Reti Logiche ad una visione più completa.
Tale approccio ci ha permesso di attraversare passo passo tutte le principali fasi di progetto: dalla progettazione e design del componente, della sua architettura e semantica, alla sua implementazione nel linguaggio VHDL, fino ad una analisi critica di quanto sintetizzato, sia a livello di comportamento atteso (si veda la Sezione 3.2), sia a livello di performance e metriche (si veda la Sezione 3.1).
Sulla base di queste considerazioni, possiamo ritenerci soddisfatti di ciò che abbiamo appreso all'interno di questo progetto e ci sentiamo di affermare una complessiva buona riuscita dello stesso.