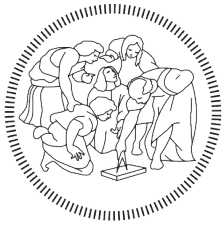

POLITECNICO
MILANO 1863

# Prova Finale - Progetto di Reti Logiche

Prof. Fabio Salice - Anno 2021/2022

Alessandro Folini (Codice Persona 10674753 - Matricola 933707)

# Indice

# 1 Introduzione

## 1.1 Descrizione della specifica

Sia dato un flusso continuo di parole di dimensione W, ognuna di 8 bit.
L'obiettivo del progetto è quello di implementare un componente hardware descritto in linguaggio VHDL che riceva in ingresso questo flusso continuo di parole e sia in grado, mediante un algoritmo di codifica convoluzionale con tasso di trasmissione 1/2, di restituire in uscita un altro flusso di parole di lunghezza Z, ognuna da 8 bit, tale per cui Z = 2W.

## 1.2 Codificatore convoluzionale

Il codificatore convoluzionale è una macchina sequenziale sincrona che, ricevuta in ingresso una sequenza U serializzata di bit di lunghezza 8*W, genera in uscita un flusso continuo Y ottenuto tramite concatenazione alternata dei due bit prodotti dalla codifica 1/2, secondo il seguente procedimento:

1. In ogni istante di tempo k viene serializzato il bit $U_k$ della parola U, a partire dal bit più significativo
2. Calcolo di p1k e p2k sulla base di U_k:
   `p1k` = `U_k` $XOR$ `U_k-2`
   `p2k` = `U_k` $XOR$ `U_k-1` $XOR$ `U_k-2`
3. Calcolo di $Y_k$, dove l'operatore && indica la concatenazione dei due bit:
   `Y_k` = `p1k && p2k`

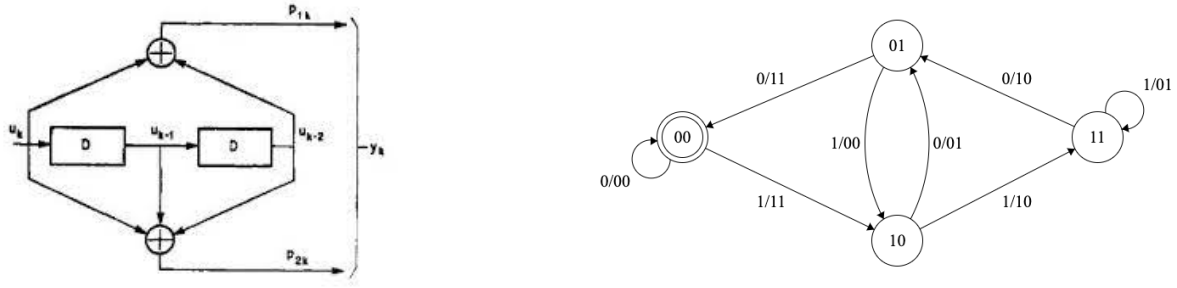


Figura 1: Schema del codificatore convoluzionale e FSM corrispondente

Ad esempio ipotizziamo di leggere da memoria la parola U = 10110110, al variare di k il convolutore produrrà i seguenti p1k e p2k, i quali andranno a formare la parola Y = 1101001010001010:

- p10 = 1, p20 = 1;
- p11 =0 , p21 = 1;
- p12 = 0, p22 = 0;
- p13 = 1, p23 = 0;
- p14 = 1, p24 = 0;
- p15 = 0, p25 = 0;
- p16 = 1, p26 = 0;
- p17 = 1, p27 = 0;

### 1.3 Interfaccia del componente

Il componente da implementare presenta la seguente interfaccia:

```
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 entity project_reti_logiche;
```

Nello specifico:

- `i_clk` : il segnale di `CLOCK` in ingresso;
- `i_rst` : il segnale di `RESET` che inizializza la macchina pronta per ricevere il segnale di `START`;
- `i_start` : il segnale di `START` generato dai TestBench;
- `i_data` : il segnale (vettore di 8 bit) che arriva dalla memoria in seguito ad una richiesta di lettura;
- `o_address` : il segnale (vettore di 16 bit) di uscita che manda l'indirizzo alla memoria;
- `o_done` : il segnale che comunica la fine dell'elaborazione e della scrittura in memoria del dato elaborato;
- `o_en` : il segnale di `ENABLE` da inviare alla memoria per poter comunicare sia in lettura che in scrittura;
- `o_we` : il segnale di `WRITE ENABLE` per abilitare la scrittura in memoria. In caso di lettura questo deve essere 0;
- `o_data` : il segnale (vettore di 8 bit) di uscita dal componente verso la memoria.

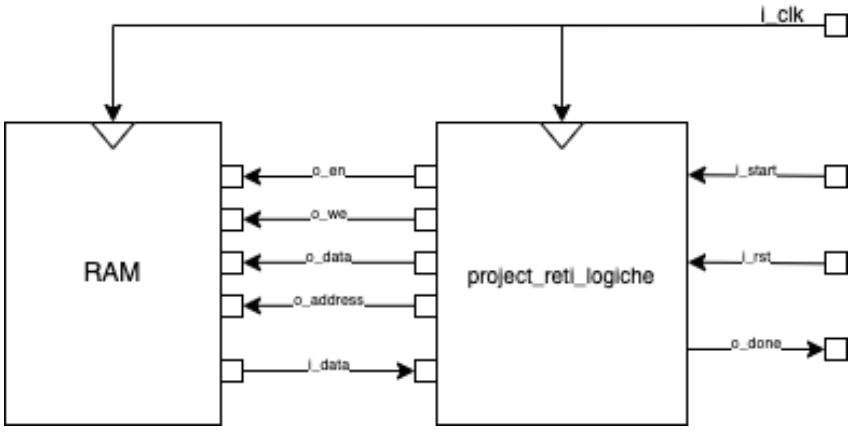


Figura 2: Schematizzazione dell'interfaccia tra il componente e la memoria

## 1.4 Descrizione della memoria e funzionamento del componente

| | |
|---|---|
| Lunghezza sequenza (W) | Indirizzo 0 |
| Byte 1 Ingresso | Indirizzo 1 |
| Byte 2 Ingresso | Indirizzo 2 |
| ... | |
| Byte W Ingresso | Indirizzo W |
| ... | |
| Byte 1 Uscita | Indirizzo 1000 |
| Byte 2 Uscita | Indirizzo 1001 |
| ... | |
| Byte 2W Uscita | Indirizzo 1000 + 2W - 1 |

Tabella 1: Memorizzazione dei dati nella memoria

La memoria, indirizzata al Byte, è organizzata come rappresentato nella Tabella 1:

1. Indirizzo 0: il numero W di parole che compongono la sequenza di ingresso (0-255);
2. Indirizzi 1 - W: i W Byte della sequenza in ingresso;
3. Indirizzi 1000 - (1000+2W-1): i 2W Byte della sequenza in uscita.

Quando il segnale di ingresso `i_start` viene portato a 1, il componente inizia la computazione spostandosi nello stato `READ_LEN` in cui inizia a richiedere dati alla memoria. Il segnale `i_start` rimmarrà alto fino a quando `o_done` non verrà portato alto.

Dopo aver codificato il Byte in ingresso e scritto i due Byte risultanti corrispondenti in memoria, il componente alza a 1 il segnale di `o_done`, per segnalare la fine dell'elaborazione. Il segnale `o_done` rimane alto fino a quando il segnale di `i_start` non viene riportato a 0; dopodichè anche `o_done` viene portato a 0. `i_start` non può essere portato a 1 fin tanto che `o_done` non è stato riportato a zero.

# 2 Architettura

Ho deciso di progettare il componente mediante una macchina a stati finiti, optando per una soluzione che presenta due process: uno che si occupa di resettare il componente ogni qualvolta `i_rst` viene portato ad 1 e di aggiornare i segnali (`segnale <= segnale_next`) su ogni fronte di clock, l'altro che è la vera e propria FSM che si occupa di modificare il valore dei segnali sulla base dello stato corrente.

## 2.1 Macchina a Stati Finiti

La macchina si compone di 12 stati, dei quali verrà data una breve descrizione nelle sezioni sottostanti. Segue inoltre una rappresentazione grafica della macchina a stati finiti realizzata (per questioni di leggibilità non sono segnati gli archi che da ogni stato portano in `START` quando `i_rst` = '1').

### 2.1.1 START

Lo stato "di riposo" della macchina, qui vengono inizializzati alcuni segnali il cui valore deve tornare ad essere quello di default ad ogni nuova sequenza anche in assenza di un reset esplicito. In questo stato si attende inoltre che `i_start` venga alzato a 1 per far partire la computazione.

### 2.1.2 READ_LEN

In questo stato si fa richiesta alla memoria di lettura del dato presente all'indirizzo 0, che corrisponde alla lunghezza della sequenza di parole da leggere.

### 2.1.3 WAIT_READ_LEN

In questo stato si attende la risposta della memoria a seguito della richiesta di lettura della cella 0.

### 2.1.4 SET_LEN

In questo stato si controlla innanzitutto che la lunghezza letta da memoria non sia 0: se effettivamente la lunghezza è diversa da 0 ci si salva l'indirizzo dell'ultima parola da leggere nel segnale `last_word_address_next`, altrimenti non è necessario procedere con la computazione in quanto non c'è nessuna sequenza di parole da leggere per cui il segnale `o_done_next` viene alzato a 1.

### 2.1.5 READ_WORD

In questo stato viene aggiornato l'indirizzo della nuova parola da leggere dalla memoria e qualora l'indirizzo dell'ultima parola letta corrisponda all'indirizzo salvato in `last_word_address_next` si alza `o_done_next` a 1 in quanto si è giunti alla fine della sequenza da leggere.
In caso contrario si fa una richiesta di lettura alla memoria, in modo da leggere la prossima parola da elaborare.

### 2.1.6 WAIT_READ_WORD

In questo stato si attende la risposta della memoria a seguito della richiesta di lettura di una parola.

### 2.1.7 CONV

In questo stato viene applicato l'algoritmo di convoluzione 1/2.
Ho deciso di non progettare il convolutore come una macchina a stati ma di utilizzare direttamente lo `XOR` tra i bit della parola in ingresso per comporre (non esplicitamente) i bit `p1k` e `p2k` che andranno a formare il vettore di 16 bit `y`, come spiegato nella Sezione 1.2. Ad ogni ciclo di clock si codifica un bit della parola in ingresso (selezionato mediante il segnale `pos`), partendo da quello più significativo, producendo due bit che vengono poi inseriti nel vettore `y` nelle rispettive posizioni.
Inizialmente, per la prima parola letta, il segnale `old_i_data` sarà composto di tutti 0 per cui lo `XOR` terrà conto solo dei bit della parola in ingresso.
In seguito questo cambierà in quanto il convolutore non si resetta tra una parola e l'altra, per cui quando si codificheranno i primi due bit di una nuova parola, bisognerà considerare nello `XOR` anche alcuni bit della parola precedente, in quanto per il convolutore tutte le parole in ingresso sono concatenate in un

flusso continuo di bit.
Fino a quando i bit da codificare non saranno finiti il segnale `curr_state` varrà sempre `CONV`.

### 2.1.8 WRITE_FIRST_WORD

In questo stato inizialmente viene salvata la parola in ingresso nel segnale `old_i_data_next` per il motivo spiegato nella Sezione 2.1.7.
Viene poi salvata nel segnale `o_data_next` la prima delle due parole risultanti dalla codifica della parola di ingresso, questa si trova negli 8 bit più significativi del segnale `y`. Si alzano inoltre i segnali `o_en_next` e `o_we_next` a 1 mandando una richiesta di scrittura alla memoria.
L'indirizzo nel quale scrivere il dato è inizialmente il 1000 e viene aggiornato ad ogni parola scritta.

### 2.1.9 WAIT_WRITE_FIRST_WORD

In questo stato si attende che la memoria abbia effettivamente scritto la prima parola in memoria a seguito della richiesta di scrittura.

### 2.1.10 WRITE_SECOND_WORD

In questo stato si manda un ulteriore richiesta di scrittura alla memoria alzando i segnali `o_en_next` e `o_we_next` a 1.
La parola che andrà scritta è la seconda delle due risultanti dalla codifica della parola di ingresso e si trova negli 8 bit meno significativi del segnale `y`.

### 2.1.11 WAIT_WRITE_SECOND_WORD

In questo stato si attende che la memoria abbia effettivamente scritto la seconda parola in memoria a seguito della richiesta di scrittura.

### 2.1.12 DONE

Lo stato finale nel quale si giunge dopo aver completato l'elaborazione dei dati in ingresso, qui si attende che il segnale `i_start` sia messo a 0 prima di ritornare nello stato iniziale `START`.

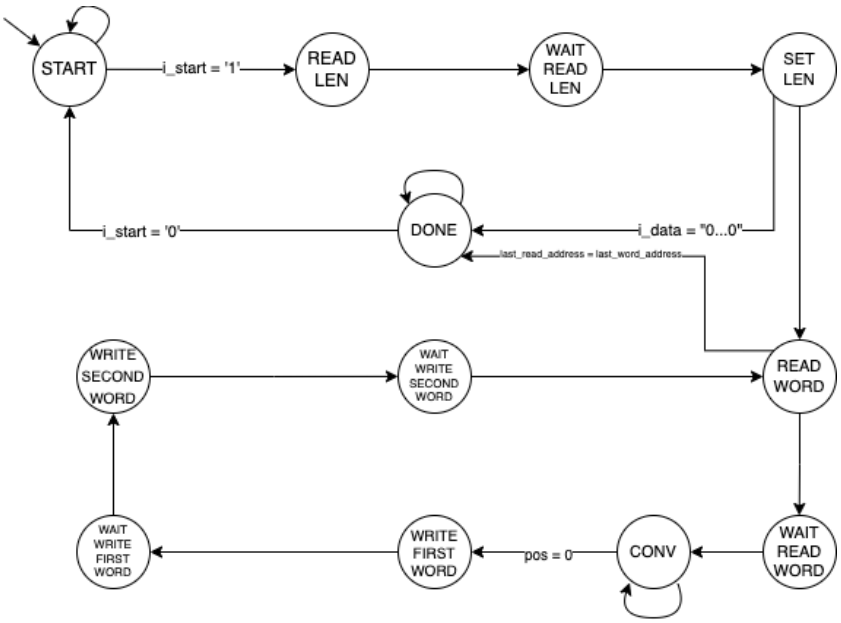


Figura 3: Schema grafico della FSM implementata

## 2.2 Signals utilizzati

Di seguito un elenco seguito da una breve spiegazione dei signals utilizzati nel progetto con i rispettivi tipi e il valore di default, non ho fatto uso di variabili all'interno dei process.
Ogni segnale ha anche un corrispondente segnale con lo stesso nome con l'aggiunta di "_next" alla fine, sono solo questi ultimi a venir direttamente modificati durante l'elaborazione e il loro valore viene salvato nel corrispondente segnale non "_next" ad ogni fronte di salita del clock. Per semplicità sono riportati solo i signal non "_next", ad eccezione di `next_state`.

```
type STATE is (START, READ_LEN, WAIT_READ_LEN, SET_LEN, READ_WORD, WAIT_READ_WORD,
               CONV, WRITE_FIRST_WORD, WAIT_WRITE_FIRST_WORD, WRITE_SECOND_WORD,
               WAIT_WRITE_SECOND_WORD, DONE);
signal curr_state, next_state : STATE := START;
signal last_word_address, last_read_address, y : std_logic_vector(15 downto 0) := (others => '0');
signal last_written_address : std_logic_vector(15 downto 0) := ("0000001111101000");
signal pos : integer range 0 to 7 := 7;
signal old_i_data : std_logic_vector(7 downto 0) := (others => '0');
```

- `curr_state`, `next_state`: contengono rispettivamente lo stato attuale e lo stato successivo della FSM;
- `last_word_address`: contiene l'indirizzo dell'ultimo Byte da leggere della sequenza in ingresso;
- `last_read_address`: contiene l'indirizzo dell'ultimo Byte letto dalla memoria;
- `y`: contiene le due parole risultanti dalla codifica di una parola della sequenza in ingresso;
- `last_written_address`: contiene l'indirizzo da utilizzare per scrivere un Byte in memoria, è inizializzato al valore 1000 come da specifica;
- `pos`: specifica la posizione del bit nel Byte corrente in ingresso che sta venendo codificato, è inizializzato a 7 in quanto si parte dal bit più significativo;
- `old_i_data`: contiene il Byte di ingresso immediatamente precedente a quello che sta venendo codificato, come meglio specificato nella Sezione 2.1.7.

# 3 Sintesi

## 3.1 Report Utilization

Una volta sintetizzato il componente tramite il tool Vivado di Xilinx, digitando nella Tcl Console il comando `report_utilization` è possibile visualizzare il seguente resoconto per quanto riguarda l'utilizzo di LUT e Flip-Flop:

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| Slice LUTs* | 115 | 0 | 134600 | 0.09 |
| LUT as Logic | 115 | 0 | 134600 | 0.09 |
| LUT as Memory | 0 | 0 | 46200 | 0.00 |
| Slice Registers | 93 | 0 | 269200 | 0.03 |
| Register as Flip Flop | 93 | 0 | 269200 | 0.03 |
| Register as Latch | 0 | 0 | 269200 | 0.00 |
| F7 Muxes | 1 | 0 | 67300 | <0.01 |
| F8 Muxes | 0 | 0 | 33650 | 0.00 |

Figura 4: Utilizzo di LUT e FF

Come si può vedere, le percentuali di utilizzo di LUT e FF sono inferiori all' 1%, per cui il componente occupa solamente una piccola parte della FPGA target, ovvero la `xc7a200tfbg484-1` della famiglia Artix-7.
I 93 FF inferiti sono tutti di tipo D, in particolare 84 di essi sono di tipo FDCE e i restanti 9 di tipo FDPE.
Come auspicato non è stato sintetizzato alcun Latch, in quanto la presenza di essi avrebbe potuto compromettere la sintesi del componente e di conseguenza il corretto funzionamento dello stesso nella fase di test post-sintesi.

## 3.2 Report Timing

Secondo la specifica il componente avrebbe dovuto funzionare con un periodo di clock inferiore ai 100ns, per cui nella fase iniziale di setup del progetto su Vivado ho inserito un file nei constraints nel quale ho specificato un vincolo temporale di 100ns al periodo di `i_clk`, e un Duty Cycle del 50%.
Una volta sintetizzato il componente, digitando nella Tcl Console il comando `report_timing` è stato possibile visualizzare che il Worst Negative Slack, ossia lo Slack nel caso pessimo, corrisponde a `95.075ns`.
Il vincolo temporale risulta quindi essere ampiamente rispettato.

# 4 Simulazioni

Per testare il corretto funzionamento del componente, questo è stato sottoposto ad una suite di test bench fornita dal docente insieme alle specifiche.
Oltre a questi 11 test, ho sottoposto il componente a più di 15000 altri test generati casualmente mediante un software scritto in Python in modo da aumentare la probabilità di scovare eventuali errori.
Il componente ha superato correttamente tutti i test ai quali è stato sottoposto, sia in `Behavioral Simulation` sia in `Post-Synthesis Functional Simulation`.
Vediamo ora una breve descrizione di ciascuno dei casi di test forniti dal docente.

## 4.1 tb_esempio / tb_example

Questa sottosezione racchiude 4 test bench il cui comportamento è pressoché identico.
Essi testano semplicemente il funzionamento del componente fornendo una sola sequenza di ingresso composta da un numero piccolo (2, 3, 6) di parole e verificando che gli indirizzi di memoria dal 1000 in poi contengano i corretti valori codificati.

## 4.2 tb_doppio_uguale

L'obiettivo di questo test è di verificare il corretto funzionamento del componente nella lettura di due sequenze in ingresso di fila dalla stessa RAM (in questo caso la stessa sequenza entrambe le volte).
Il componente quindi legge la prima sequenza di 3 parole dalla ram e scrive le 6 parole risultanti dalla codifica, il segnale `i_start` viene messo a 0 e poi nuovamente a 1 in modo che il componente parta con una nuova elaborazione della stessa sequenza elaborata precedentemente.

## 4.3 tb_re_encode / tb_tre_bis

Questi due test verificano che il componente funzioni correttamente nella lettura ed elaborazione di tre sequenze da 3 RAM diverse che vengono scambiate durante l'esecuzione.
I test, così come quello precedente, vanno a verificare come reagisce il componente a seguito di variazioni del segnale `i_start`.

## 4.4 tb_reset

Questo test sottopone il componente ad un reset asincrono durante l'elaborazione di una sequenza di 6 parole in ingresso, dopo un intervallo di tempo corrispondente a 15 volte il periodo di clock di 100ns. Il componente deve reagire correttamente al segnale di reset elaborando la sequenza senza errori.

## 4.5 tb_seq_max

Questo test verifica il funzionamento del componente nel caso in cui la sequenza di parole in ingresso abbia lunghezza massima.
Essendo la dimensione della sequenza di ingresso salvata nel primo Byte della memoria, la lunghezza massima si avrà quando `RAM[0] = "11111111"`, che corrisponde ad una sequenza di 255 parole.
Gli indirizzi di memoria dal 1000 al 1509 (compresi) dovranno quindi contenere i corretti valori della codifica delle 255 parole in ingresso.

## 4.6 tb_seq_min

Questo test è simile al test precedente, nel senso che testa una condizione limite riguardante la lunghezza della sequenza in ingresso, ovvero la lunghezza minima 0 che si ha quando `RAM[0] = "00000000"`. Il componente una volta letta la lunghezza della sequenza dalla memoria, si accorge che è 0 e quindi si porta subito nello stato `DONE` alzando `o_done` a 1 e attendendo che `i_start` venga riportato a 0 prima di tornare nello stato `START`.

## 4.7 tb_tre_reset

Questo test va a verificare che il componente funzioni correttamente quando sottoposto a 3 cambi di RAM con un reset durante l'elaborazione della sequenza di ingresso letta dalla prima RAM.

# 5 Conclusioni

Come già anticipato sopra il componente progettato è in grado di superare tutti i casi di test forniti dal docente oltre a quelli generati casualmente, sia in `Behavioral Simulation` sia in `Post-Synthesis Functional Simulation`, occupando solo una piccola parte della FPGA target e rientrando ampiamente nel vincolo temporale di 100ns sul periodo di clock.
Per cui posso dire che nonostante il percorso non sia stato privo di intoppi (quali l'iniziale inferenza di latch in seguito risolta e la poca facilità d'uso del tool Vivado) specialmente durante la fase di scrittura in VHDL, il componente progettato soddisfa tutti i requisiti richiesti dalla specifica.