POLITECNICO
MILANO 1863

PROGETTO RETI LOGICHE 2021/2022

IGNAZIO NETO DELL'ACQUA

Matricola: 937605

Codice Persona: 10704869

Prof. Fabio Salice

1. INTRODUZIONE

Lo scopo di questo progetto è la realizzazione di un modulo Convoluzionale attraverso l'utilizzo di una tecnologia FPGA che sia in grado di interfacciarsi a una memoria RAM.

1.1. Cosa è un modulo Convoluzionale

Un modulo Convoluzionale è un componente hardware utilizzato nell'ambito delle telecomunicazioni per generare a monte di una trasmissione digitale un codice convoluzionale, cioè un tipo di codice per la correzione d'errore.

1.2. Codici per la Correzione d'errore

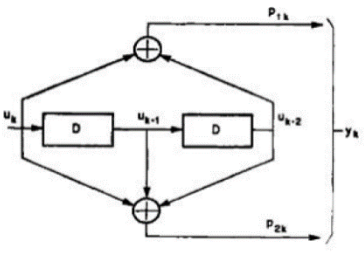
I codici per la correzione d'errore:

- Sono impiegati a valle di una comunicazione nella fase di **Forward Error Correction** (**FEC**), un meccanismo di rilevazione e successiva correzione degli errori durante una trasmissione dell'informazione.
- Sono basati su un'opportuna codifica di canale e sull'aggiunta di bit di ridondanza al flusso informativo.

In particolare, nel caso di codifica convoluzionale, ogni simbolo d'informazione a *m* bit (ogni parola da *m* bit) da codificare è "trasformato" attraverso appositi algoritmi in un simbolo a *n* bit, dove *m*/*n* è il rapporto (*rate*) del codice ($n \geq m$).

1.3. Caso di Interesse: Codice Convoluzionale ½

Il modulo in questione processa con un rate *W/Z = ½* : per ogni parola in ingresso *W* (*W* = parola da 8 bit) ne vengono generate due in uscita (*Z*= due parole da 8 bit ciascuna). L'algoritmo adottato è il seguente:
Il componente, data in ingresso una sequenza continua di W parole, le serializza generando un flusso continuo *Uk* da 1 bit che viene posto in ingresso al Codificatore Convoluzionale con tasso di trasmissione *½*. Quest'ultimo è una macchina sequenziale sincrona con un clock globale e segnale di reset. Esso è schematizzato in figura utilizzando due Flip-Flop Tipo D in cui inizialmente gli stati Uk+1 e Uk+2 sono posti a 0 tramite reset.



Il suddetto Codificatore Convoluzionale ha come funzione quella di generare un codice convoluzionale *½* : dato in input un flusso Uk, esso ne produce uno continuo Yk in output, il quale è il prodotto del concatenamento alternato dei due bit *P1k* e *P2k*. In formule:

P1k= Uk XOR Uk+2

P2k= Uk XOR Uk+1 XOR Uk+2

Infine, la sequenza d'uscita Z è ottenuta dalla parallelizzazione, su 8 bit, del flusso continuo *Yk*.

1.4. Esempio

Per comprendere meglio il processo di codifica del modulo viene proposto il seguente esempio:
Data W=01011100 la parola in ingresso al modulo. Da essa viene generato il flusso continuo serializzato Uk=[0,1,0,1,1,1,0,0] posto in ingresso al Codificatore Convoluzionale precedentemente resettato (Uk+1=Uk+2=0) che genera in output i flussi:

P1k= [0,1,0,0,1,0,1,1]
P2k= [0,1,1,0,0,1,0,1]

[NOTA: ad ogni ciclo di clock viene posto in ingresso un Uk progressivo a partire dal bit piu' significativo.]
[*NOTA: il flusso viene portato in avanti grazie all'impiego dei Flip-Flop tipo D:

D0= Uk
D1= Uk+1 .]

Questi flussi sono ottenuti nel seguente modo. Parto da t=0:

- t=0) Uk=0 Uk+1=0 Uk+2=0
  P1k=0 XOR 0= 0
  P2k=0 XOR 0 XOR 0= 0
  ➔ Flusso viene posto in avanti*: Uk+1=Uk=0, Uk+2= Uk+1=0
- t=1) Uk=1 Uk+1=0 Uk+2=0
  P1k=1 XOR 0= 1
  P2k=1 XOR 0 XOR 0= 1
  ➔ Flusso viene posto in avanti*: Uk+1= 1, Uk+2= 0
- t=2) Uk=0 Uk+1=1 Uk+2=0
  P1k=0 XOR 0= 0
  P2k=0 XOR 1 XOR 0= 1
  ➔ Flusso viene posto in avanti*: Uk+1= 0, Uk+2= 1

- t=3) Uk=1 Uk+1=0 Uk+2=1

  P1k=1 XOR 1= 0

  P2k=1 XOR 0 XOR 1= 0

  ➔ Flusso viene portato in avanti*: Uk+1= 1, Uk+2= 0

…E cosi via, fino ad ottenere i flussi riportati in precedenza:

P1k= [0,1,0,0,1,0,1,1]

P2k= [0,1,1,0,0,1,0,1]

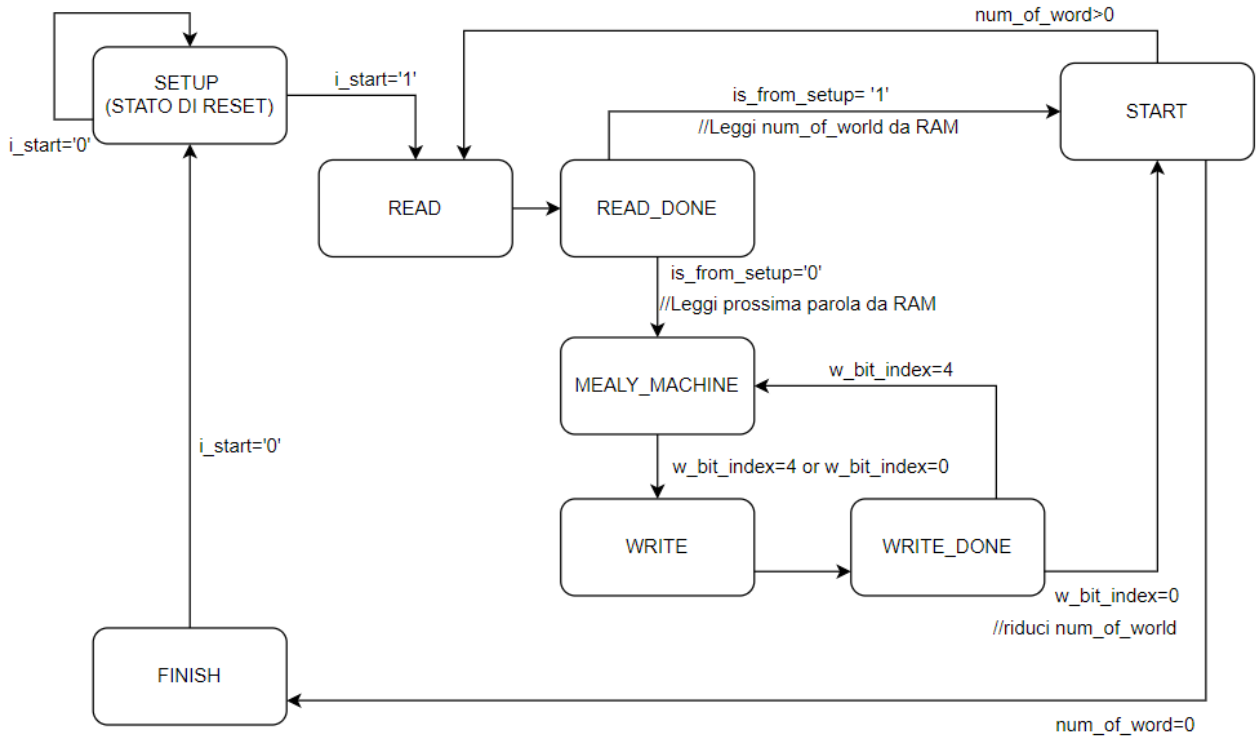
Il concatenamento dei valori Pk1 e Pk2 per produrre Yk avviene in maniera alternata (Pk1 al tempo t, Pk2 al tempo t, Pk1 al tempo t+1 Pk2 al tempo t+1 e cosi' via). Si ottiene infine Z parallelizzando a 8 bit:

Yk= [0,0,1,1,0,1,0,0,1,0,0,1,1,0,1,1] -> Z= 00110100 e 10011011

2. ARCHITETTURA

2.1. Schema funzionale

Il modulo progettato elabora seguendo il seguente schema funzionale:



Partendo da uno stato di SETUP (stato di RESET), l'idea alla base dello schema è che il modulo:

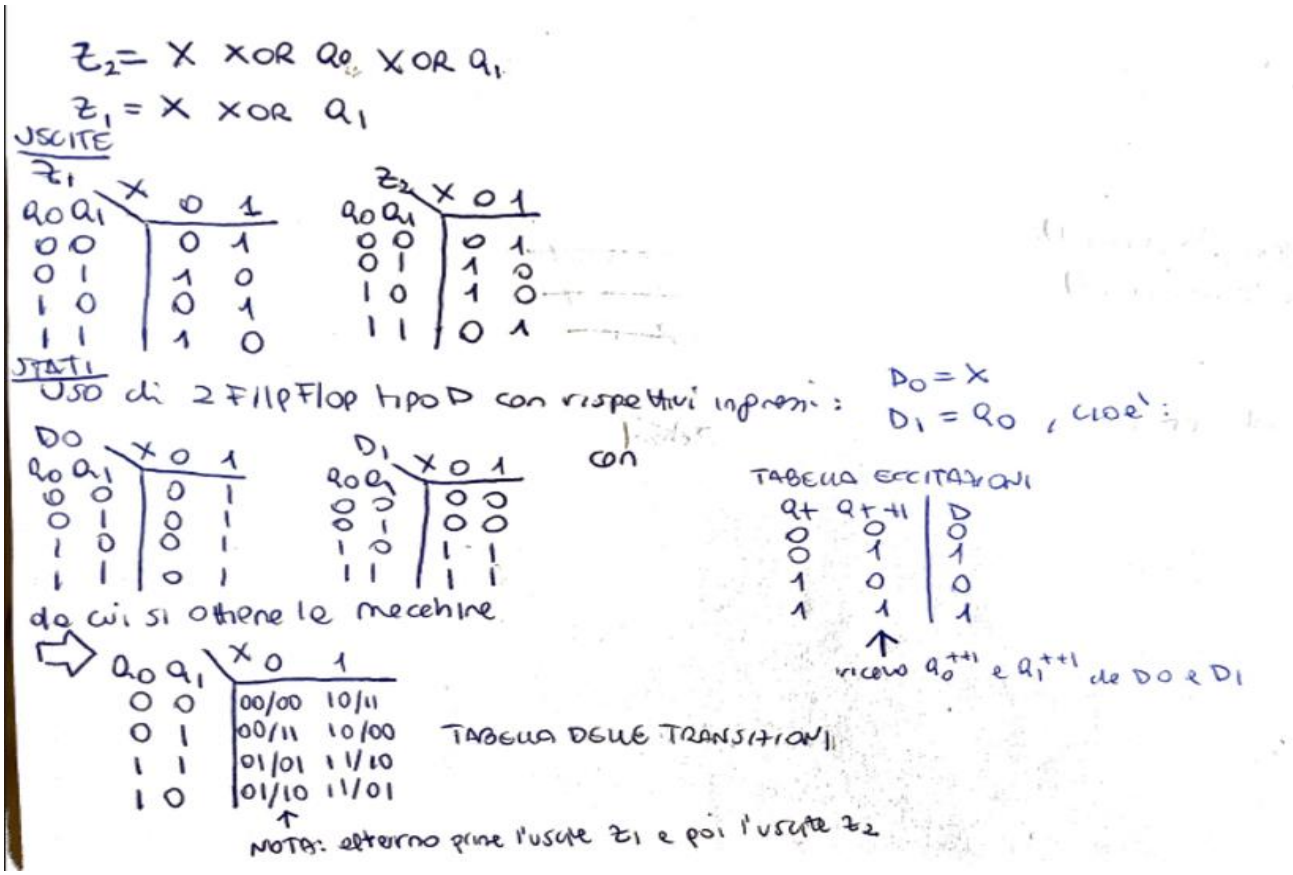
0. Attende che venga avviato ("i_start='1'");
1. Legge da memoria RAM il numero di parole "num_of_world" che esso deve processare (così che capisca quando deve porsi in stato di FINISH) e successivamente si pone in START;
2. Quando è in START, fino a che ci sono parole da leggere:
    a. Inizializza i valori per la computazione con il Codificatore Convoluzionale;
    b. Legge parola da memoria;
    c. Pone la parola letta serializzata in ingresso al Codificatore Convoluzionale, il quale è sintetizzato come una macchina di Mealy, al fine di processare la prima metà di essa (i primi 4 bit generano la prima parola Z da produrre in output);
    d. Quando si è processata la prima metà della parola scrive in memoria la codifica risultante;
    e. Processa la seconda metà della parola serializzata (i secondi 4 bit generano la seconda parola di Z da produrre in output);
    f. Riporta lo stato in ingresso al Codificatore Convoluzionale;
    g. Quando si è processata anche la seconda metà della parola scrive in memoria la codifica risultante;
    h. Riduce numero di parole che esso deve processare ("num_of_world—");
    i. Riporta il modulo in START per la lettura della prossima parola.
    [NOTA: lo stato della macchina di Mealy non viene resettato]
3. Quando è in START e non ci sono parole da leggere("num_of_world=0"), pone il suo stato in FINISH in attesa che il segnale di "i_start" venga portato a 0. Fatto ciò, il modulo si resetta (riportando anche lo stato della macchina in quello di reset) e si riporta lo stato in SETUP.

E' inoltre da aggiungere che il modulo progettato è dotato di un RESET asincrono che a partire da qualsiasi suo stato resetta i propri valori e lo riporta lo stato in SETUP.

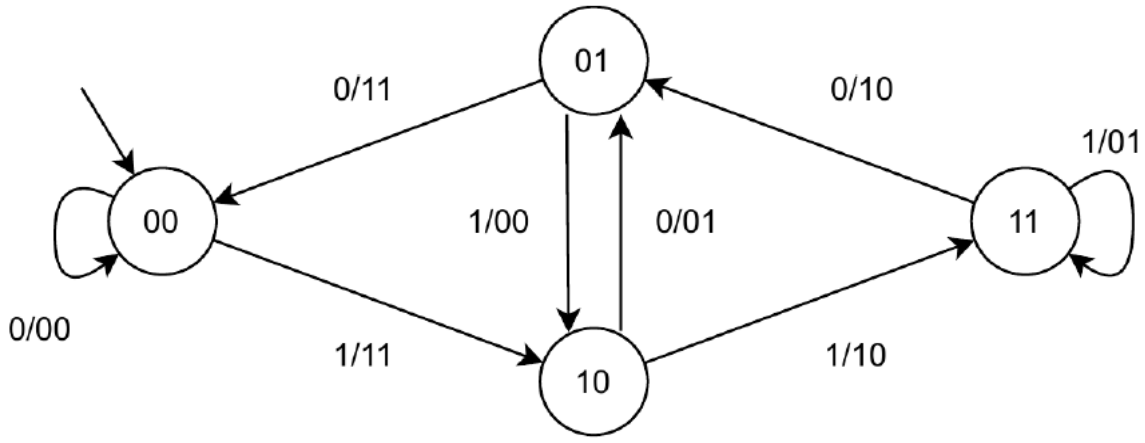
2.2. Codificatore Convoluzionale: Macchina di Mealy

La chiave dello schema funzionale sta nella comprensione di ciò che effettua il Codificatore Convoluzionale, nel quale lo stato di reset è 00 e dove, per ogni bit in ingresso, se ne generano due in uscita. In particolare, questa macchina è stata sintetizzata a partire dall'analisi del Codificatore Convoluzionale descritto nella sezione 1.3 e 1.4.

Siano, per semplicità, Uk=X, Uk+1=Q0, Uk+2=Q1, Z1=P1k, Z2=P2k:



La macchina che ne risulta, e che dunque è stata impiegata, è quindi la seguente:



2.2 Descrizione Codice VIVADO VHDL

2.1.1. Interfaccia del componente, segnali e stati utilizzati

1. Segnali di periferica:

```
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;
```

- "i_clk" è il segnale di clock di sistema;

- "i_rst" è l'eventuale segnale di reset;

- "i_start" è il segnale che fa partire il modulo;

- "i_data" è un vettore binario a 8 bit contenente il dato letto dalla memoria;

- "o_address" è un vettore binario a 8 bit che serve ad indicare l'indirizzo in memoria desiderato per la fase di lettura e di scrittura;

- "o_done" è il segnale che notifica in uscita la fine dell'elaborazione di tutte le parole;

- "o_en" è il segnale che serve ad abilitare le interazioni con la memoria;

- "o_we" è il segnale che serve ad abilitare la scrittura in memoria;

- "o_data" è il vettore contenente la stringa da salvare in memoria.

2. Stati del modulo:

```
type state_process_type is (SETUP, START ,READ, READ_DONE, MEALY_MACHINE, WRITE, WRITE_DONE, FINISH);
type state_mealy_type is (s0, s1, s2, s3);

signal mealy_state : state_mealy_type;
signal module_state : state_process_type;
```

- "module_state" sono gli stati dello schema funzionale;

- "mealy_state" sono gli stati del Codificatore Convoluzionale (s0=00, s1=10, s2=01, s3=11).

3. Segnali Ausiliari:

```
signal Z: std_logic_vector(7 downto 0);
signal W : std_logic_vector(7 downto 0);

signal is_from_setup: std_logic ;      -- flag booleano che indica come interpretare i valori letti in input

signal z_bit_index: integer;           -- indice del i-esimo bit di Z
signal w_bit_index: integer;           -- indice del i-esimo bit di W

signal num_of_read : integer;          -- numero di parole lette
signal o_address_offset : integer;     -- offset per l'indirizzo di scrittura
signal num_of_word: integer;           -- numero di parole da computare
```

- "Z" è un vettore binario a 8 bit utilizzato per contenere la parola che il modulo;

- "W" è un vettore binario a 8 bit impiegato per contenere la parola in ingresso da dover processare.

2.1.3. Spiegazione dettagliata dello schema funzionale

Il modulo è progettato ipotizzando che il segnale di reset sia invocato sempre all'inizio della computazione:

- RESET= reset asincrono che:

  - Porta il modulo in stato di SETUP;

  -Porta il Codificatore Convoluzionale in stato s0;

  - Inizializza i valori utili nell'esecuzione del modulo (num_of_read=0 e o_address_offset=1000).

[NOTA: è richiesto che le parole codificate siano scritte in memoria a partire dall'indirizzo RAM (1000).]

  - Pone is_from_setup alto.

[NOTA: All'interno del processo il codice si comporta diversamente nel caso in cui a variare sia i_rst oppure i_clk, cosi' da poter avere un reset che funzioni anche asincronamente al clock. La funzione "if rising_edge(i_clk)" viene usata per fare in modo che l'esecuzione del codice al suo interno sia sincrona sul fronte di clock.]

STATI:

- SETUP: stato che attende che i_start venga posto alto (segnale di avvio). Quando ciò avviene vengono impostati i segnali per la lettura del numero di parole del flusso (in particolare "o_address<=00000000"). Infine, il modulo viene portato in stato di READ.
  [NOTA: In RAM (0) è presente il numero di parole del flusso.]
- READ: stato durante il quale si attende che venga effettuata la lettura. Nel frattempo sono impostati i segnali per la chiusura della procedura. Infine il modulo viene portato in stato di READ_DONE.

- READ_DONE: stato di terminazione della lettura. Se is_from_setup è alto:
    - Viene memorizzata la parola letta in num_of_read (trasformazione da binario a numero intero);
    - Viene portato is_from_setup basso;
    - Modulo va in stato di START.

  Se invece il flag risulta basso:

    - Viene memorizzata la parola letta all'interno di W;
    - Viene aumentato il numero di parole lette (num_of_read);
    - porta modulo in stato di MEALY_MACHINE.

- START: è lo stato in cui inizia la computazione delle parole. Se num_of_read è nullo, il modulo viene portato in stato di FINISH. Se ciò non è vero vengono inizializzati i valori per la computazione del Codificatore Convoluzionale (in particolare: "w_bit_index<=7" e "z_bit_index<=7").
  [NOTA: W è scandito a partire dal bit più significativo; Z è computato a partire dal bit più significativo.]
  Successivamente vengono impostati i segnali per la lettura della prossima parola (in particolare: "o_address <= std_logic_vector(to_unsigned (num_of_read+1,16)").
  [NOTA: parole da leggere memorizzate da RAM (1) in poi.]

- MEALY_MACHINE: fase di computazione del Codificatore Convoluzionale, la macchina a stati descritta nella sezione 2.2. All'interno di questo stato sono dunque presenti i 4 sottostati s0, s1, s2, s3 .
  [NOTA: la serializzazione è stata effettuata sfruttando la gestione di VHDL delle parole come normali vettori di interi.]
  La computazione viene effettuata scandendo a retroso (diminuendo w_bit_index di uno per ciclo) W e compilando a 2 bit alla volta Z (diminuendo z_bit_index di due per ciclo).
  [NOTA: in questa fase vengono elaborati 4 bit alla volta di W generando una parola Z in uscita da 8 bit.]
  Se è stata scandita la prima metà della parola W (w_bit_index=4) vengono impostati i segnali per la scrittura della parola Z (in particolare:
  "o_address<=std_logic_vector(to_unsigned(o_address_offset,16)") e viene portato il modulo in WRITE. Stessa cosa se ad essere stata scandita è la seconda metà della parola W (w_bit_index=0).
  Se nessuna di queste due condizioni è verificata, allora vengono ridotti w_bit_index e z_bit_index mantenendo lo stato del modulo in MEALY_MACHINE.

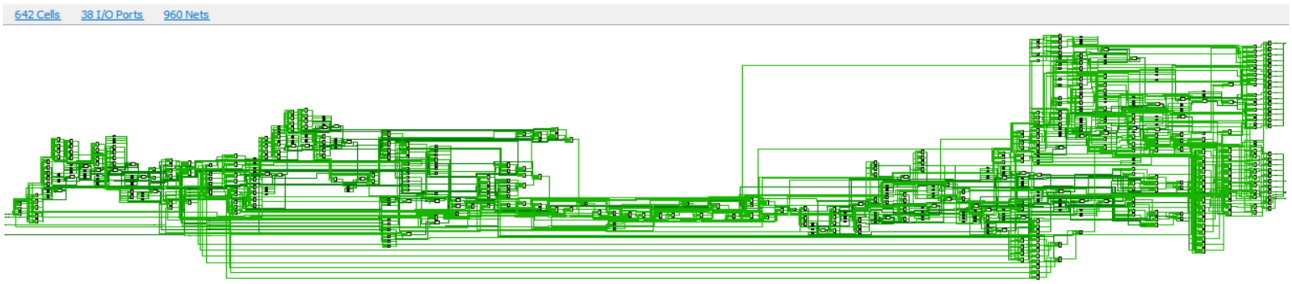
- WRITE: è lo stato di scrittura. Si pone Z in output per la scrittura in memoria e successivamente viene aumentato o_address_offset. Infine, il modulo viene portato in stato di WRITE_DONE.
- WRITE_DONE: è lo stato di scrittura completata. Si impostano i segnali per la chiusura della scrittura. Successivamente se w_bit_index=4 (modulo ha computato solo la prima metà di W):
    - Z viene reinizializzato;
    - Il modulo viene riportato in stato di MEALY_MACHINE per la computazione della seconda metà di W.

  Se invece w_bit_index=0 (modulo ha computato anche la seconda metà di W):

    - Vengono diminuite le num_of_world;
    - Il modulo viene riportato in stato di START.
- FINISH: stato di fine esecuzione del modulo. Si pone il segnale di "i_done" alto per segnalare la terminata esecuzione e si rimane in attesa di un segnale "i_start" basso. Arrivato quest'ultimo viene riposto "i_done" basso e successivamente resettato il modulo (vengono effettuate stesse operazioni del RESET).

3. RISULTATI SPERIMENTALI

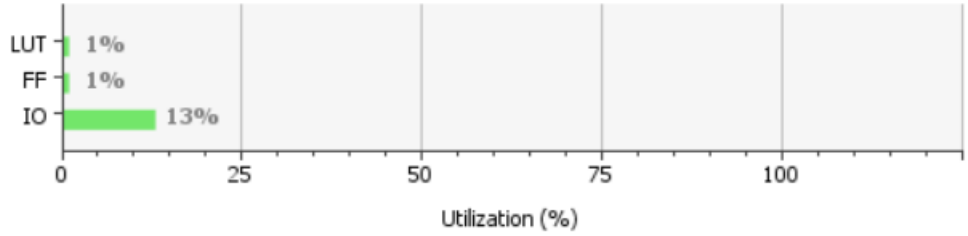
Il componente è risultato correttamente sintetizzabile e implementabile. Esso supera inoltre tutti i test branch forniti sia in pre-sintesi che in post sintesi. Di seguito un'immagine del modulo sintetizzato:



3.1 Report di Sintesi

Dall'analisi del componente si estrapolano le seguenti caratteristiche:

Resource	Utilization	Available	Utilization %
LUT	365	134600	0.27
FF	182	269200	0.07
IO	38	285	13.33



-Si può osservare un utilizzo piuttosto esiguo di Lookup Table (365) e di Flip Flop (182), meno del 0.3% di quelli disponibili dalla FPGA.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	93,278 ns	Worst Hold Slack (WHS):	0,080 ns	Worst Pulse Width Slack (WPWS):	49,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	434	Total Number of Endpoints:	434	Total Number of Endpoints:	183

All user specified timing constraints are met.

-Si può notare che il componente rispetta i parametri sul timing di 100 ns, nonché che esso presenta un valore elevato di Worst Negative Slack (93.278ns su 100ns).

3.1 Simulazioni

Come detto in precedenza il componente supera tutti i test forniti dal testbranch, volti ognuno ad esaminare uno o piu' casi limite:

- Test Base (tb_esempio_1, tb_esempio_2, tb_esempio_3, tb_example) = test volti a verificare che il componente esegua correttamente in condizioni standard (1 singolo flusso).

  tb_esempio_1:

```
Failure: Simulation Ended! TEST PASSATO
Time: 1982600 ps  Iteration: 0  Process: /project_tb/test  File: C:/Users/ignaI/Downloads/test/tb_esempio_1.vhd
$finish called at time : 1982600 ps : File "C:/Users/ignaI/Downloads/test/tb esempio 1.vhd" Line 124
```

  tb_esempio_2:

```
Failure: Simulation Ended! TEST PASSATO
Time: 1892600 ps  Iteration: 0  Process: /project_tb/test  File: C:/Users/ignaI/Downloads/test/tb_esempio_2.vhd
$finish called at time : 1892600 ps : File "C:/Users/ignaI/Downloads/test/tb_esempio_2.vhd" Line 145
```

  tb_esempio_3:

```
Failure: Simulation Ended! TEST PASSATO
Time: 1217600 ps  Iteration: 0  Process: /project_tb/test  File: C:/Users/ignaI/Downloads/test/tb_esempio_3.vhd
$finish called at time : 1217600 ps : File "C:/Users/ignaI/Downloads/test/tb_esempio_3.vhd" Line 129
```

  tb_example:

```
Failure: Simulation Ended! TEST PASSATO (ENCODE_EXAMPLE)
Time: 11950100 ps  Iteration: 0  Process: /project_tb/test  File: C:/Users/ignaI/Downloads/test/tb_example.vhd
$finish called at time : 11950100 ps : File "C:/Users/ignaI/Downloads/test/tb_example.vhd" Line 125
```

- tb_re_encode = test volto a verificare che il componente esegua correttamente quando in ingresso al modulo ci sono piu' flussi uno dietro l'altro (in questo caso 3):

```
Failure: Simulation Ended! TEST PASSATO
Time: 25650100 ps  Iteration: 0  Process: /project_tb/test  File: C:/Users/ignaI/Downloads/test/tb_re_encode.vhd
$finish called at time : 25650100 ps : File "C:/Users/ignaI/Downloads/test/tb_re_encode.vhd" Line 185
```

- tb_reset = test volto a verificare che il componente esegua correttamente quando durante l'esecuzione viene inviato un segnale di reset asincrono:

```
Failure: Simulation Ended! TEST PASSATO
Time: 12050100 ps  Iteration: 0  Process: /project_tb/test  File: C:/Users/ignaI/Downloads/test/tb_reset.vhd
$finish called at time : 12050100 ps : File "C:/Users/ignaI/Downloads/test/tb_reset.vhd" Line 122
```

- tb_seq_max = test volto a verificare che il componente esegua correttamente quando in ingresso al modulo è presente un flusso di lunghezza massima (RAM(0)=11111111 -> 255 parole):

```
Failure: Simulation Ended! TEST PASSATO
Time: 383650100 ps  Iteration: 0  Process: /project_tb/test  File: C:/Users/ignaI/Downloads/test/tb_seq_max.vhd
$finish called at time : 383650100 ps : File "C:/Users/ignaI/Downloads/test/tb_seq_max.vhd" Line 863
```

- tb_seq_min = test volto a verificare che il componente esegua correttamente quando in ingresso al modulo è presente un flusso di lunghezza nulla (RAM(0)=00000000 -> 0 parole):

```
Failure: Simulation Ended! TEST PASSATO
Time: 1150100 ps  Iteration: 0  Process: /project_tb/test  File: C:/Users/ignaI/Downloads/test/tb_seq_min.vhd
$finish called at time : 1150100 ps : File "C:/Users/ignaI/Downloads/test/tb_seq_min.vhd" Line 107
```

- tb_tre_bis = test volto a verificare che il componente esegua correttamente quando i flussi in ingresso sono associati a diverse RAM (1° flusso letto da RAM1 , codificato, poi memorizzato in RAM1, 2° flusso letto da RAM2 , codificato, poi memorizzato in RAM2 e cosi via…):

```
Failure: Simulation Ended! TEST PASSATO
Time: 3617600 ps  Iteration: 0  Process: /project_tb/test  File: C:/Users/ignaI/Downloads/test/tb_tre_bis.vhd
$finish called at time : 3617600 ps : File "C:/Users/ignaI/Downloads/test/tb_tre_bis.vhd" Line 201
```

- tb_tre_reset = test volto a verificare che il componente esegua correttamente quando durante l'esecuzione vengono inviati piu' segnali di reset (in questo caso 3):

```
Failure: Simulation Ended! TEST PASSATO
Time: 22350100 ps  Iteration: 0  Process: /project_tb/test  File: C:/Users/ignaI/Downloads/test/tb_tre_reset.vhd
$finish called at time : 22350100 ps : File "C:/Users/ignaI/Downloads/test/tb_tre_reset.vhd" Line 204
```

4. CONCLUSIONI

Dopo un'attenta valutazione e progettazione, è possibile notare che il modulo proposto presenta due principali vantaggi, i quali implicano un coretto stile di programmazione:

- Semplicità di realizzazione;

- Report di sintesi soddisfacenti;

Inoltre, i test confermano che il componente funziona correttamente anche in casi eccezionali, ulteriore elemento di conferma riguardo al corretto impiego del medesimo.