POLITECNICO
MILANO 1863

Implementazione di un codificatore convoluzionale in VHDL

Prova Finale di reti logiche 2021 - 2022

Autore: Fernando Morea

Professore: Fabio Salice

Codice Persona: 10707314

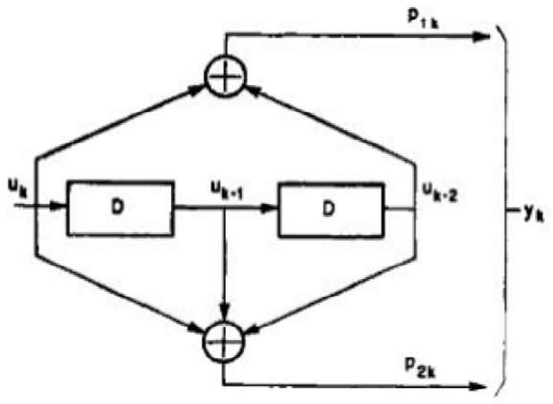
Codice Matricola: 932504

1. Introduzione e modello matematico del codificatore

Questo progetto si concentra sulla realizzazione di un encoder convoluzionale con velocità *rate* = $\frac{k}{n} = \frac{1}{2}\left(\frac{\text{bit in input}}{\text{bit in output}}\right)$ utilizzando la tecnologia FPGA (field-programmable gate array logic).

È attorno al 1950 che per la prima volta viene introdotto un approccio moderno alla teoria della correzione degli errori nei sistemi di comunicazione digitali, con i lavori pionieristici di Shannon, Hamming e Golay.

La teoria alla base è piuttosto semplice, l'idea è quella di aggiungere dell'informazione ridondante ai dati trasmessi, cosicché anche se il ricevente dovesse perdere qualche bit di informazione a causa di rumore sul canale, sarebbe comunque in grado di ricostruire il messaggio inviato nella sua interezza.



Nel nostro modello matematico di encoder chiameremo U_k la successione del bit di input all'encoder, $Y_k = \begin{pmatrix} P1_k \\ P2_k \end{pmatrix}$ la successione del vettore di bit in uscita dal nostro encoder.

Si noti che affinché la macchina produca degli output validi occorre che essa sia accesa per almeno due cicli di clock, il tempo necessario affinché il nostro bit in input si propaghi fino al flip flop D più a destra (tempo di setup della macchina).

La relazione che lega gli ingressi e le uscite diventa quindi:

$$Y_k = \begin{pmatrix} U_k \oplus U_{k+2} \\ U_k \oplus U_{k+1} \oplus U_{k+2} \end{pmatrix}$$

Sia ad esempio la successione $U_k = \{b_0, b_1, \ldots, b_7\}$ la successione del bit in input a partire dal tempo t_0.

$U_k = \{b_0,\ b_1,\ b_2,\ b_3,\ b_4,\ b_5,\ b_6,\ b_7\}$

$U_{k+1} = \{\boldsymbol{b_{-1}}, b_0,\ b_1,\ b_2,\ b_3,\ b_4,\ b_5,\ b_6\}$

$U_{k+2} = \{\boldsymbol{b_{-2}}, \boldsymbol{b_{-1}}, b_0,\ b_1,\ b_2,\ b_3,\ b_4,\ b_5\}$

Si pone da subito il problema di capire cosa assegnare ai bit b_{-1} e b_{-2} quando la macchina non è stata accesa per il tempo sufficente a popolare l'uscita dei due flip flop. La supposizione fatta è quella che la macchina sia inizialmente spenta con input in ingresso identicamente nullo:

$$\forall\ b_k \in [b_{-2}, b_{-1}] \ \Rightarrow\ b_k = 0$$

Questo piccolo dettaglio permette di passare ad un esempio pratico.

$$U_k = \{1,\ 0,\ 1,\ 0,\ 0,\ 0,\ 1,\ 0\}$$

$$U_{k+1} = \{\mathbf{0},\ 1,\ 0,\ 1,\ 0,\ 0,\ 0,\ 1\}$$

$$U_{k+2} = \{\mathbf{0},\ \mathbf{0},\ 1,\ 0,\ 1,\ 0,\ 0,\ 0\}$$

$$P_{1_k} = U_k \oplus U_{k+2} = \{1,\ 0,\ 0,\ 0,\ 1,\ 0\ ,\ 1,\ 0\ \}$$

$$P_{2_k} = U_k \oplus U_{k+1} \oplus U_{k+2} = \{1,\ 1,\ 0,\ 1,\ 1,\ 0,\ 1,\ 1\}$$

Siano infine i vettori di bit:

$Output_1 = [\ P_{1_0},\ P_{2_0},\ P_{1_1},\ P_{2_1},\ \ldots,\ P_{1_3},\ P_{2_3}]$ e

$Output_2 = [\ P_{1_4},\ P_{2_4},\ P_{1_5},\ P_{2_5},\ \ldots,\ P_{1_7},\ P_{2_7}]$

Il problema iniziale è dunque molto semplificato: Ogni flusso seriale di input da 8 bit produce, dopo un certo tempo di esecuzione dell'algoritmo, due Byte in uscita, $Output_1$ e $Output_2$.

Nel nostro esempio:

Input = U_k = [1, 0, 1, 0, 0, 0, 1, 0] (memorizzato in RAM) produce

$Output_1 = [\ 1, 1, 0, 1, 0, 0, 0, 1]$ e

$Output_2 = [\ 1, 1, 0, 0, 1, 1, 0, 1]$

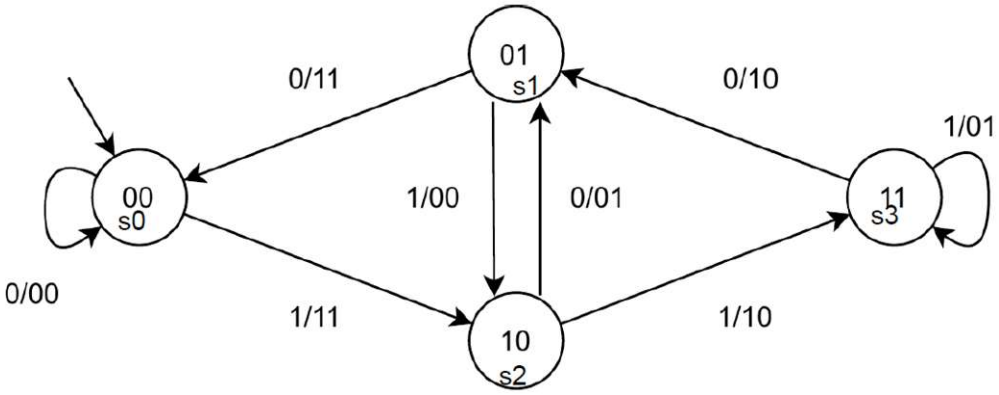
Questo flusso seriale da 8 bit in input in realtà è memorizzato in un vettore da 1 Byte in ram, dunque possiamo dire senza timore di smentite che: 1 Byte in input produce, dopo un certo tempo di esecuzione dell'algoritmo di enconding, 2 Byte in output.

Si osservi che molti bit di $Output_1$ vengono influenzati dal precedente Byte letto dalla ram, mentre i bit di $Output_2$ dipendono solamente dal Byte correntemente letto.

2. Architettura

2.a Macchina a stati: serial input, vettore di due bit in output

Il modulo essenziale per il funzionamento dell'encoder di Viterbi è quello descritto dalla macchina a stati mostrata nelle specifiche, che viene riportato qui di seguito.



L'automa si presta ad essere modellato con una macchina di Mealy a 4 stati. La scelta che è stata effettuata è invece quella di convertire questa macchina di Mealy nella sua macchina di Moore equivalente.

Le motivazioni legate a questa scelta sono ben note in letteratura:

La macchina Mealy ha uscite asincrone
– se l'input presenta problemi, anche l'output presenterà problemi
– uscita immediatamente disponibile
– l'output potrebbe non essere stabile a lungo
Non molto utile ⟶ Se l'uscita di una macchina di Mealy passa attraverso della logica combinatoria prima di essere registrata, la logica di controllo potrebbe ritardare il segnale che potrebbe non essere visto in tempo durante il fronte di salita del clock.

Una macchina a stati Moore è più efficiente di una macchina a stati Mealy.

Siccome in Moore la macchina a stati ha uscite sincrone:
– nessun glitch
– un ciclo di ritardo tra input correntemente letto e suo output
– Output stabile durante tutto il ciclo di clock

Stato	0	1
S_0	S_0 / 00	S_2 / 11
S_1	S_0 / 11	S_2 / 00
S_2	S_1 / 01	S_3 / 10
S_3	S_1 / 10	S_3 / 01

Stato	0	1	Output
S_{00}	S_{00}	S_{21}	00
S_{01}	S_{00}	S_{21}	11
S_{10}	S_{01}	S_{20}	01
S_{11}	S_{01}	S_{20}	10
S_{20}	S_{10}	S_{31}	00
S_{21}	S_{10}	S_{31}	11
S_{30}	S_{11}	S_{30}	01
S_{31}	S_{11}	S_{30}	10

Ci si può convincere del fatto che queste descrizioni sono funzionalmente equivalenti, e descrivono perfettamente il comportamento dell'automa a stati finiti delle specifiche.

In una fase di implementazione successiva si è ritenuto opportuno aggiungere, per ogni stato S_{xy} uno stato funzionalmente equivalente ad esso "S_{xy} *done*" sensibile ad un segnale di done in input, in modo da poter mettere in *stand-by* la macchina in attesa di una successiva elaborazione.

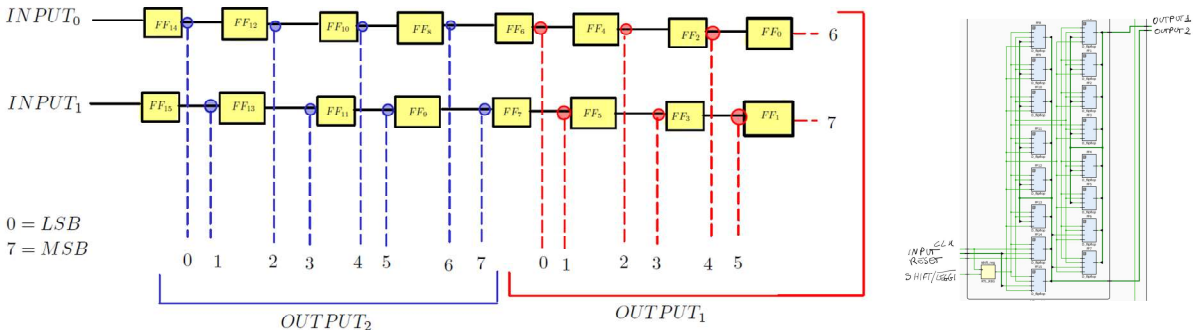
Il codice della macchina a stati di Moore è abbastanza semplice, se ne riporta quindi solo la entity ai morsetti:

```
component moore is

    port(
        clk          : in    std_logic;
        data_in      : in    std_logic;
        done      : in std_logic:='1';
        // se done == 1 la macchina si porta in stand-by;
        reset      : in    std_logic;
        data_out      : out    std_logic_vector(1 downto 0)
    );

end component;
```

2.b Modulo Viterbi*SIPO*: serial in, parallel out

L'idea successiva è stata quella di realizzare un registro a scorrimento, composto da 16 flip flop D, per parallelizzare il flusso dei due bit in output dalla macchina di Moore, che in 8 cicli di clock produrranno i due byte in output.

Dapprima si è provveduto a disegnare su carta i flip flop necessari, per poi numerarare sia i flip flop che i segnali necessari. In questo modo la descrizione del circuito in VHDL diventa abbastanza banale.



Per poter essere pilotato da una logica di controllo occorre inoltre che i flip flop D possegganno un segnale in input di clock enable, per poter decidere se shiftare il contenuto dei registri o bloccare lo shifting e leggere gli output.

Si riporta la entity ai morsetti del registro a scorrimento:

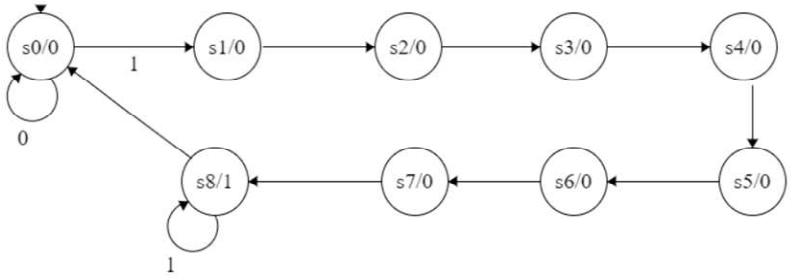
```
entity shift_register is
   port(
    clk: in std_logic;
    reset: in std_logic;
    shift: in std_logic;
     input: in std_logic_vector(1 downto 0);
     output1 : out    std_logic_vector(7 downto 0);
     output2 : out    std_logic_vector(7 downto 0)
    );
end shift_register;
```

Il passo successivo è stato quello di interfacciare i due moduli fino ad ora descritti, la macchina di Moore e il registro a scorrimento. Per fare ciò occorreva un terzo componente, un contatore modulo 8 che esponesse la sequenza di conteggio " 0, 0, 0, 0, 0, 0, 0, 1".

Difatti per riempire il registro a scorrimento servono esattamente 8 cicli di clock, il che coincide perfettamente con gli 8 cicli di clock che servono alla macchina a stati affinché un Byte dalla memoria sia totalmente codificato dall'algoritmo.

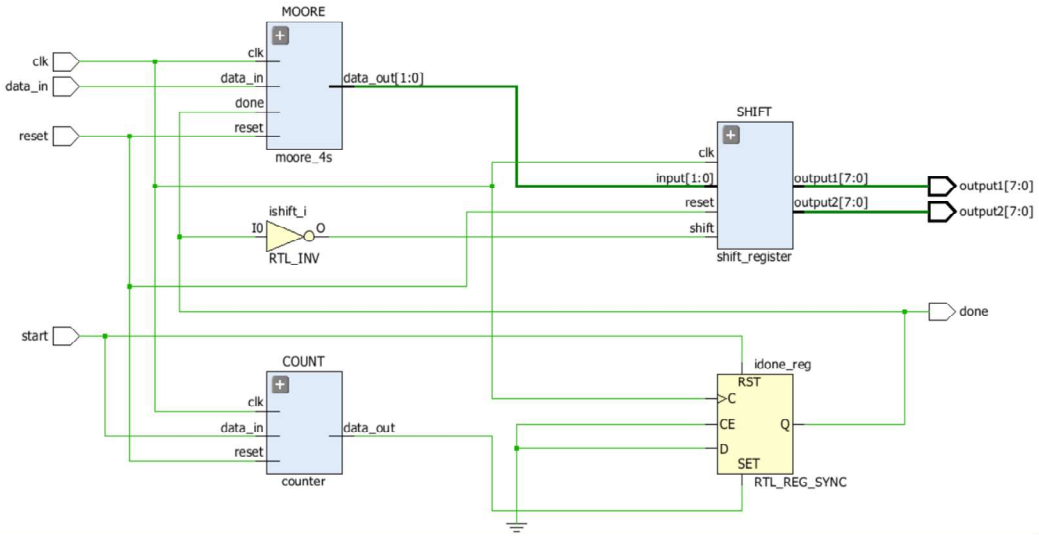
Il contatore modulo 8 possiede in realtà 9 stati, difatti lo stato s0 serve come stato di *Idle*. Una volta che il contatore viene fatto partire con un segnale *start* letto in input posto ad '1' esso non può essere arrestato (questo permette al segnale *start* di poter essere portato a 0 durante gli 8 cicli di clock) fino

a che non arriverà nello stato s8.



Il codice in VHDL è molto semplice, si rimanda quindi al file sorgente per visionarlo.

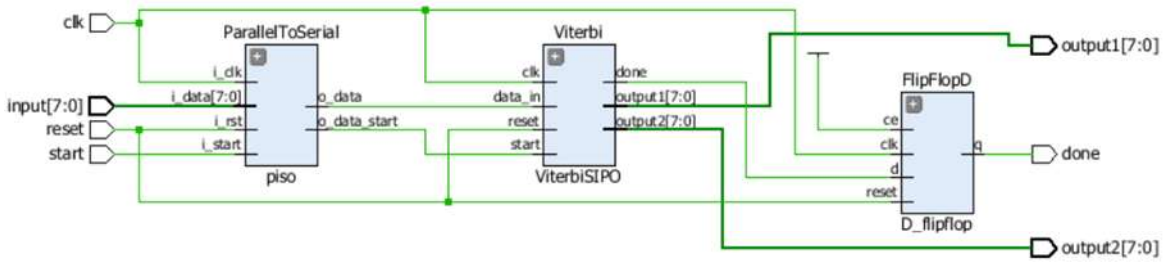
Infine, per ottenere il modulo Viterbi*SIPO* è stato sufficiente interfacciare queste componenti, che lavorano in modo perfettamente sincrono l'uno con l'altro:



Quando *start* = 1 il contatore inizia a contare, esponendo in output (il suo segnale *data_out* in figura) lo 0 logico. Dopo un ciclo di clock (il tempo di setup del flip flop set/reset) la macchina di moore parte nella sua elaborazione (eventualmente uscendo dallo stato di *stand-by* precedente in cui si trovava), e contemporaneamente il registro a scorrimento inizia a shiftare (il bit di controllo dello shift è infatti esattamente la sequenza di conteggio nottata). Quando il contatore arriverà in s8 la macchina di moore, vedendosi portare il bit done in input ad 1, si porterà in uno stato di *stand-by*, e lo shift-register smetterà di far shiftare i bit.

```
entity ViterbiSIPO is
   port(
    clk: in std_logic;
    reset: in std_logic;
    data_in: in std_logic;
    start: in std_logic:='0';
    done: out std_logic:='1';
     output1 : out    std_logic_vector(7 downto 0);
     output2 : out    std_logic_vector(7 downto 0)
    );
end ViterbiSIPO;
```

2.c Modulo Viterbi: parallel input, parallel output



È stato poi necessario introdurre un *Parallel To Serial converter (PISO)*, descritto da un unico process in stile comportamentale, per ottenere il modulo finale utilizzabile poi dalla logica di controllo.

Il protocollo di comunicazione tra questo modulo e la logica di controllo è molto semplice:

- Si prepara il Byte in input e si alza il segnale di start per (almeno) un ciclo di clock

- Si attende che il modulo alzi il segnale di done, che segnala la fine dell'esecuzione dell'algoritmo

- Il modulo fa trovare in uscita i Byte $Output_1$ e $Output_2$, che saranno poi memorizzabili in registri appositi dell'organo di controllo


```
entity Viterbi is
   port(
    clk: in std_logic;
    reset: in std_logic;
    input: in std_logic_vector(7 downto 0);
    start: in std_logic:='0';
    done: out std_logic:='1';
     output1 : out    std_logic_vector(7 downto 0);
     output2 : out    std_logic_vector(7 downto 0)
    );
end Viterbi;
```

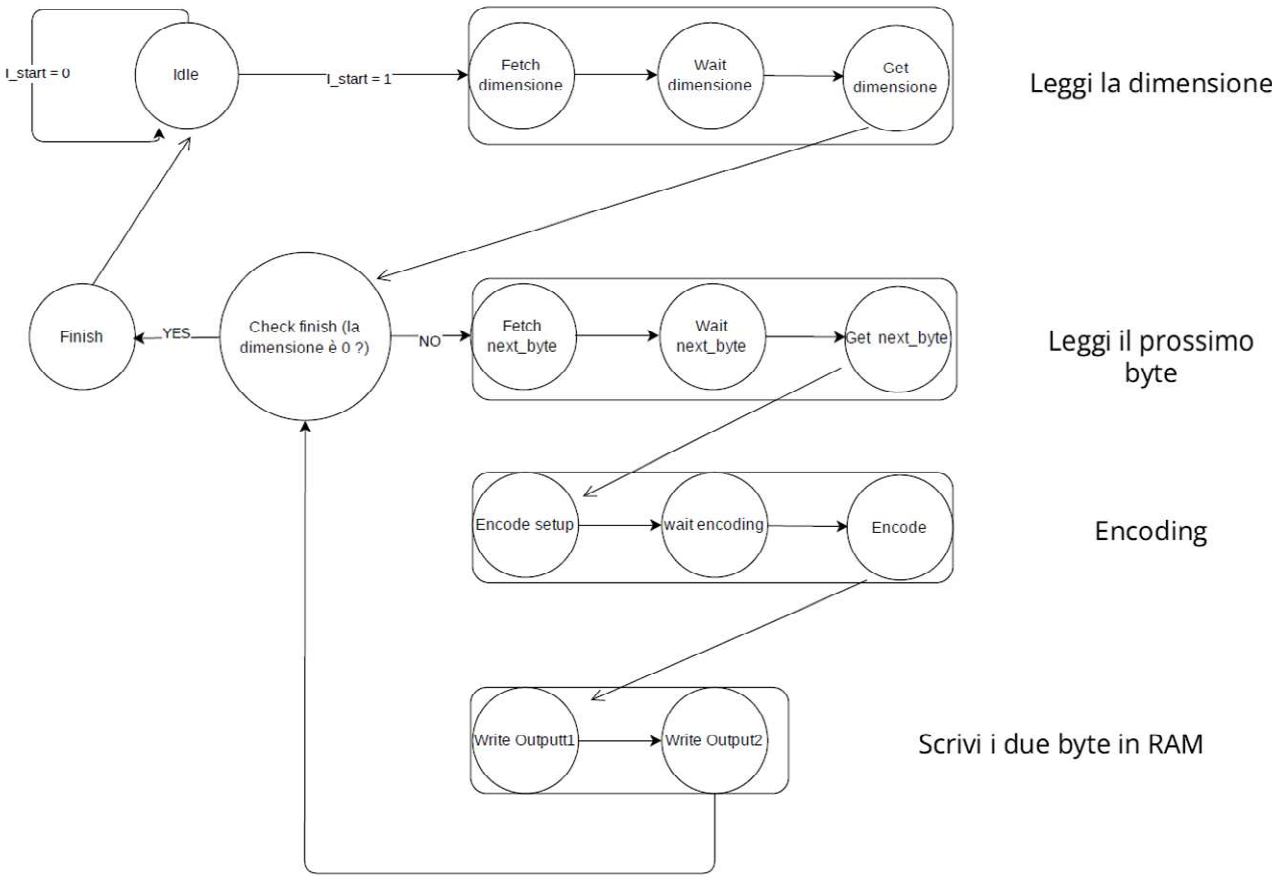
Da una descrizione di un algoritmo di encoding che è per sua natura seriale si è passati ad un protocollo totalmente parallelo, che si presta più facilmente ad essere concretamente utilizzabile dall'organo di controllo che sarà una macchina a stati.

2.d Stati dell'organo di controllo

Per determinare gli stati necessari dell'organo di controllo potrebbe essere utile analizzare il problema con uno pseudocodice:

```
Dimensione ← read RAM(0);
IndirizzoDaLeggere  ← 0;
While(Dimensione ≠ 0){
    IndirizzoDaLeggere ← IndirizzoDaLeggere + 1;
    (output1, output2) ← Calcola_algoritmo(read RAM(IndirizzoDaLeggere));
    write output1 on RAM(1000+IndirizzoDaLeggere);
    write output2 on RAM(1001+IndirizzoDaLeggere);
    Dimensione ← Dimensione - 1;
    }
```

Da questo semplice pseudocodice e dalle specifiche si deducono i seguenti stati:



Idle: È lo stato in cui la macchina è in ascolto del segnale *i_start* in attesa che questo venga portato ad 1.

Fetch_dimension: :La macchina configura i suoi output per fare la richiesta di lettura del Byte 0 alla RAM.

Wait_dimension: Stato necessario per introdurre un clock di attesa per dare il tempo materiale alla RAM di scrivere su *i_data* il contenuto del Byte 0.

Get_dimension: Il contenuto di *i_data* viene copiato in un registro.

Check_finish: Stato che coincide con il while dello pseudocodice: viene valutata la condizione «*dimension == 0*», se si la macchina viene portata in uno stato di Finish, altrimenti si procede a leggere il prossimo Byte della RAM.

Finish: Stato di fine lavorazione.

Fetch_next_Byte: La macchina configura i suoi output per fare la richiesta di lettura del prossimo Byte alla RAM.

Wait_Byte: Stato necessario per introdurre un clock di attesa per dare il tempo materiale alla RAM di scrivere su *i_data*.

Get_next_Byte: Il contenuto di *i_data* viene copiato in un registro.

Encode_setup: Viene dato il segnale di *start* al modulo che calcola l`algoritmo di encoding.

Wait_encoding: Si attende che il modulo di encoding sia effettivamente partito, con la condizione

```
if ( start = '1' or done_encoding = '1') then state_next <= WAIT_ENCODING;
```

Si rimane in *wait encoding* o se la macchina è appena partita (segnale di start alto), o se la macchina non è ancora partita (segnale di done ancora alto, la macchina non ha ancora recepito il segnale di start).

Encode: encoding vero e proprio, si rimane in questo stato per 8 cicli di clock, il tempo di esecuzione dell'algoritmo di encoding

```
if (done_encoding = '0') then state_next <= ENCODE;
                                        else state_next <= WRITE1;  end if;
```

Write$_1$: scrittura in memoria del primo Byte in output dal modulo di encoding

Write$_2$: scrittura in memoria del secondo Byte in output dal modulo di encoding

Durante il testing del programma con i test bench forniti, si è reso necessario aggiungere altri stati per sopperire a problematiche riscontrate.

- In particolare un segnare di reset può essere portato ad '1' dal test bench in maniera del tutto asincrona. Il problema sussiste quando il modulo di encoding è ancora in funzione. Ho sviluppato quindi un meccanismo per fare comunque arrivare a termine l'encoding precedentemente in atto, per non interrompere in maniera "*brusca*" il modulo, con gli stati END_ENCODING ed ENCODED_ENDED.

```
process (i_clk, i_rst)
begin
  if (i_rst = '1') then state_reg <= END_ENCODING;
            [...]
            when END_ENCODING =>
            if (done_encoding = '0') then state_next <= END_ENCODING;
            else
                 reset_fsm_next <= '1';
                 state_next <= ENCODED_ENDED;
            end if;
```

- Sempre in caso di interruzione con un segnale di reset asincrono accadeva che la macchina a stati manteneva comunque la memoria passata dell'ultimo Byte codificato, ho quindi aggiunto degli stati che vadano a resettare il modulo di encoding, ponendo a "00000000" il suo segnale di input. Gli stati così aggiunti sono PADDING, ENCODE_SETUP_FOR_PADDING, WAIT_ENCODING_FOR_PADDING, ENCODE_FOR_PADDING.

```
          when PADDING =>       input_register_next <= "00000000";
                                state_next <= ENCODE_SETUP_FOR_PADDING;


          when ENCODE_SETUP_FOR_PADDING =>
                                  start_next <= '1';
                                  state_next <= WAIT_ENCODING_FOR_PADDING;


           when WAIT_ENCODING_FOR_PADDING =>
             if ( start = '1' or done_encoding = '1')then
                 state_next <= WAIT_ENCODING_FOR_PADDING;
                 start_next <= '0';
             else state_next <= ENCODE_FOR_PADDING;
             end if;



          when ENCODE_FOR_PADDING =>
           if (done_encoding = '0') then
                    state_next <= ENCODE_FOR_PADDING;
           else state_next <= IDLE;
                [...]
                end if;
```

3.a Risultati della sintesi

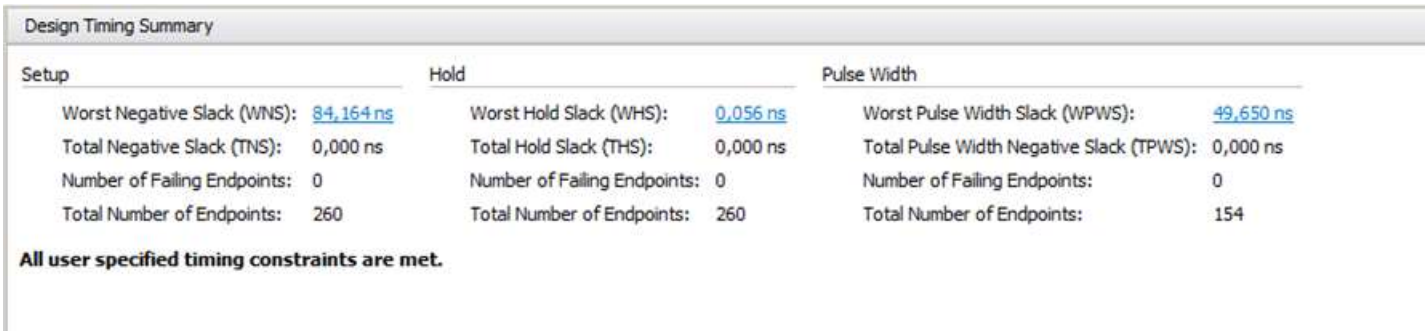
Il componente ottenuto e correttamente sintetizzato supera tutti i test bench forniti sia in *Behavioural Simulation* che in *Post Synthesis Functional Simulation.*

In particolare è interessante notare come il componente sintetizzato sia privo di *Inferred Latch*, il che è sinonimo di un buono stile di programmazione e di progettazione.

```
1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             |  292 |     0 |     41000 |  0.71 |
|   LUT as Logic          |  292 |     0 |     41000 |  0.71 |
|   LUT as Memory         |    0 |     0 |     13400 |  0.00 |
| Slice Registers         |  153 |     0 |     82000 |  0.19 |
|   Register as Flip Flop |  153 |     0 |     82000 |  0.19 |
|   Register as Latch     |    0 |     0 |     82000 |  0.00 |
| F7 Muxes                |    0 |     0 |     20500 |  0.00 |
| F8 Muxes                |    0 |     0 |     10250 |  0.00 |
+-------------------------+------+-------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and ful
```

Anche il numero di Flip Flop utilizzati è veramente esiguo rispetto alle disponibilità della board FPGA utilizzata per la simulazione.

Il componente rispetta inoltre i parametri sul timing di 100 ns. Un parametro importante per valutare la bontà del circuito nel rispettare i vincoli di timing è il *Worst Negative Slack*, presente nel report di Vivado.



Il componente rispetta quindi tutti i vincoli di specifica e supera tutti i test bench forniti.

3.b Simulazioni

Nome test bench .vhd	Behavioural simulation tempo di esecuzione (ns)	Post synthesis functional simulation tempo di esecuzione (ns)
tb_doppio _uguale	14 550 ns	16 250.100 ns
tb_esempio _ 1	1097.500 ns	1232.600 ns
tb_esempio _ 2	2237.500 ns	2372.600 ns
tb_esempio _ 3	1382.500 ns	1517.600 ns
tb_example	5050 ns	6650.100 ns
tb_re _encode	33 950 ns	35 550.100 ns
tb_reset	16 250 ns	15 950.100 ns
tb_seq _max	485 550 ns	487 250.100 ns
tb_seq _min	1050 ns	2750.100 ns
tb_tre _bis	4292.500 ns	4427.600 ns
tb_tre _reset	32 150 ns	32 350.100 ns

Ogni test bench fornito va a testare un particolare *edge case*, tra cui reset asincroni (quelli il cui nome termina in «*_reset*»), la sequenza di dimensione 0 (*tb_seq_min*), la sequenza di dimensione massima (*tb_seq_max*), esecuzioni multiple non variando il contenuto della ram, oltre a molti esempi introduttivi (quelli nel cui nome è presente la parola *esempio*).

Dai test effettuati si evince che l'architettura realizzata si avvicina, in quanto a tempi di esecuzione, all'ottimo teoricamente realizzabile solo quando non vi è la presenza di reset asincroni. Un buon 30% del tempo totale di esecuzione si perde in tal caso per portare a termine la precedente codifica in atto e resettare la macchina a stati.

Si registra una lieve differenza nel tempo di esecuzione tra i test in *behavioural simulation* e quelli in *post synthesis simulation*, con una variabilità massima dell'ordine di 17 cicli di clock (nel caso del test bench *tb_seq_min*).

Sono stati effettuati ulteriori verifiche con *test bench* realizzati in autonomia ed i risultati non discostano molto rispetto a questi.

4. Conclusioni

L' implementazione dell'encoder testé descritta possiede numerosi vantaggi:

· Meccanismo di encoding *glitch-free* evitando di utilizzare muxer e utilizzando una macchina a stati di Moore al posto di una macchina di Mealy.

· Garanzia di codifica da parte del modulo di encoding in esattamente 8 cicli di clock

· Protocollo parallelo di comunicazione con il modulo di encoding (mando in input un Byte, mi ritorna 2 Byte in output).

· Organo di controllo resiliente a segnali di reset asincroni, rendendolo perfettamente funzionante sia in caso di *cold start* (avvio a freddo, ovvero prima accensione della macchina), che in caso di *warm start* (avvio a caldo, resettando la macchina a stati e ricominciando l'esecuzione dell'algoritmo).

Il modulo sintetizzato si rivela quindi compatibile con infrastrutture di comunicazione *safety-critical* (ad esempio comunicazioni radio militari o sistemi di navigazione satellitare).

Riferimenti bibliografici

A. C.E. Shannon, "A mathematical theory of communication," Bell Sys. Tech. J., vol. 27, pp. 379–423 and 623–656, 1948..

B. R. W. Hamming, "Error detecting and correcting codes," Bell Sys. Tech. J., vol. 29, pp. 147–160, 1950.

C. M. J. E. Golay, "Notes on digital coding," Proc. IEEE, vol. 37, p. 657, 1949.

D. A. J. Viterbi, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm," IEEE Trans. on Information Theory, Vol. IT-13, pp. 260-269, April 1967.

E. URL : Intel Corporation, "VHDL Templates for State Machines", https : // www.intel.com / content / www / us / en / support / programmable / support – resources / design – examples / horizontal / vhd – state – machine.html, January 2022

F. URL : Intel Community, "Moore and Mealy", https : // community.intel.com / t5 / Programmable – Devices / Moore – and – Mealy / td – *p* / 164 833, January 2012