Progetto Reti Logiche

Davide Grazzani

Indice

1 Codifiche Convoluzionali e Introduzione al Progetto

Una codifica convoluzionale è un tipo di codifica utilizzata per la *Forward Error Correction* (FEC) in sistemi di telecomunicazioni basati su canali monodirezionali.
Quindi un codice generato da una codifica convoluzionale, detto anche codice convoluzionale, è un codice che trasforma ogni parola P_1 in una parola P_2. Definite $l_1 = lenght(P_1)$ e $l_2 = lenght(P_2)$ si definisce il rapoorto l_1/l_2 come *tasso di trasmissione del convolutore* (rate); $l_2 \geq l_1$. Inoltre la trasformazione è una funzione degli ultimi k bit in entrata, k è quindi la *lunghezza dei vincoli* del codice.
Lo scopo del progetto è quello di implementare un componente hardware, tramite l'utilizzo del linguaggio di specifica dello hardware VHDL, in grado di interfacciarsi con una memoria ram e di applicare una codifica convoluzionale con $rate = \frac{1}{2}$ e $k = 3$.

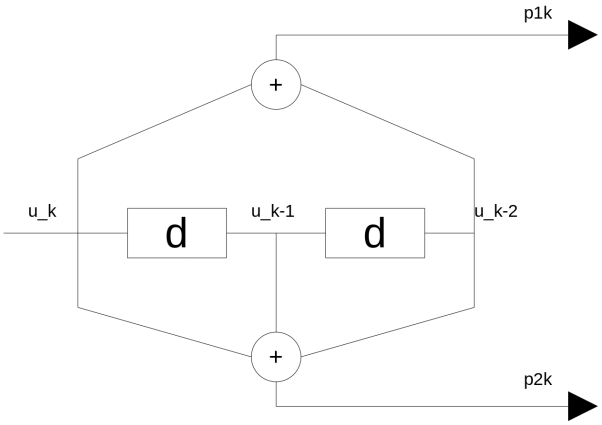

Figura 1: Codificatore convoluzionale con $r = \frac{1}{2}$ e $k = 3$

1.1 Specifiche del progetto

Di seguito viene riportata l'interfaccia del modulo hardware da sviluppare, l'interfaccia della memoria ed infine alcune specifiche progettuali.

1.1.1 Interfaccia del progetto

```
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;
```

1.1.2 Interfaccia della memoria

TODO SISTEMARE QUESTA PARTE

1.1.3 Altri constrains progettuali

Periodo di clock minimo richiesto $clockPeriod_{req} = 100ns$

Ram vedere

2 Architettura, approccio e scielte implementative

In questa sezione verrà descritta la *FSM* del progetto seguita da un rapido overview sul codice presentato. Prima però vengono riportate alcune doverose considerazioni riguardanti il linguaggio di programmazione VHDL.

2.1 Considerazioni su VHDL

In questo progetto, durante la fase di progettazione e successivamente di sviluppo, non verrà preso mai in considerazione l'utilizzo del costrutto *process* e di conseguenza di architetture di tipo *behavioral* . Questa scelta implementativa, che si riflette sia in fase di sintesi che di implementazione, non è dovuta al fatto che l'autore del progetto creda che l'utilizzo di *process* sia scorretto in qual si voglia forma o maniera; qui si vogliono riconoscere le potenzialità e le funzionalità implementative/strutturali che ne derivano dall'utilizzo di quest'ultimi ma si vuole anche risaltare il maggior strato di astrazione portato da questo costrutto rispetto ad architetture *dataflow* o *structural* (aumento presumibilmente dovuto alle serializzazione di istuzioni che per natura fisica di un componente hardware dovrebbero essere paralle).
È per il motivo sopra citato e per la non diretta corrispondenza tra codice scritto e struttura interna del sintetizzato hardware che in questo progetto sono state scartate implementazioni di tipo *behavioral*.

2.2 Primo design della FSM

Tenendo conto delle considerazioni sopra fatte viene ora presentata la prima macchina a stati in grado già di soddisfare ampiamente requisiti di timing sia post sintesi sia post implementazione; viene discussa quest'ultima in quanto alla base del design finale e da considerarsi design ottimale per periodi di clock $clockPeriod \approx [35, 100]ns$.

2.2.1 Stati della macchina

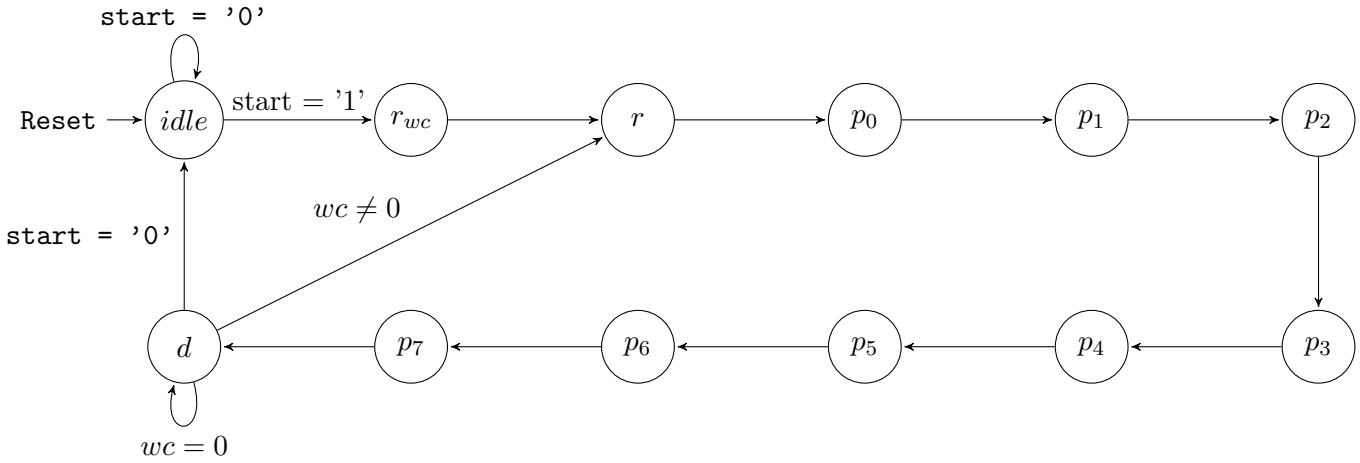

Figura 2: Primo design della *FSM*

con *wc* definito come numero di parole ancora da codificare.

Idle - *idle* **:** stato della macchina iniziale dove questa attende che il segnale `i_start` venga portato alto; una volta che ciò accade in questo stato vengono anche settati `o_en = '1'` e `o_address = 0` in

modo tale da rendere il modulo pronto a ricevere il numero di parole dalla ram. Questo anche coincide con lo stato di reset della FSM.

Read word count - r_{wc} : stato della macchina dove viene settato il numero di parole da codificare.

Read word - r : stato della macchina adibito alla lettura della prossima parola da codificare. In particolare in questo stato vengono settati i valori valori di `o_en = '1'` e `o_address = '1'` in modo tale da leggere la parola da codificare in questo ciclo della FSM

Process - $p_0 \rightarrow p_7$: serie di 8 stati della macchina utilizzati per l'effettiva codifica della parola. Questi servono in particolare a ciclare sul singolo bit della parola considerata, oltre a contribuire alla sincronizzazione dei sottomoduli del progetto(descritto in maniera dettagliata più avanti). Espandiamo specificatamente:

- p_0 : in questo stato viene anche letta la parola richiesta precedentemente nello stato r.
- p_3, p_4 : in questi stati vengono settati `o_en = '1'` e `o_we = '1'` in modo tale da poter parallelizzare la scrittura della parola in memoria con la sua effettiva computazione.

Done - d : stato della macchina che si dedica al controllo del numero di parole rimaste da codificare. Se il numero di parole è $\neq 0$ allora la FSM ritornerà alla stato r altrimenti rimarrà in qiesto stato fintanto che `i_start = '1'`.

2.3 Design finale della FSM

Come precedentemente scritto la macchina sopra specificata superava i test bench per periodi di clock $clockPeriod \approx [15, 100]ns$ nelle simulazione behavioral e post-sintesi ma falliva post-implementazione per $clockPeriod <\approx 35ns$ dove le latenze dell'FPGA erano maggiori.
Possibile soluzione sarebbe quella di aggiungere altri 2 stati così da poter mitigare i ritardi dovuti alla lettura della memoria. Vengono quindi qui sotto riportate delle semplici, seppur doverose, analisi in termini di tempo e di spazio per poter giustificare il cambiamento delle stuttura della macchina a stati.

2.3.1 Analisi di trade-off spaziale

Considerando il numero di stati $numStati = 12$ e utilizzando la codifica binaria $log_2(numStati) \approx 3.6$ quindi si utilizzano 4 bit per la rappresentazione di tali stati. È facile verificare che l'aggiunta di 2 stati, quindi $numStati = 14$, non richiede allocamento aggiuntivo di memoria su FPGA.

2.3.2 Analisi di trade-off temporale

Essendo uno di questi 2 stati aggiuntivi eseguito solo una volta in fase di lettura del numero di parole contenuto in ram esso verrà trascurato perchè asintoticamente irrilevante. Tenendo in considerazione gli stati che vengono eseguiti in *loop*, facendo riferimento alla figura 2 gli stati da r a d, si ottine $numStati_{m1} = 10$ e $numStati_{m2} = numStati_{m1} + 1 = 11$.
Quindi perchè questa modifica alla FSM sia motivata si deve avere che

$$numStati_{m1} * clockPeriod_{m1min} > numStati_{m2} * clockPeriod_{m2}$$

con $clockPeriod_{m1min} = 35ns$.
Si ottiene che $clockPeriod_{m2} \leq 31.82ns$, condizione che verrà poi verificata e discussa nella sezione riguardante i risultati sperimentali.

2.3.3 Nuova macchina a Stati

Vengono così aggiunti altri 2 stati :

Wait word count - w_{wc}

Wait - w

entrambi adibiti alla mitigazione del ritardo dovuto alla lettura della memoria e alla propagazione di tale segnale in fase di implementazione.

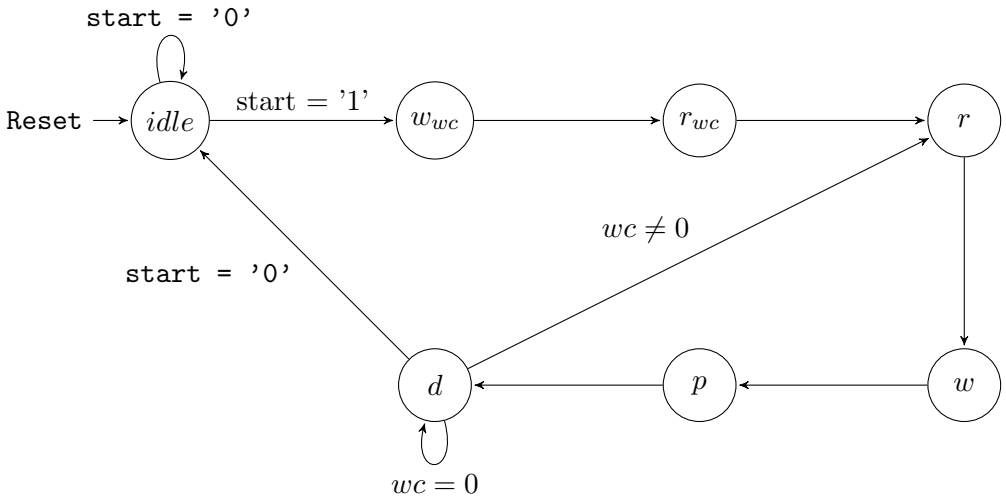

Figura 3: Design finale compatto della *FSM* finale

Nota : In figuara 3 gli stati $p_0, p_1, ..., p_7$ sono compressi in un unico stato p al solo fine di alleggerire la notazione della macchina. Essi rimangono quindi separati come mostrato in figura 2.

2.4 Code overview

L'obbitettivo di questa sezione è quello di mettere in relazione, e quindi eventualmente far chiarezza, le scelte implementative precedentemente introdotte con l'effettivo codice VHDL del progetto.
Il progetto è composto da 3 principali componenti :

- `controller`
- `convolutional_encoder`
- `string_manager`

Successivamente vengono riportate le interfaccie dei componenti e una breve spiegazione sulle funzionalità implementate.

2.4.1 `controller`

È il responsabile per il "comportamento" del componente. Oltre ad essere responsabile per il corretto cycling degli stati, controlla direttamente i segnali di `o_en`, `o_we` e `o_address`.

```
entity controller is
    port(
        clock            : in std_logic; --> i_clock
        reset            : in std_logic; --> i_reset
        start            : in std_logic; --> i_start
        data             : in std_logic_vector (7 downto 0); --> i_data
        done             : out std_logic := '-'; --> o_done
        mem_address      : out std_logic_vector(15 downto 0); --> o_address
        mem_enable       : out std_logic; --> o_en
        mem_write        : out std_logic; --> o_we
        u                : out std_logic; --> bit to be encoded
        component_enable : out std_logic := '0'; --> enable signal for other components
```

```
        component_reset  : out std_logic := '0' --> reset signal for other components
    );
end controller;
```

2.4.2 convolutional_encoder

Parte del componente adibita alla codifica di un singolo bit di una parola; implementa il codificatore convoluzionale mostrato in figura 1.

```
entity convolutional_encoder is
    port(
        u      : in std_logic; --> u
        clock  : in std_logic; --> i_clock
        reset  : in std_logic; --> component_reset
        enable : in std_logic; --> component_enable
        pk     : out std_logic_vector(1 downto 0) --> encoder's output
    );
end convolutional_encoder;
```

2.4.3 string_manager

Questo modulo del componente ha il compito di concatenare i 2 bit di output del `convolutional_encoder` per formare una parola in uscita, da scrivere successivamente in memoria.

```
entity string_manager is
    port(
    clock  : in std_logic; --> i_clock
    reset  : in std_logic; --> component_reset
    enable : in std_logic; --> component_enable
    bits   : in std_logic_vector(1 downto 0); --> pk
    half_z : out std_logic_vector(7 downto 0) --> o_data
    );
end string_manager;
```

2.4.4 Schematico del codice

Viene qui riportata un'immagine che mostra i collegamenti fra le varie entità del codice VHDL. La descrizione di tali collegamenti si può ritrovare nell'architettura, di tipo *structural*, dell'entità `project_reti_logiche`.

3 Risultati sperimentali

In questa sezione verranno discussi tutti i risultati ottenuti attraverso i vari tipi di simulazioni insieme a quali tipi di test siano stati usati per assicurare il corretto funzionamento del componente; successivamente vengono riportati alcuni dati sperimentati direttamente ottenuti dal tool di sintesi ed implementazione *Vivado*.
Nota : tutti i test eseguiti sono stati eseguiti su FPGA *Artix-7 xc7a200tfbg484-1*. Vengono prima riportate tutti i tipi di simulazioni utilizzate : ed infine i tipi di test utilizzati :

- testbench fornite dal professore
- testbench generate da un tool automatico di un collega (link : TODO link da mettere)
- testbench generate da un tool automatico appositamente sviluppato

La copertura dei test verrà discussa in seguito.

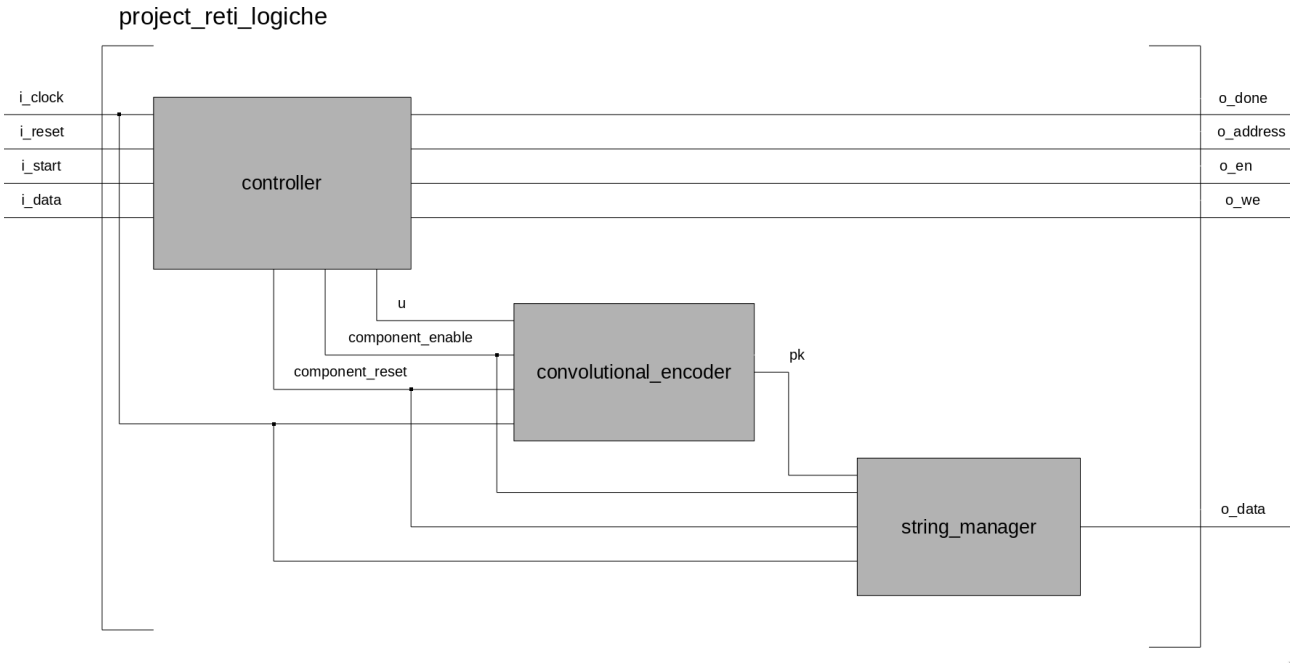

Figura 4: Schematico dei collegamenti tra le varie entità

3.1 Simulazioni

Come precedentemente accennato le simulazioni *behavioral* e quelle di *Post-Synthesis* (sia *Functional* che *Timing*) ottenevano esiti positivi già utilizzando la FSM in figura 2 con un range di clock $clockPeriod \approx [15, 100]ns$. Per assicurare però un minimo standard di qualità si è comunque preferito testare il componente anche in *Post-Implementation*. È proprio dalla neccessità di ottenere un componente in grado di funzionare correttamente in *Post-Implementation* con periodi di clock $clockPeriod \approx 15ns$ che ha portato ad una modifica della prima macchina a stati finiti ottenendo il design finale mostrato in figura 3. In precedenza, discusso al capitolo 2.3.2,