Prova Finale(Progetto Reti Logiche)

Prof. Fabio Salice - Anno Accademico 2021-2022

Matteo Luppi(Codice Persona 10722458 - Matricola 937186)
Alessandro Martinolli(Codice Persona 10700492 - Matricola 933814)

1 Aprile 2022

Indice

1 Introduzione

1.1 Scopo del progetto

Lo scopo del progetto è quello di implementare un componente hardware, descritto in linguaggio VHDL; in particolare, nel nostro caso abbiamo descritto un modulo che riceve in input una sequenza continua di W parole (ognuna da 8 bit) e restituisce in output una sequenza continua di Z parole (sempre da 8 bit). Ogni byte letto in ingresso genera 2 byte in uscita secondo **l'algoritmo di convoluzione** $\frac{1}{2}$, approfondito nella sezione 1.2.

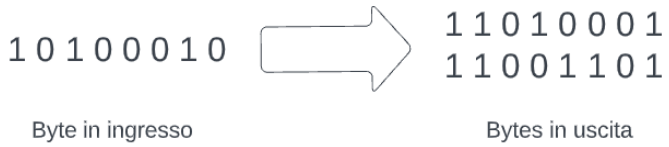


Figura 1: Meccanismo di Convoluzione $\frac{1}{2}$

1.2 L'algoritmo di convoluzione

La codifica convoluzionale, applicata in questo caso, può essere schematizzata nel seguente modo: il flusso di W parole (ognuna formata da 8 bit) viene serializzato, formando un flusso continuo di lunghezza $8 \times W$ bit. Ognuno di questi bit viene codificato in due bit, attraverso il meccanismo descritto dalla macchina a stati rappresentata in figura 2, formando un flusso continuo in uscita di lunghezza $8 \times W \times 2$.

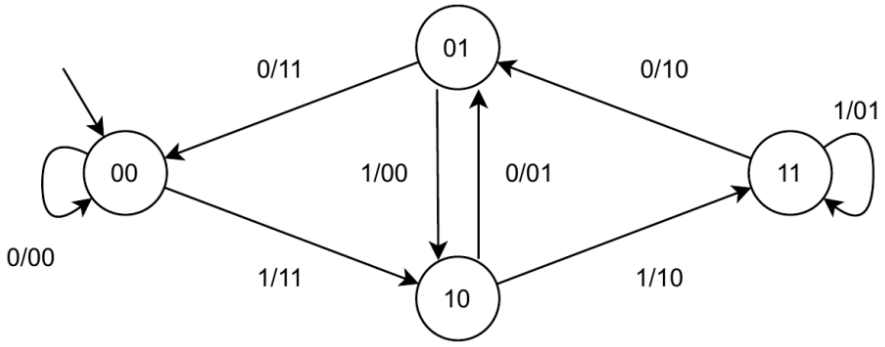


Figura 2: FSM del convolutore

Il flusso in uscita viene successivamente parallelizzato in parole formate da 8 bit, che vanno a formare l'output del nostro componente.
ESEMPIO:
Input: 10100010
Output: 11010001 e 11001101

1.3 Struttura Memoria

Le parole in input andranno lette da una memoria, poi codificate e reinserite nella medesima. Quindi, data la tipologia di codifica, a ogni parola letta corrisponderanno due parole scritte in memoria. Le celle di memoria sono indirizzate al byte, ovvero ogni indirizzo identifica un singolo byte in memoria, e sono suddivise le seguente modo:

- **Cella (0):** contiene l'informazione relativa alla dimensione *dim* dell'ingresso, ovvero il numero di parole (da 8 bit ciascuna) che andranno a formare il flusso in ingresso.
- **Cella (1) ... Cella (*dim*-1):** contengono le parole da leggere e da codificare.
- **Cella(1000) ... Cella(2*dim*-1):** contengono le parole prodotte in seguito all'applicazione dell'algoritmo di convoluzione.

La tipologia e il protocollo della memoria sono dati dalla specifica; infatti l'implementazione di quest'ultima non fa parte del progetto.
Tutta la documentazione è reperibile al seguente **link**.

1.4 Interfaccia Componente

L'interfaccia del componente viene assegnata dalla specifica del progetto ed è così definita:

```
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 segnali che vediamo qua definiti possono essere divisi in segnali di input e segnali di output:

INPUT:

- **i_clk** Segnale di CLOCK.
- **i_rst** Segnale di RESET che prepara la macchina alla ricezione del segnale di START.
- **i_start** Segnale di START che avvia la macchina.

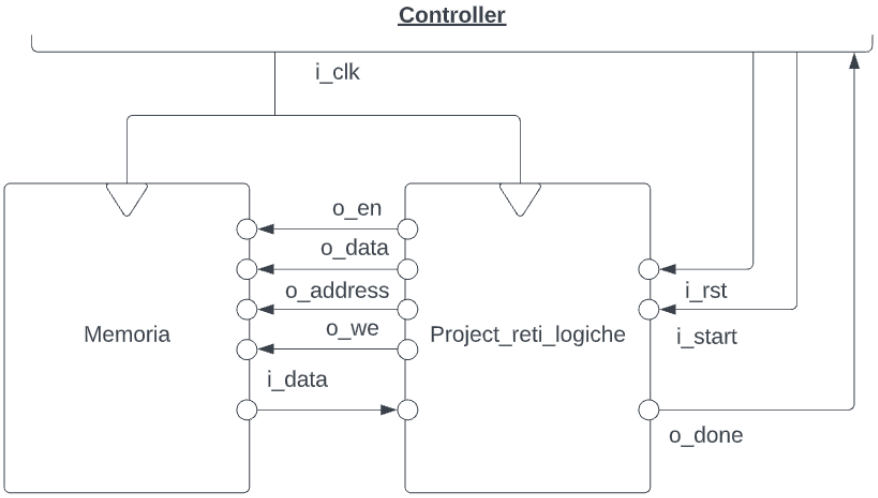
- **i_data** Segnale (vettore) che porta l'informazione dalla memoria dopo la richiesta di lettura.

OUTPUT:

- **o_address** Segnale che manda l'indirizzo alla memoria.
- **o_done** Segnale DONE che comunica la fine della procedura.
- **o_en** Segnale ENABLE che abilita la comunicazione con la memoria sia in lettura che in scrittura.
- **o_we** Segnale WRITE ENABLE che abilita la scrittura in memoria. In fase di lettura deve essere pari a 0.
- **o_data** Segnale (vettore) che invia l'informazione alla memoria dopo la richiesta di scrittura.

1.5 Utilizzo dei Segnali dell'Interfaccia

Il modulo inizierà l'elaborazione quando il segnale **i_start** viene portato a 1. Il segnale di **i_start** rimarrà alto fino a che il segnale **o_done** non verrà portato a 1, cosa che avverrà solo a fine elaborazione. Il segnale **o_done** una volta alzato rimarrà alto fino a che il segnale **i_start** non sarà riportato a 0. Se il segnale **i_start** venisse rialzato il modulo dovrà ripartire con la codifica. Prima di poter utilizzare il modulo è necessario resettarlo tramite il segnale **i_rst**; però successive elaborazioni oltre alla prima non richiedono tale passaggio.



2 Architettura

Dal punto di vista architetturale, abbiamo deciso di descrivere il modulo in esame implementando una macchina a stati finiti (FSM). Inoltre, dal momento che non si trattava di un problema di complessità elevata, abbiamo optato per una soluzione con un singolo process dettato da un segnale di clock.

2.1 Funzionamento

La macchina a stati va inizialmente a leggere dalla memoria la quantità di parole da codificare. Una volta fatto ciò inizia a leggere ogni singolo byte per processarlo e codificarlo secondo la codifica spiegata nella sezione 1.2.
Terminata la codifica del byte si procede ad una fase di scrittura in memoria. L'intero processo si conclude quando sono finiti i byte da leggere.

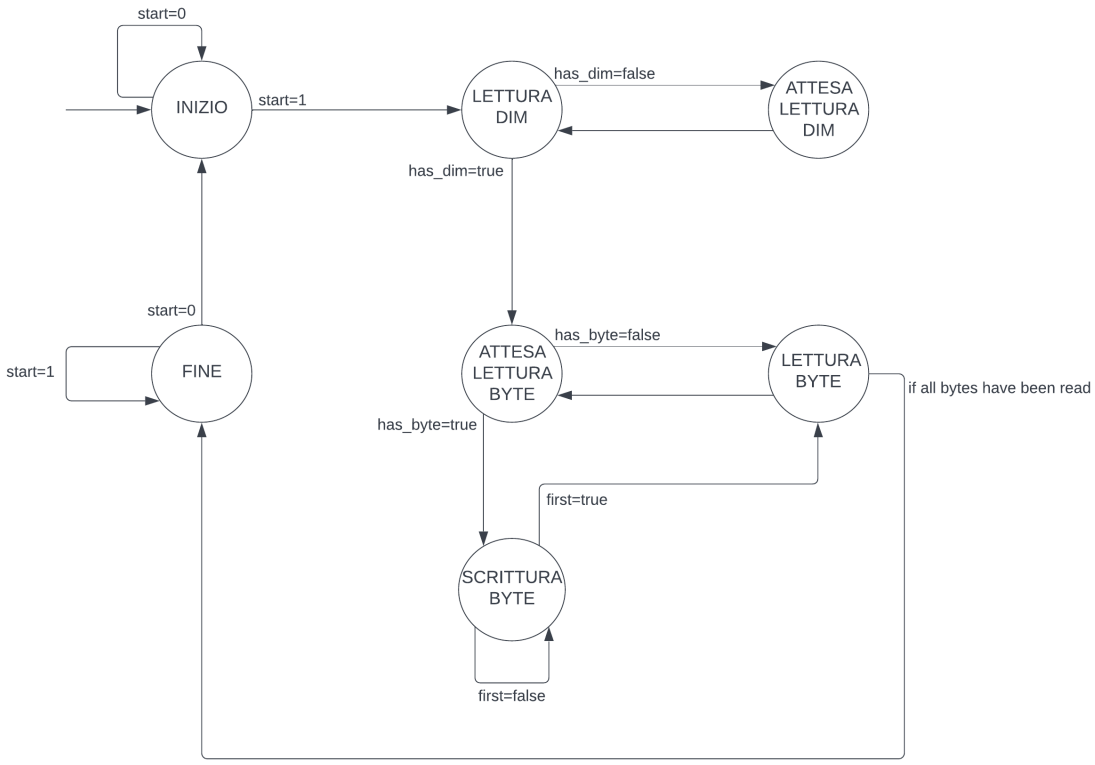


Figura 3: FSM Progetto

2.2 Stati della Macchina

La macchina in questione è composta da 7 stati. Di seguito è fornita una breve descrizione per ognuno di essi.

2.2.1 INIZIO

Stato iniziale nel quale si attende che il segnale **i_start** venga alzato. Inoltre nel caso venga alzato il segnale di **i_rst** si torna in questo stato.

2.2.2 LETTURA DIM

Stato in cui viene richiesta la lettura della dimensione in input, cioè il numero di parole da 8 bit da leggere. Se la dimensione è nulla non parte nessuna codifica, si alza il segnale **o_done** e si va nello stato 2.2.7.

2.2.3 ATTESA LETTURA DIM

Stato in cui si attende la risposta della memoria a seguito della richiesta di lettura fatta nello stato 2.2.2. [1]

2.2.4 LETTURA BYTE

Stato in cui vado a determinare l'indirizzo del byte da leggere che viene poi assegnato a **o_address**. Se ho letto tutti i byte alzo il segnale **o_done** e vado nello stato 2.2.7.

2.2.5 ATTESA LETTURA BYTE

Stato in cui si attende la risposta della memoria a seguito della richiesta di lettura di un byte fatta nello stato 2.2.4. [1]

2.2.6 SCRITTURA BYTE

Stato in cui viene svolta la codifica del byte letto secondo l'algoritmo di convoluzione spiegato nella sezione 1.2. Ogni byte letto genera due byte in uscita che poi vengono scritti in memoria a due indirizzi differenti ma contigui specificati dal segnale **o_address**.

2.2.7 FINE

Stato in cui si attende che **i_start** venga abbassato per poi in seguito abbassare **o_done** e tornare nello stato 2.2.1.

[1] Tale stato è necessario dal momento che la memoria in questione, una volta ricevuta una richiesta di lettura, impiega un ulteriore ciclo di clock a fornire la risposta.

2.3 Registri Interni della Macchina

Per la descrizione della nostra macchina ci siamo serviti dei seguenti registri:

```
signal has_dim: boolean := false;      --flag che indica se la dimensione è stata letta dalla memoria(indirizzo 0)
signal has_byte: boolean := false;     --flag che indica se il byte è stata letto dalla memoria
signal first: boolean := false;        --flag per la scrittura dei due byte in uscita
signal last_byte_address: std_logic_vector(15 downto 0 ) := (others => '0');    --indirizzo dell'ultimo byte letto
signal current_byte_address: std_logic_vector(15 downto 0 ) := (others => '0'); --indirizzo del byte corrente
signal stato_conv: std_logic_vector(1 downto 0) := "00"; --segnale che mi rappresenta gli stati del convolutore
signal offset_999: std_logic_vector(15 downto 0):= "0000001111100111"; --offset che mi aiuta per la scrittura in memoria
```

Dove:

- **has_dim** Tale registro indica se il numero di parole in input è stato letto.
- **has_byte** Tale registro indica se il byte è stato letto.
- **first** Tale registro aiuta nella scrittura dei due byte in memoria.
- **last_byte_address** Tale registro indica l'indirizzo dell'ultimo byte.
- **current_byte_address** Tale registro indica l'indirizzo del byte corrente, quello che stiamo codificando.
- **stato_conv** Tale registro permette di rappresentare i 4 stati del convolutore come si nota in figura 2.
- **offset_999** Tale registro permette di avere il giusto offset quando scriviamo i byte in memoria.

2.4 Variabili del Processo

All'interno del process abbiamo utilizzato le seguenti variabili di supporto:

```
--variabile di supporto
variable var : unsigned(15 downto 0) := (others => '0');
--variabile per memorizzare i due byte di uscita dal convolutore per ogni byte in ingresso
variable uscita: std_logic_vector(0 to 15) := (others => '0');
--contatori interi
variable count : integer := 7; --per scorrere tutti i bit del byte che leggo-->va da 7 a 0
variable i : integer := 0; --per scrivere tutti i bit nel vettore uscita -->va da 0 a 15
variable z : integer := 0; --per aiutarmi a scrivere i due byte al giusto indirizzo di memoria
```

Dove:

- **var** Variabile di supporto. Vado a salvarmi prima il numero di parole in input, poi l'indirizzo del byte successivo.
- **uscita** Variabile in cui vado a salvarmi i due byte di output(dato che ogni byte in ingresso ne genera due in uscita, come ben rappresentato in figura 1).
- **count** Variabile che utilizzo per scorrere ogni singolo bit del byte appena letto.
- **i** Variabile che utilizzo come contatore per scrivere nella variabile 'uscita'.
- **z** Variabile che utilizzo per scrivere i due byte di uscita al corretto indirizzo di memoria.

3 Risultati sperimentali

3.1 Sintesi

Il nostro componente è correttamente simulabile ed implementabile dal tool VIVADO; a questo proposito sono stati utilizzati 127 FF e 246 LUT (rispettivamente lo 0.05% e lo 0.18% del totale, come si può notare a figura 5). Inoltre, abbiamo prestato particolare attenzione nella scrittura del codice per evitare l'inserimento di *latch*, componenti asincroni che spesso generano comportamenti indesiderati e loop infiniti per il fatto che non sono controllati da un segnale di clock.

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF
synth_1	constrs_1	synth_design Complete!								283	127
impl_1	constrs_1	route_design Complete!	91.752	0.000	0.224	0.000	0.000	0.131	0	246	127

Figura 4: Sintesi

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs	246	0	134600	0.18
LUT as Logic	246	0	134600	0.18
LUT as Memory	0	0	46200	0.00
Slice Registers	127	0	269200	0.05
Register as Flip Flop	127	0	269200	0.05
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Figura 5: Report Utilization

3.2 Simulazioni

Per verificare il corretto funzionamento del nostro componente, lo abbiamo testato più volte con dei TestBench. Abbiamo simulato il componente sia in *Behavioral Simulation* che in *Post-Sintesis Functional Simulation*.
Inizialmente abbiamo utlizzato i test che ci sono stati forniti dal docente, però in seguito per ottenere una maggiore copertura, abbiamo optato per generarli con uno script python.
Di seguito sono riportati i test a nostro avviso più rilevanti, in termini di funzionamento e di copertura di corner case.[2]

3.2.1 Test 1: 'tb_esempio1.vhd'

Test che va a coprire un caso generale verificando se il componente descritto funziona correttamente in uno scenario banale.

[2]Andremo in particolar modo a descrivere i test fornitoci dal docente dato che coprono buona parte dei casi interessanti

3.2.2 Test 2: 'tb_seq_max.vhd'

Test che va a verificare il caso particolare in cui il numero di parole da leggere è massimo, cioè pari a 255 parole. In output avremo 510 parole da 8 bit.

3.2.3 Test 3: 'tb_seq_min.vhd'

Test che va a verificare il caso particolare in cui il numero di parole da leggere è pari a 0. In questo caso il componente non deve effettuare alcuna codifica nonostante la presenza di parole in memoria.

3.2.4 Test 4: 'tb_reset.vhd'

Test che va a verificare il corretto funzionamento del componente nel particolare scenario in cui un processo viene interrotto dal segnale **i_rst** e un secondo viene avviato.

3.2.5 Test 5: 'tb_doppio_uguale.vhd'

Test che va a verificare il caso particolare in cui effettuo due volte la stessa lettura e scrittura sulla stessa ram senza che che venga alzato il segnale di **i_rst**.

3.2.6 Test 6: 'tb_re_encode.vhd'

Test che va a verificare il caso di 3 flussi codificati uno dopo l'altro. La quantità di parole da leggere nei tre casi è identica e pari a 5; quello che cambia sono le parole lette e scritte.

3.2.7 Test 7: 'tb_tre_reset.vhd'

Test simile a quello descritto nella sezione 3.2.6 con la differenza che viene alzato il segnale di **i_rst** sul primo flusso. Inoltre in questo test la quantità di parole da leggere è sempre pari a 3.

4 Conclusioni

Si ritiene che l'architettura descritta rispetti *in toto* la specifica: infatti, il componente ha superato numerosi test che gli sono stati sottoposti sia in *Behavioral Simulation* che in *Post-Sintesis Functional Simulation*. Inoltre, tale architettura evita completamente l'utilizzo di *latch* che avrebbero potuto creare anomalie e cicli infiniti all'interno della nostra implementazione.
Per concludere, la scelta di optare per una soluzione monomodulare (a singolo process) ci ha permesso di ottenere una struttura del codice pulita, logica e chiara.