



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Progetto di Reti Logiche 2021-22

PROVA FINALE
RETI LOGICHE

Alessandro Sironi

Studente: Alessandro Sironi
Professore: Palermo Gianluca
Anno Accademico: 2021-22

Contents

Contents	i
1 Specifica del progetto	1
1.1 Algoritmo	1
1.2 Memoria	2
1.3 Protocollo di Start e Reset	2
1.4 Interfaccia del componente	2
2 Architettura e Implementazione	5
2.1 Algoritmo	5
2.2 Processi	5
2.3 Finite State Machine	6
2.4 Interfaccia con la memoria	6
2.5 Schematico RTL dell'implementazione	7
3 Risultati sperimentali	9
3.1 Testbench	9
3.1.1 Testbench generico	9
3.1.2 Testbench aggiuntivi	10
3.2 Report di Sintesi	10
3.2.1 Slice Logic	10
3.2.2 Clock	11
4 Conclusioni	13

1 | Specifica del progetto

La specifica della Prova Finale (Progetto di Reti Logiche) 2021-22 è ispirata alle **codifiche convoluzionali**. Nelle telecomunicazioni esistono meccanismi applicati su canali monodirezionali di rilevazione e successiva correzione degli errori, a valle di una trasmissione digitale. Questi sono basati su un'opportuna codifica di canale e sull'aggiunta di bit di ridondanza al flusso monodirezionale informativo, funzionali all'elaborazione da parte di appositi algoritmi. In questo progetto viene richiesto lo sviluppo di un codice convoluzionale

$$\text{Rate} = \frac{1}{2}, \quad (1.1)$$

ovvero: per ogni bit in ingresso, ne vengono generati due in uscita.

Il tool che è stato utilizzato per lo svolgimento del progetto è **Xilinx Vivado**, con FPGA target xc7a12ticsg325-1L.

1.1. Algoritmo

Il modulo da implementare deve presentare il seguente comportamento:

$$\begin{cases} P_1(t) = U(t) \ xor \ U(t - 2) \\ P_2(t) = U(t) \ xor \ U(t - 1) \ xor \ U(t - 2) \\ Y(t) : \text{Concatenazione tra } P_1 \ e \ P_2 \end{cases} \quad (1.2)$$

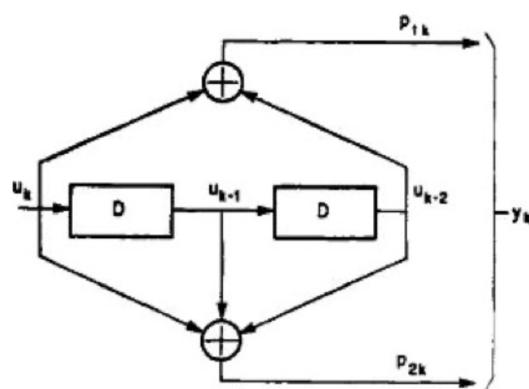


Figure 1.1: Schema di funzionamento

1.2. Memoria

Il modulo da implementare deve leggere la sequenza di dati da una memoria (istanziata dal Testbench) con indirizzamento al byte. Gli indirizzi sono espressi in binario (16bit). Il primo dato - all'indirizzo 0 - contiene il numero di byte da codificare. I successivi input sono memorizzati dall'indirizzo 1, con una dimensione massima di 255 byte. L'output deve essere memorizzato sequenzialmente a partire dall'indirizzo 1000 (qui espresso in decimale).

1.3. Protocollo di Start e Reset

Il modulo deve prevedere il rispetto del protocollo di Start-Reset-Done seguito dal Testbench. Innanzitutto, precedentemente alla prima codifica verrà sempre fornito in ingresso al modulo un segnale di reset.

Successivamente, le scelte implementative devono prevedere la codifica di più sequenze, che non dovranno attendere il segnale di reset.

Il Testbench si pone in stato di attesa, aspettando che il modulo implementato alzi il segnale di output `o_done` ad elaborazione completata. Viene qui riportato il protocollo descritto per via grafica.

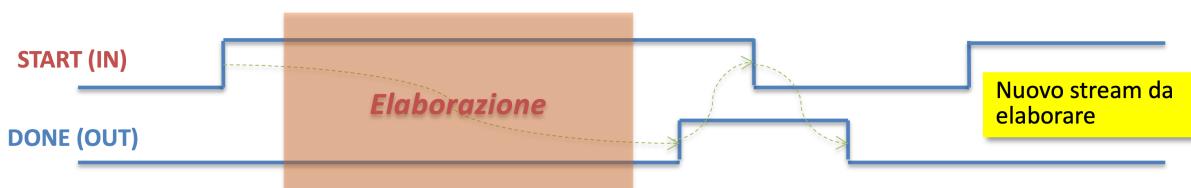


Figure 1.2: Protocollo Start - Done

1.4. Interfaccia del componente

Il componente deve definire la seguente interfaccia che descrive le porte di input e di output del componente:

```
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;
```

Segnali di input:

- **i_clk**: Segnale di clock.
- **i_RST**: Segnale di reset.
- **i_start**: Comanda l'inizio del processo di elaborazione del modulo.
- **i_data**: Vettore (8 bit) che contiene, dopo opportuna richiesta, il byte in memoria richiesto dal modulo.

Segnali di output:

- **o_address**: Contiene l'indirizzo di memoria (16 bit) che il modulo richiede di utilizzare (sia in modalità lettura che scrittura).
- **o_done**: Permette al modulo di segnalare al Testbench la fine dell'elaborazione del set di byte.
- **o_en**: Segnale che il modulo utilizza per attivare la memoria.
- **o_we**: Segnale per specificare l'intento di utilizzo della memoria (0 per la lettura, 1 per la scrittura).
- **o_data**: Segnale (8 bit) che contiene il dato elaborato da scrivere in memoria all'indirizzo **o_address** se e solo se **o_en** = 1 e **o_we** = 1.

2 | Architettura e Implementazione

2.1. Algoritmo

L'algoritmo sviluppato per l'elaborazione dei dati è organizzato per fasi, scandite da un signal interno **step**:

1. **Fase di preparazione** - $step \in [-1, 5]$: i segnali di output vengono portati agli stati consoni ($o_{en} = 1$ e $o_{we} = 0$), tutti i signal interni vengono inizializzati opportunamente (tra cui anche gli indirizzi di memoria di lettura e scrittura), e viene letto il numero di byte da codificare all'indirizzo 0 (operazione effettuata solo una volta all'interno della stessa elaborazione). Viene letto il dato dalla memoria, e viene impostato il primo bit di elaborazione.
La fase '-1' è di servizio e viene utilizzata per allineare il comportamento tra il componente e la memoria quando si comincia una nuova elaborazione.
2. **Fase di elaborazione** - $step \in [6, 13]$: il byte corrente viene elaborato dalla FSM, bit a bit, scrivendo i risultati nei signal **p1k** e **p2k**.
3. **Fase di scrittura** - $step \in [14, 17]$: i segnali interni **p1k** e **p2k** vengono concatenati e scritti nelle due posizioni adiacenti in memoria ($o_{we} = 1$). Se non sono ancora stati elaborati tutti i byte, itera, altrimenti segnala al Testbench e alla memoria la conclusione dell'elaborazione ($o_{done} = 1$).

Per la realizzazione di questo algoritmo e progetto è stata scelta un'implementazione *behavioral*, avvalendosi di una **FSM - Finite State Machine** per gestire l'elaborazione.

2.2. Processi

Come decisione progettuale, si è scelto di implementare la maggior parte del comportamento in un process con *sensitivity list* (**i_clk**, **i_RST**).

Il process **FSM_PROC**, infatti, si occupa di gestire l'avanzamento del comportamento opportunamente incrementando e impostando il segnale **step**, la lettura del byte di input, la scrittura in output, la verifica dell'eventuale conclusione di elaborazione e il rispetto del protocollo Start-Done-Reset.

Viene impiegato un secondo process **next_state** di supporto, il cui scopo è quello di effettuare l'assegnamento **PS <= NS**; necessario per far muovere la macchina a stati finiti.

Entrambi i processi sono sincronizzati con il clock. Il primo processo agisce sul fronte di salita del clock, mentre invece il secondo agisce sul fronte di discesa (verificando che sia comunque in atto la codifica).

2.3. Finite State Machine

La FSM utilizzata per l'elaborazione del segnale convoluzionale di uscita è quella data nelle specifiche, qui riportata.

La FSM di elaborazione del dato presenta 4 stati: S00, S01, S10, S11. Lo stato iniziale

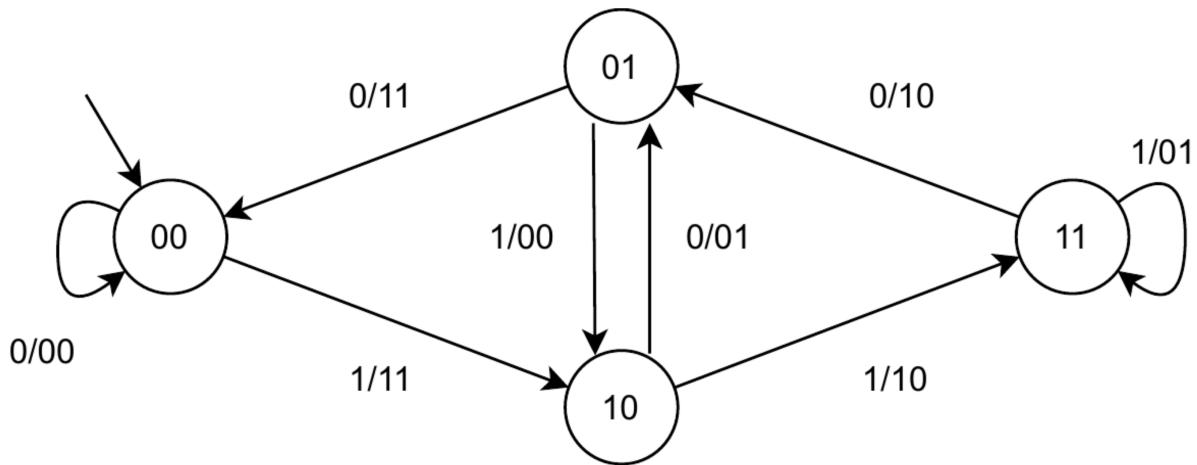


Figure 2.1: FSM

della Macchina a Stati Finiti è S00.

Ad ogni transizione, che avviene per ogni bit letto, ne vengono prodotti due in uscita e viene memorizzato il passo successivo (NS - Next Step) da compiere al successivo ciclo di clock.

Il primo bit generato viene memorizzato in p1k, mentre il secondo in p2k.

2.4. Interfaccia con la memoria

A seconda della fase in cui si trova il processo, i segnali di supporto `o_en` e `o_we` vengono alzati o abbassati, per indicare il comportamento desiderato: lettura o scrittura. Gli indirizzi di lettura e scrittura in memoria vengono mantenuti tramite due signal interni `mem_address_read` e `mem_address_write`, inizializzati rispettivamente a:

```

mem_address_read <= "0000000000000000";
mem_address_write <= std_logic_vector(to_unsigned(1000,16));
  
```

Quest'ultimo assegnamento effettua un cast, traducendo il valore 1000_{DEC} nel corrispondente valore in binario. Questi segnali vengono opportunamente incrementati ad ogni lettura e scrittura effettuata.

2.5. Schematico RTL dell'implementazione

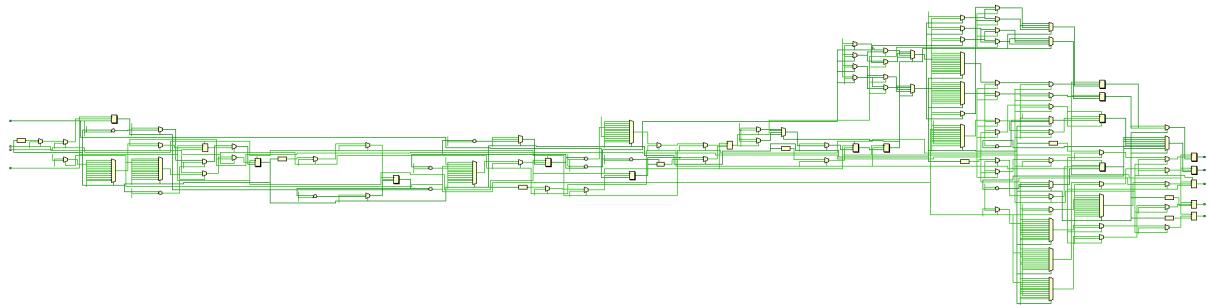


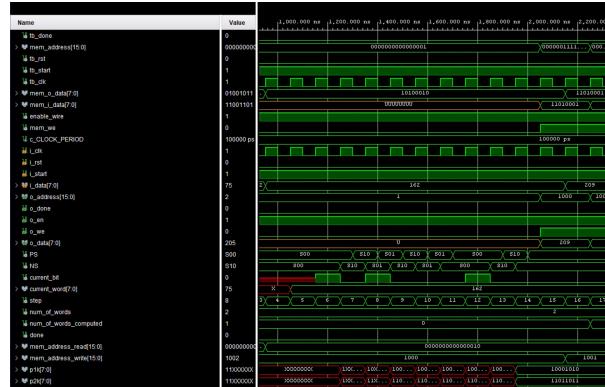
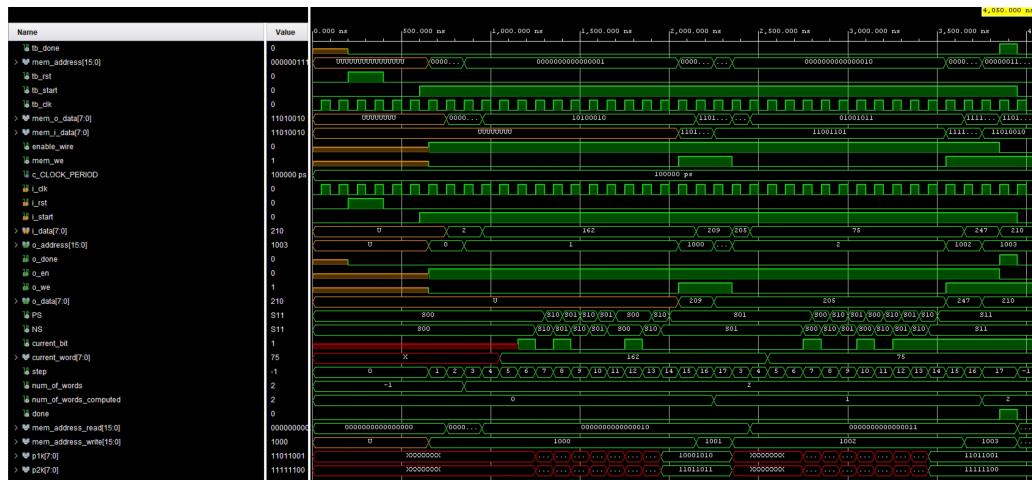
Figure 2.2: Schema dell'implementazione

3 | Risultati sperimentali

3.1. Testbench

3.1.1. Testbench generico

Il componente è stato sottoposto al Testbench fornito insieme alle specifiche, per verificare il corretto funzionamento. Vengono di seguito riportate le forme d'onda generate in fase di simulazione:



3.1.2. Testbench aggiuntivi

Di seguito vengono elencati altri Testbench, il cui obiettivo è di testare specifiche funzionalità e casi limite di comportamento. Il componente sintetizzato supera tutti i seguenti test, sia in simulazione *Behavioral* che *Post-Synthesis*:

- **Doppio uguale:** Viene effettuato un doppio processing sulla stessa RAM.
- **Re-Encode:** Viene eseguita la codifica di più flussi (tre), uno dopo l'altro.
- **Reset:** Viene verificato il corretto comportamento a fronte di un reset asincrono.
- **Sequenza massima:** Viene richiesto al componente di codificare una sequenza di lunghezza massima, ovvero 255 byte ($\text{RAM}(0) = "11111111"$).
- **Sequenza minima:** Viene testato il comportamento a fronte di una sequenza di lunghezza nulla ($\text{RAM}(0) = "00000000"$).
- **Tre Bis:** Al componente viene richiesto di scrivere in ciascuna delle tre diverse RAM presenti nel Testbench. Il segnale di reset viene asserito soltanto all'inizio.
- **Tre Reset:** Similmente al Test "Tre Bis", prevede tre codifiche consecutive; viene però asserito un reset a valle della prima codifica.

3.2. Report di Sintesi

3.2.1. Slice Logic

Di seguito viene riportato l'utilizzo dei componenti dopo la sintesi. Questi dati sono stati ottenuti attraverso il comando `report_utilization` nella console TCL di Vivado.

1. Slice Logic						

Site Type	Used	Fixed	Prohibited	Available	Util%	
Slice LUTs*	307	0	0	8000	3.84	
LUT as Logic	307	0	0	8000	3.84	
LUT as Memory	0	0	0	5000	0.00	
Slice Registers	193	0	0	16000	1.21	
Register as Flip Flop	193	0	0	16000	1.21	
Register as Latch	0	0	0	16000	0.00	
F7 Muxes	1	0	0	7300	0.01	
F8 Muxes	0	0	0	3650	0.00	

Figure 3.3: Report di Utilizzo.

Si è prestata particolare attenzione nella scrittura del codice per evitare l'utilizzo di Latch.

3.2.2. Clock

La specifica del progetto richiede che il componente funzioni con un periodo di clock di almeno 100ns.

Attraverso la creazione di un constraint è stato possibile verificare che il componente sintetizzato richiede un periodo di clock minimo di circa **52,5 ns**.

4 | Conclusioni

Il progetto supera correttamente tutti i test precedentemente descritti, sia in simulazione *Behavioral* che *Post-Synthesis*, rispettando i vincoli previsti per il suo funzionamento, come il periodo di clock minimo. L'architettura è stata pensata sulla base di una FSM, evitando sempre l'utilizzo di Latch che, per asincronismo con il clock, avrebbero potuto instaurare cicli infiniti.