Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria

Progetto di Reti Logiche
A.A. 2021/2022

Prof. Palermo Gianluca

Dario Simoni
(Codice Persona: 10697990, Matricola: 932957)

Indice

1 Requisiti di progetto

1.1 Descrizione del problema

Si vuole realizzare un modulo HW (descritto in VHDL) che si interfacci con una memoria e che ricevendo in ingresso un flusso continuo X da 1 bit restituisca un nuovo flusso continuo Y da 1 bit.

Il flusso continuo X è generato da una seguenza continua di parole serializzate in base alla specifica, mentre il flusso continuo Y restituisce il doppio dei bit rispetto al flusso X.

1.2 Interfaccia del componente

Il componente da descrivere deve avere la seguente interfaccia

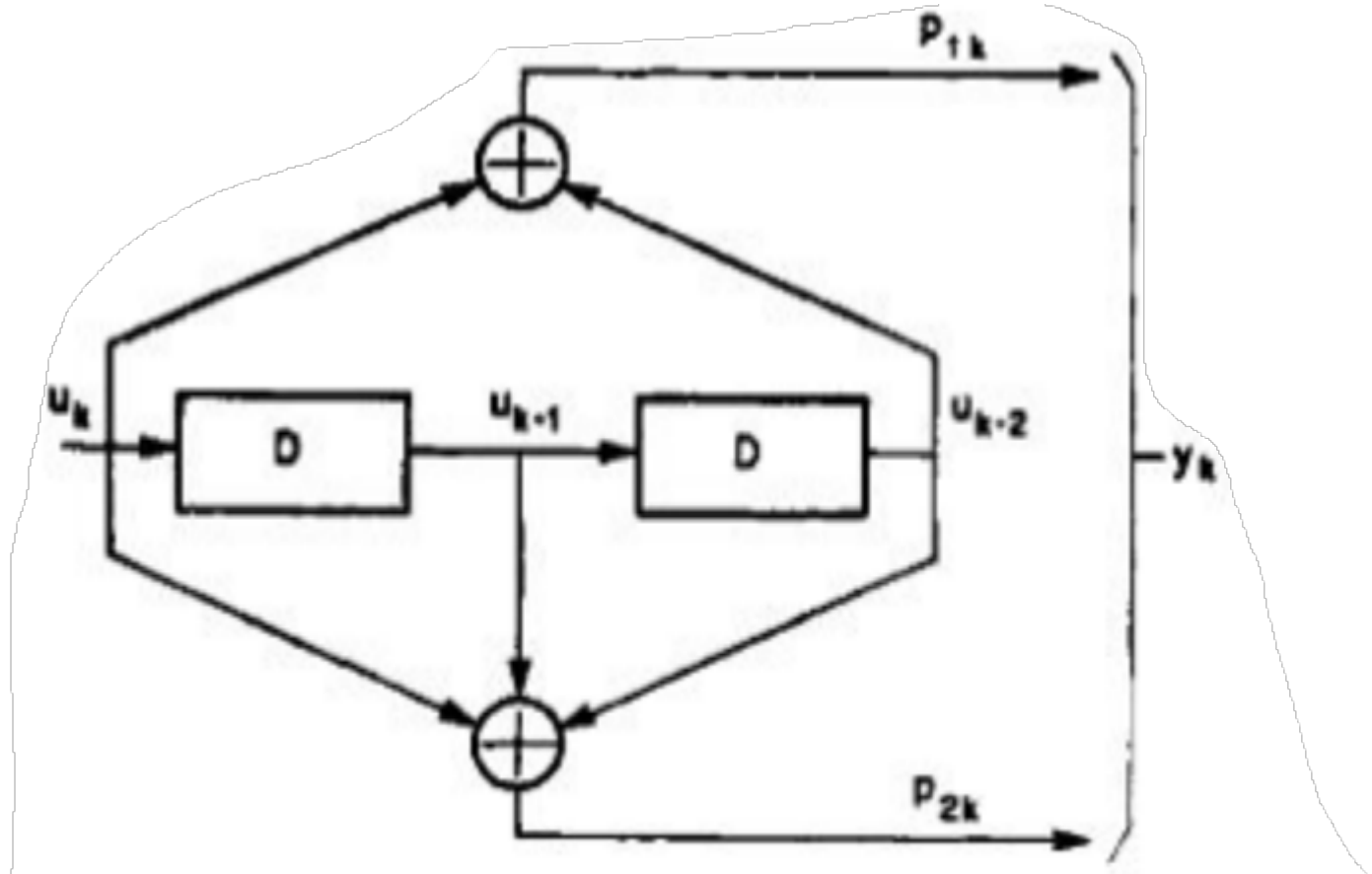
```
entity project_reti_logiche is
      port (
        i_clk     : in  std_logic;
        i_rst     : in  std_logic;
        i_start   : in  std_logic;
        i_data    : in  std_logic_vector (7 downto 0);
        o_address : out std_logic_vector (15 downto 0);
        o_done    : out std_logic;
        o_en      : out std_logic;
        o_we      : out std_logic;
        o_data    : out std_logic_vector (7 downto 0)
      );
end project_reti_logiche;
```

In particolare:

- il nome del modulo deve essere project_reti_logiche;
- `i_clk` è il segnale di CLOCK in ingresso generato dal TestBench;
- `i_rst` è il segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START;
- `i_start` è il segnale di START generato dal Test Bench;
- `i_data` è il segnale (vettore) che arriva dalla memoria in seguito a una richiesta di lettura;
- `o_address` è il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
- `o_done` è il segnale di uscita che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria;
- `o_en` è il segnale di ENABLE da mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- `o_we` è il segnale di WRITE ENABLE da mandare alla memoria (`=1`) per poter scriverci. Per leggere da memoria esso deve essere `0`;
- `o_data` è il segnale (vettore) di uscita dal componente verso la memoria.

1.3 Descrizione della memoria e dell'interazione con il componente

Il modulo riceve in ingresso una sequenza continua di W parole, ognuna di 8 bit, e restituisce in uscita una sequenza continua di Z parole, ognuna da 8 bit. Ognuna delle parole di ingresso viene serializzata; in questo modo viene generato un flusso continuo U da 1 bit. Su questo flusso viene applicato il codice convoluzionale $\frac{1}{2}$ (ogni bit viene codificato con 2 bit); questa operazione genera in uscita un flusso continuo Y. Il flusso Y è ottenuto come concatenamento alternato dei due bit di uscita. Utilizzando la notazione riportata in figura, il bit uk genera i bit p1k e p2k che sono poi concatenati per generare un flusso continuo yk (flusso da 1 bit). La sequenza d'uscita Z è la parallelizzazione, su 8 bit, del flusso continuo yk

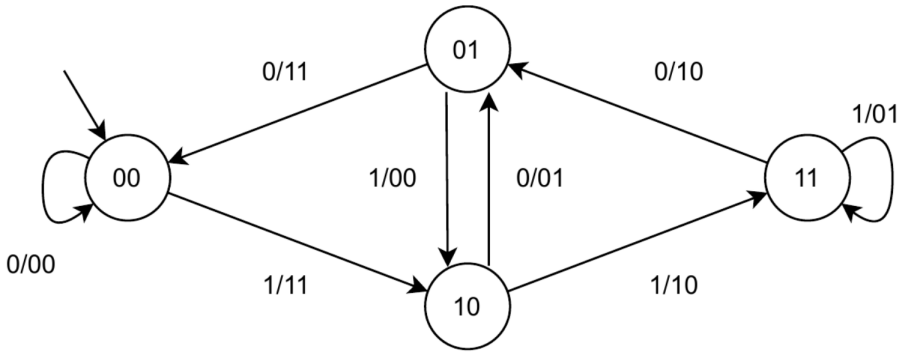


Codificatore convoluzionale con tasso di trasmissio

Indirizzo RAM	Contenuto	Azione
0	byte - lunghezza sequenza di ingresso	lettura
1	byte - prima parola da codificare	lettura
2	byte - seconda parola da codificare	lettura
⋮	indirizzi potenzialmente non utili	⋮
1000	byte - prima parola in uscita	scrittura
1001	byte - seconda parola in uscita	scrittura
1002	byte - terza parola in uscita	scrittura
1003	byte - quarta parola in uscita	scrittura
⋮	indirizzi potenzialmente non utili	⋮

Tabella 1: Rappresentazione della RAM

La lunghezza del flusso U è 8*W, mentre la lunghezza del flusso Y è 8*W*2 (Z=2*W). Il convolutore è una macchina sequenziale sincrona con un clock globale e un segnale di reset con il seguente diagramma degli stati che ha nel suo 00 lo stato iniziale, con uscite in ordine P1K, P2K (ogni transizione è annotata come Uk/p1k, p2k).



1.4 Esempio di funzionamento

(Sequenza lunghezza 6)

W: 10100011 00101111 00000100 01000000 01000011 00001101

Z: 11010001 11001110 10111101 00100101 10110000 00110111
00110111 00000000 00110111 00001110 10110000 11101000

W = Input | Z = Output

addr.	data
0	6
1	163
2	47
3	4
4	64
5	67
6	13
[...]	
1000	209
1001	206
1002	189
1003	37
1004	176
1005	55
1006	55
1007	0
1008	55
1009	14
1010	176
1011	232

2 Architettura del componente

Ho scelto di descrivere un modulo hardware tramite architettura *behavioral* (comportamentale) in linguaggio VHDL, esso opera su una macchina a stati finiti che realizza l'argoritmo esposto più avanti.

2.1 Descrizione ad alto livello

L'algoritmo definito allo svolgimento dell'operazione può essere schematizzato secondo i seguenti passaggi chiave:

1) Lettura del numero di parole `to_be_processed`;
2) Ciclo per leggere i bit della parola in ingresso ed elaborarli nella FSM;
3) Scrittura della prima parola in uscita;
4) Scrittura della seconda parola in uscita;
5) Aumento del contatore `words_processed`;
6) Ripeto dal punto `2)` fino a quando `words_processed=to_be_processed` e poi procedo al DONE.

2.2 Macchina a stati finiti

L'FSM schematizzata è composta da 10 stati, suddivisibili in 3 gruppi principali descritti più avanti.

2.2.1 Stati ausiliari

Gruppo di stati che realizza: inizio e fine del processo, richiesta di lettura e attesa della memoria.

i. `START` **- wait start**: stato di attesa del segnale di `i_start`.
In qualsiasi momento dell'elaborazione, se il segnale `i_rst` è rilevato alto[1], anche non in corrispondenza di `i_clk`, la macchina viene riportata in questo stato, tornando in attesa di un nuovo segnale di inizio elaborazione.

   Al verificarsi della condizione `i_start = 1` vengono inizializzati tutti i valori necessari al processo, prima di passare allo stato successivo;

ii. `WAIT_START` **- wait memory**: stato in cui porto `o_en` a 1 ed `o_address` all'indirizzo della RAM che deve essere letto;

iii. `WAIT_READ_NUMBER_OF_WORD` **- wait memory**: stato di attesa in cui permetto a `to_be_processed` di valere l'intero assegnatogli;

iv. `PREPARE_BIT_READ` **- read request**: stato in cui porto `o_we` a 0 e `o_address` all'indirizzo della RAM che deve essere letto;

v. `WAIT_PREPARE_BIT_READ` **- wait memory**: stato di attesa in cui permetto a `PREPARE_BIT_READ` di aggiornare i valori;

[1] Si è supposto che il segnale `i_start` vada a 0 per il periodo in cui `i_rst` è 1 in caso di reset.

vi. `WAIT_READ_BIT` - **wait memory**: stato di attesa in cui permetto a `READ_BIT` di aggiornare i valori;

vii. `WAIT_WRITE_WORD_ONE` - **wait memory**: stato di attesa in cui permetto a `PREPARE_BIT_READ` di aggiornare i valori;

viii. `WAIT_WRITE_WORD_TWO` - **wait memory**: stato di attesa in cui permetto a `PREPARE_BIT_READ` di aggiornare i valori;

ix. `DONE` - **done**: stato in cui `o_done` viene posto a `'1'` per segnalare la fine dell'elaborazione. A questo punto, si attende un valore di `i_start` basso per tornare in `START` e poter ricevere nuove parole;

2.2.2 Lettura della memoria

Gruppo di stati che si occupa della lettura delle informazioni e della loro elaborazione in `word1` e `word2`.

x. `READ_NUMBER_OF_WORD` - **read** `to_be_processed`: stato in cui viene popolato il valore `to_be_processed` con il numero di parole da processare;

xi. `READ_BIT` - **read bit**: stato in cui percorrendo la FSM leggo un bit, lo processo ottenendone due e proseguo di posizione. Questo stato si occupa anche del controllo `to_be_processed=words_processed` che porta allo stato di DONE;

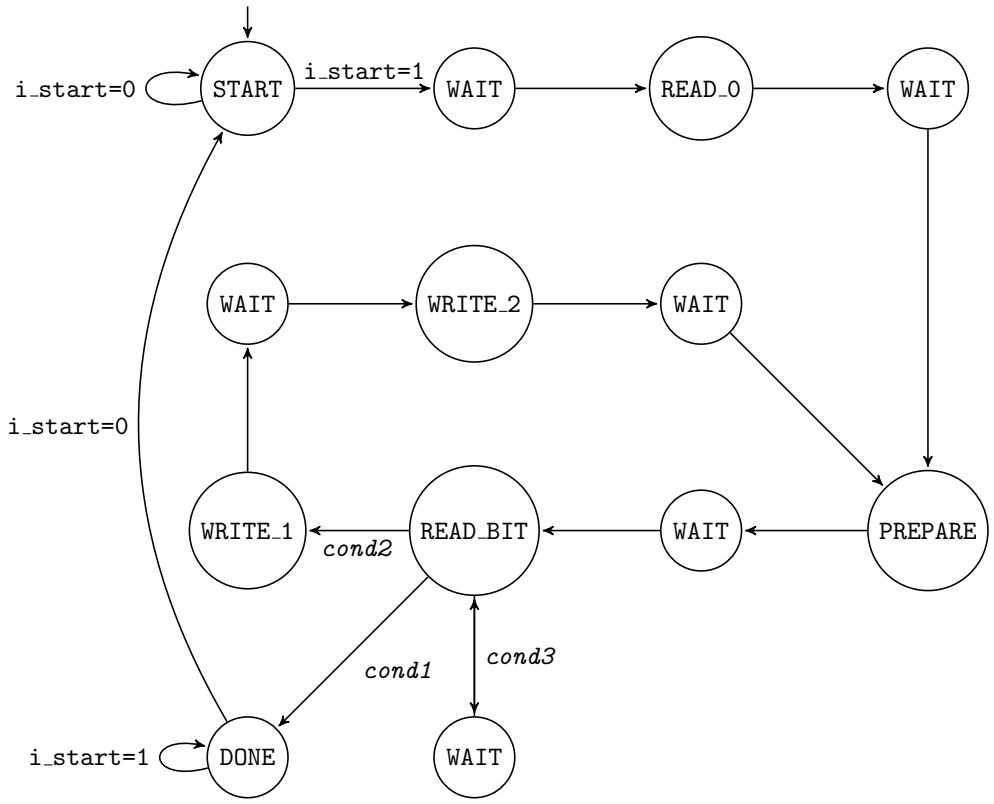
2.2.3 Scrittura delle parole nella memoria

Gruppo di stati che si occupa della scrittura nella memoria delle parole ottenute.

xii. `WRITE_WORD_ONE` - **write data**: stato in cui metto il valore dell'`std_logic_vector` word1 nella memoria;

xiii. `WRITE_WORD_TWO` - **write data**: stato in cui metto il valore dell'`std_logic_vector` word2 nella memoria;

2.2.4 Diagramma della macchina a stati finiti



In figura sono state utilizzate le seguenti abbreviazioni:

cond1	`to_be_processed = words_processed`
cond2	`!cond1 & count = 8`
cond3	`!cond1 & count != 8`

Per ogni stato dell'FSM è presente un arco uscente implicito diretto verso lo stato di reset, che permette di interrompere in qualsiasi momento l'operazione corrente, tramite un segnale `i_rst=1`.

2.3 Scelte progettuali e ottimizzazioni

Ho scelto di progettare un componente sensibile al clock su *rising_edge*.

Nell'implementazione dell'algoritmo vorrei far notare la scelta di mantenere la FSM della specifica implementata tramite if/elsif con dei signal boolean che simulano il passaggio da uno stato ad un altro.

Si noti che nella mia implementazione ci vogliono due cicli di clock per la lettura di un singolo bit per parola, portando la lettura ed elaborazione di una parola a minimo 16 cicli di clock. Nella processione della parola ricevuta mi salvo per ogni bit i due bit, ricevuti dalla FSM, nelle posizioni corrispondenti di due `std_logic_vector`.

Infine si noti che la lettura dei bit avviene 8 volte per leggere una parola intera prima di passare allo stato di `WRITE_1`.

Di seguito è presente il blocco if/elsif utilizzato per l'elaborazione dei bit attraverso la FSM data:

```
if fsm_00 then
    if i_data(7-count) ='0' then
        if (counter < 8) then
            word1(7-counter) <= '0';
            word1(6-counter) <= '0';
        else
            word2(15-counter) <= '0';
            word2(14-counter) <= '0';
        end if;
    elsif i_data(7-count) = '1' then
        if (counter < 8) then
            word1(7-counter) <= '1';
            word1(6-counter) <= '1';
        else
            word2(15-counter) <= '1';
            word2(14-counter) <= '1';
        end if;
        fsm_10 <= true;
        fsm_00 <= false;
    end if;
elsif fsm_01 then
    if i_data(7-count) ='0' then
        if (counter < 8) then
            word1(7-counter) <= '1';
            word1(6-counter) <= '1';
        else
            word2(15-counter) <= '1';
            word2(14-counter) <= '1';
        end if;
        fsm_00 <= true;
        fsm_01 <= false;
    elsif i_data(7-count) = '1' then
        if (counter < 8) then
            word1(7-counter) <= '0';
            word1(6-counter) <= '0';
        else
            word2(15-counter) <= '0';
            word2(14-counter) <= '0';
        end if;
        fsm_10 <= true;
        fsm_01 <= false;
    end if;
elsif fsm_10 then
    if i_data(7-count) ='0' then
        if (counter < 8) then
            word1(7-counter) <= '0';
            word1(6-counter) <= '1';
        else
            word2(15-counter) <= '0';
            word2(14-counter) <= '1';
        end if;
        fsm_01 <= true;
        fsm_10 <= false;
    elsif i_data(7-count) = '1' then
        if (counter < 8) then
            word1(7-counter) <= '1';
            word1(6-counter) <= '0';
        else
            word2(15-counter) <= '1';
            word2(14-counter) <= '0';
        end if;
        fsm_11 <= true;
        fsm_10 <= false;
    end if;
elsif fsm_11 then
    if i_data(7-count) ='0' then
        if (counter < 8) then
            word1(7-counter) <= '1';
            word1(6-counter) <= '0';
        else
            word2(15-counter) <= '1';
            word2(14-counter) <= '0';
        end if;
        fsm_01 <= true;
        fsm_11 <= false;
    elsif i_data(7-count) = '1' then
        if (counter < 8) then
            word1(7-counter) <= '0';
            word1(6-counter) <= '1';
        else
            word2(15-counter) <= '0';
            word2(14-counter) <= '1';
        end if;
    end if;
end if;
```

3 Risultati sperimentali

3.1 Casi di test

Il corretto funzionamento del componente sviluppato è stato verificato tramite numerosi *TestBench*.

In particolare, ho scelto di concentrare l'attenzione su diversi casi critici possibili durante l'esecuzione e sul corretto calcolo di tutti i valori utilizzati. Di seguito una breve lista di condizioni e test più significativi:

- Corretta attenzione nei boolean della fsm `(fsm_00 fsm_01 fsm_10 fsm_11)`;
- Condizione particolare: `to_be_processed = 0`;
- Casi di test con numero di parole massivo;
- Caso di reset dell'elaborazione;
- Corretto rapporto dei segnali `i_rst`, `i_start` e `o_done` durante l'esecuzione.

È stato particolarmente utile utilizzare l'analisi grafica dei segnali di input/output del modulo.

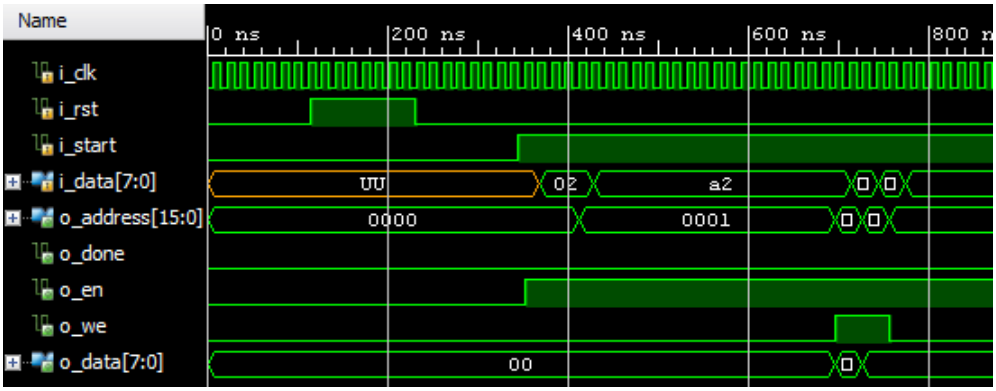


Figura 1: Analisi dei segnali di input e output nell'elaborazione di più parole consecutive.

Si sono utilizzati diversi *TestBench* con caratteristiche differenti e dimensioni variabili fino alle `5000` parole, redatti manualmente o auto-generati tramite uno script python.

3.2 Risultati sperimentali

Il report di sintesi ha evidenziato l'utilizzo nell'area del modulo sintetizzato dei seguenti componenti:

Tabella 2: Risultati della tabella[2] *"utilization"* generata dalla simulazione di *post-synthesis*.

Risorsa	Stima	Utilizzo %
`LUT`	397	0.51
`FF`	163	0.10
`IO`	38	15.20
`BUFG`	1	3.13

3.3 Risultati di simulazione

Per tutti i casi di test e *TestBench* utilizzati, compreso quello delle 5000 parole, sono state svolte con successo le simulazioni richieste dalle specifiche di progetto.

Inoltre la condizione del periodo di clock di "almeno 100 ns" è stata verificata con il seguente constraint:

```
create_clock -period 100 -name clock [get_ports i_clk]
```

4 Conclusioni

A seguito di questo processo posso affermare di aver ampliato di sicuro le mie conoscenze sulla progettazione di un componente con le caratteristiche simili a quello proposto e ritengo che l'architettura proposta rispecchi a pieno le specifiche.

[2] Percentuali riferite alla scheda `FPGA` utilizzata: `xc7z030fbv676-2`.