PROVA FINALE (PROGETTO DI RETI LOGICHE)

Prof. Fabio Salice (Anno 2021/2022)

Progetto svolto in gruppo dai seguenti studenti:

- Matteo Fiorentino (Codice Persona 10686260 – Matricola 937077)
- Luca Longinotti (Codice Persona 10707342 – Matricola 937218)

INDICE

1. INTRODUZIONE

1.1. Scopo del progetto e specifiche generali

Lo scopo del progetto è quello di implementare un modulo HW, descritto in VHDL, che, ricevuta in ingresso una sequenza continua di W parole, ognuna di 8 bit, restituisce in uscita una sequenza continua di Z parole, ognuna da 8 bit.
In primis viene generato un flusso continuo U da 1 bit, di lunghezza 8*W, in seguito alla serializzazione di ognuna delle parole in ingresso.
Successivamente, su questo flusso, viene applicato il codice convoluzionale ½ (ogni bit viene codificato con 2 bit); tale operazione genera un flusso continuo Y, di lunghezza 8*W*2, in uscita.
La sequenza d'uscita finale Z è la parallelizzazione, su 8 bit, del flusso continuo Y.

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 ad 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 dover mandare alla memoria per poter comunicare (sia in lettura che in scrittura);
- o_we è il segnale di WRITE ENABLE da dover 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.

1.1. Dati e descrizione della memoria

I dati, ciascuno di dimensione 8 bit, sono memorizzati in una memoria con indirizzamento al byte:

- L'indirizzo 0 è usato per memorizzare la quantità di parole W da codificare;
- L'indirizzo 1 è usato per memorizzare il primo byte della sequenza W;
- A partire dall'indirizzo 1000 è memorizzato lo stream di uscita Z;
- La dimensione massima della sequenza di ingresso è 255 byte.

2. DESIGN

Quando il segnale i_start in ingresso viene portato a 1, il modulo sviluppato inizia l'elaborazione spostandosi dallo stato START al primo stato di computazione (READ_START).
Una volta terminata la computazione, dopo aver scritto il risultato in memoria, il segnale o_done viene alzato a 1; tale segnale rimane alto fino a quando i_start non è stato riportato a 0.
Solo a questo punto il modulo torna nello stato IDLE in attesa che i_start torni alto.
Il modulo, inoltre, è progettato in modo che venga effettuato sempre un reset prima della prima codifica; per questo scopo è presente il segnale i_rst; tuttavia, una seconda elaborazione non dovrà attendere il reset del modulo ma solo la terminazione dell'elaborazione.

2.1. Stati della macchina

La macchina scelta è composta da 13 stati, ciascuno di essi presentato qui in breve.

2.1.1. START

Stato iniziale del nostro modulo: ogni ciclo di clock controlla se il segnale *i_start* è stato portato a 1, se si il modulo passa allo stato **READ_START**, se invece *i_start* rimane uguale a 0 il modulo rimane nello stato **START**.

2.1.2. READ_START

In questo stato viene settato l'indirizzo a cui andare a leggere in memoria per ottenere i dati necessari alla computazione che sono il numero di parole da leggere e la prima parola; ovviamente non essendo possibile leggere due dati diversi contemporaneamente questo stato verrà attraversato inizialmente 2 volte (una per dato); successivamente nella computazione, lo stato verrà attraversato una volta per ogni parola da leggere. La scelta dell'indirizzo di lettura viene effettuata attraverso i due segnali *got_n_words* e *got_input_buffer*.

2.1.3. WAIT_R_START_1

Questo stato controlla se il modulo ha ottenuto entrambi i dati necessari alla computazione (numero di parole da leggere e la parola corrente): se il controllo risulta positivo il modulo passa direttamente allo stato **PK** mentre in caso contrario passa allo stato **WAIT_R_START_2**.

2.1.4. WAIT_R_START_2

In questo stato vengono salvati, nei rispettivi registri, i dati: numero di parole da leggere e la parola corrente passando poi rispettivamente allo stato **W1** o **W2**.

2.1.5. W1

Stato che controlla se il numero di parole da leggere è uguale a 0: in caso affermativo il segnale *o_done* viene portato a 1 e il modulo passa direttamente allo stato **DONE**, in caso contrario il modulo tornerà allo stato **READ_START** per poter settare l'indirizza a cui leggere la prima parola della sequenza.

2.1.6. W2

Semplice stato di wait dove si attende la fine delle attività di lettura aspettando la risposta della memoria.

2.1.7. PK

Stato in cui vengono "costruiti" i due bit derivanti dal passaggio nel convolutore. Questi due bit verranno usati nella costruzione della sequenza in uscita attraverso concatenazione dei due.

2.1.8. WORD_CONSTRUCTION

Stato dove viene "costruita" la sequenza (16 bit per ogni parola letta), che dovrà essere poi salvata in memoria; se la costruzione è stata completata si passerà allo stato **WRITE_WORD**, mentre in caso contrario si passerà allo stato **W3**.

2.1.9. W3

In questo stato blocchiamo l'aggiornamento dei Flip-Flop e successivamente ritorniamo allo stato **PK**.

2.1.10. WRITE_WORD

Prima parte della fase di scrittura in cui viene settato l'indirizzo in cui salvare i primi 8 bit della sequenza creata in **WORD_CONSTRUCTION.**

2.1.11. WRITE_WORD_2

Seconda parte della fase di scrittura in cui viene scritta la seconda metà della sequenza creata in **WORD_CONSTRUCTION.**

2.1.12. WAIT_W_WORD

Stato in cui si controlla se sono state lette ed elaborate tutte le parole richieste: in caso affermativo si passerà allo stato **DONE**, in caso negativo si tornerà allo stato **READ_START** per leggere la parola successiva.

2.1.13. DONE

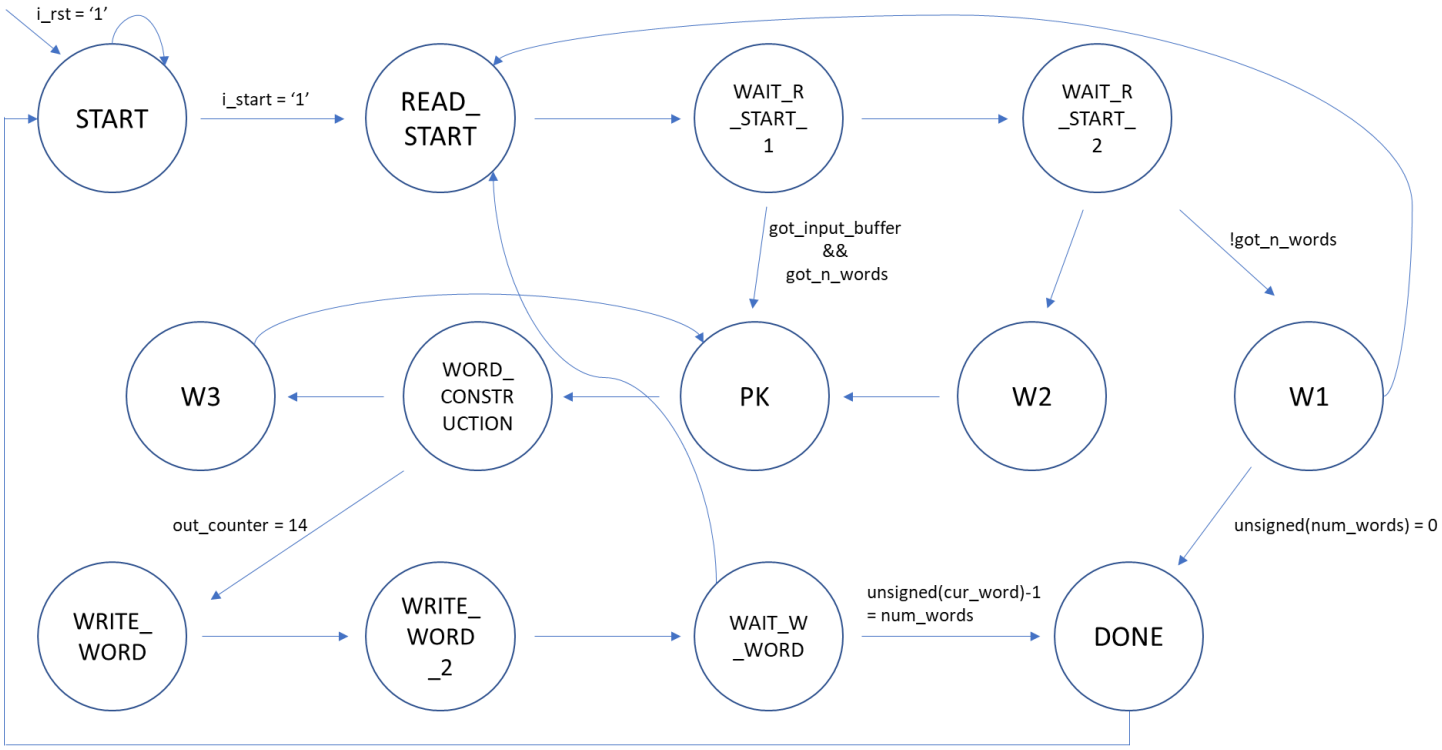
Stato finale del nostro modulo in cui il segnale *o_done* viene settato ad 1 e che riporterà il modulo al suo stato iniziale in attesa di una nuova computazione.

2.2. Scelte progettuali

Il nostro componente è formato da due processi:
Il primo è quello responsabile del funzionamento della FSM: questo viene aggiornato ogni fronte di salita del ciclo di clock e permette al modulo di passare da uno stato all'altro.

Il secondo processo è quello responsabile del comportamento dei FF.

L'algoritmo costruito è stato pensato per convertire interamente, una ad una, le parole da 8 bit in sequenze da 16 bit utilizzando contatori delle celle di memoria dove leggere o scrivere i dati (utilizzando poi operazioni di somma per l'aggiornamento di essi) e flag come, ad esempio, *got_n_words* e *got_input_buffer* per confermare l'effettiva acquisizione dei dati.

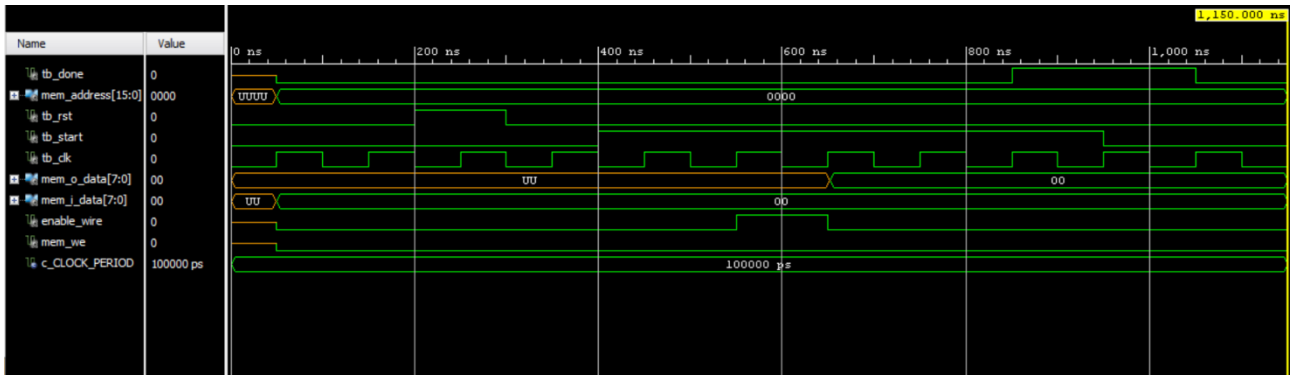


MACCHINA A STATI IMPLEMENTATA

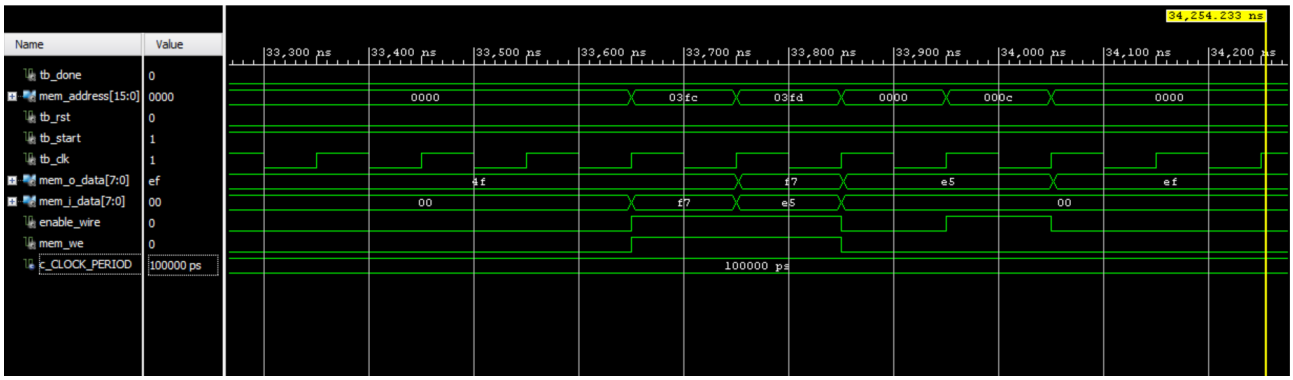
3. RISULTATI DEI TEST

Il nostro componente è stato testato con vari test bench che ne verificavano il corretto funzionamento in situazioni standard e nei corner case. Di seguito verranno presentati i test più significativi effettuati con i relativi andamenti dei segnali.

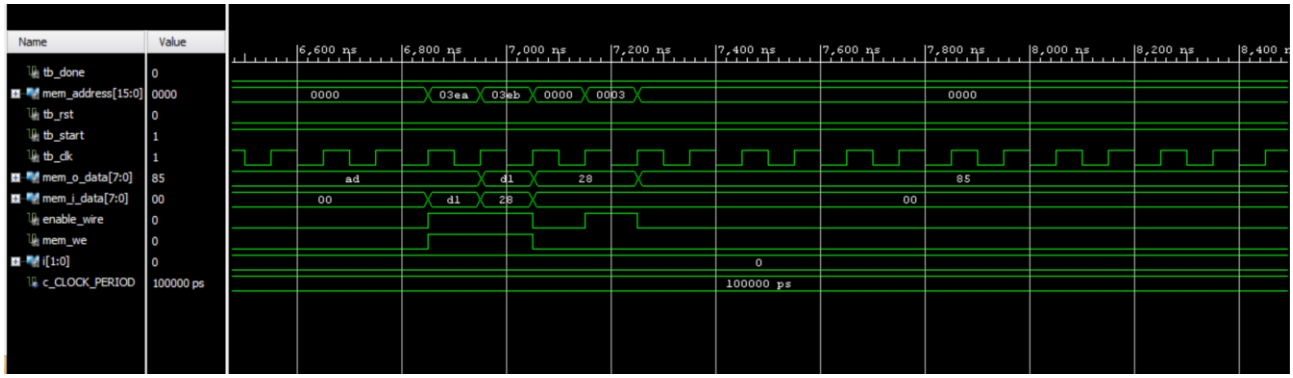
1. **tb_seq_min**: in questo test veniva controllato se, ricevendo in ingresso un numero di parole da processare uguale a zero (ovvero RAM(0) = "00000000"), non venisse scritto niente in memoria.



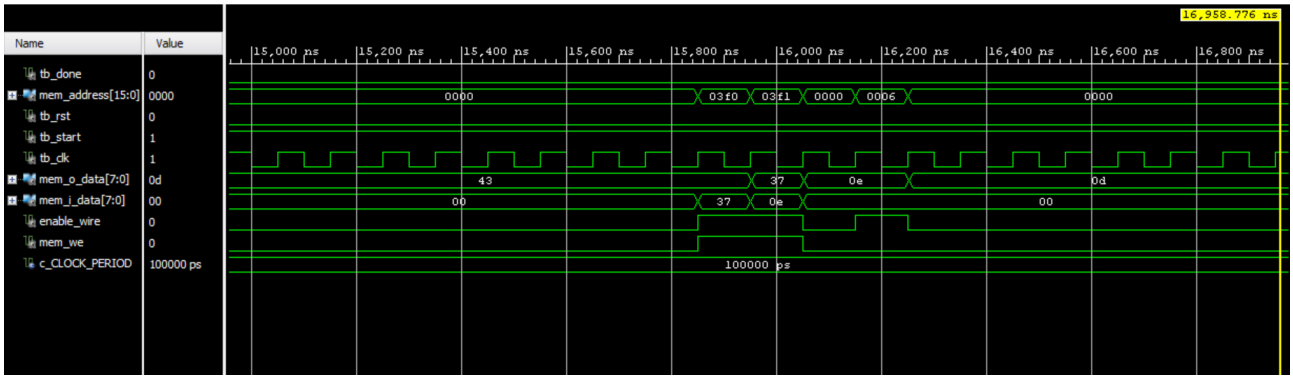
2. **tb_seq_max**: in questo test veniva controllato se, ricevendo in ingresso un numero di parole da processare uguale al massimo possibile (ovvero RAM(0) = "11111111"),il modulo funzionasse correttamente.



3. **tb_re_encode**: in questo test veniva controllato se, a seguito di più flussi consecutivi in ingresso venissero tutti gestiti correttamente.



4. **tb_reset:** in questo test viene controllato se a seguito di un segnale di reset asincrono la computazione non venga compromessa, ma bensì rincominci facendo tornare la FSM allo stato iniziale e resettando tutti i registri come dovrebbe accadere.



4. CONCLUSIONI

4.1. Risultati della sintesi

Il componente sintetizzato supera correttamente tutti i test specificati nelle due simulazioni richieste: *Behavioral* e *Post-Synthesis Functional*.
Il dispositivo utilizza 127 *Look Up Table* e 103 *Flip Flop*, senza alcun *inferred latch*.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	127	0	134600	0.09
LUT as Logic	127	0	134600	0.09
LUT as Memory	0	0	46200	0.00
Slice Registers	103	0	269200	0.04
Register as Flip Flop	103	0	269200	0.04
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

UTILITAZION REPORT

Il componente, inoltre, rimane ampiamente dentro il tempo di clock richiesto; qui di seguito sono riportati i risultati ottenuti con un ciclo di clock di 100ns.

```
Timing Report

Slack (MET) :             96.357ns  (required time - arrival time)
  Source:                 cur_word_reg[4]/C
                            (rising edge-triggered cell FDRE clocked by clock  {rise@0.000ns fall@50.000ns period=100.000ns})
  Destination:            ff_en_reg/D
                            (rising edge-triggered cell FDRE clocked by clock  {rise@0.000ns fall@50.000ns period=100.000ns})
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            100.000ns  (clock rise@100.000ns - clock rise@0.000ns)
  Data Path Delay:        3.492ns  (logic 1.123ns (32.159%)  route 2.369ns (67.841%))
  Logic Levels:           4  (LUT5=2 LUT6=2)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.100ns = ( 102.100 - 100.000 )
    Source Clock Delay      (SCD):    2.424ns
    Clock Pessimism Removal (CPR):    0.178ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

TIMING REPORT

4.2. Ottimizzazioni

Le ottimizzazioni attuate al codice sono state principalmente volte alla riduzione del numero di stati, ad esempio sono stati condensati insieme gli stati relativi all'acquisizione del numero di parole e delle parole; inoltre, le restanti ottimizzazioni hanno ridotto il numero di registri iniziali, i quali durante lo sviluppo si sono poi rivelati superflui.

.