POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE
E BIOINGEGNERIA (DEIB)

Anno 2021/2022

Prova Finale (Progetto di Reti Logiche)

Davide Tenedini
CP : 10651360
CM : 933216

Prof. Gianluca Palermo

Indice

1 Introduzione

1.1 Obiettivo

L'obiettivo del progetto è quello di progettare, scrivere, sintetizzare e testare un componente in linguaggio VHDL. Tale componente deve seguire le specifiche fornite e verrà ultimamente valutato attraverso diversi test bench.

1.2 Specifica

- Funzionamento

  Il componente riceve in input un flusso di parole di 8 bit. Deve serializzare tali parole in un flusso di bit, i quali vengono fatti passare tramite un convolutore (Figura 5). Per ogni bit fornito in ingresso, il convolutore fornisce 2 bit in uscità, i quali dovranno essere parallelizzati in parole da 8 bit.

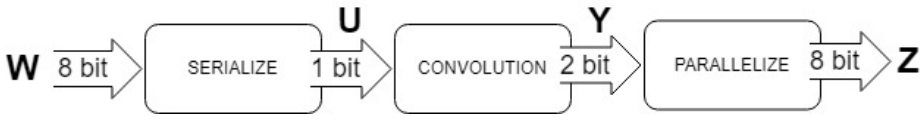

Figura 1: Data Flow.

  In riferimento alla nomenclatura dei flussi usata in Figura 1, se il flusso `W` è composto da `n` parole da 8 bit, allora :

  - il flusso `U` è composto da `n * 8` bit;
  - il flusso `Y` è composto da `n * 8 * 2` bit;
  - il flusso `Z` è composto da `n * 2` parole da 8 bit ciascuna.

- Modalità d'Uso

  Il componente riceve un segnale di reset prima del primo flusso. Invece prima di ogni flusso viene alzato il segnale di start, il quale rimane alto fino a quando non viene alzato il segnale di done. Quest'ultimo deve rimanere alto fino a che il segnale di start non viene abbassato. A questo punto il componente deve aspettare nuovamente un segnale di start.

- Tempo

  Il componente deve funzionare con un periodo di clock di massimo `100ns`.

1.3 Interfaccia 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;
```

I segnali sono utilizzati nel seguente modo:

- `i_clk` è il segnale di clock
- `i_rst` è il segnale di reset
- `i_start` è il segnale di start
- `i_data` è il segnale che porta i dati richiesti alla memoria il ciclo di clock precedente
- `o_address` è il segnale che indica l'indirizzo di memoria che si vuole accedere
- `o_done` è il segnale di done
- `o_en` è il segnale che dice alla memoria che si vuole fare un operazione in questo ciclo di clock
- `o_we` è il segnale che dice alla memoria che tipo di operazione si vuole fare (`1` per scrittura, `0` per lettura)
- `o_data` è il segnale che porta i dati che si vogliono scrivere in memoria

1.4 Descrizione della Memoria

La memoria è indirizzata al Byte e è composta nel seguente modo:

- Cella 0 => Numero di parole nel flusso `W`
- Celle 1-255 => Parole del flusso `W` (ingresso)
- Celle 1000-1509 => Parole del flusso `Z` (uscita)

1.5 Esempio

Ipotizzando di avere un test bench ad un singolo flusso di lunghezza 2.

La memoria è cosi composta :

Addr	0	1	2
Valore	2	41	0

La 2 parole vengono serializzate nel seguente flusso :

$$U = \underbrace{00101001}_{41}\underbrace{00000000}_{0}$$

Il flusso `U` viene fatto passare attraverso il convolutore a Figura 5 ottenendo :

$$Y = \underbrace{00001101}_{13}\underbrace{00011111}_{31}\underbrace{0111000}_{112}\underbrace{00000000}_{0}$$

Il flusso `Y` viene poi scritto in memoria a partire dall'indirizzo 1000 :

Addr	1000	1001	1002	1003
Valore	13	31	112	0

2 Architettura

Il progetto è diviso in 4 moduli, ognuno dei quali ha uno scopo preciso e contiene gli altri moduli che esso utilizza. Il codice è stato racchiuso il più possibile in moduli che eseguono operazioni simili, per facilitare la riusabilità e la leggibilità del codice.

2.1 FSA

L'`FSA` è il modulo principale, la cui interfaccia interagisce con i testbench e la memoria. Questo modulo si occupa di gestire la macchina a stati e farla interagire con il modulo `datapth` in esso

contenuto.

È composto da 3 processi, i quali si occupano di far avanzare la macchina a stati sul fronte di salita di ogni ciclo di clock, decidere il prossimo stato e gestire i vari segnali da/per sia il componente `datapath` che l'esterno.

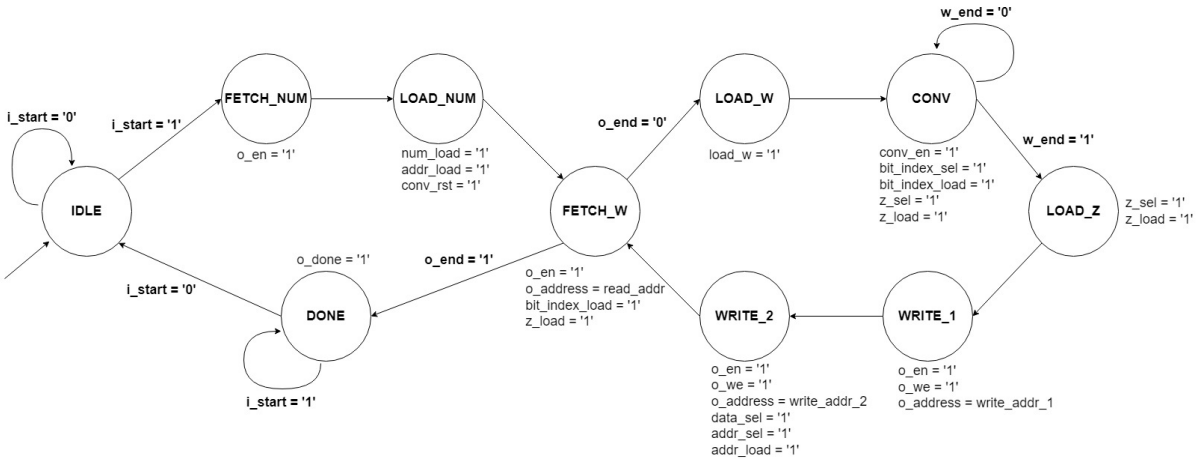

Figura 2: Diagramma degli Stati dell'FSA.

Nella Figura 2 vengono riportate tutte le transizioni del'FSA. Inoltre vengono indicati i valori che assumono i segnali in ogni stato. Se un segnale non compare in un certo stato si può assumere che il suo valore sia quello di default.

2.1.1 Segnali e Valori di Default

```
num_load       <= '0';
addr_load      <= '0';
bit_index_load <= '0';
w_load         <= '0';
z_load         <= '0';
addr_sel       <= '0';
bit_index_sel  <= '0';
z_sel          <= '0';
data_sel       <= '0';
conv_en        <= '0';
conv_rst       <= '0';
o_address      <= "0000000000000000";
```

```
o_en            <= '0';
o_we            <= '0';
o_done          <= '0';
```

2.1.2 Stati

- IDLE

  Questo è lo stato a riposo e di reset della macchina. In questo stato tutti i segnali sono al loro valore di default. La macchina, seguendo il protocollo delle specifiche, attende di vedere il segnale `i_start = '1'` prima di andare nello stato `FETCH_NUM`.

- FETCH_NUM

  In questo stato la macchina chiede alla memoria di leggere nella cella `0x00` prima di andare nello stato `LOAD_NUM`.

- LOAD_NUM

  In questo stato la macchina aspetta il risultato della richiesta alla memoria e ordina al registro `NUM_REG` di salvarselo, prima di andare nello stato `FETCH_W`.

- FETCH_W

  In questo stato la macchina chiede alla memoria di leggere nella cella `read_addr`. Se la macchina ha finito di elaborare tutte le parole (`o_end = '1'`) va nello stato `DONE`, altrimenti continua l'elaborazione nello stato `LOAD_W`.

- LOAD_W

  In questo stato la macchina aspetta il risultato della richiesta alla memoria e ordina al registro `W_REG` di salvarselo, prima di andare nello stato `CONV`.

- CONV

  In questo stato la macchina manda, uno per ciclo di clock, i bit della parola in elaborazione. Il `bit_index`-esimo bit viene elaborato dal convolutore e i 2 bit risultanti vengono salvati, concatenandoli ai precedenti risultati, nel registro `Z_REG`. La macchina rimane in questo stato fino a quando non ha processato tutti gli 8 bit della parola (`w_end = '1'`), per poi andare nello stato `LOAD_Z`.

- LOAD_Z

  In questo stato la macchina ordina al registro `Z_REG` di salvarsi l'ultima elaborazione del convolutore, prima di andare nello stato `WRITE_1`.

- WRITE_1

  In questo stato la macchina chiede alla memoria di scrivere nella cella `write_addr_1 = (read_addr << 1) + 998`, gli 8 bit più significativi del registro `Z_REG`, prima di andare nello stato `WRITE_2`.

  Notare che `read_addr << 1` equivale a `read_addr * 2`.

- WRITE_2

  In questo stato la macchina chiede alla memoria di scrivere nella cella `write_addr_2 = write_addr_1 + 1`, gli 8 bit meno significativi del registro `Z_REG`, prima di andare nello stato `FETCH_W`.

- DONE

  In questo stato la macchina segnala con `o_done = '1'` di aver terminato l'elaborazione, e aspetta che il segnale `i_start = '0'` prima di andare nello stato `IDLE`.

2.2 Datapath

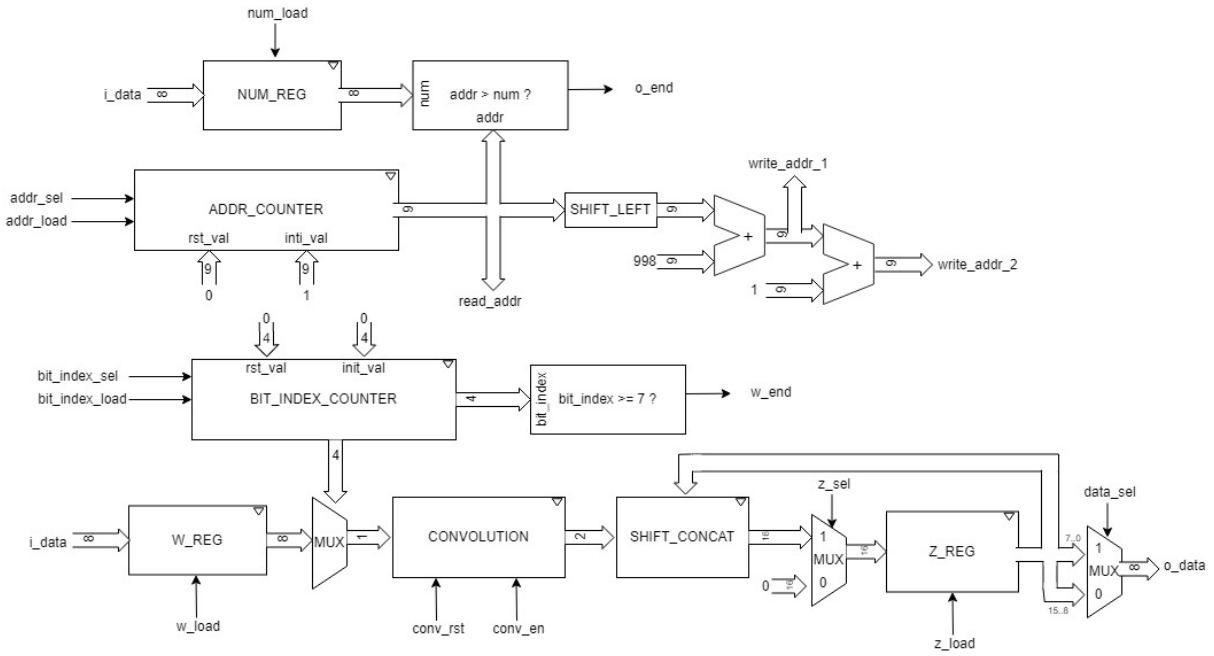

Figura 3: Versione digitalizzata del design progettato inizialmente. Il codice scritto si basa fortemente su questo design.

Il modulo `datapath` è una collezione di diversi componenti e processi, in particolare contiene 2 istanze del modulo `counter` (una per `ADDR_COUNTER` e una per `BIT_INDEX_COUNTER`), un'istanza del modulo `convolution` e 3 processi che descrivono il funzionamento dei registri `NUM_REG`, `W_REG` e `Z_REG`.
Inoltre contiene diversi snippet di codice che implementano i 3 multiplexer e i diversi moduli aritmetici:

- `SHIFT_LEFT` - esegue lo shift a sinistra di 1 posizione;
- `SHIFT_CONCAT` - prende i bit `13 downto 0` e aggiunge a destra i 2 bit in uscita dal convolutore;
- le varie ALU che sommano un segnale ad una costante.

Per finire, si occupa del gestire i segnali di fine parola `w_end` e fine elaborazione `o_end`.

Notare che i moduli contrassegnati dal sibolo di un triangolino ($\bigtriangledown$) ricevono in ingresso i segnali `i_clk` e `i_rst`, quindi funzionano solo durante il fronte di salita del clock e vengono resettati correttamente (anche in maniera asincrona).

2.2.1 Counter

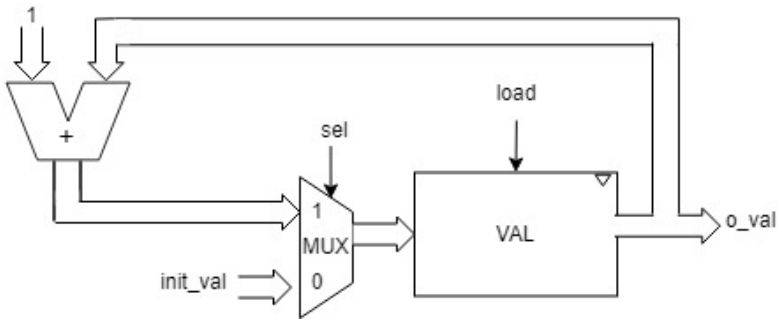

Figura 4: Versione digitalizzata del design progettato inizialmente. Il codice scritto si basa fortemente su questo design.

Il `counter` è un modulo che implementa un contatore che prende in input un valore di reset `rst_val`, e il valore iniziale del contatore `init_val`. Sul fronte di salita del clock, se `sel = '1'` e `load = '1'`, il contatore aumenta di 1 unità. Al suo interno ha un processo sincronizzato per gestire il registro e, allo stesso modo del datapath, implementa il multiplexer e la ALU.

Notare che questo modulo ha un'ulteriore uscita, `o_val_int`, che è il cast di `o_val` al tipo `INTEGER`. Questo segnale viene usato dal datapath per rendere immediati i confronti quando calcola i flag `o_end` e `w_end`.

2.2.2 Convolution

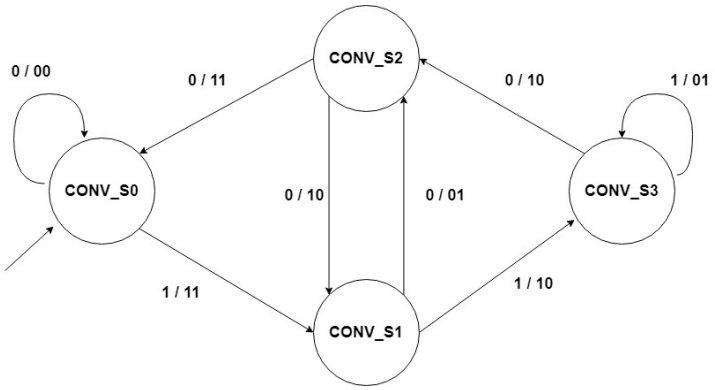

Figura 5: Diagramma degli Stati del Convolutore.

Il modulo `convolution` implementa la macchina a stati fornita nelle specifiche (Figura 5) allo stesso modo del modulo `FSA`. Ha infatti 3 processi: uno sincronizzato per i registri di stato e uscita; uno per calcolare il prossimo stato; e uno per trovare la prossima uscita.

Notare che questo modulo riceve in input, oltre ai soliti segnali `i_clk` e `i_rst`, anche i segnali `conv_en` e `conv_rst`. Questi segnali permettono di gestire opportunamente lo stato interno del convolutore tra diversi flussi e diverse parole.
All'inizio di ogni flusso successivo al primo è necessario mettere `conv_rst = '1'`, per resettare lo stato interno che potrebbe essere un qualsiasi stato residuo dal flusso precedente.
Inoltre, dal momento che tra una parola e l'altra all'interno dello stesso flusso il datapath fa altre operazioni, il segnale `conv_en = '1'` viene dato solo quando i vari segnali sono in condizioni tali da favorire l'elaborazione del bit corretto.

3 Risultati Sperimentali

3.1 Sintesi

Il componente è stato sintetizzato con successo utilizzando 51 LUT e 59 FF, senza inferire alcun latch. Inoltre ha uno slack di `96.770ns`. Non è stato effettuato nessun tipo di ottimizzazione al

design iniziale, ad eccezione dell'uso di segnali a 10 bit per il `ADDR_COUNTER` in quanto lo spazio di indirizzi di memoria non richiede l'utilizzo degli ultimi 6 bit più significativi. Questo ha permesso di risparmiare 6 FF.

Name	Constraints	Status	LUT	FF
synth_1	constrs_1	synth_design Complete!	51	59
impl_1	constrs_1	Not started		

```
required time                         102.272
arrival time                           -5.503
-------------------------------------------------
slack                                  96.770
```

Figura 6: Risultati della Sintesi

3.2 Simulazioni

Tutti i seguenti test bench sono stati simulati sia in modalità *Behavioral Pre-Synthesis* che *Post-Synthesis*.

3.2.1 Lunghezza Minima del Flusso

Questo test controlla che il componente, dopo aver ricevuto in input un flusso vuoto (il numero di parole è 0), non performi nessuna operazione di elaborazione o scrittura.

Addr	0	1	...	1000	1001
Valore	0	0	0	0	0

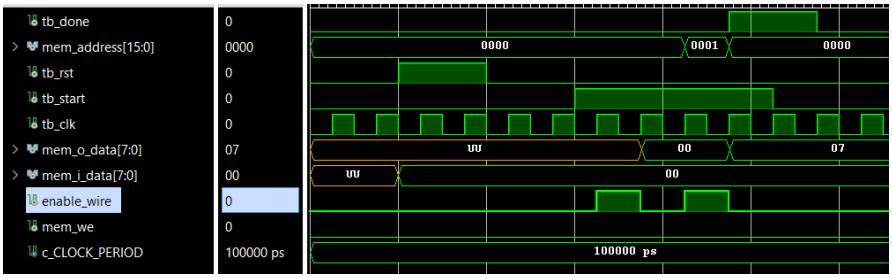

Figura 7: Risultato della Simulazione. Il componente alza immediatamente il segnale di done.

3.2.2 Lunghezza Massima del Flusso

Questo test controlla che il componente, dopo aver ricevuto in input un flusso di massima lunghezza (il numero di parole è 255), performi correttamente l'elaborazione e termini al momento giusto.

Addr	0	1	...	255	...	1000	1001	...	1509
Valore	255	255	...	216	...	229	85	...	172

3.2.3 Reset Asincrono

Questo testo controlla che il componente risponda in maniera corretta ad un segnale di reset asincrono. Il componente in questo caso riprende l'elaborazione da capo e fornise i risultati corretti. Ovviamente è stato provato anche un test in cui il reset è dato in maniera sincrona. Anche in quel caso il componente si comporta come aspettato.

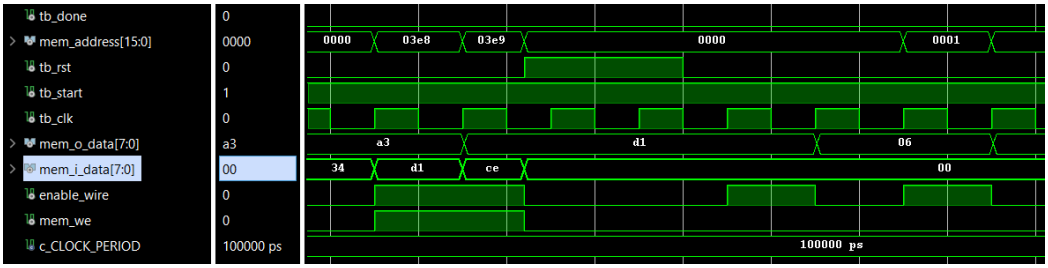

Figura 8: Risultato della Simulazione.

3.2.4 Altri TB

Per coprire ogni altro caso di test è stato utilizzato un generatore casuale di casi di test di lunghezza variabile da 0 a 255 parole per flusso. Il componente è stato sottoposto a tali test in 2 momenti. Il primo batch di 100k flussi casuali, divisi in più test run di diversa lunghezza è stato sottoposto dopo aver ottenuto un componente funzionante. Dopo aver fatto code cleanup e aver preparato il componente in un file unico con commenti vari, pronto per la consegna, è stato sottoposto un secondo batch di 100k flussi casuali.

Questi test confermano che il componente si comporta correttamente in vari aspetti critici :

- Il componente riesce ad eseguire più flussi successivamente quindi resetta in maniera corretta tutti i segnali tra un flusso e l'altro.
- Il componente mantiene correttamente lo stato interno tra una parola e l'altra.
- Il componente si comporta in maniera corretta a fronte di moltissimi casi di test di varia lunghezza.

Ovviamente è impensabile testare ogni combinazione possibile, ma aver completato quasi 200k test casuali dà un alto livello di confidenza nella correttezza del componente.

4 Conclusioni

Il componente scritto viene sintetizzato correttamente in un modulo che rispetta le specifiche di interfaccia, protocollo d'uso e funzionamento fornite.
Dal punto di vista di spazio, il modulo utilizza pochi componenti fisici quali 59 Flip-Flop e 51 Look-Up-Tables, e non ci sono evidenti metodi per ridurre drasticamente l'area coperta da esso.
Dal punto di vista del tempo, il modulo ha uno slack di `96.770ns`. Questo indica che il componente utilizza meno del 4% del periodo di clock imposto per completare la computazione nel *worst case scenario*.

Questi risultati sono molto più che accettabili per quanto riguarda performance e ottimizzazioni. Inoltre il codice è scritto per ottimizzare la leggibilità e per essere il più fedele possibile al design iniziale (Figura 3).