Prova Finale di Reti Logiche

Arturo Benedetti

31 marzo 2022

Professore: William Fornaciari
Codice persona - Matricola: -

Indice

1 Introduzione

La Prova Finale di Reti Logiche chiede di implementare un modulo HW (descritto in VHDL) che si interfacci con una memoria e che segua una particolare specifica.

1.1 Specifica

Data una sequenza continua di W parole, ognuna di 8 bit, il modulo deve restituire una sequenza continua di Z parole, ognuna di 8 bit, data dall'applicazione dell'algoritmo di Viterbi sull'ingresso. La sequenza in uscita Z sarà lunga il doppio di quella in ingresso W, dato che per ogni bit in ingresso ne verranno generati due (vedere 1.3).
Il modulo dovrà quindi:

1. Leggere da una memoria RAM i dati da computare
   (a) Dall'indirizzo 0x0 il numero di parole W in input (max. 255)
   (b) A partire dall'indirizzo 0x1 la sequenza W
2. Serializzare la sequenza in ingresso
3. Applicare il codice convoluzionale fornito (sezione 1.3) al flusso (ogni bit viene codificato con 2 bit)
4. Concatenare i bit ottenuti per ottenere il flusso Y
5. Parallelizzare, su 8 bit, il flusso continuo Y, per ottenere la sequenza d'uscita Z
6. Scrivere il risultato nella memoria RAM 8 bit alla volta, a partire dall'indirizzo 1000

Il progetto deve funzionare con un clock minore di 100ns.

1.2 Memoria

La memoria è di tipo sincrono: ciò significa che, durante un ciclo di lettura, il dato viene fornito nel ciclo di clock **successivo** a quello in cui viene fornito l'indirizzo. Nel caso di scrittura, invece, è necessario fornire indirizzo e dato nello stesso ciclo.

1.3 Convolutore

Il convolutore è una macchina sequenziale sincrona con un clock globale ed un segnale di reset, avente il seguente diagramma degli stati:

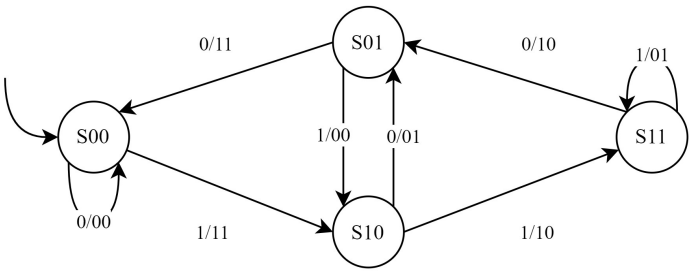

Figura 1: Convolutore - Algoritmo di Viterbi

1.4 Esempio

Di seguito è riportato un esempio di funzionamento della codifica convoluzionale. L'input viene analizzato dal convolutore, il quale genera due bit in uscita per ogni bit in ingresso. L'output è formato dalla concatenazione dei bit appena ottenuti. Ogni byte in ingresso produce quindi due byte in uscita.

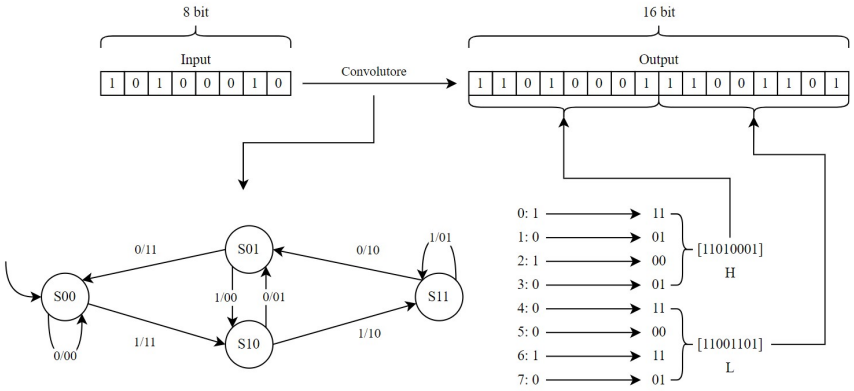

Figura 2: Esempio di codifica convoluzionale

2 Architettura

Lo strumento di sintesi utilizzato è Xilinx Vivado Webpack, con target FPGA Artix-7 xc7a200tfbg484-1.
Per la realizzazione si è partiti dalla progettazione ad alto livello del sistema (2.2) così da definire il comportamento generale del modulo; si è poi passati alla strutturazione del **Datapath** (2.3) e della relativa **Macchina a Stati Finiti** (2.4), necessari per il controllo del comportamento del sistema.

2.1 Ipotesi progettuali

Il progetto è stato sviluppato sotto le seguenti ipotesi:

1. Il numero massimo di parole W in ingresso è 255.
2. Prima della prima codifica verrà sempre dato il RESET al modulo.
3. Una volta portato alto, il segnale START rimarrà in questo stato fino a che il segnale DONE non verrà portato alto.
4. La quantità di parole da codificare sarà sempre memorizzata all'indirizzo 0, e l'uscità sarà sempre memorizzata a partire dall'indirizzo 1000.

2.2 Descrizione ad alto livello

Da un punto di vista di alto livello, il modulo eseguirà i seguenti passi:
Quando un segnale START in ingresso verrà portato ad 1:

1. Inizializzazione registri
2. Lettura della quantità di parole da codificare dall'indirizzo 0 della RAM
3. Controllo di aver codificato il numero richiesto di parole, in tal caso si passa al punto 10
4. Lettura della parola da codificare dalla RAM (8 bit)
5. Codifica mediante l'utilizzo del codificatore convoluzionale (sezione 1.3)
6. Concatenazione del risultato ottenuto (16 bit risultanti)
7. Caricamento del risultato ottenuto in RAM
   (a) Caricamento della parte alta (H) del risultato (bit 15..8)
   (b) Caricamento della parte bassa (L) del risultato (bit 7..0)
8. Aggiornamento indici per lettura/scrittura memoria
9. Torna al punto 3
10. Segnalazione terminazione settando il segnale DONE ad alto.
11. Attesa dell'abbassamento del segnale START
12. Abbassamento del segnale DONE

2.3 Datapath

Nel datapath prevediamo l'utilizzo di 4 registri primari (utili al processo di codifica) e 2 secondari (per il salvataggio degli indici di input ed output):

Primari:

- `REG_NUM`: Registro che conterrà il numero di parole da codificare (8 bit, dato che il valore massimo è 255)
- `REG_INPUT`: Registro che conterrà la parola da codificare (8 bit)
- `REG_OUTPUT`: Registro che conterrà l'output derivante dalla codifica (16 bit). Inizializzato a 0
- `FSM_STATE`: Registro che conterrà lo stato raggiunto dal codificatore convoluzionale alla fine della codifica (2 bit, dato che il convolutore ha 4 stati). Inizializzato a 0 (corrispondente allo stato di RESET del codificatore)

Particolarmente importante risulta essere la bufferizzazione del numero di parole da codificare in `reg_num`: ciò evita che per ogni lettura di tale valore si debba accedere alla memoria RAM, riducendo sensibilmente i tempi di elaborazione e quindi permettendo di lavorare con un periodo di clock minore.

Secondari:

- `REG_ININDEX`: Registro contenente l'indice a cui leggere l'input, utilizzato anche per sapere quante parole sono state codificate (fig. 4)
- `REG_OUTINDEX`: Registro contenente l'indice a cui scrivere l'output (fig. 5). Il registro è inizializzato a 1000, in quanto l'output deve essere scritto a partire da quell'indirizzo.

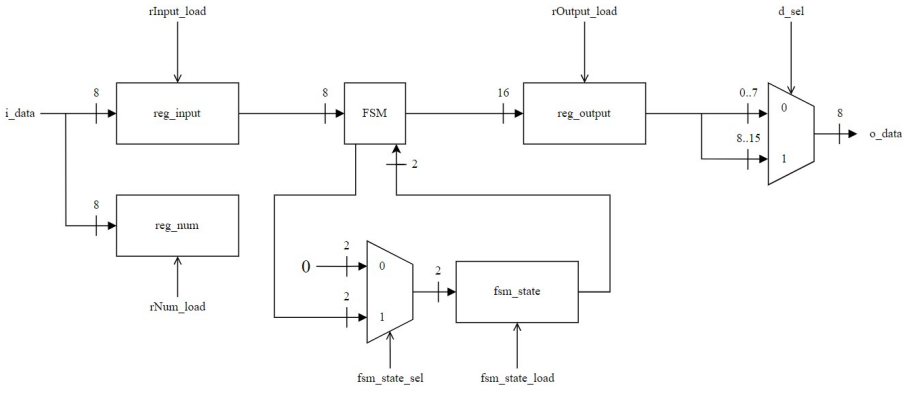

Figura 3: Datapath interno

Ad ogni registro è associato un segnale di load, il quale servirà per sapere se lo stream in ingresso al registro stesso debba essere caricato o meno.
Il blocco FSM modella la presenza del convolutore. Quest'ultimo riceve in ingresso il contenuto di `REG_INPUT` e lo codifica, generando un segnale in output che verrà salvato nel registro `REG_OUTPUT`.
Il convolutore ha anche accesso al registro `FSM_STATE`, in modo tale da poter salvare lo stato raggiunto a fine codifica (di ogni parola) e recuperare lo stato raggiunto precedentemente all'inizio di ogni codifica successiva.
Sono stati utilizzati 4 mux:

- 3 sono utilizzati per gestire l'inizializzazione del registro che precedono
- 1 è utilizzato per selezionare quale sezione dell'output verrà salvata in memoria, come spiegato in seguito

Ad ogni mux è associato il relativo segnale di selezione, controllato dalla FSM.
Vista la dimensione del registro `REG_OUTPUT` (16 bit), la scrittura del risultato in RAM è stata spezzata in due, scegliendo di caricare separatamente prima i bit da 15 a 8 (H), poi i restanti da 7 a 0 (L). Si è deciso di modellare tale meccanismo mediante l'utilizzo di un mux avente in ingresso le due "sezioni" dell'output (High - H e Low - L), il quale verrà controllato dalla FSM.

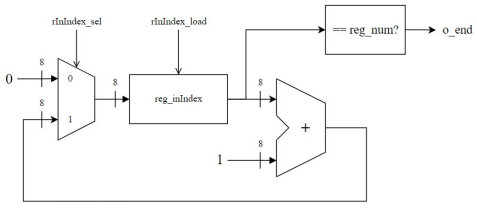

Figura 4: Contatore inputIndex

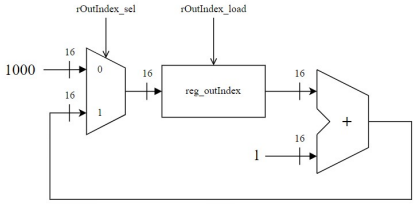

Figura 5: Contatore outputIndex

Si è deciso di utilizzare due contatori separati per gli indici di input ed output per mantenere separati i due sottosistemi (figure 4 e 5), dato che la frequenza delle parole in uscita è doppia rispetto a quella in ingresso. In questo modo si evitano anche una serie di operazioni (un'addizione su 16 bit ed una moltiplicazione per 2) necessarie nel caso si fosse utilizzato un solo contatore per l'indice di input per poter ricavare gli indici di output corrispondenti.

2.4 Macchina a Stati Finiti

La **Macchina a Stati Finiti** (FSM) è il circuito che si occupa della gestione dello stato di computazione e dei segnali interni del modulo.
Di seguito è riportata una descrizione degli stati che la compongono:

- `IDLE`: Stato di Idle, la FSM rimarrà in questo stato fino a quando non sarà ricevuto il segnale di START (`i_start` = 1). Corrisponde allo stato di reset.
- `SET_READ_ADDR`: Stato in cui viene preparata la lettura del numero di byte da codificare dall'indirizzo 0 della memoria RAM. Vengono inoltre inizializzati i registri `inIndex`, `outIndex`, `fsm_state` mediante il settaggio degli appositi segnali, come indicato in figura 6.
- `FETCH_READ_ADDR`: Stato in cui viene effettuata la lettura preparata al punto precedente, il risultato è salvato all'interno del registro `reg_num`.
- `CHECK_END`: Stato in cui si controlla di aver codificato il numero di parole richiesto. Ciò viene effettuato confrontando il contenuto dei registri `reg_num` e `reg_inIndex`: se i due valori corrispondono viene settato ad `1` il segnale `o_end` (figura 4), che altrimenti è settato a `0`. Il controllo viene effettuato subito dopo aver letto il numero di parole da codificare, ciò evita che si prosegua nella codifica nel caso il numero di byte da codificare sia 0.
- `SET_INPUT_INDEX`: Stato che si occupa dell'incremento dell'indice dell'input.
- `SET_INPUT_ADDR`: Stato che prepara la lettura da memoria RAM della parola in input. L'indirizzo di lettura corrisponde a quello salvato nel registro `reg_inIndex`, aggiornato nel precedente stato.
- `READ_INPUT`: Stato in cui viene effettuata la lettura preparata al punto precedente, il risultato è salvato all'interno del registro `reg_input`.
- `COMPUTE`: Stato che si occupa dell'effettiva codifica convoluzionale dell'input. L'input viene analizzato un bit alla volta, da `MSB` ad `LSB`. Per ogni bit in ingresso ne vengono generati due in uscita secondo la codifica mostrata in 1.3, i quali vengono poi concatenati in un unico vettore di 16 bit. Tale vettore è salvato nel registro `reg_output`. Alla fine della codifica viene salvato lo stato raggiunto dal codificatore convoluzionale nel registro `fsm_state`, mediante il settaggio dei relativi segnali di controllo.
- `WRITE_RESULT_H`: Stato che si occupa della scrittura della parte alta (H - High) del risultato (bit 15..8) in memoria RAM. Lo stato si occupa anche dell'incremento del contatore dell'indice dell'output (avendo appena effettuato una scrittura).

- `WRITE_RESULT_L`: Stato che si occupa della scrittura della parte bassa (L - Low) del risultato (bit 7..0) in memoria RAM. Lo stato si occupa anche dell'incremento del contatore dell'indice dell'output (avendo appena effettuato una scrittura).
- `NOTIFY_END`: Stato raggiunto in seguito ad aver completato la codifica del flusso W in ingresso. Viene settato ad alto il segnale `o_done` (DONE), ed atteso l'abbassamento del segnale `i_start`, come da specifica.
- `ENDING`: Stato che si occupa dell'abbassamento del segnale `o_done`, completando il ciclo di esecuzione e riportando la FSM allo stato di `IDLE`.

Di seguito è riportato il diagramma degli stati della FSM

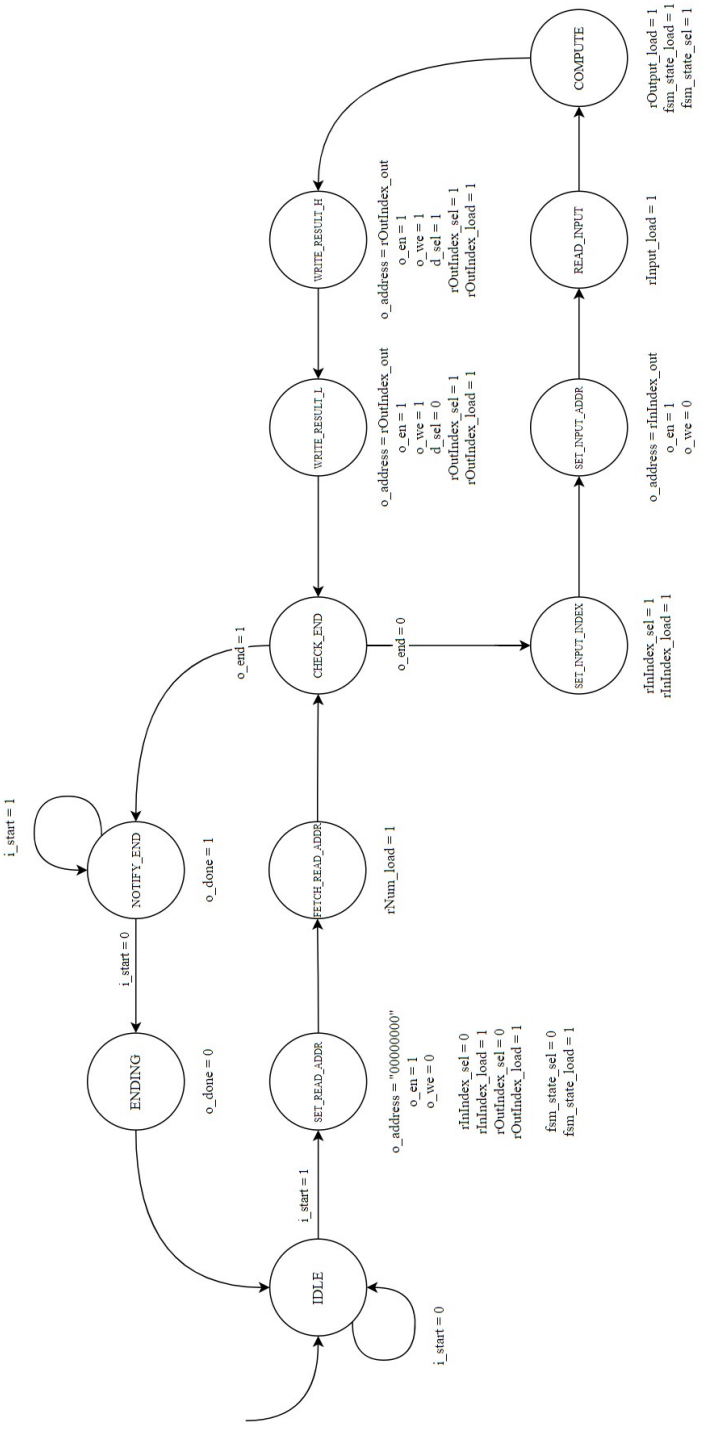

Figura 6: Diagramma degli stati della FSM

3 Testing

La fase di testing è stata effettuata sia utilizzando il testbench di esempio fornito, sia generando un elevato numero di test casuali, sia utilizzando testbench che mirassero a testare eventuali corner cases.
Per il testing di casi "standard" è stato utilizzato un generatore di testcases, grazie al quale sono stati testati oltre 100 000 casi, tutti eseguiti con successo.

3.1 Corner cases

Sono stati individuati i seguenti corner case notevoli, per i quali sono stati preparati testbench specifici:

- Sequenza in ingresso minima, `0`: questo caso controlla che il modulo non proceda erroneamente con la codifica nel caso non ci sia nulla da codificare
- Sequenza in ingresso massima, `255`: questo caso controlla che il modulo sia in grado di codificare correttamente la dimensione massima stabilita di parole W
- Presenza di flussi successivi: questo caso controlla che il modulo sia in grado di effettuare codifiche di sequenze successive
  - Con reset tra una codifica e l'altra
  - Senza reset tra una codifica e l'altra
- Reset durante la codifica: questo caso controlla che il modulo sia in grado di funzionare nel caso sia imposto un reset in un momento qualsiasi (seguito da un'eventuale nuova codifica), ad esempio se si vuole effettuare una codifica senza dover attendere il risultato di quella precedente (che verrà però sovrascritta)

In tutti questi casi il modulo si è comportato correttamente, completando con successo la codifica richiesta sia in simulazione behavioural che in simulazione functional post-synthesis.

3.2 Report di Sintesi

Analizzando i report post sintesi sui componenti utilizzati e sul timing, si evince che il modulo rispetta i vincoli imposti, in particolare si può notare l'assenza di Latch.

- `Utilization`: Il modulo è stato sintetizzato utilizzando 62 registri come Flip Flop (0.02% del totale), e 0 registri come Latch
- `Timing`: Lo slack è di 97.135 ηs (post `opt_design`), ciò significa che la parte combinatoria richiede solamente $\sim$3 ηs. Il modulo è quindi in grado di funzionare correttamente anche a fronte di una riduzione del clock del 97%, funzionando ad una frequenza di quasi 350 MHz.

4 Conclusioni

Il componente realizzato è in grado di interfacciarsi con la memoria e seguire la specifica richiesta, codificando correttamente sequenze di bit utilizzando il convolutore specificato (sotto le ipotesi a 2.1) e salvandone il risultato in memoria. Il modulo è stato sottoposto ad un elevato numero di test, che ne hanno comprovato il corretto funzionamento.