SILVESTRINI GIADA

Prova Finale di Reti Logiche

A.A. 2021-2022

Matricola: 932121

Codice persona: 10659711

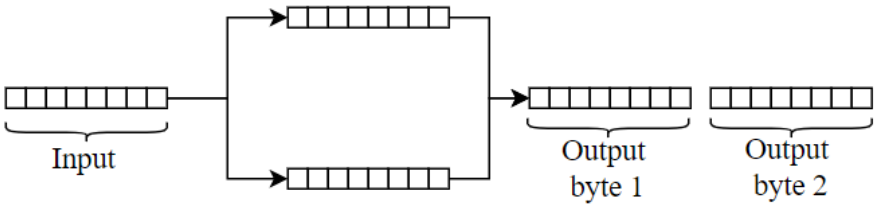
Docente: Gianluca Palermo

Sommario

1. Introduzione

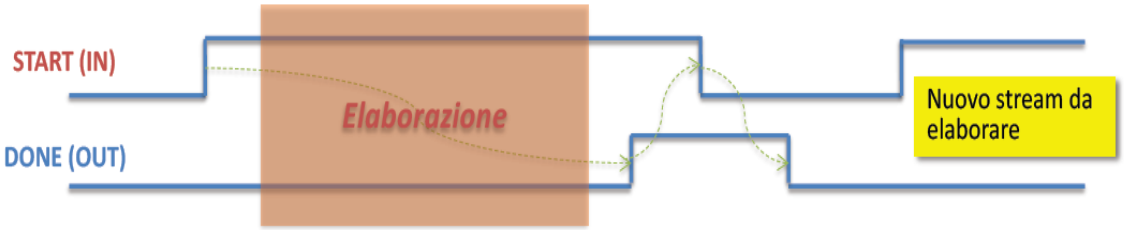
La prova finale del corso di Reti Logiche dell'anno corrente, 2021-2022, prevede l'implementazione di un modulo hardware, descritto in linguaggio VHDL, che legga dalla memoria una serie di byte, per poi elaborarli e scriverne il risultato sulla RAM. Tale processo deve poter essere iterato, consentendo così la computazione di più flussi di dati.

In particolare, per ogni flusso di elaborazione, il byte all' indirizzo 0 della memoria indica il numero n di byte successivi e consecutivi da leggere, tale numero è sempre compreso tra 1 e 255. La scrittura dei 2 · n byte di output deve cominciare invece all' indirizzo 1000 della memoria.



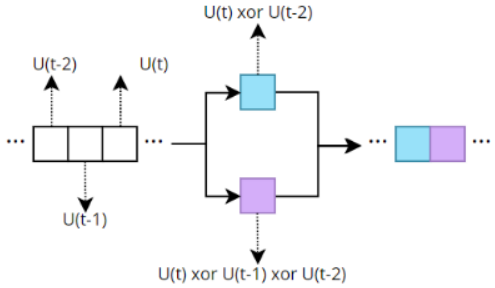
1.1 Inizio e fine della computazione

Il modulo, che entra in uno stato consistente solo dopo aver ricevuto un segnale di reset, deve iniziare ad elaborare non appena il segnale START di input passerà da 0 a 1, e rimarrà tale finché non si è completata la computazione e la scrittura in memoria di tutti i 2 · n byte di uscita, quando quindi verrà posto a 1 il segnale di DONE. DONE dovrà essere settato a 0 solo dopo che l'input START risulterà 0, e solo a questo punto sarà possibile ricevere un nuovo segnale di START, avendo o meno ricevuto il segnale di reset, e ricominciare con la fase di codifica.



1.2 La logica di trasformazione dell'input

Ogni byte letto da memoria sarà composto da 8 bit U_k, ciascuno dei quali genererà due bit $P1_k$ e $P2_k$, con $P1_k(t)=U_k(t)$ xor $U_k(t-2)$ e P2k(t) = Uk(t) xor Uk(t-1) xor Uk(t-2). L'output $Y_k(t)$ è dato dalla concatenazione di $P1_k$ e $P2_k$.



2. Architettura

2.1 Scelte implementative

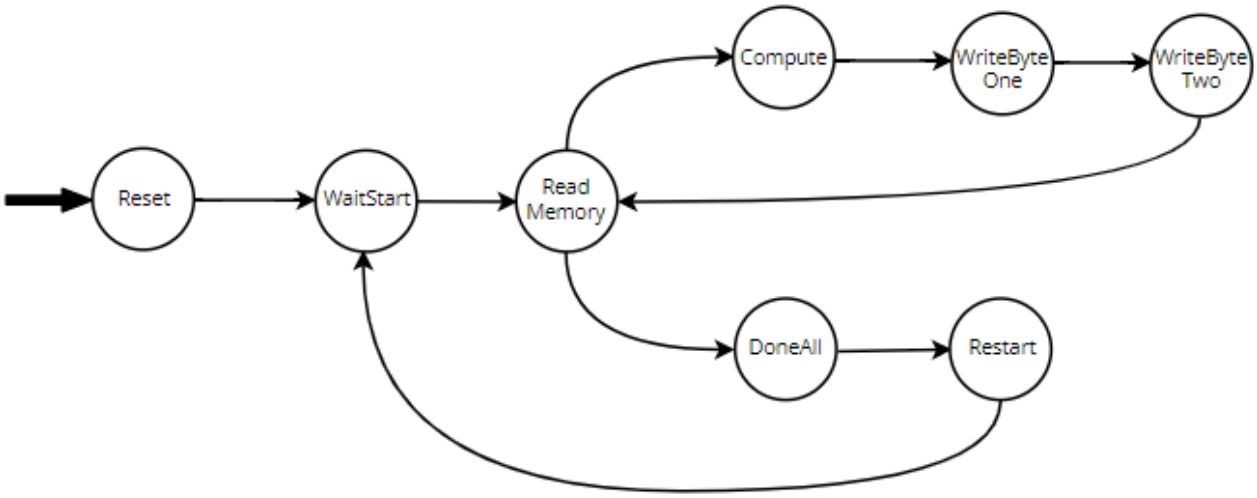
Dopo aver inizialmente sperimentato un approccio "bottom-up", iniziando a realizzare diversi moduli da poi far dialogare, ho deciso di cambiare strategia e descrivere un solo modulo, che implementa, all' interno di un unico processo, una macchina a stati finiti che legge da memoria quanti byte successivi devono essere processati, e itera per tal numero di volte il procedimento: lettura del byte da memoria, elaborazione e costruzione dell'output, scrittura in memoria di tale output. In tal modo ho implementato un codice a esecuzione sequenziale, eliminando ogni problema di concorrenza tra processi, che ho invece riscontrato con il primo approccio.

Il modulo, oltre ai segnali di input e output presenti nella entity fornitaci assieme alle specifiche, fa uso di alcuni segnali aggiuntivi, come:

- NUM: un segnale "standard logic vector" a 8 bit utilizzato per salvare il primo byte letto da memoria, che contiene il numero di byte di input da leggere successivamente;
- INCOUNT: un segnale "signed" utilizzato come contatore del numero di byte letti da input, escluso il primo (quello all' indirizzo 0 della memoria);
- OUTCOUNT: un segnale "signed" utilizzato come contatore del numero di byte scritti in output;
- P1: un segnale "standard logic vector" utilizzato per trasformare i primi 4 bit del segnale I_DATA nel primo byte da scrivere in memoria;
- P2: un segnale "standard logic vector" utilizzato per trasformare gli ultimi 4 bit del segnale I_DATA nel secondo byte da scrivere in memoria;
- ULTIMO: un segnale "standard logic" utilizzato per memorizzare l'ultimo bit del segnale I_DATA (quello in posizione 0), necessario per la corretta elaborazione dell'input successivo;

- PENULTIMO: un segnale "standard logic" utilizzato per memorizzare penultimo bit del segnale I_DATA (quello in posizione 1), necessario per la corretta elaborazione dell'input successivo

2.2 La macchina a stati



2.2.1 Stato 1: Reset

Non appena il processo ha inizio, quando viene dato il segnale di reset, vengono settati tutti i segnali con il loro valore di reset: *ultimo* e *penultimo* a 0, così come *incount, outcount*, *P1*, *P2*, *o_done, o_we,* e *o_address*; mentre *o_en* viene posto a 1 per permettere la lettura della memoria. Lo stato passa dunque da "Reset" a "WaitStart".

2.2.2 Stato 2: WaitStart

In questo stato, se il segnale *i_start* è alto, viene letto il byte all' indirizzo di *incount* (che è appena stato resettato a 0), e viene copiato in *num,* dove quindi avremo il numero di byte da processare. Fatto ciò, è possibile passare allo stato successivo: ReadMemory.

2.2.3 Stato 3: ReadMemory

Il terzo stato verifica se il contatore degli ingressi letti, *incount,* è minore del numero degli ingressi da leggere (*num):* in caso di verifica positiva, viene letto dalla memoria il byte successivo per poi passare allo stato Compute; se invece il riscontro

è negativo, e dunque sono stati letti e scritti in memoria tutti i dati richiesti, allora lo stato assume il valore "DoneAll" dell'enumerazione degli stati.

2.2.4 Stato 4: Compute

In questo stato vengono calcolati bit per bit i valori dei due segnali *P1* e *P2*, in modo tale da seguire la logica di trasformazione descritta precedentemente. Dopodiché a *penultimo* e *ultimo* vengono assegnati rispettivamente i valori *i_data(1)* e *i_data(0)*, in modo tale da "memorizzare" gli ultimi due bit del segnale di input corrente per la trasformazione del byte successivo. Lo stato viene infine modificato in "WriteByteOne".
Poiché i bit da codificare sono soltanto 16 (8 per *P1* e 8 per *P2*), e avendo ciascuno di essi una formula semplice e determinata, ho deciso di non introdurre nessun ciclo "for", bensí di scrivere singolarmente tutte e 16 le assegnazioni dei bit.

2.2.5 Stato 5: WriteByteOne

Per prima cosa, così da poter scrivere sulla memoria, nel quinto stato viene portato il valore di *o_we* alto; conseguentemente vengono //settati i segnali *o_address* e *o_data* rispettivamente con i valori 1000+*outcount* e *P1*. Lo stato passa a "WriteByteTwo".

2.2.6 Stato 6: WriteByteTwo

Utilizzando i segnali *o_address* e *o_data,* all'indirizzo 1000+*outcount*+1 viene scritto il valore di *P2*. I contatori *incount* e *outcount* vengono quindi incrementati rispettivamente di una e due unità: si può ora tornare nello stato "ReadMemory".

2.2.7 Stato 7: DoneAll

Se si è arrivati in questo stato, allora significa che sono già stati letti tutti gli input e scritti tutti i rispettivi output: il segnale *o_done* viene posto a 1, e tutti gli altri segnali vengono resettati (viene fatto un "Reset" implicito). Si passa all' ultimo stato: quello di "Restart".

2.2.8 Stato 8: Restart

In quest' ultimo stato, se *i_start* è basso, viene posto *o_done a 0* e si ritorna nello stato di "WaitStart" nell'attesa dell'inizio di un nuovo flusso di computazione.

3. Testing

3.1 Le scelte

Poiché il codice elaborato si compone di un solo modulo, e il suo processo di scrittura è stato piuttosto breve, la fase di testing è avvenuta solo a codice completato, nonostante fossi consapevole che è buona norma verificare in itinere la correttezza di ciò che si sta scrivendo. A posteriori, riconosco che avrei potuto testare il codice al completamento di ciascuno stato, così da rendermi conto se il comportamento in quello stato fosse effettivamente quello desiderato.

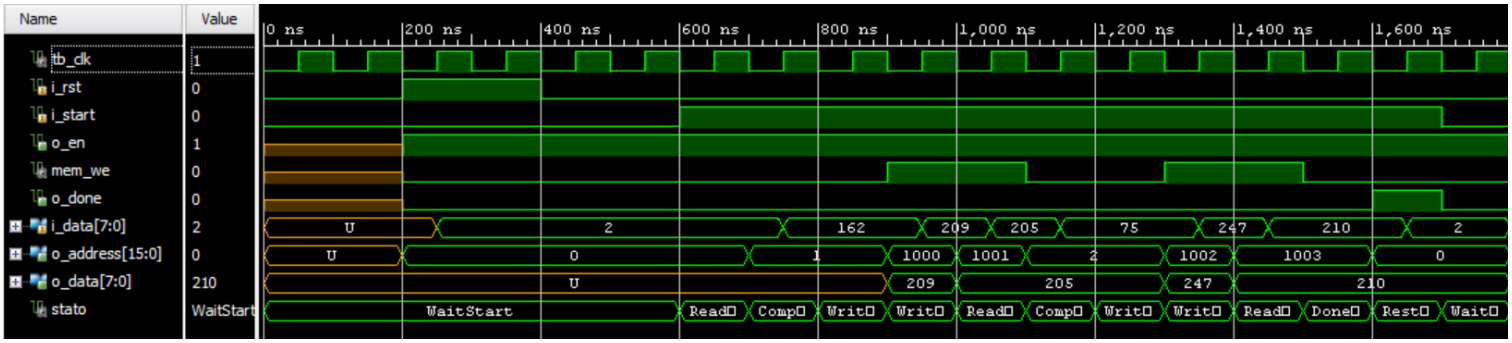
3.2 I testbench

Tra i testbench eseguiti con successo, alcuni sono volti alla verifica della corretta gestione di:

- Flussi consecutivi di dati identici;
- Caso in cui il numero di byte da leggere è nullo;
- Caso in cui il numero di byte da leggere è massimo, ovvero 255;
- Caso in cui tra un flusso e il successivo è dato esplicitamente il segnale di reset;
- Caso in cui tra un flusso e il successivo non è dato esplicitamente il segnale di reset;

Sono poi stati eseguiti altri testbench generati in modo casuale per confermare che il funzionamento del modulo fosse corretto, e quindi ridurre la probabilità di trascurare corner case mal gestiti.

3.3 Il testbench fornito



Nell' immagine si può osservare il funzionamento del modulo descritto: in particolare si nota come si succedano gli stati della macchina. Non appena è alto il segnale di reset, vengono inizializzati i segnali e ci si porta nello stato di WaitStart, dove viene letto l'indirizzo 0 della memoria, che risulta essere 2. Dopodiché si legge 162 dall'indirizzo 1, e si scrivono 209 e 205 rispettivamente agli indirizzi 1000 e 1001. In seguito, viene letto 75 dall'indirizzo 2, e vengono scritti 247 e 210 in 1002 e 1003. Fatto ciò, lo stato passa a DoneAll, poi Restart e WaitStart, quando viene terminato il testbench.

4. Conclusioni

Il funzionamento del componente descritto è stato simulato estensivamente sia in modalità "Behavioral" che "Funzionale post sintesi", con casi assegnati e con test generati casualmente. Sulla base di ciò si può affermare che il modulo rispetti le specifiche. In aggiunta, si segnala che il componente sia in grado di lavorare anche a frequenze del segnale di clock molto maggiori di quelle richieste: aggiungendo un constraint sul clock ed eseguendo il comando "report_timing" da console, infatti, si può verificare che la metrica *slack* risulti 95.796, ossia che il periodo minimo del clock per il funzionamento è 4,204 ns.