POLITECNICO
MILANO 1863

Relazione progetto di Reti Logiche

Lorenzo Simone

Prof. Gianluca Palermo - A.A. 2023/2024

1 Introduzione

L'obiettivo della prova finale di Reti Logiche era quello di implementare, attraverso l'utilizzo del linguaggio VHDL, un componente HW sincrono che, agendo sul fronte di salita del segnale di clock, fosse in grado di interfacciarsi con una memoria. Il componente doveva essere in grado di

- Leggere, contestualmente all'attivazione di un segnale di avvio (`i_start=1`), un messaggio composto da una sequenza di K parole W, con valori compresi tra 0 e 255. La sequenza di K parole è memorizzata a partire da un indirizzo di memoria specificato (ADD), ogni 2 byte (ADD, ADD+2, ADD+4, ..., ADD+2*(K-1)).
- Completare la sequenza sostituendo le parole con valore 0 con l'ultimo valore non 0 letto e inserendo un valore di "credibilità" C nel byte mancante per ogni valore della sequenza. Il valore di credibilità C, inizializzato a 0, è 31 quando il valore della parola W è diverso da 0 e viene decrementato di 1 rispetto al valore precedente ogni volta che si incontra uno zero in W. Il valore C è sempre maggiore o uguale a 0 e si reinizializza a 31 ogni volta che si incontra un valore W diverso da zero.

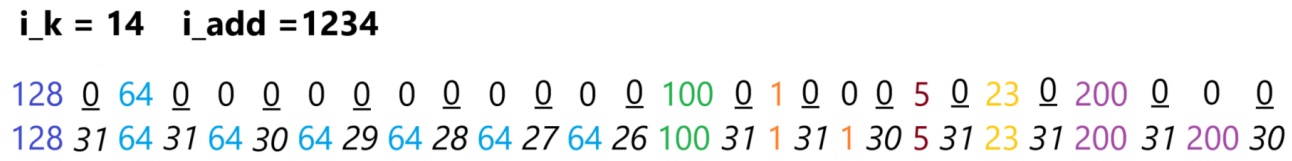

Figura 1: Esempio di sequenza in ingresso (sopra) e dopo l'elaborazione (sotto). La sequenza viene completata sostituendo quando necessario il valore 0 con l'ultimo valore valido e inserendo il valore di credibilità.

Viene di seguito fornita l'interfaccia del componente da implementare, espressa in VHDL e mostrata una rappresentazione del modulo da implementare che, descritto come una *black box*, viene direttamente inserito nell'ambiente di testing e collegato ad una memoria.

```
entity project_reti_logiche is
   port (
       i_clk   : in std_logic;
       i_rst   : in std_logic;
       i_start : in std_logic;
       i_add   : in std_logic_vector(15 downto 0);
       i_k     : in std_logic_vector(9 downto 0);

       o_done  : out std_logic;

       o_mem_addr : out std_logic_vector(15 downto 0);
       i_mem_data : in std_logic_vector(7 downto 0);
       o_mem_data : out std_logic_vector(7 downto 0);
       o_mem_we   : out std_logic;
       o_mem_en   : out std_logic
   );
end project_reti_logiche;
```

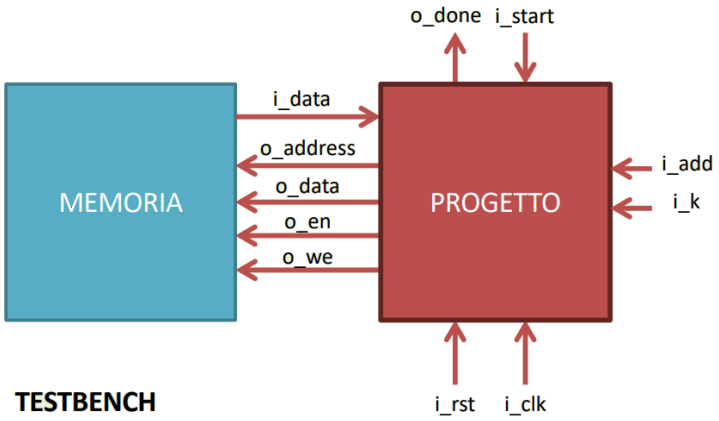

Figura 2: Descrizione astratta del modulo posto nel test bench e collegato alla memoria.

2 Architettura

Per svolgere il progetto ho optato per la realizzazione di una macchina a stati finiti sincrona, comandata dal segnale di clock `i_clk`, descritta in modo **behavioral** tramite l'utilizzo di due *process*, il primo rappresenta la parte sequenziale della macchina e serve a gestire il Register Transfer, mentre il secondo rappresenta la FSM che analizza i segnali in ingresso e lo stato corrente per determinare il prossimo stato in cui evolverà il sistema.
È stato inoltre necessario inserire dei segnali aggiuntivi che hanno permesso una gestione adeguata dello stato interno della macchina, una corretta elaborazione dei segnali in ingresso e una corretta presentazione dei segnali in uscita. La loro funzione, presentata in questa relazione in modo astratto, può essere direttamente ritrovata e compresa nel codice `VHDL`. Vengono di seguito presentati i segnali aggiuntivi che sono stati utilizzati:

2.1 Segnali utilizzati

`current_state : STATE;` Stato corrente. Si noti che il tipo `STATE` è definito come segue:

```
type STATE is ( IDLE, START, CHECK, READ, WRITE, STORE, DONE );
```

`data : std_logic_vector(7 downto 0);` Registro contenente l'ultimo valore valido (diverso da zero) letto dalla memoria nella sequenza di parole in input.

`cred : std_logic_vector(4 downto 0);` Registro contenente il valore di credibilità con cui completare la sequenza.

`offset : integer;` Intero che tiene traccia della differenza di posizione fra la cella di memoria che sta venendo letta/scritta e quella di partenza definita in `i_add`.

2.2 Macchina a stati finiti

Vengono di seguito presentati gli stati della macchina sintetizzata. Per ogni stato è fornita una descrizione dettagliata delle operazioni che vengono svolte e dei segnali interessati. Si noti che la macchina esegue le operazioni sul **fronte di salita** del clock.

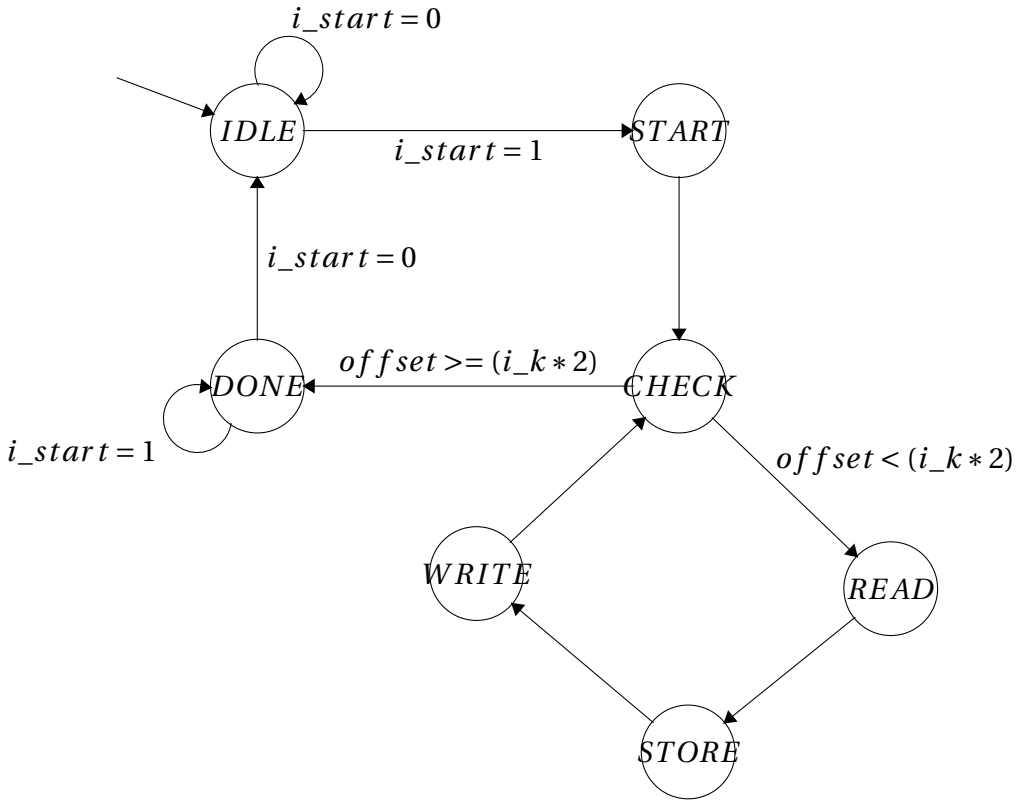

Figura 3: Diagramma degli stati della macchina. Si noti che il segnale `i_clk` è stato omesso per chiarezza, e che il segnale `i_rst` è in grado, in modo asincrono, di riportare la macchina allo stato `IDLE`, comandando il reset a tutti i segnali presenti nella macchina. Anche la transizione verso `IDLE` da ogni singolo stato dovuta all'attivazione di `i_rst` viene omessa per chiarezza.

IDLE Lo stato `IDLE` è lo stato in cui si trova la macchina al momento dell'attivazione, e viene raggiunto nel caso di reset asincrono (`i_rst=1`) o quando, al termine di una elaborazione, il segnale `i_start` viene portato a 0. La macchina rimane nello stato `IDLE` finché `i_start=0`. In questo stato la macchina si prepara a far partire una nuova elaborazione quando verrà alzato il segnale `i_start`.

START Lo stato `START` inizializza tutti i segnali a zero e imposta `o_mem_en=1` per permettere al componente di comunicare con la memoria.

CHECK Lo stato `CHECK` imposta `o_mem_we=0`, poi compara il valore di `offset` con il valore di `i_k` moltiplicato per due; se il valore di offset risulta maggiore o uguale la macchina passa allo stato `DONE`, altrimenti passa allo stato lo stato `READ`.

READ Lo stato `READ` legge dalla memoria alla posizione `i_add + offset` e, se il valore letto è maggiore di 0, lo inserisce nel registro `data` e imposta il valore di `cred` a 31.

STORE Lo stato `STORE` imposta `o_mem_we=1` per poter scrivere nella memoria in posizione `i_add + offset` il valore del registro `data`. Infine lo stato incrementa `offset` di 1.

WRITE Lo stato `WRITE` scrive in memoria in posizione `i_add + offset` il valore di credibilità contenuto in `cred`, che, se maggiore di 0, viene successivamente decrementato di 1. Infine lo stato incrementa `offset` di 1.

DONE Lo stato `DONE` viene raggiunto al termine dell'elaborazione e attende che il segnale `i_start` si abbassi per tornare allo stato `IDLE`. Finché `i_start=1` rimane vero la macchina resta nello stato `DONE` e mantiene il segnale `o_done=1`.

3 Risultati sperimentali

3.1 Report di sintesi

Il progetto è stato testato sulla versione di **Vivado 2022.2**, utilizzando come FPGA target **xc7a200tfbg484-1**. Il codice `VHDL` è correttamente sintetizzabile e la sintesi non genera alcun errore. Si noti che non sono presenti Latch nel seguente estratto del documento di design.

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	54	0	0	134600	0.04
LUT as Logic	54	0	0	134600	0.04
LUT as Memory	0	0	0	46200	0.00
Slice Registers	48	0	0	269200	0.02
Register as Flip Flop	48	0	0	269200	0.02
Register as Latch	0	0	0	269200	0.00
F7 Muxes	0	0	0	67300	0.00
F8 Muxes	0	0	0	33650	0.00

Per quanto riguarda invece il timing, viene riportato che tutti i constraint sono rispettati, ed è interessante analizzare lo *Slack*.

```
Slack (MET) :    16.379ns  (required time - arrival time)
```

In questo caso, quindi, il massimo ritardo dovuto alla percorrenza dei cammini è ampiamente minore del periodo di clock, il che significa che il sistema potrebbe operare anche con frequenze di clock F_{clk} più alte rispetto a quella con cui è stato testato.

3.2 Test Bench e simulazioni

Per quanto riguarda la fase di simulazione e testing, fondamentale per verificare il corretto comportamento sotto stimoli differenti, il componente è risultato sintetizzabile e correttamente simulabile in **Pre sintesi** e **Post sintesi comportamentale**. È stato utilizzato il Testbench fornito dal docente, che copriva il funzionamento generico del componente e la corretta gestione dei segnali `o_done` e `o_mem_en`, e sono stati sviluppati autonomamente altri 7 Testbench necessari a testare il componente in questi casi limite o casi particolari:

- Elaborazioni multiple. Dopo aver terminato un'elaborazione il componente è in grado di effettuare successive elaborazioni senza dover attendere il segnale di reset.
- Sequenza di 31 o più parole uguali a 0 che portano la credibilità ad azzerarsi. Il componente si comporta correttamente non decrementando ulteriormente il valore di C quando arriva a zero.
- Sequenza che inizia con uno o più valori uguali a 0. Il componente si comporta correttamente mantenendo i valori e la credibilità uguali a zero finché non vengono trovati valori validi.
- Sequenza di lunghezza 0. Il componente si comporta correttamente, non modificando la memoria quando la lunghezza `i_k` in input è uguale a 0.
- Sequenza di lunghezza 1 o 1023. Il componente si comporta correttamente, elaborando tutta la sequenza.
- Sequenza di soli 0. Il componente si comporta correttamente, non modificando la sequenza e di conseguenza memoria.
- Reset durante l'esecuzione. Il componente si comporta correttamente eseguendo il reset in maniera asincrona in qualsiasi momento dell'esecuzione.

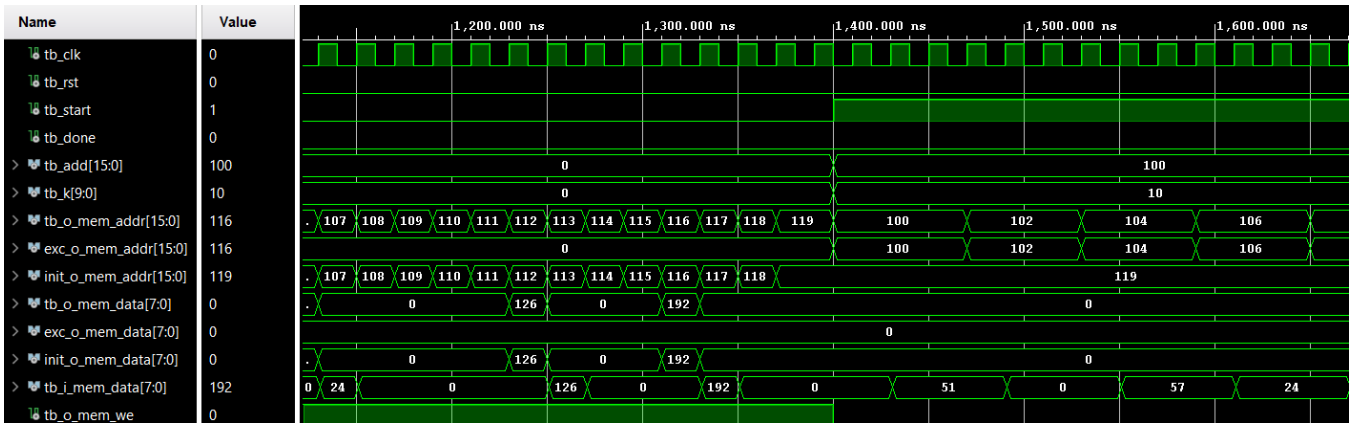

Figura 4: Schermata dell'esecuzione di un Testbench che simula elaborazioni multiple

4 Conclusioni

Il componente implementato, correttamente sintetizzabile e simulabile sia in pre che in post sintesi rispetta a pieno la specifica fornita come mostrato dall'estesa fase di simulazione e testing a cui è stato sottoposto. Una prima fase di sviluppo del componente utilizzava un array per salvare la sequenza elaborata e scriverla successivamente in memoria, rimuovendo questo sistema ridondante e salvando in memoria la sequenza aggiornata subito dopo averla letta, il numero di Flip Flop utilizzati è sceso da circa 16000 a 48. Un'ulteriore possibile ottimizzazione potrebbe essere quella di evitare di sovrascrivere inutilmente i valori in memoria già validi (diversi da 0) e passare direttamente a completare il byte successivo con la credibilità.
Le principali difficoltà sorte durante lo sviluppo del del componente sono state dovute al contatto con strumenti e tecnologie nuove. Lo sviluppo del progetto è stato particolarmente utile per approfondire e chiarire aspetti e argomenti già affrontati nel corso di Reti Logiche.