Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria

15 Luglio 2024

Prova Finale di Reti Logiche
2023 - 2024

Studente: Andrea Sanvito
Matricola: 983819
Codice Persona: 10814394

Docente: Gianluca Palermo

1 Requisiti del Progetto

1.1 Descrizione del problema

Il progetto riguarda l'implementazione in VHDL di un modulo in grado di modificare valori all'interno di una memoria RAM, seguendo determinate specifiche.

Viene dato in input al modulo un indirizzo da 16 bit della memoria `i_add` e un valore intero `i_k` da 10 bit. Il compito del modulo è di:

- accedere alla memoria all'indirizzo `i_add` fornito in input;
- intendere la memoria come una sequenza di "coppie" di numeri, dove il primo rappresenta un dato, e il secondo un valore di credibilità del dato stesso.
- scorrere le `i_k` coppie in memoria una ad una, e ad ogni passo:
  i. se il dato è specificato (ossia diverso da zero), lo si mantiene in memoria e si imposta il valore di credibilità seguente al massimo;
  ii. altrimenti, si aggiorna il dato in memoria con l'ultimo dato valido letto, e si decrementa il valore di credibilità.

L'implementazione deve essere completamente sincrona con un segnale di clock fornito esternamente, eccetto per il segnale di reset, il quale è invece asincrono.

1.2 Ipotesi Progettuali

Il progetto è stato svolto con determinate ipotesi:

i. l'indirizzo `i_add` deve essere valido, ossia non deve superare la dimensione della memoria;

ii. la somma tra `i_add` e `i_k` deve essere valida, ossia non deve superare la dimensione della memoria;

Il comportamento del modulo implementato non è specificato per i casi sopra elencati.

Inoltre, per quanto riguarda i segnali di start asseriti durante un segnale di reset, si è interpretato dalle specifiche che il modulo debba attendere un nuovo segnale di start. Di conseguenza, il modulo non avvierà l'elaborazione con lo stesso segnale di start asserito durante il reset.

E' riportato di seguito un esempio di elaborazione.

1.3 Esempio

Consideriamo l'indirizzo di memoria iniziale `i_add` pari a 127 e `i_k` pari a 16.

Il nostro modulo accede in lettura alla memoria all'indirizzo specificato: essendo il dato diverso da zero, lo salva come ultimo valore valido e porta il valore di credibilità al massimo (da specifica, il massimo equivale a 31), scrivendolo nella cella seguente.

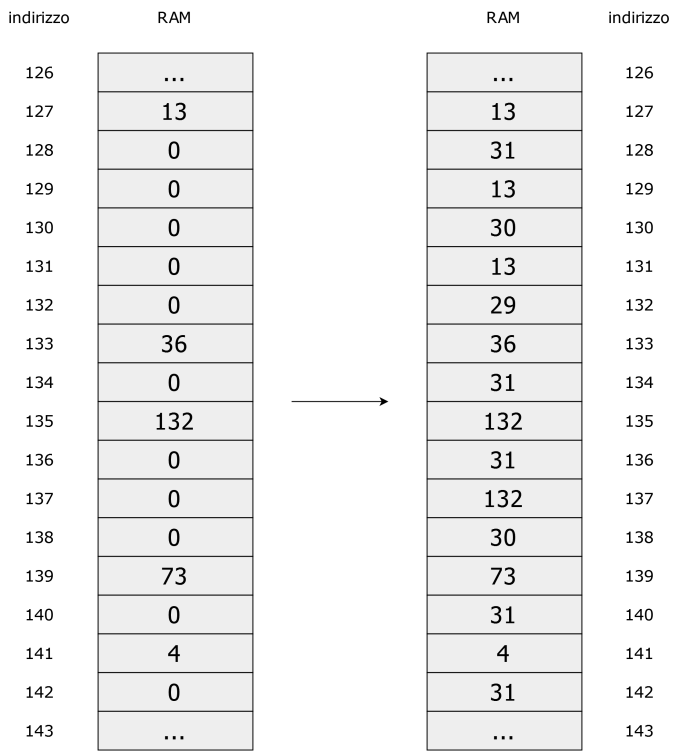


Figure 1: memoria RAM prima e dopo le operazioni svolte dal modulo

La cella successiva a quella appena scritta contiene uno zero, ossia un valore non specificato: il modulo lo sostituisce inserendo l'ultimo valore valido, e decrementa la credibilità, scrivendola nella cella successiva.

Questo processo continua fino alla cella all'indirizzo 133, dove viene trovato un dato diverso da zero. Il modulo procede ad aggiornare il proprio ultimo valore valido letto, e a riportare la credibilità a 31, scrivendola nella cella successiva.

Tali azioni si ripetono fino a quando il contatore `k`, decrementato ad ogni lettura, raggiunge lo zero.

2 Architettura

2.1 Specifica

La specifica del problema richiede che la soluzione includa un componente con 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_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;
```

Il modulo deve dunque comunicare con altre entità, in particolare:

- un utilizzatore, attraverso i segnali in input `i_clk`, `i_rst`, `i_start`, `i_add`, `i_k`;
- una memoria RAM, sfruttando i segnali in output `o_mem_addr`, `o_mem_data`, `o_mem_we`, `o_mem_en`, e un segnale in input `i_mem_data`.

E' anche utilizzato un segnale in output `o_done` per comunicare all'utilizzatore la fine dell'elaborazione da parte del modulo.

2.2 Finite State Machine

L'implementazione è stata ottenuta attraverso un singolo modulo, una FSM, con due processi. Essa comunica con la memoria, esegue computazione, e asserisce gli output.

Per gli stati nella rappresentazione seguente della FSM, descritti con maggiore dettaglio nella sezione seguente, sono state utilizzate le seguenti abbreviazioni:

```
SR_DATA:       SET_READ_DATA
W_MEM:              WAIT_MEM
F_DATA:           FETCH_DATA
W_DATA:           WRITE_DATA
W_CRED:   WRITE_CREDIBILITY
ITER:                ITERATE
```

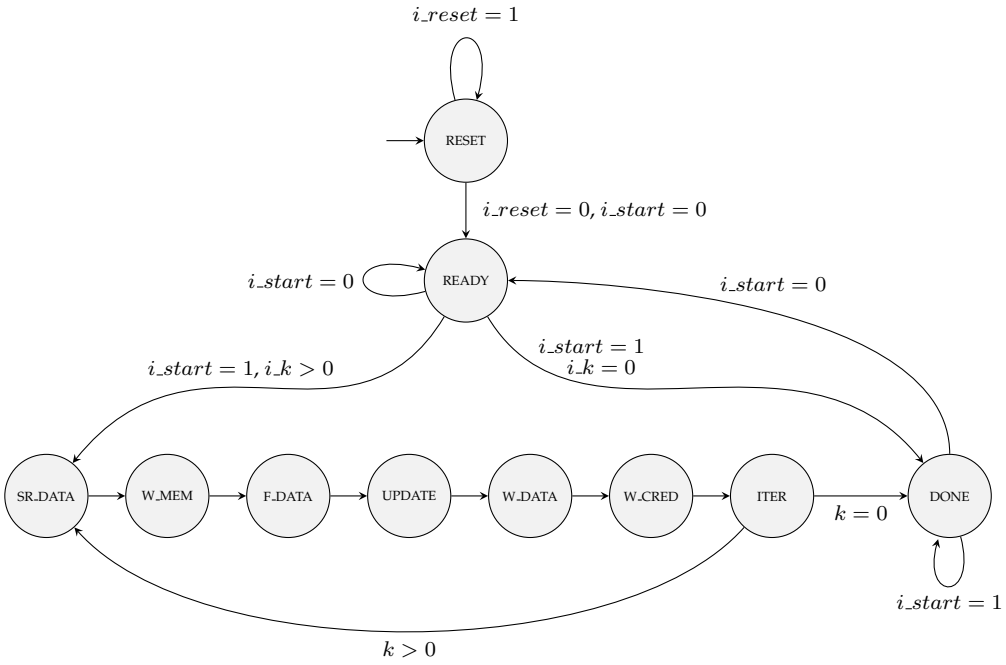


Figure 2: schema della FSM del modulo

2.3 Stati

Le operazioni compiute dal modulo in un ciclo di clock dipendono dallo stato in cui essa si trova. Gli stati utilizzati sono:

- **`RESET`**: è lo stato iniziale e lo stato in cui il modulo si troverà dopo un segnale di reset in ingresso. Da specifica, la FSM rimarrà in questo stato fino a quando `i_reset` e `i_start` non verranno portati a 0. Se il segnale di start è già alto durante la transizione a 0 del segnale di reset, il modulo aspetterà un nuovo fronte di salita del primo;
- **`READY`**: è lo stato di idling. Il modulo rimane in attesa di un segnale di start;
- **`SET_READ_DATA`**: stato in cui il modulo comunica alla memoria la volontà di accedere in lettura ad essa, ad un determinato indirizzo `addr`;
- **`WAIT_MEM`**: la memoria fornita da specifica richiede un ciclo di clock in più per asserire in output i dati richiesti. Questo rende necessario che il modulo attenda un ulteriore ciclo di clock per far sì che i dati in ingresso siano quelli corretti;
- **`FETCH_DATA`**: stato in cui il modulo recupera il dato dalla memoria;
- **`UPDATE`**: stato in cui il modulo esegue le operazioni a seconda dei dati letti.
- **`WRITE_DATA`**: stato in cui il modulo scrive il dato nella memoria;
- **`WRITE_CREDIBILITY`**: stato in cui il modulo scrive il valore di credibilità nella memoria;
- **`ITERATE`**: stato in cui i segnali interni vengono aggiornati e ”preparati” per la prossima iterazione, sfruttando i segnali di *next*;

- **DONE**: stato che sancisce la fine dell'elaborazione. Il modulo esce da questo stato solamente all'alzarsi del segnale di `i_start` in ingresso. Il modulo tornerà allo stato `READY`, abbassando il segnale di `o_done`.

2.4 Processi

Il modulo è basato su due processi:

- **STATE_REG**: permette di aggiornare i segnali interni e lo stato della FSM. Nella lista di sensibilità sono presenti il segnale di reset e il segnale di clock.
  Se il processo viene attivato dal segnale di reset, il modulo viene reimpostato azzerando tutti i segnali e gli output. Invece, se viene attivato da una variazione del segnale di clock, verifica se si tratta di un fronte di salita e, in tal caso, imposta tutti i segnali ai corrispondenti valori *next*, compreso il segnale di stato. Questo processo modifica solo i segnali "correnti";
- **LAMBDA_DELTA**: Il processo è composto da uno switch-case basato sullo stato corrente. Ogni case contiene le azioni da eseguire per lo specifico stato, in termini di aggiornamento sia dei segnali interni che degli output. Questo processo modifica solo i segnali *next*.

É importante notare l'attenzione posta alle modifiche dei segnali interni per evitare errori riguardanti segnali "multiply driven", ossia segnali modificati concorrentemente da più processi.

2.5 Segnali Interni

2.5.1 Segnali Correnti

All'interno del modulo sono stati utilizzati diversi segnali per la gestione dell'elaborazione. In particolare:

- **k**: rappresenta il numero di parole che ancora devono essere elaborate. Posto a `i_k` durante la fase di inizializzazione, viene decrementato ad ogni parola elaborata;
- **addr**: rappresenta l'indirizzo della memoria che il modulo sta elaborando. Inizialmente posto a `i_add`, viene incrementato a ogni lettura/scrittura sulla memoria;
- **data**: rappresenta il dato letto all'iterazione corrente;
- **last_valid_data**: rappresenta l'ultimo valore specificato (i.e. diverso da zero) letto in memoria. Usato per aggiornare le celle in cui il dato non è specificato;
- **credibility**: rappresenta il valore di credibilità dell'ultimo dato valido letto. Viene decrementato ogni volta che l'ultimo dato valido viene scritto e reimpostato a 31 ogni volta che viene letto un nuovo dato.

2.5.2 Segnali Successivi

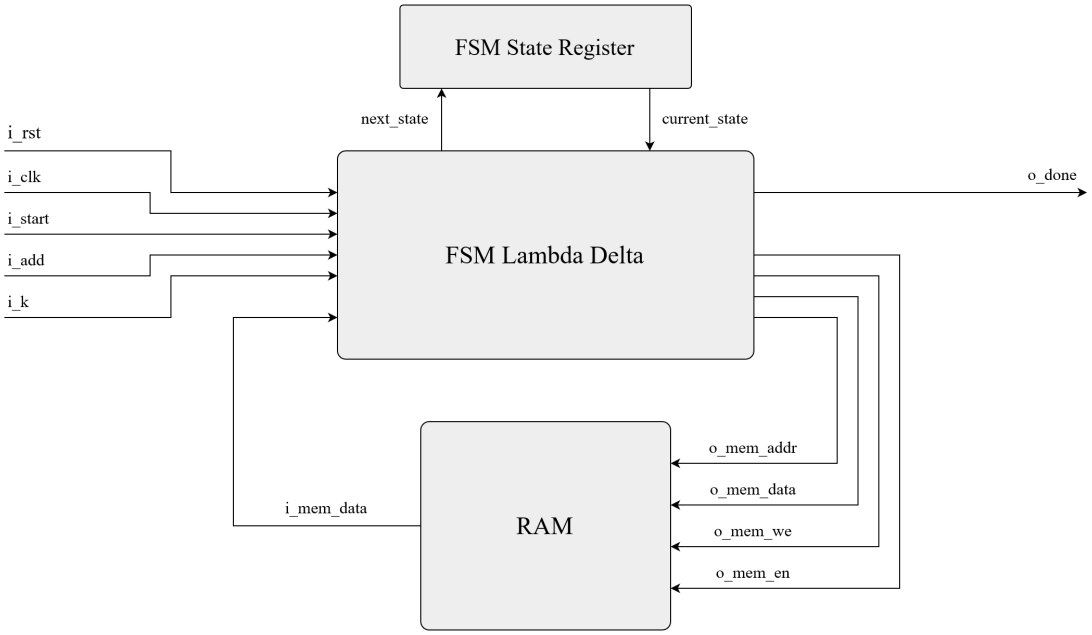
Per ogni segnale interno e di output sono anche inizializzati segnali di *next*, in modo da tenere traccia del valore che i segnali stessi dovranno avere al ciclo di clock successivo.

Ad ogni ciclo di clock, i segnali correnti vengono asseriti ai loro corrispondenti valori *next*, opportunamente preparati per il nuovo stato.

Questo design di implementazione permette di evitare l'utilizzo di latch non necessari.

2.6 Schema Implementativo

Di seguito lo schema implementativo, completo di RAM e segnali di input/output del modulo.



3 Risultati Sperimentali

3.1 Sintesi

Si è scelto per l'implementazione di utilizzare la board target **`xc7a200tfbg484-1`**. Il modulo viene correttamente sintetizzato, rispettando la specifica e ottenendo risultati notevoli in termini di report, qui sotto elencati.

3.1.1 Report di Utilizzo

Site Type	Used	Fixed	Prohibited	Available	Util %
Slice LUTs	76	0	0	134600	0.06
LUT as Logic	76	0	0	134600	0.06
LUT as Memory	0	0	0	46200	0.00
Slice Registers	87	0	0	269200	0.03
Register as Flip Flop	87	0	0	269200	0.03
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

Leggendo il report, possiamo notare che il modulo utilizza:

- **`LUT`**: 76 (0.06% del totale disponibile)
- **`FF`**: 87 (0.03% del totale disponibile)

Una percentuale così bassa può essere spiegata dalla semplicità del problema da risolvere, che non richiede una logica complessa.

Durante la scrittura del codice è stata posta molta attenzione per evitare latch non necessari, portando così il loro numero a zero.

3.1.2 Report di Timing

Da specifica è richiesto che il modulo supporti un periodo di clock di *almeno* 20 ns.

Slack:	MET
Slack Time:	16.206 ns
Requirement:	20.000 ns
Data Path Delay:	3.412 ns
Logic:	0.999 ns (29.279%)
Route:	2.413 ns (70.721%)

Dal report leggiamo che il modulo rimane inattivo per più di 16 ns, ovvero più di quattro quinti del tempo disponibile. Ne evinciamo che l'implementazione soddisfa ampiamente la specifica, con un percorso critico inferiore ai 4 ns.

3.2 Simulazioni

3.2.1 Overview

Il codice è stato sviluppato seguendo una politica test-driven. I testbench elencati hanno permesso di scovare man mano criticità nel codice, risolte con versioni aggiornate dello stesso.

I testbench sono stati progettati appositamente per affrontare i punti critici, coprendo così tutti i possibili casi limite.

Sono di seguito elencate le macro sezioni, e per ognuna i singoli test effettuati.

3.2.2 Casi limite di Segnali in Input e dei valori in Memoria

I casi di test sono stati progettati per verificare il corretto funzionamento del modulo sia in condizioni normali sia nei casi limite. Particolare cura è stata dedicata a:

- `i_k` in ingresso pari a zero;
- cella all'indirizzo iniziale `i_add` della memoria contenente zero;
- condizioni tali da portare il valore di credibilità a zero durante l'elaborazione;
- memoria contenente solo zeri.

I casi di test hanno permesso di confermare il codice già scritto in precedenza, non avendo portato a nessun errore in elaborazione.

3.2.3 Asserimento di segnali di Reset e Start

Fondamentale è rispettare la specifica per quanto riguarda il funzionamento del modulo in caso di reset e di comandi di inizio di elaborazione. Si è voluto testare:

- segnale di start asserito prima del segnale di reset durante la fase iniziale;
- segnale di start mai abbassato dopo un segnale di reset;
- segnale di start asserito durante un segnale di reset;
- segnale di reset asserito durante la lettura in memoria.

I casi di test hanno rivelato un comportamento del modulo non conforme alla specifica: in presenza di un segnale di start durante un segnale di reset, il modulo iniziava l'elaborazione non appena il segnale di reset veniva azzerato.

Il codice è stato aggiornato affinché il modulo attenda un nuovo segnale di start ad ogni reset.

3.2.4 Elaborazioni Multiple

Il modulo deve essere in grado di gestire più elaborazioni successive, anche in assenza di segnali di reset.

3.2.5 Risultati

La versione finale del modulo riesce a eseguire tutti i testbench senza difficoltà, sia in simulazione **behavioral** che in **post-synthesis functional**.

Nonostante non fosse richiesto da specifica, si è deciso di simulare le elaborazioni anche in **post-synthesis timing**, tutte completate con successo.

4 Conclusioni

4.1 Contesto

La specifica sembra descrivere un problema quale potrebbe essere l'acquisizione di grandezze esterne attraverso, per esempio, strumenti di sensoristica. Ad ogni quanto di tempo, se il sensore ha letto un valore diverso da quello precedente, lo scrive in memoria, altrimenti non specifica il valore.

Per questo motivo è richiesto il valore di credibilità, una metrica che permette di conoscere quanto tempo prima un determinato valore è stato acquisito, ergo quanto è attendibile.

4.2 Implementazione

Si ritiene innanzitutto che l'implementazione del modulo rispetti le specifiche.

L'utilizzo di una FSM per gestire il flusso dell'elaborazione ha permesso di semplificare in maniera non ininfluente sia la complessità della computazione che la leggibilità del codice.

E' comunque necessario notare come potrebbero essere apportate alcune modifiche volte a migliorare ancor più l'implementazione.

In primo luogo, la FSM non è stata ottimizzata in termini di numero di stati in quanto non ritenuto necessario per il livello di complessità del problema; si è invece preferito mantenere più chiarezza e una maggiore divisione di responsabilità.

In secondo luogo, il numero di Flip Flop utilizzati per sintetizzare il componente potrebbe essere ridotto leggermente riducendo il numero di segnali interni; anche in questo caso non lo si è considerato necessario vista la natura del problema.