POLITECNICO
MILANO 1863

Prova Finale di Reti Logiche

Prof. Gianluca Palermo - Anno 2023 / 2024

Autori:	Lorenzo Tallarico, Pietro Roggero
Matricole:	957009, 958374
Codici Persona:	10719257, 10719258

1 Introduzione 2
1.1 Scopo del progetto 2
1.2 Specifiche Generali 2
1.3 Interfaccia del modulo 3
1.4 Esempio 5
2 Design e Architettura 6
2.1 Segnali 6
2.2 Macchina a stati 7
3 Risultati sperimentali 9
3.1 Sintesi 9
3.2 Simulazioni 9
3.2.1 Multi start con credibility partenza di zeri 10
3.2.2 Reset asincrono 10
4. Conclusioni 11

1 Introduzione

1.1 Scopo del progetto

L'obiettivo del progetto è implementare un modulo hardware descritto in VHDL che sia capace di interfacciarsi con una memoria, leggere un messaggio costituito da una sequenza di numeri interi di lunghezza variabile ed eventualmente modificare alcuni di essi secondo delle specifiche ben definite.

1.2 Specifiche Generali

Il sistema deve leggere una sequenza di *K* parole dalla memoria a partire da un determinato indirizzo *ADD*.
Ogni parola *W* è un numero intero compreso tra *0* e *255*, *0* va però considerato come valore non specificato.
Tra ogni parola presente un byte mancante che dovrà essere modificato in base ai valori precedentemente letti, quindi la prima parola si trova all'indirizzo *ADD*, la seconda all'indirizzo *ADD + 2*, la terza all'indirizzo *ADD + 4*, e così via fino ad *ADD + 2*(k-1)*.
L'algoritmo per riempire i byte vuoti non inizia fin tanto che trova valori non specificati, quindi tutte la parole uguali a *0* antecedenti al primo valore significativo non saranno considerate dal sistema, verranno semplicemente ignorate senza causare alcuna azione di scrittura.
Al primo valore diverso da *0* inizia il vero e proprio meccanismo di calcolo e scrittura dei byte mancanti, che segue due semplici regole:

- Se la parola *W* non è un valore non specificato, la si memorizza per un uso successivo e nel byte successivo si scrive *31*;
- Se la parola *W* è un valore non specificato, la si rimpiazza con l'ultima parola *W* memorizzata che era diversa da *0*. Il byte successivo invece sarà uguale all'ultimo byte mancante riempito e decrementato di un'unità, tenendo conto però che non ci possono essere numeri negativi e in caso il risultato sia minore di zero verrà scritto comunque *0*.

1.3 Interfaccia del modulo

Il modulo da descrivere in VHDL presenta 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;
```

Come possiamo vedere dalla figura precedente, il componente ha tre ingressi primari e un'uscita primaria.

- **i_start**: segnale in input composto da *1 bit*, quando alzato i due ingressi successivi vengono inizializzati;
- **i_k**: segnale in input composto da *10 bits* che rappresenta la lunghezza del messaggio, ovvero le *K* parole da leggere;
- **i_add**: segnale in input composto da *16 bits* che rappresenta l'indirizzo della prima parola della sequenza;
- **o_done**: segnale in output composto da *1 bit* che viene alzato solo quando la computazione termina e non può essere abbassato fin tanto che start rimane alto.

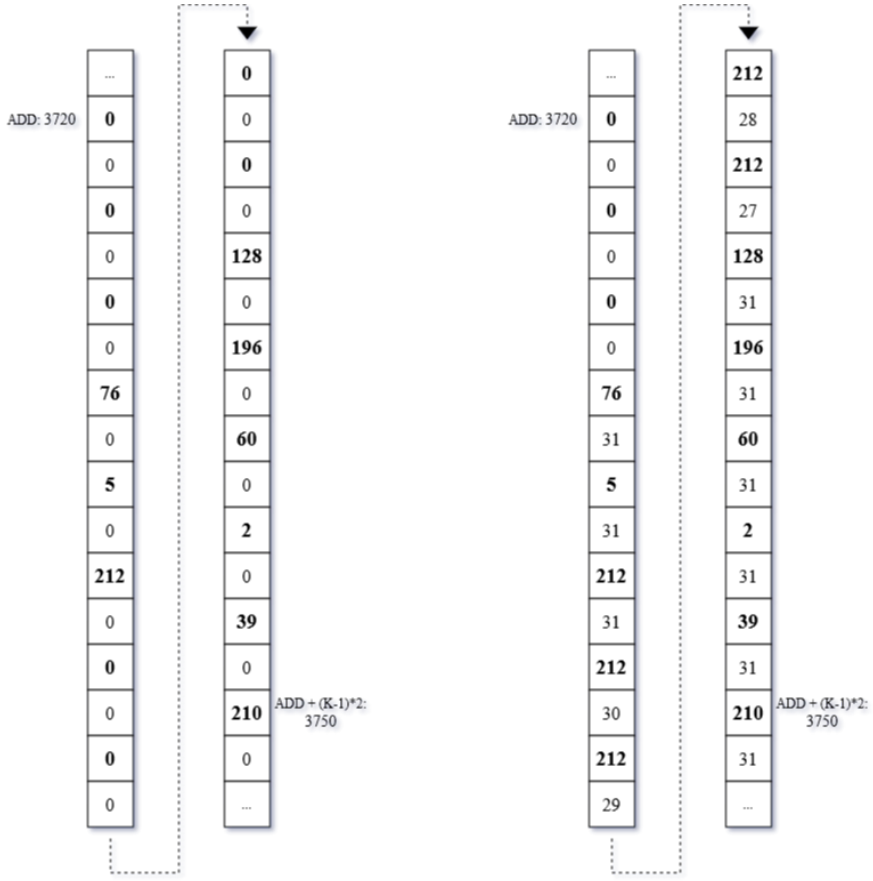
Gli ulteriori segnali presenti in ingresso e uscita sono:

- **i_clk**: segnale in input composto da *1 bit* che rappresenta il clock generato dal Test Bench, unico per tutto il sistema, tutti i segnali vengono interpretati sul fronte di salita;

- **i_rst**: segnale in input composto da *1 bit* che rappresenta il segnale di reset, è l'unico segnale asincrono. All'istante iniziale reset deve essere alto e quando viene abbassato parte l'elaborazione, ogni qual volta reset si alza viene reinizializzato il sistema;
- **i_mem_data**: segnale in input composto da *8 bits* che rappresenta il dato che arriva dalla memoria a seguito di una richiesta di lettura;
- **o_mem_addr**: segnale in output composto da *16 bits* che rappresenta l'indirizzo in memoria a cui si vuole accedere;
- **o_mem_data**: segnale in output composto da *8 bits* che rappresenta il dato da inviare alla memoria per un'operazione di scrittura;
- **o_mem_en**: segnale in output composto da *1 bit* che regola l'utilizzo della memoria, se alzato permette di comunicare con la memoria;
- **o_mem_we**: segnale in output composto da *1 bit* che indica che tipo di interazione effettuare con la memoria, scrittura quando è alto e lettura quando è basso.

1.4 Esempio

Nel seguente esempio abbiamo un messaggio lungo K = 16 che inizia all'indirizzo ADD = 3720.
A sinistra la memoria prima dell'avvio del modulo, a destra la memoria dopo che è stata terminata la computazione.



Dalla figura si nota come le prime tre parole vengano ignorate dal sistema e che solo dalla quarta parola in poi vengono riempiti i byte mancanti.
Si vede anche come le parole *7 - 8 - 9 - 10* vengono rimpiazzate con il valore della sesta parola siccome erano uguali a zero, inoltre il valore del byte mancante viene decrementato ogni volta fino all'undicesima parola dove torna a essere 31.
Tutte le celle di memoria precedenti o successive alla sequenza non vengono lette né modificate.

2 Design e Architettura

Da una prospettiva ad alto livello il modulo non è complicato, la prima bozza di una FSM prevedeva pochi stati, l'algoritmo non deve far altro che leggere dei byte dalla memoria ed eventualmente rimpiazzarli con dei valori diversi in base alle parole precedentemente lette.
La macchina a stati finiti finale è supportata da 7 principali segnali qui di seguito elencati.

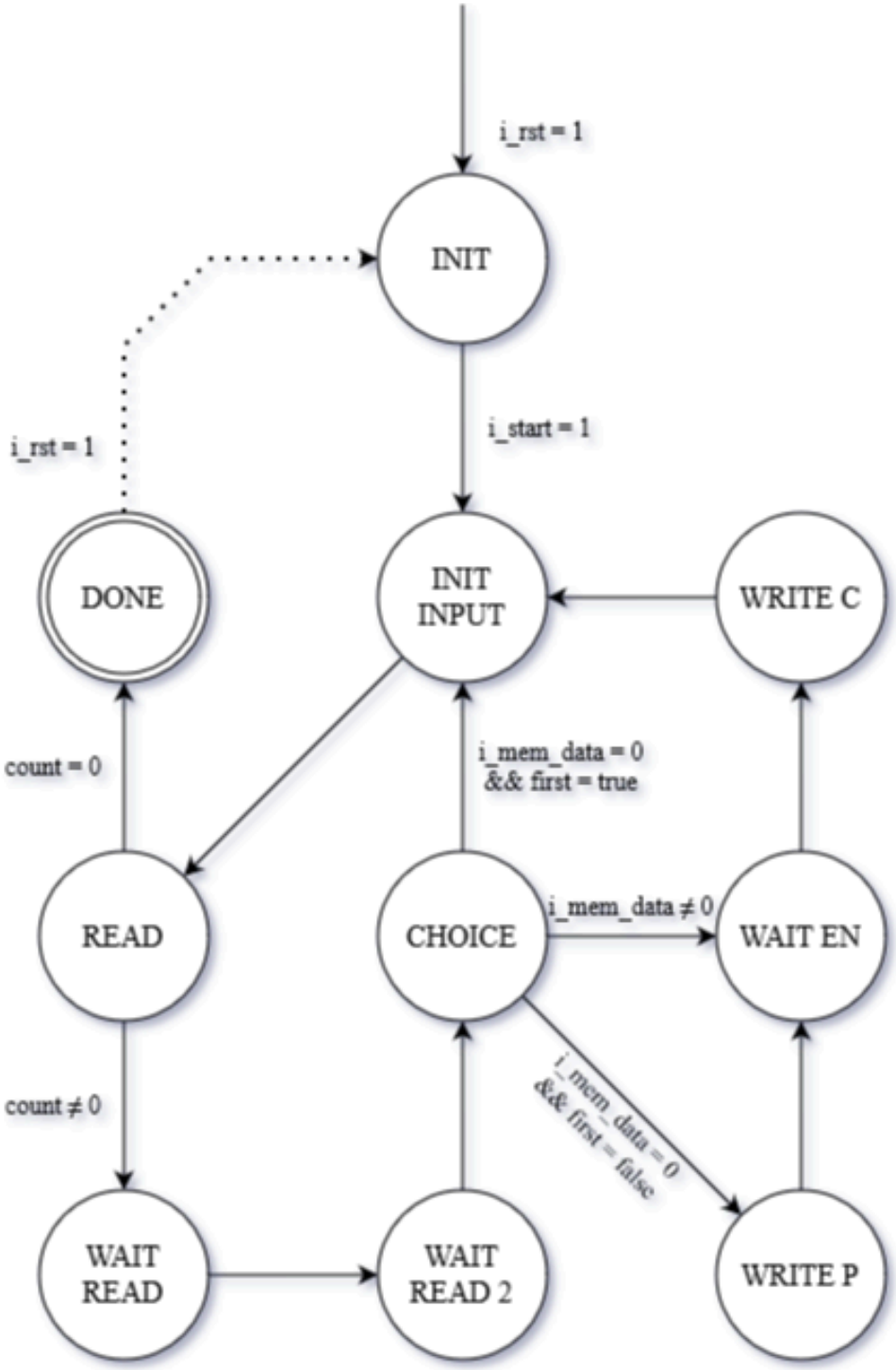
2.1 Segnali

- **curr_state**: segnale che indica lo stato corrente, si appoggia ad una *type enumeration* che comprende tutti i 10 stati della FSM;
- **next_state**: segnale che indica lo stato su cui spostarsi al prossimo ciclo di clock;
- **address**: segnale che indica l'indirizzo di memoria su cui si opera, all'inizio dell'algoritmo viene inizializzato prendendo il valore di **i_add** ed è rappresentato da un *integer*;
- **count**: segnale che indica il numero di parole da leggere, viene man mano decrementato ed è rappresentato da un *integer*;
- **previous**: segnale che memorizza l'ultima parola significativa letta che verrà eventualmente scritta in memoria, rappresentata da un *integer*;
- **credibility**: segnale che tiene nota del numero da scrivere nei byte vuoti tra una parola e l'altra, viene inizializzato a 31 all'inizio dell'algoritmo e viene decrementato o ripristinato al valore iniziale in base all'andamento della sequenza, rappresentato da un *integer*;
- **first**: segnale che indica se l'algoritmo per elaborare la sequenza è iniziato o se al momento sono stati letti solo zeri, è rappresentato da un *boolean*.

Oltre a questi appena elencati, sono presenti ulteriori segnali che si chiamano **[segnale]_next**. Essi passano il proprio valore al segnale originale ad ogni ciclo di clock, utilizzando la logica dei *flip-flop* ed evitando così la formazione di *latch*.

2.2 Macchina a stati

La macchina a stati implementata nel componente è composta da 10 stati:
INIT, INIT_INPUT, READ, DONE, WAIT_READ, WAIT_READ_2, CHOICE, WRITE_P, WAIT_EN, WRITE C e DONE.



Di seguito una breve descrizione del funzionamento di ogni stato:

- **INIT**: stato di partenza in cui ci si sposta ogni volta che *i_rst* viene alzato. Quando *i_start* è alto, vengono inizializzati *address_next* e *count_next* con i valori di *i_add* e *i_k*, *first_next* viene posto uguale true e *credibility_next* uguale a 31. Lo segue INIT_INPUT;
- **INIT_INPUT**: stato di passaggio che permette l'aggiornarsi dei valori dai segnali next ai segnali originali, lo segue READ;
- **READ**: stato che esegue operazioni diverse in base al valore di *count*. Se il segnale che conta le parole è uguale a zero significa che è terminata la sequenza su cui lavorare e quindi imposta *next_state* su DONE. In caso contrario regola i registri per leggere dalla memoria all'indirizzo address_next e passa poi a WAIT_READ;
- **WAIT_READ**, **WAIT_READ2**: stati di passaggio consecutivi che permettono il passaggio dei valori dalla memoria ai segnali *next* e poi a quelli originali, segue lo stato CHOICE;
- **CHOICE**: stato che decrementa *count_next* ed esegue operazioni diverse in base ai valori di alcuni segnali. Se *i_mem_data* è uguale a zero e *first* è *true*, allora incrementa di due *addres_next* e ritorna a INIT_INPUT siccome la macchina sta leggendo zeri dall'inizio e ancora non è partito l'algoritmo, se invece *first* è *false*, passa a WRITE_P. Nel caso in cui *i_mem_data* sia diverso zero, allora si pone *first_next* a *false* perché significa che è stato letto il primo byte diverso da zero, si salva *i_mem_data* su *previous_next*, si pone *credibility* uguale a 31 e si passa a WAIT_EN.
- **WRITE_P**: stato in cui si predispongono i segnali a memoria, vengono alzati gli appositi registri dell'interfaccia, si passano i valori di *address* e *previous* a *o_mem_addr_next* e *o_mem_data_next*, in tal modo nei prossimi cicli di clock verrà rimpiazzata una parola uguale a zero con l'ultima parola significativa letta.
- **WAIT_EN**: stato di passaggio che permette la scrittura in memoria, inoltre pone *first_next* a false;
- **WRITE_C**: stato che predispone i registri per scrivere in memoria il valore della credibilità sui byte vuoti. Incrementa *o_mem_addr_next*, assegna *credibility* a *o_mem_data_next*, alza *o_mem_we_next* e *o_mem_en_next*, decrementa *credibility* se è diverso da zero e infine torna a INIT_INPUT;

- **DONE**: stato “finale”, se *i_start* è alto allora alza *o_done_next* e non tocca *next_state*, altrimenti azzera tutti i segnali e passa a INIT_INPUT.

3 Risultati sperimentali

3.1 Sintesi

Una volta completato il modulo, dopo averlo testato in diversi contesti e casi limite, abbiamo analizzato il report del tool di sintesi.
Il primo passo è stato accertarsi che non fossero presenti *latch* utilizzando il comando *report_utilization* nella console TCL.

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	84	0	8000	1.05
LUT as Logic	84	0	8000	1.05
LUT as Memory	0	0	5000	0.00
Slice Registers	70	0	16000	0.44
Register as Flip Flop	70	0	16000	0.44
Register as Latch	0	0	16000	0.00
F7 Muxes	0	0	7300	0.00
F8 Muxes	0	0	3650	0.00

I risultati sono stati positivi, la tabella mostra la totale assenza di *latch* e l’utilizzo di 84 *look up tables* e 70 *flip flop*.
Il secondo passo è stato accertarsi che i requisiti di tempo venissero soddisfatti utilizzando il comando *report_timing*.

```
Timing Report

Slack (MET) :             16.329ns  (required time - arrival time)
```

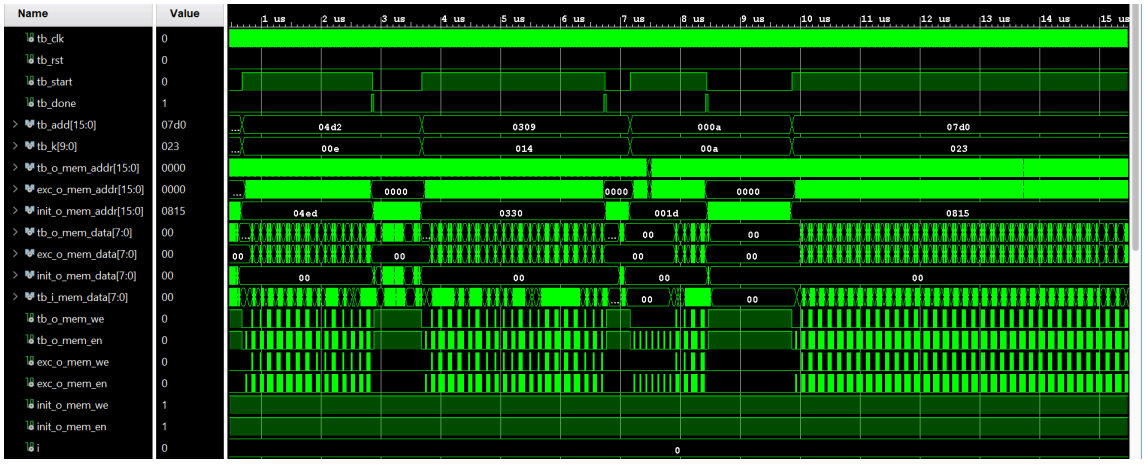
Come si può vedere, risulta Slack (MET), infatti il tempo riportato è 16ns, ben inferiore ai 20ns di limite.

3.2 Simulazioni

Per testare il modulo nel migliore dei modi, abbiamo fatto uso di diversi *test bench*, alcuni li abbiamo scritti personalmente, altri li abbiamo ottenuti online, da studenti che li hanno condivisi. Tutti i casi limite comunque sono stati testati con test scritti di prima mano per avere una maggiore sicurezza. Di seguito illustriamo due simulazioni *post-synthesis* che affrontano quattro casi particolari diversi.

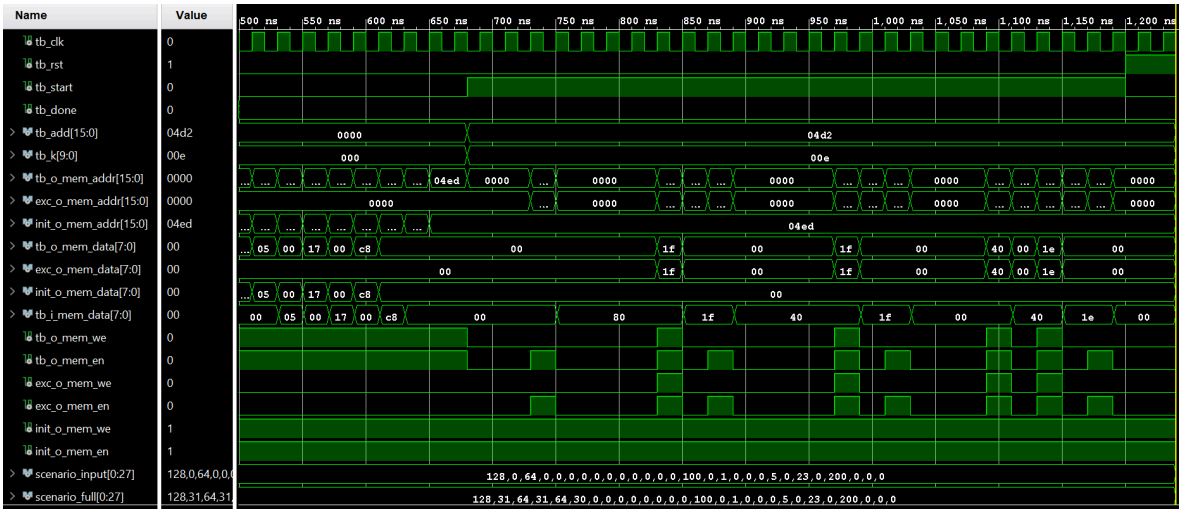
3.2.1 Multi start con credibility partenza di zeri

In questa simulazione abbiamo testato il corretto funzionamento del modulo quando deve operare su quattro sequenze diverse.
Siccome servivano più, abbiamo deciso di inserire alcuni corner case direttamente qui dentro, tra i quali una sequenza di parole non significative di una lunghezza maggiore di 31 e una sequenza presenta una serie di zeri in partenza.
Come si può vedere il segnale di start viene alzato quattro volte e il modulo ha completato le sue operazioni su tutte le sequenze, senza bloccarsi e soddisfando tutte le assert del tb.



3.2.2 Reset asincrono

In questa simulazione ci siamo assicurati che il modulo rispondesse bene al segnale di reset, interrompesse le operazioni sulla sequenza corrente e non modificasse i restanti byte.



4. Conclusioni

Il componente sviluppato risulta funzionare correttamente e supera egregiamente tutti i test a cui è stato sottoposto nei tre diversi tipi di simulazioni: **Behavioral**, **Post-synthesis Functional** e **Post-synthesis Timing**.
Non sono state implementate particolari ottimizzazioni nel corso del progetto, l'unica in una fase iniziale in cui leggevamo prima i dati e poi controllavamo se avessimo sforato con la lunghezza della sequenza, abbiamo successivamente deciso di spostare quel controllo in uno stato precedente.