POLITECNICO
MILANO 1863

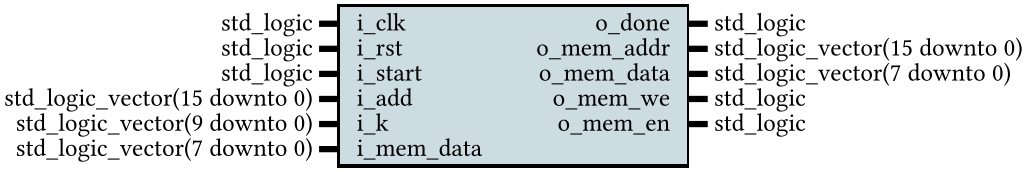
SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Prova finale: Reti Logiche 2023/2024

Angelo Prete
angelo2.prete@mail.polimi.it
10767149

1. Introduzione

Il componente realizzato elabora una sequenza di dati presenti in una memoria RAM sostituendo alle celle con valore 0, interpretabili come valori assenti, l'ultimo dato letto con valore valido con credibilità opportunamente decrementata e assegnando invece alle celle con valore diverso da zero la credibilità massima.



Rappresentazione grafica dell'interfaccia del componente

Un esempio di possibile applicazione è la correzione di letture di sensori. Le letture assenti sono spesso segnalate memorizzando il valore 0 in memoria.

Nei paragrafi successivi è fornita sia una descrizione dei segnali necessari per il corretto funzionamento, sia una descrizione in maggior dettaglio del componente stesso.

a. Collegamento a memoria RAM

Il componente deve essere collegato a una memoria RAM che ha interfaccia

```
entity ram is
    port
    (
        clk  : in std_logic;
        we   : in std_logic;
        en   : in std_logic;
        addr : in std_logic_vector(15 downto 0);
        di   : in std_logic_vector(7 downto 0);
        do   : out std_logic_vector(7 downto 0)
    );
end ram;
```

Sono specificate, nella tabella seguente, le corrispondenze tra segnali della memoria RAM e del componente

	RAM (segnale)	Componente (segnale)	Dimensione
Enable	`en`	`o_mem_en`	1 bit
Write enable	`we`	`o_mem_we`	1 bit
Address	`addr`	`o_mem_addr`	16 bit
Data in	`di`	`i_mem_data`	8 bit
Data out	`do`	`o_mem_data`	8 bit

Bisogna ricordare, infine, che RAM e componente devono condividere lo stesso segnale di clock.

b. Collegamento all'utilizzatore

L'utilizzatore dovrà fornire al componente i seguenti segnali:

- **`i_clk`**: segnale di clock
- **`i_rst`**: reset asincrono del componente
- **`i_start`**: segnale di avvio
- **`i_add`**: indirizzo iniziale
- **`i_k`**: numero di dati da processare (k parole seguite ognuna dalla propria credibilità)

Il termine della computazione, invece, sarà segnalato dal componente all'utilizzatore tramite **`o_done`**.

c. Descrizione funzionamento

È possibile individuare tre fasi principali durante il funzionamento del componente:

1. **Inizializzazione**:
   Sono forniti in input l'indirizzo iniziale, il numero di parole da processare e il segnale di start. Questa fase deve seguire un reset asincrono o la corretta terminazione di un'esecuzione precedente.
2. **Aggiornamento**:
   Il modulo processa i dati in memoria, aggiornandoli e associando loro un valore di credibilità, come descritto in dettaglio dal seguente pseudocodice

   input: indirizzo di partenza a, numero di iterazioni k
   $a_f \leftarrow a + 2 * k$ (indirizzo finale, escluso)
   $d_l \leftarrow 0$ (ultimo dato letto diverso da 0)
   $c_l \leftarrow 0$ (ultima credibilità)
   RAM (memoria RAM rappresentata come vettore)
   while $a \neq a_f$ **do**
   $d \leftarrow \text{RAM}[a]$
   if $d \neq 0$ **then**
   $d_l \leftarrow d$
   $\text{RAM}[a + 1] \leftarrow 31$
   $c_l \leftarrow 31$
   else
   $\text{RAM}[a] \leftarrow d_l$
   $c_l \leftarrow \max((c_l - 1), 0)$
   $\text{RAM}[a + 1] \leftarrow c_l$

end

$a \leftarrow a + 2$

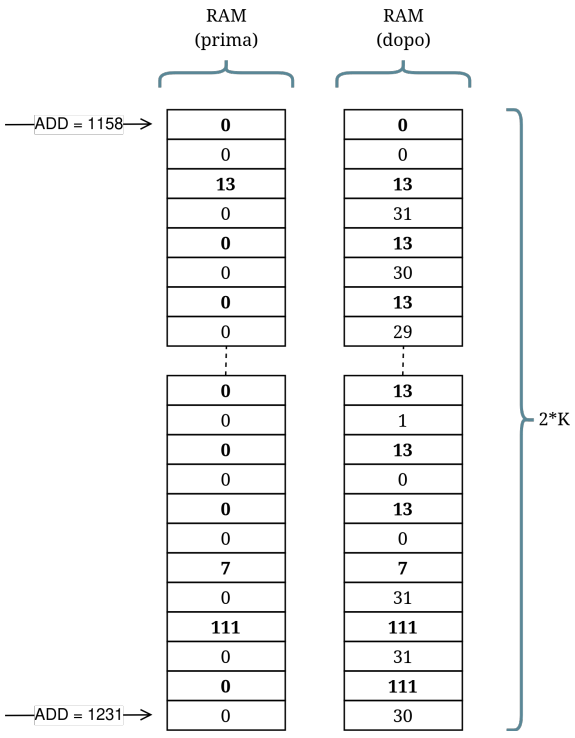
end

3. **Terminazione**:
   Il componente segnala la fine dell'esecuzione ponendo il segnale `o_done` alto e rimanendo in attesa di osservare che il segnale `i_start` diventi 0. Infine, il segnale `o_done` viene posto a 0 e la computazione è da considerarsi terminata.

d. Esempio di funzionamento

In questa sezione viene presentato il risultato di una computazione in termini di aggiornamento della memoria. È inoltre introdotto uno dei possibili edge-case: l'inizio di una sequenza con dato zero.

Siano dati in ingresso `i_k`= 37, `i_add`= 1158 e sia la situazione iniziale e finale della memoria RAM quella rappresentata in figura (i dati memorizzati sono evidenziati in grassetto)



Il primo dato letto ha valore zero. Non avendo letto alcun dato valido precedentemente, il valore rimane inalterato e la credibilità assegnata sarà 0.

Il dato successivo è 13, essendo non nullo riceve credibilità massima (31) e dato e credibilità vengono memorizzati come ultimi validi.

Segue una serie di dati nulli. Avendo salvato 13 come ultimo dato valido, vengono sovrascritti con un valore diverso da 0 e viene associato ad ognuno di loro credibilità decrementata rispetto al precedente.

Quando la credibilità è stata già decrementata a 0, se il dato nella cella successiva è nullo riceverà credibilità nulla, non potendola decrementare ulteriormente.

Dopo, è presente un dato diverso da zero, 111, che riceve quindi credibilità 31. Il dato successivo è invece pari a 0 e quindi diventa 111 e riceve credibilità decrementata (30) rispetto all'ultima memorizzata.

e. Osservazioni

Si noti che, nonostante le porzioni di memoria che ospitano la credibilità abbiano valore 0 (sia in questo esempio sia in tutti i test forniti con la specifica), non si tratta del caso generale.

È quindi necessario, qualora non si abbia la certezza che sia già presente uno zero, sovrascrivere il valore di credibilità anche se il valore da assegnare è pari a 0 stesso.

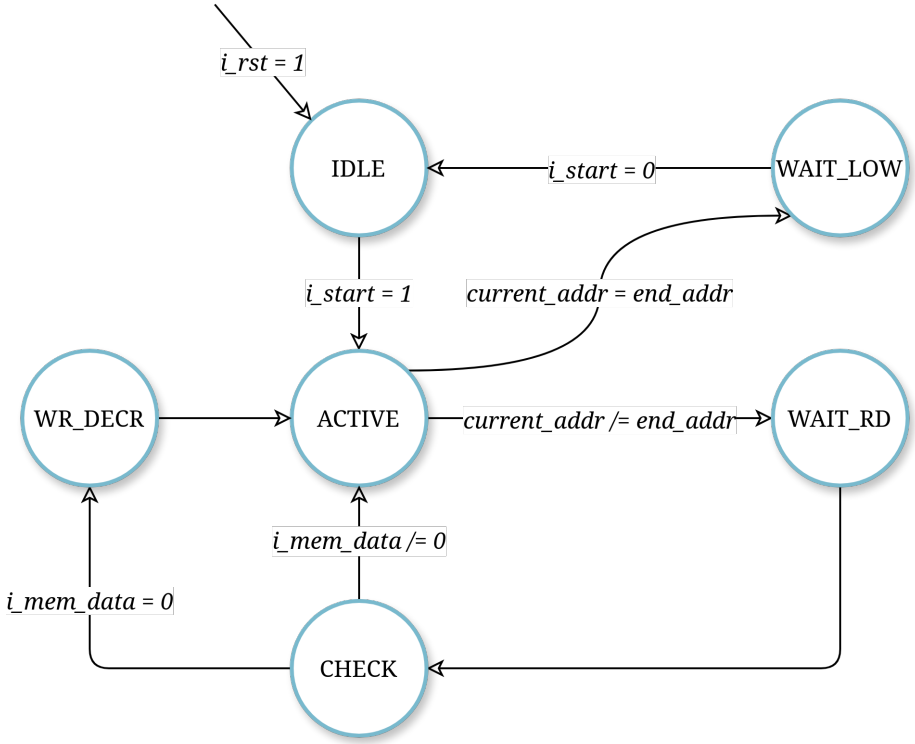
La lettura della generica cella di memoria che ospiterà la credibilità al fine di verificare che abbia valore nullo, seppur possibile, è un'operazione più costosa (un ciclo di clock in più) rispetto alla semplice sovrascrittura.

2. Architettura

Data la semplicità del componente, non si è ritenuto necessario dividerlo in più entities. Il risultato finale è quindi una singola entity, la cui architettura realizza una macchina a stati mediante due processi.

a. Macchina a stati finiti (entity project_reti_logiche)

La macchina a stati finiti dell'architecture implementata è una macchina di Mealy. Internamente, le transizioni della FSM avvengono sul fronte di salita del clock. È composta da 6 stati, sono quindi necessari 3 Flip Flop per memorizzare lo stato corrente.



Rappresentazione ad alto livello della FSM implementata

Nome stato implementato	*Abbreviazione*
`STATE_IDLE`	`IDLE`
`STATE_ACTIVE`	`ACTIVE`
`STATE_WAIT_START_LOW`	`WAIT_LOW`
`STATE_WAIT_WORD_READ`	`WAIT_RD`
`STATE_ZERO_WORD_CHECK_AND_WRITE`	`CHK_ZERO`
`STATE_WRITE_DECREMENTED_CRED`	`WR_DECR`

Tabella di mapping dei nomi degli stati tra rappresentazione grafica e implementazione

i. Stati

In questa sezione è presentata una descrizione delle funzioni svolte dal componente in ogni stato:

- **`STATE_IDLE`**
  La FSM si trova in questo stato quando è in attesa di iniziare una nuova computazione (sta as-

pettando che `i_start` diventi pari a 1). È possibile arrivare in `STATE_IDLE` sia a seguito del reset asincrono sia a seguito della fine di una computazione.

- **`STATE_ACTIVE`**
  In questo stato viene deciso se bisogna processare un nuovo indirizzo di memoria oppure porre temine alla computazione. Se l'indirizzo corrente è da processare, si preparano i segnali di memoria per leggere il dato memorizzato allo stesso indirizzo, altrimenti ci si sposta nello stato `STATE_WAIT_START_LOW`.
- **`STATE_WAIT_START_LOW`**
  Si arriva in questo stato quando non ci sono indirizzi da processare rimanenti; la FSM lo segnala all'utilizzatore ponendo il segnale `o_done` alto e aspetta che `i_start` diventi 0, tornando subito dopo nello stato di `STATE_IDLE`.
- **`STATE_WAIT_WORD_READ`**
  Questo stato serve per permettere alla memoria di fornire il dato richiesto negli stati precedenti; infatti, come stabilito nella specifica, la memoria ha un delay di 2 nanosecondi in lettura, solo al termine dei quali può fornire il dato richiesto.
- **`STATE_ZERO_WORD_CHECK_AND_WRITE`**
  Il dato è finalmente disponibile: se è uguale a 0 bisogna sovrascriverlo (comunicandolo opportunamente alla RAM) con l'ultimo dato diverso da 0 e spostarsi nello stato `STATE_WRITE_DECREMENTED_CRED`; altrimenti, all'indirizzo successivo della RAM viene salvato il valore massimo di credibilità (31), tornando poi nello stato `STATE_ACTIVE`.
- **`STATE_WRITE_DECREMENTED_CRED`**
  Si è in questo stato a seguito della lettura un valore pari a 0 in memoria nello stato `STATE_ZERO_WORD_CHECK_AND_WRITE`. Viene scritto in RAM un valore di credibilità decrementato rispetto al precedente (o 0 se l'ultima credibilità era già pari a 0 stesso). Si torna quindi in `STATE_ACTIVE`.

ii. Segnali

La macchina a stati usa diversi segnali interni e per ognuno è presente un segnale con lo stesso nome concatenato alla parola *next*. La scelta progettuale di avere segnali "doppi" è stata fatta nell'ottica di avere solo Flip Flop come registri e nessun Latch.

Sono qui elencati i vari segnali con una brevissima descrizione:

- Segnali per la gestione dello stato FSM, dove `state_t` è un tipo che definisce gli stati possibili
  - **`current_state`** : `state_t`
  - **`next_state`** : `state_t`
- Memorizzazione dell'indirizzo corrente (segnale da 16 bit)
  - **`current_address`** : `std_logic_vector(15 downto 0)`
  - **`current_address_next`** : `std_logic_vector(15 downto 0)`
- Memorizzazione dell'indirizzo finale (anche questo a 16 bit)
  - **`end_address`** : `std_logic_vector(15 downto 0)`
  - **`end_address_next`** : `std_logic_vector(15 downto 0)`
- Ultima parola letta diversa da zero (o zero stesso a seguito del reset asincrono; 8 bit)
  - **`last_word`** : `std_logic_vector(7 downto 0)`
  - **`last_word_next`** : `std_logic_vector(7 downto 0)`

- Credibilità corrente (pari a zero a seguito del reset asincrono, con valore massimo 31 e minimo 0; salvata in 8 bit anche se è possibile memorizzarla in 5)
  - **`last_credibility`** : `std_logic_vector(7 downto 0)`
  - **`last_credibility_next`** : `std_logic_vector(7 downto 0)`

Sono state inoltre dichiarate costanti per evitare ripetizioni ed essere più espliciti nel codice:

- **`zero_word`** e **`zero_credibility`** con valore zero
- **`max_credibility`** con valore 31 (credibilità massima)

iii. Processi

L'architecture della FSM contiene due processi

1. Reset asincrono e clock

Il primo svolge due funzioni:

1. **Gestione del reset asincrono**
   Quando il segnale `i_rst` è alto, al registro contenente lo stato corrente viene assegnato lo stato `STATE_IDLE`.
2. **Transizioni**
   In presenza del fronte di salita del segnale `i_clk` ai segnali contententi i valori correnti vengono assegnati i valori dei segnali *next*. Questa operazione aggiorna anche lo stato corrente che, essendo presente nella sensitivity list del secondo processo, lo "attiva".

2. Delta/Lambda

Il secondo processo realizza le funzioni δ e λ della FSM di Mealy.

Si occupa quindi di stabilire quale sarà il prossimo stato, il valore dei segnali interni e quali valori fornire in output (sia verso la memoria, sia verso l'utilizzatore del modulo).

3. Risultati sperimentali

a. Sintesi

A seguito del processo di sintesi (con target **xa7a12tcpg238-2I**), otteniamo i seguenti dati:

	Used	Fixed	Available	Util%
Slice LUTs*	78	0	134600	0.06
LUT as Logic	78	0	134600	0.06
LUT as Memory	0	0	46200	0.00
Slice Registers	51	0	269200	0.02
Register as Flip Flop	51	0	269200	0.02
Register as Latch	0	0	269200	0.00
F7 Muxes	0	0	67300	0.00
F8 Muxes	0	0	33650	0.00

Notiamo che il componente usa:

- **78 Look-Up Table** (0.06%)
- **51 Flip Flop** (0.02%), tutti e soli i previsti. Nell'implementazione del componente viene salvato l'indirizzo di fine per controllare se ci sono indirizzi rimanenti: una scelta alternativa, che avrebbe permesso di ridurre ulteriormente il numero di Flip Flop, è quella di salvare k e decrementarlo.
- **0 Latches**, risultato ottenuto grazie ad opportune scelte progettuali descritte precedentemente.

La percentuale di occupazione degli elementi disponibili è molto bassa: la logica implementata è molto semplice e non necessita di ampi spazi di memoria o complesse operazioni.

Un altro dato rilevante è la durata del percorso critico, inferiore ai **5 ns**. Il componente può quindi lavorare a periodi di clock inferiori rispetto ai 20 ns con cui è stato testato.

b. Simulazioni

Il componente è stato sottoposto a testbech scritti a mano per verificare il comportamento in presenza di diversi edge-case e a testbench generati automaticamente per controllare il corretto funzionamento su input prevedibili ma di lunghezza molto maggiore, al fine di fare benchmarks sulla velocità di esecuzione.

Sono qui descritti i primi in dettaglio.

i. Testbench fornito ed esempi da specifica

Il primo testbench ad essere stato provato è quello presente nei materiali per il progetto, funziona correttamente e rispetta i vincoli di clock.

Inoltre, sono stati scritti test per controllare che il componente funzioni correttamente anche con tutti gli altri esempi forniti nella specifica.

ii. Start multipli

Questo testbench è stato scritto per verificare il corretto funzionamento del componente in esecuzioni successive senza reset intermedi.

Ciò ha permesso di controllare che i segnali interni vengano correttamente ripristinati quando inizia una nuova esecuzione (`i_start` pari a 1).

iii. Dato iniziale nullo

Si è testato il corretto funzionamento in condizioni simili a quelle dell'esempio di funzionamento (Section d.).

iv. Credibilità nulla

Con questo testbench si è voluto testare che la credibilità, qualora sia stata decrementata fino a zero, rimanga pari a zero se i nuovi dati letti hanno valore nullo.

v. Reset durante la computazione

Grazie a questo test si è mostrato il funzionamento del componente quando il segnale di reset viene portato alto durante una computazione. Il test si è rivelato molto utile perché ha evidenziato problemi dovuti all'errata inizializzazione dei segnali.

vi. Reset durante accesso a memoria

Come nel testbench precedente, si è verificato il funzionamento a seguito di reset, questa volta durante letture e scrittura della memoria. Il fine di questo test era assicurarsi che il valore dei segnali per il controllo della RAM fosse corretto dopo aver ricevuto `i_rst`= 0.

4. Conclusioni

a. Implementazione e testing

Il componente, oltre a rispettare la specifica, è stato implementato in modo efficiente dal punto di vista temporale. Infatti, è stata posta particolare attenzione a ridurre il numero di stati e la durata delle computazioni, senza sacrificare allo stesso tempo la leggibilità del codice.

Oltre a funzionare nelle simulazioni Behavioral e Post-Synthesis Functional, il componente ha il comportamento richiesto anche quando viene testato in simulazioni Post-Synthesis Timing.

b. Possibili miglioramenti per future implementazioni

Come già anticipato, un possibile miglioramento per ridurre l'uso di Flip Flop è quello di cambiare la verifica di fine della computazione, cambiamento che però andrebbe ad aumentare l'uso di Look-up Table per svolgere l'operazione di decremento del k (memorizzato al posto di `last_address`, utilizzando quindi 6 registri in meno). Questo permetterebbe anche di eliminare il segnale `end_address_next` a favore di un segnale `k_next`, risparmiando altri 6 registri.

Un altro miglioramento che ridurrebbe il numero di registri è la memorizzazione della credibilità attraverso segnali di tipo `std_logic_vector(5 downto 0)` invece di quelli a 8 bit utilizzati nell'implementazione attuale. Questa modifica permetterebbe di ridurre ulteriormente l'uso di registri.

c. Tooling

Tra i tanti strumenti utilizzati per il progetto, una menzione speciale va a *ghdl* per aver reso piacevole lo sviluppo del componente e al framework *VUnit* per aver facilitato il processo di testing, permettendo di automatizzare l'esecuzione pre-synthesis dei testbench.