POLITECNICO
MILANO 1863

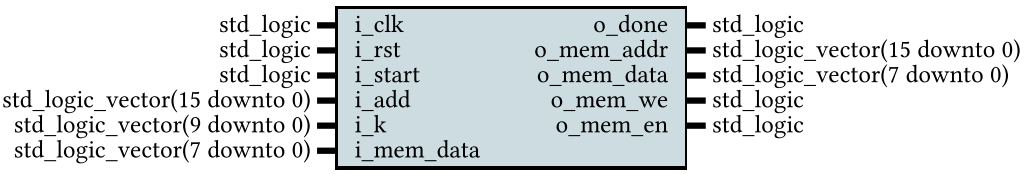
SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

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 con valore valido, con credibilità opportunamente decrementata.



Rappresentazione grafica dell'interfaccia del componente

Un esempio di applicazione è la correzione di letture assenti di sensori, solitamente segnalate tramite il valore 0.

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

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;
```

Specifichiamo, nella tabella seguente, le corrispondeze 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 bits
Data in	di	i_mem_data	8 bits
Data out	do	o_mem_data	8 bits

Ricordiamo infine che RAM e componente devono condividere lo stesso segnale di clock.

b. Collegamento all'utilizzatore

L'utilizzatore del componente qui specificato dovrà fornire come segnali di ingresso:

- **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

Per segnalare la fine di una computazione, il componente fa uso del segnale **o_done**.

c. Descrizione funzionamento

Possiamo descrivere il funzionamento dividendolo in tre fasi:

1. **Inizializzazione**: vengono forniti in input l'indirizzo iniziale, il numero di coppie (valore + credibilità) di celle da processare e un segnale di start; questa fase segue un eventuale reset o il termine di un'esecuzione precedente.
2. **Aggiornamento**: Il modulo inizia a processare i dati in memoria, aggiornandoli come descritto 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
  $a \leftarrow a + 2$

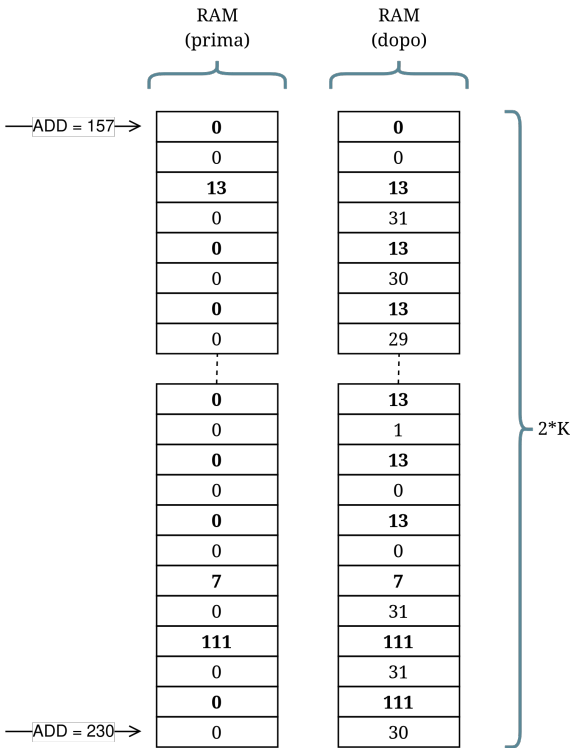
end

3. **Terminazione**: la fine della fase di aggiornamento è seguita da una segnalazione da parte del componente: *o_done* viene posto alto e si rimane in attesa di osservare basso il segnale *i_start*.

d. Esempio funzionamento

In questa sezione mostriamo il risultato di una computazione, introducendo anche uno dei possibili edge-cases, ovvero l'inizio di una sequenza con uno zero.

Siano dati in ingresso i_k = 37 e i_add = 157; sia la situazione iniziale e finale della memoria RAM quella rappresentata in figura (i dati memorizzati sono stati evidenziati in grassetto per distinguerli dalla credibilità):



Il primo dato letto è uno zero e, non avendo letto nessun altro dato prima il suo valore rimane inalterato e la sua credibilità viene posta a zero.

Il dato successivo è pari a 13, è quindi non nullo e possiamo assegnarli credibilità massima (31).

Troviamo ora una serie di dati pari a 0, ma avendo come ultimo dato salvato 13 possiamo riscriverlo, associandolo ogni volta a una credibilità decrementa rispetto l'ultima utilizzata.

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

Infine incontriamo un nuovo dato diverso da 0, 111, e gli assegnamo credibilità 31. Il dato successivo è pari a 0 e quindi, come fatto precedentemente, lo sovrascriviamo con 111 (l'ultimo dato valido) e gli assegnamo credibilità decrementata pari a 30.

e. Osservazioni

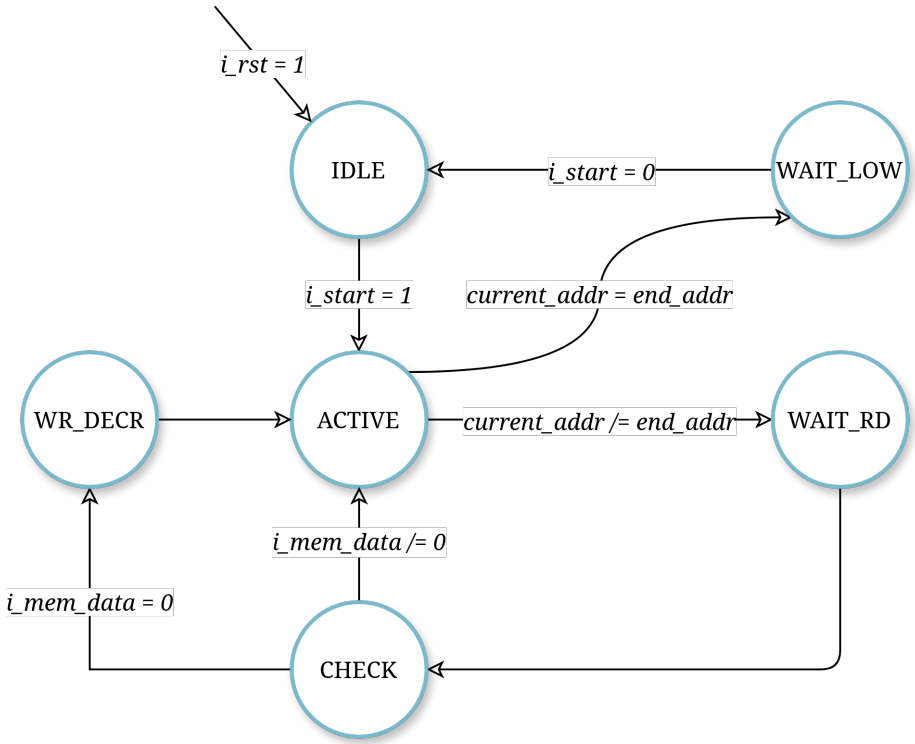
Notiamo che, nonostante le celle che ospitano la credibilità abbiano valore 0 sia in questo esempio sia in tutti i test forniti nella specifica, non è il caso generale. È quindi necessario, qualora non si abbia la certezza che sia presente uno 0, sovrascrivere il valore di credibilità anche se il valore da assegnare è pari a 0. Controllare la cella che ospiterà la credibilità aggiornata per verificare che abbia valore nullo, seppur possibile, è un'operazione più costosta (un ciclo di clock in più) della semplice sovrascrittura.

2. Architettura

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

Ogni stato ha uno specifico compito:

- **STATE_IDLE**:
  Quando la FSM è in attesa di iniziare una nuova computazione, si trova in questo stato. È possibile arrivare qui a seguito del reset asincrono o della fine di una computazione.
- **STATE_ACTIVE**:
  In questo stato la FSM decide se processare una nuova coppia di indirizzi di memoria, a seguito di un controllo sull'indirizzo da processare, oppure teminare la computazione. Se l'indirizzo da processare non è l'ultimo, si preparano i segnali di memoria per leggere il dato all'indirizzo corrente.
- **STATE_WAIT_START_LOW**:
  Arriviamo in questo stato quando gli indirizzi da processare sono finiti, la macchina segnala questo ponendo il segnale o_done alto e aspetta che i_start venga abbassato a 0, evento seguito dal ritorno nello stato di idle.
- **STATE_WAIT_WORD_READ**:
  Questo stato serve per permettere alla memoria di fornire il dato richiesto negli stati precedenti;

infatti, da specifica, la memoria ha un delay di 2 nanosecondi 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 di scrittura della credibilità decrementata, altrimenti, scriviamo nell'indirizzo successivo in RAM il massimo valore di credibilità (31) e torniamo nello stato active.
- **STATE_WRITE_DECREMENTED_CRED**:
  Siamo in questo stato se abbiamo letto un valore pari a 0 in memoria. Scriviamo in memoria quindi un valore di credibilità decrementato rispetto al precedente (o 0 se l'ultima credibilità era già pari a 0 stesso).

)

i. Processo 1: Reset asincrono e clock

Il primo processo della FSM ha due funzioni:

- **Gestione del reset asincrono**: quando il segnale i_rst è alto, al registro contenente lo stato corrente viene assegnato lo stato di IDLE.
- **Transizioni**: se siamo sul fronte di salita del segnale i_clk allora i registri contententi i valori correnti vengo assegnati i nuovi valori. Questa operazione aggiorna anche lo stato corrente che, essendo presente nella sensitivity list del Processo 2, lo "sveglia".

ii. Processo 2: Delta/Lambda

Questo processo corrisponde alle funzioni δ e λ della FSM di Mealy. Si occupa quindi di stabilire quale sarà il prossimo stato 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:

- **51 flip flop** (0.02%), tutti e soli i previsti
- **78 look-up tables** (0.06%)
- **0 latches**, risultato ottenuto grazie ad opportune scelte progettuali

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.

b. Simulazioni

Il componente è stato sottoposto sia testbeches scritti a mano per verificare il suo comportamento nei vari in edge-cases, sia a testbenches generati automaticamente per controllare il corretto funzionamento su vari range di memoria.

i. Testbench ufficiale

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

iii. Dato iniziale nullo

È stato necessario verificare che il componente funzionasse correttamente in condizioni simili a quelle dell'esempuo di funzionamento (Section d.)

iv. 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.

v. Reset durante accesso a memoria

Come nel testbench precedente, si è verificato il funzionamento a seguito di reset, questa volta durante una lettura e poi una scrittura in memoria.

4. Conclusioni

Il componente, oltre a rispettare la specifica, è stato implementato in modo efficiente. È stata posta particolare attenzione a ridurre il numero di stati, senza sacrificare allo stesso tempo la leggibilità del codice.