DOCUMENTAZIONE PER LA PROVA FINALE DI RETI LOGICHE DELL'AA 2017/2018

INTRODUZIONE:

Il seguente documento si compone di diverse sezioni atte a descrivere le fasi che ci hanno portato all'implementazione di un componente hardware descritto in VHDL per la Prova Finale di Reti Logiche dell'AA 2017/2018. In particolare, il documento si divide in:

- Composizione del gruppo: contiene codice persona e numero di matricola dei componenti del gruppo.
- Progettazione: contiene l'analisi dei requisiti progettuali e la scelta dell'approccio adottato per la realizzazione del componente.
- Implementazione: contiene l'analisi del codice contenuto nel file 10523105.vhd
- Testing: contiene l'elenco dei test effettuati per validare il componente e le motivazioni per cui sono stati scelti tali test.
- Ottimizzazione: contiene una descrizione delle modifiche effettuate al componente per migliorarne le prestazioni.

COMPOSIZIONE DEL GRUPPO:

- Giorgio Labate: codice persona: 10523105 matricola: 844588
- Daniele Mattioli: codice persona: 10502925 matricola: 846049

PROGETTAZIONE:

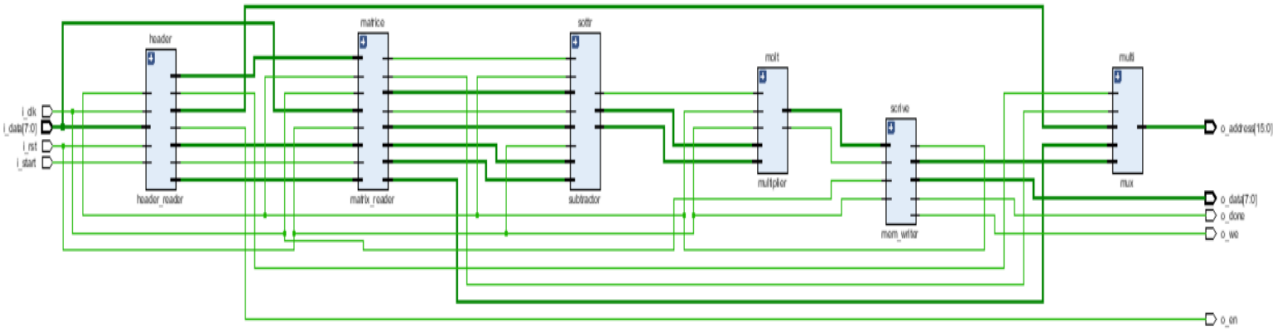
In questa sezione descriviamo l'analisi della specifica e la conseguente messa a punto di un algoritmo in grado di soddisfare le specifiche forniteci.

Dovendo calcolare l'area del rettangolo minimo che circoscrive totalmente una figura di interesse presente in un'immagine (rappresentata in memoria come una sequenza di byte, ognuno dei quali rappresenta un pixel), il componente deve essere in grado calcolare la base e l'altezza di tale rettangolo. In particolare, per calcolare la base abbiamo scelto di utilizzare due estremi per memorizzare l'indice della prima e dell'ultima riga della matrice che contengono occorrenze utili, ovvero valori maggiori o uguali alla soglia (un pixel è considerato parte della figura di interesse solo se il suo valore è maggiore o uguale al valore di soglia); analogamente, per calcolare il valore dell'altezza, abbiamo scelto di utilizzare due estremi per memorizzare l'indice della prima e dell'ultima colonna contenenti occorrenze utili. Si nota dunque la necessità di due indici, di riga e di colonna, che permettano di identificare univocamente ogni elemento della matrice. Infine, la differenza tra i valori degli estremi più uno, fornisce la base e l'altezza del rettangolo di interesse.
Abbiamo deciso di optare per un approccio modulare e quindi di dividere il componente in vari moduli, per aumentarne la comprensibilità e facilitarne la manipolazione. Ogni modulo è dunque in grado di svolgere una determinata funzione. In particolare, il componente è costituito dai seguenti moduli:

- **Header_reader**, per la lettura dalla RAM dell'header dell'immagine.
- **Matrix_reader**, per la lettura dalla RAM dell'immagine.
- **Subtractor**, per il calcolo della base e dell'altezza del rettangolo.
- **Multiplier**, per il calcolo dell'area del rettangolo.
- **Mem_writer**, per la scrittura in RAM dell'area del rettangolo.
- **Mux**, per la corretta gestione degli indirizzi della RAM.
- **Project_reti_logiche**, per collegare tra loro i vari moduli.

IMPLEMENTAZIONE:

In questa sezione analizziamo nel dettaglio i vari moduli, facendo riferimento al loro codice VHDL.



Schematico del componente (Elaborated Design)

Occorre sottolineare che tutti i moduli, ad eccezione del mux e di project_reti_logiche, hanno una rappresentazione comportamentale. Questa scelta è dovuta al fatto che ognuno dei moduli sopra elencati deve eseguire il proprio codice più volte, in particolare ogniqualvolta si verifica una variazione di uno dei segnali presenti nella lista di sensitività. Per il mux abbiamo invece optato per una rappresentazione dataflow non essendo necessario temporizzarlo in alcun modo. Project_reti_logiche infine, dovendo legare tra loro i vari blocchi, ha una rappresentazione strutturale.

Header_reader è un modulo che legge il contenuto della RAM agli indirizzi 2,3,4, ovvero l'header dell'immagine, memorizzando questi valori in tre segnali: *columns* (numero delle colonne dell'immagine), *raws* (numero delle righe dell'immagine), *threshold* (soglia dell'immagine). Questi segnali d'uscita vanno in ingresso al secondo blocco, matrix_reader, che inizia la computazione solo quando il segnale *read_matrix*, uscita di header_reader ed ingresso di matrix_reader, vale 1. L'esecuzione di header_reader invece è attivata solo quando viene alzato il segnale di start (dato come ingresso al modulo) alzando internamente il segnale *read_header*.
Il secondo modulo è dunque **matrix_reader** che, grazie agli ingressi ricevuti dal modulo precedente, analizza l'immagine presente nella RAM. In particolare, il modulo legge, ad ogni ciclo di clock, un byte della memoria, confrontando il valore contenuto nella cella *i_data* con il valore della soglia *threshold*. Se *i_data* >= *threshold* vengono aggiornati adeguatamente quattro segnali

rappresentanti gli estremi (indici di prima e ultima colonna e riga: *first_column_int*, *last_column_int, first_raw_int, last_raw_int)* che verranno forniti in uscita (*first_column, last_column, first_raw, last_raw*) al modulo subtractor per calcolare la base e l'altezza del rettangolo di area minima che circoscrive la figura di interesse. L'aggiornamento degli estremi utilizza due segnali *r_index* e *c_index* che rappresentano, rispettivamente, l'indice di riga e l'indice di colonna, numerati a partire da 0, della matrice che rappresenta l'immagine. Nel momento in cui si trova la prima occorrenza valida, gli estremi delle righe assumono il valore di *r_index* mentre quelli delle colonne assumono il valore di *c_index*. Vengono inoltre alzati due segnali, *first_occ* e *first_occurency*, che serviranno al modulo subtractor. Per quanto concerne invece le occorrenze successive alla prima, l'estremo delle righe *first_raw_int* non viene più aggiornato, l'estremo delle righe *last_raw_int* viene sempre aggiornato al valore dell'indice di riga *r_index* in cui vi è l'occorrenza valida, l'estremo delle colonne *first_column_int* viene aggiornato solo se l'indice di colonna in cui vi è l'occorrenza valida è minore del vecchio valore di *first_column_int*, l'estremo delle colonne *last_column_int* viene aggiornato solo se l'indice di colonna in cui vi è l'occorrenza valida è maggiore del vecchio valore di *last_column_int*. Terminata la lettura dell'immagine, le uscite *first_raw, first_column, last_raw* e *last_column* assumono il valore dei segnali intermedi *first_raw_int, first_column_int, last_raw_int* e *last_column_int (*questi segnali intermedi insieme a *firs_occ* sono necessari anche se equivalenti a quelli di uscita in quanto questi ultimi non possono essere letti). Viene inoltre alzato il segnale *calc_ready*, che viene utilizzato dal modulo successivo, il subtractor, per iniziare la sua computazione. Dal momento che i valori dei segnali aggiornati sono visibili solo al termine dell'esecuzione del processo, abbiamo introdotto il segnale *final_update* per ritardare l'aggiornamento finale dei quattro estremi, in modo da far fronte al caso in cui l'ultimo byte dell'immagine sia un'occorrenza valida: in tal caso altrimenti le uscite verrebbero assegnate con i valori non ancora aggiornati.

Il **subtractor** riceve in ingresso il valore degli estremi calcolati dal blocco precedente e calcola il valore dell' altezza(*width)* e della base(*length*) del rettangolo sottraendo al valore dell'ultimo indice *(last_raw e last_column*) quello del primo indice (*first_raw e first_column*) e sommando 1. Tramite il segnale *first*_occurency viene gestito il caso in cui nella matrice non vi sia alcuna occorrenza valida. In tal caso infatti sottrarre gli estremi e sommare 1 porterebbe ad avere un valore di *length* e *width* pari a 1, che ovviamente non è corretto. Dunque, se *first*_occurency=0, i valori di *length* e *width* vengono settati a 0. Una volta eseguito il calcolo, il subtractor alza un segnale *calc_area_ready*, che viene utilizzato dal modulo successivo, il multiplier, per eseguire il calcolo dell'area.

Multiplier è quindi un blocco che riceve in ingresso, oltre che il segnale *calc_area_ready*, anche il valore della base (*length)* e dell'altezza *(width)* del rettangolo. Viene quindi calcolato il valore dell'area *(area)* e viene alzato il segnale *can_write*, usato dal modulo successivo per iniziare la sua computazione.

Il blocco **mem_writer** quindi, nel momento in cui *can_write*=1 inizia la sua esecuzione e scrive ai byte 1 e 0 della RAM rispettivamente la parte più significativa e meno significativa del valore dell'area. Viene poi alzato per un ciclo di clock il segnale *o_done*. Il modulo mem_writer ha inoltre un segnale d'uscita *end_computation* che va in ingresso ai moduli precedentemente descritti e che viene settato ad 1 per un ciclo di clock quando *o_done* si alza. Se *end_computation*=1 i vari moduli reinizializzano i loro segnali interni e le loro uscite necessarie per il loro corretto funzionamento: questo segnale funge da reset implicito al termine della computazione.

Il **mux** viene usato invece per selezionare correttamente il valore di *o_address*. In particolare, il mux prende in ingresso 3 segnali *in_address_h, in_address_m* ed *in_address_w*, provenienti rispettivamente dai moduli header_reader, matrix_reader e mem_writer e, attraverso due bit di controllo *control_1* e *control_2 (*provenienti *rispettivamente* da *header_reader* e da *matrix_reader)*, seleziona il corretto valore dell'indirizzo. Durante la lettura dell'header, i valori di *control_1* e *control_2* sono entrambi a 0 e quindi l'indirizzo selezionato è quello del blocco header_reader. Terminata la lettura dell'header, *control_1* viene portato a 1 e *o_address* assume il valore di *in_address_m* (indirizzo del blocco matrix_reader), inizializzato a 6 e poi incrementato ad ogni ciclo per leggere i valori contenuti in memoria. Una volta finita di leggere l'immagine, *control_2* viene portato a 1 e quindi l'indirizzo selezionato dal mux è quello del modulo mem_writer.

In **project_reti_logiche** infine si collegano i vari moduli tra loro.

Considerazioni Generali:

La gestione del reset è asincrona rispetto al clock e quindi, in qualunque istante della computazione arrivi, i vari moduli vengono reinizializzati. Dal momento che *end_computation* funge da reset implicito, anche la sua gestione è asincrona rispetto al clock (in particolare si osserva che viene trattata nello stesso if del reset). Il segnale *end_computation* serve per prepararsi a ricevere un nuovo start dopo la prima computazione anche nel caso in cui non vi sia un reset esplicito nel testbench. Lo start viene invece gestito dal modulo header_reader in modo sincrono.

Dalla specifica si evince che il segnale di write enable *o_we* debba essere abbassato una volta terminata la scrittura in memoria. Immaginando una situazione in cui vi sia una condivisione della RAM, abbiamo deciso di abbassare, tramite il reset implicito, anche il segnale di enable *o_en* una volta terminata la scrittura del risultato.

È inoltre importante osservare che, in generale, quando l'address viene incrementato a n su un fronte di salita del clock, il valore i_data visto dal componente **sul fronte di salita del clock** è quello presente all'address n-2, in quanto la memoria 'vede' l'address n-2 solo sul fronte di salita precedente. Per questo motivo, ad esempio, nel modulo header_reader quando si aggiorna l'*address* a 4 e viene visto ancora l'address del ciclo precendente, cioè 3, si entra nell' if condizionato da "address = 3" e il valore in i_data è, sul fronte,

quello all'address 2 di memoria, cioè il numero di colonne (si mette per questo correttamente "columns <= unsigned(i_data)"). Lo stesso discorso è estendibile agli altri moduli.

In vari moduli si nota infine la presenza di segnali interni (come *first_raw_int, first_column_int, last_raw_int, last_column_int*, *first_occ*) che abbiamo utilizzato nei vari if. Non potevamo usare i rispettivi segnali di uscita (*first_raw, first_column, last_raw, last_column, first_occureny*) perché, essendo uscite, non possono essere lette.

TESTING:

In questa sezione descriviamo i test effettuati per validare il nostro componente.

Oltre ai quattro testbench fornitici su Beep, abbiamo provato i seguenti test:

- Numero delle righe o numero delle colonne uguale a 0. In questo caso volevamo verificare il corretto funzionamento del segnale *first_occurency*, ovvero volevamo verificare che nel caso in cui la matrice fosse nulla, il valore di *length* e *width* venisse settato a 0. Occorre inoltre sottolineare che *first_occurency* viene utilizzato correttamente anche nel caso in cui il numero delle righe e delle colonne sia diverso da 0, ma la matrice non contiene occorrenze valide.
- Matrice con una sola occorrenza valida al byte 5. Con questo test volevamo verificare che, pur avendo i valori dei quattro estremi *first_raw, first_column, last_raw, last_column* a 0, il risultato dell'area fosse 1. Grazie al valore alto assunto da *first_occurency*, il modulo subtractor non setta a 0 i valori di *length* e *width* ma assegna ai due segnali i valori corretti, ovvero 1.
- Matrice con una occorrenza valida all'ultimo byte. Con questo test volevamo verificare il corretto aggiornamento dei quattro estremi, anche nel caso in cui fosse l'ultimo byte dell'immagine a contenere un'occorrenza valida. Volevamo quindi verificare il corretto funzionamento del segnale *final_update* di matrix_reader.
- Test in cui le uniche due occorrenze valide sono al primo ed all'ultimo byte dell'immagine. In questo caso volevamo verificare il corretto funzionamento nel caso in cui l'area del rettangolo minimo coincidesse con l'area dell'intera matrice nel caso estremo di due sole occorrenze valide.
- Test in cui la matrice ha dimensione minima 1x1.
- Test in cui la matrice ha dimensione massima 255x255.
- Test in cui durante la computazione viene dato un reset. In questo caso volevamo verificare il corretto funzionamento del segnale di reset.
- Test in cui, dopo la prima computazione, vengono ridati un reset ed uno start. In questo caso volevamo verificare il corretto funzionamento in caso di una seconda computazione richiesta. Il test è stato anche

provato nel caso limite in cui nel momento in cui il reset si abbassa, lo start si alza.

```
    test : process is
    begin
    wait for 100 ns;
    wait for c_CLOCK_PERIOD;
    tb_rst <= '1';
    wait for c_CLOCK_PERIOD;
    tb_rst <= '0';
    wait for c_CLOCK_PERIOD;
    tb_start <= '1';
    wait for c_CLOCK_PERIOD;
    tb_start <= '0';
    wait until tb_done = '1';
    wait until tb_done = '0';
    wait until rising_edge(tb_clk);
    wait for c_CLOCK_PERIOD;
    tb_rst <= '1';
    wait for c_CLOCK_PERIOD;
    tb_rst <= '0';
    wait for c_CLOCK_PERIOD;
    tb_start <= '1';
    wait for c_CLOCK_PERIOD;
    tb_start <= '0';
    wait until tb_done = '1';
    wait until tb_done = '0';
    wait until rising_edge(tb_clk);
    assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
    assert RAM(0) = "01101110" report "FAIL low bits" severity failure;
    assert false report "Simulation Ended!, test passed" severity failure;
    end process test;
    end projecttb;
```

```
    test : process is
    begin
    wait for 100 ns;
    wait for c_CLOCK_PERIOD;
    tb_rst <= '1';
    wait for c_CLOCK_PERIOD;
    tb_rst <= '0';
    wait for c_CLOCK_PERIOD;
    tb_start <= '1';
    wait for c_CLOCK_PERIOD;
    tb_start <= '0';
    wait until tb_done = '1';
    wait for c_CLOCK_PERIOD;
    tb_rst <= '1';
    wait for c_CLOCK_PERIOD;
    tb_rst <= '0';
    tb_start <= '1';
    wait for c_CLOCK_PERIOD;
    tb_start <= '0';
    wait until tb_done = '1';
    wait until tb_done = '0';
    wait until rising_edge(tb_clk);
    assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
    assert RAM(0) = "01101110" report "FAIL low bits" severity failure;
    assert false report "Simulation Ended!, test passed" severity failure;
    end process test;
```

- Test in cui, dopo la prima computazione, viene ridato uno start. In questo caso volevamo verificare la correttezza della seconda computazione anche nel caso in cui questa non sia preceduta da un reset. Abbiamo quindi avuto modo di verificare come il segnale *end_computation* funga da reset implicito nel momento in cui termina il calcolo dell'area del rettangolo.

```
test : process is
begin
wait for 100 ns;
wait for c_CLOCK_PERIOD;
tb_rst <= '1';
wait for c_CLOCK_PERIOD;
tb_rst <= '0';
wait for c_CLOCK_PERIOD;
tb_start <= '1';
wait for c_CLOCK_PERIOD;
tb_start <= '0';
wait until tb_done = '1';
wait until tb_done = '0';
wait until rising_edge(tb_clk);
wait for c_CLOCK_PERIOD;
tb_start <= '1';
wait for c_CLOCK_PERIOD;
tb_start <= '0';
wait until tb_done = '1';
wait until tb_done = '0';
wait until rising_edge(tb_clk);
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "01101110" report "FAIL low bits" severity failure;
assert false report "Simulation Ended!, test passed" severity failure;
end process test;
```

- Test in cui, dopo il primo reset, vengono alzati lo start e, dopo pochi istanti (nello stesso ciclo di clock), il reset. In questo caso volevamo verificare il corretto funzionamento anche nel caso in cui, su un fronte di salita, sia il reset che lo start sono alti. La simulazione funziona correttamente dato che nel codice gli if del clock e del reset sono esclusivi: se non lo fossero stati, sarebbero stati eseguiti sia l'if del reset che l'if del clock (e quindi anche l'if dello start) erroneamente, dato che il reset è arrivato dopo lo start. La computazione partirà quindi solo con il secondo start.

```
test : process is
begin
wait for 100 ns;
wait for c_CLOCK_PERIOD;
tb_rst <= '1';
wait for c_CLOCK_PERIOD;
tb_rst <= '0';
wait for c_CLOCK_PERIOD;
tb_start <= '1';
wait for 2 ns;
tb_rst <= '1';
wait for 13 ns;
tb_start <= '0';
wait for 2 ns;
tb_rst <= '0';
wait for c_CLOCK_PERIOD;
tb_start <= '1';
wait for c_CLOCK_PERIOD;
tb_start <= '0';
wait until tb_done = '1';
wait until tb_done = '0';
wait until rising_edge(tb_clk);
wait until rising_edge(tb_clk);
assert RAM(1) = "00000000" report "FAIL high bits" severity failure;
assert RAM(0) = "01101110" report "FAIL low bits" severity failure;
assert false report "Simulation Ended!, test passed" severity failure;
end process test;

end projecttb;
```

Abbiamo simulato tutti i test sia in pre sintesi che in post sintesi, ottenendo risultati corretti.

OTTIMIZZAZIONE:

L'idea generale è stata quella di non cercare di ridurre in modo particolare l'area occupata dal componente o di aumentare la sua frequenza, ma di ottenere un componente che desse buoni risultati in entrambi i casi, senza sbilanciarsi ne' in un senso ne' nell'altro. Alla fine, abbiamo però osservato che i moduli subtractor e multiplier, che inizialmente eseguivano la loro computazione temporizzati con il clock (come si vede dalle immagini che rappresentano il codice precedente a quello definitivo), potevano essere lasciati asincroni, a differenza degli altri tre blocchi che devono invece interagire con la memoria (header_reader, matrix_reader e mem_writer). Abbiamo quindi modificato il nostro codice in questo senso per migliorare i risultati del componente in termini di frequenza (i warning sui latch sono dovuti a questa modifica).

In particolare, i risultati ottenuti per l'area sono:

LUT: 196

FF: 187

IO: 38

BUFG: 1

Valutando invece i risultati di timing, abbiamo verificato che il componente viene implementato raggiungendo un periodo minimo di circa 3.9 ns.

Alleghiamo per completezza il codice precedente, a quello caricato su Beep, del moltiplicatore e del sottrattore (quando ancora erano temporizzati con il clock).

```
end Behavioral;
entity subtractor is
    Port (
          calc_ready : in STD_LOGIC; --segnale che serve a condizionare l'esecuzione di subtractor. Viene alzato da matrix_reader
          first_raw : in  unsigned(7 downto 0);
          first_column : in  unsigned(7 downto 0);
          last_raw : in  unsigned(7 downto 0);
          last_column : in  unsigned(7 downto 0);
          i_clk : in std_logic;
          i_rst: in std_logic;
          end_computation: in std_logic;--segnale di reset implicito
          first_occurency : in std_logic ;  --segnale che indica a subtractor se è stata trovata almeno una occorrenza valida (valore sopra la threshol
          calc_area_ready : out STD_LOGIC;  --segnale che indica a multiplier, blocco successivo, di attivarsi.
          length : out unsigned (7 downto 0); --base
          width : out unsigned (7 downto 0)); --altezza
end subtractor;

architecture Behavioral of subtractor is
signal subtraction_done : std_logic := '0'; --segnale interno che indica a subtractor di terminare la sua esecuzione una volta eseguita la sottrazione

begin
process(i_clk, i_rst, end_computation)
begin
    if  i_rst = '1' or end_computation = '1' then
        calc_area_ready <= '0';
        subtraction_done <= '0';
    elsif i_clk' event and i_clk= '1' then
        if calc_ready = '1' and subtraction_done = '0' then
             if first_occurency = '0' then --caso particolare: l'area è 0 perchè ho trovato zero occorrenze utili (oppure la matrice aveva 0 righe o 0
                          width <= (others => '0');
                          length <= (others => '0');
             else
                          width <=  last_raw - first_raw +1 ; --devo aggiungere 1 per comprendere entrambi gli estremi
                          length <=  last_column - first_column +1 ;  --devo aggiungere 1 per comprendere entrambi gli estremi
             end if;
        calc_area_ready <= '1'; --indico a multiplier, blocco successivo, di attivarsi.
        subtraction_done <= '1'; --serve a interrompere l'esecuzione di subtractor: arrivati a questo punto ha terminato il suo compito.

        end if;
    end if;
end process;

end Behavioral;
```

Subtractor

Multiplier

```vhdl
entity multiplier is
    Port (
           i_clk : in STD_LOGIC;
           i_rst : in STD_LOGIC;
           calc_area_ready : in STD_LOGIC; -- segnale che serve a condizionare l'esecuzione di multipler. Viene alzato da subtractor
           length : in UNSIGNED (7 downto 0);
           width : in UNSIGNED (7 downto 0);
           end_computation: in std_logic; --segnale di reset implicito
           area : out UNSIGNED (15 downto 0);
           can_write : out STD_LOGIC  --segnale che indica a mem_writer, blocco successivo, di attivarsi.
           );
end multiplier;

architecture Behavioral of multiplier is
signal multiplication_done : std_logic := '0'; --segnale interno che indica a multiplier di terminare la sua esecuzione una volta eseguita la moltipli
begin
    process (i_clk, i_rst, end_computation)
    begin
        if  i_rst = '1' or end_computation = '1' then
            can_write <= '0';
            multiplication_done <= '0';
        elsif i_clk' event and i_clk = '1' then
            if calc_area_ready = '1' and multiplication_done = '0' then
              area <= length * width;
              multiplication_done <= '1';  --serve a interrompere l'esecuzione di multiplier: arrivati a questo punto ha terminato il suo compito.
              can_write <= '1'; --indico a mem_writer, blocco successivo, di attivarsi.
            end if;
        end if;
     end process;

end Behavioral;
```