POLITECNICO
MILANO 1863

Prova finale di reti logiche

Studente: Riccardo Pazzi
Matricola: 890980
Codice persona: 10571638

Introduzione:

La prova finale di RL dell'anno accademico 2020/2021 consiste nel descrivere un componente in grado di equalizzare un immagine immagazzinata in memoria, e scrivere l'immagine output nelle celle adiacenti all'immagine input.
La descrizione è stata fatta mediante VHDL, in particolare utilizzando un approccio behavioral.

Specifica:

Descrizione generale

Il metodo di equalizzazione dell'istogramma di una immagine è un metodo pensato per ricalibrare il contrasto di una immagine quando l'intervallo dei valori di intensità sono molto vicini effettuando una distribuzione su tutto l'intervallo di intensità, al fine di incrementare il contrasto.
Nella versione da sviluppare non è richiesta l'implementazione dell'algoritmo standard ma di una sua versione semplificata.
L'algoritmo di equalizzazione sarà applicato solo ad immagini in scala di grigi a 256 livelli e deve trasformare ogni suo pixel nel modo seguente:

```
DELTA_VALUE = MAX_PIXEL_VALUE – MIN_PIXEL_VALUE
SHIFT_LEVEL = (8 – FLOOR(LOG2(DELTA_VALUE +1)))
TEMP_PIXEL = (CURRENT_PIXEL_VALUE - MIN_PIXEL_VALUE) << SHIFT_LEVEL
NEW_PIXEL_VALUE = MIN( 255 , TEMP_PIXEL)
```

Dove `MAX_PIXEL_VALUE` e `MIN_PIXEL_VALUE` , sono il massimo e minimo valore dei pixel dell'immagine, `CURRENT_PIXEL_VALUE` è il valore del pixel da trasformare, e `NEW_PIXEL_VALUE` è il valore del nuovo pixel.
Il modulo da implementare dovrà leggere l'immagine da una memoria in cui è memorizzata, sequenzialmente e riga per riga, l'immagine da elaborare. Ogni byte corrisponde ad un pixel dell'immagine.
La dimensione della immagine è definita da 2 byte, memorizzati a partire all'indirizzo 0. Il byte all'indirizzo 0 si riferisce alla dimensione di colonna; il byte nell'indirizzo 1 si riferisce alla dimensione di riga. La dimensione massima dell'immagine è 128x128 pixel. L'immagine è memorizzata a partire dall'indirizzo 2 e in byte contigui. Quindi il byte all'indirizzo 2 è il primo pixel della prima riga dell'immagine.

Dati

Le dimensioni dell'immagine , ciascuna di dimensione di 8 bit, sono memorizzati in una memoria con indirizzamento al Byte partendo dalla posizione 0: il byte in posizione 0 si riferisce al numero di colonne (N -COL), il byte in posizione 1 si riferisce al numero di righe (N-RIG).
I pixel del'immagine , ciascuno di un 8 bit, sono memorizzati in memoria con indirizzamento al Byte partendo dalla posizione 2.
I pixel della immagine equalizzata , ciascuno di un 8 bit, sono memorizzati in memoria con indirizzamento al Byte partendo dalla posizione 2+(N-COL*N-RIG) .

Note ulteriori sulla specifica

1. Si noti che nel modulo da implementare, FLOOR(LOG2(DELTA_VALUE +1)) è un numero intero con valori tra 0 e 8 facilmente ricavabile da controlli a soglia.

2. Si faccia attenzione al numero di bit necessari in ogni passaggio.

3. Il modulo deve essere progettato per poter codificare più immagini, ma l'immagine da codificare non verrà mai cambiata all'interno della stessa esecuzione, ossia prima che il modulo abbia segnalato il completamento tramite il segnale *DONE*. Si veda il prossimo punto per il protocollo di re-start.

4. Il modulo partirà nella elaborazione quando un segnale *START* in ingresso verrà portato a 1. Il segnale di *START* rimarrà alto fino a che il segnale di *DONE* non verrà portato alto; Al termine della computazione (e una volta scritto il risultato in memoria), il modulo da progettare deve alzare (portare a 1) il segnale *DONE* che notifica la fine dell'elaborazione. Il segnale *DONE* deve rimanere alto fino a che il segnale di *START* non è riportato a 0. Un nuovo segnale start non può essere dato fin tanto che *DONE* non è stato riportato a zero. Se a questo punto viene rialzato il segnale di *START*, il modulo dovrà ripartire con la fase di codifica.

5. Il modulo deve essere progettato considerando che prima della prima codifica verrà sempre dato il reset al modulo. Invece, come descritto nel protocollo precedente, una seconda elaborazione non dovrà attendere il reset del modulo.

Interfaccia del Componente

Il componente da descrivere deve avere 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_data : in std_logic_vector(7 downto 0);
            o_address : out std_logic_vector(15 downto 0);
            o_done : out std_logic;
            o_en : out std_logic;
            o_we : out std_logic;
            o_data : out std_logic_vector (7 downto 0)
      );
end project_reti_logiche;
```

In particolare:
● il nome del modulo deve essere project_reti_logiche
● i_clk è il segnale di CLOCK in ingresso generato dal TestBench;
● i_rst è il segnale di RESET che inizializza la macchina pronta per ricevere il primo segnale di START;
● i_start è il segnale di START generato dal Test Bench;
● i_data è il segnale (vettore) che arriva dalla memoria in seguito ad una richiesta di lettura;
● o_address è il segnale (vettore) di uscita che manda l'indirizzo alla memoria;
● o_done è il segnale di uscita che comunica la fine dell'elaborazione e il dato di uscita
scritto in memoria;
● o_en è il segnale di ENABLE da dover mandare alla memoria per poter comunicare
(sia in lettura che in scrittura);
● o_we è il segnale di WRITE ENABLE da dover mandare alla memoria (=1) per poter
scriverci. Per leggere da memoria esso deve essere 0;
● o_data è il segnale (vettore) di uscita dal componente verso la memoria.

Descrizione della memoria

NOTA : La memoria è già istanziata all'interno del Test Bench e non va sintetizzata
La memoria e il suo protocollo può essere estratto dalla seguente descrizione VHDL che fa parte del test bench e che è derivata dalla User guide di VIVADO disponibile al seguente link:
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_3/ug901-vivado-synthesis.pdf

```
-- Single-Port Block RAM Write-First Mode (recommended template)
--
-- File: rams_02.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity rams_sp_wf 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)
);
```

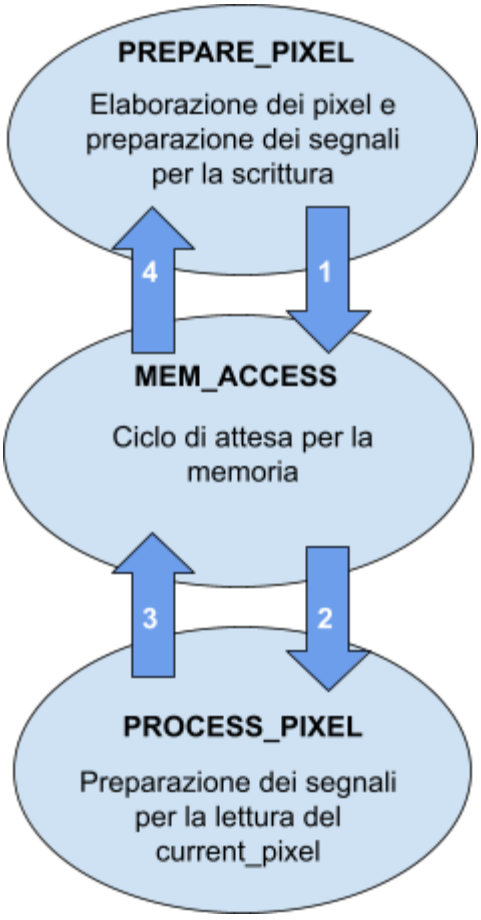
```
end rams_sp_wf;
architecture syn of rams_sp_wf is
type ram_type is array (65535 downto 0) of std_logic_vector(7 downto
0);
signal RAM : ram_type;
begin
process(clk)
begin
if clk'event and clk = '1' then
if en = '1' then
if we = '1' then
RAM(conv_integer(addr)) <= di;
do <= di after 2 ns;
else
do <= RAM(conv_integer(addr)) after 2 ns;
end if;
end if;
end if;
end process;
end syn;
```

Elaborazione specifica:

I primi passi che ho mosso per tradurre la specifica si sono concentrati sulla prima parte, che consiste nell'identificare `MAX_PIXEL_VALUE` e `MIN_PIXEL_VALUE`, poichè a monte di tutto il calcolo dei singoli pixel nell'immagine di output bisogna calcolare `DELTA_VALUE`.
Siccome i pixel dell'immagine non sono ordinati in alcun modo secondo il proprio valore 0-255 l'unica soluzione è quella di scorrere l'intera immagine in modo sequenziale e trovare i due valori estremi. Data la sequenzialità di questa operazione di scorrimento, poiché la memoria consente di accedere ad una sola cella alla volta, ho optato per la modellizzazione di una FSM.

1.Descrizione FSM ad alto livello dello stato FIND_MIN_MAX

Una volta trovati massimo e minimo possiamo applicare l'algoritmo fornito in specifica per calcolare i singoli pixel dell'immagine in output e scriverli in memoria, affrontiamo prima il problema del calcolo.
Per il calcolo di `NEW_PIXEL_VALUE` possiamo utilizzare una rete combinatoria, che riceve in ingresso `MIN_VALUE`, `MAX_VALUE` e `CURRENT_PIXEL_VALUE`. Così facendo evitiamo di sprecare cicli di clock aggiuntivi e calcoliamo in un solo ciclo il valore del pixel da scrivere. Il processo di calcolo sarà fatto chiaramente per ogni pixel dell'immagine, e la sequenzialità di lettura e scrittura sarà facilmente gestibile dalla nostra macchina a stati finiti.
Per la lettura e scrittura in memoria in particolare dobbiamo prestare attenzione poiché i process in VHDL aggiornano i segnali solo alla fine dell'esecuzione, ciò significa che non è possibile in un singolo stato eseguire più letture o scritture ma è necessario attendere che la memoria scriva i dati in i_data. Questo comporta l'**attesa di un ciclo di clock** affinché il dato richiesto o inviato sia letto/scritto.
Per ovviare a questo problema ci basta utilizzare uno stato di buffer MEM_ACCESS, il cui scopo è aspettare un ciclo di clock tra la richiesta di un dato e la sua lettura o l'invio di un dato e l'invio del dato successivo. In un contesto reale questo comportamento consente di evitare fenomeni di glitch nei segnali che rimangono stabili per un ciclo di clock.

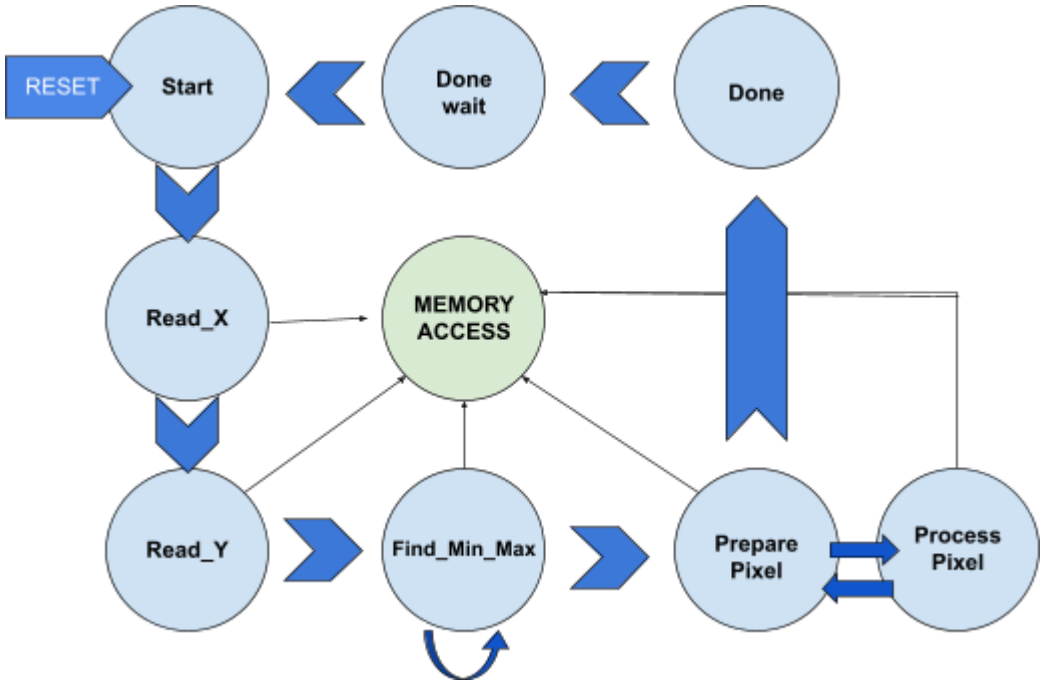


Possiamo modellizzare questo calcolo mediante tre stati, di cui uno servirà da buffer per la memoria, le transizioni saranno seguite in ordine numerico ed il modulo verrà eseguito subito dopo aver individuato minimo e massimo.

A questo punto si può procedere alla modellizzazione dell'intero componente, rimane solo da fare un breve appunto sui segnali di RESET e START che verrà affrontato successivamente.

2. Descrizione FSM ad alto livello degli stati PREPARE_PIXEL, MEM_ACCESS e PROCESS_PIXEL

Possiamo quindi utilizzare la seguente FSM composta da 9 stati:



Descrizione degli Stati:

1. **Start:** E' lo stato di partenza nel quale si trova la macchina dopo un reset, attende che RESET si abbassi e START = 1. Poi prepara la lettura della prima cella di memoria e imposta le variabili al loro valore iniziale.
2. **Read_X:** E' lo stato che legge la larghezza dell'immagine dalla prima (x0000) cella di memoria (come da specifica) e prepara la lettura della seconda
3. **Read_Y:** Legge l'altezza dell'immagine dalla seconda (x0001) cella di memoria (come da specifica) e calcola il prodotto X*Y, uguale al numero di celle occupate dall'immagine. Se questo risulta 0 salta direttamente allo stato di DONE.
4. **Find_Min_Max:** Scorre l'immagine e confronta ogni pixel con minimo e massimo correnti, eventualmente li aggiorna. Dopo aver letto X*Y celle si ferma e passa allo stato seguente.
5. **Prepare_Pixel:** Prepara la lettura dei pixel e dopo aver letto tutte le X*Y celle passa a **Done**.
6. **Process_Pixel:** Legge il pixel corrente, calcola il nuovo pixel equalizzato e prepara i segnali per la scrittura in memoria.
7. **Done:** Alza il segnale di *DONE*, che indica la fine dell'elaborazione.
8. **Done_Wait:** Abbassa *DONE* solo se *START* è tornato a 0, altrimenti attende. Una volta che *DONE* è a zero la macchina è pronta a ricevere un nuovo segnale di *START* ed elaborare l'immagine successiva.
9. **MEMORY ACCESS(nel codice MEM_ACCESS):** Questo stato fa da buffer di memoria come illustrato prima. Viene richiamato ogni qualvolta si vogliono aggiornare i segnali di *o_address* e *o_en/o_we.*

Nota sulla gestione del segnale di RESET:

Per gestire il segnale di RESET è stato utilizzato il codice seguente:

```
if(i_rst = '1') then -- Reset di stato e variabili
        --Riporto i segnali a valore iniziale
        o_en <= '0';
        o_we <= '0';
        o_done <= '0';
        --Riporto gli stati al valore iniziale
        P_STATE <= START;
        STATE <= START;
```

Si noti come siano riportati al loro stato originale i segnali, ma non le variabili. Queste vengono resettate all'interno dello stato di Start poiché questo reset va eseguito ad ogni esecuzione, mentre il segnale di RESET è alzato solo per la prima. In particolare **è importante resettare il minimo e massimo** affinché esecuzioni consecutive della macchina non abbiano effetti indesiderati.

Test Bench:

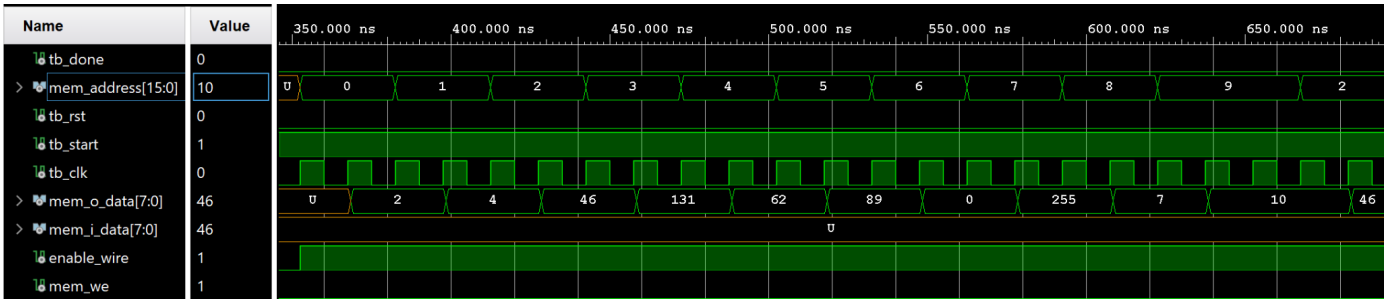
Ho scritto test bench personalizzati per testare casi limite (Immagine di dimensione 0, Immagini multiple consecutive, immagine già equalizzata, immagine a due valori che saturano, immagine con un valore singolo) che non hanno mostrato criticità in alcun caso.

In particolare i due di maggiore interesse sono l'immagine di dimensione 0, che in caso di gestione scorretta può portare allo stallo, e le esecuzioni multiple richieste da specifica.

Per gestire la dimensione 0 è stato aggiunto un controllo nello stato READ_Y, se infatti il prodotto risulta essere zero la FSM passerà immediatamente allo stato di DONE.

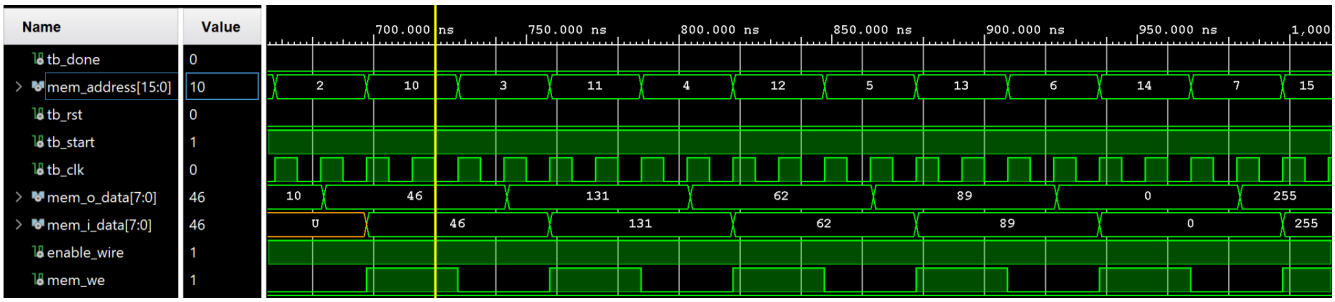
Invece per ciò che riguarda le esecuzioni multiple valgono le osservazioni fatte sul RESET.

Vediamo un esempio tratto dal test bench "immagine già equalizzata":



Come prima cosa la FSM scorre le celle 0 e 1 e legge le dimensioni (2,4) da mem_o_data dopodichè legge le 8 celle dell'immagine fino al mem_address 9.

Possiamo notare come l'immagine contenga i valori 0 e 255, questo vuol dire che è appunto già equalizzata, ci aspettiamo che vengano riscritti in memoria gli stessi valori letti.



E infatti come notiamo dalle forme d'onda successive la macchina legge da mem_o_data un valore e lo riscrive in memoria partendo dall'indirizzo 10 (immediatamente successivo al 9 che è l'ultimo dell'immagine).
La macchina presenta perciò il comportamento atteso.

Sintesi e Report:

La sintesi avviene senza alcun warning, qui sotto ho riportato i componenti usati.

```
Detailed RTL Component Info :
+---Adders :
         2 Input   16 Bit       Adders := 2
         3 Input    8 Bit       Adders := 1
+---Registers :
                   16 Bit    Registers := 3
                    8 Bit    Registers := 5
                    4 Bit    Registers := 2
                    1 Bit    Registers := 3
+---Muxes :
         2 Input   16 Bit        Muxes := 1
        10 Input   16 Bit        Muxes := 2
        10 Input    8 Bit        Muxes := 2
         2 Input    8 Bit        Muxes := 1
        10 Input    4 Bit        Muxes := 1
         2 Input    4 Bit        Muxes := 2
         7 Input    4 Bit        Muxes := 2
         2 Input    3 Bit        Muxes := 5
        10 Input    3 Bit        Muxes := 1
         2 Input    2 Bit        Muxes := 2
        10 Input    1 Bit        Muxes := 16
Report Cell Usage:
+------+-------+------+
|      |Cell   |Count |
+------+-------+------+
|1     |BUFG   |     1|
|2     |CARRY4 |    30|
|3     |LUT2   |    39|
|4     |LUT3   |    21|
|5     |LUT4   |    93|
|6     |LUT5   |    29|
|7     |LUT6   |   107|
```

```
|8     |FDCE   |    11|
|9     |FDRE   |    88|
|10    |FDSE   |     8|
|11    |IBUF   |    11|
|12    |OBUF   |    27|
+------+-------+------+
Synthesis finished with 0 errors, 0 critical warnings and 0
warnings.
```