Politecnico di Milano
Prova Finale della Laurea in Ingegneria Informatica

Progetto di Reti Logiche
AA 2020-21

Studenti

Bendinelli Marco
Codice Persona: 10673478
Matricola: 911063

Bosisio Andrea
Codice Persona: 10669287
Matricola: 911814

Supervisori

Prof. Fornaciari William

Terraneo Federico

POLITECNICO
MILANO 1863

Indice

1. Introduzione

La Prova Finale del corso di Reti Logiche[1] dell'AA 2020-21 consiste nella descrizione in VHDL e nella sintetizzazione di un componente hardware che implementi una versione semplificata dell'algoritmo di *equalizzazione dell'istogramma*[2].

1.1 Equalizzazione dell'istogramma

Questo metodo di elaborazione digitale consiste nella regolazione del *contrasto* dell'immagine utilizzando il suo *istogramma*, ottenuto contando il numero di pixel per ciascun livello di tono di grigio.
L'algoritmo standard[2], impiegato per questo tipo di equalizzazione, utilizza un approccio globale e altera i valori di livello tonale di ciascun pixel, calibrando così il contrasto dell'immagine e distribuendo uniformemente i livelli di grigio dell'istogramma.

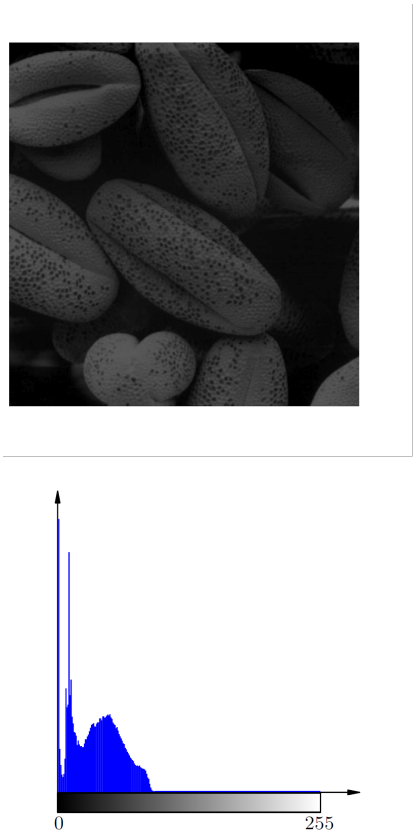

Figura 1.1: Immagine con relativo istogramma non equalizzato

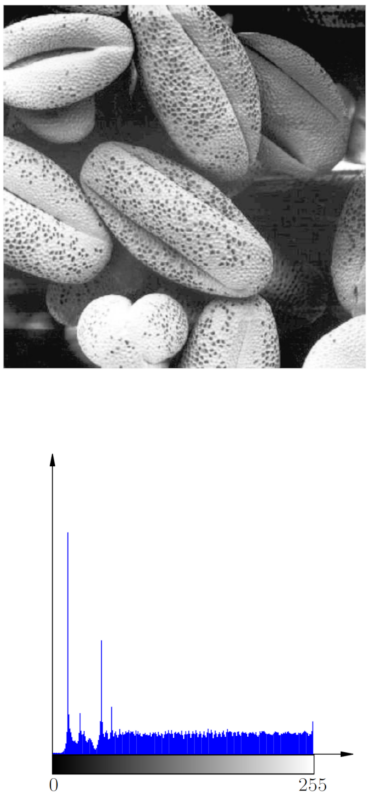

Figura 1.2: Immagine con relativo istogramma equalizzato

Fonte immagini: `http://tesi.cab.unipd.it/44161/1/Tesi.pdf`

[1] Maggiori informazioni riguardo la Prova Finale del corso
[2] `https://it.wikipedia.org/wiki/Equalizzazione_dell%27istogramma`

Dall'istogramma della fotografia mostrato in Figura 1.2, ottenuto dopo l'applicazione dell'algoritmo, si può notare come le sue componenti risultino maggiormente distribuite rispetto alle componenti dell'istogramma della Figura 1.1.
Il risultato ottenuto è un aumento del *contrasto*, inteso come differenza tra il valore minimo ed il valore massimo di tono di grigio. Inoltre, dopo l'applicazione dell'algoritmo, l'immagine risulta visivamente più nitida, evidenziando un maggior numero di dettagli.

1.2 Specifica del progetto

Nella versione richiesta, l'implementazione lavora in modo puntuale, cambiando il valore di ciascun pixel in base al solo valore massimo e minimo di intensità di grigio dell'immagine, composta al più da 128×128 pixel. In particolar modo, l'algoritmo segue le seguenti fasi:

1. calcolo della differenza `DELTA_VALUE` tra il valore massimo (`MAX_PIXEL_VALUE`) ed il valore minimo (`MIN_PIXEL_VALUE`) dei pixel dell'immagine;
2. calcolo del valore di *shift*[3] che dovrà essere applicato al singolo pixel e definito dalla seguente formula: `SHIFT_LEVEL = 8 - FLOOR(LOG2(DELTA_VALUE + 1))`;
3. calcolo di `TEMP_PIXEL`, definito come la differenza tra il valore tonale del pixel corrente `CURR_PIXEL_VALUE` e il valore minimo (`MIN_PIXEL_VALUE`) shiftata di `SHIFT_LEVEL` volte, ovvero:
   `TEMP_PIXEL = (CURR_PIXEL_VALUE - MIN_PIXEL_VALUE) « SHIFT_LEVEL;`
4. calcolo del pixel risultante `NEW_PIXEL_VALUE`, dato dal *minimo* tra `TEMP_PIXEL` e `255`, valore massimo della *scala di grigi* a 8 bit[4].

Indirizzo	Contenuto	
0	C	Dati immagine originale
1	R	
2	1° pixel	
3	2° pixel	
...	...	
C*R + 1	(C*R)° pixel	
C*R + 2	1° pixel processato	Risultato del processamento
C*R + 3	2° pixel processato	
...	...	
2(C*R) + 1	(C*R)° pixel processato	
...	...	
65535		

Figura 1.3: Rappresentazione della memoria dopo un'elaborazione

Il modulo implementato legge le informazioni necessarie da una memoria esterna con indirizzamento al byte e tale che:

- i primi due indirizzi contengano le informazioni che determinano la dimensione dell'immagine: il byte all'indirizzo 0 contiene il numero di colonne `C` dell'immagine ed il byte all'indirizzo 1 contiene il numero di righe `R`;
- dal terzo indirizzo in poi siano memorizzati i valori tonali dei singoli $\texttt{C} \times \texttt{R}$ pixel che possono variare da `0` a `255`, potendo quindi assumere 256 livelli di tonalità.

[3] https://docs.microsoft.com/it-it/cpp/cpp/left-shift-and-right-shift-operators-input-and-output?view=msvc-160
[4] https://gimp.linux.it/www/manual/2.10/html/it/gimp-image-convert-grayscale.html

Il componente deve essere progettato utilizzando l'interfaccia descritta nella figura 1.4.
La specifica impone inoltre le seguenti condizioni:

- l'inizio della prima elaborazione deve avvenire non prima del reset del modulo;
- il processamento dell'immagine comincia solo quando il segnale `i_start` viene posto a `'1'`. Questo segnale deve rimanere a `'1'` fino alla fine della elaborazione;
- il segnale `o_done` durante la computazione è tenuto a `'0'` e solo al termine viene portato a `'1'`;
- il segnale `o_done` deve restare a `'1'` fino a che il segnale `i_start` non è riportato a `'0'`;
- il segnale `i_start`, che indica l'inizio di una nuova nuova elaborazione, non viene riposto a `'1'` fin tanto che `o_done` non è stato riportato a `'0'`.

```
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;
```

Figura 1.4: Interfaccia del componente da realizzare

Si definiscono quindi:

- `i_clk` è il segnale di `CLOCK` in ingresso;
- `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` in ingresso;
- `i_data` è il segnale (vettore) che arriva (un ciclo di clock dopo) 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;
- `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 scrivere, `'0'` per poter leggere;
- `o_data` è il segnale (vettore) di uscita dal componente verso la memoria.

1.3 Esempio di esecuzione

Per rendere più chiara la specifica del progetto ed il comportamento che il componente dovrebbe assumere a fronte del segnale `i_start = '1'`, si supponga che esso debba processare un'immagine di dimensione 3×2 pixel, i cui valori sono: 23, 89, 201, 240, 12 e 82. Assumendo che i valori dei pixel siano stati precedentemente salvati in memoria come da specifica, alla fine del processamento, ovvero quando il modulo porterà ad `'1'` il segnale `o_done`, il contenuto della memoria dovrà essere il seguente (figura 1.5) :

L'immagine equalizzata, di dimensione pari a quella dell' immagine originale, sarà composta dai pixel processati, ovvero quelli contenuti dall'indirizzo 8 all'indirizzo 13 della memoria in figura 1.5 e ottenuti mediante l'algoritmo descritto nella sezione 1.2.
Per ottenere questo risultato, il modulo ha quindi effettuato i seguenti passaggi:

1. lettura degli indirizzi 0 e 1 per ricavare la dimensione dell'immagine. In questo esempio `C = 3` e `R = 2`, quindi la dimensione sarà $\texttt{C} \times \texttt{R} = 6$;
2. lettura dei 6 indirizzi immediatamente successivi contenenti i valori dei pixel;
3. individuazione di `MIN_PIXEL_VALUE` e `MAX_PIXEL_VALUE`, che in questo caso sono rispettivamente 12 e 240;
4. calcolo di `DELTA_VALUE = 240 - 12 = 228`;
5. calcolo dello `SHIFT_LEVEL = 8 - FLOOR(LOG2(228 + 1)) = 8 - 7 = 1`.
6. I seguenti passi verranno applicati a tutti i pixel dell'immagine.
   Per semplicità, si mostrano i passaggi di processamento di un solo pixel, per esempio il secondo, contenuto all'indirizzo 3 della memoria e di valore `CURR_PIXEL_VALUE = 89`.
   (a) Calcolo di `TEMP_PIXEL = (89 - 12) « 1 = 77 « 1 = 154`;
   (b) calcolo di `NEW_PIXEL_VALUE = MIN(154, 255) = 154`;
   (c) scrittura di `154`, valore del pixel processato, all'indirizzo 9, calcolato con la formula generale $\texttt{C} \times \texttt{R} + \texttt{indice del pixel} + 1$, che applicata a questo pixel diventa $6 + 2 + 1 = 9$.

Indirizzo	Contenuto	
0	3	Dimensione immagine
1	2	
2	23	Pixel immagine originale
3	89	
4	201	
5	240	
6	12	
7	82	
8	22	Pixel immagine equalizzata
9	154	
10	255	
11	255	
12	0	
13	140	
...	...	
65535		

Figura 1.5: Rappresentazione della memoria dopo il processamento dell'immagine di esempio

Per questo esempio è stata effettuata una simulazione dove vengono mostrati i segnali di particolare interesse, il cui risultato è mostrato in Figura 3.1

2. Architettura

Per la realizzazione di questo progetto si è scelta un'architettura formata da un *data-path* controllato da una *Finite State Machine* che viene sollecitata da segnali in ingresso al componente (`i_start` e `i_rst`) e da segnali prodotti dal data-path stesso (`o_end0`, `o_end1` e `o_end2`). Compito della FSM è quindi la gestione dei vari componenti e segnali del data-path attraverso le sue uscite computate in ogni stato.
In particolare, la relazione tra FSM e data-path è rappresentata in Figura 2.1.

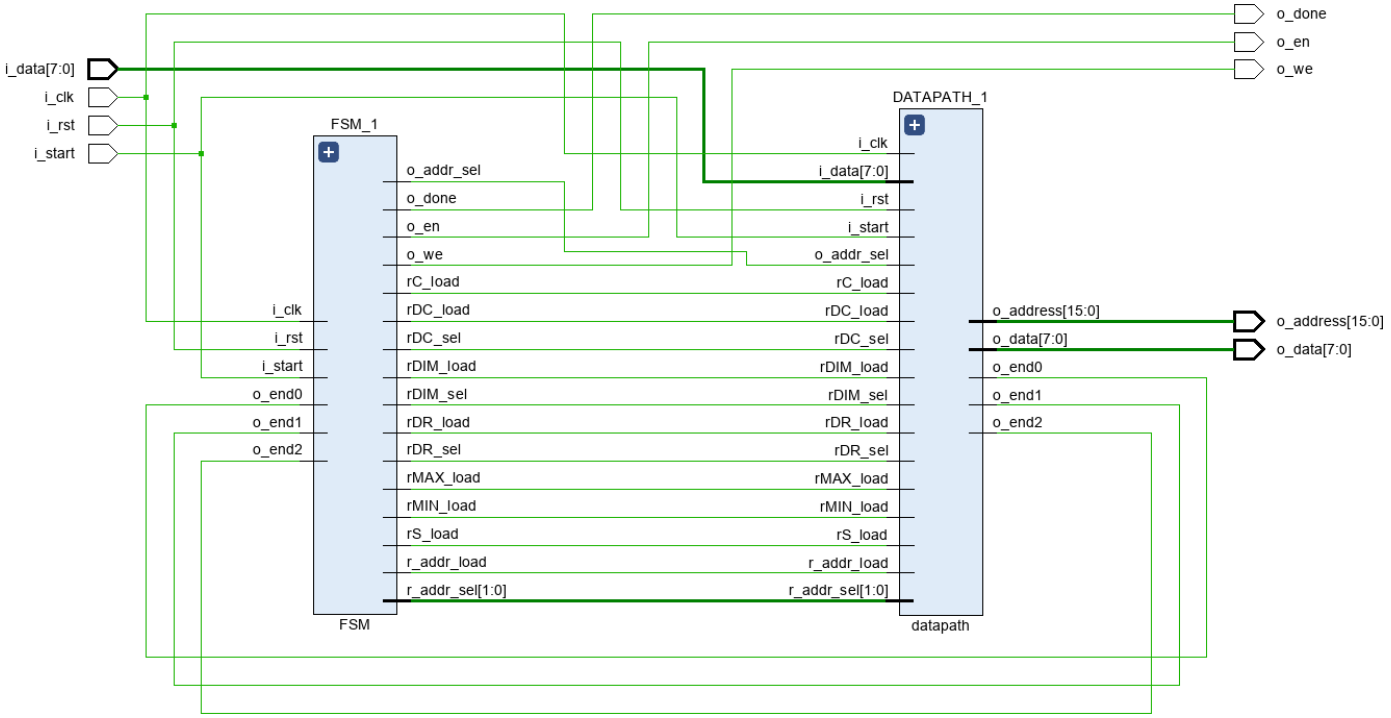

Figura 2.1: Schematico del componente

La progettazione è iniziata con la definizione del data-path, cercando di capire quali componenti fossero necessari e quali relazioni ci fossero tra essi. Successivamente si è passati alla realizzazione della FSM designando gli stati essenziali per una corretta computazione.

Per il calcolo della dimensione dell'immagine, rilevante è stata la scelta di design di non utilizzare l'operatore di libreria di moltiplicazione (`*`), molto oneroso sia da un punto di vista spaziale che temporale. Si è optato invece per la realizzazione di un modulo che utilizzi in modo più opportuno la FSM, come spiegato in seguito nella sezione 2.1.2.

Dopo aver descritto il più dettagliatamente possibile i due componenti, si è iniziata l'implementazione del modulo in VHDL, per la quale si è adottata una specifica mista: un approccio *strutturale* per l'istanziamento di data-path e FSM, ed una rappresentazione *behavioral* per la loro descrizione. In particolare, per il data-path si è utilizzato anche un approccio *dataflow* in modo tale da sfruttare al meglio le potenzialità di ogni tipo di specifica. Inoltre, per entrambi i componenti si è utilizzata una logica *multiprocesso*.

2.1 Data-path

Il data-path mostrato in Figura 2.2 è stato progettato mediante registri, sommatori, sottrattori, multiplexer e moduli necessari per il corretto funzionamento dell'algoritmo. Il componente può essere logicamente scomposto in quattro regioni aventi specifiche funzionalità che verranno descritte in seguito nel dettaglio. Ognuna di queste parti è stata concepita per lavorare in parallelo in modo da ottimizzare il componente da un punto di vista temporale, ma anche per essere riutilizzate in più frangenti e quindi per minimizzare il più possibile il numero di unità di calcolo utilizzate.
Per l'implementazione in VHDL di registri e moduli del componente sono stati definiti dei *process* specifici, mentre per i sommatori e sottrattori dei vettori *unsigned* sono stati utilizzati gli operatori definiti nelle librerie `IEEE.STD_LOGIC_1164.ALL` e `IEEE.STD_LOGIC_UNSIGNED.ALL`.

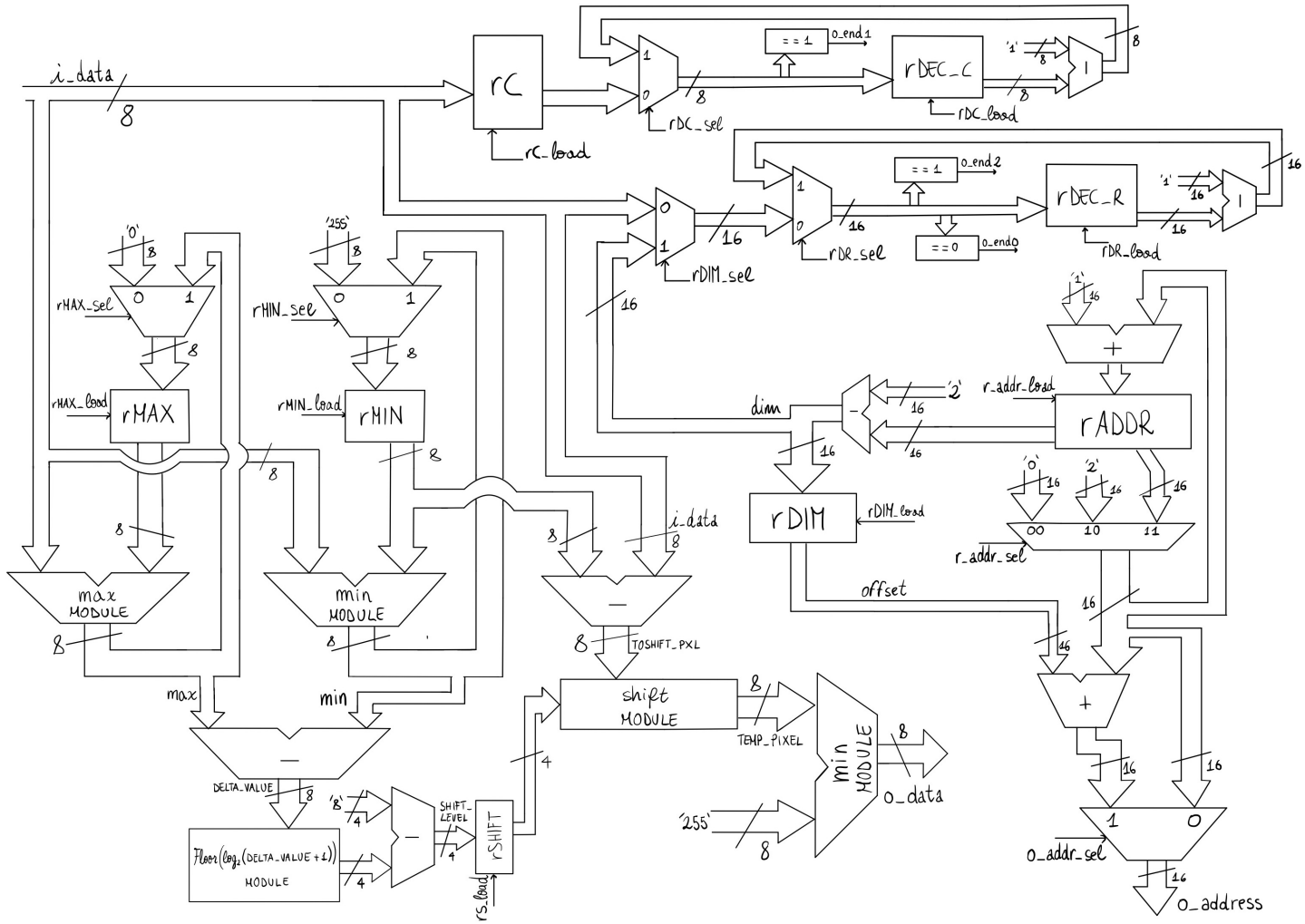

Figura 2.2: Schema del data-path

2.1.1 Gestione degli indirizzi di memoria

Questa parte del data-path, rappresentata in Figura 2.3, gestisce l'incremento del vettore `o_address`, consentendo l'accesso all'indirizzo di memoria opportuno. Il segnale `o_address` deve poter rappresentare $128 \times 128 \times 2 + 2$ indirizzi e per questo motivo nella zona in questione vengono utilizzati componenti a 16 bit. Si precisa che, per ragioni di semplicità e coerenza, si è continuato ad utilizzare 16 bit anche dove non strettamente necessario.

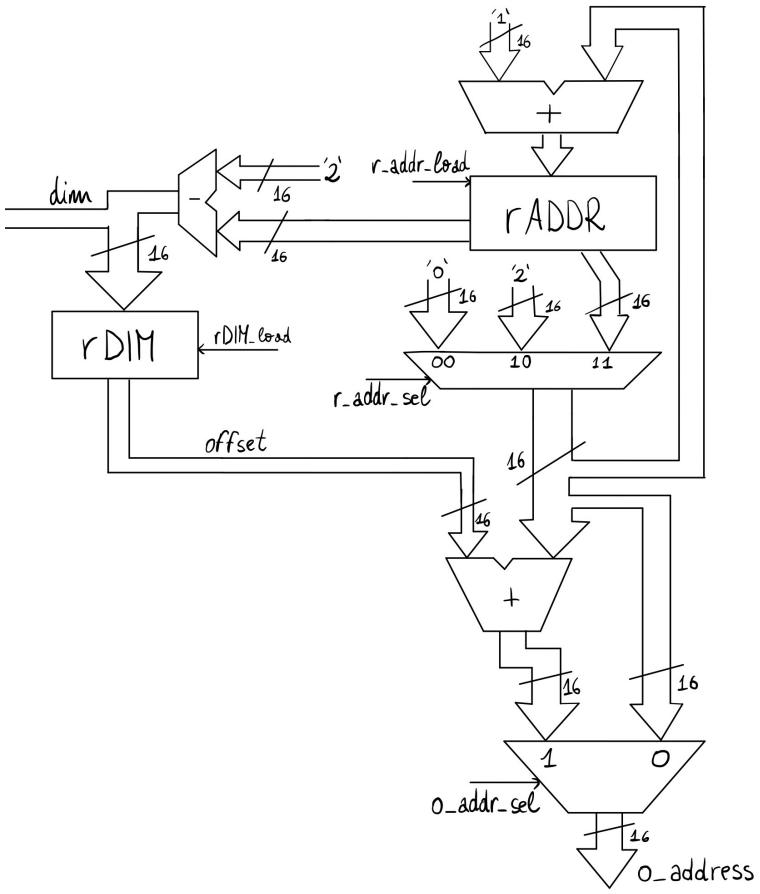

Figura 2.3: Schema della parte del data-path che gestisce gli indirizzi

In fase di inizializzazione, la FSM pone i valori dei selettori `r_addr_sel` e `o_addr_sel` rispettivamente a `"00"` e `'0'` per far sì che il valore di `o_address` corrisponda all'indirizzo 0 della memoria. Durante la fase di lettura, invece, il valore di `o_addr_sel` rimane inalterato, mentre il selettore `r_addr_sel` viene posto a `"11"` per selezionare il segnale di uscita del registro `rADDR`, contenente il valore dell'indirizzo successivo.
Finita la prima fase di lettura sequenziale dei dati, per ripartire a leggere dal primo pixel, `o_address` viene inizializzato all'indirizzo 2 tramite il selettore `r_address_sel` posto a `"10"`.
Il selettore `o_addr_sel`, invece, verrà posto a `'1'` in fase di scrittura in memoria dove il pixel contenuto all'indirizzo i-esimo verrà processato e scritto all'indirizzo i-esimo più un offset pari alla dimensione dell'immagine ($\texttt{C} \times \texttt{R}$).
Questo *offset* viene salvato nel registro `rDIM` durante il ciclo di clock successivo a quello della lettura dell'ultimo pixel dell'immagine ed è ottenuto sottraendo 2 al valore salvato nel registro `rADDR`. La costante 2 deriva dal fatto che sono stati contati anche i 2 primi indirizzi della memoria.

2.1.2 Conteggio dimensione dell'immagine

Questa zona di data-path, mostrata in Figura 2.4, si occupa di iterare per un numero di cicli di clock pari alla dimensione dell'immagine, in modo tale da leggere il numero corretto di pixel presenti in memoria.

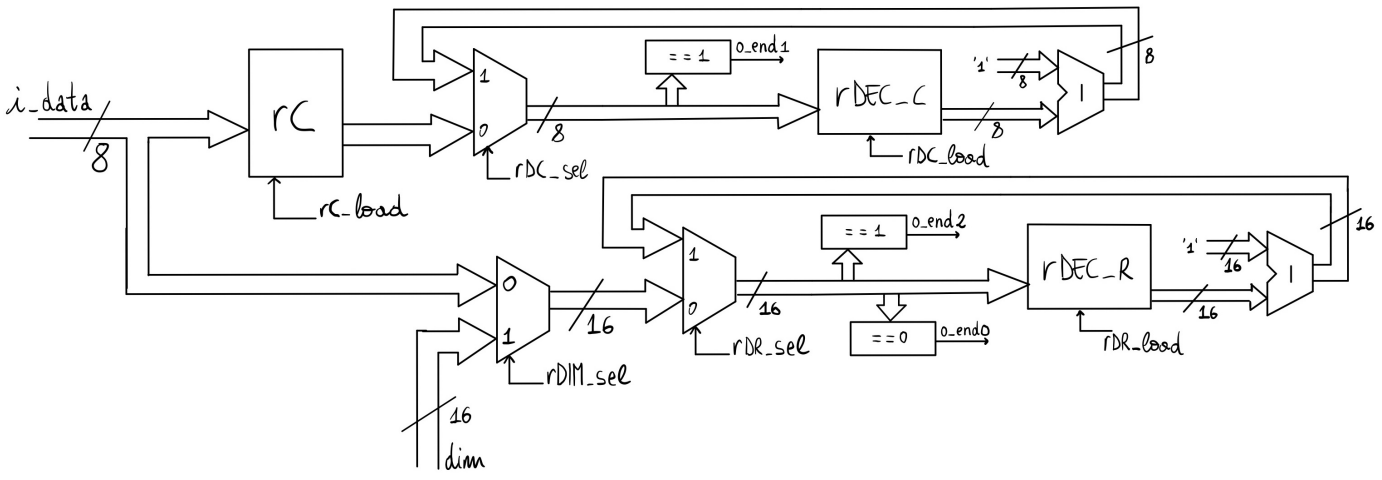

Figura 2.4: Schema della parte del data-path incaricata del conteggio della dimensione dell'immagine

Durante la prima fase, non avendo calcolato il valore di `C` $\times$ `R` tramite un moltiplicatore, si è deciso di iterare scandendo ciascun pixel riga per riga, immaginando che siano disposti in memoria come una matrice di dimensione `C` $\times$ `R`.

Per fare ciò, all'inizio della computazione, nel registro `rC` viene salvato il valore `C` che verrà propagato in `r_DEC_C` al ciclo di clock successivo, durante il quale in ingresso a `r_DEC_R` si troverà il valore `R`. Il valore in `r_DEC_C` viene quindi decrementato ad ogni ciclo di clock fino al raggiungimento del valore `1`, ovvero fino alla lettura dell'ultimo pixel della "riga" corrente. Questo evento provoca l'asserzione del segnale `o_end1` tramite il comparatore `==1` che comporterà, al ciclo di clock successivo, il decremento del valore contenuto in `r_DEC_R` e la re-inizializzazione di `r_DEC_C` con il valore `C`. Questo procedimento persiste fintanto che il valore salvato in `r_DEC_R` è diverso da zero, ovvero finché non si hanno più "righe" da leggere. Finita questa fase di lettura, il registro `r_DEC_R` viene sovrascritto con il valore `C` $\times$ `R` proveniente dal segnale `dim`[1] ed ottenuto con il metodo spiegato in 2.1.1, con lo scopo di riutilizzare i componenti di questa parte anche per la fase successiva dell'algoritmo: per accedere al corretto numero di pixel verrà decrementato solamente il nuovo valore contenuto nel registro `r_DEC_R`. Quest'ultima verrà comunque ripresa nella sezione 2.1.4. Infine, si noti che per applicare questa *ottimizzazione*, che comporta l'utilizzo del segnale `dim`, è necessario l'utilizzo di alcuni segnali e componenti a 16 bit.

2.1.3 Individuazione massimo e minimo valore tonale

Lo scopo di questa zona di data-path, rappresentata in Figura 2.5, è la ricerca del massimo e minimo valore tonale dell'immagine e lavora esclusivamente durante la prima fase dell'algoritmo. Infatti, una volta trovati i due valori, la sezione che si occupa dell'effettiva equalizzazione dell'immagine (2.1.4) può iniziare a processare i singoli pixel, dando inizio alla fase successiva.

La ricerca inizia con l'arrivo del primo pixel letto dalla memoria, che viene comparato con i valori contenuti in `rMIN` e `rMAX`, precedentemente inizializzati a `255` (massimo valore tonale) e `0` (minimo valore tonale) rispettivamente. Successivamente, i confronti verranno effettuati per ogni pixel dell'immagine con il valore precedentemente salvato nei due registri, ponendo entrambi i selettori a `'1'` e abilitandone la scrittura. Questi confronti vengono realizzati attraverso i componenti che in Figura 2.5 sono chiamati `max MODULE` e `min MODULE`, tradotti in VHDL con il seguente codice:

```
-- max MODULE
    process (i_data, o_rMAX)
    begin
        if (i_data > o_rMAX) then
            max <= i_data;
        else
            max <= o_rMAX;
        end if;
    end process;
```

dove `o_rMAX` è l'uscita del registro `rMAX`. In maniera duale si è tradotto `min MODULE`.

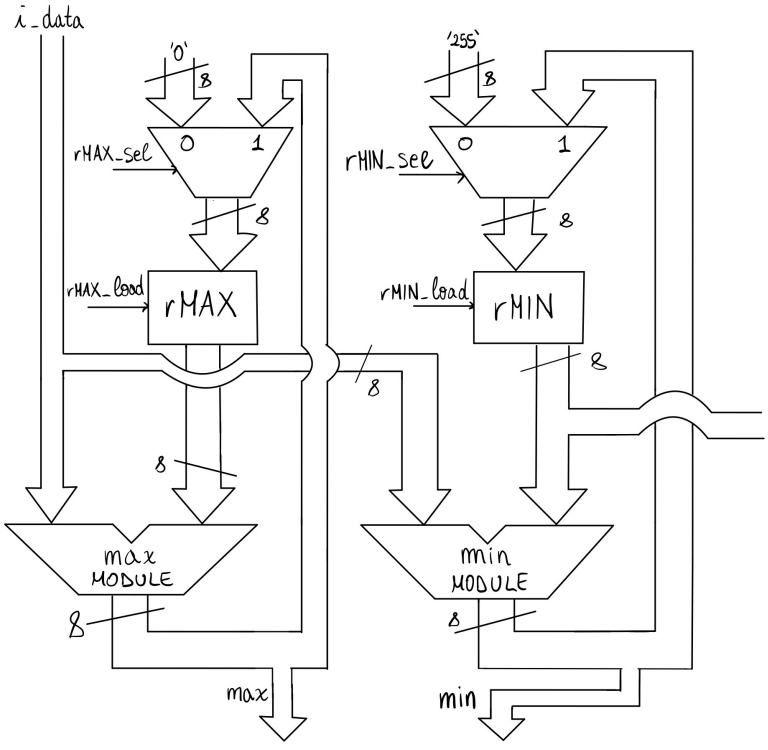

Figura 2.5: Schema della parte del data-path che individua massimo e minimo valore tonale dell'immagine

2.1.4 Processamento dei pixel

Questa parte di data-path, raffigurata in Figura 2.6, si occupa della seconda fase dell'algoritmo di equalizzazione: processamento del singolo pixel e scrittura in memoria del risultato prodotto.

[1] Segnale a 16 bit proveniente dalla gestione degli indirizzi (2.1.1) e contenente la dimensione dell'immagine

Essa ha inizio al ciclo di clock successivo alla lettura dell'ultimo pixel, durante il quale i segnali `min` e `max` contengono i valori di minimo e massimo corretti per il calcolo del `DELTA_VALUE`, necessario in ingresso al modulo `Floor(log2(DELTA_VALUE + 1))`. Quest'ultimo, implementato in VHDL attraverso opportuni controlli a soglia, produce in uscita un valore contribuente al calcolo dello `SHIFT_VALUE`, che viene salvato nel registro `r_SHIFT` per essere utilizzato durante tutta la fase successiva.

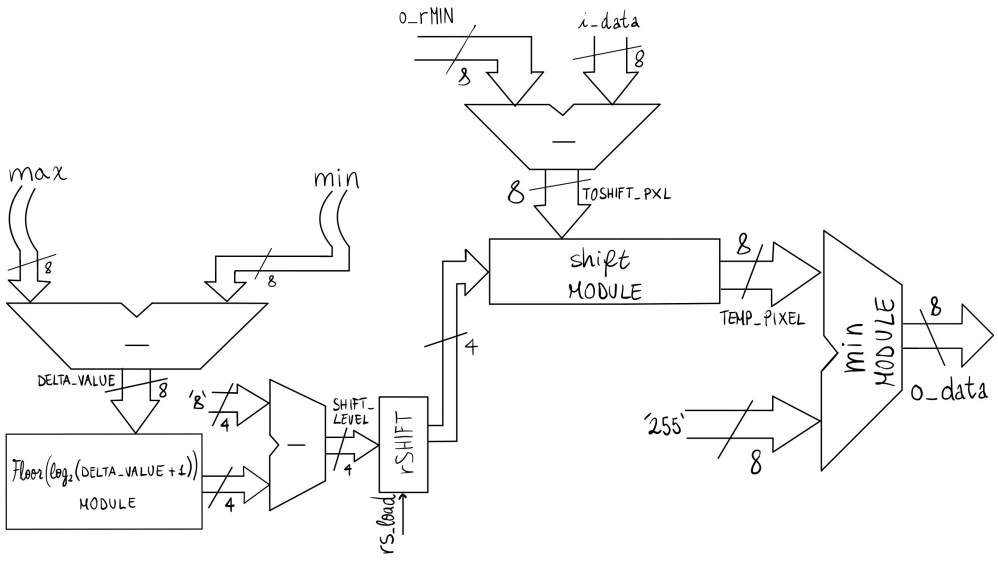

Figura 2.6: Parte del data-path che si occupa del processamento dei pixel

Al ciclo di clock successivo della richiesta di lettura del primo pixel, esso viene sottratto al `MIN_PIXEL_VALUE` (proveniente da `o_rMIN`) e shiftato dallo `shift MODULE` componendo così il primo `TEMP_PIXEL`. Il `min MODULE` porterà quindi in uscita il minimo valore tra il `TEMP_PIXEL` appena calcolato e `255`.
Per essere in grado di ripetere questo procedimento per tutti i pixel rimanenti, la FSM è stata progettata per effettuare una corretta alternanza di richieste di lettura e scrittura in memoria. In particolare, se al ciclo i-esimo si ha in ingresso il pixel contenuto nell'indirizzo j (richiesto al ciclo *i-1*), questo deve essere processato e scritto (all'indirizzo $j+offset$) nel medesimo ciclo di clock.
Lo `shift MODULE` è stato implementato in VHDL usando una particolare configurazione di concatenamento di segnali per ogni possibile valore di `SHIFT_VALUE`.
Di seguito si riporta il codice VHDL con cui sono stati tradotti entrambi i moduli.

```vhdl
-- Floor(log2(DELTA_VALUE + 1)) MODULE
--- DELTA_VALUE_1 = DELTA_VALUE + 1
process(DELTA_VALUE_1)
begin
    if (DELTA_VALUE_1  = "00000000") then
        floorlog <= "1000";
    elsif (DELTA_VALUE_1  = "00000001") then
        floorlog <= "0000";
    elsif (DELTA_VALUE_1  >= "00000010" and DELTA_VALUE_1  < "00000100") then
        floorlog <= "0001";
    elsif (DELTA_VALUE_1  >= "00000100" and DELTA_VALUE_1  < "00001000") then
        floorlog <= "0010";
    elsif (DELTA_VALUE_1  >= "00001000" and DELTA_VALUE_1  < "00010000") then
        floorlog <= "0011";
    elsif (DELTA_VALUE_1  >= "00010000" and DELTA_VALUE_1  < "00100000") then
        floorlog <= "0100";
    elsif (DELTA_VALUE_1  >= "00100000" and DELTA_VALUE_1  < "01000000") then
        floorlog <= "0101";
    elsif (DELTA_VALUE_1  >= "01000000" and DELTA_VALUE_1  < "10000000") then
        floorlog <= "0110";
    elsif (DELTA_VALUE_1  >= "10000000") then
        floorlog <= "0111";
    else floorlog <= "XXXX";
    end if;
end process;
```

```vhdl
-- shift MODULE
process(TOSHIFT_PXL, o_rSHIFT)
begin
    case o_rSHIFT is
        when "0000" =>
            TEMP_PIXEL <= "00000000" & TOSHIFT_PXL;
        when "0001" =>
            TEMP_PIXEL <= "0000000" & TOSHIFT_PXL & '0';
        when "0010" =>
            TEMP_PIXEL <= "000000" & TOSHIFT_PXL & "00";
        when "0011" =>
            TEMP_PIXEL <= "00000" & TOSHIFT_PXL & "000";
        when "0100" =>
            TEMP_PIXEL <= "0000" & TOSHIFT_PXL & "0000";
        when "0101" =>
            TEMP_PIXEL <= "000" & TOSHIFT_PXL & "00000";
        when "0110" =>
            TEMP_PIXEL <= "00" & TOSHIFT_PXL & "000000";
        when "0111" =>
            TEMP_PIXEL <= '0' & TOSHIFT_PXL & "0000000";
        when "1000" =>
            TEMP_PIXEL <= TOSHIFT_PXL & "00000000";
        when others =>
            TEMP_PIXEL <= "XXXXXXXXXXXXXXXX";
    end case;
end process;
```

2.2 Finite State Machine

La Finite State Machine (FSM), componente che si occupa della gestione dello stato della computazione, è stata realizzata mediante una macchina di Moore ed in particolare attraverso i seguenti processi: *processo di reset*, *processo di calcolo della funzione di uscita* e *processo di calcolo della funzione stato prossimo*.

La FSM in questione è composta da 12 stati:

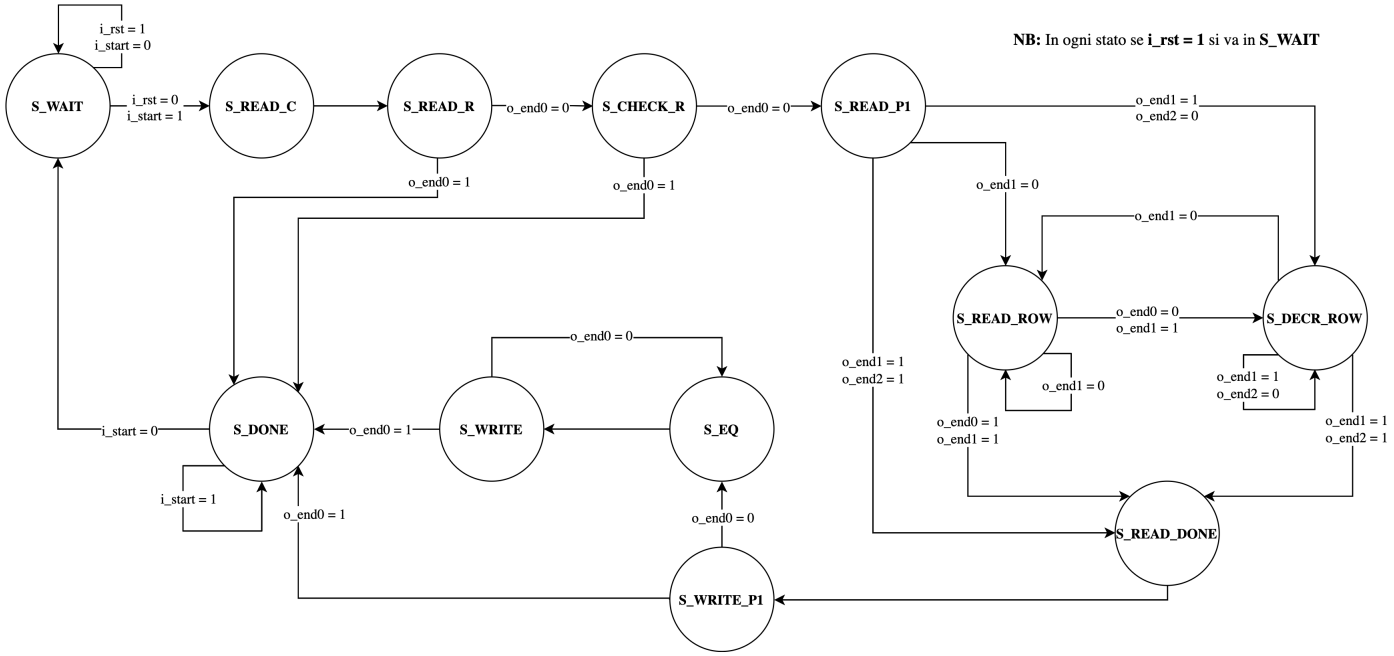

Figura 2.7: Diagramma degli stati della FSM

- `S_WAIT`: stato di attesa dell'asserzione del segnale `i_start` e stato in cui si posiziona la FSM dopo la ricezione del segnale `i_rst`. Inizializzazione dei registri.

Prima fase dell'algoritmo:

- `S_READ_C`: stato in cui si richiede alla memoria la lettura dell'indirizzo 0, contenente il numero di colonne `C`.
- `S_READ_R`: stato in cui si richiede alla memoria la lettura dell'indirizzo 1, contenente il numero di righe `R`. Abilitazione alla scrittura in `rC` del dato in arrivo dalla memoria (`C`). Controllo della dimensione dell'immagine: se `C = 0` il comparatore `==0` pone `o_end0 = '1'` provocando la transizione nello stato `S_DONE`.
- `S_CHECK_R`: stato in cui si controlla il valore di `R`: se è `0`, il comparatore `==0` pone `o_end0 = '1'` provocando la transizione nello stato `S_DONE`; altrimenti passa allo stato `S_READ_P1`.
- `S_READ_P1`: stato in cui si richiede la lettura del primo pixel. Abilitazione alla scrittura in `r_DEC_C` del valore `C` ed in `r_DEC_R` del dato in arrivo dalla memoria (`R`). Controllo della dimensione dell'immagine: se `C = 1` e `R = 1`, `o_end1` e `o_end2` sono entrambi pari ad `'1'` e provocano quindi la transizione nello stato `S_READ_DONE`; se solo `C = 1` allora solamente `o_end1 = '1'` e la FSM si porta nello stato `S_DECR_ROW`. Altrimenti procede nello stato `S_READ_ROW`.

Con l'arrivo del primo pixel in ingresso al componente, nei seguenti stati si inizia la ricerca del minimo e massimo valore tonale, portando `rMIN_load` e `rMAX_load` ad `'1'`.

- `S_READ_ROW`: stato in cui vengono letti i pixel di una "riga", ovvero fintanto che `o_end1 = '0'`. Al termine (`o_end1 = '1'`), se `o_end0 = '1'` significa che si è letta l'ultima "riga" e la FSM si porta nello stato `S_READ_DONE`; altrimenti si porta nello stato `S_DECR_ROW`.
- `S_DECR_ROW`: stato in cui si decrementa il valore `R` delle "righe" rimanenti. Se non si è nel caso `1×N` (`C = 1`), `o_end1 = '0'` e la FSM ritorna a leggere una nuova "riga" in `S_READ_ROW`; altrimenti rimane in questo stato fino alla lettura dell'ultimo pixel, ovvero fino a che `o_end2 = '0'`.

Seconda fase dell'algoritmo:

- `S_READ_DONE`: stato nel quale la FSM si posiziona alla ricezione dell'ultimo pixel. Richiesta di lettura del primo pixel da processare. Abilitazione alla scrittura in `r_DEC_R` e in `rDIM` del numero di pixel dell'immagine. Salvataggio del valore di *shift* all'interno di `rSHIFT`.

I valori di massimo e minimo sono stati calcolati e può quindi iniziare la fase di processamento e scrittura in memoria attraverso i seguenti stati:

- `S_WRITE_P1`: stato effettivo di processamento e scrittura del primo pixel. Se la dimensione dell'immagine è unitaria, la FSM, dopo la scrittura, si posiziona nello stato `S_DONE`.
- `S_EQ`: stato in cui si richiede la lettura del pixel successivo.
- `S_WRITE`: stato di effettivo processamento del pixel in arrivo dalla memoria e conseguente scrittura del suo nuovo valore. A differenza dello stato `S_EQ`, il contenuto del registro `r_DEC_R` non viene decrementato, permettendo il corretto conteggio del numero di pixel. Se `o_end0 = '0'` la FSM ritorna allo stato `S_EQ`, altrimenti, avendo processato e scritto l'ultimo pixel, si posiziona nello stato `S_DONE`.
- `S_DONE`: stato di fine della computazione. La FSM rimarrà in questo stato fintanto che il segnale `i_start` non verrà posto a zero, segno di corretta ricezione da parte della memoria del segnale `i_done = '1'`.

3. Risultati sperimentali

3.1 Sintesi

La sintesi del componente progettato, effettuata con Vivado sulla FPGA xc7a200tfbg484-1, è avvenuta con successo. Tramite il comando `report_utilization` si è appreso che sono stati utilizzati:

- 221 LUT as Logic (0.16%)
- 96 Register as Flip Flop (0.04%)
- 0 Register as Latch (risultato necessario per il corretto funzionamento del componente)

Inoltre, il comando `report_timing` ha riportato un *Data Path Delay* di $6.187ns$, esito che rispetta ampiamente la specifica sul periodo di clock (`CLK_PERIOD` $\leq 100ns$). Questo ritardo è causato dalla presenza di un percorso critico del modulo implementato che è risultato equivalente a 8 *Logic Levels*. In aggiunta, è stato calcolato uno *Slack (required time - arrival time)* pari a $93.662ns$. Questi risultati hanno permesso un'esecuzione delle simulazioni ad una frequenza superiore. Per la maggior parte dei testbench, ad esempio, si è impostato un valore di `CLK_PERIOD` pari a $10ns$.

3.2 Simulazioni

La fase di testing si è svolta in due diversi momenti: il primo tramite un'analisi statica del codice, il secondo attraverso un'analisi dinamica durante la quale si è valutato il comportamento del codice in esecuzione.
I testbench sono stati creati in modo tale da rispettare le regole della specifica, permettendoci di verificare la presenza di difetti ed errori che avrebbero compromesso la correttezza del risultato. Per tutti i test è stata effettuata la simulazione *behavioral* e successivamente la simulazione *functional* e *timing post-synthesis*, tutte con successo.

3.2.1 Configurazioni limite dell'immagine

I seguenti test ci hanno permesso di verificare la correttezza dell'intero componente ed in particolar modo della parte del data-path incaricata dell'applicazione dell'algoritmo di equalizzazione (2.1.4) e della parte incaricata del conteggio dimensionale dell'immagine (2.1.2). Inoltre, avendo utilizzato opportuni controlli per specifici casi limite, attraverso questi test siamo riusciti a verificare la correttezza delle transizioni da uno stato all'altro della macchina a stati finiti.

- **Immagine contenente entrambi i valori limite dei pixel (`0` e `255`)**:
  viene verificato il corretto funzionamento agli estremi del dominio dei valori tonali. Infatti, la loro presenza denota un caso limite dell'algoritmo in quanto il valore `0` implica `TEMP_PIXEL = CURR_PIXEL_VALUE` ed il valore `255` determina `SHIFT_LEVEL = 0`. Si verifica, quindi, che la nuova immagine sia uguale a quella originale.
- **Immagine contenente pixel dello stesso valore tonale**:
  lo scopo di questo testbench è verificare che il risultato del processamento sia un'immagine composta da pixel con unico valore pari a `0`, in quanto `TEMP_PIXEL = 0`.
- **Immagine avente dimensione nulla (`0`×`N` oppure `N`×`0`)**:
  viene verificato il corretto funzionamento all'estremo inferiore del dominio della dimensione dell'immagine. Si controlla che il segnale `o_end0` venga asserito nello stato `S_READ_R`, se `C = 0`, o nello stato `S_CHECK_R`, se `R = 0`, ed infine che il processo termini lasciando la memoria invariata.
- **Immagine avente dimensione massima (`128`×`128`)**:
  viene verificato il corretto funzionamento all'estremo superiore del dominio della dimensione dell'immagine (16384 pixel).
- **Immagine avente dimensione del tipo `1`×`N`**:
  questo testbench verifica il corretto funzionamento a fronte di quella che è risultata essere una configurazione critica durante l'implementazione. In particolare, è di interesse controllare che, all'asserzione del segnale `o_end1`, causata dalla condizione `C = 1`, la FSM rimanga nello stato `S_DECR_ROW` fino alla lettura dell'ultimo pixel.

- **Immagine avente dimensione unitaria**:
  questo testbench verifica il corretto funzionamento a fronte di quest'ulteriore configurazione critica rilevata durante l'implementazione. È quindi necessario verificare che, grazie all'asserzione dei segnali `o_end1` e `o_end2`, la FSM si sposti dallo stato `S_READ_P1` allo stato `S_READ_DONE`, leggendo così l'unico pixel presente in memoria.

3.2.2 Transizioni di fine elaborazione

I seguenti test ci hanno permesso in particolar modo di verificare la correttezza e la robustezza della FSM a fronte di diverse configurazioni di segnali in ingresso per il cambiamento di stato dell'elaborazione.

- **Asserzione di un reset casuale durante l'esecuzione**:
  lo scopo di questo testbench è quello di verificare che il componente, a fronte del segnale `i_rst = 1` in un istante di tempo casuale (durante `i_start = 1` e `o_done = 0`), interrompa l'esecuzione e si porti nello stato `S_WAIT` grazie al processo di reset della FSM.
- **Asserzione di un reset casuale dopo l'esecuzione**:
  lo scopo di questo testbench è quello di verificare che il componente, a fronte del segnale `i_rst = 1` in un istante di tempo casuale (durante `i_start = 1` e `o_done = 1`), si comporti come descritto nel punto precedente.
- **Inizio di una nuova computazione senza la ricezione del segnale di reset**:
  con questo testbench si vuole verificare che dopo aver concluso il processamento di un'immagine (`o_done = 1`), il modulo, a fronte di un nuovo segnale di *start*, possa iniziare correttamente un'ulteriore elaborazione anche senza essere resettato.

3.2.3 Testbench serie di immagini casuali

Tramite un generatore sono stati creati vari test contenenti immagini di dimensione variabile composte da differenti valori di pixel. Si è quindi progettato un testbench che, durante la simulazione sequenziale di questi test, producesse configurazioni di segnali in ingresso al componente che causino, in modo casuale, le possibili transizioni descritte nella sezione precedente (3.2.2). La correttezza del risultato scritto in memoria e del comportamento del modulo è verificata dal testbench stesso.
Esso ci ha permesso di testare in modo completo la robustezza del modulo, anche a fronte di un grande numero di elaborazioni consecutive.

3.2.4 Testbench immagine di esempio

Questo testbench è stato effettuato per mostrare il corretto funzionamento del componente attraverso le forme d'onda dei segnali più significativi. Inoltre, per capire meglio il suo comportamento, è stata effettuata una simulazione *behavioral* in modo da poter vedere il `curr_state` della FSM e altri segnali caratteristici di questa implementazione (come `o_end0`, `o_end1` e `o_end2`). L'immagine da equalizzare è la stessa del semplice esempio descritto in 1.3:
Dimensione immagine: 3×2;
Pixel in ingresso: `23`, `89`, `201`, `240`, `12` e `82`;
Valore corretto dei pixel equalizzati: `22`, `154`, `255`, `255`, `0` e `140`;

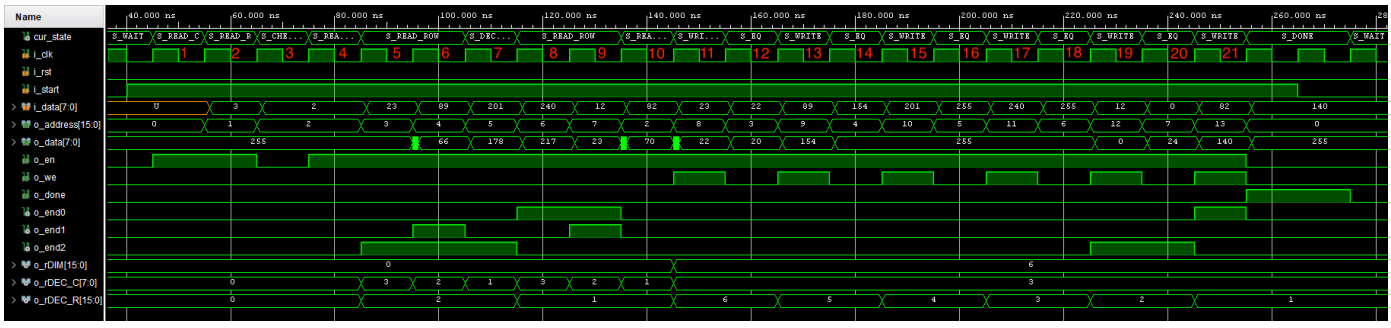

Figura 3.1: Forme d'onda del testbench di esempio

4. Conclusioni

La realizzazione di questo progetto ci ha permesso di mettere in pratica quanto appreso durante il corso e di apprendere i principi fondamentali della progettazione digitale tramite linguaggi di descrizione dell'*hardware*, in particolare il VHDL. Il risultato, come già dichiarato, è un componente funzionante e correttamente sintetizzabile che vanta *performance* temporali ben più elevate di quelle richieste, e di un utilizzo molto ridotto dell'hardware disponibile sulla FPGA d'interesse.
Facendo una semplice analisi di complessità dell'algoritmo da implementare, considerando una memoria con le caratteristiche indicate nella specifica, si può verificare che il limite inferiore del numero di cicli di clock necessari per l'esecuzione, è circa pari a

$$2 \times \mathtt{P} + \mathtt{P} + 2 \tag{4.1}$$

dove `P` è il numero di pixel dell'immagine ed i termini rappresentano rispettivamente: le due fasi di lettura, la fase di scrittura e la lettura dei primi due indirizzi. Come si può verificare dal testbench in Figura 3.1, il modulo progettato, non considerando i cicli di clock in cui si segnala la fine dell'elaborazione, impiega solamente un ciclo di clock in più rispetto a quanto calcolato con la formula 4.1. Ciò deriva dalla presenza dello stato `S_CHECK_R`, durante il quale viene posto a `'0'` il segnale `o_en` per evitare di leggere un pixel non esistente nel caso di un'immagine di dimensione nulla.
Il raggiungimento di una complessità temporale dell'ordine del limite inferiore precedentemente calcolato, è stato ottenuto grazie all'impiego della *concorrenzialità*, caratteristica distintiva della progettazione *hardware*, che ha permesso a diverse parti del data-path di funzionare in parallelo.
Inoltre, si sarebbe potuto diminuire il ritardo dovuto al percorso critico aggiungendo registri tra le varie unità di calcolo, soprattutto nella parte di processamento (2.1.4), a discapito di un aumento del numero di stati della FSM (e quindi di cicli di clock) e del numero di componenti utilizzati. Tuttavia, si è preferito non applicare quest'ulteriore ottimizzazione poiché, oltre a non essere necessaria, avrebbe complicato ulteriormente la progettazione e la descrizione dell'implementazione stessa.