Progetto finale di Reti Logiche

Piani Alessandro - Matricola n. 910280

Anno Accademico 2020/2021

Indice

1 Introduzione **2**
1.1 Specifica del progetto . . . 2
1.2 Interfaccia del componente . . . 2
1.3 Dati e descrizione memoria . . . 2
1.4 Equalizzazione dell'immagine . . . 3
1.4.1 Esempio . . . 3

2 Architettura **6**
2.1 Esecuzione della computazione . . . 6
2.2 Macchina a Stati Finiti . . . 6
2.3 Schema funzionale . . . 8
2.3.1 Registri . . . 8
2.3.2 Funzionalità in dettaglio . . . 8
2.4 Scelte progettuali . . . 10

3 Risultati sperimentali **11**
3.1 Sintesi . . . 11
3.1.1 Registri sintetizzati . . . 11
3.1.2 Area occupata . . . 11
3.1.3 Report di timing . . . 11
3.1.4 Warnings post synthesis . . . 12
3.2 Simulazioni . . . 12

4 Conclusioni **14**

1 Introduzione

1.1 Specifica del progetto

Lo scopo di questo progetto è implementare in linguaggio VHDL un componente hardware che consente di equalizzare una o più immagini in scala di grigi a 256 livelli, per mezzo del metodo di equalizzazione dell'istogramma. Questo metodo consiste nel trasformare ogni pixel di un'immagine al fine di ricalibrarne il contrasto quando l'intervallo dei valori di intensità sono molto vicini, effettuandone una distribuzione su tutto l'intervallo di intensità, in modo tale da incrementare il contrasto.

1.2 Interfaccia del componente

L'interfaccia del componente è così descritta:

```
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:

- `i_clk` rappresenta il segnale di CLOCK in ingresso generato dal test bench;
- `i_rst` è il segnale generato dal test bench che riporta la macchina nello stato di reset e che inizializza la macchina pronta per ricevere il primo segnale di START;
- `i_start` è il segnale di inizio elaborazione generato dal test bench;
- `i_data` rappresenta il segnale che arriva dalla memoria in seguito ad una richiesta di lettura;
- `o_address` è il segnale di uscita che manda l'indirizzo (su cui si vuole scrivere o che si vuole leggere) alla memoria;
- `o_done` è il segnale di uscita generato dal componente che comunica la fine dell'elaborazione e il dato di uscita scritto in memoria;
- `o_en` è il segnale di ENABLE generato dal componente usato per poter comunicare con la memoria (sia in lettura sia in scrittura);
- `o_we` è il segnale di WRITE ENABLE generato dal componente che viene inviato alla memoria per poterci scrivere. Per leggere da memoria esso deve essere 0;
- `o_data` rappresenta il segnale di uscita dal componente verso la memoria.

1.3 Dati e descrizione memoria

Per poter leggere l'immagine da elaborare ed in seguito scrivere l'immagine equalizzata, il componente deve interfacciarsi con una memoria dove è memorizzata l'immagine. Essa è salvata, sequenzialmente e riga per riga, in una memoria con indirizzamento al byte nel seguente modo:

- Nelle prime due posizioni da 0 a 1 della memoria si salvano le dimensioni dell'immagine, ciascuna di dimensione di 8 bit: il byte in posizione 0 si riferisce al numero di colonne (N-COL), mentre il byte in posizione 1 si riferisce al numero di righe (N-RIG);

- Dalla posizione 2 si salvano i pixel dell'immagine da elaborare, ciascuno di un 8 bit, in byte contigui. Quindi il byte all'indirizzo 2 è il primo pixel della prima riga dell'immagine;
- Dalla posizione 2 + (N-COL · N-RIG) vengono memorizzati i pixel dell'immagine equalizzata, ciascuno di un 8 bit.

Occorre precisare che la dimensione massima dell'immagine è di $128 \cdot 128$ pixel.

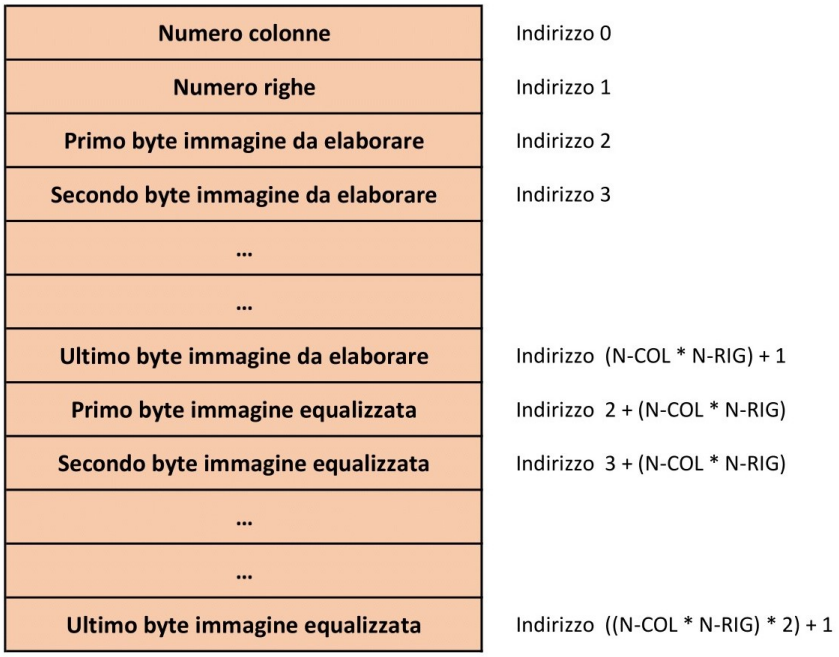


Figura 1: Rappresentazione della memoria

1.4 Equalizzazione dell'immagine

L'immagine viene equalizzata tramite una versione semplificata dell'algoritmo standard di equalizzazione dell'istogramma, che deve trasformare ogni suo pixel come segue:

$$DELTA\_VALUE = MAX\_PIXEL\_VALUE - MIN\_PIXEL\_VALUE$$
$$SHIFT\_LEVEL = (8 - \lfloor \log_2(DELTA\_VALUE + 1) \rfloor)$$
$$TEMP\_PIXEL = (CURRENT\_PIXEL\_VALUE - MIN\_PIXEL\_VALUE) << SHIFT\_LEVEL$$
$$NEW\_PIXEL\_VALUE = MIN\,(255, TEMP\_PIXEL)$$

Dove MAX_PIXEL_VALUE e MIN_PIXEL_VALUE sono il massimo e il minimo valore dei pixel dell'immagine, $CURRENT\_PIXEL\_VALUE$ è il valore del pixel da elaborare e NEW_PIXEL_VALUE è il valore del pixel trasformato che apparterrà all'immagine equalizzata.

1.4.1 Esempio

Si consideri il seguente stato della memoria contenente la sequenza dei pixel di un'immagine da equalizzare, di dimensione $4 \cdot 3$, prima dell'implementazione da parte del componente:

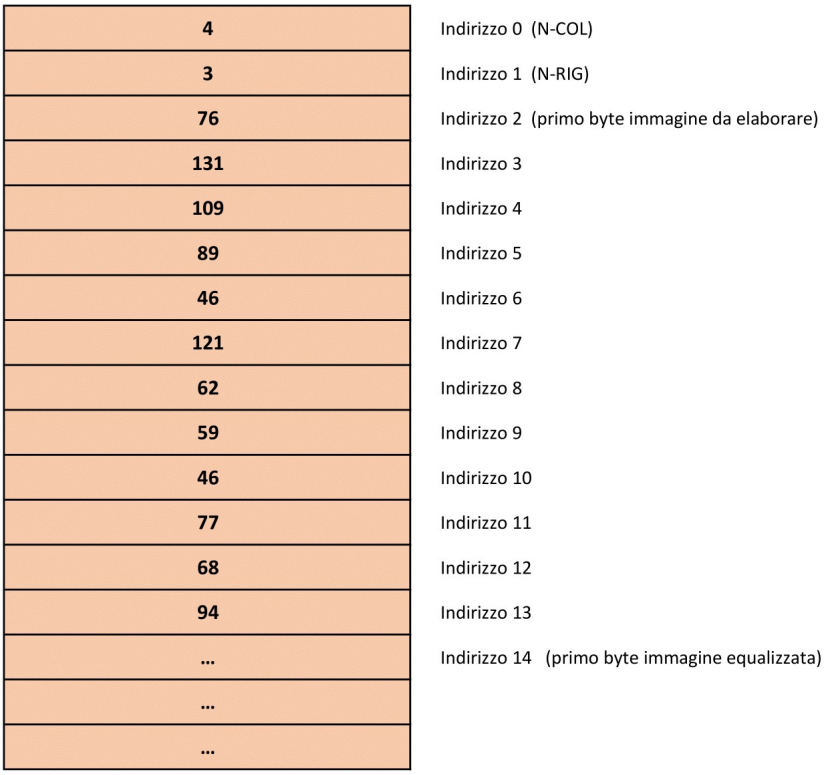


Figura 2: Esempio del contenuto della memoria prima dell'elaborazione **(si precisa che i valori che qui sono rappresentati in decimale, sono memorizzati in memoria con l'equivalente codifica binaria su 8 bit senza segno)**

In questo caso, il massimo e il minimo valore dei pixel dell'immagine (ovvero MAX_PIXEL_VALUE e MIN_PIXEL_VALUE) sono rispettivamente uguali a 131 e a 46. La differenza fra questi due valori permette di calcolare il valore di $DELTA\_VALUE$, che è pari a 85.
L'operazione successiva consiste nel calcolo del valore di $SHIFT\_LEVEL$ che risulta 2 (dal momento che il $\lfloor \log_2(DELTA\_VALUE + 1) \rfloor$ è uguale a 6, da cui consegue l'espressione $8 - 6 = 2$).
A questo punto dell'algoritmo, viene eseguita la vera e propria trasformazione dei pixel, che è descritta dal seguente processo:

1. Inizialmente, partendo dall'indirizzo 2, viene letto da memoria il valore del pixel da elaborare (chiamato $CURRENT\_PIXEL\_VALUE$) ed ad esso viene sottratto il valore di MIN_PIXEL_VALUE;
2. Il risultato ottenuto dalla sottrazione precedente, viene "shiftato a sinistra" di un numero di livelli pari al valore di $SHIFT\_LEVEL$ e il valore risultante viene salvato in $TEMP\_PIXEL$. Occorre precisare che, in questo caso, l'operazione "shift a sinistra" di un numero equivale a moltiplicarlo per 2^{SHIFT_LEVEL} in quanto i valori presenti in memoria sono codificati in binario su 8 bit senza segno;
3. Infine, il valore del pixel trasformato (cioè NEW_PIXEL_VALUE) è pari al minimo tra 255 e $TEMP\_PIXEL$, e viene scritto in memoria dal componente a partire dall'indirizzo
$2 + (\text{N-COL} \cdot \text{N-RIG})$.

Il processo appena descritto viene iterato per ogni pixel dell'immagine da equalizzare.
Ritornando all'esempio, se consideriamo il primo pixel dell'immagine, cioè
$CURRENT\_PIXEL\_VALUE = 76$, si ha un valore di $TEMP\_PIXEL = (76 - 46) \cdot 2^2 = 120$. Di conseguenza $NEW\_PIXEL\_VALUE = MIN\,(255, 120) = 120$, che corrisponde al pixel trasformato scritto in memoria all'indirizzo 14.
Quindi, al termine dell'elaborazione dell'immagine, lo stato della memoria è il seguente:

4	Indirizzo 0 (N-COL)
3	Indirizzo 1 (N-RIG)
76	Indirizzo 2 (primo byte immagine da elaborare)
131	Indirizzo 3
109	Indirizzo 4
89	Indirizzo 5
46	Indirizzo 6
121	Indirizzo 7
62	Indirizzo 8
59	Indirizzo 9
46	Indirizzo 10
77	Indirizzo 11
68	Indirizzo 12
94	Indirizzo 13
120	Indirizzo 14 (primo byte immagine equalizzata)
255	Indirizzo 15
252	Indirizzo 16
172	Indirizzo 17
0	Indirizzo 18
255	Indirizzo 19
64	Indirizzo 20
52	Indirizzo 21
0	Indirizzo 22
124	Indirizzo 23
88	Indirizzo 24
192	Indirizzo 25

Figura 3: Esempio del contenuto della memoria dopo l'elaborazione

2 Architettura

2.1 Esecuzione della computazione

Il componente dà inizio alla computazione quando il test bench ad esso fornito pone il segnale `i_start` a 1; in questo istante il componente passa dallo stato **S_reset** allo stato **S_enable**. Una volta terminata l'elaborazione, dopo avere scritto il risultato in memoria, il componente alza ad 1 il segnale `o_done` e si posiziona nello stato **S_done**. A questo punto, il test bench risponde ponendo a 0 il segnale `i_start` e stimolando il componente a resettare tutti i segnali dichiarati. Inoltre lo stimola a porre `o_done` a 0 e a riposizionarsi nello stato **S_reset**, in attesa che il segnale `i_start` venga alzato ad 1 per ripartire con una nuova elaborazione.
Il componente può inoltre ricevere in input un segnale `i_rst` che, se alzato ad 1, porta incondizionatamente il componente nello stato **S_reset**.

Per rispettare il funzionamento descritto brevemente in precedenza, il componente è stato realizzato progettando una FSM(D), macchina a stati finiti deterministica di Moore con *data path*, che combina una normale FSM di Moore con tipici circuiti sequenziali.
Nelle seguenti sezioni si trovano sia la descrizione della FSM, sia la descrizione della parte sequenziale della macchina che gestisce i registri utilizzati e il calcolo dell'immagine equalizzata.

2.2 Macchina a Stati Finiti

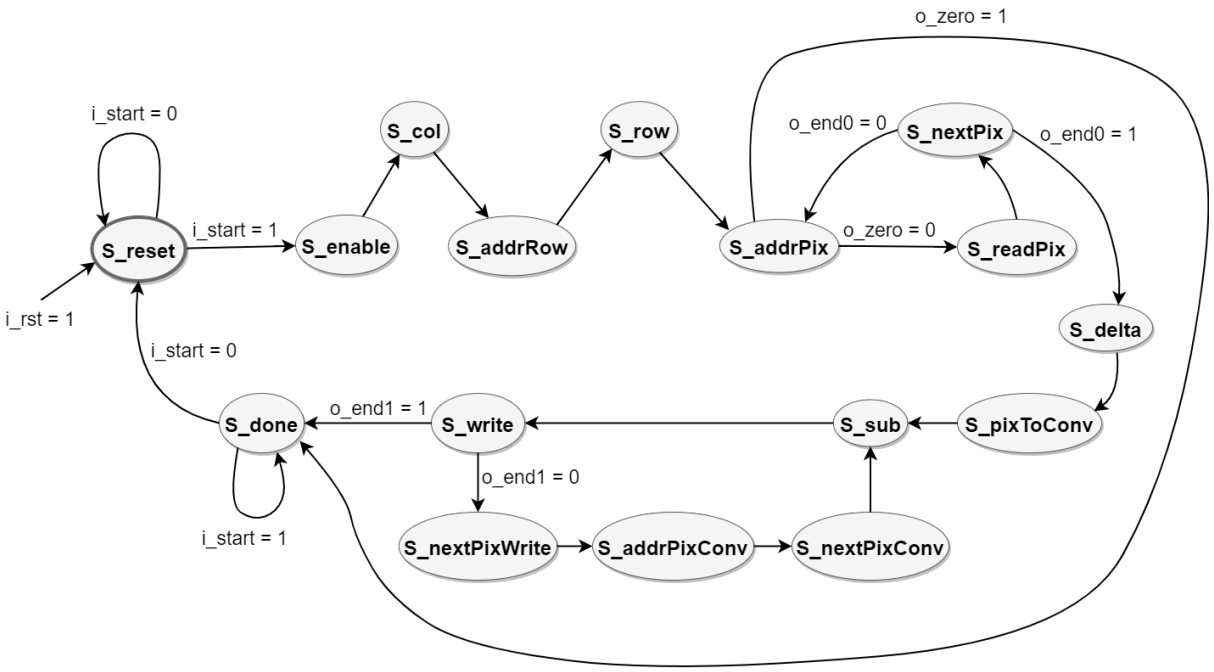


Figura 4: Macchina a Stati Finiti implementata

La macchina progettata è composta da 16 stati ed è rappresentata in figura 4, mentre ciò che ogni stato rappresenta è spiegato in dettaglio di seguito:

- **S_reset**: stato iniziale in cui vengono settati ad un valore di default i segnali del componente e si attende la ricezione di un segnale `i_start` per iniziare la computazione. In caso venga alzato il segnale `i_rst` si torna sempre in questo stato.
- **S_enable**: stato in cui viene alzato il segnale `o_en` a 1 per poter comunicare con la memoria in lettura. Inoltre, viene fornito alla RAM l'indirizzo della cella che contiene il numero di colonne dell'immagine da equalizzare (esso si trova sempre nella cella di memoria di indice 0).
- **S_col**: stato in cui il componente legge e salva in un registro (*col_reg*) il numero di colonne dell'immagine che gli è arrivato da memoria. Parallelamente, salva in un registro (*addrRead_reg*) l'indirizzo della cella di memoria che contiene il primo pixel da trasformare.

- **S_addrRow**: stato in cui viene fornito alla RAM l'indirizzo della cella contenente il numero di righe dell'immagine da equalizzare (esso si trova sempre nella cella di memoria di indice 1).
- **S_row**: stato in cui il componente legge e salva in un registro (*row_reg*) il numero di righe dell'immagine che gli è arrivato da memoria.
- **S_addrPix**: stato in cui il componente salva in un registro (*addrWrite_reg*) l'indirizzo della cella di memoria in cui sarà scritto il primo pixel equalizzato e, in contemporanea, controlla la dimensione dell'immagine, cioè il prodotto tra il numero di colonne e il numero di righe. Se essa è nulla, viene alzato ad 1 il segnale `o_zero` perché non ci sono pixel da convertire, quindi la computazione deve terminare e, di conseguenza, si va nello stato **S_done**. Diversamente, se non è nulla, il segnale `o_zero` rimane a 0 e si procede con la conversione dei pixel, quindi viene fornito alla RAM l'indirizzo della cella contenente il pixel dell'immagine da equalizzare e si passa allo stato **S_readPix**.
- **S_readPix**: stato in cui il componente legge e salva in un registro (*currentPix_reg*) il pixel dell'immagine che gli è arrivato da memoria. In aggiunta, aggiorna il registro *addrRead_reg* memorizzando al suo interno l'indirizzo della cella di memoria che contiene il pixel successivo da trasformare.
- **S_nextPix**: stato in cui il componente effettua opportuni controlli per determinare e salvare in 2 registri dedicati il massimo e il minimo valore dei pixel dell'immagine da elaborare, confrontando progressivamente tra di loro il pixel letto nello stato precedente (il pixel corrente) con il valore del pixel temporaneamente massimo e con il valore del pixel temporaneamente minimo (memorizzati temporaneamente rispettivamente nel registro dedicato al massimo, *maxPix_reg*, e in quello dedicato al minimo, *minPix_reg*). In contemporanea, il componente verifica se ci sono successivi pixel dell'immagine da leggere (rispetto al pixel corrente), cioè viene confrontato l'indirizzo di memoria contenuto nel registro *addrRead_reg* con quello contenuto nel registro *addrWrite_reg*. Se i due indirizzi sono diversi, significa che ci sono ulteriori pixel dell'immagine da leggere, quindi il segnale `o_end0` rimane a 0, e si ritorna allo stato **S_addrPix**. Se, invece, i due indirizzi sono uguali, significa che sono stati letti tutti i pixel dell'immagine da equalizzare: viene alzato ad 1 il segnale `o_end0`, poichè termina la procedura di individuazione e memorizzazione del massimo e del minimo valore dei pixel dell'immagine, e si passa allo stato **S_delta**.
- **S_delta**: stato in cui viene fornito alla RAM l'indirizzo della cella di memoria contenente il primo pixel dell'immagine da equalizzare e, in aggiunta, aggiorna il registro *addrRead_reg* memorizzando al suo interno lo stesso indirizzo. In parallelo, il componente calcola e salva in un registro (*delta_reg*) il valore di $DELTA\_VALUE$.
- **S_pixToConv**: stato in cui il componente legge e salva nel registro *currentPix_reg* il pixel dell'immagine che gli è arrivato da memoria. In contemporanea, calcola il valore di $SHIFT\_LEVEL$ attraverso un algoritmo ad hoc.
- **S_sub**: stato in cui il componente calcola la differenza tra il valore del pixel corrente (letto nello stato precedente) e quello del minimo pixel dell'immagine (contenuto nel registro *minPix_reg*). Il risultato viene memorizzato in un registro (*sub_reg*).
- **S_write**: stato in cui il componente calcola il valore di $TEMP\_PIXEL$ e, in seguito, quello di NEW_PIXEL_VALUE (cioè il pixel equalizzato), che corrisponde al minimo tra 255 e $TEMP\_PIXEL$. Inoltre, alza il segnale `o_we` a 1 (per poter comunicare in scrittura con la memoria) e scrive il valore del pixel equalizzato nella RAM all'indirizzo di memoria pari a quello contenuto nel registro *addrWrite_reg*. In più, aggiorna il registro *addrRead_reg* memorizzando al suo interno l'indirizzo della cella di memoria che contiene il pixel successivo da trasformare. Infine, il componente verifica se ci sono successivi pixel dell'immagine da trasformare (rispetto al pixel corrente), cioè viene confrontato l'indirizzo della cella di memoria contenuto nel registro *addrWrite_reg* con quello che conterrà l'ultimo pixel dell'immagine equalizzata (segnale `addrEnd`). Se i due indirizzi sono diversi, significa che ci sono ulteriori pixel dell'immagine da trasformare, quindi il segnale `o_end1` rimane a 0 e si passa allo stato **S_nextPixWrite**. Se, invece, i due indirizzi sono uguali, significa che sono stati trasformati e scritti in memoria tutti i pixel dell'immagine da equalizzare. Di conseguenza, viene alzato ad 1 il segnale `o_end1` perché termina la procedura di equalizzazione dell'immagine, viene posto ad 1 il segnale `o_done` e si passa allo stato **S_done**.
- **S_nextPixWrite**: stato in cui viene abbassato il segnale `o_we` a 0 per poter comunicare in lettura con la memoria. In parallelo, il componente aggiorna il registro *addrWrite_reg* memorizzando al suo interno l'indirizzo della cella di memoria in cui sarà scritto il prossimo pixel equalizzato.

- **S_addrPixConv**: stato in cui viene fornito alla RAM l'indirizzo della cella di memoria contenente il pixel dell'immagine da equalizzare.
- **S_nextPixConv**: stato in cui il componente legge e salva nel registro *currentPix_reg* il pixel dell'immagine che gli è arrivato da memoria.
- **S_done**: stato in cui il componente attende che il segnale `i_start` venga abbassato a 0, per poter abbassare a sua volta il segnale `o_done` a 0 e tornare nello stato **S_reset**.

2.3 Schema funzionale

2.3.1 Registri

Come si può vedere dallo schema funzionale in figura 6, il *data path* sviluppato è composto da 9 registri che hanno lo scopo di salvare il valore di alcuni importanti segnali:

- **col_reg**: il numero di colonne dell'immagine;
- **row_reg**: il numero di righe dell'immagine;
- **currentPix_reg**: il pixel corrente dell'immagine da equalizzare;
- **maxPix_reg**: il massimo pixel dell'immagine;
- **minPix_reg**: il minimo pixel dell'immagine;
- **delta_reg**: $DELTA\_VALUE$;
- **sub_reg**: la differenza tra il valore del pixel corrente e il valore del minimo pixel dell'immagine;
- **addrWrite_reg**: l'indirizzo della cella di memoria in cui viene scritto il pixel trasformato;
- **addrRead_reg**: l'indirizzo della cella di memoria da cui viene letto il pixel da trasformare.

N.B.: Si è adottata la convenzione che i segnali `o_reg*` rappresentano le uscite dei registri.

2.3.2 Funzionalità in dettaglio

Si prosegue ora con la descrizione del funzionamento dei 2 process presenti nel *data path*:

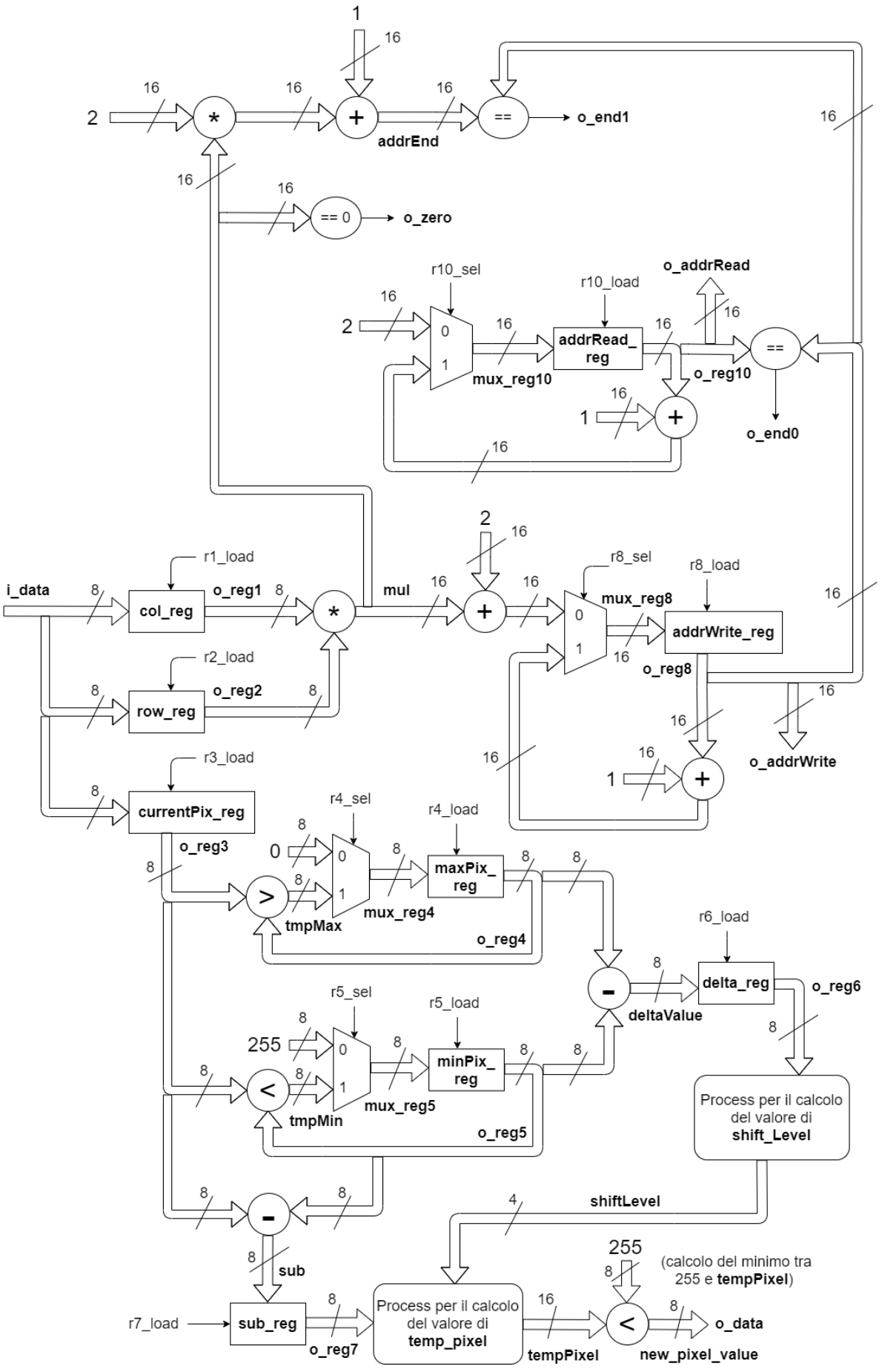
1. Process per il calcolo del valore del segnale $SHIFT\_LEVEL$: calcola un numero intero con valori tra 0 e 8, ricavato da controlli a soglia. Precisamente, controlla il valore di $DELTA\_VALUE$ (con opportuni if statement) e ricava il corrispondente valore di $SHIFT\_LEVEL$ (si veda tabella in figura 5).
2. Process per il calcolo del valore del segnale $TEMP\_PIXEL$: calcola il valore temporaneo del pixel equalizzato, ricavato applicando l'operazione "shift a sinistra" (funzione **shift_left()**) di un numero di livelli pari a $SHIFT\_LEVEL$ sul segnale `o_reg7`.

Figura 5: Tabella calcolo SHIFT_LEVEL

DELTA_VALUE	FLOOR(X)	SHIFT_LEVEL	DELTA_VALUE	FLOOR(X)	SHIFT_LEVEL	DELTA_VALUE	FLOOR(X)	SHIFT_LEVEL
0	0	8	14	3	5	63	6	2
1	1	7	15	4	4	64	6	2
2	1	7	16	4	4	...	...	...
3	2	6	...	...	...	126	6	2
...	...	...	30	4	4	127	7	1
6	2	6	31	5	3	128	7	1
7	3	5	32	5	3	...	...	...
8	3	5	...	...	...	254	7	1
...	...	...	62	5	3	255	8	0

$X = \log_2(DELTA\_VALUE + 1)$

Figura 6: Schema funzionale della parte sequenziale del componente



2.4 Scelte progettuali

La principale scelta progettuale effettuata è stata quella di descrivere il componente con due moduli (entity):

1. Il primo rappresenta la parte sequenziale della macchina (*data path*), che serve per gestire i registri implementati e per eseguire le operazioni per calcolare i pixel dell'immagine equalizzata.
2. Il secondo rappresenta, invece, la FSM che pilota il *data path*, e che analizza i segnali in ingresso ed uscita e lo stato corrente per determinare il prossimo stato in cui evolverà il sistema.

Inoltre, per limitare al minimo indispensabile il tempo di lettura di un pixel dalla memoria RAM, è stato scelto di utilizzare il registro **currentPix_reg** per memorizzare il pixel letto, affinchè venga utilizzato direttamente il valore salvato nel registro per le operazioni successive senza doverlo nuovamente leggere dalla RAM. Per quanto riguarda tutti gli altri registri, essi sono stati implementati così da accorciare i path del *data path*. Queste decisioni hanno permesso di ottimizzare il tempo di elaborazione del componente e, di conseguenza, di poter funzionare con un periodo di clock più stretto.
Infine, per scrivere un codice semplice e di facile manutenzione si è optato di usare il meno possibile funzionalità algoritmiche (process), così da non utilizzare il linguaggio VHDL come se fosse un linguaggio di programmazione software.

---

3 Risultati sperimentali

3.1 Sintesi

3.1.1 Registri sintetizzati

Analizzando il "Vivado Synthesis Report" si nota che sono stati sintetizzati i registri come descritto nel codice. Viene quindi riportata la creazione di 9 registri (per un totale di 92 Flip Flop a singolo bit utilizzati):

Tabella 1: Registri sintetizzati

Num. bit	Num. registri	Contenuto
8	7	Numero di colonne dell'immagine; Numero di righe dell'immagine; Pixel corrente dell'immagine; Massimo pixel; Minimo pixel; Delta_value; Differenza tra il pixel corrente e il minimo pixel dell'immagine.
16	2	Indirizzo della cella di memoria in cui viene scritto il pixel trasformato; Indirizzo della cella di memoria da cui viene letto il pixel da trasformare.
4	1	Stato della FSM. Usa 4 bit dato che si hanno 16 (2^4) stati.

3.1.2 Area occupata

Eseguendo un "Report Utilization", si vede ora l'area occupata dal design sintetizzato.

Tabella 2: Report di utilizzo

Risorsa	Utilizzo	Disponibilità	Utilizzo in %
Look Up Table	234	134600	0.17%
Flip Flop	92	269200	0.03%

Si noti che i valori di utilizzo hanno svariati ordini di grandezza in meno rispetto alla disponibilità della FPGA (è stata utilizzata, come suggerito dalla specifica, la FPGA xc7a200tfbg484-1).

3.1.3 Report di timing

Analizzando il report di timing si può vedere quanto il design sintetizzato sia veloce in un singolo ciclo di clock. Con il clock della specifica di $100ns$ si è ottenuto uno Slack pari a $90.683ns$. Da questo valore, possiamo calcolare il minimo periodo di clock applicabile al componente creato:
$T_{min} = T_{curr} - Slack = 100ns - 90.683ns = 9.317ns$

Quindi, il design creato ha una massima frequenza di clock pari a: $f_{max} = 1/T_{min} \approx 107.3MHz$.

```
Timing Report

Slack (MET) :             90.683ns  (required time - arrival time)
  Source:                 DATAPATH0/o_reg1_reg[3]/C
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@5.000ns period=100.000ns})
  Destination:            DATAPATH0/o_reg8_reg[15]/D
                            (rising edge-triggered cell FDCE clocked by clock  {rise@0.000ns fall@5.000ns period=100.000ns})
  Path Group:             clock
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            100.000ns  (clock rise@100.000ns - clock rise@0.000ns)
  Data Path Delay:        9.199ns  (logic 4.647ns (50.516%)  route 4.552ns (49.484%))
  Logic Levels:           11  (CARRY4=7 LUT4=1 LUT6=3)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.100ns = ( 102.100 - 100.000 )
    Source Clock Delay      (SCD):    2.424ns
    Clock Pessimism Removal (CPR):    0.178ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

Figura 7: Report di timing

3.1.4 Warnings post synthesis

Tutti i warning generati dal tool di sintesi sono stati risolti durante lo sviluppo. Tra questi, figurano anche i warning più "comuni" come quelli per latch inferiti e per segnali presenti nel processo ma non inseriti nella sensitivity list. Pertanto, non è presente alcun warning nella versione finale del componente.

3.2 Simulazioni

Per verificare il corretto funzionamento del componente sintetizzato, dopo averlo testato con il test bench fornito come esempio, sono stati definiti altri test (tra i quali anche quelli che spingono la simulazione verso i corner case) in modo da cercare di massimizzare la copertura di tutti i possibili cammini che la macchina può attraversare durante la computazione.
Di seguito, è fornita una breve descrizione dei test eseguiti e, per quelli più significativi, viene anche mostrato l'effettivo corretto funzionamento grazie allo *screenshot* dell'andamento dei segnali durante la simulazione.

I test bench per la verifica dei corner case sono:

1. **Immagine di dimensione nulla** ($0 \cdot 0$ pixel, $1 \cdot 0$ pixel, $0 \cdot 1$ pixel, ...): l'immagine da equalizzare non ha pixel, cioè non esiste. Il test verifica che il componente alzi ad 1 il segnale `o_zero` (poichè il prodotto tra il numero di colonne e quello delle righe dell'immagine è nullo) e passi direttamente allo stato **S_done** per terminare la computazione, senza eseguire l'equalizzazione.

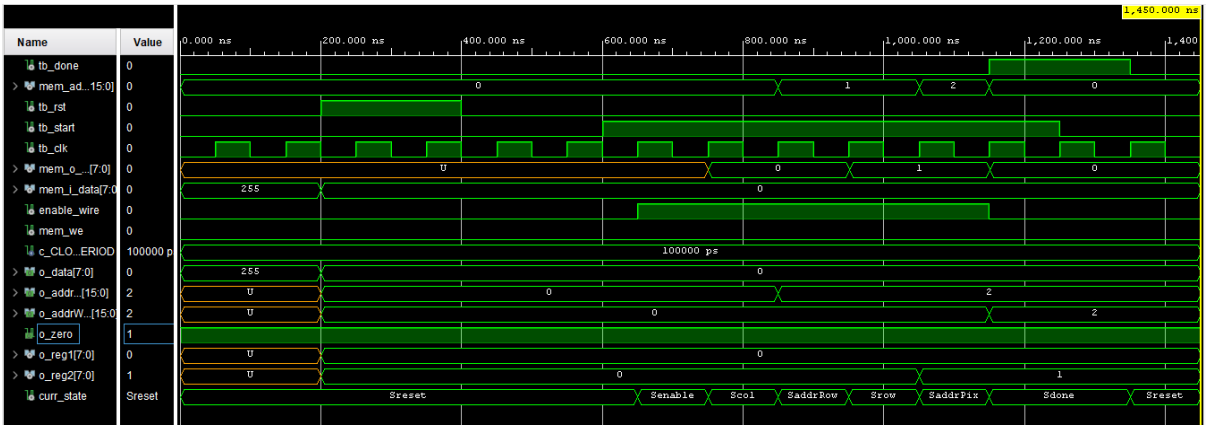


Figura 8: Simulazione con immagine $0 \cdot 1$ pixel

2. **Immagine di dimensione massima** ($128 \cdot 128$ pixel): l'immagine da equalizzare ha dimensione $128 \cdot 128$ pixel. Il test verifica che il componente esegua correttamente sia l'equalizzazione, sia la scrittura di ogni pixel dell'immagine nella memoria RAM.

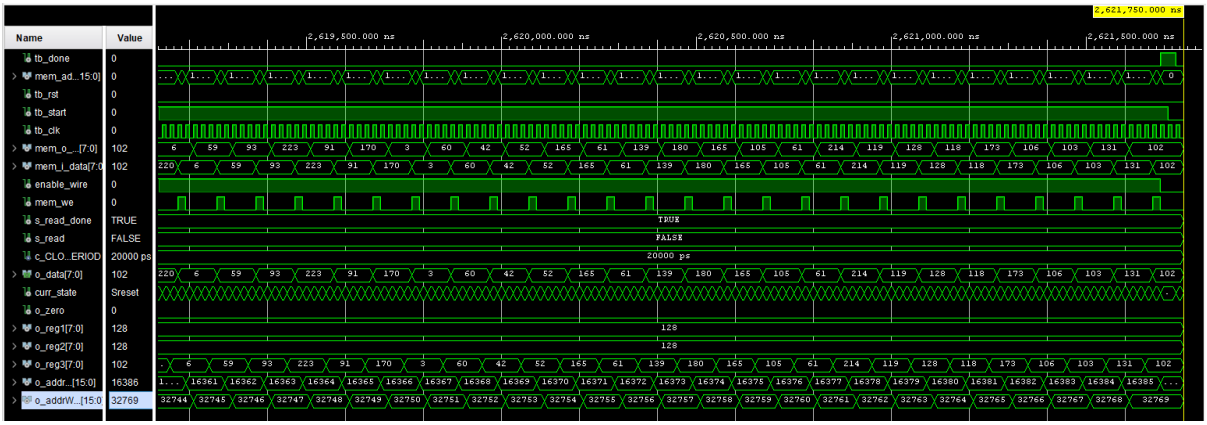


Figura 9: Simulazione con immagine $128 \cdot 128$ pixel (si riporta solo una parte della simulazione perché troppo lunga da mostrare per intero)

I test bench che controllano il corretto funzionamento dei segnali sono:

1. **Equalizzazione immagine con Reset Asincrono**: il test verifica che il trigger asincrono del segnale di reset (`i_rst`) non comprometta la computazione e che questa ricominci dallo stato iniziale della macchina **S_reset**.

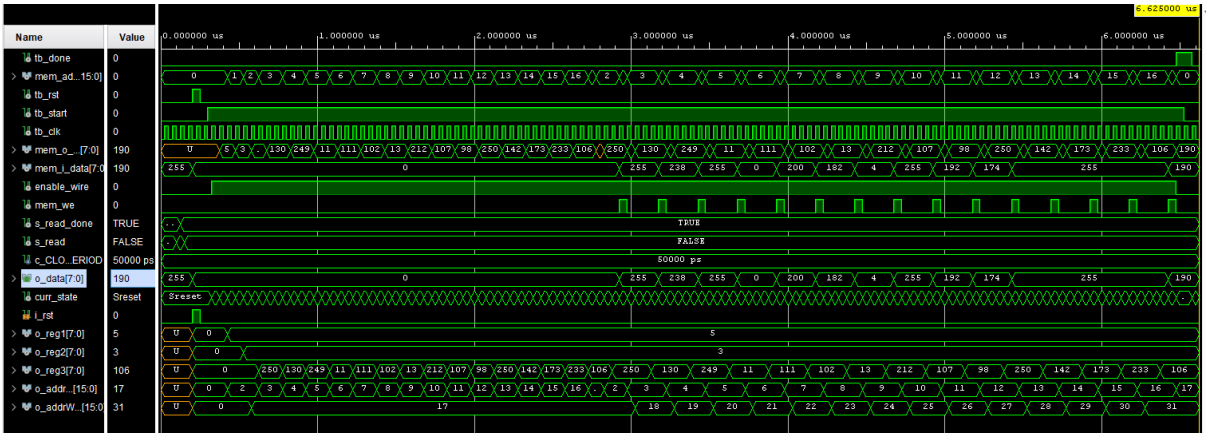


Figura 10: Simulazione con immagine $5 \cdot 3$ pixel con **i_rst** triggerato in modo asincrono durante la computazione

2. **Equalizzazione di più immagini in sequenza**: il test verifica la corretta sincronizzazione dei segnali `i_start`, `i_rst` e `o_done` nel caso di codifica di più immagini di fila.

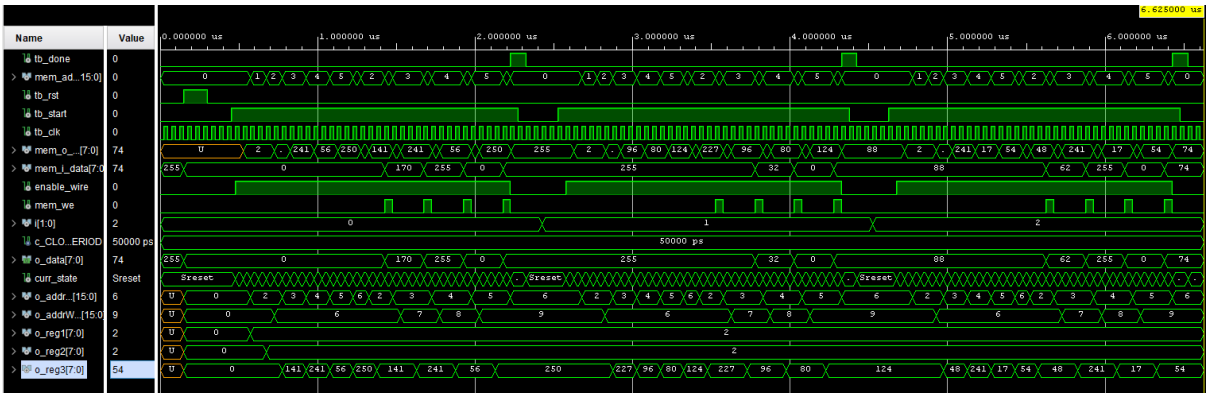


Figura 11: Simulazione con 3 immagini in sequenza di dimensione $2 \cdot 2$ pixel

Inoltre, sono stati creati altri test bench per verificare alcuni casi limite e per controllare la robustezza del componente. Non vengono descritti in dettaglio siccome seguono tutti essenzialmente lo stesso principio di esecuzione.
Si precisa che i test bench sviluppati sono stati testati correttamente anche con un periodo di clock inferiore al minimo richiesto dalla specifica (ad es.: 10ns).

- 3 test bench in cui l'immagine da equalizzare ha dimensione nulla ($2 \cdot 0$ pixel, $0 \cdot 0$ pixel e $1 \cdot 0$ pixel).
- 2 test bench in cui si hanno 3 immagini in sequenza da equalizzare, di dimensione rispettivamente $128 \cdot 128$ pixel, $2 \cdot 0$ pixel e $80 \cdot 83$ pixel. Il primo test controlla il corretto funzionamento del componente nel caso di assenza di segnali di reset asincrono; il secondo esegue lo stesso controllo ma nel caso in cui, durante l'elaborazione, viene mandato un segnale di reset asincrono.
- 2 test bench in cui si hanno 2000 immagini in sequenza da equalizzare, di dimensione casuale con limite massimo $128 \cdot 128$ pixel. Il primo test controlla il corretto funzionamento del componente nel caso di assenza di segnali di reset asincrono; il secondo esegue lo stesso controllo ma nel caso in cui, durante l'elaborazione, viene mandato un segnale di reset asincrono.
- 2 test bench in cui si hanno 10000 immagini in sequenza da equalizzare, di dimensione casuale con limite massimo $16 \cdot 16$ pixel. Il primo test controlla il corretto funzionamento del componente nel

caso di assenza di segnali di reset asincrono; il secondo esegue lo stesso controllo ma nel caso in cui, durante l'elaborazione, viene mandato un segnale di reset asincrono.

- Infine, sono stati definiti una decina di altri test che non verificano casi particolari, ma semplicemente casi normali con immagini di dimensione e valore dei pixel diversi (tra cui anche le 5 immagini di dimensione $4 \cdot 3$ pixel presenti come esempio nella specifica).

---

4 Conclusioni

Per concludere, si è creato un design con le seguenti caratteristiche:

- Funzionante in pre e post-sintesi (*Behavioral* e *Post-Synthesis Functional*).
- Ottimizzato così da ottenere un *Data path delay* minore possibile, il minimo periodo di clock più stretto ed una frequenza massima di clock maggiore. Ciò è stato possibile mantenendo, rispettivamente, i registri **delta_reg** e **sub_reg** e lo stato **S_enable** della FSM. Al contrario, eliminandoli, il *Data path delay* sarebbe stato maggiore, il periodo di clock più largo, la frequenza di clock minore e il numero di LUT maggiore. Così facendo il progetto funzionerebbe comunque, ma avrebbe caratteristiche in parte differenti.
- Una frequenza massima di clock impostabile a $107.3MHz$.
- Un utilizzo di LUT pari al 0.17%.
- Un utilizzo di FF pari al 0.03%.