



Tesina di

Sistemi elettronici embedded

Corso di Laurea in Ingegneria Informatica e Robotica – A.A. 2022-2023

DIPARTIMENTO DI INGEGNERIA

docente

Prof. Pisana PLACIDI

Point operations su immagini tramite FPGA e interfaccia UART

studenti

Alex Ardelean alexnicolae.ardelean@studenti.unipg.it
Billal Nadi billal.nadi@studenti.unipg.it

0. Indice

1	Introduzione	2
1.1	Specifiche di progetto	2
1.2	Interfaccia	2
2	Analisi RTL	5
2.1	UART	5
2.1.1	Ricevitore UART	7
2.1.2	Trasmettitore UART	13
2.2	Filtri	19
2.2.1	Filtro trasparente	20
2.2.2	Filtro negativo	22
2.2.3	Filtro soglia	23
2.2.4	Filtro luminosità	26
2.2.5	Demultiplexer	29
2.2.6	Multiplexer	31
2.2.7	Schema dei filtri	33
2.3	Schema completo	40
3	Simulazione	43
3.1	Simulazione comportamentale	46
3.2	Simulazione post synthesis	48
3.3	Simulazione post implementation	51
3.4	Utilizzo di risorse	53
4	Sperimentazione tramite PySerial	55
5	Conclusione	63
6	Bibliografia	64

1. Introduzione

Il lavoro consiste nella realizzazione di un progetto su FPGA relativo all'applicazione di filtri su immagini. In particolare il dispositivo FPGA dovrà comunicare con un PC tramite un protocollo UART. Il PC dovrà quindi inviare i dati relativi ad una immagine all'FPGA che risponderà con la sua versione filtrata.

Si è inoltre implementata una UI (User Interface) in python che permette di interagire con la scheda garantendo una UX (User Experience) semplificata.

1.1 Specifiche di progetto

Il progetto presenta come specifica l'applicazione di filtri a immagini ad una velocità di 4 Mbit/s=500KB/s, i filtri implementati sono:

- filtro trasparente;
- filtro negativo;
- filtro soglia;
- filtro luminosità;

1.2 Interfaccia

L'interfaccia in ingresso presenta:

- 2 switch per la selezione del filtro da utilizzare con la seguente configurazione:
 - "00" per il filtro trasparente;
 - "01" per il filtro negativo;

- "10" per il filtro soglia;
- "11" per il filtro luminosità;
- 8 switch per la selezione del valore di soglia/luminosità interpretati come intero unsigned;
- linea RsRx per la ricezione di dati seriali;

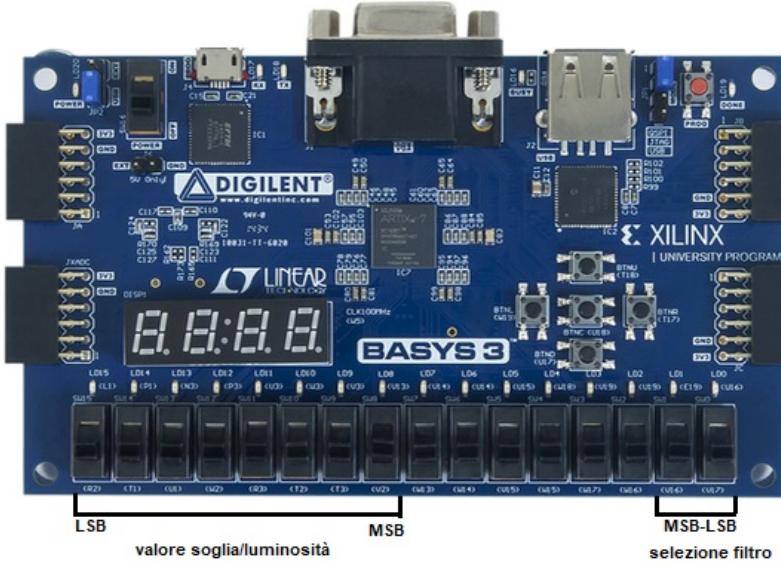


Figura 1.1: Switch utilizzati

L’interfaccia in uscita invece presenta:

- linea RsTx per la trasmissione di dati seriali;

Al fine di configurare correttamente le interfacce di I/O si è utilizzato il file di constraint 1.1.

Source Code 1.1: Basys3_Master.xdc

```

1 ## Clock signal
2 set_property PACKAGE_PIN W5 [get_ports clk]
3 set_property IO_STANDARD LVCMOS33 [get_ports clk]
4 create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports
   ↳ clk]
5
6 ## Switches
7 set_property PACKAGE_PIN V17 [get_ports {switch_selezione[0]}]
8 set_property IO_STANDARD LVCMOS33 [get_ports {switch_selezione[0]}]
9 set_property PACKAGE_PIN V16 [get_ports {switch_selezione[1]}]
```

```

10  set_property IOSTANDARD LVCMOS33 [get_ports {switch_selezione[1]}]
11 set_property PACKAGE_PIN V2 [get_ports {switch_valore[7]}]
12 set_property IOSTANDARD LVCMOS33 [get_ports {switch_valore[7]}]
13 set_property PACKAGE_PIN T3 [get_ports {switch_valore[6]}]
14 set_property IOSTANDARD LVCMOS33 [get_ports {switch_valore[6]}]
15 set_property PACKAGE_PIN T2 [get_ports {switch_valore[5]}]
16 set_property IOSTANDARD LVCMOS33 [get_ports {switch_valore[5]}]
17 set_property PACKAGE_PIN R3 [get_ports {switch_valore[4]}]
18 set_property IOSTANDARD LVCMOS33 [get_ports {switch_valore[4]}]
19 set_property PACKAGE_PIN W2 [get_ports {switch_valore[3]}]
20 set_property IOSTANDARD LVCMOS33 [get_ports {switch_valore[3]}]
21 set_property PACKAGE_PIN U1 [get_ports {switch_valore[2]}]
22 set_property IOSTANDARD LVCMOS33 [get_ports {switch_valore[2]}]
23 set_property PACKAGE_PIN T1 [get_ports {switch_valore[1]}]
24 set_property IOSTANDARD LVCMOS33 [get_ports {switch_valore[1]}]
25 set_property PACKAGE_PIN R2 [get_ports {switch_valore[0]}]
26 set_property IOSTANDARD LVCMOS33 [get_ports {switch_valore[0]}]
27
28 ##USB-RS232 Interface
29 set_property PACKAGE_PIN B18 [get_ports RsRx]
30 set_property IOSTANDARD LVCMOS33 [get_ports RsRx]
31 set_property PACKAGE_PIN A18 [get_ports RsTx]
32 set_property IOSTANDARD LVCMOS33 [get_ports RsTx]

```

2. Analisi RTL

Si è utilizzato un approccio bottom-up, nel quale parti individuali del sistema sono specificate in dettaglio, e poi connesse tra loro in modo da formare componenti più grandi, a loro volta interconnesse fino a realizzare il sistema completo.

2.1 UART

UART è l'acronimo di Universal Asynchronous Receiver / Transmitter, supporta la trasmissione seriale di dati bidirezionali ed è asincrona.

La comunicazione si dice asincrona quando non dipende da un segnale di clock sincronizzato tra i due dispositivi che comunicano (PC ed FPGA). Il sistema UART funziona in full-duplex, ovvero i dati possono essere trasmessi e ricevuti contemporaneamente da entrambi i lati della comunicazione. Utilizza due linee dati chiamate tx (per la trasmissione) e rx (per la ricezione) e una linea di terra per discriminare il segnale digitale e consentire la comunicazione. Lo schema di collegamento è mostrato in figura 2.1.

Poiché non condividono un clock, entrambe le estremità devono trasmettere alla stessa velocità (baud rate) prestabilita per avere la stessa tempistica dei bit. Oltre

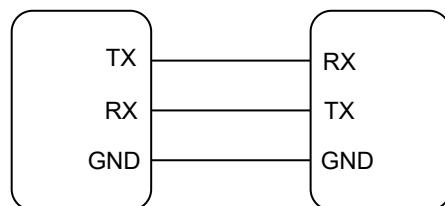


Figura 2.1: Schema di collegamento

ad avere lo stesso baud rate, entrambi i lati di una connessione UART devono anche utilizzare la stessa struttura di frame.

Dato che l'interfaccia UART è asincrona, il trasmettitore deve segnalare che i bit di dati stanno arrivando. Ciò viene realizzato utilizzando il bit di start. Il bit di start è rappresentato da una transizione dallo stato alto a uno stato basso, ed è seguito dai bit di dati utili. Dopo che i bit utili sono finiti, il bit di stop ne indica la fine. Il bit di stop è rappresentato o da una transizione di ritorno allo stato alto o la permanenza allo stato alto. I bit utili sono trasmessi con il bit meno significativo per primo.

Ad esempio per trasmettere la stringa binaria "10011011" corrispondente al numero decimale 155 si dovrà effettuare la trasmissione raffigurata in figura 2.2.

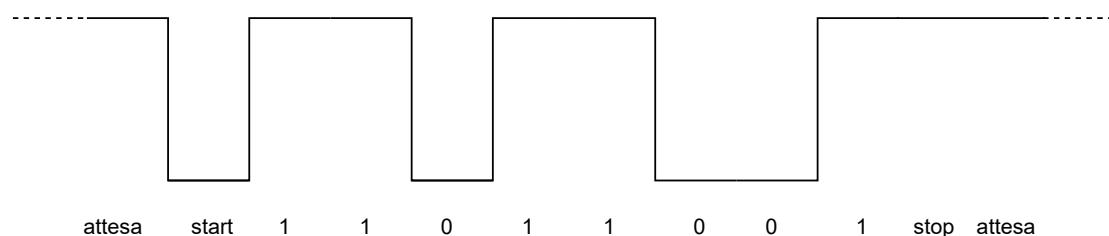


Figura 2.2: Esempio di frame UART

La scheda Basys 3 come specificato in [1] include un bridge USB-UART FTDI FT2232HQ (collegato al connettore J4) che consente di utilizzare il PC per comunicare con la scheda utilizzando i comandi standard della porta COM di Windows. Una volta installati i driver, i comandi di I/O possono essere utilizzati dal PC diretti alla porta COM per produrre dati seriali sui pin B18 e A18 della scheda.

Due LED di stato a bordo della scheda forniscono un feedback visivo sul traffico che scorre attraverso le porte:

- (LD18): LED di trasmissione (TX);
- (LD17): LED di ricezione (RX);

I nomi dei segnali (RX e TX), indicano la direzione dei dati dal punto di vista del DTE (Data Terminal Equipment), in questo caso il PC.

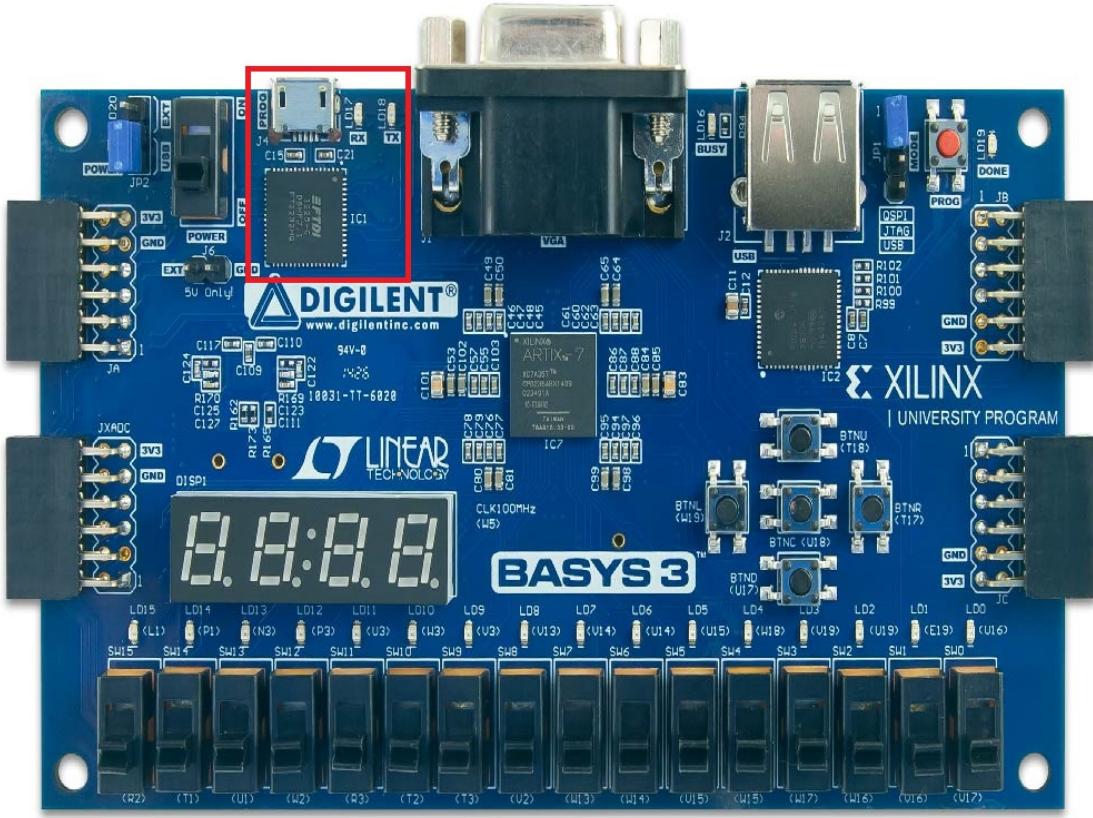


Figura 2.3: Posizione del bridge USB-UART FTDI FT2232HQ sulla scheda Basys 3

2.1.1 Ricevitore UART

Il ricevitore può essere descritto tramite una macchina a stati finiti. In particolare tale macchina presenta quattro stati:

- ATTESA: in attesa del bit di start.
- START: bit di start ricevuto.
- RICEZIONE: ricezione dei bit utili.
- STOP: bit di stop ricevuto.

La macchina viene inizializzata nello stato di ATTESA. Finché in ingresso è presente un 1 (RX=1) la macchina rimane nello stato di ATTESA. La transizione nello stato di START avviene quando si rileva il bit di start, ovvero uno 0 in ingresso (RX=0). Una volta entrato in questo stato la macchina ci rimane per mezzo

periodo di baud al fine di posizionarsi al centro del bit. Trascorso il mezzo periodo di baud si passa nello stato di RICEZIONE, in questo stato viene inizializzato un contatore che viene utilizzato per memorizzare il numero di bit utili ricevuti, ognuno degli 8 bit utili mantiene la macchina nello stato di RICEZIONE per un periodo di tempo pari al periodo baud. Una volta raggiunto il centro dell'ultimo bit utile si passa nello stato di STOP, nel quale si rimane per un ulteriore periodo di baud. Una volta raggiunto il centro del bit di stop si torna in stato di ATTESA.

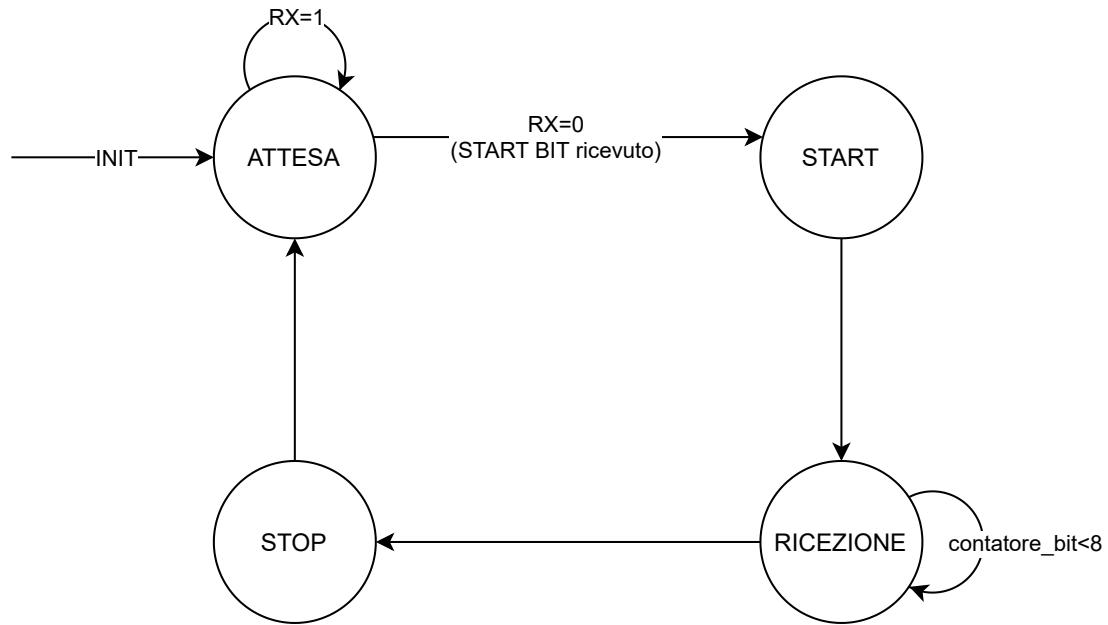


Figura 2.4: Macchina a stati del ricevitore

Il ricevitore presenta la struttura di I/O descritta in figura 2.5.

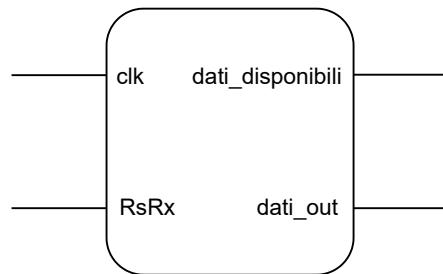


Figura 2.5: Ricevitore UART

La descrizione delle varie porte invece è presente nella tabella 2.1.

Port	Width	Mode	Descrizione
clk	1	in	Clock di sistema
RsRx	1	in	Linea di ricezione seriale
dati_disponibili	1	out	Lo stato HIGH indica che i dati in uscita sono validi. Lo stato LOW indica che i dati in uscita non sono validi.
dati_out	8	out	Ultimo byte ricevuto

Tabella 2.1: Ingressi e uscita del ricevitore

Seguendo l'approccio descritto in [4] si è implementata la macchina a stati raffigurata nella figura 2.4 e precedentemente descritta. Il codice di tale macchina è il seguente:

Source Code 2.1: ricevitore_uart.vhd

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity ricevitore_uart is
5 port(
6   -- Input
7   clk      : in std_logic;
8   RsRx    : in std_logic;
9   -- Output
10  dati_out  : out std_logic_vector(7 downto 0);
11  dati_disponibili : out std_logic
12 );
13 end ricevitore_uart;
14
15 architecture Behavioral of ricevitore_uart is
16 constant f_clock : integer := 100000000;      -- 100.000.000 Hz = 100 MHz
17 constant f_baud : integer := 4000000;        -- 4.000.000 bit/s
18 constant campioni_per_baud : integer := f_clock/f_baud; -- 25 cicli di clock
-- ogni bit
19
20 type stato_t is (attesa, start, ricezione, stop);
21 signal stato : stato_t := attesa;
22
23 signal buffer_ricezione : std_logic_vector(7 downto 0) := (others => '1');
24 begin
25   -- Gestisce lo stato della macchina
26   process(clk)
27     variable contatore_campioni : integer range 0 to campioni_per_baud-1 := 0;
28     variable contatore_bit : integer range 0 to 7 := 0;
29   begin

```

```

30  if(rising_edge(clk)) then
31    case stato is
32      when attesa =>
33          -- Un 1 sulla linea di ricezione indica nessun dato presente
34          if(RsRx = '1') then
35              stato <= attesa;
36          -- Uno 0 sulla linea di ricezione indica l'inizio della ricezione
37          else
38              stato <= start;
39              contatore_campioni := 1;
40          end if;
41      when start =>
42          -- Centro del bit di start non ancora raggiunto
43          -- NOTA: il -1 e' necessario perche' il contatore parte da 0
44          if(contatore_campioni < campioni_per_baud/2-1) then
45              stato <= start;
46              contatore_campioni := contatore_campioni + 1;
47          -- Centro del bit di start raggiunto
48          else
49              stato <= ricezione;
50              contatore_campioni := 0;
51          end if;
52      when ricezione =>
53          -- Centro del bit utile non ancora raggiunto
54          if(contatore_campioni < campioni_per_baud-1) then
55              stato <= ricezione;
56              contatore_campioni := contatore_campioni + 1;
57          -- Centro del bit utile raggiunto
58          else
59              contatore_campioni := 0;
60              -- Ci sono ancora bit da ricevere
61              if(contatore_bit < 7) then
62                  buffer_ricezione <= RsRx & buffer_ricezione(7 downto 1);
63                  stato <= ricezione;
64                  contatore_bit := contatore_bit + 1;
65                  -- Ricevuti 8 bit
66              else
67                  buffer_ricezione <= RsRx & buffer_ricezione(7 downto 1);
68                  stato <= stop;
69                  contatore_bit := 0;
70              end if;
71          end if;
72      when stop =>
73          -- Centro del bit di stop non ancora raggiunto
74          if(contatore_campioni < campioni_per_baud-1) then
75              stato <= stop;
76              contatore_campioni := contatore_campioni + 1;
77          -- Centro del bit di stop raggiunto
78          else

```

```

79      stato <= attesa;
80      contatore_campioni := 0;
81      end if;
82    end case;
83  end if;
84 end process;

85
86 -- Gestisce l'output della macchina
87 process(stato)
88 begin
89  case stato is
90    -- Finche' non riceve il bit di stop i dati in uscita non sono validi
91    when attesa =>
92      dati_disponibili <= '0';
93    when start =>
94      dati_disponibili <= '0';
95    when ricezione =>
96      dati_disponibili <= '0';
97    -- Bit di stop ricevuto => dati in uscita validi
98    when stop =>
99      dati_disponibili <= '1';
100   end case;
101  dati_out <= buffer_ricezione;
102 end process;
103 end Behavioral;

```

Si è quindi utilizzata una descrizione di tipo comportamentale, in particolare si sono utilizzati due **process**:

- Il primo process (riga 26) gestisce lo stato della macchina e presenta nella sensitivity list solamente il segnale **clk**. Ad ogni fronte di salita del clock tale processo valuta lo stato corrente della macchina, memorizzato nel segnale **stato**, e decide in quale stato transitare, in particolare:
 - nello stato di **attesa**: si rimane in tale stato finché non si incontra uno 0 sulla linea di ricezione, in tal caso si transita nello stato di **start**.
 - nello stato di **start**: si rimane in tale stato per mezzo periodo di baud al fine di posizionarsi al centro del bit, si transita infine nello stato di **ricezione**. Da notare che per posizionarsi al centro del bit di start si è fatto uso di un contatore, in particolare si è utilizzata la variabile **contatore_campioni** definita nella parte dichiarativa del processo. Tale variabile viene utilizzata per “contare” il numero di cicli di clock per cui si è rimasti in un particolare stato. In questo caso, dato che ci si voleva posizionare al centro del bit di start, e la durata per cui ogni bit è presente sulla linea di ricezione è pari **f_clock/f_baud** ovve-

ro `campioni_per_baud`, si è atteso per `campioni_per_baud/2` cicli di clock.

- nello stato di `ricezione`: per posizionarsi al centro di ogni bit si attende un numero di cicli di clock pari a `campioni_per_baud`, per fare ciò si utilizza la variabile `contatore_campioni`. Una volta posizionati al centro del bit si fa “scorrere verso destra” il `buffer_ricezione` e si aggiunge nella posizione MSB di tale buffer il bit presente sulla linea di ricezione. Una volta posizionati tutti gli 8 bit utili nel buffer di ricezione si transita nello stato di `stop`.
- nello stato di `stop`: si rimane in tale stato per un periodo di baud al fine di posizionarsi al centro del bit, si transita infine nello stato di `attesa`.
- Il secondo processo (riga 87) gestisce l’output della macchina e presenta nella sensitivity list il segnale `stato`, ad ogni variazione di tale segnale esso decide il valore da assegnare all’uscita, in particolare:
 - negli stati di `attesa`, `start` e `ricezione` viene segnalato che i dati in uscita non sono disponibili tramite il segnale `dati_disponibili`.
 - nello stato di `stop` si trasmette in uscita il buffer di ricezione e si segnala la disponibilità dei dati.

Si vuole ora esaminare la descrizione RTL generata dal codice 2.1. L’RTL completo è presente in figura 2.6.

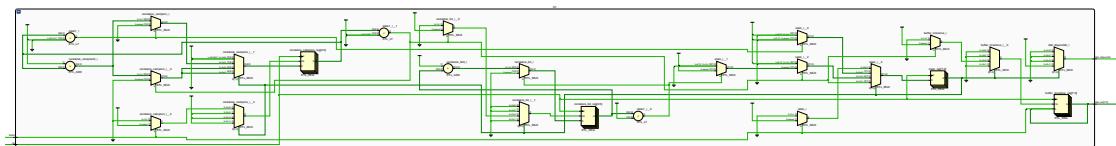


Figura 2.6: RTL completo del ricevitore

Si evidenzia subito la presenza di 4 registri, ognuno dei quali utilizzati come dispositivi di archiviazione sequenziali. In particolare il loro scopo è di memorizzare:

- il numero di bit ricevuti (figura 2.7a);
- il numero di cicli di clock (figura 2.7b);
- il buffer di ricezione (figura 2.7c);
- lo stato della macchina (figura 2.7d);

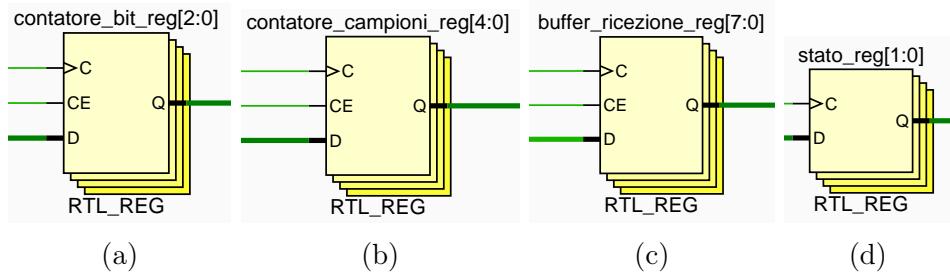


Figura 2.7: Registri presenti nell'RTL del ricevitore

I primi tre presentano load sincrono, mentre l'ultimo no.

Si nota inoltre la presenza di multiplexer, operatori aritmetici (+) e operatori relazionali (<) utilizzati per la logica combinatoria.

Si tratta quindi di un circuito sequenziale in cui gli elementi di memoria sono costituiti da 4 registri mentre la logica combinatoria, che risulta particolarmente intricata, è descritta tramite multiplexer e operatori.

2.1.2 Trasmettitore UART

Anche il trasmettitore può essere descritto tramite una macchina a stati finiti. In particolare tale macchina presenta quattro stati:

- ATTESA: in attesa del segnale di abilitazione;
- START: invio del bit di start;
- TRASMISSIONE: invio dei bit utili;
- STOP: invio del bit di stop;

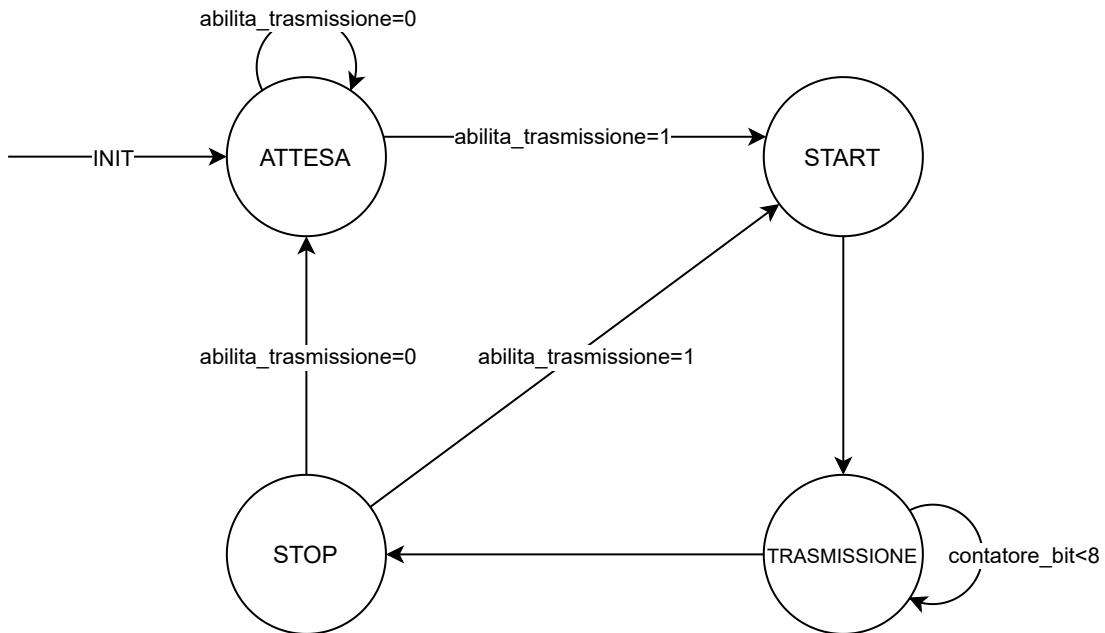


Figura 2.8: Macchina a stati del trasmettitore

Il trasmettitore presenta la struttura di I/O descritta in figura 2.9.

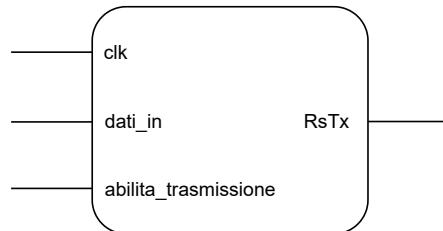


Figura 2.9: Trasmettitore UART

La descrizione delle varie porte invece è presente nella tabella 2.2.

La macchina viene inizializzata nello stato di ATTESA e rimane in tale stato finché non riceve il segnale di abilitazione. La transizione nello stato di START avviene quando si rileva un '1' sul segnale di abilitazione. Una volta entrata in questo stato ci rimane per un periodo di baud nel quale l'uscita viene posta a 0. Trascorso il periodo di baud si passa nello stato di TRASMISSIONE, nel quale i bit presenti nel buffer, a partire dal LSB, vengono assegnati all'uscita ciascuno per un periodo di baud. Una volta inviato l'ultimo bit utile si passa nello stato di STOP, nel quale l'uscita viene posta a 1. Da questo stato è possibile transitare nello stato di

Port	Width	Mode	Descrizione
clk	1	in	Clock di sistema
dati_in	8	in	Linea di ricezione
abilita_trasmissione	1	in	Lo stato HIGH indica che i dati in ingresso sono validi e devono essere trasmessi. Lo stato LOW indica che i dati in ingresso non sono validi.
RsTx	1	out	Uscita seriale

Tabella 2.2: Ingressi e uscita del trasmettitore

ATTESA, se il segnale di abilitazione è basso, oppure nello stato di START, se il segnale di abilitazione è alto.

Seguendo l'approccio illustrato in [4] si è implementata la macchina a stati raffigurata nella figura 2.8 e precedentemente descritta. Il codice di tale macchina è il seguente:

Source Code 2.2: trasmettitore_uart.vhd

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity trasmettitore_uart is
5 port(
6   -- Input
7   clk      : in std_logic;
8   abilita_trasmissione : in std_logic;
9   dati_in    : in std_logic_vector(7 downto 0);
10  -- Output
11  RsTx      : out std_logic
12 );
13 end trasmettitore_uart;
14
15 architecture Behavioral of trasmettitore_uart is
16 constant f_clock : integer := 100000000;      -- 100.000.000 Hz = 100 MHz
17 constant f_baud : integer := 4000000;        -- 4.000.000 bit/s
18 constant campioni_per_baud : integer := f_clock/f_baud; -- 25 cicli di clock
-- ogni bit
19
20 type stato_t is (attesa, start, trasmissione, stop);
21 signal stato : stato_t := attesa;
22
23 signal buffer_trasmissione : std_logic_vector(7 downto 0) := (others => '0');
24 begin
25   -- Gestisce lo stato della macchina
26 process(clk)

```

```

27  variable contatore_campioni : integer range 0 to campioni_per_baud-1 := 0;
28  variable contatore_bit : integer range 0 to 7 := 0;
29 begin
30  if(rising_edge(clk)) then
31    case stato is
32      when attesa =>
33        -- Attende segnale di abilitazione della trasmissione
34        if(abilita_trasmissione = '0') then
35          stato <= attesa;
36        -- Segnale di abilitazione della trasmissione ricevuto
37        else
38          stato <= start;
39          contatore_campioni := 1;
40        end if;
41      when start =>
42        -- Trasmette il bit di start
43        if(contatore_campioni < campioni_per_baud-1) then
44          stato <= start;
45          contatore_campioni := contatore_campioni + 1;
46          buffer_trasmissione <= dati_in;
47        -- Bit di start trasmesso
48        else
49          stato <= trasmissione;
50          contatore_campioni := 0;
51        end if;
52      when trasmissione =>
53        -- Trasmette il bit utile
54        if(contatore_campioni < campioni_per_baud-1) then
55          stato <= trasmissione;
56          contatore_campioni := contatore_campioni + 1;
57        -- Controlla se ci sono ancora bit da trasmettere
58        else
59          contatore_campioni := 0;
60          -- Ci sono ancora bit da trasmettere
61          if(contatore_bit < 7) then
62            stato <= trasmissione;
63            contatore_bit := contatore_bit + 1;
64            buffer_trasmissione <= '0' & buffer_trasmissione(7 downto 1);
65          -- Trasmessi 8 bit
66          else
67            stato <= stop;
68            contatore_bit := 0;
69          end if;
70        end if;
71      when stop =>
72        -- Trasmette il bit di stop
73        if(contatore_campioni < campioni_per_baud-1) then
74          stato <= stop;
75          contatore_campioni := contatore_campioni + 1;

```

```

76      -- Bit di stop trasmesso
77  else
78      -- Segnale di abilitazione non ricevuto
79  if(abilita_trasmissione = '0') then
80      stato <= attesa;
81      contatore_campioni := 0;
82      -- Segnale di abilitazione della trasmissione ricevuto
83  else
84      stato <= start;
85      contatore_campioni := 0;
86  end if;
87  end if;
88  end case;
89 end if;
90 end process;
91
92 -- Gestisce l'output della macchina
93 process(clk)
94 begin
95  if(rising_edge(clk)) then
96      case stato is
97          -- In attesa imposta la linea alta
98  when attesa =>
99      RsTx <= '1';
100     -- Trasmette il bit di start '0'
101  when start =>
102      RsTx <= '0';
103     -- Trasmette i dati utili
104  when trasmissione =>
105      RsTx <= buffer_trasmissione(0);
106     -- Trasmette il bit di stop '1'
107  when stop =>
108      RsTx <= '1';
109  end case;
110 end if;
111 end process;
112 end Behavioral;

```

Si è quindi utilizzata una descrizione di tipo comportamentale, in particolare si sono utilizzati due **process**:

- Il primo process (riga 26) gestisce lo stato della macchina e presenta nella sensitivity list solamente il segnale **clk**. Ad ogni fronte di salita del clock tale processo valuta lo stato corrente della macchina, memorizzato nel segnale **stato**, e decide in quale stato transitare, in particolare:
 - nello stato di **attesa**: si rimane in tale stato finché il segnale di abilitazione non viene posto alto **abilita_trasmissione = '1'**. Una volta

abilitata la trasmissione si transita nello stato di **start**.

- nello stato di **start**: si rimane in tale stato per un periodo di baud al fine di inviare il bit di start '0', si transita infine nello stato di **trasmissione**.
- nello stato di **trasmissione**: ogni periodo di baud si fa “scorrere verso destra” il buffer in modo da avere in posizione 0 il bit da trasmettere. Una volta passati gli otto periodi di baud si passa in stato di **stop**.
- nello stato di **stop**: si rimane in tale stato per un periodo di baud al fine di trasmettere il bit di stop, si transita infine nello stato di **attesa** o **start** in base al segnale di abilitazione della trasmissione.
- Il secondo processo (riga 92) gestisce l’output della macchina e presenta nella sensitivity list il segnale **stato**, ad ogni variazione di tale segnale esso decide il valore da assegnare all’uscita, in particolare:
 - nello stato di **attesa**: si imposta la linea di uscita alta.
 - nello stato di **start**: si trasmette il bit di start '0'.
 - nello stato di **trasmissione**: si trasmette il bit LSB del buffer di trasmissione.
 - nello stato di **stop**: si trasmette il bit di stop '1'.

Si vuole ora esaminare la descrizione RTL generata dal codice 2.2. L’RTL completo è presente in figura 2.10.

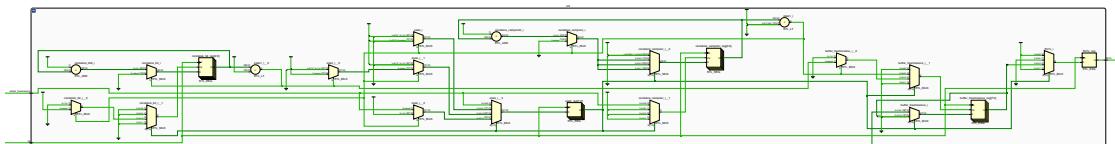


Figura 2.10: RTL completo del trasmettitore

Anche in questo caso si evidenzia la presenza di 4 registri, ognuno dei quali utilizzati come dispositivi di archiviazione sequenziali. In particolare il loro scopo è di memorizzare:

- il numero di bit ricevuti (figura 2.11a);
- il numero di cicli di clock (figura 2.11b);
- il buffer di trasmissione (figura 2.11c);
- lo stato della macchina (figura 2.11d);

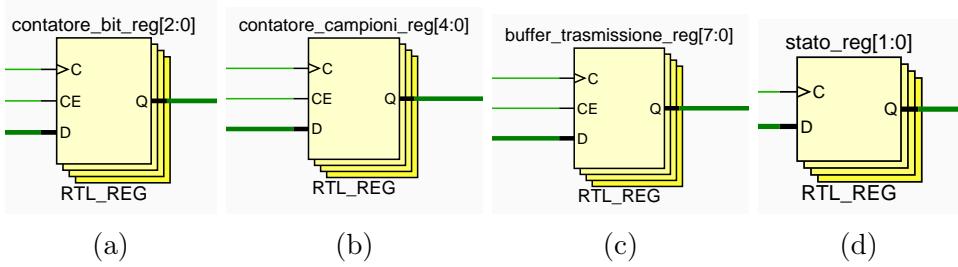


Figura 2.11: Registri presenti nell'RTL del trasmettitore

Anche in questo caso i primi tre presentano load sincrono, mentre l'ultimo no.

Si nota inoltre la presenza di multiplexer, operatori aritmetici (+) e operatori relazionali (<) utilizzati per la logica combinatoria.

Come nel caso del ricevitore si tratta quindi di un circuito sequenziale in cui gli elementi di memoria sono costituiti da 4 registri mentre la logica combinatoria, che risulta particolarmente intricata, è descritta tramite multiplexer e operatori.

2.2 Filtri

Proseguendo con l'approccio bottom-up si descriveranno in dettaglio i filtri implementati per poi collegare tutti i componenti e ottenere il sistema completo.

Si prenderanno in considerazione immagini bitmap, ovvero rappresentate da una matrice di punti (o pixel). Nelle immagini a colori, viene memorizzato solitamente il livello di intensità dei colori fondamentali, nel modello di colore RGB, uno dei più usati, sono tre: rosso, verde e blu. Il numero (detto anche “profondità”) di colori o di livelli possibili dipende dal massimo numero di combinazioni permesse dalla quantità di bit utilizzata per ognuno di questi dati.

Tutti i filtri implementati assumono di ottenere in ingresso una sequenza di pixel, relativi ad una immagine bitmap RGB con profondità pari a 8 bit, le cui componenti transitano attraverso il filtro nella sequenza rosso, verde e blu. I filtri inoltre non si preoccupano di determinare l'inizio o la fine dell'immagine ma solo di effettuare delle operazioni sui pixel. In particolare le operazioni implementate si definiscono “point operation”, dato che modificano solamente il valore del pixel e non la dimensione, la geometria o la struttura locale dell'immagine. Tali tipi di operazioni possono quindi essere descritte tramite una funzione $f(pixel)$. La funzione f prende come parametro solamente il valore del pixel e risulta quindi indipendente dalla coordinata del pixel nell'immagine.

2.2.1 Filtro trasparente

Il filtro trasparente è il più semplice dei quattro implementati e il suo scopo è quello di far passare l'immagine inalterata. La funzione realizzata è quindi l'identità:

$$f(pixel) = pixel \quad (2.1)$$

Il filtro presenta la struttura di I/O descritta in figura 2.12.

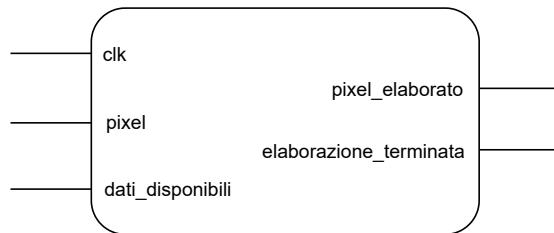


Figura 2.12: Filtro trasparente

La descrizione delle varie porte invece è presente nella tabella 2.3.

Port	Width	Mode	Descrizione
clk	1	in	Clock di sistema
pixel	8	in	Componente (R, G o B) del pixel nel range [0,255]
dati_disponibili	1	in	Lo stato HIGH indica che i dati in ingresso sono validi e devono essere elaborati. Lo stato LOW indica che i dati in ingresso non sono validi.
pixel_elaborato	8	out	Valore della componente (R, G o B) elaborata
elaborazione_terminata	1	out	Lo stato HIGH indica che i dati in uscita sono validi. Lo stato LOW indica che i dati in uscita non sono validi.

Tabella 2.3: Ingressi e uscita del filtro trasparente

Per realizzare tale filtro si è utilizzata una descrizione comportamentale, il codice è il seguente:

Source Code 2.3: filtro_tr.vhd

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity filtro_tr is
5 port(
6   -- Inputs
7   clk : in std_logic;
8   pixel : in std_logic_vector(7 downto 0); -- Valore del pixel [0,255]
9   dati_disponibili : in std_logic; -- '1' indica dati pixel disponibili
10  -- Output
11   pixel_elaborato : out std_logic_vector(7 downto 0); -- Valore del pixel
12   elaborazione_terminata : out std_logic
13 );
14 end filtro_tr;
15
16 architecture Behavioral of filtro_tr is
17
18 begin
19 process(clk)
20 begin
21 if(rising_edge(clk)) then
22   if(dati_disponibili = '1') then
23     elaborazione_terminata <= '1';
24   else
25     elaborazione_terminata <= '0';
26   end if;
27   pixel_elaborato <= pixel;
28 end if;
29 end process;
30 end Behavioral;

```

Si vuole ora esaminare la descrizione RTL generata dal codice 2.3. L'RTL completo è presente in figura 2.13.

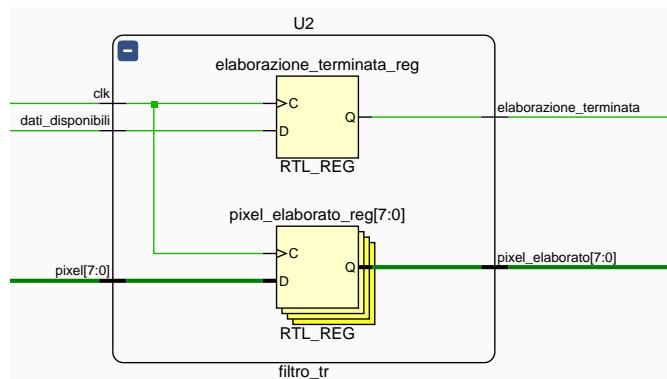


Figura 2.13: RTL completo del filtro trasparente

Dallo schema RTL è possibile evidenziare la presenza di due registri, lo scopo di tali registri è quello di memorizzare i segnali `dati_disponibili` e `pixel` al fine di renderli disponibili in uscita sul fronte di salita del clock.

2.2.2 Filtro negativo

La funzione realizzata dal filtro negativo è la seguente:

$$f(pixel) = 255 - pixel \quad (2.2)$$

La struttura di I/O e la descrizione delle porte è identica a quella del filtro trasparente presente in figura 2.12 e tabella 2.3.

Per realizzare tale filtro si è utilizzata una descrizione comportamentale, il codice è il seguente:

Source Code 2.4: filtro_negativo.vhd

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity filtro_negativo is
6   port(
7     -- Inputs
8     clk : in std_logic;
9     pixel : in std_logic_vector(7 downto 0); -- Valore del pixel [0,255]
10    dati_disponibili : in std_logic; -- '1' indica dati pixel disponibili
11    -- Output
12    pixel_elaborato : out std_logic_vector(7 downto 0); -- Valore del pixel
13    elaborazione_terminata : out std_logic
14  );
15 end filtro_negativo;
16
17 architecture Behavioral of filtro_negativo is
18
19 begin
20   -- pixel_elaborato <= pixel;
21   process(clk)
22   begin
23     if(rising_edge(clk)) then
24       if(dati_disponibili = '1') then
25         elaborazione_terminata <= '1';
26       else
27         elaborazione_terminata <= '0';
28       end if;
29       pixel_elaborato <= std_logic_vector(255-unsigned(pixel));

```

```

30      end if;
31  end process;
32 end Behavioral;

```

In questo caso si è fatto uso della libreria `IEEE.NUMERIC_STD` al fine di effettuare l'operazione tra interi senza segno presente a riga 29.

Si vuole ora esaminare la descrizione RTL generata dal codice 2.4. L'RTL completo è presente in figura 2.14.

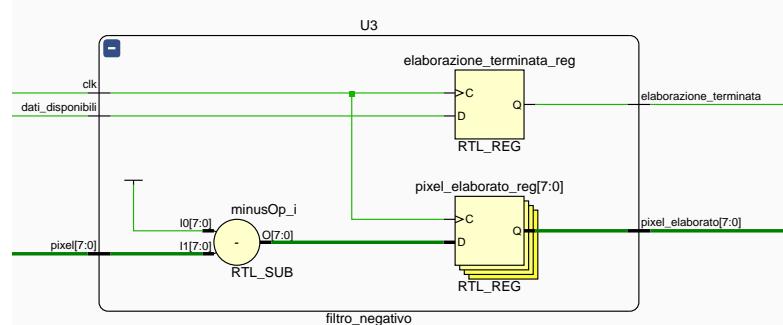


Figura 2.14: RTL completo del filtro negativo

Dallo schema RTL si evidenziano due registri utilizzati in maniera analoga al caso del filtro trasparente. Rispetto al filtro trasparente però in questo caso è presente un operatore aritmetico necessario per calcolare il valore negativo del pixel. Il risultato di tale operazione viene poi mandata in ingresso al registro contenente il valore del pixel elaborato da inviare in uscita.

2.2.3 Filtro soglia

La funzione realizzata dal filtro soglia è la seguente:

$$f(pixel) = \begin{cases} 0 & \text{se } pixel \leq soglia \\ pixel & \text{se } pixel > soglia \end{cases} \quad (2.3)$$

dove *soglia* è una costante $\in [0, 255]$.

Il filtro presenta la struttura di I/O descritta in figura 2.15.

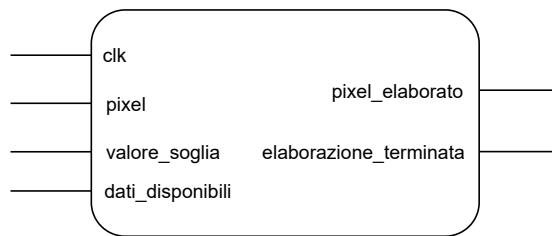


Figura 2.15: Filtro soglia

La descrizione delle varie porte invece è presente nella tabella 2.4.

Port	Width	Mode	Descrizione
clk	1	in	Clock di sistema
pixel	8	in	Componente (R, G o B) del pixel nel range [0,255]
dati_disponibili	1	in	Lo stato HIGH indica che i dati in ingresso sono validi e devono essere elaborati. Lo stato LOW indica che i dati in ingresso non sono validi.
valore_soglia	8	in	Soglia da applicare al pixel espressa nel range [0,255]
pixel_elaborato	8	out	Valore della componente (R, G o B) elaborata
elaborazione_terminata	1	out	Lo stato HIGH indica che i dati in uscita sono validi. Lo stato LOW indica che i dati in uscita non sono validi.

Tabella 2.4: Ingressi e uscita del filtro soglia

Per realizzare tale filtro si è utilizzata una descrizione comportamentale, il codice è il seguente:

Source Code 2.5: filtro_soglia.vhd

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity filtro_soglia is
5 port(
6   -- Inputs
7   clk : in std_logic;
8   pixel : in std_logic_vector(7 downto 0);    -- Valore del pixel [0,255]
9   dati_disponibili : in std_logic;           -- '1' indica dati pixel disponibili
10  valore_soglia : in std_logic_vector(7 downto 0); -- Soglia

```

```

11      -- Output
12      pixel_elaborato : out std_logic_vector(7 downto 0);      -- Valore del pixel
13      → elaborato [0,255]
14      elaborazione_terminata : out std_logic
15  );
16  end filtro_soglia;
17
18 architecture Behavioral of filtro_soglia is
19
20 begin
21 process(clk)
22 begin
23 if(rising_edge(clk)) then
24     if(dati_disponibili = '1') then
25         if(pixel <= valore_soglia) then
26             pixel_elaborato <= (others => '0');
27             elaborazione_terminata <= '1';
28         else
29             pixel_elaborato <= pixel;
30             elaborazione_terminata <= '0';
31         end if;
32     else
33         elaborazione_terminata <= '0';
34     end if;
35 end process;
36 end Behavioral;

```

Si vuole ora esaminare la descrizione RTL generata dal codice 2.5. L'RTL completo è presente in figura 2.16.

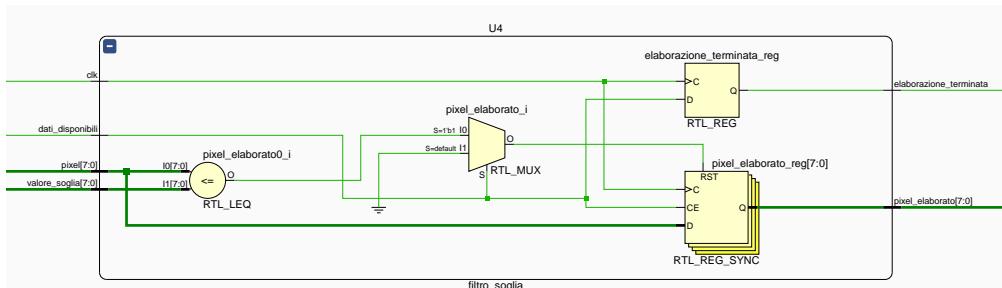


Figura 2.16: RTL completo del filtro soglia

Anche in questo caso si nota la presenza dei due registri con la stessa funzione dei casi precedenti. Ciò che distingue tale schema è invece la presenza di un operatore relazione (\leq) e di un multiplexer. L'uscita del multiplexer è collegata al reset del registro contenente il pixel elaborato, tale reset ha la priorità sul load sincrono. Il

reset viene quindi abilitato quando il risultato dell'operazione relazionale è positivo, ovvero si ha $\text{pixel} \leq \text{valore_soglia}$. Quando ciò avviene il registro viene resettato e il valore memorizzato viene posto a 0. Quando invece il risultato dell'operazione relazionale è negativo ($\text{pixel} > \text{valore_soglia}$) il reset non viene abilitato, si fa quindi uso del load parallelo sincrono per caricare il valore `pixel` nel registro.

2.2.4 Filtro luminosità

La funzione realizzata dal filtro luminosità è la seguente:

$$f(\text{pixel}) = \begin{cases} 255 & \text{se } \text{pixel} \times \text{luminosita} > 255 \\ \text{pixel} \times \text{luminosita} & \text{altrimenti} \end{cases} \quad (2.4)$$

dove *luminosita* è una costante $\in [0, 2]$.

Il filtro presenta la struttura di I/O descritta in figura 2.17.

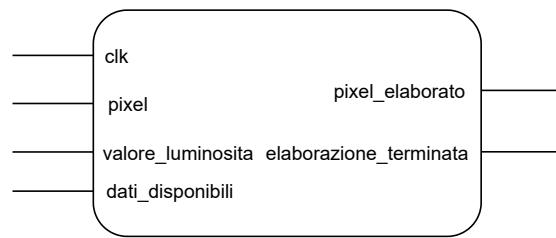


Figura 2.17: Filtro Luminosità

La descrizione delle varie porte invece è presente nella tabella 2.5.

Port	Width	Mode	Descrizione
clk	1	in	Clock di sistema
pixel	8	in	Componente (R, G o B) del pixel nel range [0,255]
dati_disponibili	1	in	Lo stato HIGH indica che i dati in ingresso sono validi e devono essere elaborati. Lo stato LOW indica che i dati in ingresso non sono validi.
valore_luminosita	8	in	Luminosità da applicare al pixel espressa nel range [0,255]
pixel_elaborato	8	out	Valore della componente (R, G o B) elaborata
elaborazione_terminata	1	out	Lo stato HIGH indica che i dati in uscita sono validi. Lo stato LOW indica che i dati in uscita non sono validi.

Tabella 2.5: Ingressi e uscita del filtro luminosità

Per realizzare tale filtro si è utilizzata una descrizione comportamentale, il codice è il seguente:

Source Code 2.6: filtro_luminosita.vhd

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity filtro_luminosita is
6 port(
7   -- Inputs
8   clk : in std_logic;
9   pixel : in std_logic_vector(7 downto 0);      -- Valore del pixel [0,255]
10  dati_disponibili : in std_logic;           -- '1' indica dati pixel disponibili
11  valore_luminosita : in std_logic_vector(7 downto 0); -- Luminosita'
12  -- Output
13  pixel_elaborato : out std_logic_vector(7 downto 0);    -- Valore del pixel
14  -- elaborato [0,255]
15  elaborazione_terminata : out std_logic
16 );
17 end filtro_luminosita;
18
19 architecture Behavioral of filtro_luminosita is
20
21 begin
  process(clk)

```

```

22     variable temp : std_logic_vector(15 downto 0);
23 begin
24     if(rising_edge(clk)) then
25         temp := std_logic_vector(unsigned(pixel) * unsigned(valore_luminosita));
26         if(dati_disponibili = '1') then
27             if(temp(15) = '1') then
28                 pixel_elaborato <= (others => '1');
29                 elaborazione_terminata <= '1';
30             else
31                 pixel_elaborato <= temp(14 downto 7);
32                 elaborazione_terminata <= '1';
33             end if;
34         else
35             elaborazione_terminata <= '0';
36         end if;
37     end if;
38 end process;
39 end Behavioral;

```

Si vuole ora esaminare la descrizione RTL generata dal codice 2.6. L'RTL completo è presente in figura 2.18.

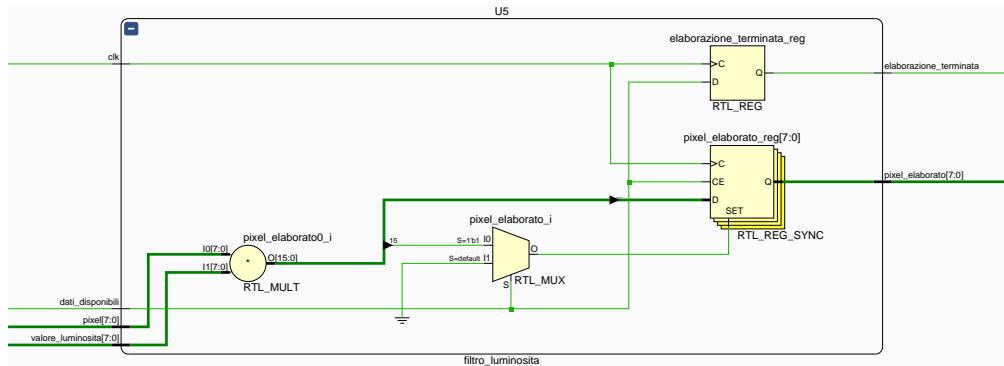


Figura 2.18: RTL completo del filtro luminosità

Anche in questo caso si nota la presenza dei due registri con la stessa funzione dei casi precedenti. Ciò che distingue tale schema è invece la presenza di un operatore aritmetico (\times) e di un multiplexer. L'uscita del multiplexer è collegata al set del registro contenente il pixel elaborato, tale set ha la priorità sul load sincrono. Il set viene quindi abilitato quando il MSB del risultato dell'operazione aritmetica è '1', ovvero quando $\text{pixel} \times \text{valore_luminosita} > 32767$, il valore memorizzato dal registro risulta quindi essere 255. Quando ciò non avviene invece ($\text{pixel} \times \text{valore_luminosita} < 32767$) il set è disabilitato e grazie al load parallelo si memorizza il risultato dell'operazione aritmetica considerando il range 14 `downto` 7 (il primo bit avenire indice 0).

Tale schema quindi non fa altro che assegnare al registro il valore $pixel \times \frac{valore_luminosita}{128}$ se tale valore è < 255 altrimenti gli assegna 255. Da notare che l'operazione di divisione per 128 è stata effettuata scartando i primi 7 LSB, mentre il controllo sul valore superiore a 255 è stato effettuato controllando il MSB. In questo modo si è implementata la funzione 2.4, infatti il valore per cui viene moltiplicato il pixel di fatto risulta essere $\frac{valore_luminosita}{128} \in [0, 2]$.

2.2.5 Demultiplexer

Al fine di indirizzare i segnali `pixel` e `dati_disponibili` verso il filtro corretto, ovvero quello selezionato tramite il segnale `selezione` collegato ai due switch V17 e V16, si è fatto uso di due demultiplexer.

I due demultiplexer sono stati implementati tramite una descrizione dataflow e il codice è il seguente:

Source Code 2.7: `demux_1to4_1bit.vhd`

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity demux_1to4_1bit is
5 port(
6   -- Input
7   ingresso : in std_logic;  -- Ingresso
8   selezione : in std_logic_vector(1 downto 0); -- Seleziona il segnale
9   -- Output
10  tr : out std_logic;      -- Uscita verso il filtro trasparente
11  neg : out std_logic;     -- Uscita verso il filtro negativo
12  soglia : out std_logic;  -- Uscita verso il filtro soglia
13  lum : out std_logic;    -- Uscita verso il filtro di luminosita'
14 );
15 end demux_1to4_1bit;
16
17 architecture Dataflow of demux_1to4_1bit is
18
19 begin
20   tr <= ingresso when selezione="00" else '0';
21   neg <= ingresso when selezione="01" else '0';
22   soglia <= ingresso when selezione="10" else '0';
23   lum <= ingresso when selezione="11" else '0';
24 end Dataflow;
```

Source Code 2.8: `demux_1to4_8bit.vhd`

```

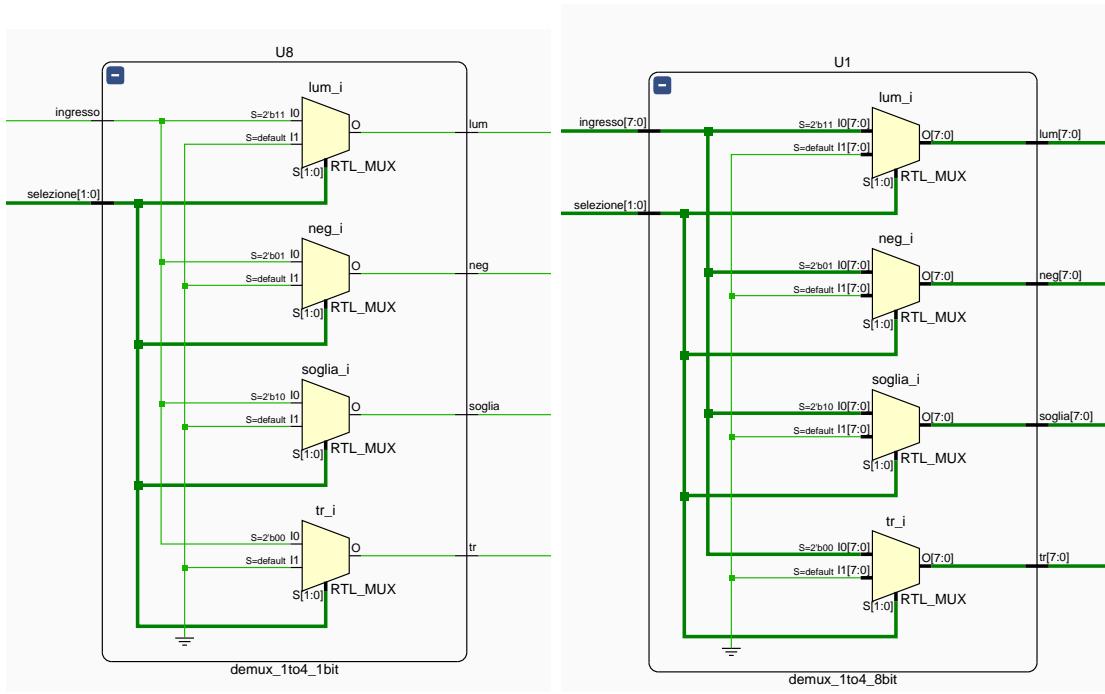
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity demux_1to4_8bit is
```

```

5   port(
6     -- Input
7     ingresso : in std_logic_vector(7 downto 0);    -- Ingresso
8     selezione : in std_logic_vector(1 downto 0); -- Seleziona il segnale
9     -- Output
10    tr : out std_logic_vector(7 downto 0);      -- Uscita verso il filtro
11    ↪ trasparente
12    neg : out std_logic_vector(7 downto 0);    -- Uscita verso il filtro negativo
13    soglia : out std_logic_vector(7 downto 0);  -- Uscita verso il filtro soglia
14    lum : out std_logic_vector(7 downto 0)      -- Usicta verso il filtro di
15    ↪ luminosita'
16  );
17 end demux_1to4_8bit;
18
19 architecture rtl of demux_1to4_8bit is
20
21 begin
22   tr <= ingresso when selezione="00" else (others => '0');
23   neg <= ingresso when selezione="01" else (others => '0');
24   soglia <= ingresso when selezione="10" else (others => '0');
25   lum <= ingresso when selezione="11" else (others => '0');
26 end rtl;

```

Si vuole ora esaminare la descrizione RTL generata dal codice 2.7 e 2.8. L'RTL completo è presente in figura 2.19.



(a) RTL completo del demultiplexer 4-1 a 1 bit
 (b) RTL completo del demultiplexer 4-1 a 8 bit

Figura 2.19: RTL del demultiplexer

Entrambi i demultiplexer sono rappresentati tramite 4 multiplexer, ognuno dei quali ha come selezione il segnale di selezione del demultiplexer e come ingressi l’ingresso del demultiplexer e la costante 0. Le uscite dei multiplexer sono invece collegate individualmente alle 4 uscite del demultiplexer. In questo modo è possibile “dirigere” il segnale sull’uscita desiderata, mentre nelle altre uscite è presente uno 0.

2.2.6 Multiplexer

Al fine di indirizzare i segnali `pixel_elaborato` e `elaborazione_terminata` provenienti dai 4 filtri verso il trasmettitore si fa uso di due multiplexer.

I due multiplexer sono stati implementati tramite una descrizione dataflow e il codice è il seguente:

Source Code 2.9: mux_4to1_1bit.vhd

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
```

```

4  entity mux_4to1_1bit is
5    port(
6      -- Input
7      selezione : in std_logic_vector(1 downto 0); -- Seleziona il segnale
8      tr : in std_logic;
9      neg : in std_logic;
10     soglia : in std_logic;
11     lum : in std_logic;
12      -- Output
13     uscita : out std_logic
14   );
15 end mux_4to1_1bit;
16
17 architecture Dataflow of mux_4to1_1bit is
18 begin
19   with selezione select
20     uscita <=  tr when "00",
21       neg when "01",
22       soglia when "10",
23       lum when others;
24 end Dataflow;

```

Source Code 2.10: mux_4to1_8bit.vhd

```

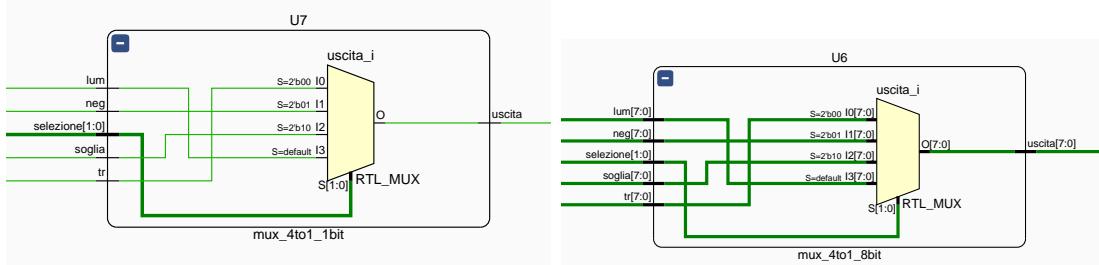
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity mux_4to1_8bit is
5   port(
6     -- Input
7     selezione : in std_logic_vector(1 downto 0); -- Seleziona il segnale
8     tr : in std_logic_vector(7 downto 0);        -- Ingresso proveniente dal filtro
→   trasparente
9     neg : in std_logic_vector(7 downto 0);        -- Ingresso proveniente dal filtro
→   negativo
10    soglia : in std_logic_vector(7 downto 0);      -- Ingresso proveniente dal
→   filtro soglia
11    lum : in std_logic_vector(7 downto 0);        -- Ingresso proveniente dal filtro
→   di luminosita'
12      -- Output
13    uscita : out std_logic_vector(7 downto 0)      -- Uscita
14  );
15 end mux_4to1_8bit;
16
17 architecture Dataflow of mux_4to1_8bit is
18 begin
19   with selezione select
20     uscita <=  tr when "00",
21       neg when "01",
22       soglia when "10",

```

```

23      lum when others;
24 end Dataflow;
```

Si vuole ora esaminare la descrizione RTL generata dal codice 2.9 e 2.10. L'RTL completo è presente in figura 2.20.



(a) RTL completo del multiplexer 1-4 a 1 bit (b) RTL completo del multiplexer 1-4 a 8 bit

Figura 2.20: RTL del demultiplexer

Lo schema RTL dei multiplexer risulta estremamente semplice, essi sono stati infatti descritti tramite un multiplexer.

2.2.7 Schema dei filtri

Si vuole ora ottenere uno schema completo dei filtri in modo da poter essere inseriti facilmente nel sistema completo. Per fare ciò si è utilizzata la seguente descrizione Structural:

Source Code 2.11: filtri.vhd

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity filtri is
5 port(
6   -- Input
7   clk : in std_logic;
8   dati_ricevitore : in std_logic_vector(7 downto 0);
9   dati_ricevitore_disponibili : in std_logic;
10  switch_valore : in std_logic_vector(7 downto 0); -- Controllano la soglia o
11  ↵ la luminosita'
12  switch_selezione : in std_logic_vector(1 downto 0); -- Selezionano il filtro
13  -- Output
14  dati_trasmettitore : out std_logic_vector(7 downto 0);
15  dati_trasmettitore_disponibili : out std_logic
16 );
```

```

16  end filtr;
17
18 architecture Structural of filtr is
19 component demux_1to4_8bit
20 port(
21   -- Input
22   ingresso : in std_logic_vector(7 downto 0); -- Ingresso
23   selezione : in std_logic_vector(1 downto 0); -- Seleziona il segnale
24   -- Output
25   tr : out std_logic_vector(7 downto 0); -- Uscita verso il filtro
26   ↳ trasparente
27   neg : out std_logic_vector(7 downto 0); -- Uscita verso il filtro negativo
28   soglia : out std_logic_vector(7 downto 0); -- Uscita verso il filtro
29   ↳ soglia
30   lum : out std_logic_vector(7 downto 0) -- Usicta verso il filtro di
31   ↳ luminosita'
32 );
33 end component;
34
35 component demux_1to4_1bit
36 port(
37   -- Input
38   ingresso : in std_logic; -- Ingresso
39   selezione : in std_logic_vector(1 downto 0); -- Seleziona il segnale
40   -- Output
41   tr : out std_logic; -- Uscita verso il filtro trasparente
42   neg : out std_logic; -- Uscita verso il filtro negativo
43   soglia : out std_logic; -- Uscita verso il filtro soglia
44   lum : out std_logic -- Usicta verso il filtro di luminosita'
45 );
46 end component;
47
48 component mux_4to1_8bit
49 port(
50   -- Input
51   selezione : in std_logic_vector(1 downto 0); -- Seleziona il segnale
52   tr : in std_logic_vector(7 downto 0); -- Ingresso proveniente dal filtro
53   ↳ trasparente
54   neg : in std_logic_vector(7 downto 0); -- Ingresso proveniente dal filtro
55   ↳ negativo
56   soglia : in std_logic_vector(7 downto 0); -- Ingresso proveniente dal
57   ↳ filtro soglia
58   lum : in std_logic_vector(7 downto 0); -- Ingresso proveniente dal filtro
59   ↳ di luminosita'
60   -- Output
61   uscita : out std_logic_vector(7 downto 0) -- Uscita
62 );
63 end component;

```

```

58 component filtro_tr
59   port(
60     -- Inputs
61     clk : in std_logic;
62     pixel : in std_logic_vector(7 downto 0); -- Valore del pixel [0,255]
63     dati_disponibili : in std_logic; -- '1' indica dati pixel disponibili
64     -- Output
65     pixel_elaborato : out std_logic_vector(7 downto 0); -- Valore del pixel
66     → elaborato [0,255]
67     elaborazione_terminata : out std_logic
68   );
69 end component;
70
70 component filtro_negativo
71   port(
72     -- Inputs
73     clk : in std_logic;
74     pixel : in std_logic_vector(7 downto 0); -- Valore del pixel [0,255]
75     dati_disponibili : in std_logic; -- '1' indica dati pixel disponibili
76     -- Output
77     pixel_elaborato : out std_logic_vector(7 downto 0); -- Valore del pixel
78     → elaborato [0,255]
79     elaborazione_terminata : out std_logic
80   );
81 end component;
82
82 component filtro_soglia
83   port(
84     -- Inputs
85     clk : in std_logic;
86     pixel : in std_logic_vector(7 downto 0); -- Valore del pixel [0,255]
87     dati_disponibili : in std_logic; -- '1' indica dati pixel disponibili
88     valore_soglia : in std_logic_vector(7 downto 0); -- Soglia
89     -- Output
90     pixel_elaborato : out std_logic_vector(7 downto 0); -- Valore del pixel
91     → elaborato [0,255]
92     elaborazione_terminata : out std_logic
93   );
94 end component;
95
95 component filtro_luminosita
96   port(
97     -- Inputs
98     clk : in std_logic;
99     pixel : in std_logic_vector(7 downto 0); -- Valore del pixel [0,255]
100    dati_disponibili : in std_logic; -- '1' indica dati pixel disponibili
101    valore_luminosita : in std_logic_vector(7 downto 0); -- Soglia
102    -- Output

```

```

103      pixel_elaborato : out std_logic_vector(7 downto 0);    -- Valore del pixel
→   elaborato [0,255]
104      elaborazione_terminata : out std_logic
105  );
106 end component;
107
108 component mux_4to1_1bit
109  port(
110    -- Input
111    selezione : in std_logic_vector(1 downto 0); -- Seleziona il segnale
112    tr : in std_logic;
113    neg : in std_logic;
114    soglia : in std_logic;
115    lum : in std_logic;
116    -- Output
117    uscita : out std_logic
118  );
119 end component;
120
121 signal ingresso_tr : std_logic_vector(7 downto 0);
122 signal ingresso_neg : std_logic_vector(7 downto 0);
123 signal ingresso_soglia : std_logic_vector(7 downto 0);
124 signal ingresso_lum : std_logic_vector(7 downto 0);
125
126 signal dd_tr : std_logic;
127 signal dd_neg : std_logic;
128 signal dd_soglia : std_logic;
129 signal dd_lum : std_logic;
130
131 signal uscita_tr : std_logic_vector(7 downto 0);
132 signal uscita_neg : std_logic_vector(7 downto 0);
133 signal uscita_soglia : std_logic_vector(7 downto 0);
134 signal uscita_lum : std_logic_vector(7 downto 0);
135
136 signal et_tr : std_logic;
137 signal et_neg : std_logic;
138 signal et_soglia : std_logic;
139 signal et_lum : std_logic;
140 begin
141 U1: demux_1to4_8bit
142 port map(
143    -- Input
144    ingresso => dati_ricevitore,
145    selezione => switch_selezione,
146    -- Output
147    tr => ingresso_tr,
148    neg => ingresso_neg,
149    soglia => ingresso_soglia,
150    lum => ingresso_lum

```

```

151     );
152
153 U8: demux_1to4_1bit
154 port map(
155     -- Input
156     ingresso => dati_ricevitore_disponibili,
157     selezione => switch_selezione,
158     -- Output
159     tr => dd_tr,
160     neg => dd_neg,
161     soglia => dd_soglia,
162     lum => dd_lum
163 );
164
165 U2: filtro_tr
166 port map(
167     -- Inputs
168     clk => clk,
169     pixel => ingresso_tr,
170     dati_disponibili => dd_tr,
171     -- Output
172     pixel_elaborato => uscita_tr,
173     elaborazione_terminata => et_tr
174 );
175
176 U3: filtro_negativo
177 port map(
178     -- Inputs
179     clk => clk,
180     pixel => ingresso_neg,
181     dati_disponibili => dd_neg,
182     -- Output
183     pixel_elaborato => uscita_neg,
184     elaborazione_terminata => et_neg
185 );
186
187 U4: filtro_soglia
188 port map(
189     -- Inputs
190     clk => clk,
191     pixel => ingresso_soglia,
192     dati_disponibili => dd_soglia,
193     valore_soglia => switch_valore,
194     -- Output
195     pixel_elaborato => uscita_soglia,
196     elaborazione_terminata => et_soglia
197 );
198
199 U5: filtro_luminosita

```

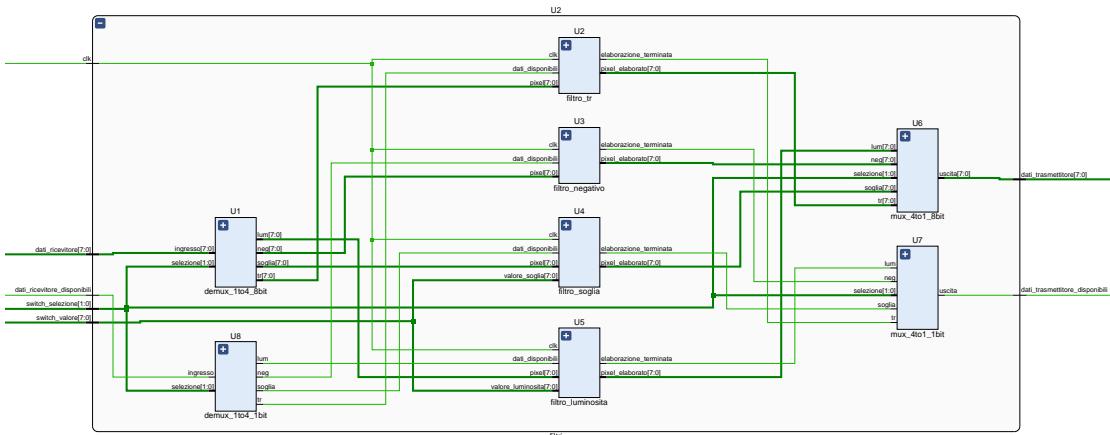
```

200  port map(
201    -- Inputs
202    clk => clk,
203    pixel => ingresso_lum,
204    dati_disponibili => dd_lum,
205    valore_luminosita => switch_valore,
206    -- Output
207    pixel_elaborato => uscita_lum,
208    elaborazione_terminata => et_lum
209  );
210
211 U6: mux_4to1_8bit
212 port map(
213   -- Input
214   selezione => switch_selezione,
215   tr => uscita_tr,
216   neg => uscita_neg,
217   soglia => uscita_soglia,
218   lum => uscita_lum,
219   -- Output
220   uscita => dati_trasmettitore
221 );
222
223 U7: mux_4to1_1bit
224 port map(
225   -- Input
226   selezione => switch_selezione,
227   tr => et_tr,
228   neg => et_neg,
229   soglia => et_soglia,
230   lum => et_lum,
231   -- Output
232   uscita => dati_trasmettitore_disponibili
233 );
234
235 end Structural;

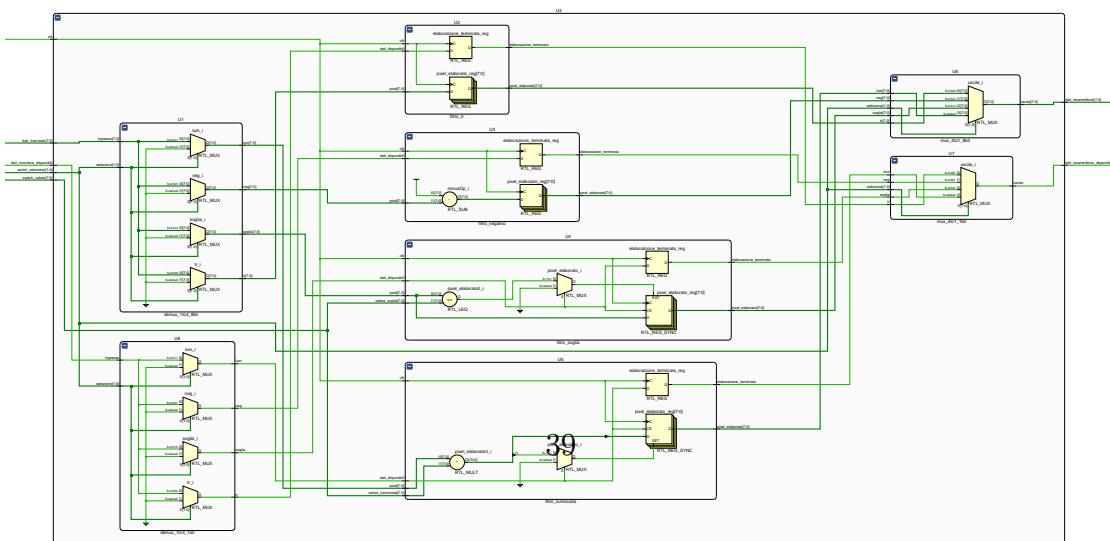
```

Port	Width	Mode	Descrizione
clk	1	in	Clock di sistema
dati_ricevitore	8	in	Dati provenienti dal ricevitore, da interpretare come interi unsigned
dati_ricevitore_disponibili	1	in	Lo stato HIGH indica che i dati in del ricevitore sono validi. Lo stato LOW indica che i dati del ricevitore non sono validi.
switch_selezione	2	in	Selezionano il filtro desiderato
switch_valore	8	in	Valore di soglia/luminosità
dati_trasmettitore	8	out	Valore filtrato
dati_trasmettitore_disponibili	1	out	Lo stato HIGH indica che i dati in uscita sono validi. Lo stato LOW indica che i dati in uscita non sono validi.

Tabella 2.6: Ingressi e uscita dei filtri



(a) RTL dello schema dei filtri



(b) RTL espanso dello schema dei filtri

Figura 2.21: RTL dei filtri

2.3 Schema completo

Ora non resta che collegare il ricevitore agli ingressi dei filtri e il trasmettitore all'uscita dei filtri. Per fare ciò si è utilizzata la seguente descrizione Structural:

Source Code 2.12: filtro_immagine.vhd

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity filtro_immagine is
5     port(
6         -- Input
7         clk : in std_logic;          -- Clock di sistema
8         RsRx : in std_logic;        -- Linea in ingresso seriale
9         switch_valore : in std_logic_vector(7 downto 0); -- Controllano la soglia o
→      la luminosita'
10        switch_selezione : in std_logic_vector(1 downto 0); -- Selezionano il filtro
11        -- Output
12        RsTx : out std_logic;       -- Linea in uscita seriale
13    );
14 end filtro_immagine;
15
16 architecture Structural of filtro_immagine is
17 component ricevitore_uart
18     port(
19         -- Input
20         clk      : in std_logic;
21         RsRx    : in std_logic;
22         -- Output
23         dati_out : out std_logic_vector(7 downto 0);
24         dati_disponibili : out std_logic
25     );
26 end component;
27
28 component filtri
29     port(
30         -- Input
31         clk : in std_logic;
32         dati_ricevitore : in std_logic_vector(7 downto 0);
33         dati_ricevitore_disponibili : in std_logic;
34         switch_valore : in std_logic_vector(7 downto 0); -- Controllano la soglia o
→      la luminosita'
35         switch_selezione : in std_logic_vector(1 downto 0); -- Selezionano il filtro
36         -- Output
37         dati_trasmettitore : out std_logic_vector(7 downto 0);
38         dati_trasmettitore_disponibili : out std_logic
39     );
```

```

40  end component;
41
42 component trasmittitore_uart
43 port(
44    -- Input
45    clk      : in std_logic;
46    abilita_trasmissione : in std_logic;
47    dati_in      : in std_logic_vector(7 downto 0);
48    -- Output
49    RsTx      : out std_logic
50 );
51 end component;
52
53 signal dati_ricevitore : std_logic_vector(7 downto 0);
54 signal ricevitore_disponibile : std_logic;
55
56 signal dati_trasmittitore : std_logic_vector(7 downto 0);
57 signal dati_elaborati_disponibili : std_logic;
58 begin
59 U1: ricevitore_uart
60 port map(
61    -- Input
62    clk => clk,
63    RsRx => RsRx,
64    -- Output
65    dati_out => dati_ricevitore,
66    dati_disponibili => ricevitore_disponibile
67 );
68
69 U2: filtri
70 port map(
71    -- Input
72    clk => clk,
73    dati_ricevitore => dati_ricevitore,
74    dati_ricevitore_disponibili => ricevitore_disponibile,
75    switch_valore => switch_valore,
76    switch_selezione => switch_selezione,
77    -- Output
78    dati_trasmittitore => dati_trasmittitore,
79    dati_trasmittitore_disponibili => dati_elaborati_disponibili
80 );
81
82 U3: trasmittitore_uart
83 port map(
84    -- Input
85    clk => clk,
86    abilita_trasmissione => dati_elaborati_disponibili,
87    dati_in => dati_trasmittitore,
88    -- Output

```

```
89      RsTx => RsTx  
90  );  
91 end Structural
```

La descrizione RTL del codice 2.12 è presente nelle figure 2.22, 2.23 e 2.24 con un grado crescente di dettaglio.

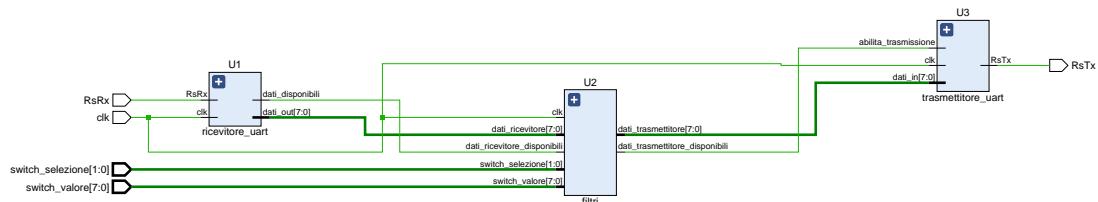


Figura 2.22: RTL dello schema completo

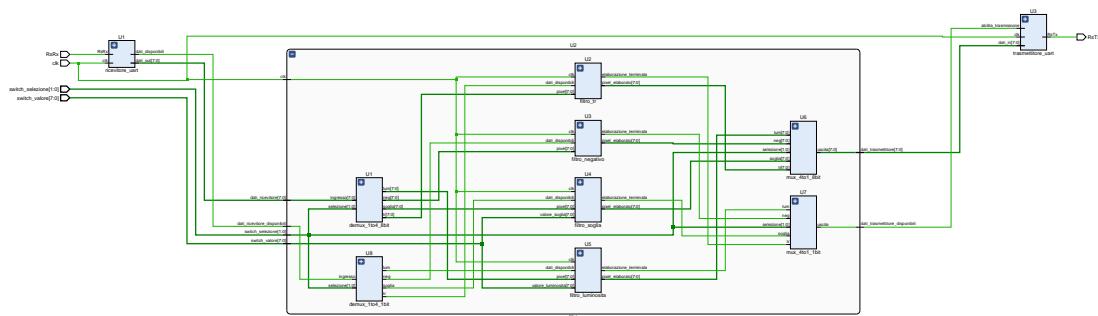


Figura 2.23: RTL dello schema completo

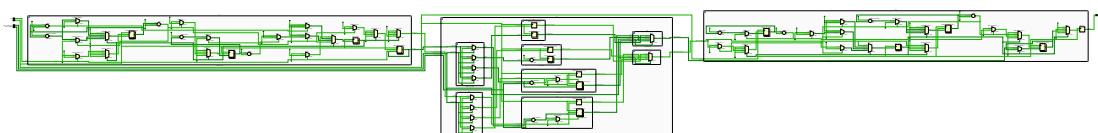


Figura 2.24: RTL dello schema completo

3. Simulazione

Al fine di valutare la correttezza di quanto descritto precedentemente, prima di effettuare il test sulla scheda, sono state svolte alcune simulazioni. In particolare, grazie al testebench 3.1 è stato possibile verificarne il corretto comportamento al variare dell'input.

Source Code 3.1: filtro_immagine_tb.vhd

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity filtro_immagine_tb is
6 end filtro_immagine_tb;
7
8 architecture Behavioral of filtro_immagine_tb is
9 component filtro_immagine
10 port(
11     -- Input
12     clk : in std_logic;          -- Clock di sistema
13     RsRx : in std_logic;        -- Linea in ingresso seriale
14     switch_valore : in std_logic_vector(7 downto 0); -- Controllano la soglia o
→      la luminosita'
15     switch_selezione : in std_logic_vector(1 downto 0); -- Selezionano il filtro
16     -- Output
17     RsTx : out std_logic;       -- Linea in uscita seriale
18 );
19 end component;
20
21 signal clk_tb : std_logic := '0';
22 signal RsRx_tb : std_logic := '0';
23 signal switch_valore_tb : std_logic_vector(7 downto 0) := (others => '0');
24 signal switch_selezione_tb : std_logic_vector(1 downto 0) := (others => '0');
25 signal RsTx_tb : std_logic := '0';
```

```

26
27 constant f_clock : integer := 100000000;      -- 100.000.000 Hz = 100 MHz
28 constant f_baud : integer := 4000000;        -- 4.000.000 bit/s
29 constant campioni_per_baud : integer := f_clock/f_baud; -- 25 cicli di clock
-- ogni bit
30 constant p_clock : time := 1sec/f_clock;
31 constant p_baud : time := 1sec/f_baud;
32 begin
33   U1: filtro_immagine
34   port map(
35     -- Input
36     clk => clk_tb,
37     RsRx => RsRx_tb,
38     switch_valore => switch_valore_tb,
39     switch_selezione => switch_selezione_tb,
40     -- Output
41     RsTx => RsTx_tb
42   );
43
44   gen_clock: process
45   begin
46     wait for p_clock/2;
47     clk_tb <= '0';
48     wait for p_clock/2;
49     clk_tb <= '1';
50   end process;
51
52   sim_rsrx: process
53   variable byte : std_logic_vector(7 downto 0) := (others => '0');
54   begin
55     RsRx_tb <= '1';
56     wait for p_baud*5;
57     switch_valore_tb <= "01000000";
58     for sel in 0 to 3 loop
59       switch_selezione_tb <= std_logic_vector(to_unsigned(sel, 2));
60       for i in 0 to 255 loop
61         byte := std_logic_vector(to_unsigned(i, 8));
62         -- Bit di start
63         RsRx_tb <= '0';
64         wait for p_baud;
65         -- 8 bit utili
66         for j in 0 to 7 loop
67           RsRx_tb <= byte(j);
68           wait for p_baud;
69         end loop;
70         -- Bit di stop
71         RsRx_tb <= '1';
72         wait for p_baud;
73       end loop;

```

```

74      end loop;
75      wait;
76  end process;
77 end Behavioral;
```

È possibile subito notare che l'unica componente istanziata è quella relativa al blocco con livello gerarchico più alto, ovvero quella descritta nel file 2.12.

Il corpo di tale **architecture** è composto, oltre che dall'istanziazione della componente, anche da due processi utilizzati per simulare il clock e i tre ingressi.

Il processo **gen_clock** non fa altro che generare un clock con un periodo di **p_clock=10 ns**, dato che il clock di sistema è pari a 100 MHz.

Il processo **sim_rsr** invece viene utilizzato per simulare l'ingresso seriale, quello relativo agli switch utilizzati per selezionare il filtro e il valore di soglia/luminosità. Per fare ciò si sono utilizzati tre cicli **for** annidati. In particolare il primo ciclo **for** (riga 58) è utilizzato per selezionare i 4 filtri, il secondo (riga 60) è utilizzato per generare i byte in ingresso nel range 0-255 ed è rappresentato tramite un **std_logic_vector** di lunghezza 8. Il ciclo **for** più annidato viene invece utilizzato per inviare in maniera seriale i singoli bit contenuti nel **byte** precedentemente generato, partendo dal bit LSB. L'invio di tali bit è preceduto dall'invio del bit di start, e susseguito dal bit di stop. Dopo l'invio di ogni bit si attende per un periodo di tempo pari al periodo di baud grazie all'istruzione **wait for p_baud**. Da notare che il valore del segnale **switch_valore_tb** rimane costante per tutta l'esecuzione del processo, questo perché esso rappresenta il valore di soglia/luminosità, e dato che facendo variare tale valore per ottenere tutte le combinazioni possibili di soglia/luminosità sarebbe risultato troppo oneroso in termini di tempo di simulazione, si è scelto di mantenere tale termine costante. Si è comunque verificato il corretto funzionamento direttamente sulla scheda.

Le simulazioni sono state eseguite per un tempo totale di 2.6 ms, sufficiente a testare tutti i possibili pixel di input (range [0, 255]) per i quattro filtri implementati. Dato che il dispositivo opera ad una velocità di 4Mbps e per ogni byte di dati utili trasmessi è necessario inviare 10 bit (bit di start + 8 bit utili + bit di stop), è stato necessario inviare in totale $255 \times 10 \times 4 = 10200\text{bit}$, il tempo totale è dunque $\frac{10200\text{bit}}{4\text{Mbps}} = 2.55\text{ms}$. Da notare che il tempo totale di simulazione non è effettivamente 2.55 ms, ma leggermente superiore, dato che, come è possibile notare a riga 56 del file 3.1, prima di inviare i dati, si simula uno stato di attesa impostando la linea di ricezione alta per un breve periodo di tempo.

3.1 Simulazione comportamentale

La prima simulazione effettuata è stata quella comportamentale. Tale simulazione è stata utile al fine di verificare il corretto funzionamento logico.

Si è innanzitutto verificato che il filtro trasparente funzionasse correttamente. Questa verifica è risultata particolarmente semplice dato che è bastato controllare che in uscita si presentassero gli stessi dati presenti in ingresso. In figura 3.1 è presente un esempio di tale controllo.



Figura 3.1: Controllo della correttezza del filtro trasparente

Si è poi controllata anche la correttezza del filtro negativo. Ricordando che la funzione di tale filtro è di eseguire il calcolo $255 - pixel$, la figura 3.2 mostra un esempio di calcolo corretto.

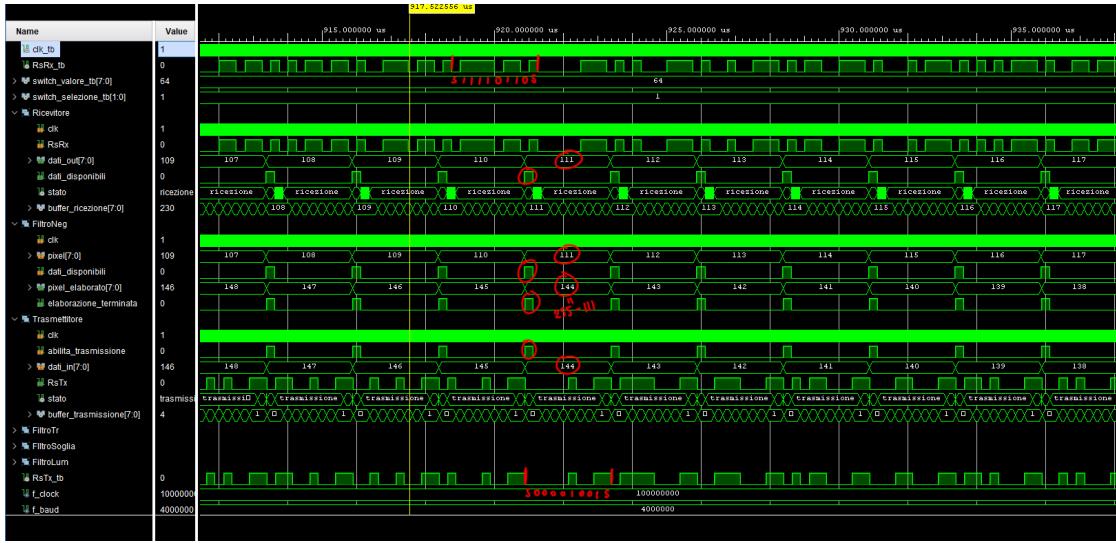


Figura 3.2: Controllo della correttezza del filtro negativo

La correttezza del filtro soglia è stata verificata controllando un pixel sotto la soglia e uno sopra la soglia. Anche in questo caso la figura 3.3 conferma il corretto funzionamento logico di tale filtro.



Figura 3.3: Controllo della correttezza del filtro soglia

Anche il comportamento del filtro di luminosità risulta essere corretto come conferma la figura 3.4



Figura 3.4: Controllo della correttezza del filtro luminosità

3.2 Simulazione post synthesis

Dalla sintesi è possibile notare come il ricevitore e il trasmettitore siano stati sintetizzati esclusivamente da Look-Up Table con un numero di ingressi variabile (LUTx [6]) e Flip-Flop D con Clock Enable e reset sincrono (FDRE [5]).

Il filtro trasparente è invece caratterizzato esclusivamente da Flip-Flop D con Clock Enable e reset sincrono (FDRE [5]).

Nel filtro negativo invece, oltre ai Flip-Flop D con Clock Enable e reset sincrono (FDRE [5]), sono presenti diverse LUT con un numero di ingressi variabile (LUTx [6]).

Il filtro soglia è stato sintetizzato tramite Flip-Flop D con Clock Enable e reset sincrono (FDRE [5]) e tramite un Fast Carry Logic con Look Ahead (CARRY4 [7]).

Il filtro luminosità invece presenta svariati Flip-Flop D con Clock Enable e reset sincrono (FDRE [5]), alcuni Fast Carry Logic with Look Ahead (CARRY4 [7]) e anche diverse LUT con un numero di ingressi variabile (LUTx [6]). Esso inoltre presenta anche dei Flip-Flop D con Clock Enable e set sincrono (FDSE [8]).

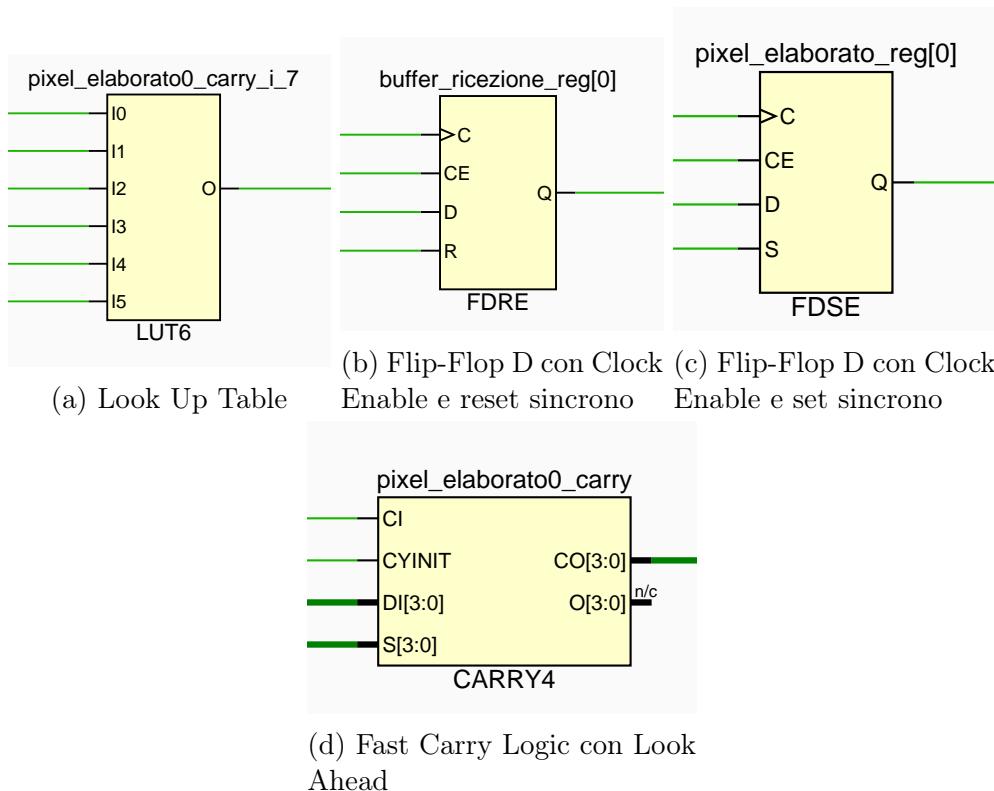


Figura 3.5: Esempio di componenti sintetizzate

Come è possibile notare dalle figure 3.6 e 3.7 il comportamento del circuito risulta corretto sia nel caso della simulazione post synthesis functional che post synthesis timing.

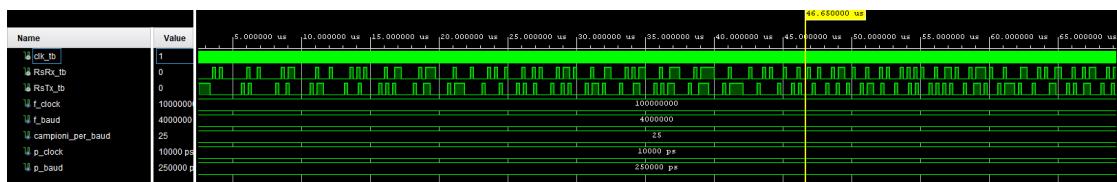


Figura 3.6: Simulazione post synthesis functional

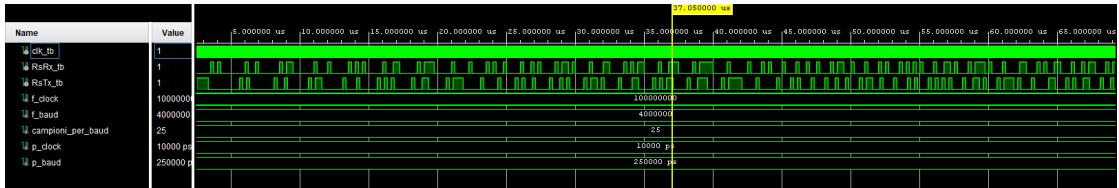


Figura 3.7: Simulazione post synthesis timing

Una analisi più accurata della simulazione post synthesis timing permette tuttavia di evidenziare i ritardi introdotti dalla sintesi. Tali ritardi non compromettono però il corretto funzionamento del circuito. In particolare si è misurato il ritardo tra l'inizio della ricezione e l'inizio della trasmissione nelle due tipologie di simulazione precedentemente illustrate. Dalle figure 3.8 e 3.9 è possibile notare nella simulazione timing è presente un ritardo pari a $2.144252ns - 2.140100ns = 0.004152ns = 4.152ps$.

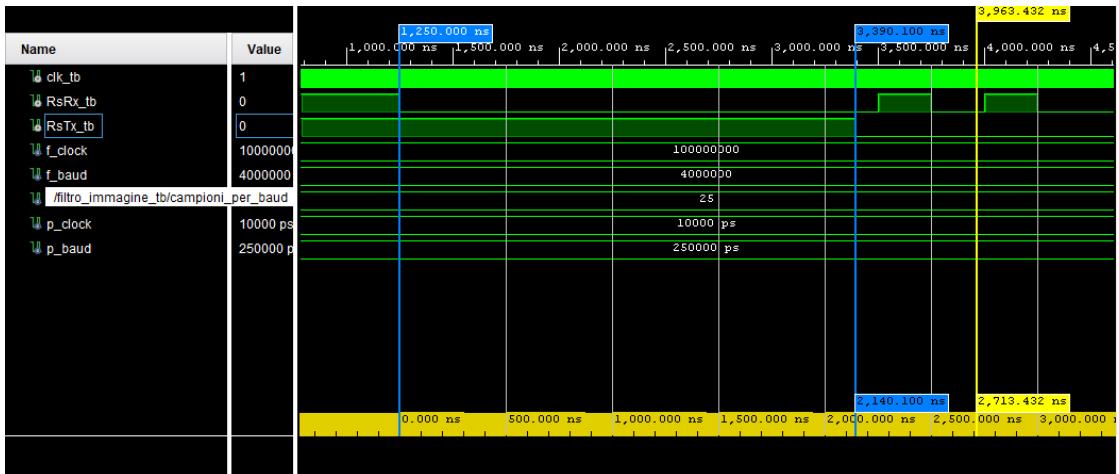


Figura 3.8: Simulazione post synthesis functional - misurazione del ritardo

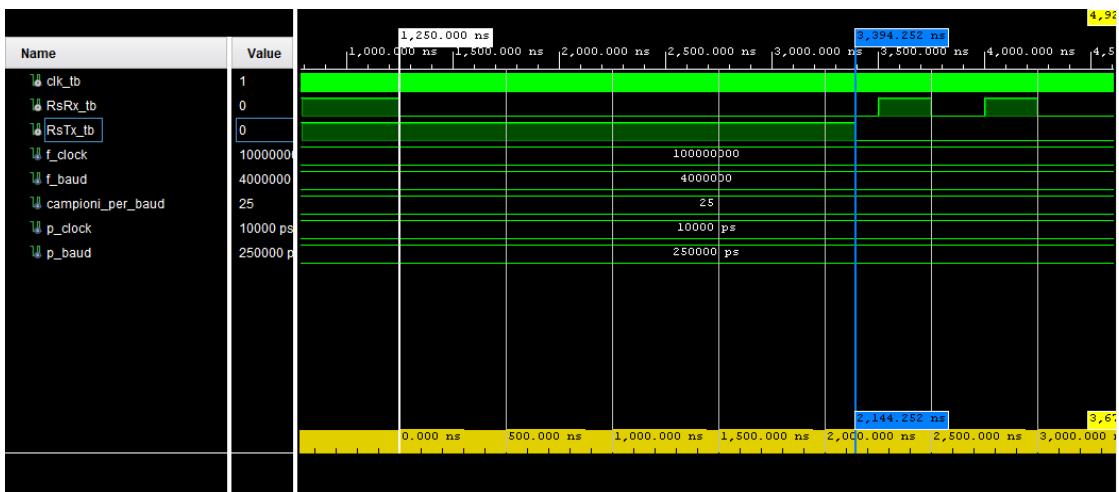


Figura 3.9: Simulazione post synthesis timing - misurazione del ritardo

3.3 Simulazione post implementation

Si sono analizzate le simulazioni post implementation al fine di verificare la presenza di eventuali glitch. Le figure 3.10 e 3.11 non evidenziano alcuna problematica e confermano il corretto funzionamento del circuito anche post implementation.

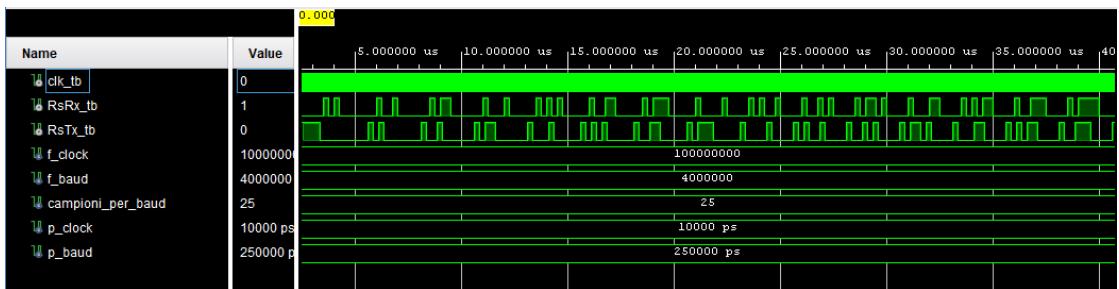


Figura 3.10: Simulazione post implementation functional

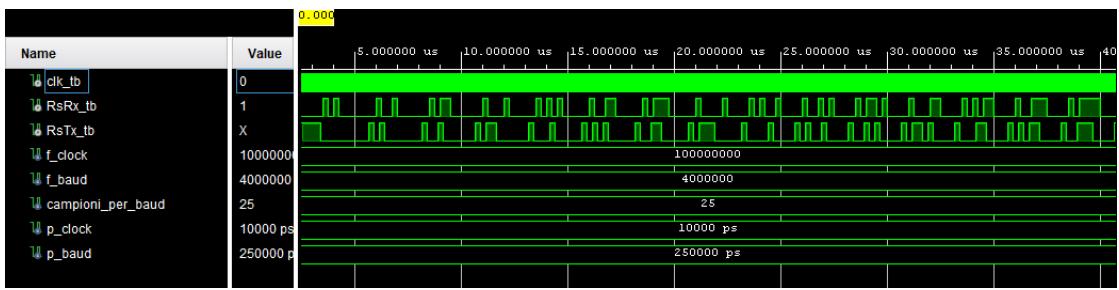


Figura 3.11: Simulazione post implementation timing

Anche in questo caso il ritardo tra invio e ricezione evidenziato nelle figure 3.12 e 3.13 di $2.160616ns - 2.140100ns = 0.020516 = 20.516ps$ non compromette il corretto funzionamento del circuito.

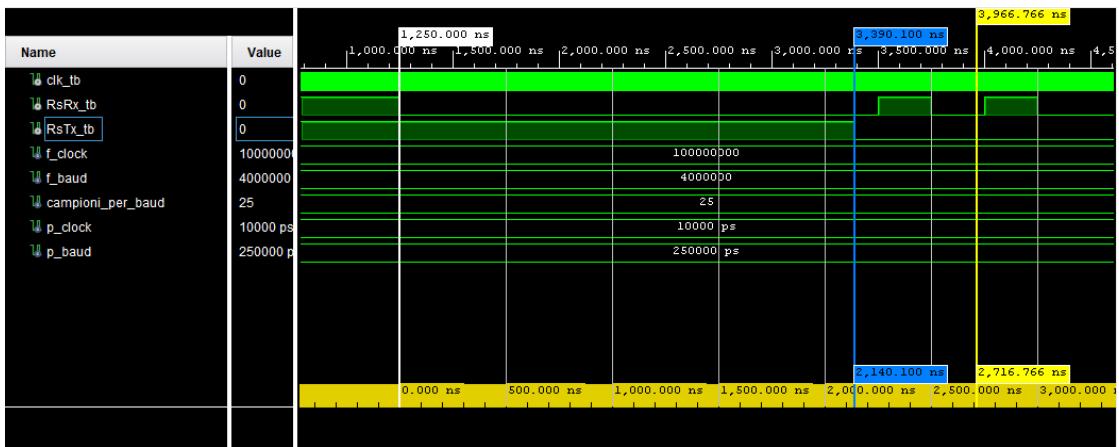


Figura 3.12: Simulazione post implementation functional - misurazione del ritardo

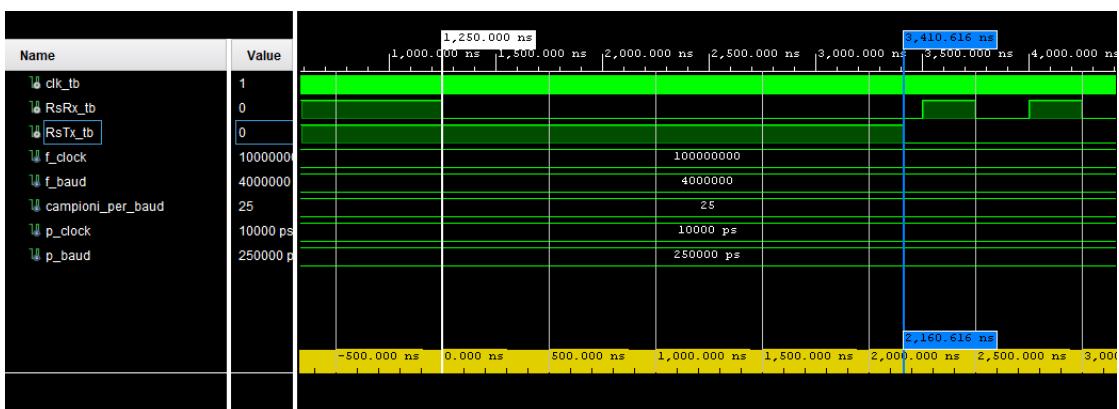


Figura 3.13: Simulazione post implementation timing - misurazione del ritardo

3.4 Utilizzo di risorse

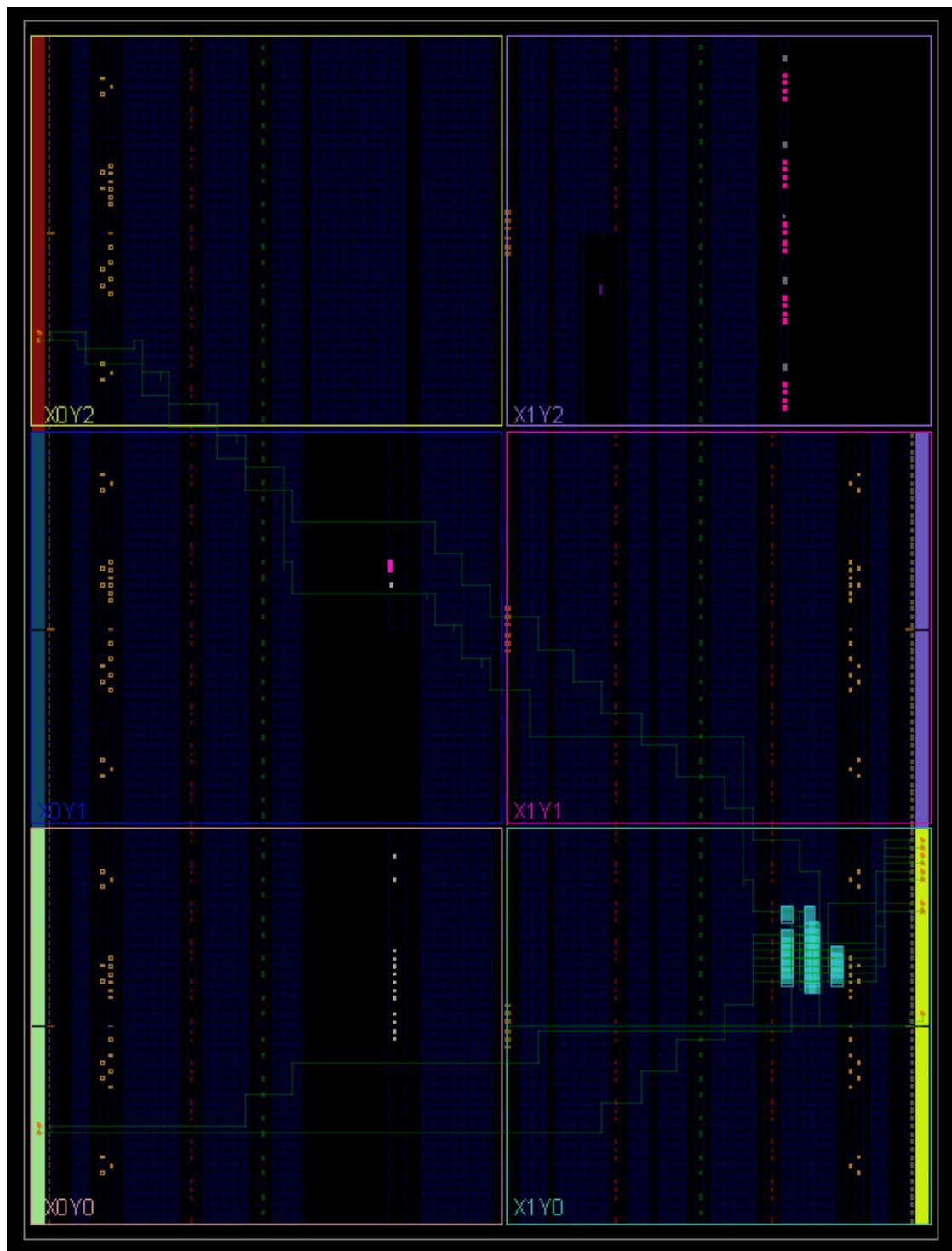


Figura 3.14: Vista del device post implementation

Resource	Utilization	Available	Utilization %
LUT	138	20800	0.66
FF	73	41600	0.18
IO	13	106	12.26

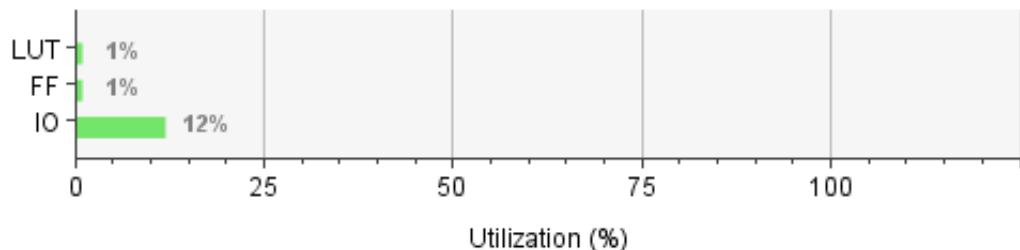


Figura 3.15: Utilizzo di risorse post implementation

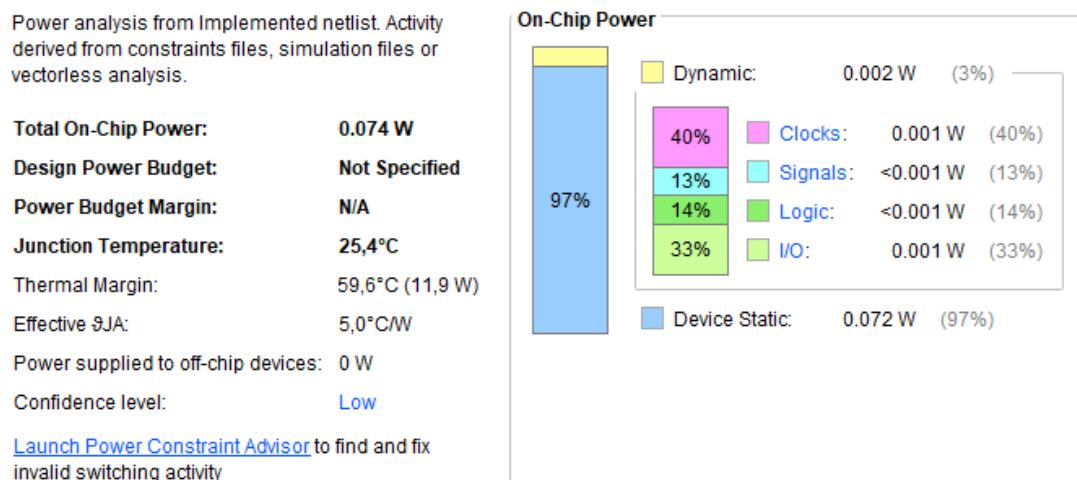


Figura 3.16: Consumo di potenza post implementation

Dalla figura 3.16 è possibile evidenziare che il 97% della potenza consumata risulta essere statica. Inoltre la potenza dinamica consumata per lo più risulta essere dovuta al clock e all'IO, solo il 14% della potenza dinamica (0.42% della potenza totale) è consumata dalla logica.

4. Sperimentazione tramite PySerial

Al fine di poter verificare il corretto funzionamento del circuito implementato sull'FPGA è stato realizzato un piccolo script python. Tale script fa uso della libreria PySerial per inviare i dati relativi ad una immagine all'FPGA tramite connessione USB-UART. Al fine di semplificare l'interazione è stata sviluppata un'interfaccia grafica ad hoc. Per fare ciò si è utilizzata la libreria Tkinter facente parte della standard library di Python. Di seguito il codice relativo a tale implementazione:

Source Code 4.1: gui_test.py

```
1 from tkinter import * # Python 3
2 from PIL import ImageTk, Image
3 from tkinter import filedialog
4 from tkinter import ttk
5 import os
6 import time
7 import serial
8 import numpy as np
9 from threading import Thread
10 from tkinter import font as tkFont # for convenience
11
12 class App:
13     def __init__(self):
14         # root window
15         self.root = Tk()
16         self.root.geometry("1200x500")
17         self.root.minsize(1200, 500)
18         self.root.title('Image Point Operation')
19
20         # configure the grid
21         self.root.columnconfigure(0, weight=1)
22         self.root.columnconfigure(1, weight=1)
23         self.root.columnconfigure(2, weight=1)
```

```

24
25     # immagine originale
26     self.left_canvas = Canvas(self.root, width=500, height=500)
27     self.left_canvas.grid(column=0, row=0, sticky=NS, rowspan=5)
28     self.left_canvas.create_image(0, 0, anchor=NW, image=None)
29
30     # buttoni
31     self.selected_COM = StringVar()
32     self.com_cb = ttk.Combobox(self.root, textvariable=self.selected_COM)
33     self.com_cb['values'] = (
34         'COM1',
35         'COM2',
36         'COM3',
37         'COM4',
38         'COM5',
39         'COM6',
40         'COM7',
41         'COM8',
42         'COM9',
43         'COM10',
44     )
45     self.com_cb.bind('<<ComboboxSelected>>', self.com_selezionata)
46     self.com_cb.current(3)
47     self.com_selezionata()
48     self.com_cb.grid(column=1, row=0, sticky="ew")
49     helv26 = tkFont.Font(family='Helvetica', size=20, weight='bold')
50     self.carica_img_button = Button(self.root, text="Carica Immagine",
51                                     command=self.apri_immagine,
52                                     bd=5, fg='black', relief=GROOVE, font=helv26)
53     self.carica_img_button.grid(column=1, row=1, sticky="news")
54
55     self.invia_dati_button = Button(self.root, text="Invia Dati",
56                                     command=self.invia_dati,
57                                     bd=5, fg='black', relief=GROOVE, font=helv26)
58     self.invia_dati_button.grid(column=1, row=2, sticky="news")
59
60     self.salva_button = Button(self.root, text="Salva",
61                                command=self.salva_immagine,
62                                bg='green', bd=5, fg='black', relief=GROOVE, font=helv26)
63     self.salva_button.grid(column=1, row=3, sticky="news")
64
65     self.esci_button = Button(self.root, text="Esci",
66                               command=self.root.destroy,
67                               bg='red', bd=5, fg='black', relief=GROOVE, font=helv26)
68     self.esci_button.grid(column=1, row=4, sticky="news")
69
70     # immagine elaborata
71     self.right_canvas = Canvas(self.root, width=500, height=500)

```

```

69         self.right_canvas.grid(column=2, row=0, sticky=NS, rowspan=5)
70         self.right_canvas.create_image(0, 0, anchor=NW, image=None)
71
72     self.root.mainloop()
73
74     def apri_immagine(self):
75         self.filename = filedialog.askopenfilename(title='open')
76         self.img = Image.open(self.filename)
77         self.w, self.h = self.img.size
78         self.array = np.asarray(self.img, np.uint8)
79         self.data = self.array.tobytes()
80         print(f"Aperta immagine: {self.img}")
81         self.visualizza_immagine(self.img, self.left_canvas)
82
83     def salva_immagine(self):
84         try:
85             elab_filename = '\\'.join(self.filename.split('\\')[:-1]) +
86             "elab_img.jpg"
87             self.elab_img.save(elab_filename, 'JPEG')
88         except:
89             print("Immagine elaborata non ancora disponibile")
90
91     def visualizza_immagine(self, img, canvas):
92         w, h = img.size
93         if w > h:
94             scaleFactor = 500/self.w
95         else:
96             scaleFactor = 500/self.h
97
98         temp = img.copy()
99         width, height = temp.size
100        temp = temp.resize((int(width*scaleFactor), int(height*scaleFactor)),
101                           Image.ANTIALIAS)
102
103        vImg = ImageTk.PhotoImage(temp)
104
105        canvas.delete("all")
106        canvas.create_image(0, 0, anchor=NW, image=vImg)
107        self.root.mainloop()
108
109    def com_selezionata(self):
110        print (self.com_cb.get())
111        try:
112            self.ser = serial.Serial \
113            (
114                port=self.com_cb.get(),
115                baudrate=4000000,
116                parity=serial.PARITY_NONE,
117                stopbits=serial.STOPBITS_ONE,

```

```

116             bytesize=serial.EIGHTBITS,
117             timeout = 20
118         )
119         print(self.ser)
120     except:
121         print(f"Nessun dispositivo rilevato su {self.com_cb.get()}")
122
123     def invia_dati(self):
124         tx = Thread(target=self.TxThread)
125         rx = Thread(target=self.RxThread)
126         tx.start()
127         rx.start()
128         tx.join()
129         rx.join()
130         self.visualizza_immagine(self.elab_img, self.right_canvas)
131
132     def TxThread(self):
133         print("Transmission started.")
134         d = [self.data[i:i+10000] for i in range(0, len(self.data), 10000)]
135         for t in d:
136             self.ser.write(t)
137             time.sleep(0.0000001)
138             #time.sleep(1)
139         print("Transmission finished.")
140
141     def RxThread(self):
142         print("Reception started.")
143         r_data = self.ser.read(len(self.data))
144         print(f"received: {len(r_data)} bytes")
145         print("Reception finished.")
146         print(f"Received data == data: {self.data == r_data}")
147
148         ba = bytearray(r_data)
149
150         int_values = [x for x in ba]
151
152         int_values = np.asarray(int_values)
153         int_values = int_values.reshape(self.h,self.w,3)
154
155         img = Image.fromarray(np.uint8(int_values))
156         self.elab_img = img
157
158     if __name__ == "__main__":
159         app = App()

```

Segue una carrellata di immagini relativi a test effettuati tramite il codice precedentemente descritto.

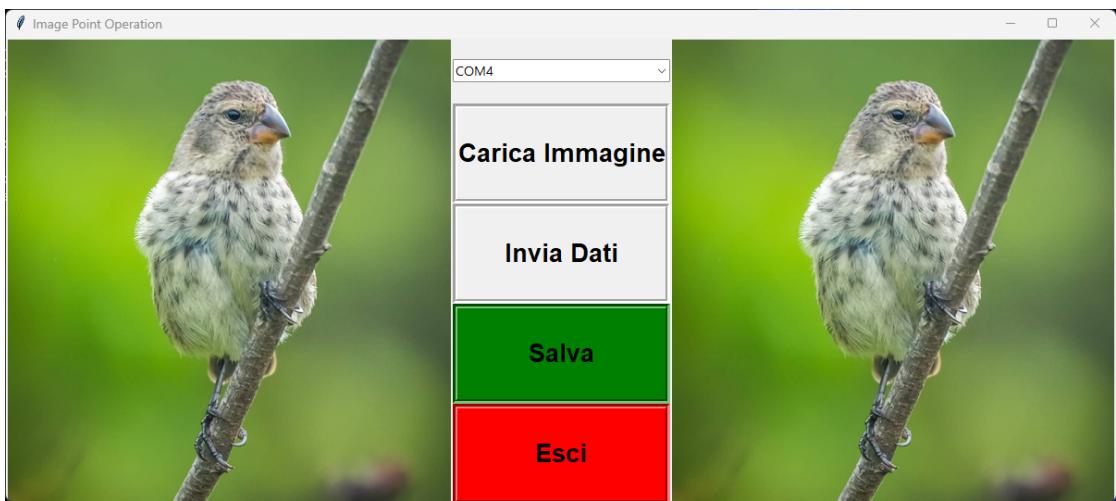


Figura 4.1: Test del filtro trasparente

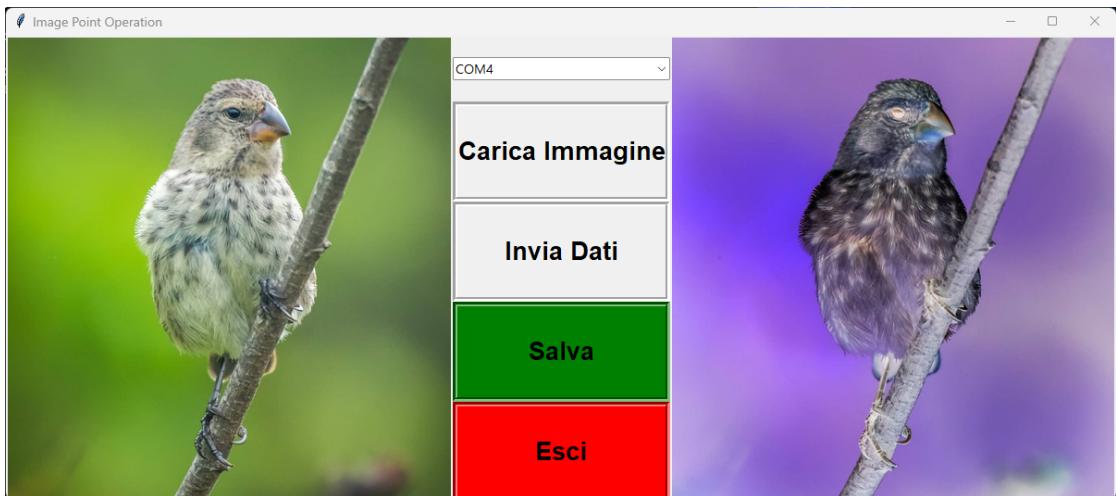


Figura 4.2: Test del filtro negativo

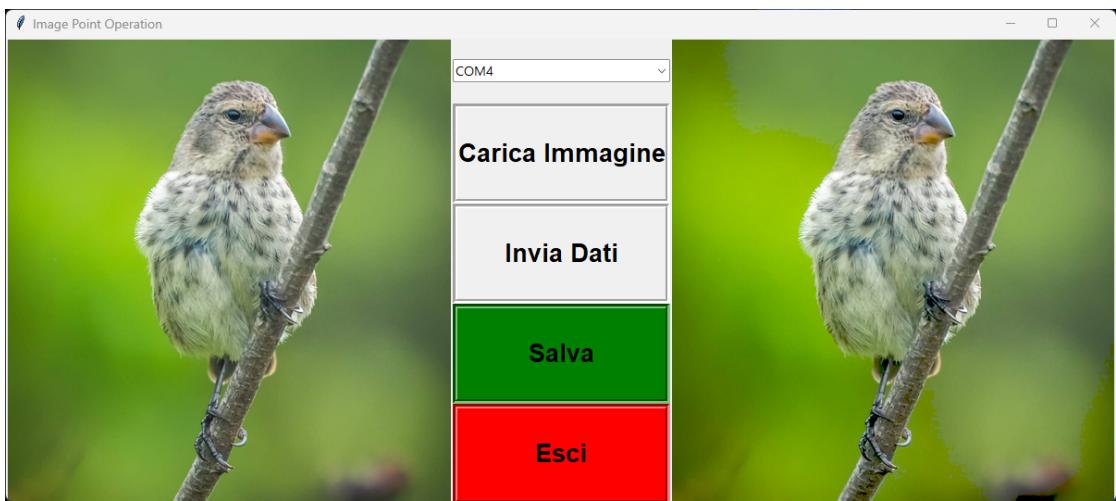


Figura 4.3: Test del filtro soglia con un valore basso di soglia

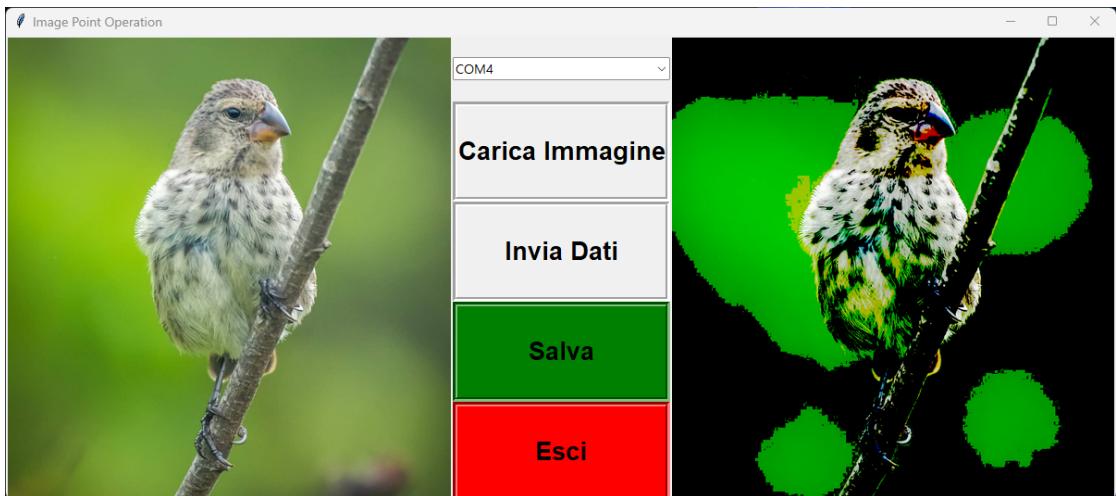


Figura 4.4: Test del filtro soglia con un valore alto di soglia

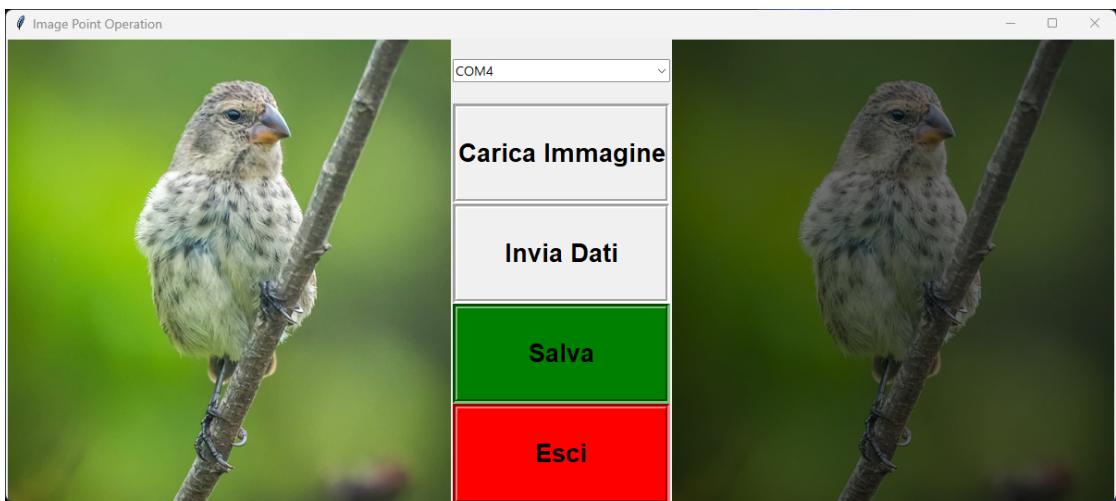


Figura 4.5: Test del filtro luminosità con un valore basso di luminosità

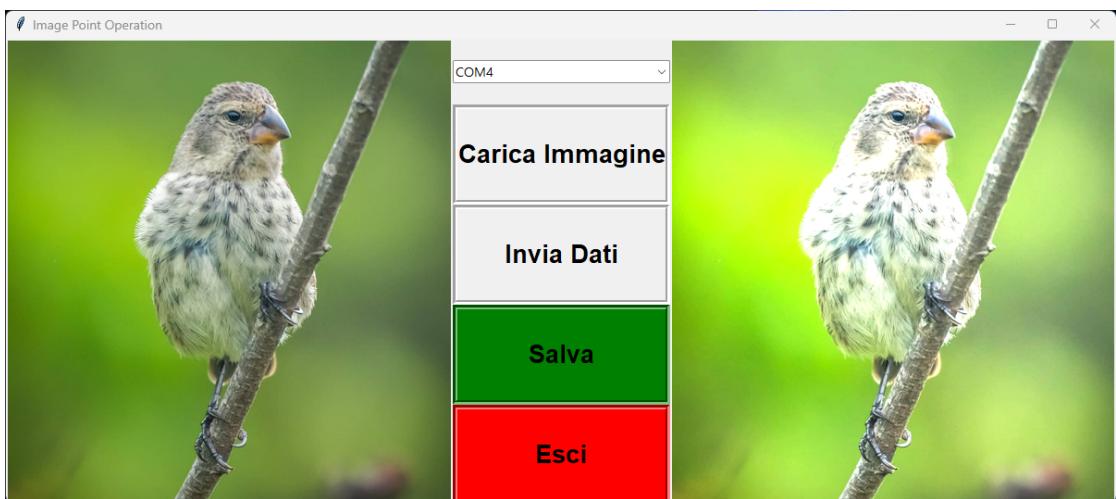


Figura 4.6: Test del filtro luminosità con un valore alto di luminosità

L'utilizzo di tale interfaccia prevede i seguenti step:

1. Selezione della porta COM corretta tramite l'apposita combobox.
2. Caricamento in RAM dell'immagine desiderata tramite il pulsante “carica immagine”.
3. Invio dei dati all’FPGA tramite il pulsante “invia dati”.

4. Il salvataggio (facoltativo) dell'immagine elaborata tramite il pulsante “salva”. Tale immagine verrà salvata nella stessa cartella dell'immagine sorgente e avrà un nome predefinito (da cambiare in caso di salvataggi multipli).
5. Chiusura dell'applicazione tramite il pulsante “Esci”.

5. Conclusione

In conclusione si vuole evidenziare il fatto che tale progetto potrebbe risultare inutilizzabile in qualsiasi contesto pratico, e probabilmente per come è stato implementato lo è. Tuttavia l'applicazione di filtri ad immagini è molto comune in area medica, basta ad esempio considerare l'articolo [9] dove si utilizza un filtro soglia per l'individuazione di tumori al cervello. L'idea di tale progetto è quindi scaturita pensando a possibili applicazioni pratiche, si è tuttavia semplificato il progetto il più possibile perdendo in questo modo qualsiasi riferimento a possibili applicazioni. Ad esempio l'interfaccia UART risulta essere troppo lenta per lo scambio di immagini, e quindi probabilmente nessun sistema medico la utilizzerebbe, si è tuttavia scelto di implementarla perché risulta semplice e facile da testare con un PC.

6. Bibliografia

- [1] https://digilent.com/reference/_media/basys3:basys3_rm.pdf
- [2] <http://unina.stidue.net/Architettura%20dei%20Sistemi%20di%20Elaborazione/Materiale/VHDL.pdf>
- [3] https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter
- [4] https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/hdl/vhdl/vhdl_pro_state_machines.htm
- [5] <https://docs.xilinx.com/r/en-US/ug974-vivado-ultrascale-libraries/ FDRE>
- [6] <https://docs.xilinx.com/r/en-US/ug953-vivado-7series-libraries/ LUT5>
- [7] <https://docs.xilinx.com/r/en-US/ug953-vivado-7series-libraries/ CARRY4>
- [8] <https://docs.xilinx.com/r/en-US/ug974-vivado-ultrascale-libraries/ FDSE>
- [9] P. Natarajan, N. Krishnan, N. S. Kenkre, S. Nancy and B. P. Singh, "Tumor detection using threshold operation in MRI brain images," 2012 IEEE International Conference on Computational Intelligence and Computing Research, 2012, pp. 1-4, doi: 10.1109/ICCIC.2012.6510299.