

ALMA MATER STUDIORUM · UNIVERSITÀ DI
BOLOGNA

SCUOLA DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica
Progetto del Corso di Sistemi Digitali M

An FPGA-based pipeline for multiple real-time Template Matching

Albertazzi Riccardo
Andraghetti Lorenzo
Berlati Alessandro
Corni Gabriele

Prof.
Eugenio Faldella

Anno Accademico 2016 / 2017

Contents

I INTRODUZIONE	4
1 Introduzione al Template Matching	4
2 Strumenti di sviluppo utilizzati	7
2.1 Altera DE2	7
2.2 OV7670	10
3 Descrizione dell'architettura realizzata	12
II IMPLEMENTAZIONE	15
4 Interfacciamento del sensore OV7670	17
4.1 Clock Domain Crossing: fifo asincrona e fifo_adapter	22
5 Interfacciamento alla SRAM	25
5.1 SRAM	25
5.2 Implementazione del controller della SRAM	27
5.3 Implementazione dei moduli di scrittura e lettura dell'immagine	33
5.4 Incapsulamento e interfacciamento con l'architettura	41
6 Interfacciamento alla VGA	46
7 Template Matching	53
7.1 Campionamento della finestra di confronto: window_extractor	58
7.2 SAD e template_finder	66
7.3 Interpretazione dei punteggi: template_aggregator	77
7.4 Aggregazione dei moduli di template matching	83
8 Visualizzazione dei risultati	89
8.1 VGA: square generator	89
8.2 Display a 7 segmenti	95
9 Interfacciamento seriale	101
9.1 Receiver	101

10 Programmazione template	106
10.1 Programmazione da seriale	106
10.2 Programmazione da flusso video	110
10.3 Interfacciamento alla pipeline	117
11 Potenziamento della pipeline di template matching: il caso 64x64	120
III RISULTATI e CONCLUSIONI	127

Lo scopo del progetto è realizzare un'architettura su FPGA in grado di ricavare in tempo reale informazioni sulla presenza e sulla posizione di determinati oggetti all'interno di un flusso video generato da un sensore di immagini interfacciato con l'architettura. Lo studio e l'implementazione delle metodologie di raccolta di tali informazioni rientrano nel settore più generale della Computer Vision denominato "Template Matching". L'implementazione sulla scheda Altera DE2-115 ha portato al riconoscimento real-time di 4 template in parallelo di dimensione fino a 64×64 pixel.

Part I

INTRODUZIONE

1 Introduzione al Template Matching

Il Template Matching è una tecnica di elaborazione dell’immagine utilizzata per ricercare un’immagine sorgente, detta template, all’interno di un’immagine target di dimensioni maggiori. Questa tecnica viene usata in ambito industriale per effettuare controlli di qualità, o come supporto ad operazioni di elaborazione automatica più complesse, tra cui il motion-tracking, l’edge-detection, la face-detection o la guida autonoma.

La tecnica più semplice, implementata in questo progetto, prevede di confrontare il template lungo tutta l’immagine target e, in ogni posizione, ricavare un punteggio (o costo) che sintetizza la bontà della comparazione tra il template e l’immagine in quel preciso punto. In un’architettura altamente parallelizzabile come quella implementabile su FPGA è possibile confrontare più template contemporaneamente: in questo modo è possibile effettuare la ricerca di oggetti diversi, oppure dello stesso oggetto ripreso sotto condizioni geometriche (rotazioni, distanza) e di illuminazione differenti.

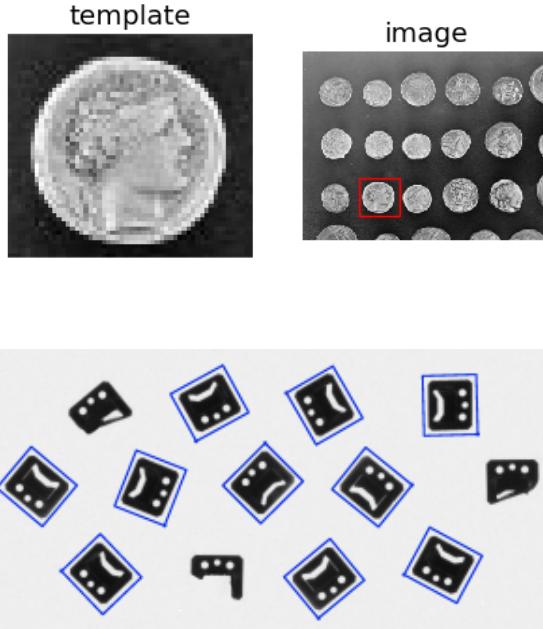


Figure 1: Esempi di applicazione concreta del template matching

Esistono differenti algoritmi per effettuare il template matching. Vengono qui riportati i più conosciuti ed utilizzati: si noti che questi algoritmi prevedono di utilizzare un’immagine in scala di grigi (grayscale). Indicando con I l’immagine target da analizzare, T il template, m e n coordinate di riga e colonna all’interno del template, i e j coordinate di riga e colonna all’interno dell’immagine target:

- Sum of Absolute Differences (SAD): viene fatta la differenza in valore assoluto tra i pixel corrispondenti del template e della porzione di immagine analizzata. Tali differenze vengono poi sommate, ottenendo un punteggio che è tanto più vicino allo zero quanto più le immagini sono simili:

$$SAD(i, j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} |I(i + m, j + n) - T(m, n)|$$

- Normalised Cross-Correlation (NCC): viene effettuata una operazione vettoriale tra il template e la porzione di immagine in oggetto. È una

funzione robusta a cambiamenti lineari dell'intensità (funziona bene cioè anche quando $I = \alpha T$). Il valore ottenuto è tra 0 e 1 ed è tanto più alto quanto più le immagini combaciano:

$$NCC(i, j) = \frac{\sum_{m=0}^{M-1} \sum_{n=0}^{N-1} I(i+m, j+n)T(m, n)}{\sqrt{\sum_{m=0}^{M-1} \sum_{n=0}^{N-1} I(i+m, j+n)^2} \sqrt{\sum_{m=0}^{M-1} \sum_{n=0}^{N-1} T(m, n)^2}}$$

- Zero Normalized Cross-Correlation (ZNCC): versione migliorata di NCC in cui i termini del prodotto vettoriale sono prima sottratti alla relativa media dell'immagine. La funzione è robusta a cambiamenti lineari dell'intensità sommati ad una costante ($I = \alpha T + \beta$).

2 Strumenti di sviluppo utilizzati

2.1 Altera DE2

Il progetto è stato testato inizialmente sulla scheda Altera DE1. Successivamente, poiché le risorse logiche e di memoria presenti sulla DE1 si sono rivelate insufficienti per realizzare un'architettura di riconoscimento di template di dimensione elevata, il progetto è stato traslato sulla board Altera DE2-115. Non volendo entrare nel dettaglio dei componenti della scheda, vengono qui riportate soltanto le interfacce utilizzate:

- Cyclone IV-E FPGA: è il cuore della scheda, contenente 114,480 risorse logiche e 3.9 Mbits di RAM (memoria integrata)
- Expansion Header: pin di GPIO utilizzati per interfacciare il sensore di immagine alla board
- SRAM: utilizzata come buffer per permettere la contemporanea scrittura delle immagini in arrivo dal sensore e la lettura per la VGA
- VGA: utilizzata per visualizzare il flusso video su monitor VGA e segnalare la posizione di eventuali template riconosciuti
- Display a 7 segmenti: utilizzato per contare le occorrenze dei template trovati
- Porta seriale RS-232: utilizzata per consentire la programmazione dei template dall'esterno
- Bottoni: permettono la programmazione di uno specifico template utilizzando come sorgente il flusso video del sensore stesso
- LED e switch: utilizzati per varie funzionalità di testing descritte successivamente

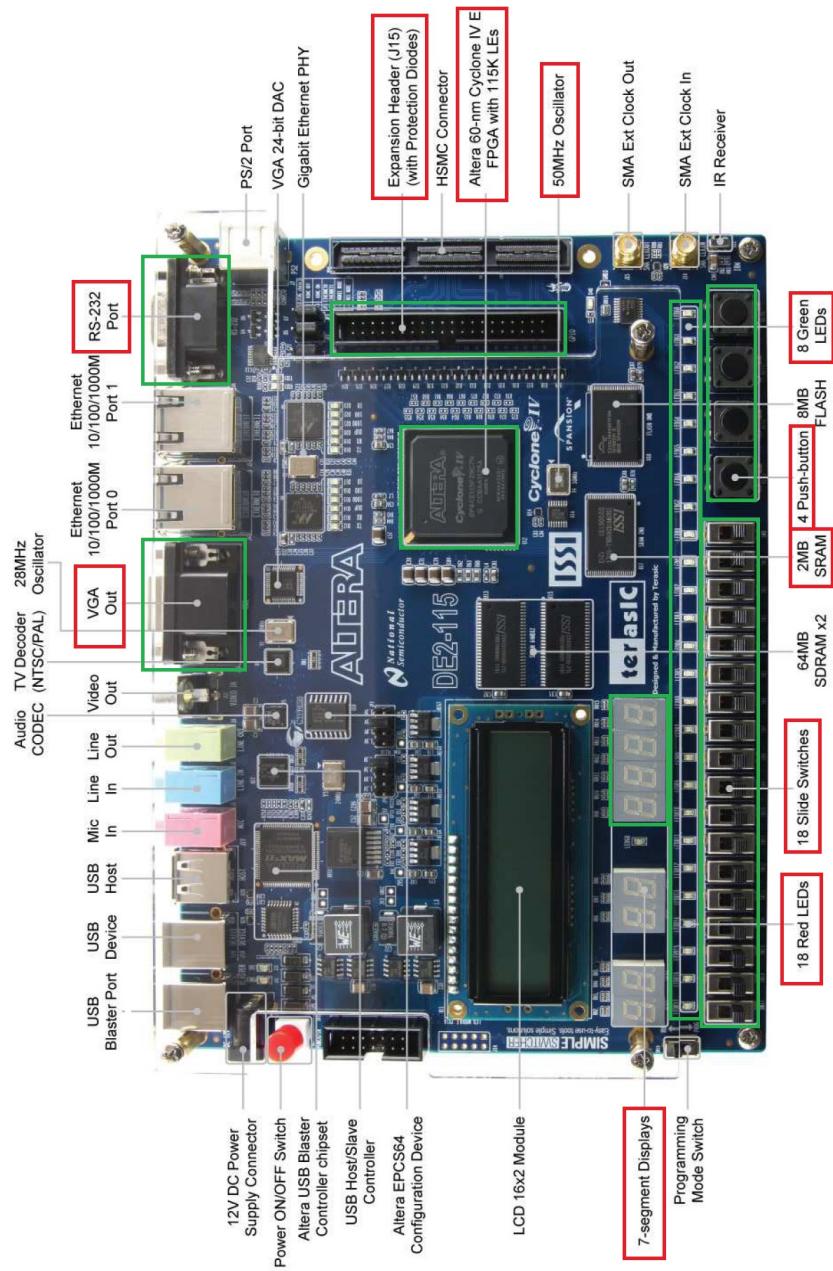


Figure 2: Scheda Altera DE2 e componenti utilizzati



Figure 3: Setup della board

2.2 OV7670

L'OV7670 è un sensore di immagine low-cost prodotto dalla OmniVision e facilmente utilizzabile: infatti per poter essere utilizzato nella configurazione di default non ha bisogno di alcun tipo di programmazione.



Figure 4: OV7670

I segnali di input/output del sensore sono i seguenti:

- 3V3: tensione a 3.3V
- GND: massa
- XCLK (input): clock di ingresso del sensore; i valori ammissibili sono tra 10 MHz e 48 MHz, con valore tipico (qui utilizzato) di 24 MHz.
- RESET (input): reset del sensore, attivo basso.
- PWDN (input): power down, di default a 0.
- PCLK (output): il pixel clock con cui il sensore invia i dati in uscita. Di default è uguale a XCLK.
- VSYNC (output): vertical synchronization, viene settato a 1 prima dell'arrivo di un frame.
- HREF (output): horizontal synchronization, rimane attivo per tutto il tempo in cui viene trasferita una riga di un'immagine (si veda il datasheet in dettaglio più avanti).

- D[7..0] (output): dati generati dal sensore.

Sono presenti altri due segnali utilizzati per la programmazione del sensore (qui non utilizzati) e che usano il protocollo di comunicazione seriale SCCB (di fatto equivalente a I2C):

- SDIOC (input): clock usato nella comunicazione (valore tipico 400 kHz).
- SDIOD (input-output): linea seriale per lo scambio dei dati.

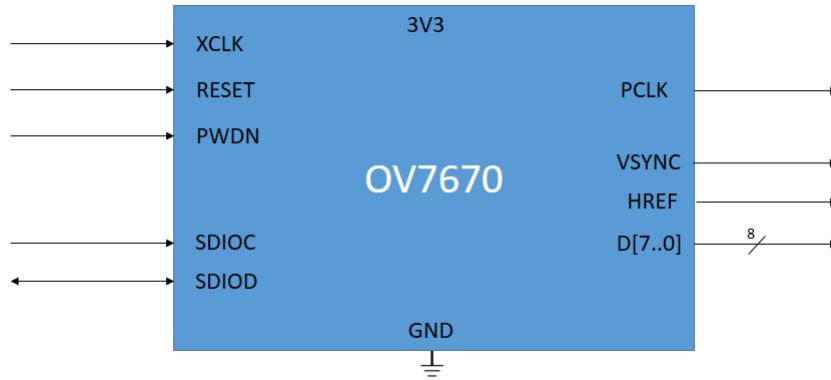


Figure 5: Interfaccia di I/O dell'OV7670

Di default il frame generato dal sensore è a risoluzione VGA (640x480 pixel) e il formato di uscita è YCbCr; in questo formato un pixel viene codificato utilizzando tre componenti, uno per la luminanza (Y, corrisponde all'intensità in grayscale) e due componenti per la crominanza (Cb e Cr). In particolare il sensore OV7670 utilizza un byte per ogni componente e condivide le componenti di crominanza tra due pixel consecutivi, consentendo di memorizzare un pixel utilizzando in media 2 byte. L'ordine temporale di uscita delle tre componenti è il seguente:

N	Byte
1st	Cb0
2nd	Y0
3rd	Cr0
4th	Y1
5th	Cb2
6th	Y2
7th	Cr2
8th	Y3
...	...

Figure 6: Ordine di uscita dei valori YCbCr nell'OV7670 a default

Come si può notare per utilizzare i sensori in grayscale è sufficiente campionare i dati dispari in uscita del sensore.

3 Descrizione dell'architettura realizzata

Inizialmente il sistema campiona l'immagine in grayscale proveniente dal sensore OV. Il flusso di pixel viene utilizzato per 3 differenti scopi:

- Salvataggio delle immagini all'interno della SRAM: la dimensione di un frame è di $640 \times 480B = 300kB$ (1 pixel = 1 byte). L'FPGA sulla DE1 non prevedeva abbastanza memoria integrata per salvare un intero frame, per cui si è scelto di utilizzare come buffer temporaneo di salvataggio delle immagini la SRAM presente sia sulla DE1 sia sulla DE2. Nel caso della DE1 la dimensione della SRAM era di 512 kB, consentendo di utilizzare un unico buffer per lettura e scrittura. Tale struttura è stata mantenuta anche nel passaggio alla DE2 nonostante la SRAM ora presente sia di 2 MB.

Si noti che, nonostante la memoria integrata presente sulla FPGA Cyclone IV della DE2 consenta, al contrario della DE1, il salvataggio di un frame, la scelta di utilizzare una memoria esterna rimane comunque la più ragionevole, in modo da utilizzare la memoria integrata della FPGA per l'implementazione degli algoritmi di visione, che richiedono l'utilizzo di buffer piuttosto grandi e un accesso ad ogni clock.

- Template Matching: in questo macro blocco viene implementato lo

specifico algoritmo di match; qui sono inoltre contenuti i template, che possono essere programmati attraverso una specifica porta.

- Programmazione template: il blocco è responsabile della programmazione dei template; è possibile configurare i template sia da porta seriale sia dallo stesso flusso video (campionando una specifica porzione di immagine).

La porzione finale di sistema si occupa di leggere il frame dalla SRAM, combinarlo con i dati prodotti dal template matching e visualizzare immagine e risultati a video. Nell'implementazione concreta viene disegnato un quadrato su schermo le cui coordinate sono basate su quelle generate dall'algoritmo di template matching. Lo stesso quadrato viene utilizzato in fase di programmazione dei template per avere un "mirino" che consenta di riprendere correttamente con il sensore il template desiderato.

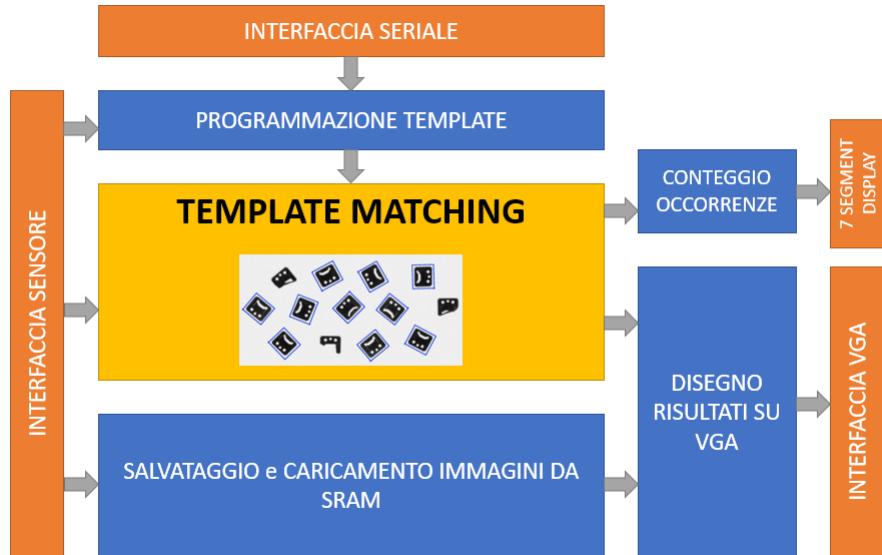


Figure 7: Schema a macro blocchi del progetto

Il sistema consente inoltre di interagire attraverso bottoni e switch. In particolare è possibile:

- Resetare l'intero sistema e il sensore in fase di inizializzazione (1 switch di reset)

- Attivare la visualizzazione di un quadrato in posizione statica in modo da consentire la corretta programmazione dei template tramite flusso video(1 switch di programmazione)
- Programmare uno specifico template (4 bottoni di programmazione, uno per ogni template)
- Impostare il valore di soglia (threshold) al di sopra del quale la comparazione col template viene considerata fallita (6 switch di threshold).
- Mascherare selettivamente i singoli moduli di template matching (4 switch masked/enabled#, uno per ogni template); un template mascherato non verrà mai riconosciuto.

Ulteriori feedback in uscita si hanno attraverso i display a 7 segmenti, in cui viene segnalato il numero di template trovati, e attraverso i led, su cui per esempio viene segnalata la presenza o meno di un template e il numero del template eventualmente trovato.

Part II

IMPLEMENTAZIONE

L'ordine in cui sono presentati i vari blocchi segue l'ordine temporale di implementazione, che rispecchia un determinato ordine logico: prima è stata implementata la pipeline base (sensore + SRAM + VGA), in seguito è stato implementato il template matching e la programmazione dei template.

I vari moduli sono stati realizzati secondo una struttura master-slave, in cui un componente "master" a monte elabora dati per un componente "slave" a valle. Questa struttura ha portato alla realizzazione di un'architettura in pipeline altamente efficiente e modulare.

Come protocollo di handshake master-slave è stato scelto, dove possibile, un meccanismo molto semplice che prevede l'invio del dato e di un bit che indica quando il dato corrispondente è valido. Pertanto si incontrerà spesso nel progetto un segnale denominato "x" accoppiato ad un segnale "x_valid".

Oltre alla semplicità di implementazione, si è scelto di utilizzare questo meccanismo in quanto si prevede che in una applicazione real-time il generico modulo a valle sia sempre pronto a ricevere i dati. Dove questa ipotesi non sia applicabile è necessario introdurre un segnale di ready dal modulo a valle al modulo a monte (si veda più avanti il caso delle fifo asincrone).

Si riporta qui il codice delle costanti utilizzate lungo tutto il progetto.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package global is

    constant IMG_HEIGHT : natural := 480;
    constant IMG_WIDTH  : natural := 640;

    -- bit per ogni pixel
    constant PIXEL_WIDTH : natural := 8;
    subtype pixel is std_logic_vector(PIXEL_WIDTH - 1 downto 0);

    subtype pixel_int is integer range 0 to 2**PIXEL_WIDTH - 1;

end package global;
```

4 Interfacciamento del sensore OV7670

Il sensore OV è stato collegato ai GPIO della scheda. Il clock XCLK a 24 MHz, necessario per il funzionamento del sensore, è generato da un PLL a partire da un clock a 50 MHz disponibile alla FPGA. Come segnale di clock per i moduli all'inizio della pipeline, che si interfacciano direttamente con il sensore, viene usato il clock PCLK restituito dal sensore OV alla scheda. Per poter campionare correttamente i pixel inviati dal sensore è necessario gestire opportunamente i segnali di controllo HREF e VSYNC. Si riporta dal datasheet del sensore il timing dei segnali HREF e VSYNC a risoluzione VGA (default).

Figure 5 Horizontal Timing

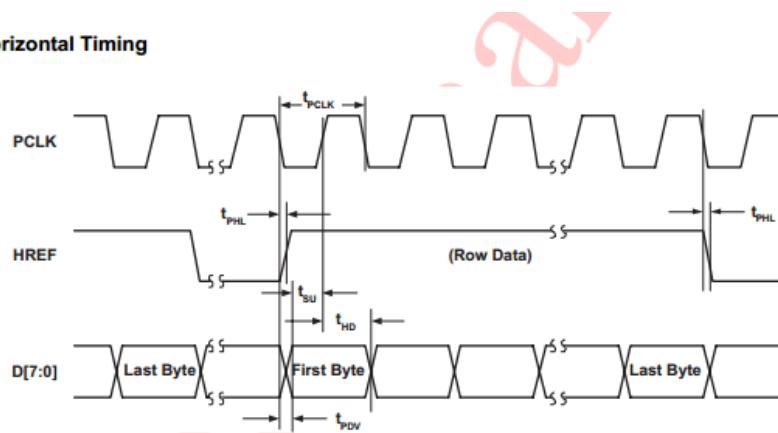


Figure 6 VGA Frame Timing

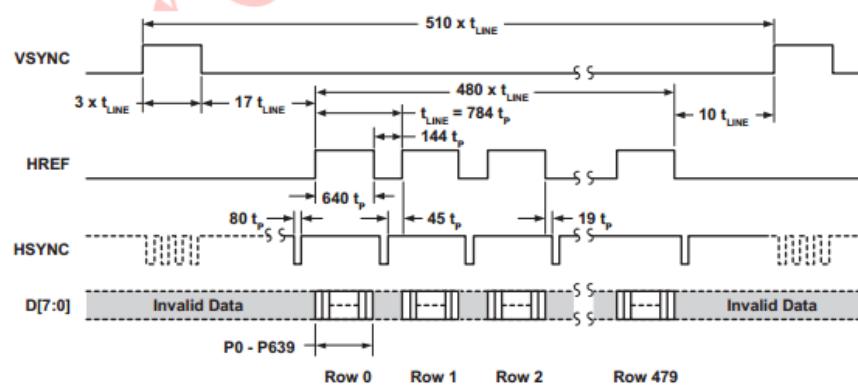


Figure 8: Diagramma OV

Il primo modulo, denominato *camera_controller*, è direttamente interfacciato con i segnali generati del sensore OV7670.

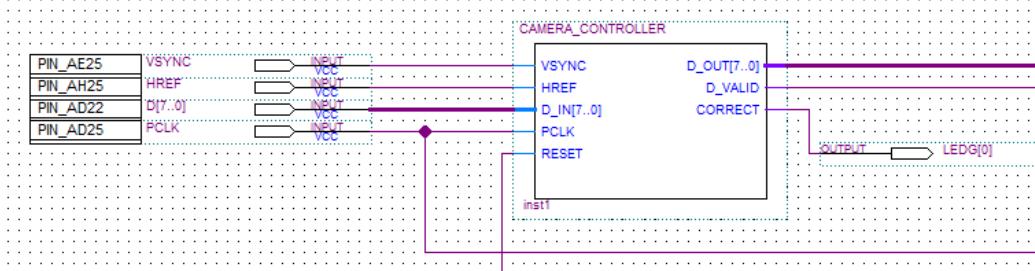


Figure 9: Interfaccia del camera_controller

Questo componente campiona opportunamente i segnali VSYNC, HREF e D e restituisce in uscita la sola componente grayscale dell’immagine su D_OUT, assieme al relativo segnale di validità D_VALID.

Una versione iniziale di questo modulo prevedeva la sola gestione del segnale HREF, in quanto sufficiente per sapere quando in ingresso si ha un dato valido. In seguito a problemi rilevati sull’immagine quando si spostano i cavi, è stata inserita una parte di conteggio dei pixel in arrivo dalla camera che verifica che ad ogni frame arrivino realmente 640×480 pixel. Al termine dell’invio dei pixel (cioè al salire di VSYNC) viene verificato il valore del contatore e aggiornata l’uscita CORRECT, che segnala appunto la correttezza del frame ricevuto. Il segnale CORRECT è stato collegato ad un led (LEDG[0]) ed è possibile verificare che quando sulla VGA si hanno errori sull’immagine il led è spento.

Il funzionamento del modulo è governato da una macchina stati, riportata in figura. Al momento del reset il modulo passa nello stesso di RST resettando il contatore dei pixel. Nel momento in cui viene tolto il reset è necessario risincronizzarsi con il flusso video, quindi ci si pone in attesa dell’evento VSYNC = 1. Una volta transitato nello stato VSYNC_STATE comincia il ciclo tra gli stati VSYNC_STATE e CAMPIONA: lo stato VSYNC_STATE segnale sempre che il segnale VSYNC è attivo, viceversa nello stato CAMPIONA il segnale VSYNC è basso e quando HREF è attivo è possibile campionare i pixel in arrivo (vengono campionati solo i byte dispari per ottenere l’immagine in grayscale). Ogni volta che un pixel grayscale viene campionato

viene incrementato il contatore e il pixel viene prontamente inviato sull'uscita asserendo il relativo segnale di validità; nella transizione di stato da CAMPIONA a VSYNC_STATE viene infine aggiornato il valore dell'uscita CORRECT e resettato il contatore.

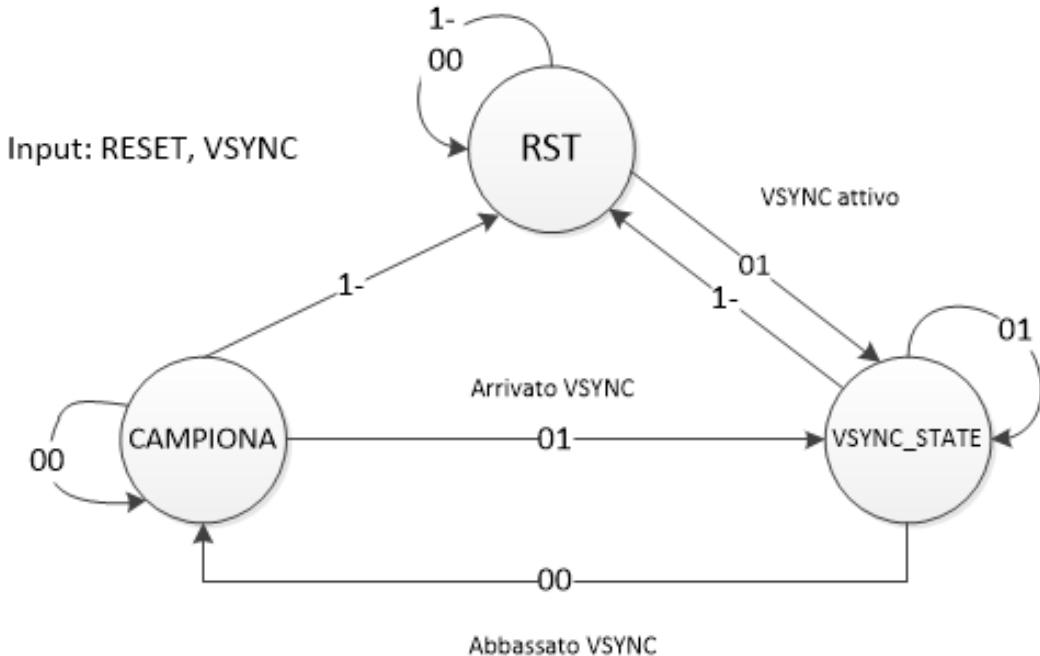


Figure 10: Macchina a stati finiti del camera_controller (sono riportati solo i segnali di input)

Viene infine riportato il codice vhdl del camera_controller:

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use work.global.all;

ENTITY CAMERA_CONTROLLER IS
PORT(
    VSYNC  : IN STD_LOGIC;
    HREF   : IN STD_LOGIC;
    D_IN   : IN STD_LOGIC_VECTOR(7 downto 0);
    D_OUT  : OUT STD_LOGIC_VECTOR(7 downto 0);
    
```

```

D_VALID  : OUT STD_LOGIC;
CORRECT   : OUT STD_LOGIC;
-- CLK and RESET --
PCLK, RESET: IN STD_LOGIC
);
END CAMERA_CONTROLLER;

ARCHITECTURE CAMERA_CONTROLLER_ARCH OF CAMERA_CONTROLLER is

TYPE type_state is (RST, VSYNC_STATE, CAMPIONA);

signal STATE : type_state := RST;
signal ODD_BYTE : STD_LOGIC := '0';

BEGIN

PROCESS(PCLK,RESET)
variable COUNT : integer range 0 to IMG_WIDTH*IMG_HEIGHT;
BEGIN
IF(RESET = '1')
THEN
STATE <= RST;
ODD_BYTE <= '0';
D_VALID <= '0';
COUNT := 0;
ELSIF (RISING_EDGE(PCLK))
THEN

D_VALID <= '0';
ODD_BYTE <= '0';

case STATE is

when RST =>
if (VSYNC = '1') then
STATE <= VSYNC_STATE;
end if;

```

```

when VSYNC_STATE =>
    if (VSYNC = '0') then
        STATE <= CAMPIONA;
    end if;

when CAMPIONA =>

    if (VSYNC = '0') then
        IF(HREF = '1')
        THEN
            IF(ODD_BYTE = '1')
            THEN
                D_OUT <= D_IN;
                D_VALID <= '1';
                ODD_BYTE <= '0';
                COUNT := COUNT + 1;
            ELSE
                ODD_BYTE <= '1';
            END IF;
        END IF;
    else
        if (COUNT = IMG_WIDTH*IMG_HEIGHT) then
            CORRECT <= '1';
        else
            CORRECT <= '0';
        end if;

        COUNT := 0;
        STATE <= VSYNC_STATE;
    end if;

end case;

END IF;
END PROCESS;

END CAMERA_CONTROLLER_ARCH;

```

4.1 Clock Domain Crossing: fifo asincrona e fifo_adapter

Poiché sia la parte di computer vision sia la parte di salvataggio delle immagini su SRAM necessita di compiere operazioni per più intervalli di clock per ogni pixel si è scelto di inviare i pixel in un dominio governato da un clock a frequenza superiore, 100 MHz. La frequenza di arrivo dei pixel è di 12 MHz (24MHz / 2 in quanto vengono campionati solo i byte dispari), quindi è possibile effettuare operazioni che richiedono fino a 8 clock per il loro completamento.

Altera offre agli sviluppatori un componente DCFIFO che realizza la "fifo asincrona", cioè una coda scrivibile e leggibile con clock differenti che permette di effettuare il cosiddetto Clock Domain Crossing (passaggio a domini di clock differenti). La parte di scrittura è stata collegata direttamente al camera_controller, mentre l'uscita è stata collegata ad un modulo denominato *fifo_adapter* il cui compito è gestire i segnali di handshake della fifo per riproporre in uscita l'handshake dato + dato_valid.

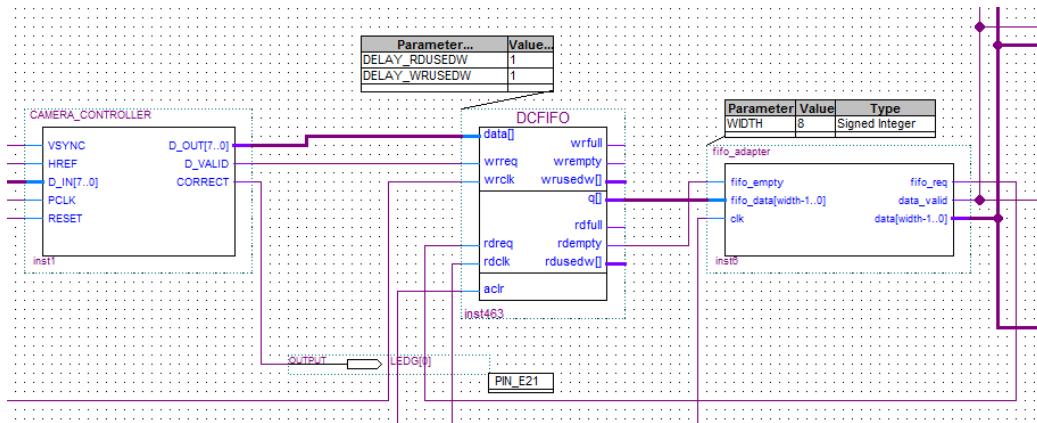


Figure 11: Interfaccia del camera_controller

Si noti come non sia stato utilizzato il segnale *wrfull*, che segnalerebbe alla porta di scrittura quando la fifo è piena. Si suppone infatti, per le ipotesi già enunciate, che la pipeline a valle sia sempre più veloce o al limite ugualmente veloce quanto la parte a monte. Pertanto non si verificherà mai che la fifo sia piena; si verificherà invece che nella fifo saranno contenuti 0 zero o 1 pixel.

La fifo è stata programmata impostando il parametro LPM_SHOWAHEAD al valore "ON". Questo parametro fa sì che la fifo esponga il dato in uscita prima ancora di ricevere il segnale di richiesta di lettura *rdreg*. In questo modo il modulo *fifo_adapter* non deve preoccuparsi di assere il segnale di richiesta prima di ricevere il segnale *rdempty* a 0. Allo stesso tempo però è necessario considerare che nel momento in cui il *fifo_adapter* porta il segnale di richiesta a 1 riceverà sempre al clock successivo il segnale *rdempty* ancora a 0; questo si verifica per il comportamento sincrono delle due reti. Per evitare quindi di leggere dati spuri è necessario non campionare dati dalla fifo al clock successivo in cui sono stati letti dei dati validi.

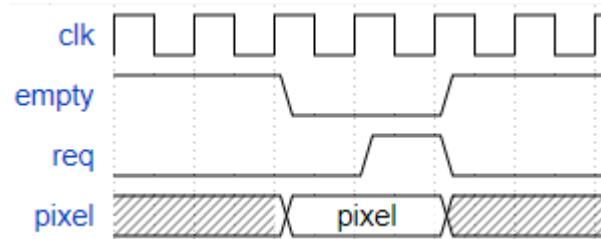


Figure 12: Descrizione del funzionamento della fifo (segnali empty e pixel) e del *fifo_adapter* (segnale req)

Da qui in avanti qualsiasi componente abbia bisogno del flusso dati della camera può interfacciarsi con il *fifo_adapter* (che è già collegato al clock a 100 MHz). Si riporta infine il codice del *fifo_adapter*:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity fifo_adapter is
generic
(
    WIDTH : natural := 8
);
port
(
    fifo_req    : out std_logic;
    fifo_empty   : in std_logic;
```

```

fifo_data    : in std_logic_vector(WIDTH - 1 downto 0);

data_valid   : out std_logic;
data        : out std_logic_vector(WIDTH - 1 downto 0);

clk         : in std_logic
);
end fifo_adapter;

architecture fifo_adapter_arch of fifo_adapter is
  type state_type is (Idle, Transmit);
begin
  process(clk)
    variable state: state_type := Idle;
  begin
    if (rising_edge(clk)) then

      if (fifo_empty = '1') then
        fifo_req    <= '0';
        data_valid  <= '0';
      else
        case state is
          when Idle =>
            fifo_req    <= '1';
            data_valid  <= '1';
            data <= fifo_data;
            state := Transmit;
          when Transmit =>
            fifo_req    <= '0';
            data_valid  <= '0';
            state := Idle;
        end case;
      end if;

      end if; -- clock
    end process;
  end architecture;

```

5 Interfacciamento alla SRAM

Il primo macro blocco interfacciato con il flusso dati in uscita dal fifo_adapter è il blocco incaricato di scrivere e leggere su SRAM le immagini generate dal sensore. Si noti che, al di là della quantità più o meno grande di memoria integrata su FPGA, la soluzione non sarebbe implementabile con una semplice fifo, in quanto il sensore OV e la VGA hanno tempi totalmente diversi (basti pensare che il sensore genera immagini a 30 fps mentre la VGA visualizza immagini a 60 fps). E' necessario quindi disaccoppiare i canali di lettura e scrittura in modo da renderli totalmente indipendenti l'uno dall'altro. Allo stesso tempo la memoria fisica è una sola, pertanto sarà necessario predisporre un arbitro che impedisca situazioni di starvation per entrambi i canali.

5.1 SRAM

La SRAM presente sulle schede DE1 e DE2 è una memoria statica CMOS asincrona (priva di clock); la particolarità delle memorie presenti su queste schede è che ogni cella è di 16 bit. In questo modo è possibile trasferire 16 bit alla volta e i bit di indirizzamento della memoria sono uno in meno rispetto ad una memoria di pari dimensione ma con celle di 1 byte. L'interfaccia della SRAM è composta dai canonici segnali di chip enable, write enable, output enable, indirizzo e dati. In più sono presenti due segnali UB# e LB# (Upper byte control e Lower byte control) che permettono di specificare se effettuare un trasferimento a 16 o a 8 bit e, nel secondo caso, se salvare il dato nel byte più o meno significativo.

Si riportano di seguito alcune immagini di interesse tratte dal datasheet della memoria presente sulla DE1 (la memoria sulla DE2 ha solamente 2 bit di indirizzo in più essendo 4 volte più grande):

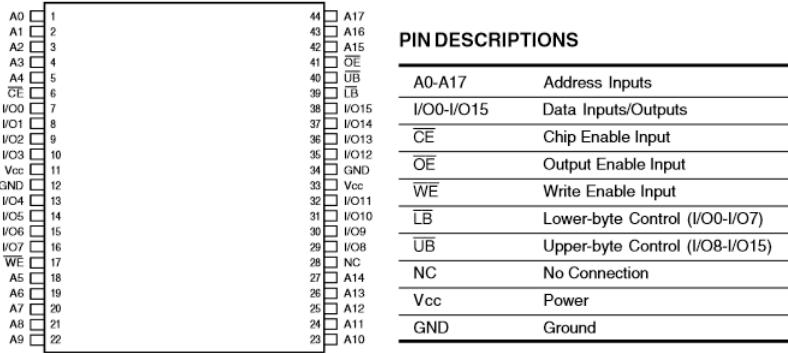
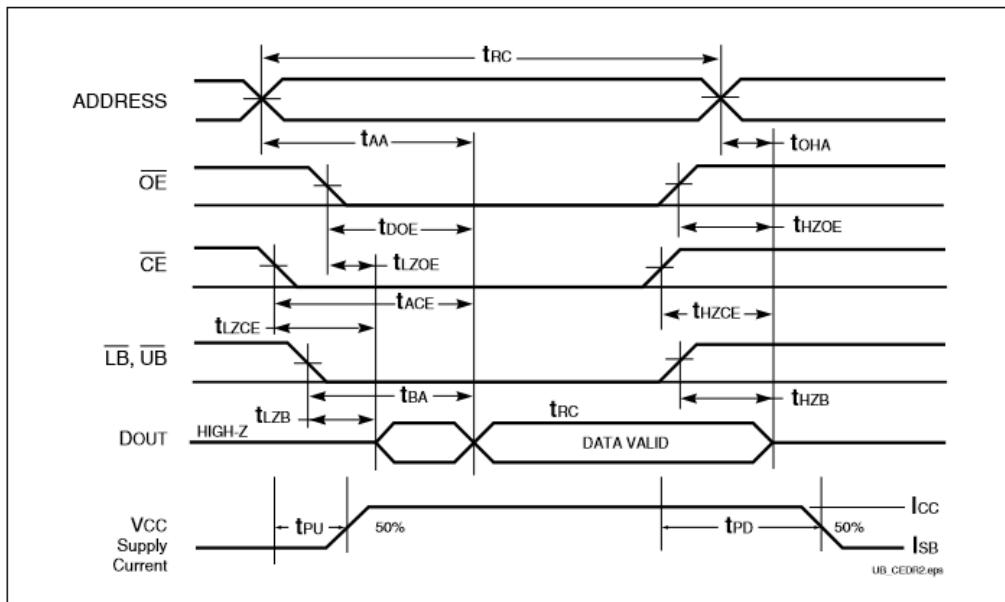


Figure 13: Interfaccia della SRAM presente sulla DE1

READ CYCLE NO. 2^(1,3)



Notes:

1. WE is HIGH for a Read Cycle.
2. The device is continuously selected. \overline{OE} , \overline{CE} , \overline{UB} , or $\overline{LB} = V_{IL}$.
3. Address is valid prior to or coincident with CE LOW transition.

Figure 14: Forme d'onda per accesso in lettura

WRITE CYCLE NO. 1 (\overline{CE} Controlled, \overline{OE} is HIGH or LOW) (1)

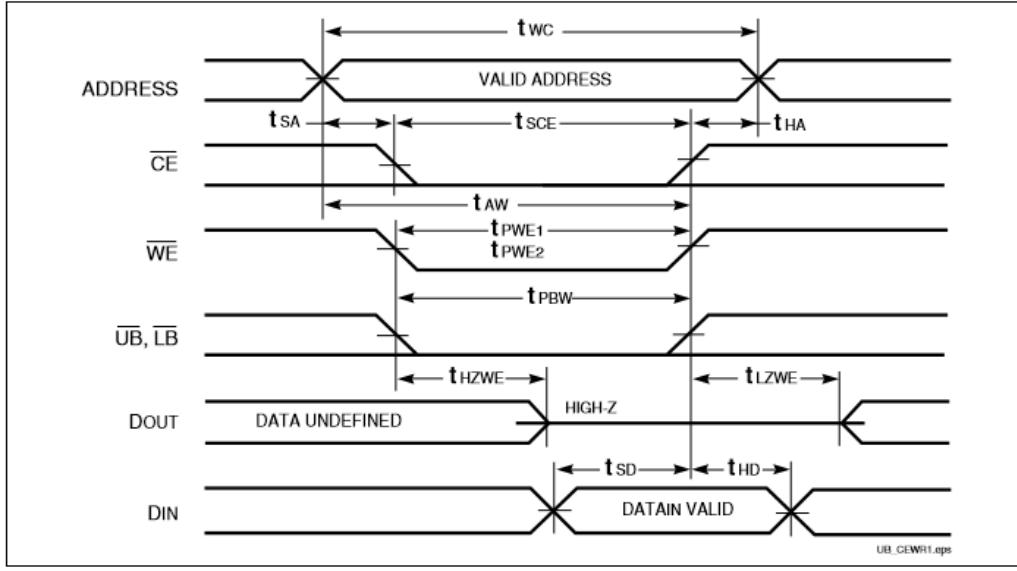


Figure 15: Forme d'onda per accesso in scrittura

Per effettuare una lettura ed una scrittura correttamente è necessario rispettare i tempi tRC e tWC . Il valore minimo di questi due tempi è lo stesso ed è di 10 ns, che corrisponde ad una frequenza di 100 MHz (motivo che ha portato alla scelta di 100 MHz come frequenza comune per SRAM e template matching).

5.2 Implementazione del controller della SRAM

Il primo modulo realizzato è il controller, denominato *sram_controller*, che espone una porta di scrittura e una porta di lettura, realizza la logica di arbitraggio e si interfaccia con la SRAM, rispettando i timing richiesti. Le porte di lettura e scrittura sono composte da due segnali di handshake, il primo di richiesta (REQ) e il secondo di notifica da parte del controller (ACK). Per il corretto funzionamento del protocollo è necessario che il segnale di REQ sia tenuto alto finché non viene ricevuto il corrispondente ACK.

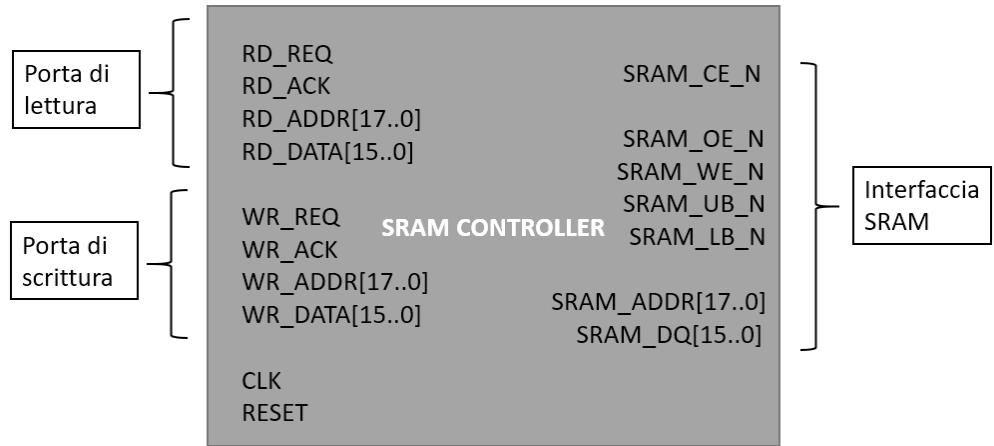


Figure 16: Interfaccia del controller della SRAM

Il meccanismo di arbitraggio realizzato si basa su una semplice macchina a stati: uno stato di scrittura, uno di lettura e un doppio stato di idle che ricorda quale è stata l'ultima richiesta elaborata (lettura o scrittura). Se il controller si trova in uno dei due stati di idle e riceve una richiesta contemporanea dalle due porte allora privilegerà la porta che non ha effettuato un trasferimento per ultima.

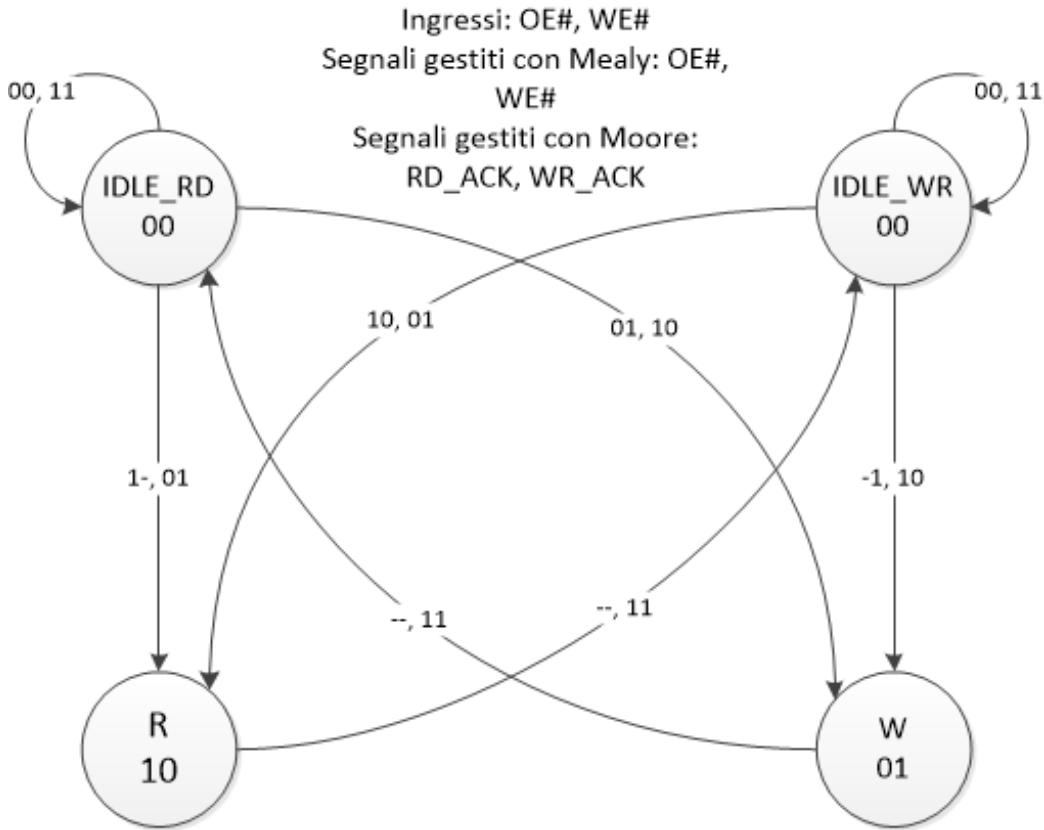


Figure 17: Macchina a stati del controller della SRAM

Per retrocompatibilità con la versione su DE1 è stato mantenuto il bus degli indirizzi a 18 bit. Nell'implementazione su DE2 è sufficiente cablare a 0 (o volendo ad un qualsiasi valore costante) i due bit più significativi aggiuntivi. I segnali di chip enable e di upper/lower byte control sono invece sempre abilitati (si trasferiscono sempre 16 bit alla volta). Si riporta infine il codice vhdl:

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity sram_controller is
port(
    -- READ CHANNEL --

```

```

RD_REQ:  in STD_LOGIC;
RD_ACK : out STD_LOGIC;
RD_ADDR: in STD_LOGIC_VECTOR(17 downto 0);
RD_DATA: out STD_LOGIC_VECTOR(15 downto 0);

-- WRITE CHANNEL --
WR_REQ:  in STD_LOGIC;
WR_ACK: out STD_LOGIC;
WR_ADDR: in STD_LOGIC_VECTOR(17 downto 0);
WR_DATA: in STD_LOGIC_VECTOR(15 downto 0);

-- SRAM BUS SIGNALS --
SRAM_CE_N, SRAM_OE_N, SRAM_WE_N, SRAM_UB_N, SRAM_LB_N: out
    STD_LOGIC;
SRAM_ADDR: out STD_LOGIC_VECTOR(17 downto 0);
SRAM_DQ: inout STD_LOGIC_VECTOR(15 downto 0);

-- CLK and RESET --
CLK, RESET: in STD_LOGIC

);

end sram_controller;

architecture sram_controller_arch of sram_controller is

type state_type is (IDLE_RD, IDLE_WR, R, W);
signal curr_state, next_state : state_type;
-- altri segnali interni
signal rd_data_reg : std_logic_vector(15 downto 0);

begin

sync_proc: process(CLK, next_state, RESET)
begin
  if (RESET = '1') then
    curr_state <= IDLE_RD;
  elsif(rising_edge(CLK)) then
    curr_state <= next_state;

```

```

    end if;
end process sync_proc;

state_proc: process(curr_state)
    variable rdwr : std_logic_vector(1 downto 0);
begin
    rdwr := RD_REQ & WR_REQ;
    -- pre assign output --
    RD_ACK <= '0'; WR_ACK <= '0';
    SRAM_ADDR <= (others => '0');
    SRAM_DQ <= (others => 'Z');
    SRAM_OE_N <= '1'; SRAM_WE_N <= '1';

    case curr_state is
        when IDLE_RD =>

            case rdwr is
                when "00" =>
                    next_state <= IDLE_RD;
                when "01" =>
                    SRAM_WE_N <= '0';
                    SRAM_DQ <= WR_DATA;
                    SRAM_ADDR <= WR_ADDR;
                    next_state <= W;
                when "10" | "11" =>
                    SRAM_OE_N <= '0';
                    SRAM_ADDR <= RD_ADDR;
                    next_state <= R;
                when others =>
                    next_state <= IDLE_RD;

            end case;

        when IDLE_WR =>

            case rdwr is
                when "00" =>

```

```

        next_state <= IDLE_WR;
when "01" | "11" =>
    SRAM_WE_N <= '0';
    SRAM_DQ <= WR_DATA;
    SRAM_ADDR <= WR_ADDR;
    next_state <= W;
when "10" =>
    SRAM_OE_N <= '0';
    SRAM_ADDR <= RD_ADDR;
    next_state <= R;
when others =>
    next_state <= IDLE_WR;

end case;

when R =>

    RD_ACK <= '1';
    rd_data_reg <= SRAM_DQ;
    SRAM_ADDR <= RD_ADDR;
    next_state <= IDLE_WR;

when W =>

    WR_ACK <= '1';
    SRAM_ADDR <= WR_ADDR;
    next_state <= IDLE_RD;

when others => -- should never happen
    next_state <= IDLE_RD;
    RD_ACK <= '0'; WR_ACK <= '0';
    rd_data_reg <= (others => '0');
    SRAM_ADDR <= (others => '0');
    SRAM_DQ <= (others => 'Z');
    SRAM_OE_N <= '1'; SRAM_WE_N <= '1';

end case;

```

```

end process state_proc;

-- assegnamenti asincroni
RD_DATA <= rd_data_reg;

-- sempre abilitati
SRAM_CE_N <= '0'; SRAM_UB_N <= '0'; SRAM_LB_N <= '0';

end sram_controller_arch;

```

5.3 Implementazione dei moduli di scrittura e lettura dell'immagine

Una volta creato l'arbitro per la SRAM sono stati implementati gli specifici moduli di scrittura e lettura dei frame, denominati *sram_writer* e *sram_reader*. L'interfaccia e il funzionamento di questi moduli è pressoché duale: il writer bufferizza internamente due pixel, fa una richiesta al controller e una volta ricevuto il segnale di avvenuta scrittura ricomincia il ciclo; il reader invece fa una richiesta di lettura e, una volta ricevuto il segnale di avvenuta lettura, bufferizza i due pixel letti e li manda in uscita in maniera sequenziale. Poiché il reader deve interfacciarsi con un'altra fifo asincrona, in quanto la VGA ha necessità di funzionare ad un clock ben preciso (circa 25 MHz), è necessario prevedere anche un segnale di handshake che va dalla fifo al reader. Poiché la VGA infatti è molto più lenta del reader, la fifo sarà quasi sempre piena. Il segnale *wrfull* della fifo deve quindi essere gestito opportunamente dal reader. Si riportano quindi le interfacce dei componenti *sram_writer* e *sram_reader*:

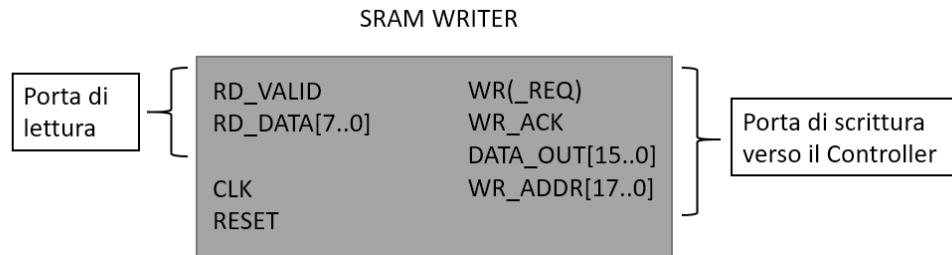


Figure 18: Interfaccia *sram_writer*

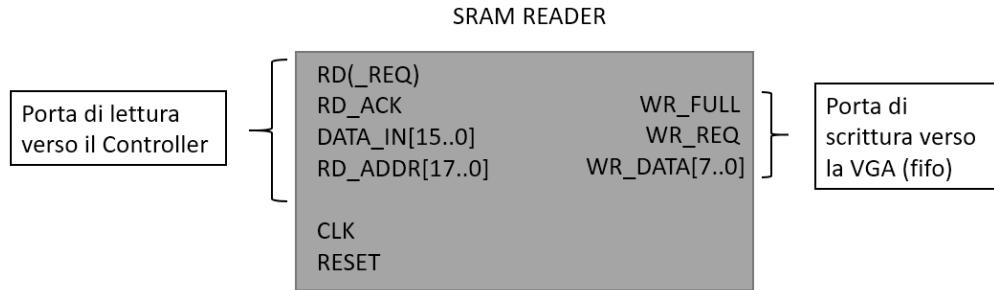


Figure 19: Interfaccia sram_reader

Il modulo sram_writer è governato da una macchina a stati che comprende due stati:

- RD_FIFO: il modulo è in attesa di leggere dalla porta di lettura i pixel da salvare. Tali pixel vengono salvati quando RD_VALID è alto. Nel momento in cui viene salvato l'ultimo dei due byte, il modulo invia una richiesta di scrittura al controller e transita nello stato WR_SRAM.
- WR_SRAM: il modulo ha effettuato una richiesta di scrittura e tiene a 1 il segnale di WR finché non riceve la notifica dal controller. Una volta ricevuta tale notifica, torna nello stato RD_FIFO; per aumentare la responsiveness del writer, nel momento in cui riceve la notifica esso verifica già la presenza di un dato sulla porta di lettura.

Si riporta il codice vhdl del componente sram_writer:

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;
use work.global.all;

entity sram_writer is
port(
    -- INPUT CHANNEL --
    RD_VALID: in STD_LOGIC;
    RD_DATA: in STD_LOGIC_VECTOR(7 downto 0);

```

```

-- SRAM CONTROLLER CHANNEL --
WR_ACK : in STD_LOGIC;
DATA_OUT: out STD_LOGIC_VECTOR(15 downto 0);
WR_ADDR : out STD_LOGIC_VECTOR(17 downto 0);
WR       : out STD_LOGIC;

-- CLK and RESET --
CLK, RESET: in STD_LOGIC

);

end sram_writer;

architecture sram_writer_arch of sram_writer is

type state_type is (RD_FIFO, WR_SRAM);
signal curr_state, next_state : state_type;

signal TEMP_DATA : std_logic_vector(15 downto 0);
signal COUNT, next_count : integer range 0 to 1 := 0;
signal ADDRESS, next_address : integer range 0 to IMG_WIDTH *
→ IMG_HEIGHT / 2 - 1 := 0;

begin

sync_proc: process(CLK, next_state, RESET)
begin
  if (RESET = '1') then
    COUNT <= 0;
    ADDRESS <= 0;
    curr_state <= RD_FIFO;
  elsif(rising_edge(CLK)) then
    curr_state <= next_state;
    ADDRESS <= next_address;
    COUNT <= next_count;
  end if;
end process sync_proc;

state_proc: process(curr_state)

```

```

begin
    -- pre assign output --
    next_count <= COUNT;
    next_address <= ADDRESS;

    case curr_state is

        when WR_SRAM =>
            if (WR_ACK = '0') then
                WR <= '1';
                next_state <= WR_SRAM;
            else

                next_address <= (ADDRESS + 1) mod (IMG_WIDTH *
                    → IMG_HEIGHT / 2);
                next_state <= RD_FIFO;
                WR <= '0';
                if (RD_VALID = '1') then
                    TEMP_DATA(7 downto 0) <= RD_DATA;
                    next_count <= 1;
                else
                    next_count <= 0;
                end if;
            end if;

        when RD_FIFO =>

            if (RD_VALID = '1') then

                if (COUNT = 0) then
                    WR <= '0';
                    TEMP_DATA(7 downto 0) <= RD_DATA;
                    next_count <= COUNT + 1;
                    next_state <= RD_FIFO;
                else
                    WR <= '1';
                    TEMP_DATA(15 downto 8) <= RD_DATA;
                end if;
            end if;
    end case;
end process;

```

```

        DATA_OUT <= TEMP_DATA;
        next_state <= WR_SRAM;
    end if;

    else
        WR <= '0';
        next_state <= RD_FIFO;
    end if;

end case;
end process state_proc;

-- assegnamenti asincroni
WR_ADDR <= std_logic_vector(to_unsigned(ADDRESS, 18));

end sram_writer_arch;

```

Il modulo sram_reader è governato da una macchina a stati duale, che comprende due stati:

- RD_SRAM: il modulo ha inviato una richiesta di lettura asserendo il segnale RD e permane in questo stato (mantenendo RD a 1) finché non riceve la notifica di avvenuta lettura da parte del controller. Una volta ricevuta la notifica transita in stato WR_FIFO. Durante la transizione, se la fifo non è piena, può già mandare in uscita il primo byte.
- WR_FIFO: il modulo deve mandare uno o entrambi i byte che ha letto. Nel momento in cui la fifo si dichiara non piena viene effettuato un trasferimento. Quando viene trasferito l'ultimo byte viene effettuata nuovamente una richiesta di lettura al controller, transitando nello stato RD_SRAM.

Si riporta il codice vhdl del componente sram_reader:

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;
use work.global.all;

entity sram_reader is
port(
    -- FIFO CHANNEL --
    WR_FULL : in STD_LOGIC;
    WR_REQ  : out STD_LOGIC;
    WR_DATA : out STD_LOGIC_VECTOR(7 downto 0);

    -- SRAM CONTROLLER CHANNEL --
    RD_ACK  : in STD_LOGIC;
    DATA_IN : in STD_LOGIC_VECTOR(15 downto 0);
    RD_ADDR : out STD_LOGIC_VECTOR(17 downto 0);
    RD      : out STD_LOGIC;

    -- CLK and RESET --
    CLK, RESET: in STD_LOGIC
);

```

```

end sram_reader;

architecture sram_reader_arch of sram_reader is

type state_type is (WR_FIFO, RD_SRAM);
signal curr_state, next_state : state_type;
-- altri segnali interni
signal TEMP_DATA : std_logic_vector(15 downto 0);
signal COUNT, next_count : integer range 0 to 1 := 0;
signal ADDRESS, next_address : integer range 0 to IMG_WIDTH *
→ IMG_HEIGHT / 2 - 1 := 0;

begin

sync_proc: process(CLK, next_state, RESET)
begin
  if (RESET = '1') then
    COUNT <= 0;
    ADDRESS <= 0;
    curr_state <= RD_SRAM;
  elsif(rising_edge(CLK)) then
    curr_state <= next_state;
    ADDRESS <= next_address;
    COUNT <= next_count;
  end if;
end process sync_proc;

state_proc: process(curr_state)
begin
  -- pre assign output --
  next_count <= COUNT;
  next_address <= ADDRESS;

  case curr_state is

when RD_SRAM =>

  if (RD_ACK = '0') then

```

```

WR_REQ <= '0';
RD <= '1';
next_state <= RD_SRAM;
else
    next_address <= (ADDRESS + 1) mod (IMG_WIDTH *
        ↵ IMG_HEIGHT / 2);
    TEMP_DATA <= DATA_IN;
    RD <= '0';
    next_state <= WR_FIFO;

    if (WR_FULL = '0') then
        WR_REQ <= '1';
        WR_DATA <= TEMP_DATA(7 downto 0);
        next_count <= 1;
    else
        WR_REQ <= '0';
        next_count <= 0;
    end if;

end if;

when WR_FIFO =>

if (WR_FULL = '0') then
    WR_REQ <= '1';

    if (COUNT = 0) then
        RD <= '0';
        WR_DATA <= TEMP_DATA(7 downto 0);
        next_count <= COUNT + 1;
        next_state <= WR_FIFO;
    else
        RD <= '1';
        WR_DATA <= TEMP_DATA(15 downto 8);
        next_state <= RD_SRAM;
    end if;

```

```

        else
            WR_REQ <= '0';
            RD <= '0';
            next_state <= WR_FIFO;
        end if;

    end case;
end process state_proc;

-- assegnamenti asincroni
RD_ADDR <= std_logic_vector(to_unsigned(ADDRESS, 18));

end sram_reader_arch;

```

5.4 Incapsulamento e interfacciamento con l'architettura

I tre moduli realizzati sono stati collegati e incapsulati in un unico componente denominato *frame_buffer*. Si riporta il codice del frame_buffer (si tratta solamente di 'glue code' con l'uso del port map) e lo schematico del collegamento del modulo al resto della pipeline.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY FRAME_BUFFER IS
PORT(
    -- IN SIGNALS
    RD_VALID: IN STD_LOGIC;
    RD_DATA: IN STD_LOGIC_VECTOR(7 downto 0);

    --SRAM
    SRAM_CE_N, SRAM_OE_N, SRAM_WE_N, SRAM_UB_N, SRAM_LB_N: OUT
        STD_LOGIC;
    SRAM_ADDR: OUT STD_LOGIC_VECTOR(17 downto 0);
    SRAM_DQ: INOUT STD_LOGIC_VECTOR(15 downto 0);

    -- OUT SIGNALS

```

```

WR_FULL: IN STD_LOGIC;
WR_REQ: OUT STD_LOGIC;
WR_DATA: OUT STD_LOGIC_VECTOR(7 downto 0);

--CLOCK AND RESET
CLK, RESET: IN STD_LOGIC
);
END FRAME_BUFFER;

ARCHITECTURE FRAME_BUFFER_ARCH OF FRAME_BUFFER IS
--BETWEEN WRITER AND CONTROLLER
SIGNAL WR_ACK_REG, WR_REQ_REG: STD_LOGIC;
SIGNAL WR_DATA_REG: STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL WR_ADDR_REG: STD_LOGIC_VECTOR(17 DOWNTO 0);

--BETWEEN READER AND CONTROLLER
SIGNAL RD_REQ_REG: STD_LOGIC;
SIGNAL RD_ACK_REG: STD_LOGIC;
SIGNAL RD_ADDR_REG: STD_LOGIC_VECTOR(17 downto 0);
SIGNAL RD_DATA_REG: STD_LOGIC_VECTOR(15 downto 0);

COMPONENT SRAM_WRITER IS
PORT(
-- FIFO CHANNEL --
RD_VALID: IN STD_LOGIC;
RD_DATA: IN STD_LOGIC_VECTOR(7 downto 0);

-- SRAM CONTROLLER CHANNEL --
WR_ACK : IN STD_LOGIC;
DATA_OUT: OUT STD_LOGIC_VECTOR(15 downto 0);
WR_ADDR : OUT STD_LOGIC_VECTOR(17 downto 0);
WR      : OUT STD_LOGIC;

-- CLK and RESET --
CLK, RESET: IN STD_LOGIC
);
END COMPONENT SRAM_WRITER;

```

```

COMPONENT SRAM_CONTROLLER IS
PORT(
    -- READ CHANNEL --
    RD_REQ: IN STD_LOGIC;
    RD_ACK: OUT STD_LOGIC;
    RD_ADDR: IN STD_LOGIC_VECTOR(17 downto 0);
    RD_DATA: OUT STD_LOGIC_VECTOR(15 downto 0);

    -- WRITE CHANNEL --
    WR_REQ: IN STD_LOGIC;
    WR_ACK: OUT STD_LOGIC;
    WR_ADDR: IN STD_LOGIC_VECTOR(17 downto 0);
    WR_DATA: IN STD_LOGIC_VECTOR(15 downto 0);

    -- SRAM BUS SIGNALS --
    SRAM_CE_N, SRAM_OE_N, SRAM_WE_N, SRAM_UB_N, SRAM_LB_N: OUT
        STD_LOGIC;
    SRAM_ADDR: OUT STD_LOGIC_VECTOR(17 downto 0);
    SRAM_DQ: INOUT STD_LOGIC_VECTOR(15 downto 0);

    -- CLK and RESET --
    CLK, RESET: IN STD_LOGIC
);
END COMPONENT SRAM_CONTROLLER;

COMPONENT SRAM_READER IS
PORT(
    -- FIFO CHANNEL --
    WR_FULL: IN STD_LOGIC;
    WR_REQ: OUT STD_LOGIC;
    WR_DATA: OUT STD_LOGIC_VECTOR(7 downto 0);

    -- SRAM CONTROLLER CHANNEL --
    RD_ACK: IN STD_LOGIC;
    DATA_IN: IN STD_LOGIC_VECTOR(15 downto 0);
    RD_ADDR: OUT STD_LOGIC_VECTOR(17 downto 0);
    RD: OUT STD_LOGIC;

```

```

-- CLK and RESET --
CLK, RESET: IN STD_LOGIC
);
END COMPONENT SRAM_READER;

BEGIN
WRITER: SRAM_WRITER PORT MAP(RD_VALID, RD_DATA, WR_ACK_REG,
    ↳ WR_DATA_REG, WR_ADDR_REG, WR_REQ_REG, CLK, RESET);
CONTROLLER: SRAM_CONTROLLER PORT MAP(
    RD_REQ_REG, RD_ACK_REG, RD_ADDR_REG, RD_DATA_REG,
    WR_REQ_REG, WR_ACK_REG, WR_ADDR_REG, WR_DATA_REG,
    SRAM_CE_N, SRAM_OE_N, SRAM_WE_N, SRAM_UB_N, SRAM_LB_N,
    ↳ SRAM_ADDR, SRAM_DQ,
    CLK, RESET
);
READER: SRAM_READER PORT MAP(WR_FULL, WR_REQ, WR_DATA,
    ↳ RD_ACK_REG, RD_DATA_REG, RD_ADDR_REG, RD_REQ_REG, CLK,
    ↳ RESET);
END FRAME_BUFFER_ARCH;

```

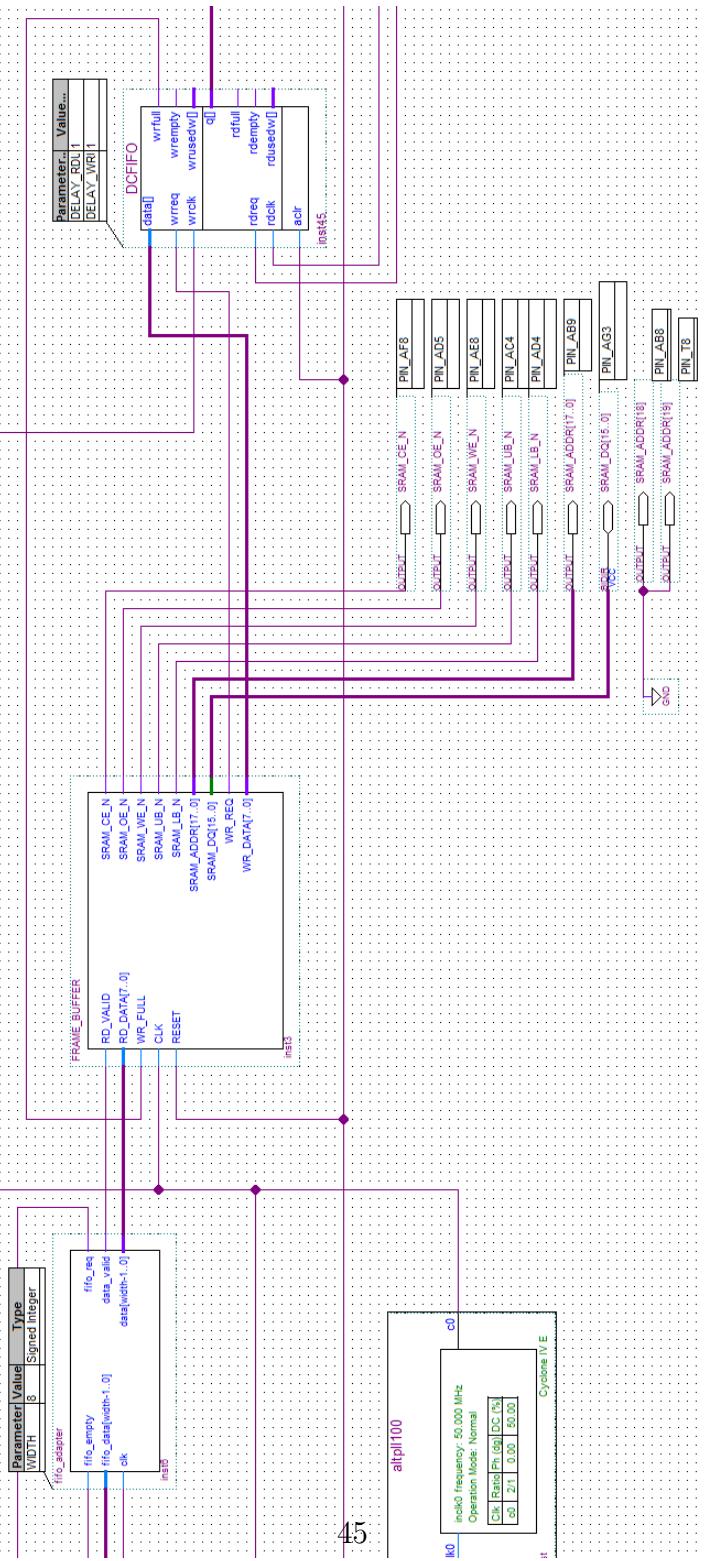


Figure 20: Interfacciamento del modulo frame_buffer

6 Interfacciamento alla VGA

Per visualizzare un’immagine un display riceve i tre segnali R, G e B che caratterizzano i pixel e, partendo dall’angolo in alto a sinistra, li inserisce uno alla volta per righe. Raggiunta la fine della prima riga l’indirizzo della colonna viene portato a zero e quello di riga incrementato di uno; al termine dell’ultima riga il processo di aggiornamento del display riprende dall’inizio. La risoluzione del flusso video proveniente dal sensore utilizzato nel progetto è 640×480 pixel e imponendo una velocità di refresh di 60Hz risultano approssimativamente 40ns per pixel. Il controller per l’interfacciamento alla VGA, che è stato chiamato vga_controller, di conseguenza avrà in ingresso un clock da 25Mhz (generato attraverso un PLL). Come anticipato precedentemente anche questo modulo è separato dalla parte centrale di interfacciamento alla SRAM attraverso una fifo asincrona; da tale fifo è possibile recuperare il valore del successivo pixel da inviare al display (in grayscale le tre componenti RGB hanno lo stesso valore). Per visualizzare correttamente un’immagine sul display non basta semplicemente inviare i tre segnali RGB con la giusta frequenza ma sono necessari due segnali di sincronismo:

- HSYNC: quando l’horizontal sync viene asserito (in logica negata) indica al monitor la fine di una riga.
- VSYNC: quando il vertical sync viene asserito (in logica negata) si indica al monitor che l’immagine è terminata ed è possibile iniziare un altro ciclo di refresh.

Entrambi i segnali devono rimanere nello stato logico alto mentre i pixel vengono disegnati (active region), vengono poi portati allo stato logico basso dopo un tempo detto *front porch* (vertical e horizontal), vi rimangono per un tempo *sync pulse* ed infine tornano nello stato iniziale per un tempo *back porch* prima di passare alla nuova riga o alla nuova immagine. La durata dei tempi *sync pulse*, *front porch* e *back porch* sia vertical che horizontal è calcolata a partire dalla risoluzione. L’insieme di tutti questi periodi dà origine alla cosiddetta *blanking region*. Nella blanking region i segnali RGB devono essere a "0" e nessun pixel viene effettivamente disegnato.

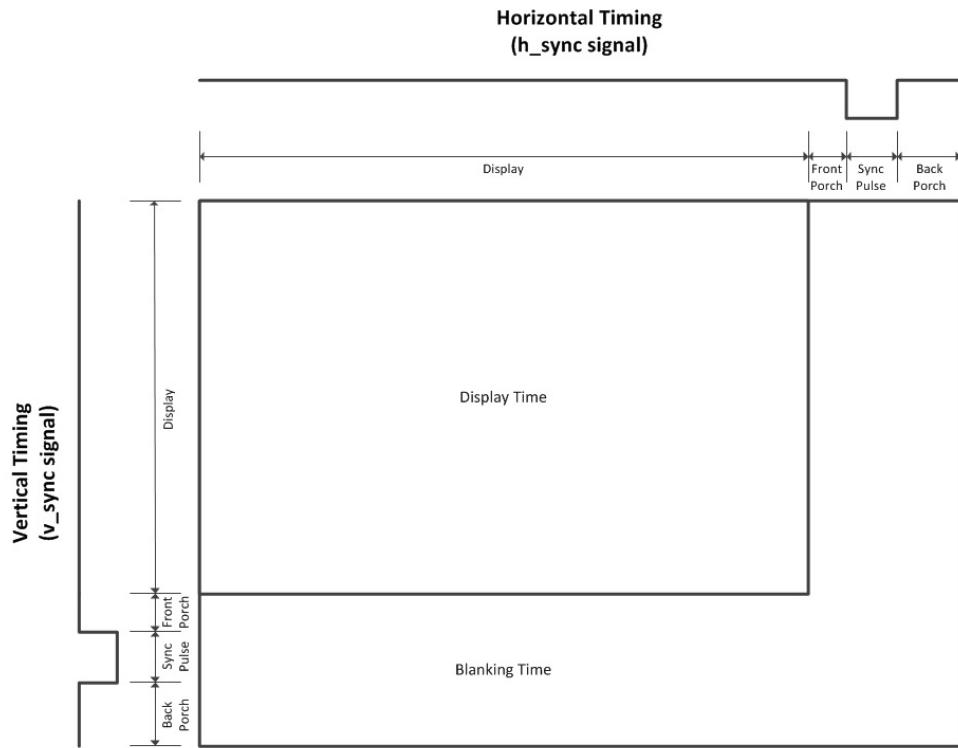


Figure 21: VGA timings

Il controller riceve in ingresso direttamente i tre segnali RGB e non il singolo pixel grayscale, in modo da permettere successivamente di visualizzare i template trovati tramite dei colori (la gestione verrà fatta a monte del vga controller). La risoluzione dell'immagine e i timing sono stati parametrizzati in modo da garantire la riusabilità. Ad ogni clock viene aggiornata la posizione del pixel corrente, tenendo conto anche della blanking region durante la quale i segnali RGB in uscita saranno a '0'. Inoltre durante il *Vertical/Horizontal Sync Pulse* vengono portati a '0' rispettivamente VSYNC e HSYNC. E' presente infine un segnale, denominato *DATA_ACK*, retroazionato ai moduli a monte della pipeline, in modo da permettere la corretta sincronizzazione con i timing della VGA. I moduli a monte devono portare in uscita i colori da disegnare prima ancora di ricevere il segnale *DATA_ACK* a 1, che segnala pertanto l'avvenuta lettura del dato da parte della vga (i moduli a monte al prossimo clock dovranno portare in uscita i colori del prossimo pixel).

Si riporta infine il codice del vga_controller:

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

ENTITY VGA_CONTROLLER IS
PORT(
    HSYNC: OUT STD_LOGIC;
    VSYNC: OUT STD_LOGIC;
    R: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    G: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    B: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    R_IN: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    G_IN: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    B_IN: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    DATA_ACK: OUT STD_LOGIC;

    CLK: IN STD_LOGIC;
    RST: IN STD_LOGIC
);
END VGA_CONTROLLER;

ARCHITECTURE VGA_CONTROLLER_ARCH OF VGA_CONTROLLER IS
--640x480 @ 60 Hz pixel clock 25.175MHz

CONSTANT HOR: INTEGER := 640;
CONSTANT HORIZONTAL_FP: INTEGER := 16;
CONSTANT HORIZONTAL_BP: INTEGER := 48;
CONSTANT HORIZONTAL_SYNC_PULSE: INTEGER := 96;

CONSTANT H_STARTSYNC: INTEGER := HOR + HORIZONTAL_FP;
CONSTANT H_ENDSYNC: INTEGER := H_STARTSYNC +
    HORIZONTAL_SYNC_PULSE;
CONSTANT HOR_TOT: INTEGER := HOR + HORIZONTAL_BP +
    HORIZONTAL_FP + HORIZONTAL_SYNC_PULSE;

CONSTANT VER: INTEGER := 480;

```

```

CONSTANT VERTICAL_FP: INTEGER := 10;
CONSTANT VERTICAL_BP: INTEGER := 33;
CONSTANT VERTICAL_SYNC_PULSE: INTEGER := 2;

CONSTANT V_STARTSYNC: INTEGER := VER + VERTICAL_FP;
CONSTANT V_ENDSYNC: INTEGER := V_STARTSYNC +
    VERTICAL_SYNC_PULSE;
CONSTANT VER_TOT: INTEGER := VER + VERTICAL_BP + VERTICAL_FP
    + VERTICAL_SYNC_PULSE;

SIGNAL BLANK: STD_LOGIC := '1';
BEGIN

PROCESS(CLK, RST)
VARIABLE HPOS: INTEGER RANGE 0 TO HOR_TOT-1:=0;
VARIABLE VPOS: INTEGER RANGE 0 TO VER_TOT-1:=0;
BEGIN
    IF(RST = '1')
    THEN
        HPOS := 0;
        VPOS := 0;
        BLANK <= '1';
        R<=(others=>'0');
        G<=(others=>'0');
        B<=(others=>'0');

ELSIF(RISING_EDGE(CLK))
THEN
    --AGGIORNAMENTO POSIZIONE
    IF(HPOS = HOR_TOT-1)
    THEN
        HPOS := 0;
        VPOS := (VPOS + 1) MOD VER_TOT;
    ELSE
        HPOS := HPOS+1;
    END IF;
    --FINE AGGIORNAMENTO

```

```

--GESTIONE FLUSSO OUTPUT
IF(BLANK = '1')
THEN
  R<=(others=>'0');
  G<=(others=>'0');
  B<=(others=>'0');
ELSE
  R <= R_IN;
  G <= G_IN;
  B <= B_IN;
END IF;
--FINE GESTIONE FLUSSO OUTPUT

--BLANK TIMING
IF(VPOS >= VER)
THEN
  BLANK <= '1';
ELSE
  IF(HPOS < HOR)
  THEN
    BLANK <='0';
  ELSE
    BLANK <='1';
  END IF;
END IF;
--FINE BLANK TIMING

-- SYNC TIMING
IF(HPOS > H_STARTSYNC AND HPOS <= H_ENDSYNC)
THEN
  HSYNC <= '0';
ELSE
  HSYNC <= '1';
END IF;

IF(VPOS >= V_STARTSYNC AND VPOS < V_ENDSYNC)
THEN
  VSYNC <= '0';

```

```
    ELSE
        VSYNC <= '1';
    END IF;
    -- FINE SYNC TIMING
END IF;
END PROCESS;

--LOGICA NEGATA
DATA_ACK <= NOT BLANK;

END VGA_CONTROLLER_ARCH;
```

7 Template Matching

Prima di entrare nei dettagli della pipeline di riconoscimento dei template, è bene specificare che la scelta progettuale è stata quella di riconoscere come match al massimo una sola porzione di immagine per ogni frame, quella che combacia maggiormente con uno dei template. La scelta è stata fatta in relazione al problema di dover disegnare un numero di quadrati di match arbitrario sulla VGA; nel caso fosse necessario riconoscere più oggetti della stessa tipologia all'interno di una immagine, basterà però modificare uno soltanto dei componenti della pipeline, che è quello che si occupa di "interpretare" i punteggi calcolati.

Viene qui riportata la struttura della pipeline di template matching ai morsetti, che si compone di diverse porte:

- Porta di lettura pixel: vengono campionati i pixel in arrivo dalla camera (attraverso il componente fifo_adapter).
- Porta di programmazione template: permette ad ogni clock di programmare uno specifico pixel di un template; è formata dai seguenti segnali:
 - PROG_VALID: indica un dato valido per l'intervallo di clock considerato
 - PROG_INDEX: indica il numero binario del template che si vuole programmare
 - PROG_ROW, PROG_COL: indici di riga e colonna all'interno del template dove posizionare il pixel
 - PROG_PIXEL: pixel che si vuole inserire all'interno del template
- Porta di tuning: è composta da segnali di controllo che permettono di interagire con la pipeline modificandone parzialmente il comportamento; in particolare:
 - MASKED: è un vettore di bit lungo quanto il numero di template e permette di abilitare o disabilitare la ricerca di uno specifico template. Se MASKED[i] = 1 l'i-esimo template non potrà mai comparire in uscita come trovato.

- THRESHOLD_MULTIPLIER: è un valore che consente di impostare la soglia di riconoscimento dei template ad un valore più o meno alto. La modifica di questo valore permette di riconoscere a colpo d'occhio sulla VGA come a valori alti di soglia vengano riconosciuti template con maggiore facilità (fino ad arrivare a valori per cui il template viene trovato in zone in cui in realtà non è presente); viceversa, per valori bassi di soglia, solo le zone che combaciano fortemente con un template vengono riconosciute (per valori troppo bassi di soglia non viene riconosciuto alcun template).

Entrambi i segnali sono stati collegati agli switch della board (per un totale di 10 switch di tuning).

- Risultati in uscita: contiene le informazioni calcolate dalla pipeline; in particolare:
 - TEMPLATE_MATCHING_FINISHED: viene asserito per un clock al termine di ogni frame per segnalare che sono stati calcolati dei nuovi risultati in uscita.
 - TEMPLATE_MATCHING_FOUND: segnale che indica se è stato trovato un template all'interno dell'immagine.
 - TEMPLATE_MATCHING_WINNER: indica il numero binario del template vincitore, cioè quello per cui è stato trovato il punteggio migliore di match
 - TEMPLATE_MATCHING_ROW/COL: riga e colonna all'interno del frame che indicano il punto in alto a sinistra della regione di immagine che ha fatto match.

I segnali di found, winner, row e col sono mantenuti costanti in uscita fino a un loro futuro aggiornamento (al frame successivo). In questo modo anche i moduli che non hanno bisogno di ottenere le informazioni "al clock" possono usufruire dei risultati prodotti dalla pipeline. Come è possibile immaginare, i segnali di winner, row e col hanno significato solo in corrispondenza di found = 1.



Figure 22: Interfaccia della pipeline di template matching

Si riporta qui il codice delle costanti utilizzate per i moduli riguardanti il template matching. Il significato di tali costanti sarà spiegato lungo la trattazione della pipeline.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.global.all;

package convolution_package is

    constant COMPUTATION_STEPS : natural := 8;    -- numero di
    -- intervalli elementari di clock usati
    -- nel template matching e
    -- trasferimento window

    constant TEMPLATE_SIZE : natural := 32;        -- dimensione del
    -- lato del template (quadrato)

    subtype window_column is std_logic_vector((PIXEL_WIDTH *
    -- TEMPLATE_SIZE - 1) downto 0);
    type window_type is array(natural range 0 to TEMPLATE_SIZE -
    -- 1) of window_column;

    type window_packet is array(0 to (TEMPLATE_SIZE /
    -- COMPUTATION_STEPS - 1)) of window_column;

    constant SCORE_MAX : natural := (2**PIXEL_WIDTH - 1) *
    -- TEMPLATE_SIZE * TEMPLATE_SIZE;

    constant SCORE_WIDTH : natural := 18; -- 16 per 16, 18 per
    -- 32, 20 per 64

    subtype score_type is natural range 0 to 2**SCORE_WIDTH - 1;
    subtype score_type_vector is std_logic_vector(SCORE_WIDTH - 1
    -- downto 0);

    constant TEMPLATE_MATCHING_MODULES_WIDTH : natural := 2;

```

```

constant TEMPLATE_MATCHING_MODULES : natural := 4;

-- Non posso usare score_type (che è un natural) altrimenti
-- quartus mi dice che non
-- riesce a generare il symbol per una porta così
-- complicata..
type template_matching_score_array is array(0 to
    TEMPLATE_MATCHING_MODULES - 1) of score_type_vector;

constant DEFAULT_CAMERA_PROGRAMMING_ROW      : natural :=
    (IMG_HEIGHT - TEMPLATE_SIZE) / 2 - TEMPLATE_SIZE; --
-- sottraggo TEMPLATE_SIZE per evitare il punto sporco sulla
-- camera
constant DEFAULT_CAMERA_PROGRAMMING_COL     : natural :=
    (IMG_WIDTH - TEMPLATE_SIZE) / 2;

end package convolution_package;

```

7.1 Campionamento della finestra di confronto: window_extractor

Il primo blocco da implementare è quello che si occupa della generazione della finestra di scorrimento. Il template matching infatti, così come tanti altri algoritmi di computer vision quali i filtri di media e di edge detection (in generale i filtri di convoluzione), ha necessità di estrarre, in ogni posizione differente dell'immagine, una finestra di dimensioni pari a quella del template per poterla confrontare col template stesso. Questa finestra, mano a mano che i pixel di una riga arrivano in ingresso, deve scorrere verso destra (effettuando quindi uno shift a sinistra delle colonne). Nel momento in cui ci si sposta alla riga successiva, la finestra deve scendere di una riga (effettuando quindi uno shift verso l'alto delle righe). E' necessario inoltre prendere un punto di riferimento per stabilire in quale posizione si trovi il pixel corrente in ingresso; nel nostro caso si trova nella cella della finestra in basso a destra. Dunque la finestra contiene tutti i valori precedenti (relativi a un determinato numero di righe e di colonne) rispetto al pixel correntemente campionato.

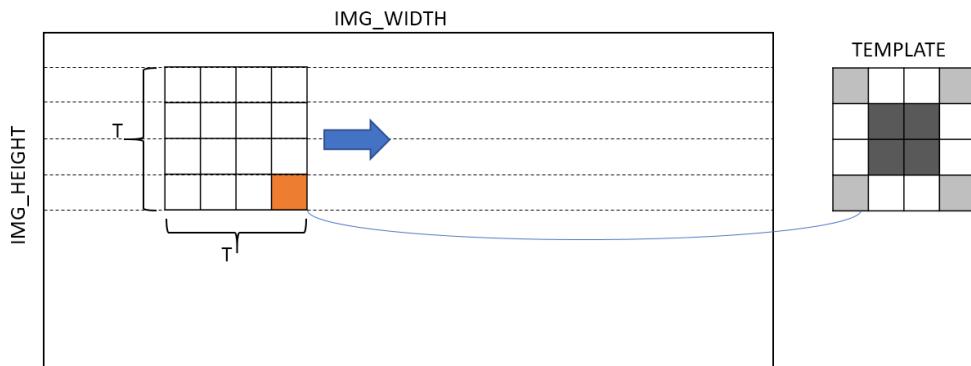


Figure 23: Scorrimento della finestra lungo l'immagine

Come è possibile notare, affinché il meccanismo funzioni e funzioni rapidamente, è necessario bufferizzare un numero di righe intere pari alla dimensione del template (in realtà è sufficiente TEMPLATE_SIZE - 1 poiché la riga in basso è quella correntemente campionata). Questo buffer, denominato usualmente *line buffer*, è di dimensioni molto grandi, pari a $IMG_WIDTH * (TEMPLATE_SIZE - 1)$ pixel, e può causare facilmente problemi di risorse

per chip con poca memoria integrata. Questa memoria è implementata in vhdl utilizzando un semplice template Altera, riportato qui sotto, che realizza la funzione di una ram con una porta di scrittura e una porta di lettura; per ogni pixel infatti sarà necessario effettuare una lettura (leggere una colonna) e una scrittura (fare lo shift verso l'alto della colonna corrente inserendo in coda il pixel corrente). Nel nostro caso DATA_NUM è pari al numero di colonne (640), mentre DATA_WIDTH è pari al numero di righe moltiplicato per la width di un pixel ((TEMPLATE_SIZE - 1) * PIXEL_WIDTH). Si accede dunque a questa memoria con il numero della colonna e si ottiene come dato il contenuto della colonna richiesta.

```
-- Quartus II VHDL Template
-- Simple Dual-Port RAM with different read/write addresses
-- but
-- single read/write clock

library ieee;
use ieee.std_logic_1164.all;

entity simple_dual_port_ram_single_clock is

generic
(
    DATA_WIDTH : natural;
    DATA_NUM : natural
);

port
(
    clk : in std_logic;
    raddr : in natural range 0 to DATA_NUM - 1;
    waddr : in natural range 0 to DATA_NUM - 1;
    data : in std_logic_vector((DATA_WIDTH-1) downto 0);
    we : in std_logic := '1';
    q : out std_logic_vector((DATA_WIDTH -1) downto 0)
);
```

```

end simple_dual_port_ram_single_clock;

architecture rtl of simple_dual_port_ram_single_clock is

    -- Build a 2-D array type for the RAM
    subtype word_t is std_logic_vector((DATA_WIDTH-1) downto 0);
    type memory_t is array(DATA_NUM-1 downto 0) of word_t;

    -- Declare the RAM signal.
    signal ram : memory_t;

begin

    process(clk)
    begin
        if(rising_edge(clk)) then
            if(we = '1') then
                ram(waddr) <= data;
            end if;

            -- On a read during a write to the same address, the read
            -- will
            -- return the OLD data at the address
            q <= ram(raddr);
        end if;
    end process;

end rtl;

```

Le funzioni che il modulo adibito alla generazione della finestra, denominato *window_extractor*, deve compiere sono cicliche e si ripetono ogni volta che un nuovo pixel arriva in ingresso al componente; tali funzioni sono qui riassunte:

1. Effettuare una shift verso sinistra delle colonne della finestra.
2. Leggere una colonna dal line buffer, concatenarla al pixel corrente, e inserirla nella colonna più destra della finestra (quella rimasta libera dallo shift).

3. Effettuare uno shift verso l'alto della colonna del line buffer corrente e inserire in basso il pixel corrente; una volta effettuata questa operazione la nuova colonna viene riscritta nel line buffer. Questa colonna rappresenta il valore aggiornato della colonna nel momento in cui la finestra ripasserà per quella colonna alla prossima riga.

La finestra, di tipo *window_type*, è stata definita come un array di colonne (*window_column*). In questo modo è possibile effettuare facilmente l'operazione di shift delle colonne.

L'ultima ottimizzazione effettuata per il modulo *window_extractor* riguarda la trasmissione della finestra in uscita. Se consideriamo una finestra di 32×32 pixel, ogni volta che aggiorniamo la posizione della finestra è necessario mandare in uscita $32 \times 32 \times 8 = 8192$ bit. Un parallelismo così elevato rappresenta un problema non da poco per quanto riguarda il consumo di risorse. Per ridurre il parallelismo quanto più possibile, è stato considerato il fatto che la frequenza di arrivo dei pixel è, come è già stato sottolineato, 8 volte inferiore alla frequenza di lavoro dei moduli (12 MHz contro 100 MHz). Pertanto, una volta modificata opportunamente la finestra e aggiornato il line buffer come descritto nei passi precedenti, è possibile mandare in uscita la finestra in 8 passi consecutivi: considerando ancora il caso di template di dimensione 32, l'uscita passa così da 8192 a 1024 bit. La riduzione del parallelismo ha comportato però un overhead di gestione in più: l'uscita del *window_extractor* è infatti composta da un segnale di validità, da un frammento (denominato *window_packet*) della finestra e da un numero di sequenza compreso tra 0 e 7 che il *window_extractor* deve settare opportunamente. I moduli che utilizzano la finestra dovranno pertanto adeguarsi a questo protocollo. Nel caso di algoritmi di template matching questa frammentazione porta solamente dei vantaggi, in quanto basterà predisporre le risorse di calcolo per un solo frammento di finestra e non per la finestra intera.

Si riporta infine il codice del *window_extractor*:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.global.all;
use work.convolution_package.all;
```

```

entity window_extractor is

port
(
    pixel_in_valid : in std_logic;
    pixel_in : in pixel;

    window_valid : out std_logic;
    window_seq : out natural range 0 to COMPUTATION_STEPS - 1;
    window_out : out window_packet;

    clk : in std_logic;
    reset : in std_logic
);

end window_extractor;

architecture window_extractor_arch of window_extractor is

component simple_dual_port_ram_single_clock is

generic
(
    DATA_WIDTH : natural;
    DATA_NUM : natural
);

port
(
    clk : in std_logic;
    raddr : in natural range 0 to DATA_NUM - 1;
    waddr : in natural range 0 to DATA_NUM - 1;
    data : in std_logic_vector((DATA_WIDTH-1) downto 0);
    we : in std_logic;
    q : out std_logic_vector((DATA_WIDTH -1) downto 0)
);

```

```

end component simple_dual_port_ram_single_clock;

-- line buffer

signal raddr, waddr : natural range 0 to IMG_WIDTH - 1;
signal line_buffer_data_rd, line_buffer_data_wr :
  std_logic_vector(((PIXEL_WIDTH * (TEMPLATE_SIZE - 1)) - 1)
  downto 0);
signal wen : std_logic;

-- state

type state_type is (READ_PIXEL, TRANSFER_WINDOW);

begin

line_buffer: simple_dual_port_ram_single_clock
generic map
(
  DATA_WIDTH => (PIXEL_WIDTH * (TEMPLATE_SIZE - 1)),
  DATA_NUM => IMG_WIDTH
)
port map
(
  clk  => clk,
  raddr  => raddr,
  waddr  => waddr,
  data  => line_buffer_data_wr,
  we    => wen,
  q     => line_buffer_data_rd
);

process (clk, reset)

-- window

variable window : window_type;

```

```

variable window_column_temp : window_column;

variable col : natural range 0 to IMG_WIDTH - 1 := 0;
variable step_counter : natural range 0 to
    ↵ COMPUTATION_STEPS - 1 := 0;
variable start_index : natural range 0 to TEMPLATE_SIZE - 1
    ↵ := 0;

variable line_buffer_temp : std_logic_vector(((PIXEL_WIDTH
    ↵ * (TEMPLATE_SIZE - 1)) - 1) downto 0);

begin
    if (rising_edge(clk)) then
        if (reset = '1') then
            -- sync reset
            col := 0;
            step_counter := 0;
        else

            window_valid <= '0';
            wen <= '0';
            raddr <= col;
            waddr <= col;

            if (step_counter = 0) then

                if (pixel_in_valid = '1') then

                    -- lettura colonna corrente dal line buffer
                    line_buffer_temp := line_buffer_data_rd;

                    -- shift a sinistra della window
                    for i in 0 to TEMPLATE_SIZE - 2 loop
                        window(i) := window(i + 1);
                    end loop;

```

```

-- assegnamento alla window della colonna corrente
→ del line buffer
window(TEMPLATE_SIZE - 1) := line_buffer_temp &
→ pixel_in;

line_buffer_temp := line_buffer_temp(((PIXEL_WIDTH
→ * (TEMPLATE_SIZE - 2)) - 1) downto 0) &
→ pixel_in;

-- shift verso l'alto della colonna del line buffer
→ e
-- scrittura in ram della colonna del line buffer
wen <= '1';
line_buffer_data_wr <= line_buffer_temp;

-- In uscita mettiamo le colonne meno significative
→ della finestra
start_index := 0;

window_valid <= '1';

for I in 0 to TEMPLATE_SIZE / COMPUTATION_STEPS - 1
→ loop
    window_out(I) <= window(I);
end loop;

step_counter := 1; -- cambio stato

-- Incremento il contatore di colonna
col := (col + 1) mod IMG_WIDTH;

end if;

else -- trasferimento finestra

window_valid <= '1';

```

```

-- aggiornamento finestra di uscita
start_index := start_index + TEMPLATE_SIZE /
    COMPUTATION_STEPS;

for I in 0 to TEMPLATE_SIZE / COMPUTATION_STEPS - 1
    loop
        window_out(I) <= window(start_index + I);
    end loop;

step_counter := (step_counter + 1) mod
    COMPUTATION_STEPS;
-- se step_counter = 0, cambio stato

end if; -- step_counter

end if; -- reset

window_seq <= step_counter - 1;

end if; -- clock

end process;

end architecture;

```

7.2 SAD e template_finder

Il modulo che utilizza la finestra generata dal window_extractor è il componente adibito al confronto della finestra con il template. Il modulo realizzato, denominato *SAD* come l'algoritmo che implementa, rappresenta pertanto il cuore della pipeline di template matching.

Compito del modulo SAD è di ricevere la finestra, effettuare la differenza in valore assoluto tra i pixel corrispondenti della finestra e del template, sommare le differenza e mandare in uscita tale somma, che rappresenta il punteggio (denominato *score*) calcolato per quella determinata posizione all'interno dell'immagine. Il punteggio massimo, ottenibile confrontando un template completamente nero con una finestra di immagine completamente bianca (o

viceversa), è pari a $(2^{PIXEL_WIDTH} - 1) \times TEMPLATE_SIZE^2$: nel caso di template di dimensione 32 (e pixel di 8 bit) corrisponde ad un valore esprimibile con 18 bit (unsigned).

L'implementazione del modulo sfrutta ancora una volta il meccanismo del pipelining ed è realizzabile grazie al fatto di aver suddiviso la finestra in vari frammenti.

I tre stadi della pipeline hanno i seguenti compiti:

- Il primo stadio riceve il frammento di finestra e lo confronta con il corrispondente frammento del template, facendo la differenza in valore assoluto tra i pixel corrispondenti. Il risultato di questa operazione è perciò a sua volta un frammento di finestra, che viene inviato al secondo stadio.
- Il secondo stadio calcola la somma dei valori all'interno della finestra inviatagli dal primo stadio e invia il risultato al terzo stadio.
- Il terzo stadio raccoglie le somme parziali del secondo stadio e le somma ottenendo il punteggio finale. Tale punteggio viene considerato raggiunto quando viene ricevuto l'ultimo frammento di finestra. A questo punto viene inviata in uscita la somma totale assieme al corrispondente segnale di validità. Il registro accumulatore delle somme viene infine resettato per ricominciare il ciclo.

Oltre alla parte algoritmica è necessario prevedere una porta di programmazione del template, che è contenuto all'interno del modulo stesso: i segnali necessari sono un segnale di validità, il pixel che si vuole inserire e l'indicazione di riga e colonna all'interno del template. Si noti che nella realizzazione vhdl al momento del reset i bit del template vengono tutti impostati a 1; in questo modo è stato possibile debuggare la pipeline di template matching prima ancora di realizzare la parte di programmazione dei template: il modulo SAD viene di fatti programmato per riconoscere una finestra completamente bianca.

Si riporta infine il codice del modulo SAD:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.global.all;
use work.convolution_package.all;

-- Template matching implementato con algoritmo SAD --
-- Confronto la finestra con il template facendo la
-- differenza in valore assoluto tra pixel corrispondenti
-- e sommando tali differenze.

-- Il modulo è implementato con pipeline a 3 stadi:
-- I stadio: faccio la differenza in valore assoluto in
-- → parallelo
--         all'interno del frammento di finestra ricevuto
-- II stadio: faccio la somma tra tutte le differenze nel
-- → frammento
--         di finestra (somma parziale)
-- III stadio: faccio la somma totale (score) di tutte le somme
-- → parziali e la
--         mando in uscita se sono nell'ultimo frammento
entity SAD is
port
(
    -- Input ports
    window_valid      : in std_logic;
    window_seq        : in natural range 0 to COMPUTATION_STEPS -
    → 1;
    window_in         : in window_packet;

    -- Input programmazione template
    pixel_valid       : in std_logic;
    pixel_in          : in pixel;
    pixel_row         : in natural range 0 to TEMPLATE_SIZE - 1;
    pixel_col         : in natural range 0 to TEMPLATE_SIZE - 1;
```

```

-- Output ports
score_valid      : out std_logic;
score           : out score_type_vector;

clk : in std_logic;
reset : in std_logic
);
end SAD;

architecture SAD_arch of SAD is

signal window_valid_1, window_valid_2 : std_logic;
signal window_seq_1, window_seq_2 : natural range 0 to
    → COMPUTATION_STEPS - 1;
signal template : window_type;
signal temp_sad : window_packet;
signal partial_sum_2_3 : integer range 0 to SCORE_MAX /
    → COMPUTATION_STEPS - 1;

begin
process(clk)
-- stage 0 --
variable window_pixel : pixel_int;
variable template_pixel : pixel_int;
-- stage 1 --
variable partial_sum : integer range 0 to SCORE_MAX /
    → COMPUTATION_STEPS - 1;
-- stage 2 --
variable total_sum : score_type;

variable programming_row : natural range 0 to TEMPLATE_SIZE -
    → 1;
begin
if (rising_edge(clk)) then

```

```

if (reset = '1') then
    score_valid <= '0';
    window_valid_1 <= '0';
    window_valid_2 <= '0';
    total_sum := 0;
    for row in 0 to TEMPLATE_SIZE - 1 loop
        template(row) <= (others => '1');
    end loop;
else

    if (pixel_valid = '1') then
        programming_row := TEMPLATE_SIZE - 1 - pixel_row;
        template(pixel_col)(programming_row*PIXEL_WIDTH +
        → PIXEL_WIDTH - 1 downto programming_row*PIXEL_WIDTH)
        → <= pixel_in;
    end if;

    for stage in 0 to 2 loop
        case stage is

            when 0 => -- Effettuo la differenza in valore
            → assoluto tra i pixel corrispondenti del frammento
            → di finestra

                window_valid_1 <= window_valid;
                window_seq_1 <= window_seq;

                if (window_valid = '1') then

                    for col in 0 to TEMPLATE_SIZE /
                    → COMPUTATION_STEPS - 1 loop
                        for row in 0 to TEMPLATE_SIZE - 1 loop

                            window_pixel :=
                                → to_integer(unsigned(window_in(col)(row*PIXEL_WIDTH
                                → + PIXEL_WIDTH - 1 downto
                                → row*PIXEL_WIDTH)));

```

```

template_pixel :=
    to_integer(unsigned(template(col +
    window_seq * TEMPLATE_SIZE /
    COMPUTATION_STEPS)(row*PIXEL_WIDTH +
    PIXEL_WIDTH - 1 downto
    row*PIXEL_WIDTH)));

temp_sad(col)(row*PIXEL_WIDTH + PIXEL_WIDTH
    - 1 downto row*PIXEL_WIDTH) <=
    std_logic_vector(to_unsigned(abs(window_pixel
    - template_pixel), PIXEL_WIDTH));

end loop; -- row loop
end loop; -- col loop

end if;

when 1 => -- Sommo le differenze calcolate nel
    precedente clock e ottengo la somma per il
    frammento di finestra

window_valid_2 <= window_valid_1;
window_seq_2 <= window_seq_1;

if (window_valid_1 = '1') then

    partial_sum := 0;
    for col in 0 to TEMPLATE_SIZE /
    COMPUTATION_STEPS - 1 loop
        for row in 0 to TEMPLATE_SIZE - 1 loop

            partial_sum := partial_sum +
                to_integer(unsigned(temp_sad(col)(row*PIXEL_WIDTH
                + PIXEL_WIDTH - 1 downto
                row*PIXEL_WIDTH)));
        end loop; -- row loop
    end loop; -- col loop
end if;

```

```

        end loop; -- col loop

        partial_sum_2_3 <= partial_sum;

    end if;

when 2 => -- Sommo le somme parziali e ottengo la
    -- somma totale per la finestra
    score_valid <= '0';
    if (window_valid_2 = '1') then

        total_sum := total_sum + partial_sum_2_3;

        if (window_seq_2 = COMPUTATION_STEPS - 1) then

            score <=
                -- std_logic_vector(to_unsigned(total_sum,
                -- SCORE_WIDTH));
            score_valid <= '1';
            total_sum := 0;

        end if;

    end if;

when others => null;

end case; --stage
end loop; --stage

end if; --reset
end if; --clock
end process;
end architecture;

```

Per ottenere un'alta indipendenza dal numero effettivo di moduli SAD che si vogliono predisporre, è stato creato il modulo *template_finder* che racchiude al suo interno la logica di generazione dei moduli SAD. E' possibile infatti, attraverso la direttiva *generate* di vhdl, specificare a tempo di compilazione il numero di moduli SAD che si voglio istanziare. E' stato inoltre predisposto un decoder con enable (predisposto da Altera), anch'esso parametrizzabile, che, ricevuto in ingresso l'indice del template che si vuole programmare, va ad abilitare il segnale *pixel_valid* dello specifico modulo SAD. Il segnale di enable è invece pilotato dal segnale omonimo *pixel_valid* in ingresso alla pipeline di template matching. E' riportata qui sotto la struttura per la sola parte di programmazione dei template.

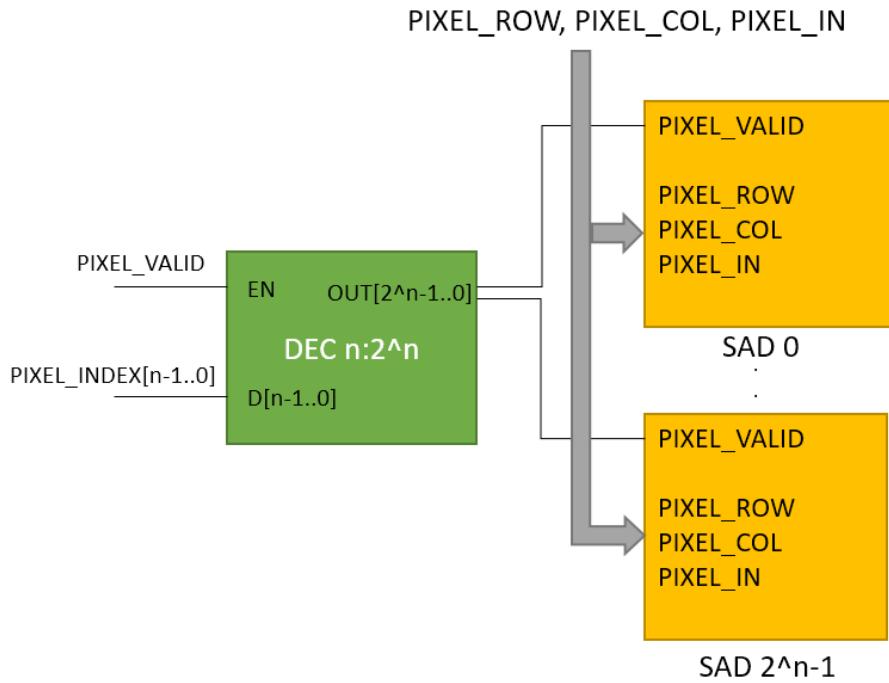


Figure 24: Programmazione parametrizzabile dei moduli SAD

Si riporta infine il codice del template_finder. Si noti che in questo modulo è gestita anche la concatenazione dei punteggi e la generazione di un unico score_valid di uscita, che è l'and degli score_valid dei moduli SAD (in realtà, lavorando i moduli in parallelo, si sarebbe potuto anche utilizzare il

segnalet generato da uno soltanto dei moduli SAD).

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.global.all;
use work.convolution_package.all;

entity template_finder is

port
(
    -- Input ports
    window_valid      : in std_logic;
    window_seq        : in natural range 0 to COMPUTATION_STEPS
    --> - 1;
    window_in         : in window_packet;

    -- Input programmazione template
    pixel_valid       : in std_logic;
    template_index    : in
        std_logic_vector(TEMPLATE_MATCHING_MODULES_WIDTH-1
        --> downto 0);
    pixel_in          : in pixel;
    pixel_row         : in natural range 0 to TEMPLATE_SIZE - 1;
    pixel_col         : in natural range 0 to TEMPLATE_SIZE - 1;

    -- Output ports
    score_valid       : out std_logic;
    scores            : out template_matching_score_array;

    clk : in std_logic;
    reset : in std_logic
);
end template_finder;

architecture template_finder_arch of template_finder is
```

```

component decoder_param IS
  GENERIC
  (
    INPUT_WIDTH    : natural := 2;
    OUTPUT_DECODES : natural := 4
  );
  PORT
  (
    data      : IN STD_LOGIC_VECTOR (INPUT_WIDTH - 1 DOWNTO 0);
    enable   : IN STD_LOGIC;
    selected : OUT STD_LOGIC_VECTOR(OUTPUT_DECODES - 1 DOWNTO
      → 0)
  );
END component;

component SAD
  port
  (
    -- Input ports
    window_valid      : in std_logic;
    window_seq        : in natural range 0 to
      → COMPUTATION_STEPS - 1;
    window_in         : in window_packet;

    -- Input programmazione template
    pixel_valid       : in std_logic;
    pixel_in          : in pixel;
    pixel_row         : in natural range 0 to TEMPLATE_SIZE - 1;
    pixel_col         : in natural range 0 to TEMPLATE_SIZE - 1;

    -- Output ports
    score_valid       : out std_logic;
    score             : out score_type_vector;
    clk               : in std_logic;
    reset             : in std_logic
  );

```

```

end component;

signal pixel_valids :
  std_logic_vector(TEMPLATE_MATCHING_MODULES - 1 downto 0);
signal score_valids :
  std_logic_vector(TEMPLATE_MATCHING_MODULES - 1 downto 0);

signal scores_inner : template_matching_score_array;

constant all_ones :
  std_logic_vector(TEMPLATE_MATCHING_MODULES - 1 downto 0)
  := (others => '1');

begin

  score_valid <= '1' when score_valids = all_ones
    else '0';

  scores <= scores_inner;

  decoder_map: decoder_param
  GENERIC MAP
  (
    INPUT_WIDTH    => TEMPLATE_MATCHING_MODULES_WIDTH,
    OUTPUT_DECODES => TEMPLATE_MATCHING_MODULES
  )
  PORT MAP
  (
    data      => template_index,
    enable    => pixel_valid,
    selected  => pixel_valids
  );

  gen_sad: for i in 0 to TEMPLATE_MATCHING_MODULES - 1 generate
    sadx: SAD port map
    (
      window_valid    => window_valid,
      window_seq      => window_seq,

```

```

    window_in      => window_in,
    pixel_in       => pixel_in,
    pixel_row      => pixel_row,
    pixel_col      => pixel_col,
    pixel_valid    => pixel_valids(i),
    score_valid    => score_valids(i),
    score          => scores_inner(i),
    clk            => clk,
    reset          => reset
  );
end generate gen_sad;

end template_finder_arch;

```

7.3 Interpretazione dei punteggi: template_aggregator

Il modulo finale della pipeline di template matching si occupa di interpretare i punteggi inviati dai vari moduli SAD in parallelo e di generare risultati significativi in uscita. Nel nostro caso, come già esposto, il compito di questo modulo, denominato *template_aggregator* è quello di trovare il punteggio minimo all'interno dell'intera immagine. Se questo valore è inferiore ad un valore di threshold, parzialmente settabile dall'esterno, allora il template è stato trovato.

Entrando maggiormente nel dettaglio, il template_aggregator riceve un array di punteggi in ingresso (la dimensione dell'array è pari al numero di moduli SAD disposti in parallelo), calcola il minimo dell'array (e l'indice del minimo) e verifica che tale valore sia inferiore al minimo corrente; in caso affermativo allora aggiorna il valore di riga e colonna e il valore del minimo corrente con il minimo trovato. Queste operazioni vengono ripetute finché non si giunge al termine dell'immagine, dove il minimo corrente viene confrontato con il valore di threshold: se il minimo è inferiore al threshold viene settato il segnale di found e allo stesso tempo viene aggiornata l'uscita contenente riga e colonna del punto in alto a sinistra della finestra che ha generato il match; inoltre viene mandato in uscita anche l'indice del modulo che ha

generato il punteggio minimo. Se il minimo corrente è superiore al valore di threshold il segnale di found viene messo a 0. In ogni caso al termine dell'immagine viene settato per un intervallo di clock il segnale di finished che indica che la computazione per quel frame è terminata e che sono disponibili nuovi risultati in uscita.

All'interno di questo modulo viene gestita anche la situazione ai bordi dell'immagine: quando la finestra si trova nelle prime TEMPLATE_SIZE - 1 righe o colonne significa che la finestra non è stata ancora interamente riempita. In questo caso il template_aggregator non aggiorna il valore del minimo corrente. Pertanto sarà possibile trovare un template solo dove la finestra è completamente valida.

Viene infine gestita anche la condizione di mascheramento di un template: il punteggio in ingresso viene considerato solo quando il relativo bit di masked è uguale a 0.

Si riporta infine il codice del template_aggregator:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.global.all;
use work.convolution_package.all;

-- TEMPLATE AGGREGATOR STATICO --
-- Verifico se lo score è inferiore a un certo valore
-- costante di threshold

entity template_aggregator is
port
(
    score_valid          : in std_logic;
    scores                : in template_matching_score_array;
    masked                : in
        std_logic_vector(TEMPLATE_MATCHING_MODULES - 1 downto
        0);

```

```

threshold_multiplier      : in std_logic_vector(5 downto
→  0);

template_matching_finished : out std_logic;  -- a 1 per un
→  solo clock
template_matching_found   : out std_logic;
template_matching_winner  : out natural range 0 to
→  TEMPLATE_MATCHING_MODULES - 1;
template_matching_row     : out natural range 0 to
→  IMG_HEIGHT - 1;
template_matching_col     : out natural range 0 to IMG_WIDTH
→  - 1;

clk : in std_logic;
reset : in std_logic
);
end template_aggregator;

architecture template_aggregator_arch of template_aggregator is
begin
process(clk)
-- inner state: template matching
variable current_winner      : natural range 0 to
→  TEMPLATE_MATCHING_MODULES - 1;
variable current_winner_col   : natural range 0 to
→  IMG_WIDTH - 1 := 0;
variable current_winner_row   : natural range 0 to
→  IMG_HEIGHT - 1 := 0;
variable current_winner_score : score_type := SCORE_MAX -
→  1;
variable temp_score          : score_type;
variable temp_winner         : natural range 0 to
→  TEMPLATE_MATCHING_MODULES - 1;

-- inner state: row and col
variable col_count : natural range 0 to IMG_WIDTH - 1 := 0;

```

```

variable row_count : natural range 0 to IMG_HEIGHT - 1 :=
→ 0;

variable threshold : natural;

begin
if (rising_edge(clk)) then

if (reset = '1') then
-- reset variables
current_winner := 0;
current_winner_col := 0;
current_winner_row := 0;
current_winner_score := SCORE_MAX - 1;
col_count := 0;
row_count := 0;
-- reset outputs
template_matching_finished <= '0';
template_matching_found <= '0';
template_matching_row <= 0;
template_matching_col <= 0;
template_matching_winner <= 0;
else
template_matching_finished <= '0';

if (score_valid = '1') then
temp_score := SCORE_MAX - 1;
temp_winner := 0;
for i in 0 to TEMPLATE_MATCHING_MODULES - 1 loop
if (masked(i) = '0' and
→ to_integer(unsigned(scores(i))) < temp_score)
→ then
temp_score := to_integer(unsigned(scores(i)));
temp_winner := i;
end if;
end loop;

```

```

if (row_count >= TEMPLATE_SIZE and col_count >=
→ TEMPLATE_SIZE and temp_score <
→ current_winner_score) then
    current_winner := temp_winner;
    current_winner_score := temp_score;
    current_winner_row := row_count;
    current_winner_col := col_count;
end if;

if (col_count = IMG_WIDTH - 1 and row_count =
→ IMG_HEIGHT - 1) then -- end of image

    template_matching_finished <= '1';
    threshold :=
        → to_integer(unsigned(threshold_multiplier)) *
        → TEMPLATE_SIZE * TEMPLATE_SIZE;

    if (current_winner_score < threshold) then
        template_matching_found <= '1';
        template_matching_row <= current_winner_row -
            → TEMPLATE_SIZE;
        template_matching_col <= current_winner_col -
            → TEMPLATE_SIZE;
        template_matching_winner <= current_winner;
    else
        template_matching_found <= '0';
    end if;

    --reset computation
    current_winner_score := SCORE_MAX - 1;
end if;

col_count := (col_count + 1) mod IMG_WIDTH;
-- update current state (row and col)
if (col_count = 0) then
    row_count := (row_count + 1) mod IMG_HEIGHT;
end if;

```

```
    end if; -- score_valid

    end if; -- reset
  end if; -- clock
end process;

end architecture;
```

7.4 Aggregazione dei moduli di template matching

I moduli creati sono stati collegati opportunamente generando il macro blocco *template_matching_pipeline*. Di seguito si riporta la sua struttura a blocchi e il codice vhdl.

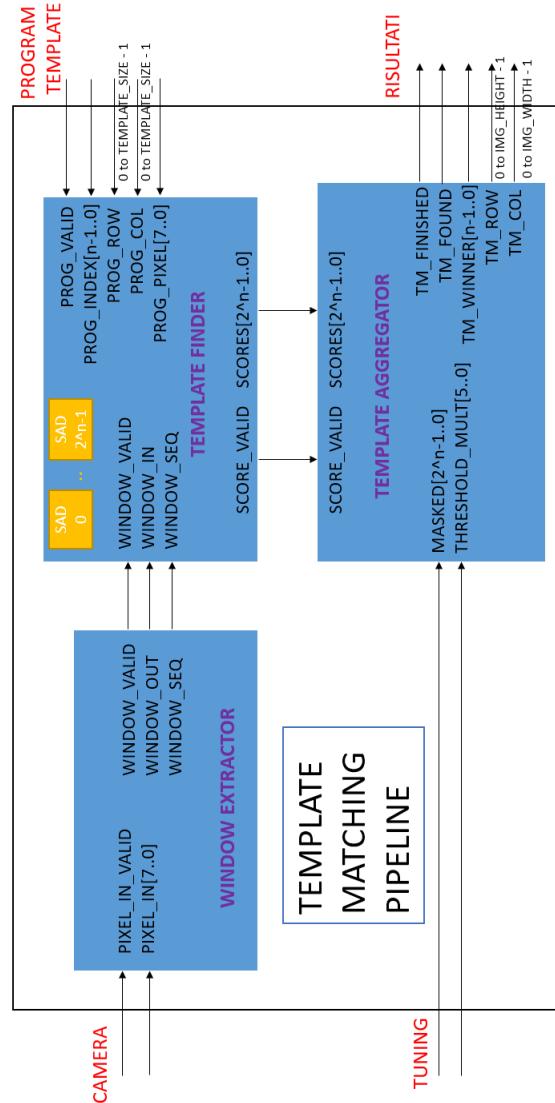


Figure 25: Struttura a blocchi della pipeline di template matching

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.global.all;
use work.convolution_package.all;

entity template_matching_pipeline is
    port
    (
        -- Input ports programming
        prog_valid          : in std_logic;
        prog_index          : in
            -- std_logic_vector(TEMPLATE_MATCHING_MODULES_WIDTH-1
            -- downto 0);
        prog_row             : in natural range 0 to TEMPLATE_SIZE
            -- - 1;
        prog_col             : in natural range 0 to TEMPLATE_SIZE
            -- - 1;
        prog_pixel           : in pixel;

        -- Input ports camera
        camera_in_valid     : in std_logic;
        camera_in_pixel      : in pixel;

        -- Output ports template matching
        template_matching_finished : out std_logic;  -- a 1 per un
            -- solo clock
        template_matching_found   : out std_logic;
        template_matching_winner  : out natural range 0 to
            -- TEMPLATE_MATCHING_MODULES - 1;
        template_matching_row     : out natural range 0 to
            -- IMG_HEIGHT - 1;
        template_matching_col     : out natural range 0 to IMG_WIDTH
            -- - 1;

        -- clock, reset
        clk    : in std_logic;
        reset : in std_logic;

```

```

threshold_multiplier      : in std_logic_vector(5 downto
→ 0);
masked                  : in
→ std_logic_vector(TEMPLATE_MATCHING_MODULES - 1 downto
→ 0)
);
end template_matching_pipeline;

architecture tm_pipeline_arch of template_matching_pipeline is

component window_extractor is
port
(
    pixel_in_valid : in std_logic;
    pixel_in : in pixel;

    window_valid : out std_logic;
    window_seq : out natural range 0 to COMPUTATION_STEPS - 1;
    window_out : out window_packet;

    clk : in std_logic;
    reset : in std_logic
);

end component;

component template_finder is
port
(
    -- Input ports
    window_valid      : in std_logic;
    window_seq        : in natural range 0 to COMPUTATION_STEPS
    → - 1;
    window_in         : in window_packet;

    -- Input programmazione template

```

```

pixel_valid      : in std_logic;
template_index   : in
  ↳ std_logic_vector(TEMPLATE_MATCHING_MODULES_WIDTH-1
  ↳ downto 0);
pixel_in         : in pixel;
pixel_row        : in natural range 0 to TEMPLATE_SIZE - 1;
pixel_col        : in natural range 0 to TEMPLATE_SIZE - 1;

-- Output ports
score_valid      : out std_logic;
scores           : out template_matching_score_array;

clk : in std_logic;
reset : in std_logic
);
end component;

component template_aggregator is
port
(
  score_valid      : in std_logic;
  scores           : in template_matching_score_array;
  masked           : in
    ↳ std_logic_vector(TEMPLATE_MATCHING_MODULES - 1 downto
    ↳ 0);
  threshold_multiplier : in std_logic_vector(5 downto
    ↳ 0);

  template_matching_finished : out std_logic; -- a 1 per
  ↳ un solo clock
  template_matching_found    : out std_logic;
  template_matching_winner   : out natural range 0 to
  ↳ TEMPLATE_MATCHING_MODULES - 1;
  template_matching_row     : out natural range 0 to
  ↳ IMG_HEIGHT - 1;
  template_matching_col     : out natural range 0 to
  ↳ IMG_WIDTH - 1;

```

```

    clk : in std_logic;
    reset : in std_logic
);
end component;

-- signals from window_extractor to template_finder
signal window_valid_sig : std_logic;
signal window_seq_sig : natural range 0 to COMPUTATION_STEPS
    - 1;
signal window_out_sig : window_packet;

-- signals from template_finder to template_aggregator
signal score_valid_sig : std_logic;
signal scores_sig : template_matching_score_array;

begin

window_extractor_port_map: window_extractor
port map
(
    pixel_in_valid => camera_in_valid,
    pixel_in       => camera_in_pixel,

    window_valid   => window_valid_sig,
    window_seq     => window_seq_sig,
    window_out     => window_out_sig,

    clk           => clk,
    reset         => reset
);

template_finder_port_map: template_finder
port map
(
    window_valid   => window_valid_sig,
    window_seq     => window_seq_sig,
    window_in      => window_out_sig,

```

```

pixel_valid      => prog_valid,
template_index   => prog_index,
pixel_in         => prog_pixel,
pixel_row        => prog_row,
pixel_col        => prog_col,

score_valid      => score_valid_sig,
scores           => scores_sig,

clk              => clk,
reset            => reset
);

template_aggregator_port_map: template_aggregator
port map
(
    score_valid      => score_valid_sig,
    scores           => scores_sig,
    threshold_multiplier   => threshold_multiplier,
    masked           => masked,

    template_matching_finished  => template_matching_finished,
    template_matching_found    => template_matching_found,
    template_matching_winner   => template_matching_winner,
    template_matching_row      => template_matching_row,
    template_matching_col      => template_matching_col,

    clk              => clk,
    reset            => reset
);

end tm_pipeline_arch;

```

8 Visualizzazione dei risultati

I segnali `template_matching_found` e `template_matching_index` in uscita dalla pipeline di template matching sono stati collegati a dei led sulla scheda. Si analizza ora in maggiore dettaglio la parte di visualizzazione dei risultati su VGA e sui display a 7 segmenti.

8.1 VGA: square generator

Per visualizzare a video l'area di riconoscimento del template è stato creato il modulo *square_generator*. Questo modulo è posto a monte del vga controller e a valle della fifo di passaggio dal dominio a 100 MHz al dominio a 25 MHz. Il suo compito è ricevere i pixel dalla fifo e i risultati dalla pipeline di template matching, mantenere uno stato interno in termini di riga e colonna correnti e mandare al vga controller i corretti colori R, G e B. In particolare:

- Se il template è trovato e la riga e colonna correnti fanno parte del bordo del template, l'uscita RGB corrisponde al colore viola.
- In caso contrario si ha $R = G = B = \text{pixel grayscale}$

L'aggiornamento dei valori di riga e colonna viene fatto nel momento in cui il vga controller invia la richiesta di lettura, richiesta che viene prontamente retroazionata alla fifo. Si noti infine che i valori `DRAW`, `DRAW_ROW` e `DRAW_COL`, che segnalano al modulo di disegnare e dove disegnare, vengono campionati al primo pixel dell'immagine. In questo modo si evitano a priori problemi dovuti ai diversi tempi di elaborazione della vga e della pipeline di template matching (si potrebbe verificare il caso di un quadrato disegnato a metà perché durante il disegno del frame a VGA la pipeline ha modificato i dati).

Il segnale di `DRAW` è collegato al segnale `template_matching_found` della parte di riconoscimento dei template, e i segnali di riga e colonna ai corrispondenti segnali di uscita della medesima pipeline.

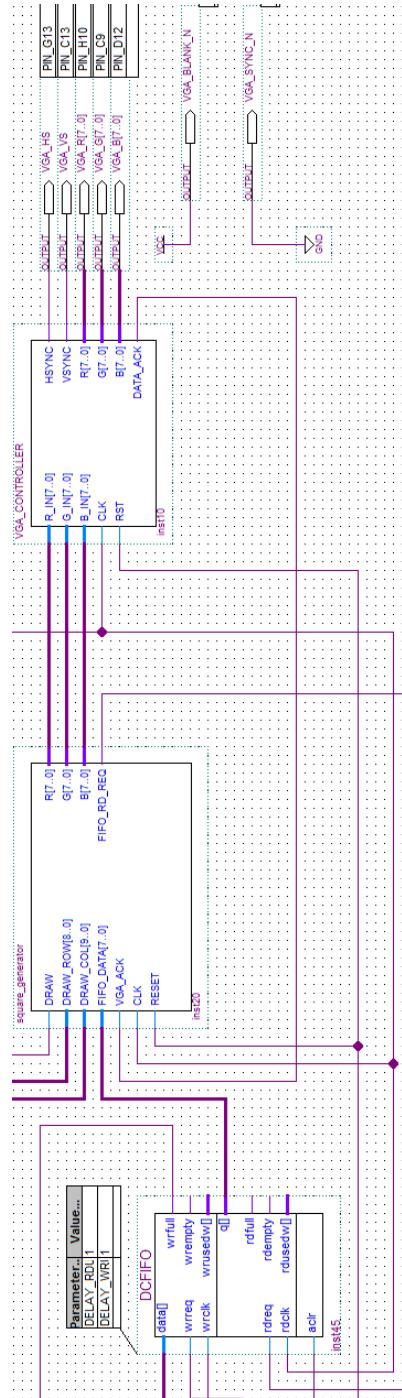


Figure 26: Schematico dello square generator

Il codice vhdl dello square_generator è il seguente:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.global.all;
use work.convolution_package.all;

entity square_generator is

port
(
    -- input
    DRAW : IN STD_LOGIC;
    DRAW_ROW : IN NATURAL range 0 TO IMG_HEIGHT - 1;
    DRAW_COL : IN NATURAL range 0 TO IMG_WIDTH - 1;

    FIFO_DATA : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    VGA_ACK : IN STD_LOGIC;

    --output
    R: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    G: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    B: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    FIFO_RD_REQ: OUT STD_LOGIC;

    --control
    CLK : IN STD_LOGIC;
    RESET : IN STD_LOGIC

);

end square_generator;

architecture square_generator_arch of square_generator is
begin
```

```

process (clk, reset)

    variable row : natural range 0 to IMG_HEIGHT - 1 := 0;
    variable col : natural range 0 to IMG_WIDTH - 1 := 0;

    variable DRAW_inner : std_logic;
    variable DRAW_ROW_inner : natural range 0 to IMG_HEIGHT -
        < 1;
    variable DRAW_COL_inner : natural range 0 to IMG_WIDTH - 1;

begin
    if (rising_edge(clk)) then
        if (reset = '1') then
            -- sync reset
            row := 0;
            col := 0;
            DRAW_ROW_inner := 0;
            DRAW_COL_inner := 0;
            DRAW_inner := '0';
            R <= (others=>'0');
            G <= (others=>'1');
            B <= (others=>'0');
        else
            if (row = 0 and col = 0) -- sono in (0,0) --> campiono
                -- riga e colonna
            then
                DRAW_inner := DRAW;
                DRAW_ROW_inner := DRAW_ROW;
                DRAW_COL_inner := DRAW_COL;
            end if;

            --calcolo del dato in uscita
            if( DRAW_inner = '1' AND
            (
                -- Condizione per i due bordi orizzontali

```

```

((row = DRAW_ROW_inner OR row = DRAW_ROW_inner +
    ↵ TEMPLATE_SIZE - 1) -- Verifico o riga sopra o
    ↵ riga sotto
AND col >= DRAW_COL_inner AND col <=
    ↵ DRAW_COL_inner + TEMPLATE_SIZE - 1)

OR

((col = DRAW_COL_inner OR col = DRAW_COL_inner +
    ↵ TEMPLATE_SIZE - 1) -- Verifico o colonna
    ↵ sinistra o colonna destra
AND row >= DRAW_ROW_inner AND row <=
    ↵ DRAW_ROW_inner + TEMPLATE_SIZE - 1)
)
)
then --viola
    R <= (others=>'1');
    G <= (others=>'0');
    B <= (others=>'1');
else --pixel corrente
    R <= FIFO_DATA;
    G <= FIFO_DATA;
    B <= FIFO_DATA;
end if;

--aggiornamento indici di riga e colonna
if(VGA_ACK = '1')
then
    FIFO_RD_REQ <= '1';
    if (col = IMG_WIDTH - 1) then
        row := (row + 1) mod IMG_HEIGHT;
    end if;
    col := (col + 1) mod IMG_WIDTH;
else
    FIFO_RD_REQ <= '0';
end if;

end if;

```

```
    end if;  
end process;  
  
end architecture;
```

8.2 Display a 7 segmenti

Sono stati utilizzati 4 display a 7 segmenti per visualizzare il numero di template trovati. Questo blocco di elaborazione è stato suddiviso in 3 parti:

- occurrence_counter: si occupa di interpretare i risultati prodotti dalla pipeline di template matching per incrementare il contatore delle occorrenze. Nel nostro caso il contatore viene incrementato quando il segnale di found va a 1 mentre nel frame precedente era a 0 (segnaliamo quindi l'ingresso di un oggetto nell'immagine). Si potrebbero costruire contatori con una logica arbitrariamente complessa: per esempio si potrebbe anche analizzare le coordinate di riga e colonna ed incrementare il contatore se le nuove coordinate sono molto distanti da quelle precedenti, oppure si potrebbe concedere che la pipeline per qualche frame non riconosca bene il template e quindi dichiarare il template come "fuori dall'immagine" solo dopo aver ricevuto un numero prefissato di valori found = 0.
- digit_extractor: è una rete combinatoria che si occupa di ricavare dal contatore le 4 cifre decimali meno significative (unità, decine, centinaia e migliaia). Attualmente utilizza divisioni e moduli in base 10, 100 e 1000, pertanto è altamente non ottimizzato (utilizza molte risorse).
- binary_digit_to_7_segments: è la rete combinatoria che si occupa di trasformare una cifra decimale nella sua rappresentazione a 7 segmenti. Si noti che i display a 7 segmenti presenti sulla scheda richiedono una rappresentazione in logica negata (0 significa acceso).

Si riporta infine il codice di questi 3 moduli e la loro connessione al resto dell'architettura:

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;

entity occurrence_counter is
generic
(
    COUNTER_WIDTH : natural := 10;
    COUNTER_BASE : natural := 1000
);
port
(
    template_matching_finished : in std_logic;
    template_matching_found     : in std_logic;

    count                      : out std_logic_vector(COUNTER_WIDTH - 1
                                                & downto 0);

    clk                         : in std_logic;
    reset                       : in std_logic
);
end occurrence_counter;

architecture occurrence_counter_arch of occurrence_counter is

begin

process(clk)
    variable counter      : natural range 0 to COUNTER_BASE - 1
    &:= 0;
    variable old_found    : std_logic;
begin
    if (rising_edge(clk)) then
        if (reset = '1') then
            counter := 0;
        else
            if (template_matching_finished = '1') then

```

```

    if (old_found = '0' and template_matching_found =
        '1') then
        counter := (counter + 1) mod COUNTER_BASE;
    end if;

    old_found := template_matching_found;
    end if;
end if;

count <= std_logic_vector(to_unsigned(counter,
    COUNTER_WIDTH));

end if;
end process;

end architecture;

```

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;

entity digit_extractor is
generic
(
    COUNTER_WIDTH : natural := 10;
    COUNTER_BASE : natural := 1000
);
port
(
    count      : in std_logic_vector(COUNTER_WIDTH - 1 downto 0);

    digit3     : out natural range 0 to 9;
    digit2     : out natural range 0 to 9;
    digit1     : out natural range 0 to 9;
    digit0     : out natural range 0 to 9
);
end digit_extractor;

architecture digit_extractor_arch of digit_extractor is

signal count_int : integer;

begin

    count_int    <= to_integer(unsigned(count));
    digit0      <= count_int mod 10;           -- unita'
    digit1      <= (count_int mod 100) / 10;    -- decine
    digit2      <= (count_int mod 1000) / 100;   -- centinaia
    digit3      <= (count_int mod 10000) / 1000;  -- migliaia

end architecture;

```

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity binary_digit_to_7_segments is
port
(
    digit      : in natural range 0 to 9;
    seg        : out std_logic_vector(6 downto 0)
);
end binary_digit_to_7_segments;

architecture binary_digit_to_7_segments_arch of
→ binary_digit_to_7_segments is

begin

    with digit select
    seg <=    "1000000"    when 0,
                "1111001"    when 1,
                "0100100"    when 2,
                "0110000"    when 3,
                "0011001"    when 4,
                "0010010"    when 5,
                "0000011"    when 6,
                "1111000"    when 7,
                "0000000"    when 8,
                "0011000"    when 9,
                "0000110"    when others;

end architecture;

```

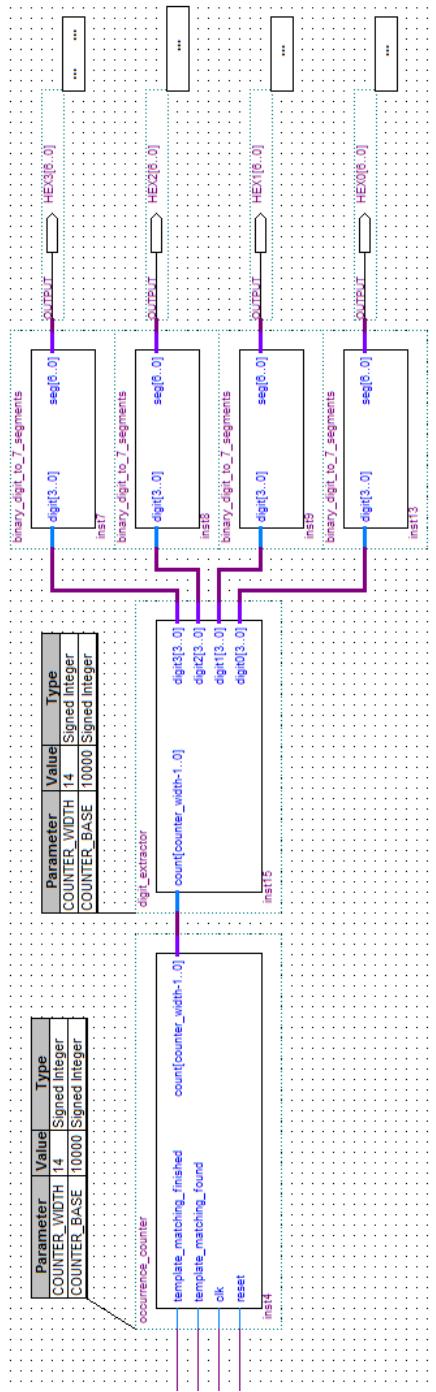


Figure 27: Schematico del collegamento alla pipeline della logica di conteggio su display a 7 segmenti

9 Interfacciamento seriale

Per realizzare la comunicazione con un calcolatore esterno alla scheda è stata utilizzata la porta seriale. Sono stati realizzati in particolare un modulo per la ricezione di dati, utilizzato per programmare i template, ed un modulo per l'invio di dati dalla FPGA, che non è stato poi utilizzato nel progetto. In particolare i moduli realizzati seguono la linea di un *Universal Asynchronous Receiver Transmitter* o UART. Il trasmettitore prende un byte alla volta e lo trasmette in maniera sequenziale; in modo duale il ricevitore raccoglie 8 bit e li porta in uscita. La linea sulla quale vengono trasmessi i bit è di norma a valore logico alto, per questo motivo ogni byte è preceduto da un bit a '0' che indica l'inizio di una trasmissione. Dopo l'invio degli 8 bit che compongono il byte inviato si deve prontamente tornare allo stato logico di partenza. Nel progetto la velocità di trasmissione dei dati attraverso la porta seriale è di 9600b/s mentre il clock dei moduli realizzati è 50MHz. E' possibile notare che un bit sarà valido per un certo numero di clock della FPGA.

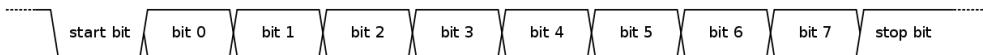


Figure 28: Data framing

9.1 Receiver

Il receiver seriale riceve in ingresso la linea sulla quale sono trasmessi i dati e restituisce in uscita un segnale BUSY, attivo quando è in corso una ricezione, ed il byte ricevuto, campionabile quando BUSY passa da stato logico alto a basso. La costante DATA_DURATION è calcolata dividendo la frequenza di clock per la velocità di trasmissione della seriale (in questo caso $50M/9600$) ed indica il numero di clock tra un bit e il successivo. Il receiver ha quattro stati:

- STATE IDLE: in questo stato si attende che RX_LINE passi allo stato logico '0' segnalando l'inizio della trasmissione di un byte.
- STATE RX_START_BIT: è necessario verificare che lo start bit non sia spurio: per questo motivo in questo stato si campiona RX_LINE a metà di DATA_DURATION in quanto si considera stabile il valore a

questo punto. Se il valore dello start bit viene confermato si azzera il contatore dei clock e si passa al trasferimento vero e proprio.

- STATE RX_DATA: il receiver permane in questo stato fino a che non vengono ricevuti tutti i bit. Il campionamento avviene ogni DATA_DURATION clock ma il precedente azzeramento del contatore a metà conteggio ci permette di campionare ogni bit circa a metà della sua permanenza in input (valore stabile).
- STATE RX_STOP_BIT: si attende la ricezione del bit di fine trasmissione. Successivamente si porta il bit BUSY a valore logico basso e contemporaneamente si trasmette il byte ricevuto in uscita.

Si riporta il codice del ricevitore da seriale rx:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY RX IS
PORT(
    CLK: IN STD_LOGIC;
    RESET: IN STD_LOGIC;
    RX_LINE: IN STD_LOGIC;
    BUSY: OUT STD_LOGIC;
    DATA: OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
);
END RX;

ARCHITECTURE RX_ARCH OF RX IS

CONSTANT DATA_DURATION: INTEGER := 5208;

TYPE type_state IS (IDLE, RX_START_BIT, RX_DATA,
                    RX_STOP_BIT);
signal STATE : type_state := IDLE;

signal DATABUFF: STD_LOGIC_VECTOR (7 DOWNTO 0);
```

```

BEGIN
PROCESS(CLK)
    variable CLOCK_COUNT: integer RANGE 0 TO
        DATA_DURATION-1:=0;
    variable INDEX: integer RANGE 0 TO 9:=0;
BEGIN
    IF(RISING_EDGE(CLK))
THEN
    IF (RESET = '1')
THEN
    STATE <= IDLE;
    INDEX := 0;
    CLOCK_COUNT := 0;
    BUSY <= '0';
ELSE
    case STATE is
when IDLE =>
    INDEX := 0;
    CLOCK_COUNT := 0;
    --start bit found
    if(RX_LINE = '0')
    then
        BUSY <= '1';
        STATE <= RX_START_BIT;
    else
        BUSY <= '0';
        STATE <= IDLE;
    end if;

when RX_START_BIT =>
-- controllo a metà della durata del bit di start che
-- sia ancora basso per evitare bit spuri
if(CLOCK_COUNT = (DATA_DURATION-1)/2) then
    if(RX_LINE = '0') then

```

```

CLOCK_COUNT := 0; --ho trovato la metà, da questo
                  -- momento campioniamo il dato circa a metà del
                  -- periodo in cui è disponibile
STATE <= RX_DATA;

else
    --se era uno start bit spurio torno a idle
    STATE <= IDLE;
end if;
else
    CLOCK_COUNT := CLOCK_COUNT + 1;
    STATE <= RX_START_BIT;
end if;

when RX_DATA =>
    if(CLOCK_COUNT < DATA_DURATION-1) then
        CLOCK_COUNT := CLOCK_COUNT +1;
        STATE <= RX_DATA;
    else
        --salvo il bit e azzero il contatore
        DATABUFF(INDEX) <= RX_LINE;
        CLOCK_COUNT := 0;

        --controllo e aggiornamento indice
        if(INDEX < 7) then
            INDEX := INDEX +1;
            STATE <= RX_DATA;
        else
            INDEX := 0;
            STATE <=RX_STOP_BIT;
        end if;
    end if;

when RX_STOP_BIT =>
    if(CLOCK_COUNT < DATA_DURATION-1) then
        CLOCK_COUNT := CLOCK_COUNT +1;
        STATE <= RX_STOP_BIT;
    else
        BUSY <= '0';

```

```
    DATA <= DATABUFF;
    STATE <= IDLE;
end if;

end case;

END IF;
END IF;
END PROCESS;
END RX_ARCH;
```

10 Programmazione template

10.1 Programmazione da seriale

Per permettere la programmazione dei template da un host esterno è stato ideato un protocollo ad hoc; il protocollo, molto semplice da realizzare, prevede che l'host invii sul primo byte l'indice del template che vuole programmare (quindi il protocollo garantisce la possibilità di programmare fino a 256 template differenti) e a susseguirsi tutti i pixel del template su ogni byte, inviati per righe. Il modulo così realizzato, denominato *template_injector*, prevede quindi di interfacciarsi direttamente con il modulo RX di gestione della seriale e di campionare sui fronti di discesa del segnale BUSY l'indice del template e poi tutti i pixel. Lo stato di riga e colonna corrente è quindi mantenuto dalla FPGA.

Si riporta di seguito il codice vhdl del template_injector:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.global.all;
use work.convolution_package.all;

entity template_injector is
    port
    (
        -- Input ports
        UART_RXD      : IN STD_LOGIC;  -- seriale

        -- Output ports
        prog_valid     : out std_logic;
        prog_index     : out
            std_logic_vector(TEMPLATE_MATCHING_MODULES_WIDTH - 1
            downto 0);
        prog_row       : out natural range 0 to TEMPLATE_SIZE - 1;
        prog_col       : out natural range 0 to TEMPLATE_SIZE - 1;
        prog_pixel     : out pixel;

        --control
```

```

        CLK      : IN STD_LOGIC;
        RESET   : IN STD_LOGIC
    );
end template_injector;

architecture template_injector_arch of template_injector is

type programming_state is (PROGRAM_INDEX, PROGRAM_PIXEL);

SIGNAL RX_DATA: STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL RX_BUSY: STD_LOGIC := '0';

COMPONENT RX
PORT(
    CLK      : IN STD_LOGIC;
    RESET   : IN STD_LOGIC;
    RX_LINE : IN STD_LOGIC;
    DATA    : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    BUSY    : OUT STD_LOGIC
);
END COMPONENT;

begin

RX_1: RX PORT MAP (CLK, RESET, UART_RXD, RX_DATA, RX_BUSY);

process(clk)

variable row : natural range 0 to TEMPLATE_SIZE - 1 := 0;
variable col : natural range 0 to TEMPLATE_SIZE - 1 := 0;
variable old_rx_busy : std_logic := '0';

-- indice ricevuto al primo byte e utilizzato per
-- → programmare
-- il template nelle successive iterazioni
variable temp_index :
    → std_logic_vector(TEMPLATE_MATCHING_MODULES_WIDTH - 1
    → downto 0) := (others => '0');

```

```

variable state : programming_state;

begin
if (rising_edge(clk)) then

    prog_valid    <= '0';

    if (reset = '1') then

        -- reset inner state
        state := PROGRAM_INDEX;
        row := 0;
        col := 0;

        -- reset outputs
        prog_index    <= (others => '0');
        prog_row     <= 0;
        prog_col     <= 0;
        prog_pixel   <= (others => '0');

    else

        if (old_rx_busy = '1' and RX_BUSY = '0') then -- falling
            -- edge di RX_BUSY

            -- E' stato ricevuto un byte

        case state is
            when PROGRAM_INDEX =>

                -- campiona l'indice del template da programmare
                temp_index :=
                    RX_DATA(TEMPLATE_MATCHING_MODULES_WIDTH - 1
                    downto 0);
                -- transizione di stato
                state := PROGRAM_PIXEL;

            when PROGRAM_PIXEL =>

```

```

        prog_valid    <= '1';
        prog_index    <= temp_index;
        prog_pixel    <= RX_DATA;
        prog_row      <= row;
        prog_col      <= col;

        --Incrementare row e col
        col := (col + 1) mod TEMPLATE_SIZE;
        if (col = 0) then
            row := (row + 1) mod TEMPLATE_SIZE;
        end if;

        -- transizione di stato
        if (row = 0 and col = 0) then
            state := PROGRAM_INDEX;
        end if;

    end case;

end if; -- falling edge rx_busy

old_rx_busy := RX_BUSY;

end if; -- reset
end if; --clk

end process;

end template_injector_arch;

```

10.2 Programmazione da flusso video

Per quanto riguarda la programmazione tramite flusso video, è stato scelto di impostare una posizione fissa, in termini di riga e colonna all'interno dell'immagine, in cui campionare il template. Ovviamente è stato reso possibile visualizzare correttamente tale posizione, in modo da poter "prendere la mira". E' stato quindi creato un nuovo modulo, denominato *quadrato_ centrale*, posto tra la pipeline di template matching e il modulo square generator, il cui comportamento è di fatto quello di un mux: se il segnale *center* (collegato ad uno switch) è a uno il quadrato viene sempre visualizzato e posto in una posizione fissa (centrale nello schermo); se *center* vale 0 i valori della pipeline di template matching sono proposti senza modifica sull'uscita.

Si riporta per completezza lo schema di funzionamento; il codice è riportato sotto:

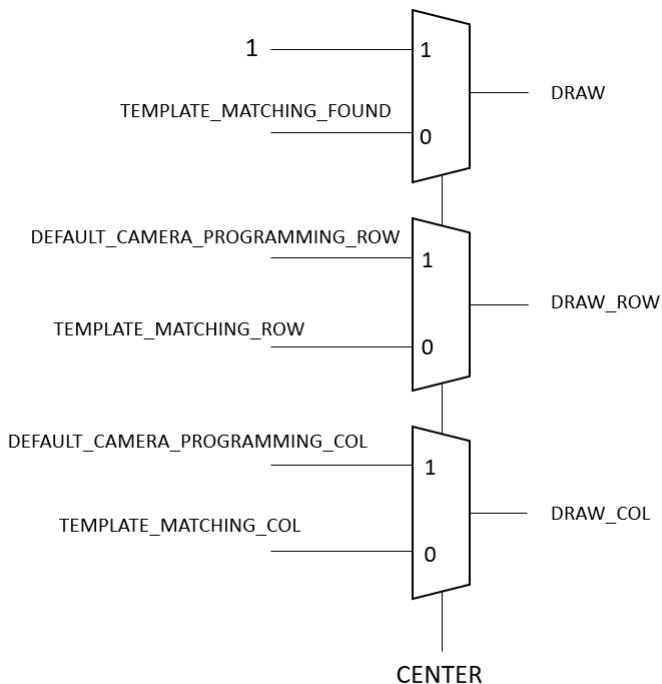


Figure 29: Logica di integrazione della pipeline per disegnare un quadrato in posizione fissa

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.global.all;
use work.convolution_package.all;

entity quadrato_centrale is
port
(
    template_matching_found : in std_logic;
    template_matching_row    : in natural range 0 to IMG_HEIGHT -
        < 1;
    template_matching_col    : in natural range 0 to IMG_WIDTH - 1;

    center                  : in std_logic;

    draw                    : out std_logic;
    draw_row                : out natural range 0 to IMG_HEIGHT - 1;
    draw_col                : out natural range 0 to IMG_WIDTH - 1
);
end quadrato_centrale;

architecture quadrato_centrale_arch of quadrato_centrale is
begin

    draw <= template_matching_found or center;
    draw_row <= template_matching_row when center = '0' else
        < DEFAULT_CAMERA_PROGRAMMING_ROW;
    draw_col <= template_matching_col when center = '0' else
        < DEFAULT_CAMERA_PROGRAMMING_COL;

end architecture;

```

Il modulo che si occupa della vera e propria cattura dei pixel di programmazione è stato denominato *template_camera_programmer*. Ovviamente anche questo modulo deve essere a conoscenza della posizione statica disegnata su VGA al momento della programmazione.

Il modulo realizzato è costantemente in stato di IDLE e rimane in ascolto

del flusso di pixel che passa attraverso la pipeline; anche quando rimane in stato di IDLE incrementa il suo stato interno di riga e colonna in modo da rimanere coerente con il flusso dati. Quando arriva una richiesta di programmazione, pilotata dal segnale *prog_enable*, il modulo transita in uno stato transitorio denominato WAIT_NEW_IMAGE, dove resta finché riga e colonna non ritornano a 0. Questo stato transitorio è necessario per evitare situazioni in cui il segnale di *prog_enable* arriva quando il modulo ha già oltrepassato la "zona di programmazione", o ancora peggio si trova dentro di essa. Questo stato transitorio permette quindi di essere assolutamente certi di campionare correttamente il template. Lo stato WAIT_NEW_IMAGE transita infine nello stato PROG_TEMPLATE, stato che viene mantenuto per un intero frame del flusso dati. Quando le coordinate di riga e colonna correnti si trovano all'interno dell'area richiesta, il pixel corrente viene campionato e inviato alla pipeline di template matching assieme alle coordinate di riga e colonna. Una volta arrivato al termine dell'immagine, si ritorna nello stato IDLE. Per evitare di programmare inutilmente più volte di fila un template in seguito ad un'unica pressione di un bottone, la transizione da IDLE a WAIT_NEW_IMAGE viene effettuata solo sul fronte di salita del segnale *prog_enable* (cioè se *prog_enable* al clock precedente valeva 0).

Si riporta il codice del template_camera_programmer:

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.global.all;
use work.convolution_package.all;

entity template_camera_programmer is
port
(
    pixel_valid : in std_logic;
    pixel_in     : in pixel;

    prog_enable  : in std_logic;

    prog_valid   : out std_logic;
    prog_row     : out natural range 0 to TEMPLATE_SIZE - 1;

```

```

prog_col      : out natural range 0 to TEMPLATE_SIZE - 1;
prog_pixel    : out pixel;

clk          : in std_logic;
reset        : in std_logic
);
end template_camera_programmer;

architecture template_camera_programmer_arch of
→ template_camera_programmer is
type state_type is (IDLE, WAIT_NEW_IMAGE, PROG_TEMPLATE);
begin

process(clk)
variable last_prog_enable : std_logic;
variable state : state_type := IDLE;
variable image_row : natural range 0 to IMG_HEIGHT - 1      := → 0;
variable image_col : natural range 0 to IMG_WIDTH - 1      := → 0;
variable template_row : natural;
variable template_col : natural;
begin
if (rising_edge(clk)) then
if (reset = '1') then

-- reset inner state
state := IDLE;
image_row := 0;
image_col := 0;
last_prog_enable := '1';

-- reset outputs
prog_valid <= '0';
prog_row <= 0;
prog_col <= 0;
prog_pixel <= (others => '0');

else

```

```

prog_valid <= '0';

case state is
  when IDLE =>
    if (prog_enable = '1' and last_prog_enable = '0')
      then
        state := WAIT_NEW_IMAGE;
    end if;

  when WAIT_NEW_IMAGE =>
    if (image_row = 0 and image_col = 0) then
      state := PROG_TEMPLATE;
    end if;

  when PROG_TEMPLATE =>

    if ( pixel_valid = '1') then
      if ( image_row >= DEFAULT_CAMERA_PROGRAMMING_ROW
        and
        image_row < DEFAULT_CAMERA_PROGRAMMING_ROW +
          TEMPLATE_SIZE and
        image_col >= DEFAULT_CAMERA_PROGRAMMING_COL and
        image_col < DEFAULT_CAMERA_PROGRAMMING_COL +
          TEMPLATE_SIZE
      ) then

        template_row := image_row -
          DEFAULT_CAMERA_PROGRAMMING_ROW;
        template_col := image_col -
          DEFAULT_CAMERA_PROGRAMMING_COL;

        prog_valid <= '1';
        prog_row <= template_row;
        prog_col <= template_col;
        prog_pixel <= pixel_in;

    end if;

```

```

        if (image_row = IMG_HEIGHT - 1 and image_col =
           ↵  IMG_WIDTH - 1) then
            state := IDLE;
        end if;

        end if;

    end case;

    if (pixel_valid = '1') then
        -- Update row and col when pixel valid
        image_col := (image_col + 1) mod IMG_WIDTH;
        if (image_col = 0) then
            image_row := (image_row + 1) mod IMG_HEIGHT;
        end if;
    end if;

    last_prog_enable := prog_enable;

    end if; -- reset
    end if; -- clk
end process;
end architecture;

```

Come si può notare rimane da gestire ancora l'indice del template da programmare. Il template_camera_programmer infatti non è a conoscenza di tale valore (il che lo rende più indipendente dal resto dell'architettura).

Nell'architettura realizzata sono presenti 4 moduli SAD in parallelo; la loro programmazione è stata mappata sui 4 bottoni disponibili sulla DE2. Per trasformare il vettore a 4 bit dei bottoni (che arriva in logica negata) nel corrispondente numero binario è stato creato un encoder, denominato *template_index_encoder*, che oltre a operare la trasformazione canonica abilita anche un segnale di output che viene mandato in ingresso al prog_enable del

template_camera_programmer. Si riporta qui il codice:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity template_index_encoder is
port
(
    keys      : in std_logic_vector(3 downto 0);
    index     : out std_logic_vector(1 downto 0);
    enabled   : out std_logic
);
end template_index_encoder;

architecture template_index_encoder_arch of
    template_index_encoder is
begin

    -- KEY[3..0] arrivano in logica negata

    with keys select
    enabled <= '1' when "1110" | "1101" | "1011" | "0111",
                '0' when others;

    with keys select
    index <= "00" when "1110",
            "01" when "1101",
            "10" when "1011",
            "11" when "0111",
            "00" when others;

end architecture;
```

Riassumendo, per programmare i template è sufficiente:

1. Abilitare lo switch di programmazione, che permette di visualizzare un quadrato in posizione statica sullo schermo.
2. Porre il template da riconoscere all'interno del quadrato.
3. Spingere uno dei 4 bottoni per programmare il modulo SAD corrispondente.

10.3 Interfacciamento alla pipeline

Per poter consentire la programmazione sia tramite seriale sia tramite flusso video è stato creato un ulteriore modulo di raccordo, denominato *programmazione_mux*, che funge appunto da mux. Il segnale di controllo dei mux è il segnale di valid della seriale. Si riporta qui il codice:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.global.all;
use work.convolution_package.all;

entity programmazione_mux is
port
(
    serial_prog_valid : in std_logic;
    serial_prog_index : in
        std_logic_vector(TEMPLATE_MATCHING_MODULES_WIDTH - 1
        downto 0);
    serial_prog_row : in natural range 0 to TEMPLATE_SIZE - 1;
    serial_prog_col : in natural range 0 to TEMPLATE_SIZE - 1;
    serial_prog_pixel : in pixel;

    camera_prog_valid : in std_logic;
    camera_prog_index : in
        std_logic_vector(TEMPLATE_MATCHING_MODULES_WIDTH - 1
        downto 0);
    camera_prog_row : in natural range 0 to TEMPLATE_SIZE - 1;
```

```

camera_prog_col : in natural range 0 to TEMPLATE_SIZE - 1;
camera_prog_pixel : in pixel;

prog_valid      : out std_logic;
prog_index      : out
  ↵ std_logic_vector(TEMPLATE_MATCHING_MODULES_WIDTH - 1
  ↵ downto 0);
prog_row        : out natural range 0 to TEMPLATE_SIZE - 1;
prog_col        : out natural range 0 to TEMPLATE_SIZE - 1;
prog_pixel      : out pixel
);
end programmazione_mux;

architecture programmazione_mux_arch of programmazione_mux is
begin

  prog_valid <= serial_prog_valid or camera_prog_valid;

  prog_index <= serial_prog_index when serial_prog_valid = '1'
    else camera_prog_index;

  prog_row <= serial_prog_row when serial_prog_valid = '1'
    else camera_prog_row;

  prog_col <= serial_prog_col when serial_prog_valid = '1'
    else camera_prog_col;

  prog_pixel <= serial_prog_pixel when serial_prog_valid = '1'
    else camera_prog_pixel;

end architecture;

```

Si riporta infine lo schematico completo relativo alla parte di programmazione dei template:

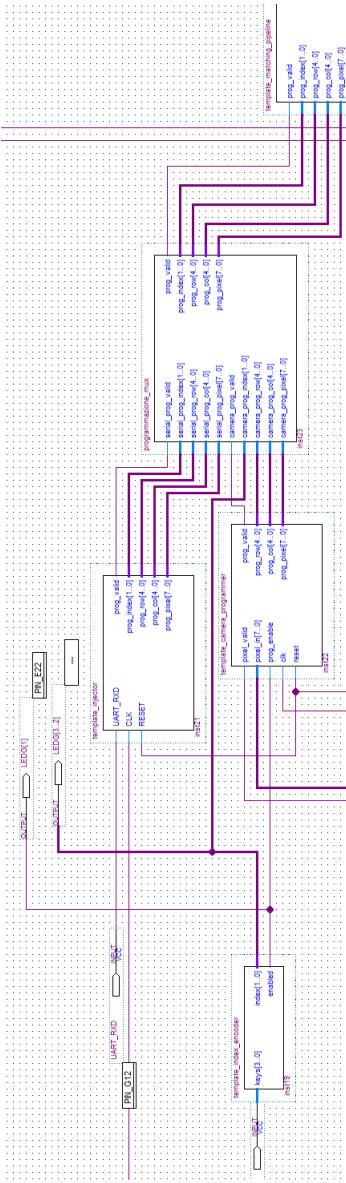


Figure 30: Schematico della logica di programmazione dei template tramite seriale e tramite flusso video

11 Potenziamento della pipeline di template matching: il caso 64x64

Con il riconoscimento di 4 template in parallelo di dimensione 16×16 si raggiunge una percentuale di risorse utilizzate del 20%. Aumentando la dimensione a 32 si raggiunge l'80% delle risorse logiche della FPGA. E' chiaro che aumentare ulteriormente la risoluzione del template porterebbe ad un design non implementabile sulla scheda. E' stato però trovato uno stratagemma che ha consentito di portare la dimensione della finestra di riconoscimento template a 64 pixel pur mantenendo le risorse della pipeline con template di dimensione 32.

La soluzione si basa sul sottocampionare una finestra di dimensione 64×64 soltanto sui pixel con indici di riga e colonna entrambi pari; in questo modo si ottiene una finestra di dimensioni pari a quella di un template di dimensione 32×32 . Si veda l'immagine sotto come esempio.

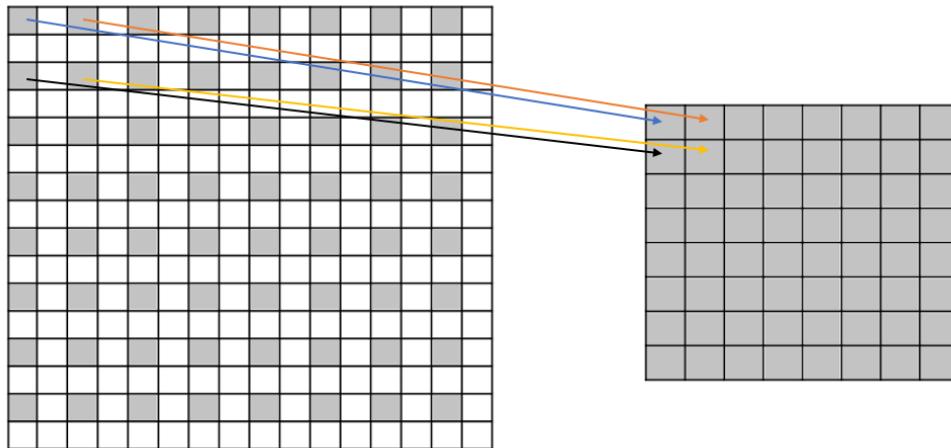


Figure 31: Sottocampionamento di una finestra di dimensione 16×16 per ottenere un template di dimensione 8×8

Per rendere possibile l'implementazione di questa soluzione sono state apportate delle modifiche abbastanza radicali ad alcuni moduli specifici:

- La parte di disegno del quadrato deve lavorare su una dimensione doppia.
- La parte di programmazione del template tramite flusso dati lavora anch'esso su una finestra 64×64 e deve campionare opportunamente i pixel desiderati, cioè quelli per cui $row \bmod 2 = col \bmod 2 = 0$. E' necessario inoltre posizionare correttamente il pixel all'interno del template di lato 32. Per fare ciò è sufficiente divere per 2 la differenza $riga_corrente - riga_statica$ (stesso procedimento per le colonne).
- Il modulo window_extractor deve lavorare anch'esso su una finestra 64×64 e mantenere perciò un line buffer di dimensione 64. La parte più articolata è quella di compattazione della finestra da 64×64 a 32×32 selezionando i pixel opportuni.

La cosa significativa da sottolineare è che tutto il resto della architettura, inclusi i moduli della SAD e il template_aggregator, non sono stati minimamente modificati. Ciò è indice di un'elevata modularità del design realizzato.

Si riporta per completezza il codice del window_extractor aggiornato:

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.global.all;
use work.convolution_package.all;

entity window_extractor is

  port
  (
    pixel_in_valid : in std_logic;
    pixel_in : in pixel;

    window_valid : out std_logic;
    window_seq : out natural range 0 to COMPUTATION_STEPS - 1;
    window_out : out template_packet;

    clk : in std_logic;
  
```

```

        reset : in std_logic
    );
end window_extractor;

architecture window_extractor_arch of window_extractor is

component simple_dual_port_ram_single_clock is

generic
(
    DATA_WIDTH : natural;
    DATA_NUM : natural
);

port
(
    clk : in std_logic;
    raddr : in natural range 0 to DATA_NUM - 1;
    waddr : in natural range 0 to DATA_NUM - 1;
    data : in std_logic_vector((DATA_WIDTH-1) downto 0);
    we : in std_logic;
    q : out std_logic_vector((DATA_WIDTH -1) downto 0)
);

end component simple_dual_port_ram_single_clock;

-- line buffer

signal raddr, waddr : natural range 0 to IMG_WIDTH - 1;
signal line_buffer_data_rd, line_buffer_data_wr :
    std_logic_vector(((PIXEL_WIDTH * (DRAW_SIZE - 1)) - 1)
    downto 0);
signal wen : std_logic;

-- state

```

```

type state_type is (READ_PIXEL, TRANSFER_WINDOW);

begin

line_buffer: simple_dual_port_ram_single_clock
generic map
(
    DATA_WIDTH => (PIXEL_WIDTH * (DRAW_SIZE - 1)),
    DATA_NUM => IMG_WIDTH
)
port map
(
    clk  => clk,
    raddr  => raddr,
    waddr  => waddr,
    data  => line_buffer_data_wr,
    we    => wen,
    q     => line_buffer_data_rd
);

process (clk, reset)

-- window

variable window : window_type;

variable window_column_temp : window_column;

variable row : natural range 0 to IMG_HEIGHT - 1 := 0;
variable col : natural range 0 to IMG_WIDTH - 1 := 0;
variable step_counter : natural range 0 to
    <→ COMPUTATION_STEPS - 1 := 0;
variable start_index : natural range 0 to DRAW_SIZE - 1 :=
    <→ 0;

variable line_buffer_temp : std_logic_vector((PIXEL_WIDTH
    <→ * (DRAW_SIZE - 1)) - 1) downto 0;

```

```

begin
    if (rising_edge(clk)) then
        if (reset = '1') then
            -- sync reset
            row := 0;
            col := 0;
            step_counter := 0;
        else

            window_valid <= '0';
            wen <= '0';
            raddr <= col;
            waddr <= col;

            if (step_counter = 0) then

                if (pixel_in_valid = '1') then

                    -- lettura colonna corrente dal line buffer
                    line_buffer_temp := line_buffer_data_rd;

                    -- shift a sinistra della window
                    for i in 0 to DRAW_SIZE - 2 loop
                        window(i) := window(i + 1);
                    end loop;

                    -- assegnamento alla window della colonna corrente
                    -- del line buffer
                    window(DRAW_SIZE - 1) := line_buffer_temp &
                    -- pixel_in;

                    line_buffer_temp := line_buffer_temp(((PIXEL_WIDTH
                    -- * (DRAW_SIZE - 2)) - 1) downto 0) & pixel_in;

                    -- shift verso l'alto della colonna del line buffer
                    -- e
                    -- scrittura in ram della colonna del line buffer
                    wen <= '1';

```

```

line_buffer_data_wr <= line_buffer_temp;

-- In uscita mettiamo le colonne meno significative
-- della finestra
start_index := 0;

window_valid <= '1';

for J in 0 to TEMPLATE_SIZE / COMPUTATION_STEPS - 1
    loop
        for I in 0 to TEMPLATE_SIZE - 1 loop
            window_out(J)(I*PIXEL_WIDTH + PIXEL_WIDTH - 1
                ↓  downto I*PIXEL_WIDTH) <= window(J*2)((I*2 +
                ↓  1)*PIXEL_WIDTH + PIXEL_WIDTH - 1 downto (I*2 +
                ↓  + 1)*PIXEL_WIDTH);
        end loop;
    end loop;

step_counter := 1; -- cambio stato

end if;

else -- trasferimento finestra

window_valid <= '1';

-- aggiornamento finestra di uscita
start_index := start_index + DRAW_SIZE /
    COMPUTATION_STEPS;

for J in 0 to TEMPLATE_SIZE / COMPUTATION_STEPS - 1
    loop
        for I in 0 to TEMPLATE_SIZE - 1 loop
            window_out(J)(I*PIXEL_WIDTH + PIXEL_WIDTH - 1
                ↓  downto I*PIXEL_WIDTH) <= window(start_index +
                ↓  J*2)((I*2 + 1)*PIXEL_WIDTH + PIXEL_WIDTH - 1
                ↓  downto (I*2 + 1)*PIXEL_WIDTH);
        end loop;
    end loop;

```

```

        end loop;
    end loop;

    step_counter := (step_counter + 1) mod
        COMPUTATION_STEPS;
    -- se step_counter = 0, cambio stato

end if; -- step_counter

if (pixel_in_valid = '1') then
    -- Incremento il contatore di colonna
    col := (col + 1) mod IMG_WIDTH;
    if (col = 0) then
        row := (row + 1) mod IMG_HEIGHT;
    end if;
end if;

end if; -- reset

window_seq <= step_counter - 1;

end if; -- clock

end process;

end architecture;

```

Part III

RISULTATI e CONCLUSIONI

L'architettura di visione realizzata consente quindi di riconoscere ad una velocità di 30 frame al secondo fino a 4 template in parallelo di dimensione 32×32 oppure di dimensione 64×64 utilizzando una finestra sottocampionata.

La struttura realizzata è altamente modulare: come già sottolineato la pipeline di template matching è stata costruita sopra una struttura base composta dalla sola estrazione dei pixel dal sensore, dal loro salvataggio su SRAM, e dal loro recupero e visualizzazione su VGA. Sarebbe possibile senza alcun problema implementare altri algoritmi di visione sostituendo quello del template matching senza apportare modifiche alla struttura base. Allo stesso modo anche la pipeline di template matching manifesta una elevata modularità: come si è già in parte visto nella pipeline ottimizzata, è possibile:

- Modificare la logica di generazione della finestra senza modificare il calcolo dei punteggi (SAD) e la loro interpretazione (template_aggregator).
- Modificare il calcolo dei punteggi (sostituire per esempio la SAD con le più robuste NCC o ZNCC) senza modificare la generazione della finestra e l'interpretazione dei punteggi.
- Modificare l'interpretazione finale dei punteggi calcolati dalla pipeline senza modificare gli stadi a monte.

Tutti i moduli realizzati sono stati inoltre parametrizzati (dove possibile) tramite l'uso dei generici o di costanti dichiarate in package esterni.

Per quanto riguarda l'utilizzo delle risorse, come già sottolineato l'implementazione della architettura con riconoscimento di template di dimensione 16 porta ad un utilizzo di risorse logiche del 20% di quelle disponibili sulla DE2. Portando la dimensione a 32, o utilizzando la versione ottimizzata a dimensione 64, si raggiunge rispettivamente l'80% e l'87% delle risorse. Per quanto riguarda la memoria integrata, utilizzata soltanto dal line buffer per la generazione della finestra, si raggiunge l'8% nel caso di 64 linee, lasciando perciò spazio a possibili e ulteriori ottimizzazioni che utilizzano finestre di dimensione maggiore.

Il sensore purtroppo senza una opportuna programmazione non consente di ottenere immagini di grande qualità, perciò si è dimostrata più efficace la programmazione tramite flusso video, che ricava il template sorgente dalla stessa immagine rumorosa prodotta dal sensore, piuttosto che quella tramite seriale, che prevede il riconoscimento di immagini "pulite". L'algoritmo SAD si è rivelato, come previsto dalla teoria, poco robusto a cambiamenti di luminosità e a gestire il rumore della camera; tuttavia, per gli scopi di questo progetto, non abbiamo avuto difficoltà ad utilizzarlo per i test e a verificarne il funzionamento.

Nei test effettuati la pipeline di riconoscimento template con finestra di dimensione 64 sottocampionata non ha comportato una perdita di qualità nel match rispetto alle architetture realizzate con finestra completamente utilizzata.

Concludiamo infine lasciando alcune immagini di esempio del ritrovamento di alcuni semplici template:

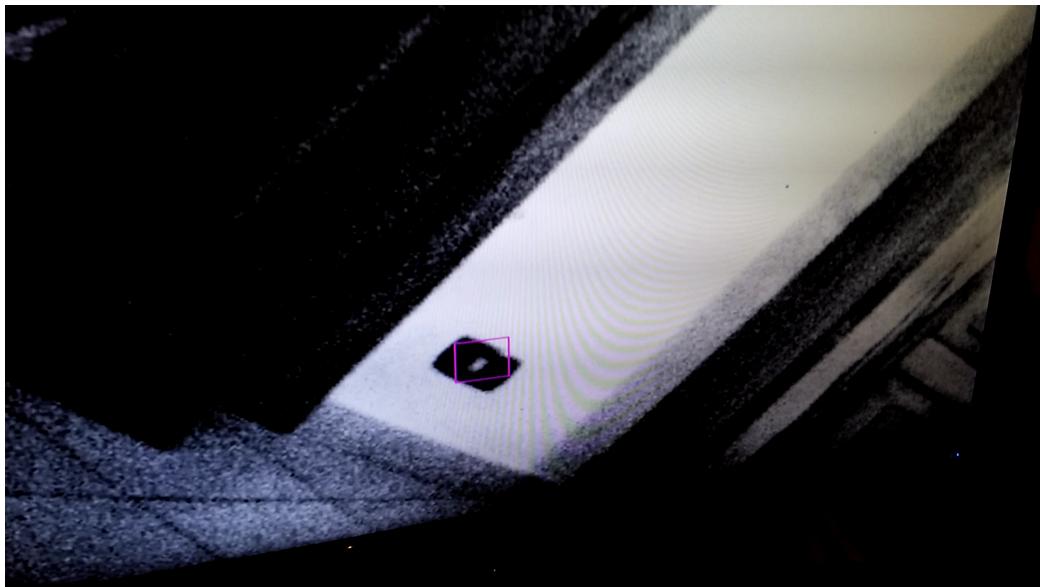


Figure 32: Riconoscimento di una presa di corrente con template di dimensione 32×32 . I 4 template erano stati programmati per riconoscere lo stesso oggetto ma sotto differenti rotazioni.

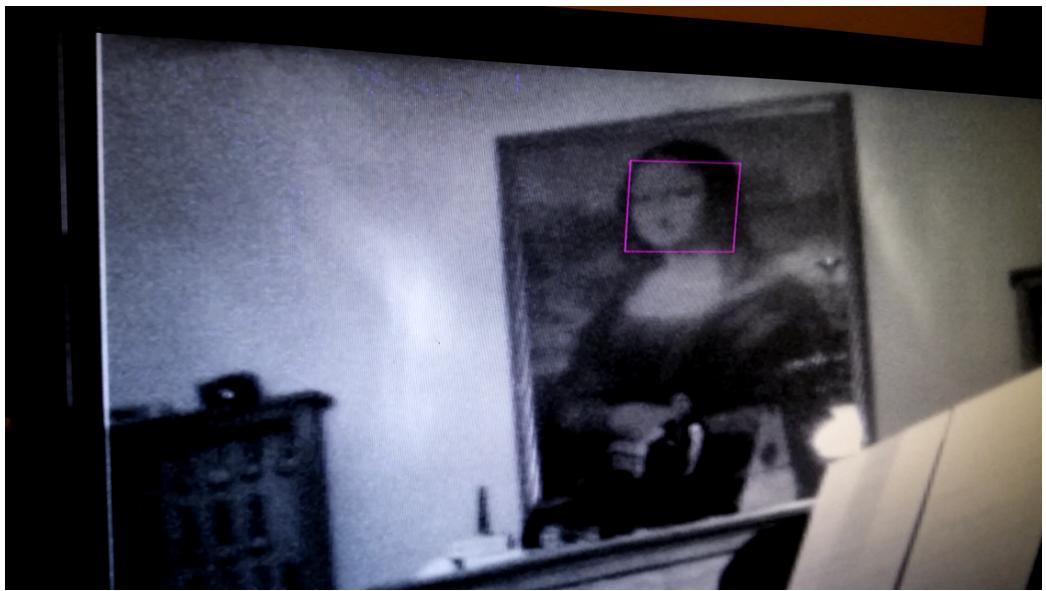


Figure 33: Riconoscimento del volto della Gioconda utilizzando la pipeline ottimizzata a dimensione 64×64 .