# CS 476: Labo DE1-SoC
# Final Project

Due on Monday, June 6, 2016

*Process multiple images stored or generated by a VHDL component and display them on a monitor via a VGA interface*

**Olivier Lévêque, Maxime Maurin & Kevin Tang**

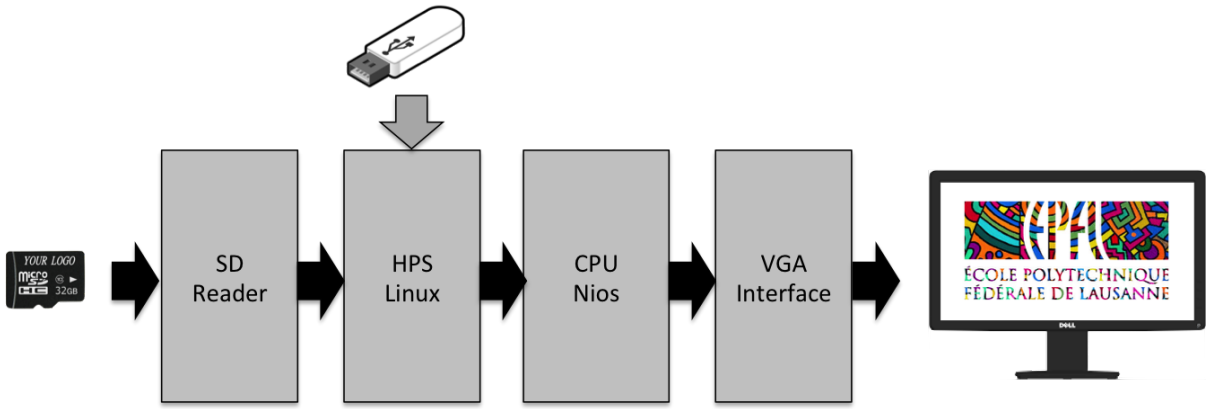
# Contents

# 1 Introduction

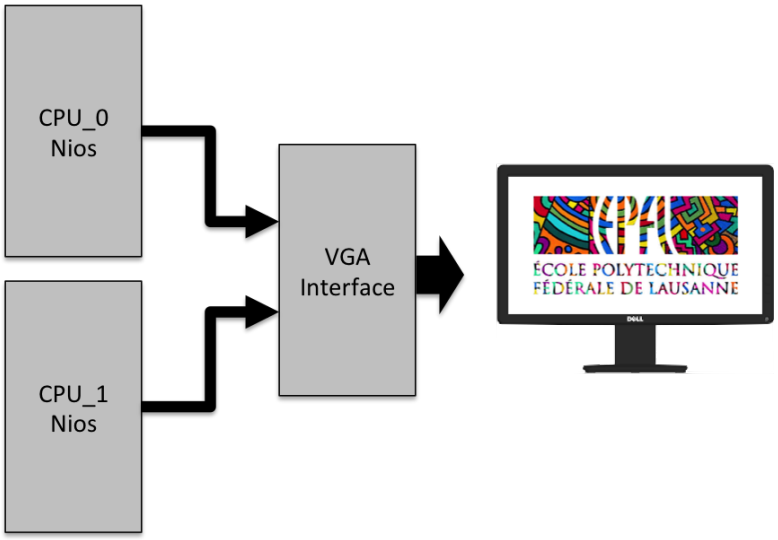
At the beginning of the project, we wanted to acquire a jpeg image from a USB key, apply a image processing on it and display it on a monitor via a VGA interface.

To realize these different functions, we needed to use a Hard Processor System (HPS) with Linux to acquire a jpeg image from USB key and decompress it before to store it in a SDRAM memory. An other processor, a CPU Nios, would have received a confirmation from the HPS that the image acquisition works, and it would have started the process to transform and display the image on a monitor. This process would have been realized by a VHDL component.

The following figure illustrates this initial process.

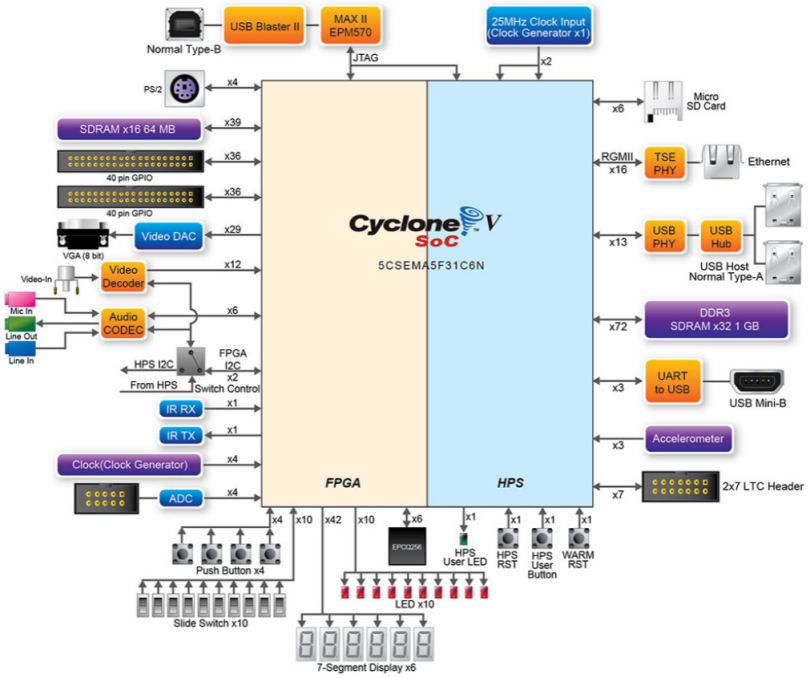



Unfortunately, we did not manage to install Linux on the HPS...So we removed this part. We built a new architecture with two CPU Nios, CPU_0 and CPU_1, and kept our VHDL component, called 'VGA interface' which can process an array image and display it on a monitor via a VGA interface.
The idea is to change the image displayed with CPU_0 and change the image processing with CPU_1. For our demonstration, the picture changes once all the different image processing were executed and the image processing is updated every second.
The two CPUs cannot access the component 'display img' simultaneously because the access is controlled by a mutex.
The following figure illustres this new process.



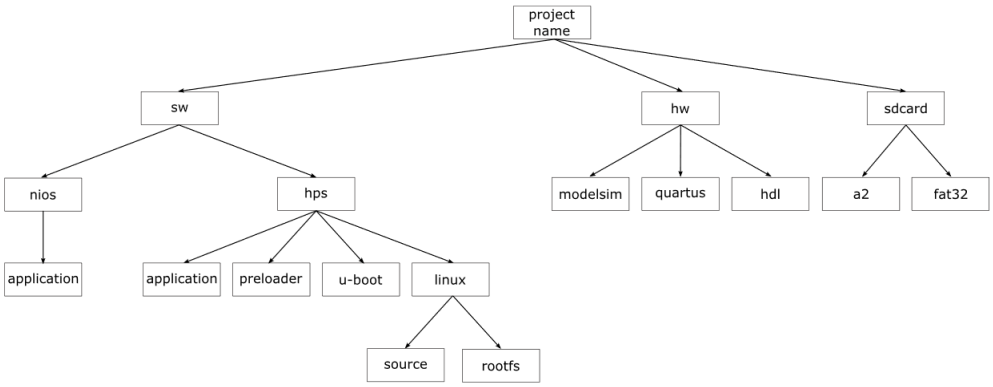
# 2 Hard Processor/Linux Boot

Being part of our original project architecture, we wanted to use the hard processor to be able to upload an image in our FPGA 64 MB SDram from an USB key connected to the HPS part of the board. We would have used a bridge between HPS and FPGA part to perform the data exchange and to synchronize the end of the upload and start the image processing on the FPGA. To be able to do so we would have needed the Hard Processor to run Linux operating system.




We didn't succeed in making this part of our project work, but we achieve some other parts. To configure correctly the ARM Cortex-A9 we refer to the DE1-SoC Guide provided at the beginning of the course.

To organize our project well we structured our template to be able to distinguish the "hw" part, the "sw" part and the sd-card part containing every target files such as the we obtained a valid sdcard from which the ARM cortex A9 can boot properly.




After this, the first step was to make the hardware part. It was composed of :

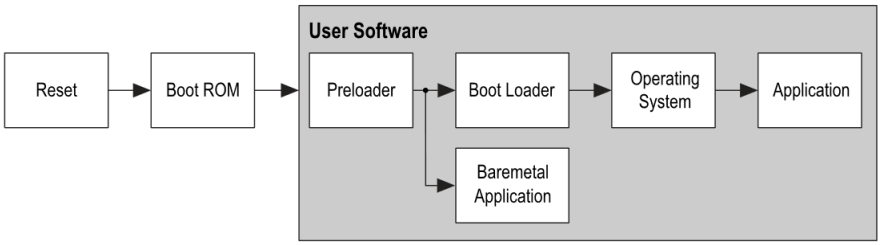
FPGA Component :

- An Sdram controller
- PLL
- Nios II/f processor
- Jtag UART
- VGA COMPONENT

HPS Part :

- Arria v Hard Processor System
- Parallel I/O for testing purpose.

We realized this achitecture in Qsys, configuring the Cyclone V Hard Processor according to the tutorial and to our own needs. After it we instanciated every component in our top level, executed TCL script for HPS DDR3 Pin assignement automatically generate by Altera Qsys Compiler.
Then we moved to the second step, configuring the sd card to run linux on HPS part. The ARM can only boot from an Sd card on the DE1-SoC board we have.




When we start the card, the reset executes the code contain in the Boot ROM which refers to the preloader which would call the boot source such that BOOT can be perform. Thus Operating System/Application are initialized.

First we partitionate the sdcard in FAT32 partition for various boot-time files like U-Boot configuration script or linux kernel zIMAGE file. Lot of manipulation are needed to manage a correct implementation of the preloader, the bootloader, the Linux kernel and ubuntu cores files and we are not going into details right now.

After this operation you can now plug your sd card in the HPS and use the HPS Jtag-Uart serial communiction to check if your board boots correctly.

We succeeded to establish the communication between our computer and the board using minicom, an application available on Linux Operating System. Unfortunately, The board sent us back an error explaining the U-Boot file cannnot be read / executed. We didn't have enough time to find where the problem came from.

After that we would have configured our Linux system with the help of ARM DS-5 Altera tools.

# 3 Design of our 'VGA Interface' component

## 3.1 Introduction

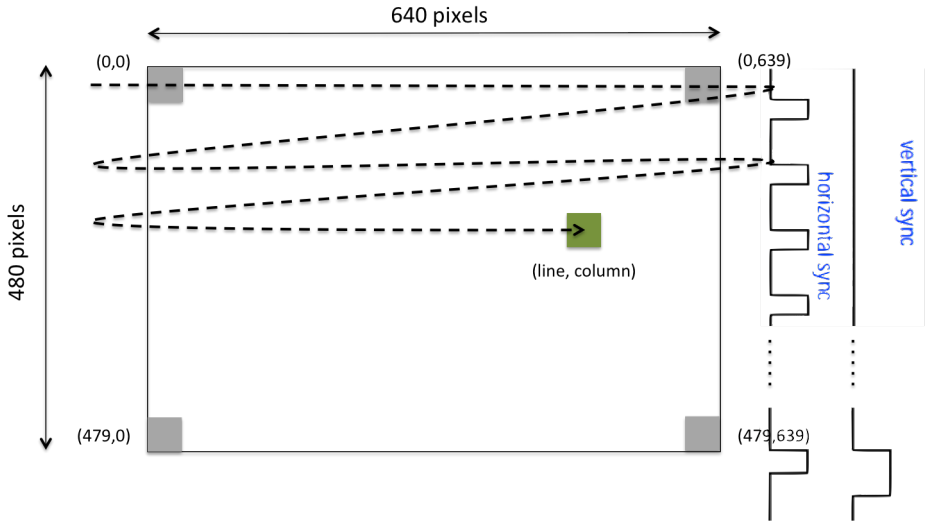
VGA is a standard interface for controlling analog monitors, developed by IBM company in 1987. It is widely used in color displays because it has high resolution, high refresh rate and color diversity.
In our project, we will use the VGA connector available on the Altera board to display on a monitor an image.

## 3.2 Principle to display an image on monitor

To generate an colorful image, the screen will be scanned by a pointer to display it pixel by pixel.

The scan begins from the top left corner of the screen and scans pixel by pixel from left to the right line by line. After one line is scanned, the pointer moves back to the left start point of the next line and the pointer is synchronized by horizontal synchronization signal. The pointer scans the screen line by line until it reaches the bottom right corner of the screen. After all lines scanned, the process restarts and the pointer is synchronized by horizontal synchronization signal.

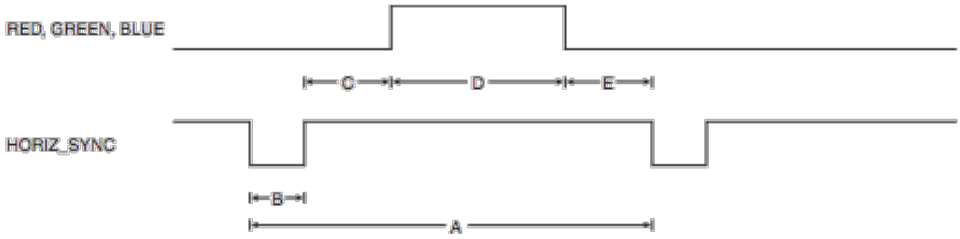
The following figure illustrates this principle.




## 3.3 Timing Analysis

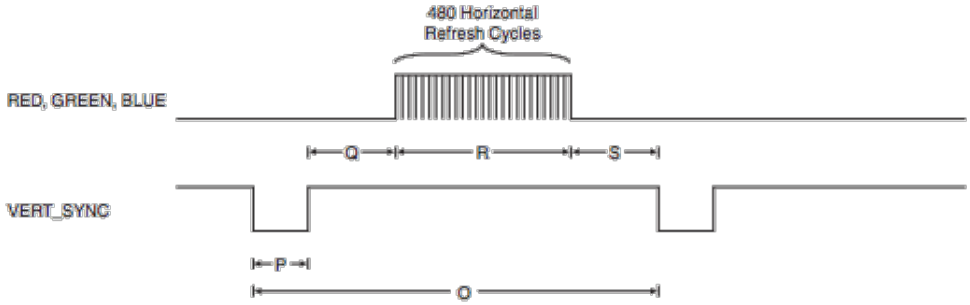
VGA Timing includes horizontal timing and vertical timing. Both the horizontal timing and vertical timing contain Synchronization Pulse, Back Porch, Display Interval and Front porch.

At the beginning of the horizontal timing, a negative pulse is generated for horizontal synchronization. It represents the end of one line and the start of the next line. After the synchronization pulse is the back porch. During the display interval, RGB data drives pixels of one line, generating one line of an image. At the end of a line there is the front porch. The image is only displayed during the display interval. The synchronization pulse (B), back porch (E), and front porch (C) are in the Horizontal Blanking Interval during which the RGB data is invalid and the image is not displayed on the screen. The following figure illustrates the VGA horizontal timing.




A: Whole line / B: Sync Pulse / C: Front Porch / D: Visible area / E: Back Porch

The VGA vertical timing is basically the same as the VGA Horizontal Timing. The difference is that vertical synchronization pulse represents the end of one image frame and the start of the next frame. The RGB data during display interval includes all lines of the screen. The following figure demonstrates the VGA vertical timing.




O: Whole frame / P: Sync Pulse / Q: Front Porch / R: Visible area / S: Back Porch

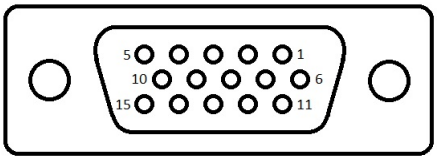
VGA timing parameters vary at different resolutions. In this project, we have used the Dell UltraSharp U2412M Monitor avaibled in INF3. This monitor has a 1920x1200 resolution.
According VGA Timing Specifications avaibled on `http://tinyvga.com/vga-timing/640x480@60Hz`, we used the following specifications.

| | | | |
|---|---|---|---|
| **Resolution (pixels)** | | 640x480 | |
| **Refresh Rate** | | 60 $Hz$ | |
| **Pixel Clock** | | 25 $MHz$ | |
| **Horizontal** | | | |
| | Visible area | 640 pixels | 25.42$\mu s$ |
| | Front Porch | 16 pixels | 0.63 $\mu s$ |
| | Sync Pulse | 96 pixels | 3.81 $\mu s$ |
| | Back Porch | 48 pixels | 1.91 $\mu s$ |
| | Whole line | 800 pixels | 31.78 $\mu s$ |
| **Vertical** | | | |
| | Visible area | 480 pixels | 15.25 $ms$ |
| | Front Porch | 10 pixels | 0.32 $ms$ |
| | Sync Pulse | 2 pixels | 0.06 $ms$ |
| | Back Porch | 33 pixels | 1.05 $ms$ |
| | Whole frame | 525 pixels | 16.68 $ms$ |
| **h_sync Polarity** | | n | |
| **v_sync Polarity** | | p | |

Table: Timing Specifications for 640x480 at 60Hz

## 3.4 VGA interface

The VGA interface has fifteen pins: three lines and each line has five pins. The following table explains the definition and description for each pin. The most important pins are: three video pins (Red, Green and Blue Video output pins) and two synchronization output pins (Horizontal Sync and Vertical Sync pins).

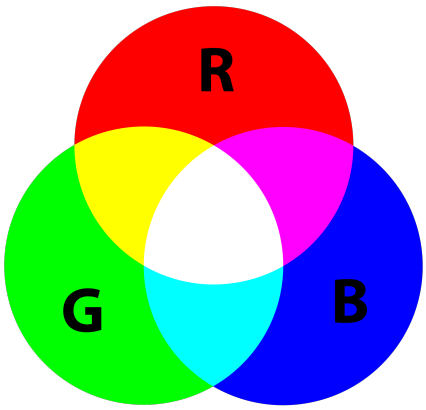



VGA Connector female

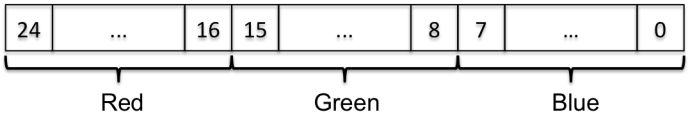
| | | | |
|---|---|---|---|
| **1** | R | analog red, 0-0.7V | DAC output |
| **2** | G | analog green, 0-0.7V or 0.3-1V (if sync-on-green) | DAC output |
| **3** | B | analog blue, 0-0.7V | DAC output |
| **4** | EDID Interface | | no connect |
| **5** | GND | general | GND |
| **6** | GND | for R | GND |
| **7** | GND | for G | GND |
| **8** | GND | for B | GND |
| **9** | no pin | (optional 5V) | no connect |
| **10** | GND | for h_sync and v_sync | GND |
| **11** | EDID Interface | | no connect |
| **12** | EDID Interface | | no connect |
| **13** | h_sync | horizontal sync, 0V/5V waveform | FPGA output |
| **14** | v_sync | vertical sync, 0V/5V waveform | FPGA output |
| **15** | EDID Interface | | no connect |

## 3.5 RGB Color model

RGB Color Model is an additive color model. It generates various colors by adding three basic colors (red, green and blue), as shown in the following Figure.



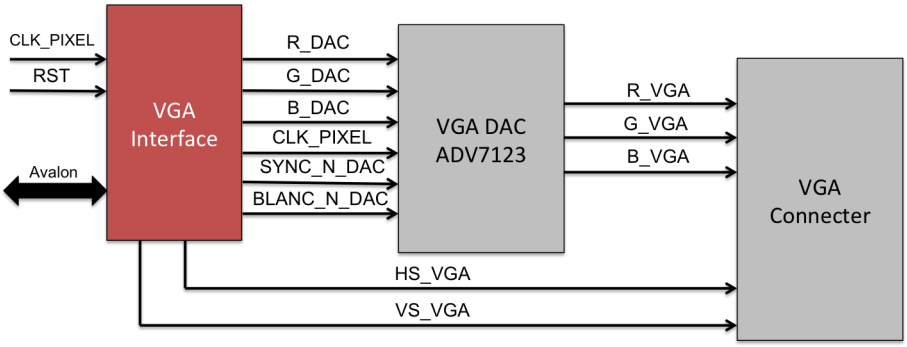

To obtain a color, we use a combinaison of these three colors, called 'component'. A component is defined as an integer number ranging from 0 to 255 (8 bits), as illustruted in the following Figure.




For example, black is represented as (0, 0, 0) because it contains zero of all the three basic colors, Green is represented as (0, 255, 0), and White is represented as (255, 255, 255).

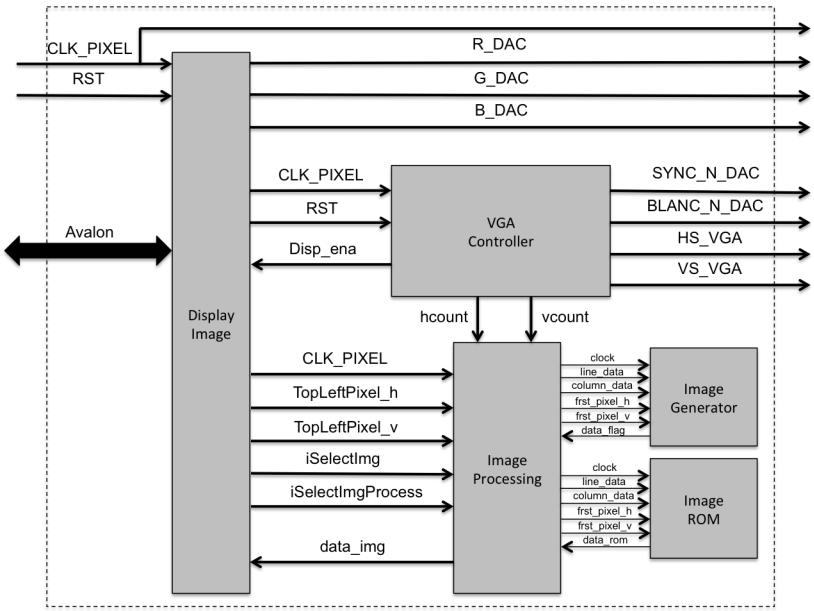
## 3.6 Implementation on the Altera board

To displayed an image, the computing side of the interface needs to provide the monitor with horizontal and vertical sync signals, color magnitudes, and ground references, like illutrates the following figure.




The horizontal and vertical sync signals are 0V/5V digital waveforms that synchronize the signal timing with the monitor. Being digital, they are provided directly by the FPGA (3.3V meets the minimum threshold for a logical high, so 3.3V can be used instead of 5V).
Concerning color magnitudes signal, we need to use a ADV7123 component, already implemented on the Altera board, to convert digital signals to analog signals.

To use the VGA interface, we need to build a VGA Controller to generate the SYNC_N, BLANC_N, HS, VS signals as presented previously, and to build the rest of component in VHDL to provide a image as describe in our scope statement.
The following Figure illustres our architecture.



| Internal Register | Name | Function | Write/Read |
|---|---|---|---|
| 0 | Start | Display image on the monitor | W/R |
| 1 | SelectImg | Select an image | W/R |
| 2 | SelectImgProcess | Select an image proccessing | W/R |

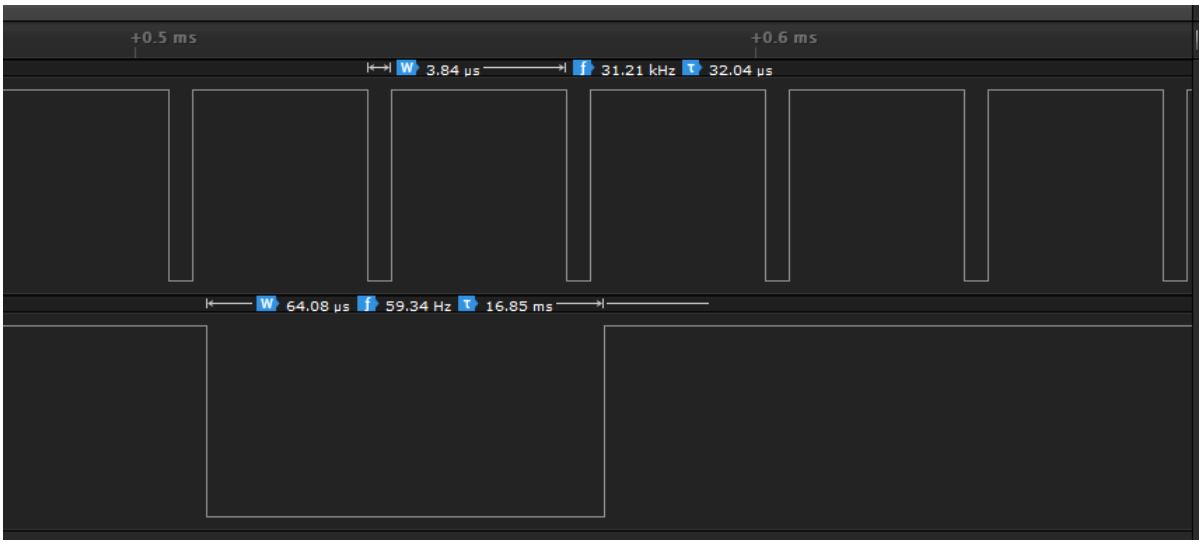
We use the PLL provided by Quartus to generate the CLK_PIXEL clock and we use a switch on the board to generate the RST signal.

### 3.6.1 VGA Controller

The VGA controller contains two counters:

- One counter increments on pixel clocks and controls the timing of the H_SYNC (horizontal sync) signal. The display time starts at counter value 0, the counter value equals the pixel's column coordinate during the display time. The horizontal display time is followed by a blanking time, which includes a horizontal front porch, the horizontal sync pulse itself, and the horizontal back porch, each of specified duration. At the end of the row, the counter resets to start the next row.
- The second counter increments as each row completes, therefore controlling the timing of the V_SYNC (vertical sync) signal. Again, this is set up such that the display time starts at counter value 0, so the counter value equals the pixel's row coordinate during the display time. As before, the vertical display time is followed by a blanking time, with its corresponding front porch, sync pulse, and back porch. Once the vertical blanking time completes, the counter resets to begin the next screen refresh.

The following screenshot (obtained with the Logic Analyser) illustrates the timing signals produced by our VGA controller. Timings are respected.




This VHDL Code used to synthetise our VGA Controller is avaible in Annexes.

# 4 Image processing

## 4.1 Image storage

As we have not succeeded to install Linux on the HPS. We found a new way to store an image in our architecture.

We wrote a matlab program to decompress a jpeg image and convert it in an array of pixel where the color is coded in hexadecimal (on 24 bits).

The following code is the Matlab code that we written and used.

Listing 1: Matlab code for pre-processing

```
close all; clear all;

filename = 'EPFL-color'; %image path
src = imread([filename, '.jpg']); %decompression

array = [];
for i = 1:size(src,1) %for each horizontal line
    disp(i);
    line = ['('];
    for j = 1:size(src,2) %for each pixel in the line
         R = double(src(i,j,1));
         G = double(src(i,j,2));
         B = double(src(i,j,3));
         line = [line, 'X"', dec2hex(65536*R + 256*G + B, 6), '"']; %extract the color pixel in
         if j<size(src,2)
             line = [line, ','];
         end
    end
    line = [line, '),'];

    if isempty(array)
        array = line;
    else
        array = [array; line];
    end
end

dlmwrite([filename, '.txt'], array,'delimiter','');  %save in txt file
```

After this operation the result is saved in a txt file. We copy the result in our VHDL component 'Img_ROM' where the image is stored in an array of std_logic_vector of 24 bits.

Listing 2: Part of VHDL code from 'Img_ROM' component

```
type rom_type is array(0 to 159, 0 to 329) of std_logic_vector(23 downto 0);

 -- ROM definition
constant ROM: rom_type :=(
     (X"FEFFFF",X"D7D6D2",X"FFC1AD",X"FFACBF",X"FFA2B0",....),
     (X"FEFFFF",X"858480",X"4D0000",X"7D081B",X"8C000C",....),
```

```
    ......
    ......
    -- All pixel are not written here
    ......
    ......
    (X"FFF3FF",X"F7EFED",X"F5F5DD",X"FBFEDF",X"F4F2E5",....),
    (X"FFF9FF",X"FFFDFB",X"FEFEE6",X"FDFFE1",X"FFFEF1",....)
);
```

## 4.2 Processing

To process an image, we need, in the first time to extract its color information.

The following part of VHDL code illustres how we get RGB color and how we compute grayscale with hardware approximation.

Listing 3: Part of VHDL code from 'Image processing' component

```
red <= unsigned(data_rom(23 downto 16));
green <= unsigned(data_rom(15 downto 8));
blue <= unsigned(data_rom(7 downto 0));

gray <= (unsigned(data_rom(23 downto 16)) srl 2)+(unsigned(data_rom(23 downto 16)) srl 5)+(unsig
              (unsigned(data_rom(15 downto 8)) srl 1)+(unsigned(data_rom(15 downto 8)) srl 4)+(
              (unsigned(data_rom(7 downto 0)) srl 4)+(unsigned(data_rom(7 downto 0)) srl 5)+(un
```

We can apply three different processing on an image: grayscale, binary image, and invert color.
After recuperation of RGB color data, we can compute grayscale with the following formula:

$$intensity = 0.2989 * R + 0.5870 * G + 0.1140 * B$$

To compute it, we have used hardware approximation:

$$\begin{aligned} intensity = & R >> 2 + R >> 5 + R >> 6 + \\ & G >> 1 + G >> 4 + G >> 5 + G >> 6 + G >> 7 + \\ & B >> 4 + B >> 5 + B >> 6 + B >> 7 \end{aligned}$$

The binary image is obtained comparing the grayscale intensity with a threshold. And the invert color process is obtained with the following formula :

$$\begin{aligned} R_{inv} &= 255 - R \\ G_{inv} &= 255 - G \\ B_{inv} &= 255 - B \end{aligned}$$

**All of our processing image are executed pixel by pixel.**

The following part of VHDL code illustres how we have implemented these processing.

Listing 4: Part of VHDL code from 'Image processing' component

```
if (slct_Prc_img = "00") then -- no image processing
    data_out(23 downto 16) <= std_logic_vector(red);
```

```
     data_out(15 downto 8) <= std_logic_vector(green);
     data_out(7 downto 0) <= std_logic_vector(blue);

elsif (slct_Prc_img = "01") then    -- grayscale
     data_out(23 downto 16) <= std_logic_vector(gray);
     data_out(15 downto 8) <= std_logic_vector(gray);
     data_out(7 downto 0) <= std_logic_vector(gray);

elsif (slct_Prc_img = "10") then    -- binary image
     if (gray > 235) then
          data_out <= (others => '1');
     else
          data_out <= (others => '0');
     end if;

elsif (slct_Prc_img = "11") then    -- invert color
     data_out(23 downto 16) <= std_logic_vector(255-red);
     data_out(15 downto 8) <= std_logic_vector(255-green);
     data_out(7 downto 0) <= std_logic_vector(255-blue);

end if;
```

Some screenshots of our results.


a) Original image b) Grayscale

c) Binary image d) Inverted color

# 5 General conclusion

To conclude, we have realised a real time system on an Altera board which is able to process and display an image via VGA interface. The system use two different CPUs which can access to common ressources. The access regulation is realised with a mutex.

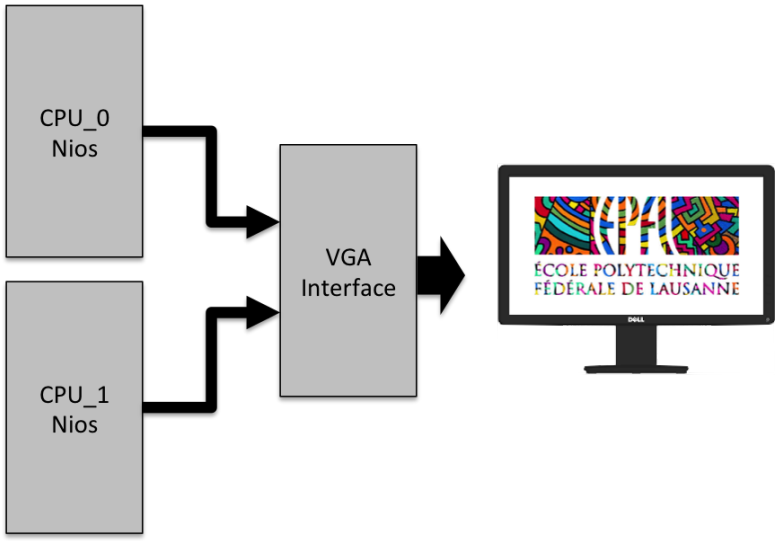
A page resume of our project and the integrality of our VHDL codes are avaibled in annexes.

# 6 Annexes

## 6.1 Page resume

### 6.1.1 Our project

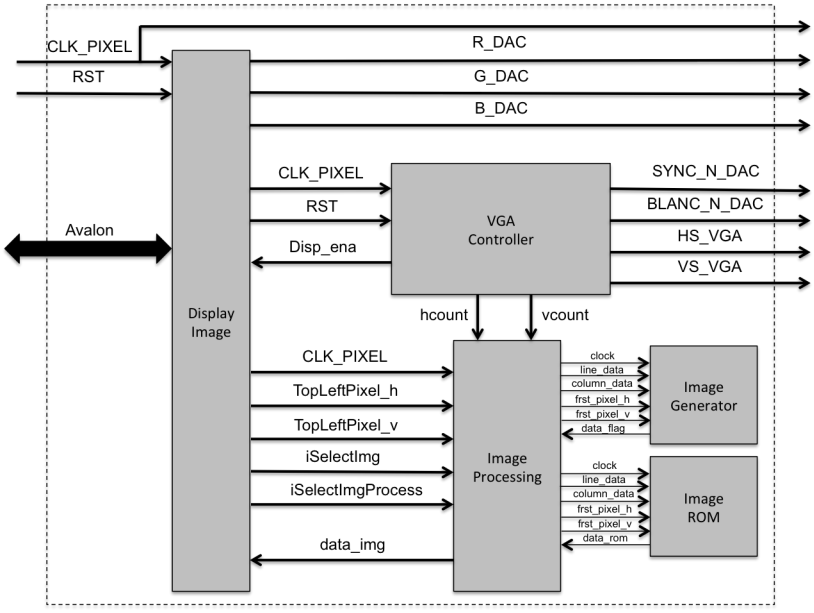
Objective: Process multiple images stored or generated by a VHDL component and display them on a monitor via a VGA interface.




- The CPU_0 changes the picture once all the different image processing were executed
- The CPU_1 changes the image processing every second

### 6.1.2 Details of VGA Display component

VGA Display is constituted with several VHDL component. We have designed all these components.



The register map to communicate with the Avalon Slave to Display image component is resumed in the following tab.

| Internal Register | Name | Function | Write/Read |
|---|---|---|---|
| 0 | Start | Display image on the monitor | W/R |
| 1 | SelectImg | Select an image | W/R |
| 2 | SelectImgProcess | Select an image proccessing | W/R |

### 6.1.3 Work distribution

| Name | Olivier Lévêque | Maxime Maurin | Kévin Tang |
|---|---|---|---|
| HPS & Linux | | x | x |
| Design of architecture with Qsys | | x | x |
| Design of VHDL components for processing image | x | | x |
| Design of VHDL components to store or generate image | x | | |
| Design of VHDL components for VGA interface | x | | |

## 6.2 VHDL Codes

### 6.2.1 Display image component in VHDL

This following VHDL Code is used to synthetise our Display image component.

Listing 5: Display image component

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

entity display_img is
    PORT(
        --Avalon interfaces signals
        reset          : in std_logic;
        clk            : in std_logic;
        Address        : in std_logic_vector(2 downto 0);
        ChipSelect     : in std_logic;
        Read           : in std_logic;
        Write          : in std_logic;
        ReadData  : out std_logic_vector (31 downto 0);
        WriteData : in std_logic_vector(31 downto 0);

        --VGA signals
        HS_VGA : out std_logic;
        VS_VGA : out std_logic;

        N_BLANK_DAC : out std_logic;
        N_SYNC_DAC     : out std_logic;

        G_DAC : out std_logic_vector(7 downto 0);
        R_DAC : out std_logic_vector(7 downto 0);
        B_DAC : out std_logic_vector(7 downto 0);

        switch1 : in std_logic;
        switch2 : in std_logic;
        switch3 : in std_logic;
        switch4 : in std_logic;
        switch5 : in std_logic
    );
end display_img;

architecture Behavioral of display_img is

    COMPONENT vga_controller_640_60 IS
    PORT(
        rst         : in std_logic;
        pixel_clk   : in std_logic;

        HS          : out std_logic;
        VS          : out std_logic;

        hcount      : out std_logic_vector(10 downto 0);
        vcount      : out std_logic_vector(10 downto 0);
```

```
        disp_ena    : out std_logic;

        n_blank        : out std_logic;
        n_sync         : out std_logic
    );
    END COMPONENT;

    COMPONENT img_process IS
    port(
        --port for img_ROM
        clk               : in  std_logic;
        v_count           : in std_logic_vector(10 downto 0);
        h_count           : in std_logic_vector(10 downto 0);
        TopLeftPixel_h : in std_logic_vector(10 downto 0);
        TopLeftPixel_v : in std_logic_vector(10 downto 0);

        --port for img_process
        slct_img       : in  std_logic_vector(1 downto 0);
        slct_Prc_img   : in  std_logic_vector(1 downto 0);
        data_out       : out std_logic_vector(23 downto 0)
    );
    END COMPONENT;

    signal iStart                 : std_logic;
    signal iSelectImg             : std_logic_vector(1 downto 0);
    signal iSelectImgProcess      : std_logic_vector(1 downto 0);

    signal green :  std_logic_vector(7 downto 0);
    signal blue  :  std_logic_vector(7 downto 0);
    signal red   :  std_logic_vector(7 downto 0);

    signal display : std_logic;

    signal h_count_int : std_logic_vector(10 downto 0);
    signal v_count_int : std_logic_vector(10 downto 0);

    signal data_img : std_logic_vector(23 downto 0);


    begin
    i1 : vga_controller_640_60 port map (
        rst => reset,
        pixel_clk => clk,

        HS => HS_VGA,
        VS => VS_VGA,

        hcount => h_count_int,
        vcount => v_count_int,

        disp_ena => display,
```

```
        n_blank => N_BLANK_DAC,
        n_sync => N_SYNC_DAC
    );

    i2 : img_process port map (
        --port for img_ROM
        clk               => clk,
        v_count           => v_count_int,
        h_count           => h_count_int,
        TopLeftPixel_h => "00010011011", --70 pixels
        TopLeftPixel_v => "00010100000", --119 pixels

        --port for img_process
        slct_img         => iSelectImg,
        slct_Prc_img     => iSelectImgProcess,
        data_out        => data_img
    );


    R_DAC <= red;
    G_DAC <= green;
    B_DAC <= blue;



    --Avalon communications
    pRegWr    : process(clk,reset)
    begin
        if reset = '1' then -- asynchonous Reset
            iStart <= '0';
            iSelectImg <= "00";
            iSelectImgProcess <= "00";
        elsif rising_edge(Clk) then
            if ChipSelect = '1' and Write = '1' and switch5='1' then
                case Address (2 Downto 0) is
                    when "000" => iStart <= WriteData(0);
                    when "001" => iSelectImg <= WriteData(1 downto 0);
                    when "010" => iSelectImgProcess <= WriteData(1 downto 0);
                    when others => null;
                end case;
            elsif switch5='0' then
                iStart<=switch1;
                iSelectImg(0)<=switch2;
                iSelectImg(1)<='0';
                iSelectImgProcess(0)<=switch3;
                iSelectImgProcess(1)<=switch4;
            end if;
        end if;
    end process pRegWr;

    pRegRd: process(Clk)
    begin
        if rising_edge(Clk) then
```

```
                ReadData <= (others => '0');
                if ChipSelect = '1' and Read = '1' then
                    case Address(2 Downto 0) is
                        when "000" => ReadData(0) <= iStart;
                        when "001" => ReadData(1 downto 0) <= iSelectImg;
                        when "010" => ReadData(1 downto 0) <= iSelectImgProcess;
                        when others => null;
                    end case ;
                end if;
            end if;
        end process pRegRd;

        --Display VGA interface
        affichage: process(h_count_int,v_count_int,display)
        begin
            if (display='1' or iStart='0') then
                blue <= (others => '0');
                green <= (others => '0');
                red <= (others => '0');

            else
                blue <= data_img(7 downto 0);
                green <= data_img(15 downto 8);
                red <= data_img(23 downto 16);

            end if;
        end process;

end Behavioral;
```

### 6.2.2 VGA controller in VHDL

This following VHDL Code is used to synthetise our VGA Controller.

Listing 6: VGA Controller in VHDL

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


entity vga_controller_640_60 is
port(
   rst         : in std_logic;          --active high asycnchronous reset
   pixel_clk   : in std_logic;          --pixel clock at frequency of VGA mode being used

   HS          : out std_logic;         --horiztonal sync pulse
   VS          : out std_logic;         --vertical sync pulse

   hcount      : out std_logic_vector(10 downto 0);   --horizontal pixel coordinate
   vcount      : out std_logic_vector(10 downto 0);   --vertical pixel coordinate

   disp_ena    : out std_logic;           --display enable ('1' = display time, '0' = blanking t

   n_blank     :  out std_logic;          -- direct blacking output to DAC
   n_sync      :  out std_logic           -- sync-on-green output to DAC
);
end vga_controller_640_60;

architecture Behavioral of vga_controller_640_60 is

-----------------------------------------------------------------------
-- CONSTANTS
-----------------------------------------------------------------------

-- maximum value for the horizontal pixel counter
constant HMAX  : std_logic_vector(10 downto 0) := "01100100000"; -- 800

-- maximum value for the vertical pixel counter
constant VMAX  : std_logic_vector(10 downto 0) := "01000001101"; -- 525

-- total number of visible columns
constant HLINES: std_logic_vector(10 downto 0) := "01010000000"; -- 640

-- value for the horizontal counter where front porch ends
constant HFP   : std_logic_vector(10 downto 0) := "01010001000"; -- 648

-- value for the horizontal counter where the synch pulse ends
constant HSP   : std_logic_vector(10 downto 0) := "01011101000"; -- 744

-- total number of visible lines
constant VLINES: std_logic_vector(10 downto 0) := "00111100000"; -- 480

-- value for the vertical counter where the front porch ends
constant VFP   : std_logic_vector(10 downto 0) := "00111100010"; -- 482
```

```
50
   -- value for the vertical counter where the synch pulse ends
   constant VSP   : std_logic_vector(10 downto 0) := "00111100100"; -- 484

   -- polarity of the horizontal and vertical synch pulse
55 -- only one polarity used, because for this resolution they coincide.
   constant SPP   : std_logic := '0';

   ---------------------------------------------------------------------------
   -- SIGNALS
60 ---------------------------------------------------------------------------

   -- horizontal and vertical counters
   signal hcounter : std_logic_vector(10 downto 0) := (others => '0');
   signal vcounter : std_logic_vector(10 downto 0) := (others => '0');
65
   -- active when inside visible screen area.
   signal video_enable: std_logic;

   begin
70
      -- output horizontal and vertical counters
      hcount <= hcounter;
      vcount <= vcounter;

75    -- disp_ena is active when outside screen visible area
      -- color output should be blacked (put on 0) when disp_ena in active
      -- disp_ena is delayed one pixel clock period from the video_enable
      -- signal to account for the pixel pipeline delay.
      disp_ena <= not video_enable when rising_edge(pixel_clk);
80
      -- enable video output when pixel is in visible area
      video_enable <= '1' when (hcounter < HLINES and vcounter < VLINES) else '0';

      --no direct blanking
85    n_blank <= '1';
      --no sync on green
      n_sync <= '0';

      -- increment horizontal counter at pixel_clk rate
90    -- until HMAX is reached, then reset and keep counting
      h_count: process(pixel_clk)
      begin
         if(rising_edge(pixel_clk)) then
            if(rst = '1') then
95             hcounter <= (others => '0');
            elsif(hcounter = HMAX) then
               hcounter <= (others => '0');
            else
               hcounter <= std_logic_vector(unsigned(hcounter) + 1);
100         end if;
         end if;
      end process h_count;
```

```
    -- increment vertical counter when one line is finished
    -- (horizontal counter reached HMAX)
    -- until VMAX is reached, then reset and keep counting
    v_count: process(pixel_clk)
    begin
       if(rising_edge(pixel_clk)) then
          if(rst = '1') then
             vcounter <= (others => '0');
          elsif(hcounter = HMAX) then
             if(vcounter = VMAX) then
                vcounter <= (others => '0');
             else
                vcounter <= std_logic_vector(unsigned(vcounter) + 1);
             end if;
          end if;
       end if;
    end process v_count;

    -- generate horizontal synch pulse
    -- when horizontal counter is between where the
    -- front porch ends and the synch pulse ends.
    -- The HS is active (with polarity SPP) for a total of 96 pixels.
    do_hs: process(pixel_clk)
    begin
       if(rising_edge(pixel_clk)) then
          if(hcounter >= HFP and hcounter < HSP) then
             HS <= SPP;
          else
             HS <= not SPP;
          end if;
       end if;
    end process do_hs;

    -- generate vertical synch pulse
    -- when vertical counter is between where the
    -- front porch ends and the synch pulse ends.
    -- The VS is active (with polarity SPP) for a total of 2 video lines
    -- = 2*HMAX = 1600 pixels.
    do_vs: process(pixel_clk)
    begin
       if(rising_edge(pixel_clk)) then
          if(vcounter >= VFP and vcounter < VSP) then
             VS <= SPP;
          else
             VS <= not SPP;
          end if;
       end if;
    end process do_vs;

end Behavioral;
```

### 6.2.3 Image processing component in VHDL

This following VHDL Code is used to synthetise our Image processing component.

Listing 7: Image processing component

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

entity img_process is
   port(
          --port for img_ROM
          clk              : in  std_logic;
          v_count          : in std_logic_vector(10 downto 0);
          h_count          : in std_logic_vector(10 downto 0);
          TopLeftPixel_h: in std_logic_vector(10 downto 0);
          TopLeftPixel_v: in std_logic_vector(10 downto 0);

          --port for img_process
          slct_img         : in  std_logic_vector(1 downto 0);
          slct_Prc_img     : in  std_logic_vector(1 downto 0);
          data_out         : out std_logic_vector(23 downto 0)
    );
end img_process;

architecture Behavioral of img_process is

    component img_rom is
    port(
          clock               : in std_logic;

          line_data      : in std_logic_vector(10 downto 0);
          column_data         : in std_logic_vector(10 downto 0);

          frst_pixel_h   : in std_logic_vector(10 downto 0);
          frst_pixel_v   : in std_logic_vector(10 downto 0);

          data                : out std_logic_vector(23 downto 0)
    );
    end component;

    component img_gen is
    port(
          clock               : in std_logic;

          line_data      : in std_logic_vector(10 downto 0);
          column_data         : in std_logic_vector(10 downto 0);

          frst_pixel_h   : in std_logic_vector(10 downto 0);
          frst_pixel_v   : in std_logic_vector(10 downto 0);

          data                : out std_logic_vector(23 downto 0)
    );
    end component;
```

```
50
     signal red          : unsigned(7 downto 0);
     signal green   : unsigned(7 downto 0);
     signal blue    : unsigned(7 downto 0);

55   signal data_rom     : std_logic_vector(23 downto 0);
     signal data_flag: std_logic_vector(23 downto 0);

     signal gray    : unsigned(7 downto 0);

60
begin
     i1 : img_rom port map (
          clock => clk,

65        line_data => v_count,
          column_data => h_count,

          frst_pixel_h => TopLeftPixel_h,
          frst_pixel_v => TopLeftPixel_v,
70
          data => data_rom
     );

     i2 : img_gen port map (
75        clock => clk,

          line_data => v_count,
          column_data => h_count,

80        frst_pixel_h => TopLeftPixel_h,
          frst_pixel_v => TopLeftPixel_v,

          data => data_flag
     );
85
     process (data_rom, data_flag, red, green, blue, slct_img, slct_Prc_img,clk)
   begin
          if (rising_edge(clk)) then
          if (slct_img = "00") then
90                red <= unsigned(data_flag(23 downto 16));
                  green <= unsigned(data_flag(15 downto 8));
                  blue <= unsigned(data_flag(7 downto 0));
                  gray <= (unsigned(data_flag(23 downto 16)) srl 2)+(unsigned(data_flag(23 dow
                                (unsigned(data_flag(15 downto 8)) srl 1)+(unsigned(data_flag(
95                              (unsigned(data_flag(7 downto 0)) srl 4)+(unsigned(data_flag(7


               elsif (slct_img = "01") then
                    red <= unsigned(data_rom(23 downto 16));
100                 green <= unsigned(data_rom(15 downto 8));
                    blue <= unsigned(data_rom(7 downto 0));
```

```
                    gray <= (unsigned(data_rom(23 downto 16)) srl 2)+(unsigned(data_rom(23 downt
                                (unsigned(data_rom(15 downto 8)) srl 1)+(unsigned(data_rom(15
                                (unsigned(data_rom(7 downto 0)) srl 4)+(unsigned(data_rom(7 d


                end if;

                if (slct_Prc_img = "00") then -- no image processing
                    data_out(23 downto 16) <= std_logic_vector(red);
                    data_out(15 downto 8) <= std_logic_vector(green);
                    data_out(7 downto 0) <= std_logic_vector(blue);

                elsif (slct_Prc_img = "01") then   -- grayscale
                    data_out(23 downto 16) <= std_logic_vector(gray);
                    data_out(15 downto 8) <= std_logic_vector(gray);
                    data_out(7 downto 0) <= std_logic_vector(gray);

                elsif (slct_Prc_img = "10") then   -- binary image
                    if (gray > 235) then
                        data_out <= (others => '1');
                    else
                        data_out <= (others => '0');
                    end if;

                elsif (slct_Prc_img = "11") then   -- invert color
                    data_out(23 downto 16) <= std_logic_vector(255-red);
                    data_out(15 downto 8) <= std_logic_vector(255-green);
                    data_out(7 downto 0) <= std_logic_vector(255-blue);

                end if;
            end if;
        end process;
end Behavioral;
```

### 6.2.4 Image generator component in VHDL

This following VHDL Code is used to synthetise our Image generator component.

Listing 8: Image generator component

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


entity img_gen is
   port(
      clock          : in std_logic;

      line_data      : in std_logic_vector(10 downto 0);
      column_data    : in std_logic_vector(10 downto 0);

      frst_pixel_h   : in std_logic_vector(10 downto 0);
      frst_pixel_v   : in std_logic_vector(10 downto 0);

      data           : out std_logic_vector(23 downto 0)
          );
end img_gen;

architecture Behavioral of img_gen is

signal tx: std_logic_vector(23 downto 0);

begin
   -- addr register to infer block RAM
   process (clock)
   begin
      if (rising_edge(clock)) then
         if (unsigned(column_data) <= X"50") then
            tx <= X"FFFFFF";

         elsif (X"50" < unsigned(column_data) and unsigned(column_data) <= X"A0") then
            tx <= X"FF0000";

         elsif (X"A0" < unsigned(column_data) and unsigned(column_data) <= X"F0") then
            tx <= X"FF8000";

               elsif (X"F0" < unsigned(column_data) and unsigned(column_data) <= X"140") then
            tx <= X"FFFF00";

         elsif (X"140" < unsigned(column_data) and unsigned(column_data) <= X"190") then
            tx <= X"00FF00";

         elsif (X"190" < unsigned(column_data) and unsigned(column_data) <= X"1E0") then
            tx <= X"0080FF";

               elsif (X"1E0" < unsigned(column_data) and unsigned(column_data) <= X"230") then
            tx <= X"FF00FF";
```

```
50          elsif (X"230" < unsigned(column_data) and unsigned(column_data) <= X"280") then
               tx <= X"FF007F";

            else
               tx <= (others=>'0');
55
            end if;
         end if;
      end process;

60    data <= tx;
end Behavioral;
```

### 6.2.5 Image ROM component in VHDL

This following VHDL Code is used to synthetise our Image ROM component.

Listing 9: Image ROM component

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


entity img_rom is
   port(
      clock          : in std_logic;

      line_data      : in std_logic_vector(10 downto 0);
      column_data    : in std_logic_vector(10 downto 0);

      frst_pixel_h   : in std_logic_vector(10 downto 0);
      frst_pixel_v   : in std_logic_vector(10 downto 0);

      data           : out std_logic_vector(23 downto 0)
          );
end img_rom;

architecture Behavioral of img_rom is
   signal tx: std_logic_vector(23 downto 0);
   type rom_type is array(0 to 159, 0 to 329) of std_logic_vector(23 downto 0);

   -- ROM definition
constant ROM: rom_type :=(
     (X"FEFFFF",X"D7D6D2",X"FFC1AD",X"FFACBF",X"FFA2B0",....),
     (X"FEFFFF",X"858480",X"4D0000",X"7D081B",X"8C000C",....),
     ......
     ......
     -- All pixel are not written here
     ......
     ......
     (X"FFF3FF",X"F7EFED",X"F5F5DD",X"FBFEDF",X"F4F2E5",....),
     (X"FFF9FF",X"FFFDFB",X"FEFEE6",X"FDFFE1",X"FFFEF1",....)
);
begin
   -- addr register to infer block RAM
   process (clock)
   begin
      if (rising_edge(clock)) then
        if ((unsigned(frst_pixel_v)-1 < unsigned(line_data)) and (unsigned(line_data) < X"A0"+un
            and (unsigned(frst_pixel_h)-1 < unsigned(column_data)) and (unsigned(column_data) <
          tx <= ROM(to_integer(unsigned(line_data)-unsigned(frst_pixel_v)), to_integer(unsigned(

        else
          tx <= (others=>'1');
        end if;
      end if;
   end process;
```

```
50
    data <= tx;
end Behavioral;
```