Ecole Polytechnique Fédérale de Lausanne

**Embedded Systems**

# Lab 4.0 Camera & LCD implementation

*Authors:*
**RUIZ DE VELASCO Pablo 261406**
**PEETERS Thomas 288239**

February 2, 2022

# Contents

# 1 Introduction

The aim of this lab is to implement the detailed design proposed in lab 3.0. Our group focused on the design of a camera controller that transfers the frames captured with the TRRB-D5M camera to the memory of our DEO-Nano-Soc board.

The camera controller will read the valid data pixels coming from the camera and will convert the 48-bit data pixels (12-12-12-12,G1-R-B-G2) to 16-bit RGB pixels (5-6-5,R-G-B), before writing them in the DC FIFO. Once there are enough pixels in the DC FIFO, the DMA will start a 16-word burst transfer to write those pixels in the memory.

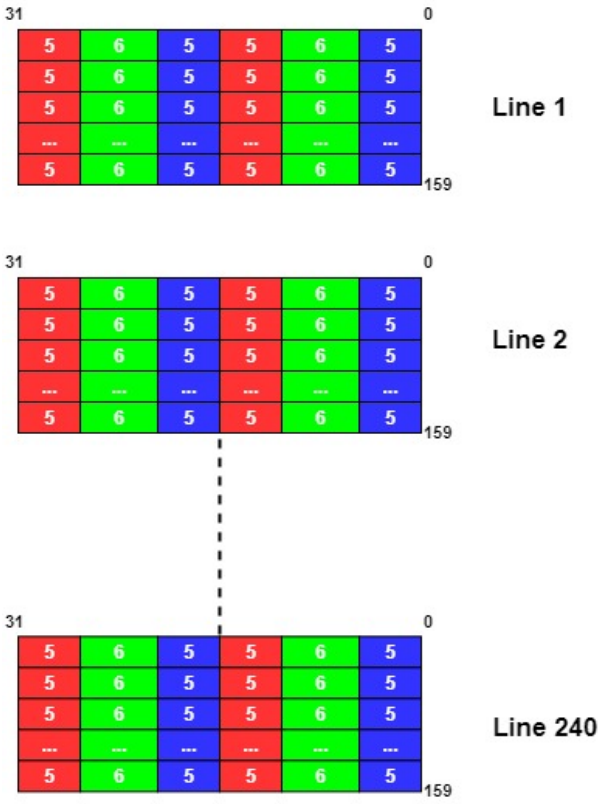


Figure 1: Organisation of the memory

# 2 Camera Controller Architecture

Compared to lab 3, the architecture of our camera controller is nearly the same :

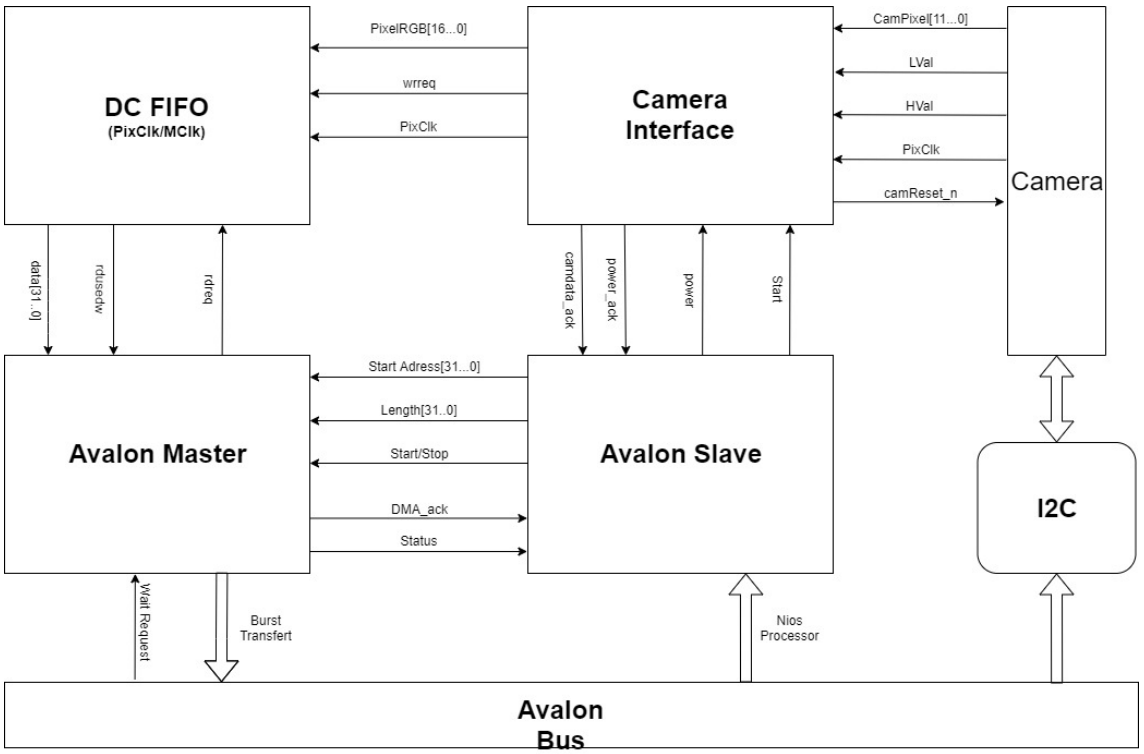


Figure 2: Camera controller architecture

Compared to lab 3, four signals were added :

- status_pw : this signal is sent from the camera interface to the Avalon slave and gives information on the power status of the camera : powered on ("01"), powered off ("00") or powering ("10").
- camdata_ack : the last 19 bits give information on the number of pixels read from the camera and the 3 higher bits give information on the state of the camera interface (idle, wait_frame, readRG1_line, readBG2_line or wait_endframe).
- DMA_ack : gives information on the state of our DMA (idle,WaitData, WriteData) and on the number of burst transfer.
- camReset_n : used to reset the camera (active low).

These three signals are not necessary to make our system work but they were very useful during the test of our system to understand where our system was not working correctly.

# 3 Simulation of our different components using testbenches

## 3.1 Simulation on our DMA

Here is the initial FSM of the DMA that we implemented in lab 3.0 :

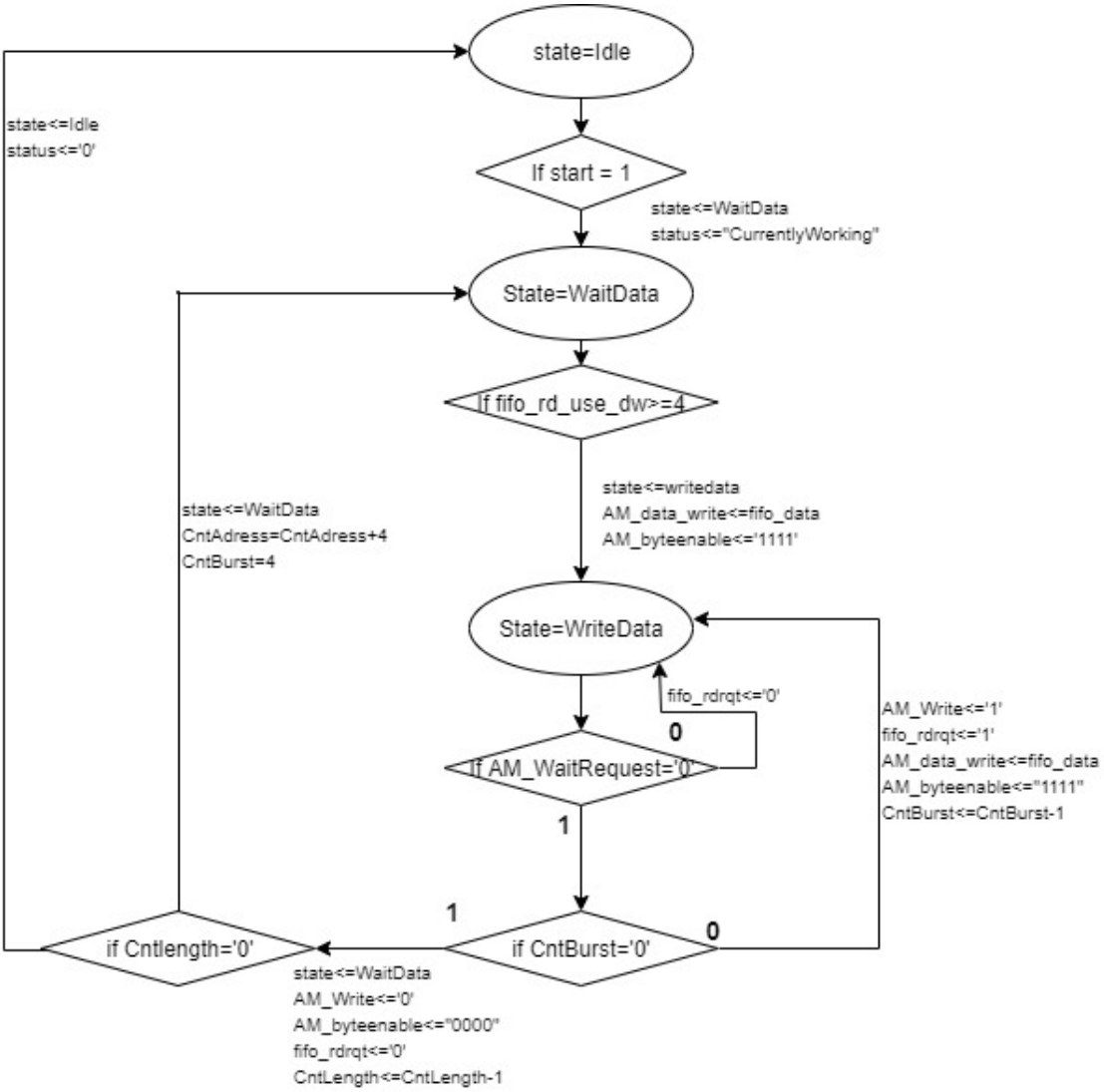


Figure 3: FSM of the DMA implemented in lab 3.0

Making some tests of this DMA with a testbench we realized that we had a problem with the fifo_rdreq signal :

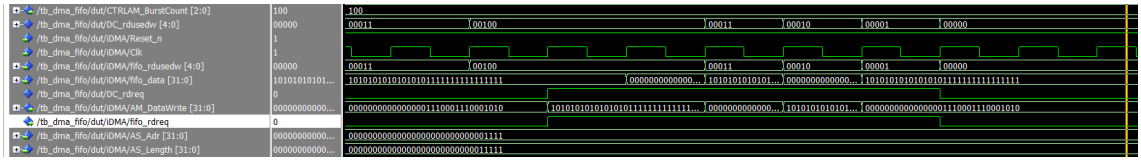

Figure 4: Testbench of our initial DMA

We realized that when our rdreq was going from 0 to 1, we need 2 clock cycles before getting the new data in AM_DataWrite and therefore, we were writing twice the same pixel. This problem appears when we start a new burst transfer or after a wait request (fifo_rdreq goes from 0 to 1). Moreover we observe that we have a transfer of 5 data instead of 4.
We made different modifications to this DMA to correct these errors. Here is the new FSM of our DMA:

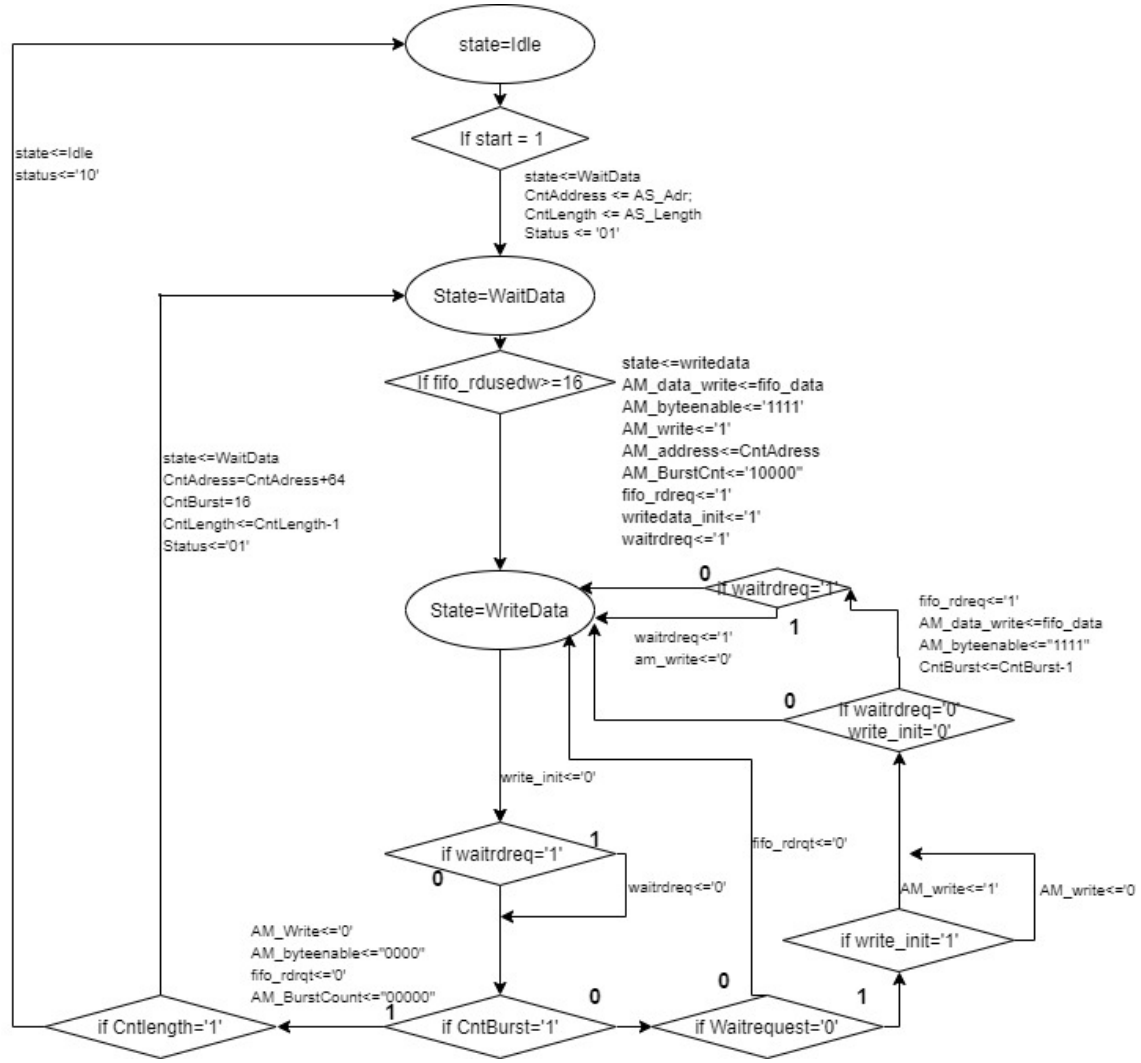


Figure 5: FSM of our final DMA

In this new version of the DMA, we use two signals 'waitrdreq' and 'write_data_init' that will allow us to deal with the problems mentioned earlier. write_data_init will make the signal AM_write go from 1 to 0 after one clock cycle at the beginning of the burst transfer (so that we don't send twice the same pixel at the beginning of the transfer). The signal 'waitrdreq' is used to make the signal AM_write go from 1 to 0 after a wait request so that we don't send twice the same pixel after a wait request.
Here is a simulation made with the new DMA :

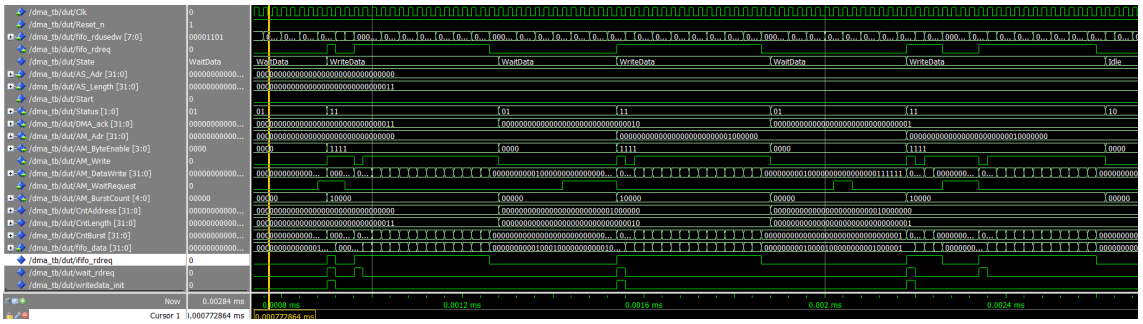

Figure 6: Testbench with the new DMA

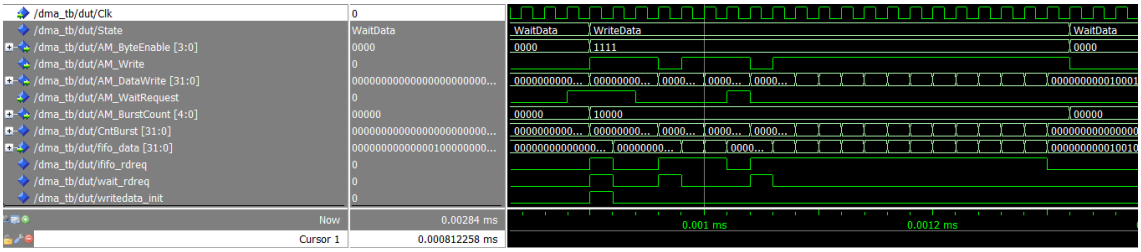


Figure 7: Testbench with the new DMA

Looking at this new testbench we observe that the DMA is working correctly. At every burst transfer, we have 16 words transferred. We observe that after the beginning of the transfer or after a wait request, AM_write goes to 0 during one clock cycle so that the DC FIFO has the time to send the new data to the signal AM_data_write.

We also decided to change the value for the burst count (16 instead of 4). Increasing this value will allow us to do less burst transfers but will make the bus busy with the transfers for a longer time. It didn't cause any problem on our system.

## 3.2 Simulation on our Avalon Slave

Some small changes have been added to the Avalon slave. First, since we added three signals : status_pw, camdata_ack and DMA_ack, we added three internal signals to be able to read them. Here is the new Avalon slave register map :

| Address | Write register | Writedata[31..0] | Read register | Read register |
|---|---|---|---|---|
| 0 | address | =>iRegAdress | address | iRegAddress=> |
| 1 | length | =>iRegLength | length | iRegLength=> |
| 2 | start | =>iRegStart | start | iRegStart=> |
| 3 | cmd | =>iRegCmd | cmd | iRegCmd=> |
| 4 | - | Don't care | status | iRegstatus=> |
| 5 | - | Don't care | status_pw | iRegstatus_pw=> |
| 6 | - | Don't care | camdataack | icamdataack=> |
| 7 | - | Don't care | DMA_ack | iRegDMA_ack=> |

Figure 8: Avalon slave register map

Moreover, we added a process in the Avalon slave to make our signal iRegStart go back to 0 so that once a frame is caught the DMA and the camera interface stay in the idle state until the start signal is set to '1' again. Here is a testbench of the Avalon slave :

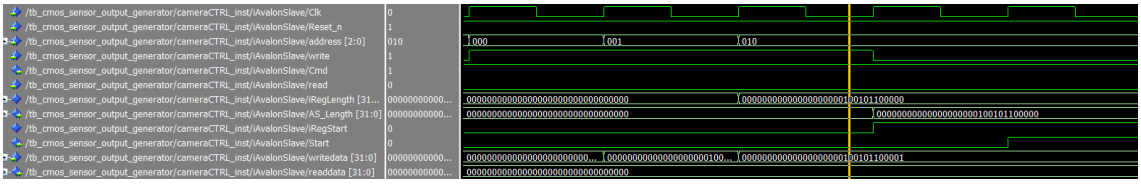


Figure 9: Testbench of the Avalon slave

## 3.3 Simulation on the camera interface

We also had to modify the FSM for the camera interface, mainly because of two reasons: the first one is that the power part of the FSM runs on the 50MHz clock coming from the FPGA while the acquisition part needs to run on the pixel clock (PIXCLK) coming from the camera and both clocks are not necessarily the same. Thus, we implemented two separate FSMs. The second reason is that we were

doing the transition to a state where we acquire pixels with the condition LVAL=1. However, by doing so, we realized that we were missing the very first pixel of each line, which is why the states "wait for next line" have been removed. Another change we made was to have a pixel counter rather than a line counter because it allowed us to have a better control on the acquisition.

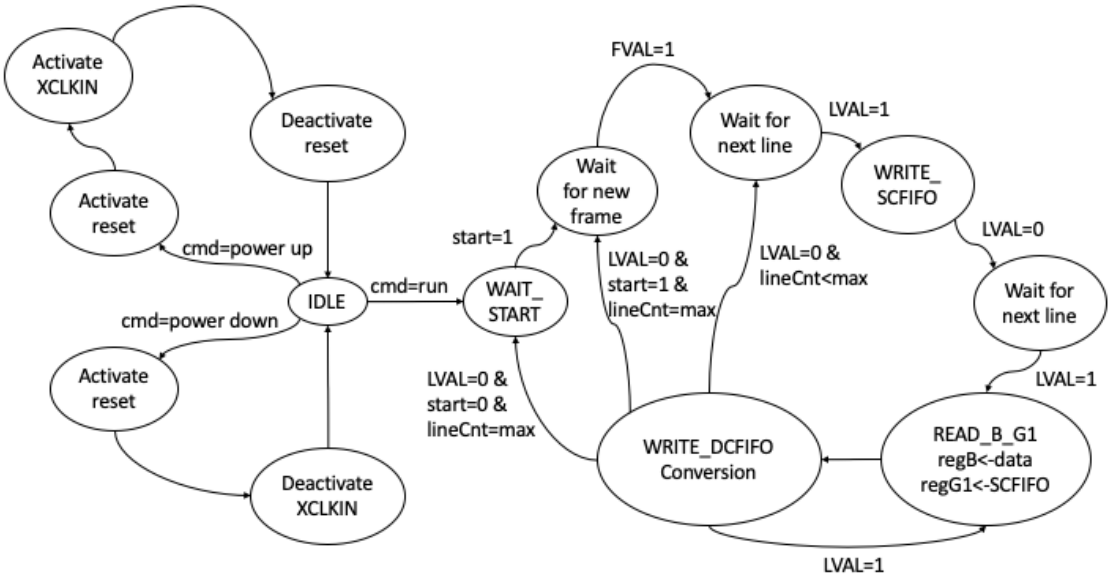


Figure 10: Old FSM from lab 3

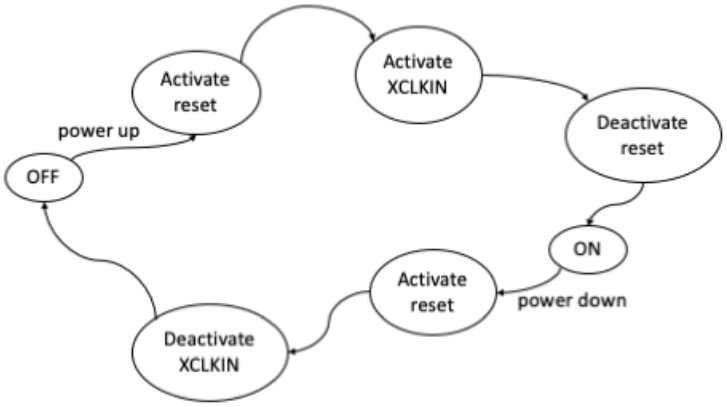


Figure 11: Power FSM

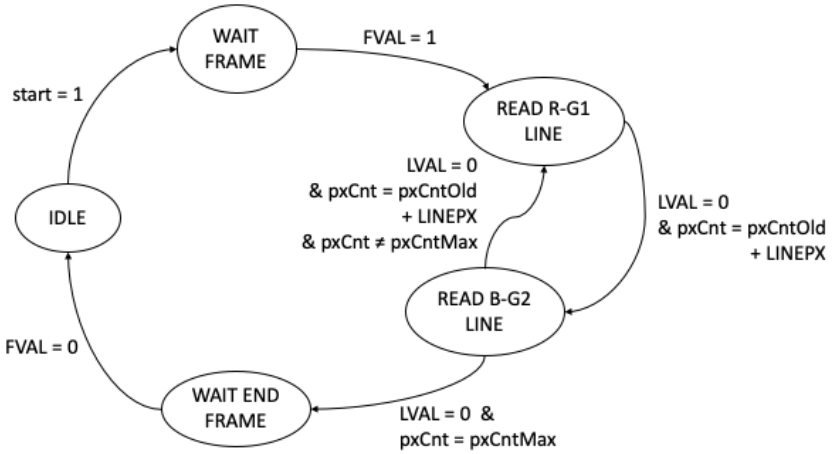


Figure 12: Acquisition FSM

The debayerization, i.e. the construction of the final RGB pixel, is done in the most straightforward way: for the first RGB pixel, we use the two first Bayer pixels of the first R-G1 and B-G2 lines, and we continue with the following Bayer pixels of the two lines for the next RGB pixel. Since we have two components for the green color, they are both averaged.

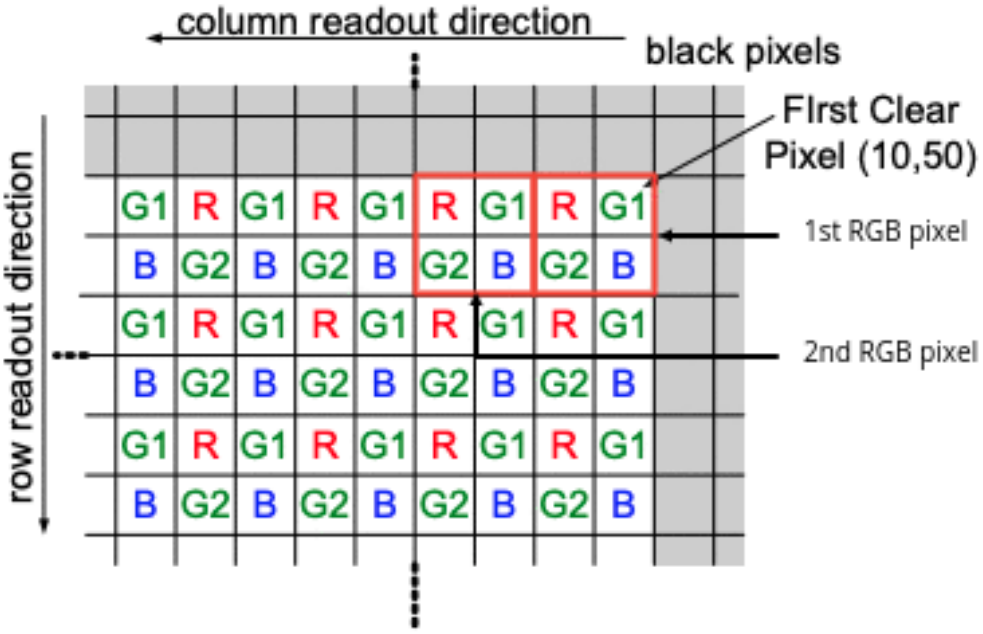


Figure 13: Pixel debayerization

We tested the camera interface together with the SC FIFO by writing a testbench on Modelsim. To simplify the simulation, we set the number of pixels per line to 20 and the number of lines per frame to 4. The data was generated by gradually incrementing the value of D (Bayer pixels).

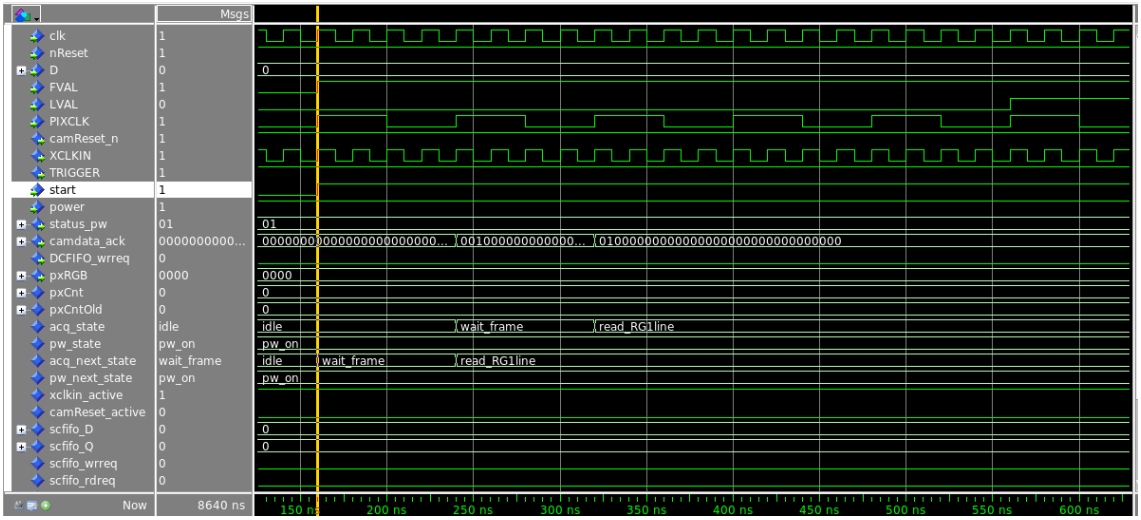


Figure 14: Camera interface simulation: start of the acquisition

Figure 13. shows the beginning of the acquisition, caused by a high level on the signal start which places the FSM on the WAIT FRAME state on the next clock cycle. On that same clock cycle, FVAL is already high so on the following clock cycle the state becomes READ R-G1 LINE, where we read the first line of valid pixels and store it in the SC FIFO. At the beginning of a line, the value of the pixel counter is stored in another register so that at the end of the line we can easily check that we have read the correct number of pixels.

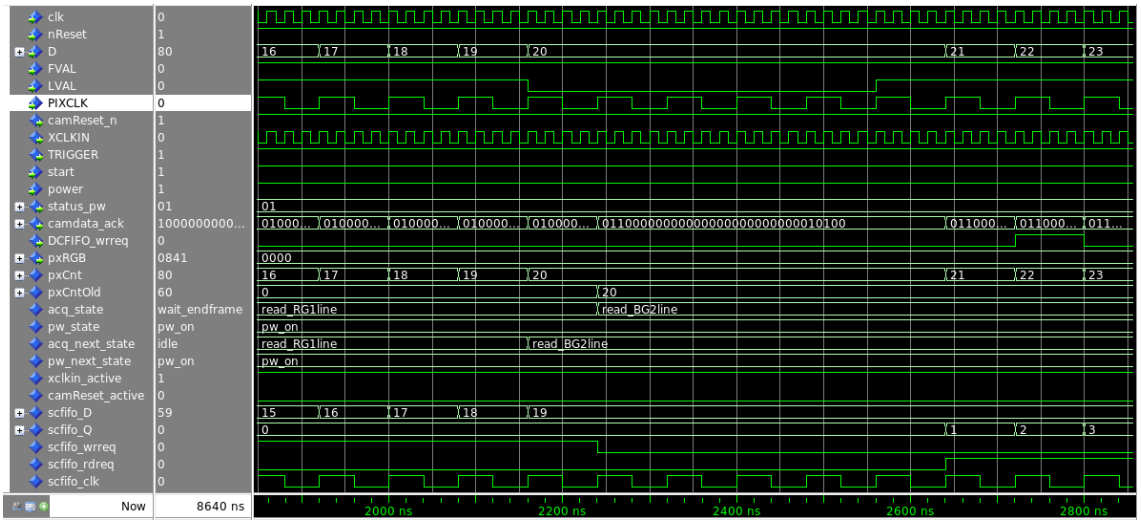


Figure 15: Camera interface simulation: transition between two lines

On figure 14. we can observe the transition between two lines. Once LVAL goes low and the value of the pixel counter is correct, the FSM goes to the READ B-G2 LINE state, where we acquire the second row of valid pixels. While reading the B-G2 line, every two clock cycles the four color components of each pixel are available and we can thus convert them to a 16-bit RGB pixel which will be written in the DC FIFO.

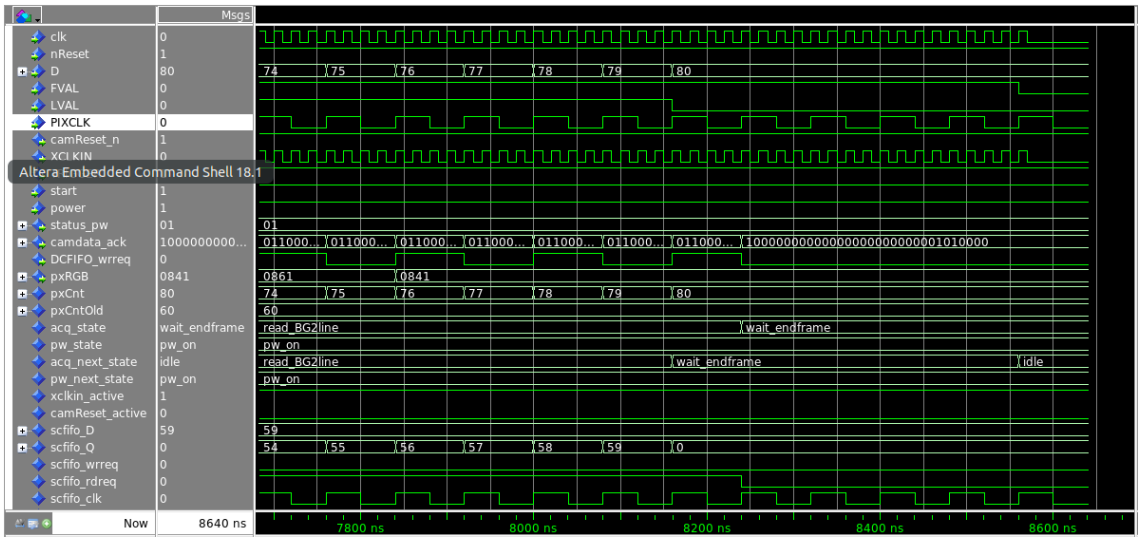


Figure 16: Camera interface simulation: end of the acquisition

Figure 15. shows the end of the acquisition of the frame. First LVAL goes low and the pixel counter has reached the total number of pixels in a frame so the FSM goes to the WAIT END FRAME state. After that, the FSM returns to the IDLE mode once the FVAL signal goes low.

## 3.4 Testbench of the camera controller

To test the entire system, we used the cmos_sensor_output_generator that exports a conduit interface containing the frame_valid, line_valid, and data signals which a camera would have outputted :

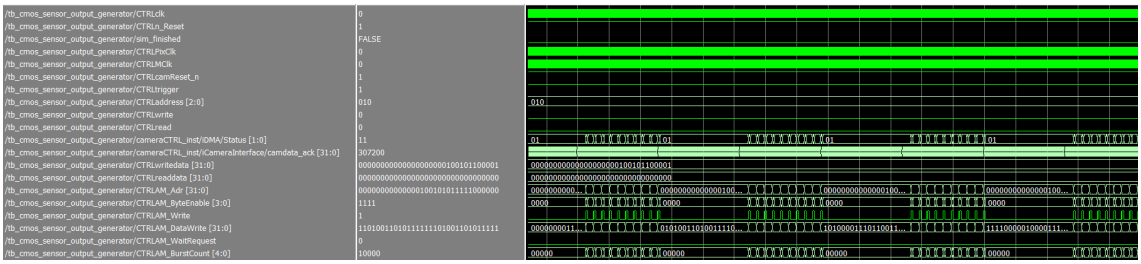

Figure 17: Testbench of the cameraCTRL using the cmos_sensor_output_generator

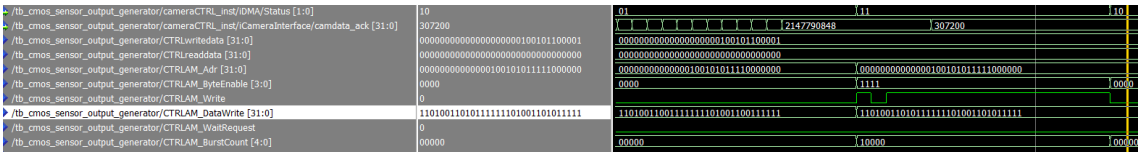

Figure 18: Testbench of the camera controller using the cmos_sensor_output_generator

We see on the figure above that our camera controller has correctly caught 307200 pixels from the cmos_sensor_output_generator (camdata_ack=307200). Moreover, we observe that the Status='10' which means that our camera controller is back in Idle mode and is waiting for a new start signal to catch a new frame.

# 4 Implementation of the camera controller on Quartus

Our camera controller is working correctly on the testbench, so we implemented our system on Quartus to make it work on the DEO-Nano-Soc board.
Here is Soc-System :

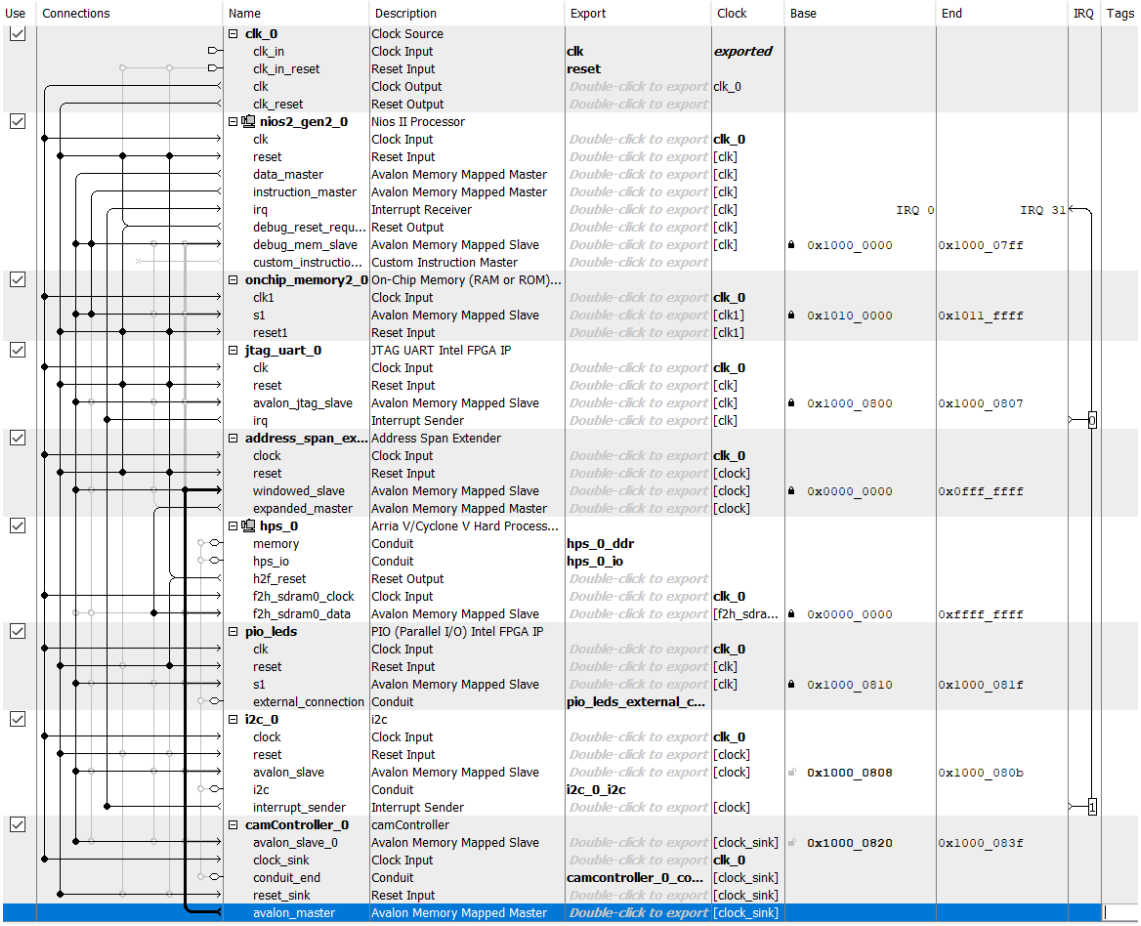


Figure 19: QSYS SYSTEM

The I2C interface control is used to configure the internal registers of the TRDB-D5M camera. We connect the Avalon master of the camera controller to the window slave of the address span extender. The address span extender is used to access a specific region of the the DDR3 memory where we have reserved 256 MB of continuous memory.
In the top-level VHDL file, "Deo_Nano_Soc_TRDB_D5M_LT24_top_level.vhd", we connected the different signals with the correct GPIO1 pins :

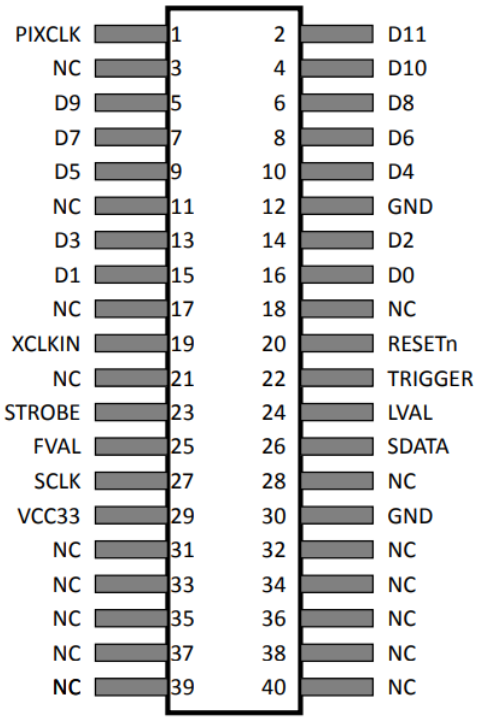


Figure 20: Camera registers and their configuration

```
i2c_0_i2c_scl                                => GPIO_1_D5M_SCLK,
i2c_0_i2c_sda                                => GPIO_1_D5M_SDATA,
camcontroller_0_conduit_end_export_data      => GPIO_1_D5M_D,
camcontroller_0_conduit_end_export_fval      => GPIO_1_D5M_FVAL,
camcontroller_0_conduit_end_export_lval      => GPIO_1_D5M_LVAL,
camcontroller_0_conduit_end_export_xclkin    => GPIO_1_D5M_XCLKIN,
camcontroller_0_conduit_end_export_pixclk    => GPIO_1_D5M_PIXCLK,
camcontroller_0_conduit_end_export_camresetn => GPIO_1_D5M_RESET_N,
camcontroller_0_conduit_end_export_trigger   => GPIO_1_D5M_TRIGGER
```

Figure 21: Portion of the code of the file :"Deo_Nano_Soc_TRDB_D5M_LT24_top_level.vhd"

# 5 Software

Once our system has been successfully compiled on Quartus, we wrote a C program that can be run on the Nios II processor. In this program, we need to configure the internal camera registers using the I2C interface and write in the slave component of our camera controller to make it work.

## 5.1 I2c Interface

Here is a table with the register that we need to configure:

| Register name | Configuration |
|---|---|
| R0x003 : row size | R0x003 = 1919 |
| R0x004 : column size | R0x004 = 2559 |
| R0x01E : read mode 1 | R0x01E = 0x0100 |
| R0x022 : row address mode | R0x022 = 0x0033 |
| R0x023 : column address mode | R0x023 = 0x0033 |
| R0x00B : restart | R0x00B = 0x0003 |
| R0x00A: pixel clock control | R0x00A = 0x8000 |
| R0x00B : Restart | R0x00B=0x0001 |
| | |

Figure 22: Camera registers and their configuration

- This configuration for row size, column size, row address mode and column address mode will make the resolution of the image output by the camera equal to 640*480 Bayer pixels and the camera will use binning as sub-sampling method. After debayerization, the final resolution will thus be of 320*240, which corresponds to the resolution of the LCD.
- Setting read mode 1 equal to 0x0100 will make the camera enter in snapshot mode. This will make the camera wait for the next trigger to capture a frame instead of continuously outputting frames.
- Writing 3 to the register restart will make the camera pause at row 0. Writing 1 to the register signal will make the camera resume.
- The pixel clock control register is used to invert the pixel clock so that our data can be captured on the rising edge of Pixclk. After trying without success to divide the external clock provided to the camera in order to have a pixel clock running at a slower frequency, we decided to keep the same frequency for the pixel clock and the external clock, which is the same as the clock provided by the FPGA: 50 MHz.

Now that our camera is configured, we can set the trigger bit of the 'restart' register to make the next trigger occur (we are in snapshot mode) and catch the next frame. Since the trigger bit is not reset automatically by the camera, we immediately set it to 0 after triggering a frame so that the once the camera finishes sending the data, it will wait for the next trigger.

| Register name | Configuration |
|---|---|
| R0x00B : Restart | R0x00B=0x0004 |
| R0x00B : Restart | R0x00B=0x0000 |

Figure 23: Configuration to trigger a frame

## 5.2 Configuration of the Avalon slave

| Register Name | Configuration |
|---|---|
| iRegadress | HPS_0_BRIDGES_BASE |
| | HPS_0_BRIDGES_BASE+offset |
| iReglength | iRegLength=2400 |
| iRegCmd | iRegCmd=1 |
| iRegStart | iRegStart=1 |

Figure 24: Avalon slave registers and their configuration

- We must write in the Avalon slave the initial address in the memory where we want to write the image. Since we use the address span extender, the initial address for the first frame will be equal to the constant: HPS_0_BRIDGES_BASE. We have to different buffers in memory to write different images, so we can add an offset to the initial address to write a new image at an other place in the memory. For one frame, we write 38400 words and therefore, the minimal value for the offset is 38400*4 = 156000.

- The size of the image will not change so the value of iRegLength is equal to 2400 (number_pixel_frame /(2*16)) : the number of burst transfers necessary to write the frame in the memory.
- iRegCmd is set to 1 to power on the camera
- Start is set to 1 to make our camera controller go out of the state idle . We must set start to 1 every time we want to catch a new frame.

### 5.3 Transfer the image in our host PC

To check if the image captured is correct, we have to transfer it from the board's memory to our PC. To do that we enable the altera_hostfs package and we use the function fprintf to write pixels value in a file. We decided to use the PPM format that allows us to get an image with colour and has the advantage of being relatively easy to write. This file is composed of a header with the characteristics of the image and then the value of each pixel in RGB format.
Here is an example of one image written in PPM format :

```
P3
320 240
63
7 7 3   5 7 3   7 7 3   7 7 3   7 7 3   7 7 3   7 7 3   7 7 3   7 7 3   7 7 3   7 7 3   7 7 3   7 8 3   7
8 9 3   7 8 3   7 7 3   7 8 3   7 8 3   7 9 3   9 11 5   10 13 6   13 15 7   15 15 7   15 15 7   15 14 7   13 12
  5 7 3   7 7 3   7 7 3   7 7 3   7 7 3   7 7 3   7 7 3   7 7 3   7 7 3   7 7 3   7 8 3   7 9 4   7 9 4
7 7 3   7 8 3   7 7 3   7 8 3   7 7 3   7 8 3   7 8 3   8 9 3   8 8 3   8 9 3   8 8 3   9 9 3   9 9 3   8 9 3   9
3   5 7 3   5 7 3   5 6 3   5 5 3   5 7 3   5 7 3   5 8 4   7 11 5   9 15 7   13 19 9   17 21 11   19 20 10   18
1 14 7   11 13 6   9 11 5   7 9 3   6 7 3   5 6 3   5 7 3   7 7 3   7 7 3   7 7 3   7 7 3   7 7 3   7 8 3   7 8 3
8 8 3   7 9 3   8 9 3   9 9 3   8 9 3   9 9 3   8 9 3   8 9 3   9 9 3   9 9 3   9 9 3   9 9 3   9 9 3   9 9 3   9
7 3   5 8 4   6 7 3   5 6 3   4 5 3   5 7 3   6 9 4   7 9 5   8 12 6   11 17 9   15 20 10   18 21 11   19 22 11
  7 13 5   9 14 6   11 15 7   11 15 7   11 15 7   11 13 5   9 11 4   7 8 3   6 7 3   6 7 3   7 7 3   7 7 3   7 7
8 8 3   7 9 3   8 9 3   8 9 3   8 8 3   8 8 3   8 8 3   8 9 3   9 9 3   8 9 3   8 9 3   8 9 3   8 8 3   7 8 3   7
8 4   7 9 5   7 9 4   6 7 3   5 6 3   5 7 3   6 9 5   7 11 5   9 13 6   11 16 8   14 19 9   17 19 9   17 19 9   1
11 4   7 11 4   7 11 4   7 11 4   7 9 3   6 7 3   5 7 3   5 7 3   7 7 3   7 7 3   7 7 3   7 7 3   7 7 3   8 9 3
7 7 3   6 7 3   7 7 3   7 7 3   7 7 3   6 7 3   6 7 3   7 7 3   7 7 3   7 7 3   7 7 3   6 7 3   6 6 3   5 6 3   5
5   7 9 4   7 7 3   5 5 3   4 8 3   7 9 5   8 11 5   9 13 7   11 16 8   15 19 9   16 17 9   15 17 9   15 21 11
7 3   5 7 3   5 7 3   6 7 3   7 7 3   7 8 3   7 7 3   7 8 3   7 8 3   8 9 3   9 9 3   9 9 3   9 8 3   8 8 3   7 8
5 5 3   4 5 3   5 5 3   5 5 3   5 5 3   5 5 3   5 5 3   5 5 3   5 6 3   5 6 3   5 5 3   4 5 3   3 5 3   3 5 3   3
5 3   5 7 3   7 10 5   8 11 5   9 13 7   11 17 9   15 20 11   18 20 11   18 21 11   18 23 11   20 21 11   19 16 8
4   6 8 3   6 7 3   6 7 3   7 7 3   8 8 3   8 7 3   7 8 3   8 8 3   9 9 3   9 9 3   9 9 3   9 8 3   7 8 3   7 9 3
5 5 3   3 5 3   5 5 3   5 6 3   5 5 3   5 5 3   5 5 3   5 6 3   5 6 3   5 5 3   5 5 3   3 5 3   3 5 3   3 5 3   3
13 7   11 17 9   16 22 11   19 23 13   21 23 13   21 24 13   21 22 11   19 17 8   13 9 5   7 5 3   4 3 3   3 4 3
12 5   8 9 4   7 7 3   7 7 3   7 7 3   8 8 3   7 8 3   8 8 3   8 9 3   9 9 4   9 9 3   9 9 3   9 8 3   7 8 3   7
8 8 3   7 9 3   7 9 3   9 10 4   9 9 4   9 9 3   7 7 3   6 7 3   5 6 3   5 6 3   5 5 3   4 5 3   3 5 3   3 5 3
1   19 23 12   21 24 13   21 24 13   21 21 11   18 16 8   13 10 5   7 6 3   4 3 3   3 4 3   3 4 3   4 5 3   5 5 3
5   9 9 4   7 7 3   7 7 3   7 7 3   7 8 3   8 7 3   8 8 3   8 9 3   9 9 4   9 9 3   9 9 3   9 8 3   8 8 3   7 8 4
```

Figure 25: Image written in ppm format

## 6 First results

### 6.1 Test on the colours distribution

Here is the images obtained with the initial configuration of the camera :

Figure 26: Picture of a red object

Figure 27: Picture of a closet

Looking at these two images we observe that the background is green and that red objects are green on the image. To confirm these hypotheses, we configure our camera so that it is in test pattern mode (registers 160 to 164). Instead of outputting the real image, the camera will output an image as configured in the test pattern. In our case we configure the test pattern so that the output is a diagonal gradient of a specific colour (red, green or blue).

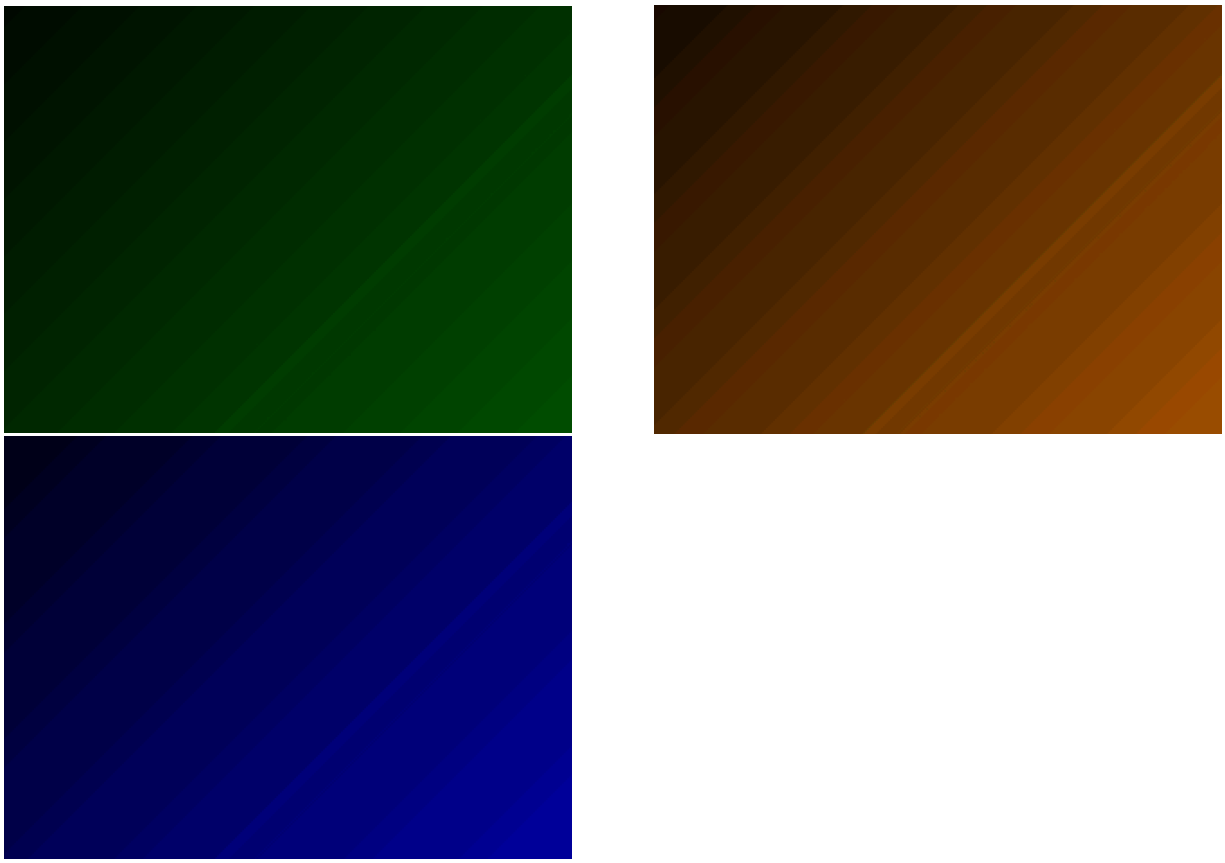

Figure 28: Images received when we asked for a red (top left), green (top right) and blue (bottom left) diagonal gradient

These tests confirm that we have an error in our debayerization. In the figure below, we can see the pattern that we should have for the debayerization (the one on the left) and the pattern that we had (the one on the right). This could be the result of a delay of one clock cycle that we did not consider in our VHDL code but after checking the simulations on Modelsim it did not seem to be the case.


Figure 29: True pixel pattern (left) and our pixel pattern (right) for debayerisation

Correcting that, we made the same test using the diagonal gradient test pattern mode and here are the three images that we obtained :

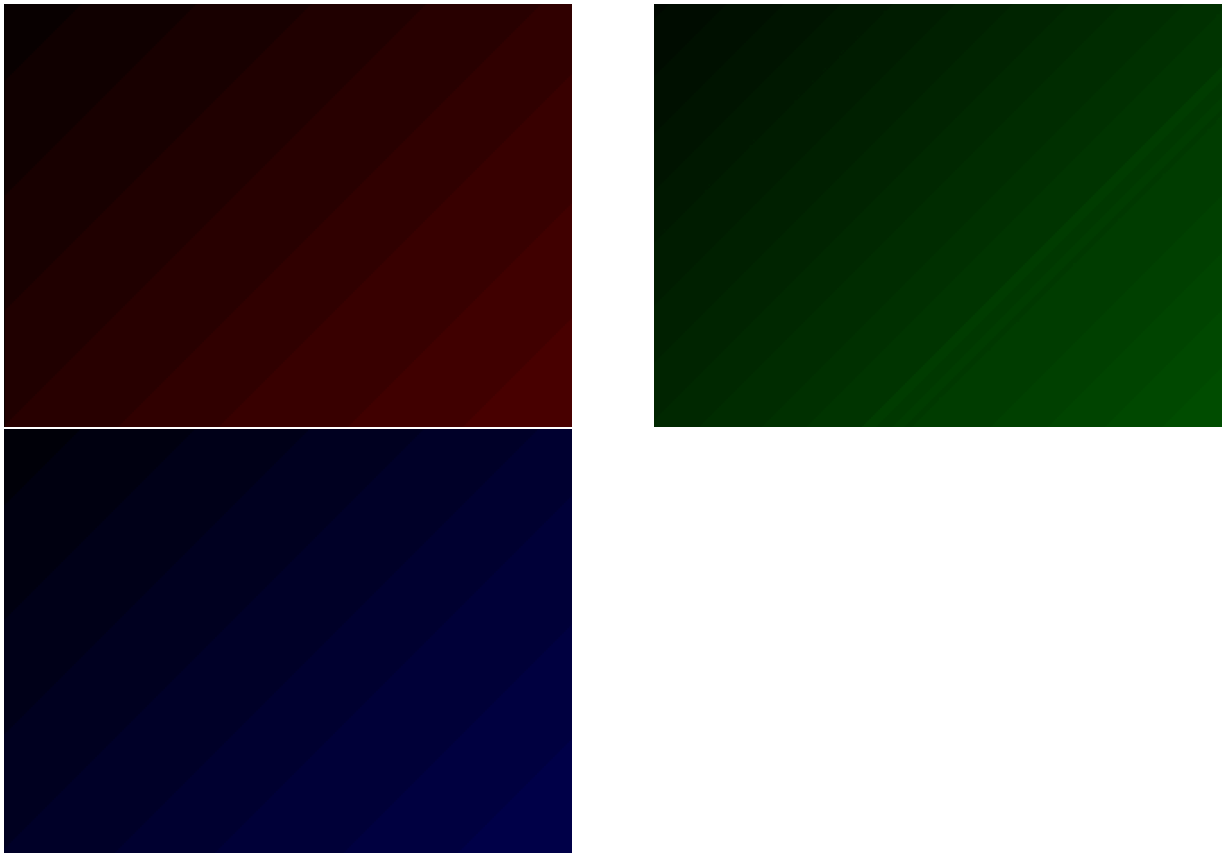

Figure 30: Image obtained when we asked for a red (top left), green (top right) and blue (bottom left) diagonal gradient

With this new configuration, we observe that we get the correct colors in the image.

## 6.2 Test with different configurations for the conversion from 12 to 5/6 bits

Our first choice was to take respectively the 5,5 and 6 most significant bits for the red, blue and green pixels and to make the average between the two green pixels to convert our 48 bits to 16 bits. With this configuration, we realized that the image were dark and that most of the time the value of the two most significant bits were equal to 0.

Figure 31: Picture of the closet using the most significant bits for the 48 to 16 bits conversion

By taking also into account less significant bits we found different images :

Figure 32: Picture of the closet using the 6,7,6 (R-G-B) most significant bits for the 48 to 16 bits conversion

Figure 33: Picture of the closet using the 8,9,8 (R-G-B) most significant bits for the 48 to 16 bits conversion

We observe that if we neglect too much the most significant bits the colour in our images are not correct anymore.
Here is the method we use for the final conversion from 48 bits to 16 bits :

```
-- RED
pxRGB(15) <= camdata_RED(11) or camdata_RED(10);
pxRGB(14 downto 12) <= camdata_RED(9 downto 7);
pxRGB(11)<= camdata_RED(6) or camdata_RED(5);
--GREEN
pxRGB(10) <=  camdata_avgG1G2(11) or camdata_avgG1G2(10);
pxRGB(9 downto 7) <= camdata_avgG1G2(9 downto 7);
pxRGB(6)<=  camdata_avgG1G2(6) or camdata_avgG1G2(5);
pxRGB(5)<=  camdata_avgG1G2(4);

--BLUE
pxRGB(4) <= camdata_BLUE(11) or camdata_BLUE(10);
pxRGB(3 downto 1) <= camdata_BLUE(9 downto 7);
pxRGB(0) <= camdata_BLUE(6) or camdata_BLUE(5);
```

Figure 34: Final method for the conversion from 48 to 16 bits data pixels

# 7 Test with the LCD

For the implementation with the LCD, we decided to use 2 buffers. At the initialisation, the camera controller will write in the first buffer. After that, it will enter in a while loop that will stop when the number of frames asked has been caught. In the while loop, the camera controller will write in the second buffer and the LCD controller will read in the first one. Once the LCD has finished reading the frame from the second buffer, the camera controller will write in the first buffer and the LCD in the second. We return at the beginning of the while loop when the LCD finished reading the frame from the second buffer.

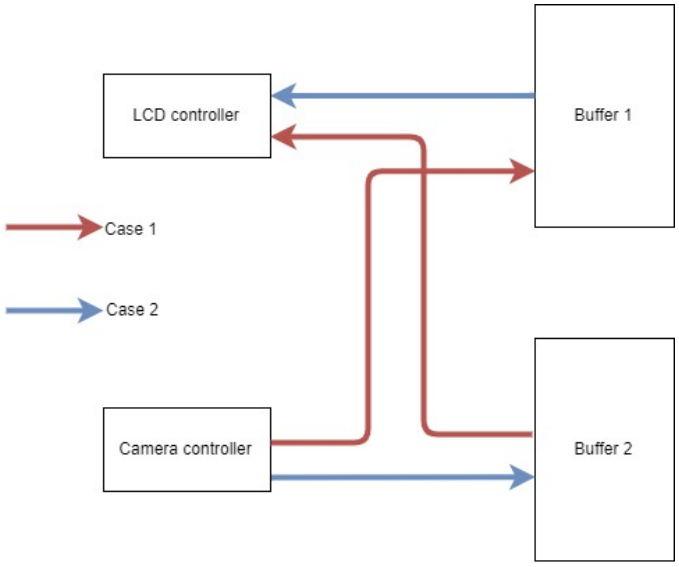


Figure 35: Read and write in buffers

With this implementation we were able to display a video on the LCD.

Figure 36: Image on the LCD during the video

Nevertheless, we realized that we had an other problem in our camera controller : when our camera is pointing at a source of light, our program is blocked and is not able to write the whole frame in memory. By doing several tests, we observed that when we have this problem, the camera pixels read by the camera interface is equal to (307201 instead of 307200) and that the number of burst transfers made by the DMA is lower than 2400 (meaning that it didn't make all the burst transfers). We were not able to find where this problem could come from. We checked if the DC FIFO was full but it was not the case and since the camera is working in the other cases we didn't find the error in the camera interface or in the DMA.

# 8 Conclusion

The aim of this lab was to design a camera controller capable of acquiring and writing in memory frames captured by the TRDB-D5M camera. Our camera is working correctly and because each group have

made a lot of test on their it didn't take a lot of time to assemble the camera controller and the LCD controller on the same board. We realize that most of the time, our camera controller is working well but don't work when it is pointing to a source of light. To improve our system we could add a signal in our camera controller that allows us to skip the actual frame when we are not able to write the all frame in memory. To do that, we should clear the DC fifo and put the camera interface and avalon master in idle state.