Ecole Polytechnique Fédérale de Lausanne

CS-473

Embedded Systems

Lab 4: Camera Implementation
Report

Authors:
Victor Casas - 237770
Hugo Miranda Queiros - 336706
Co-group (LCD):
Francesco Nonis - 300540
Eric Richter - 287433

Supervisors:
Prof. René Beuchat

Due January 5, 2022

Contents

List of Figures

1 Documentation

1.1 Problem statement

In the last lab, we proposed a detailed design of an FPGA-based system which could interface with the TRDB-D5M camera on the DE0-Nano-SoC. In this lab, we implement this design in the Quartus environment. We are satisfied with the design proposed in the last lab but some slight changes have to be made so our system works perfectly. In summary, we add some registers, change the length of some registers and add two state machines for the FIFO to work properly. The changes are detailed in each part.

1.2 High level architecture

The TRDB-D5M is a 5 megapixel digital camera available as an extension card for Terasic FPGAs. It connects to a development board through a 2x20 pin GPIO connector. In Figure 1, one can see how we will connect the pins of the camera on the GPIO_1 connector on the development board (DE0-Nano-SoC).

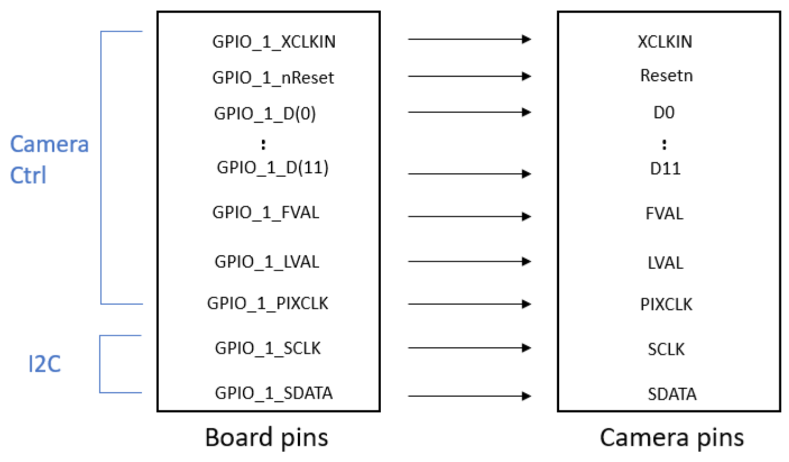


Figure 1: ***Board and camera pins***

A provided I2C controller will also be used to communicate with the camera for configuration purposes. One can also see in Figure 2 how it will communicate with the Camera.

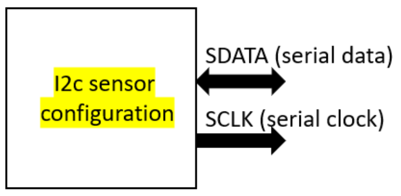


Figure 2: ***I2c controller***

In Figure 3 the high level diagram of our full system with the camera and the LCD is represented.

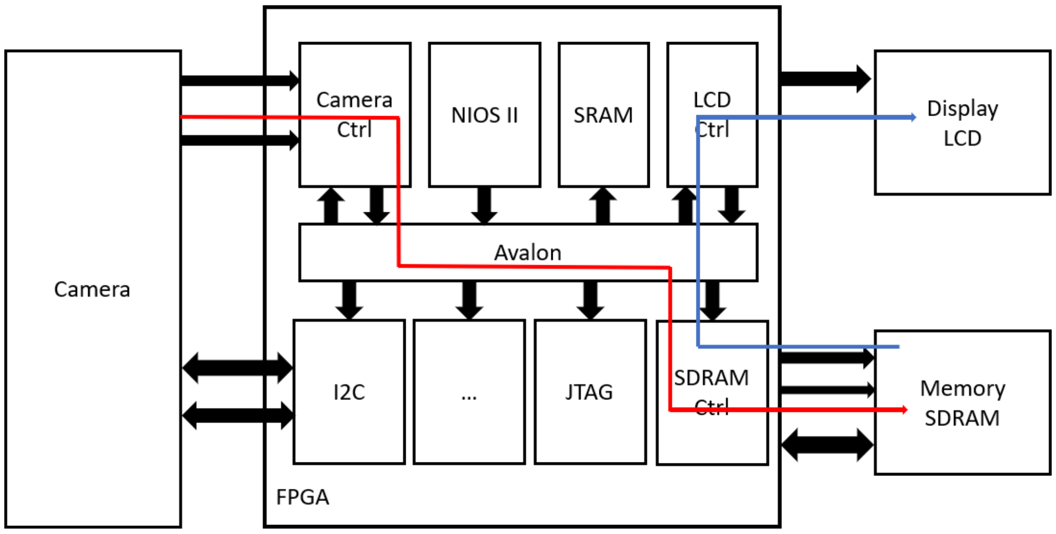


Figure 3: ***High level diagram***

To read the pixels from the Camera, convert them to the format and resolution of the LCD and to store them in the SDRAM memory, we created a custom IP component named Camera_Ctrl (Camera Controller). This component is directly connected to the camera through a conduit and to the Avalon bus through a Master and Slave interface.

Our Camera_Ctrl IP component is composed of the three main blocks in Figure 4 :

- one Avalon Master to send the pixels read from the Camera to the DDR3 memory through the Avalon Bus.
- one Avalon Slave that will be used to configure/read the registers of Camera_Ctrl and to receive the orders from the NIOS CPU to start/stop the storage of new frames or to know how to access memory.
- one Camera Interface with three FIFO :
  - 2 entry FIFO that both receive, one row of color pixel of each pair of rows, and then empty themselves in parallel in a component that will convert the color pixels sent by the Camera into the resolution and the format that is compatible with the LCD.
  - 1 exit FIFO that stores the final pixels that will be sent to the DDR3 memory.

The whole Camera_Ctrl component will be a functional DMA unit to allow the processor to offload tasks of packet data transfer (here the pixels of the camera).

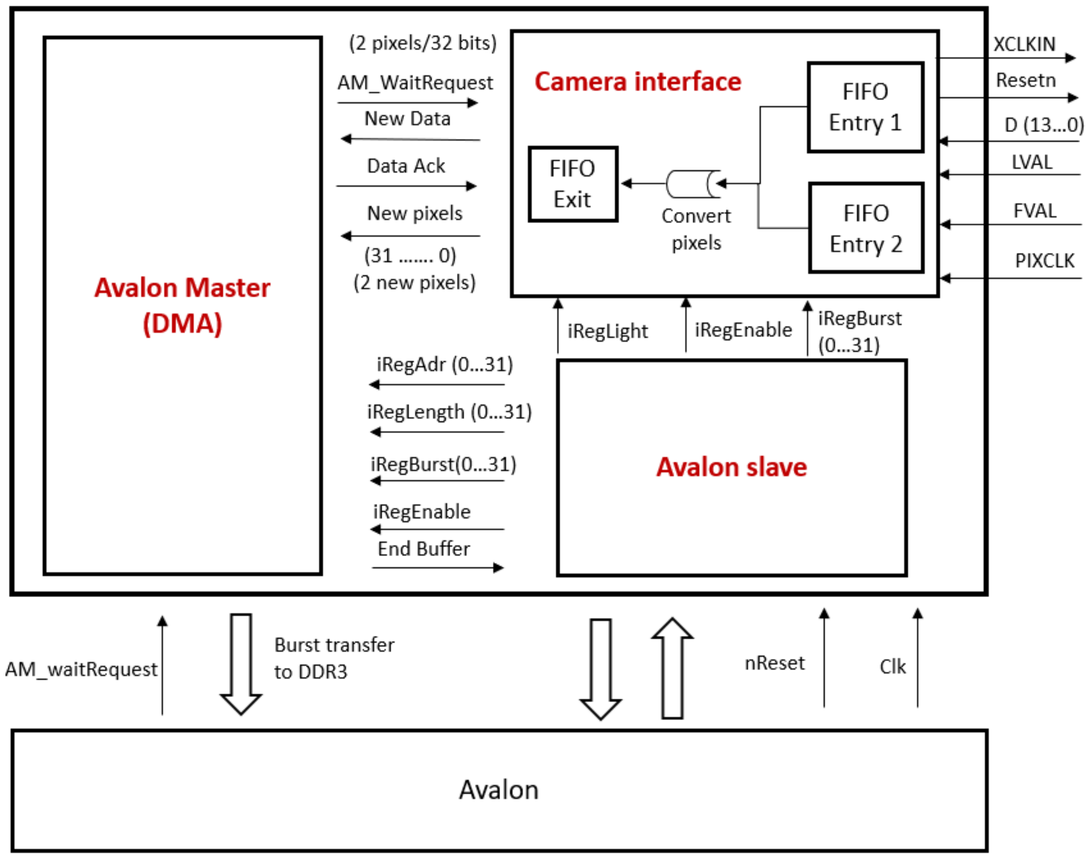


Figure 4: *Camera control*

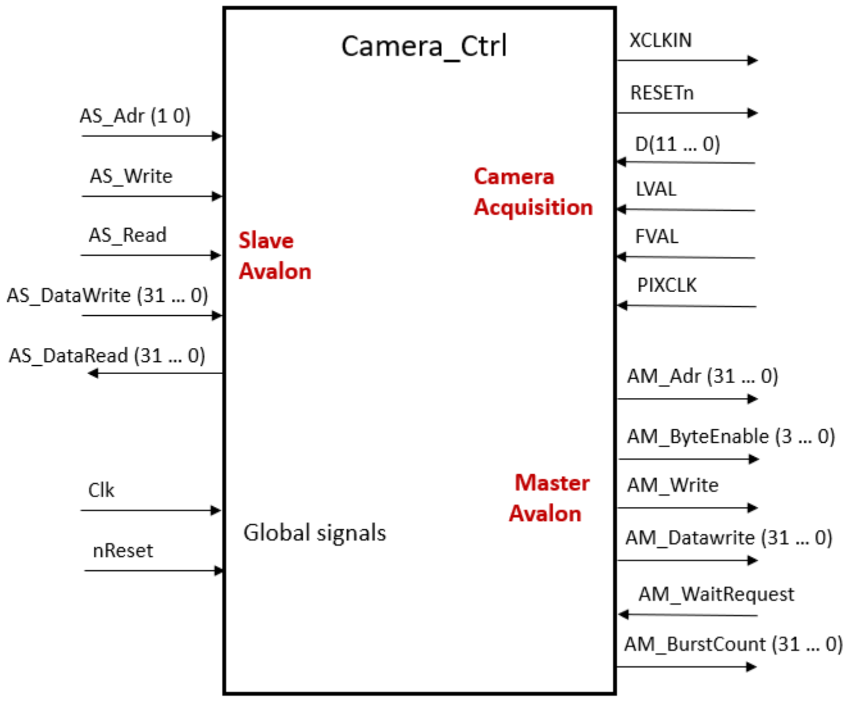


Figure 5: *Camera control IP component*

1.3 Low level architecture

1.3.1 Camera pixels reading and formatting

The TRDB-D5M camera has by default a 2592x1944 active color pixel image as one can see in the Figure 6.

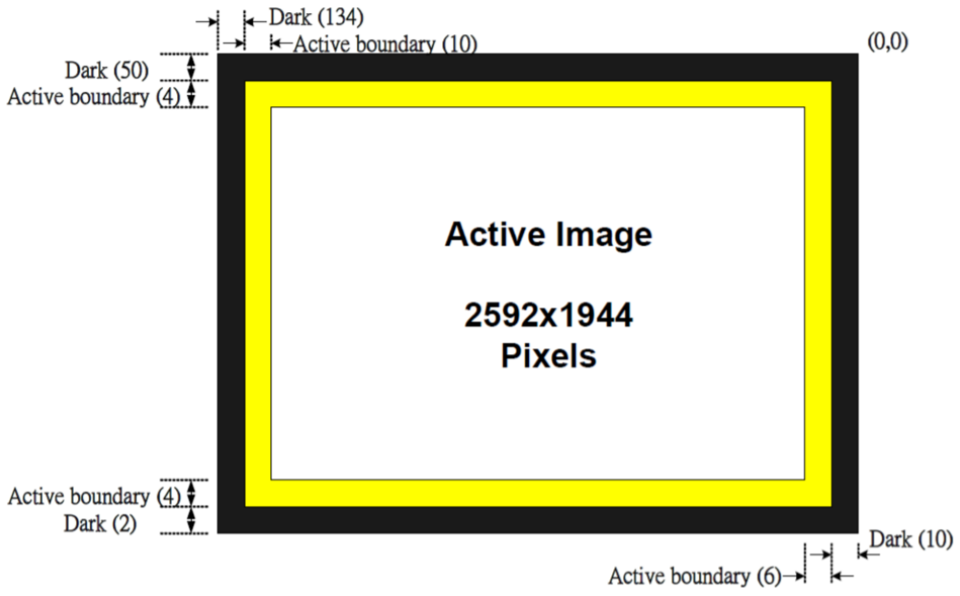


Figure 6: ***Camera pixels organisation***

These pixels are output in a Bayer pattern format consisting of four “colors”—Green1, Green2, Red, and Blue (G1, G2, R, B)—representing three filter colors each in a size of 12 bits as one can see the Figure 7.

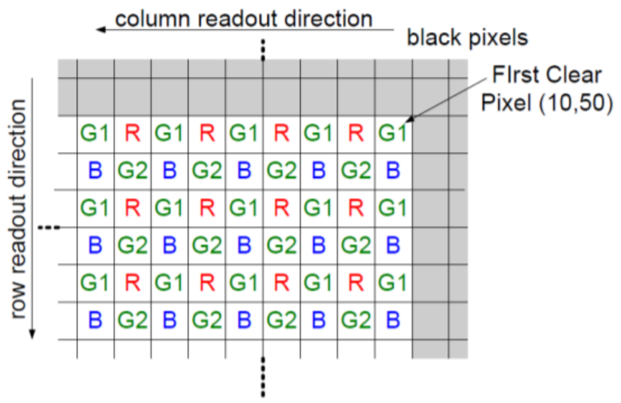


Figure 7: ***Bayer configuration***

However, the LCD driver that will be created by the other group is only compatible with RGB pixels of total size 16 bits and with a resolution of 320x240. To achieve this format, the first thing to do is to configure the camera to output a frame of resolution 640 x 480. To do so, we will order the camera to apply binning x4 to the columns and the rows, i.e that there are 4 columns to be read and averaged per column, and 4 rows to be read and averaged per row. This operation allows us to reduce the impact of aliasing introduced by the use of skip modes. Then, the camera will also apply skipping x4 to the rows and the columns, i.e that for 1 pair of rows the next 3 pairs are skipped and for 1 pair of columns the next 3 pairs are skipped.

After doing all these operations, we can deduce from the datasheet that with a clock of 50 MHz, the camera will output new frames at a rate of 40.3 fps (as opposed to the 77.4 fps with 96 MHz clock in the datasheet) .

1.3.2 Memory organisation

We chose with the group that is creating the LCD driver to store two frames in memory. Therefore we will use two buffers, one for each frame (Double buffering). As each frame needs to have 320 RGB pixels of 16 bits per row, that means that we have 160 words of 32 bits per row. As a frame has 240 rows the total size of a frame in the memory will be : 160 * 240 = 38400 words of 32 bits, i.e 38 400 * 32 = 1 228 800 bits.

As we will send the pixels in burst tranfers of 16 words of 32 bits (this value may change while merging the system with the other group) we will need to do : 38400/16 = 2400 burst transfers to write a full frame in memory. The Figure 8 shows an illustration of our memory.

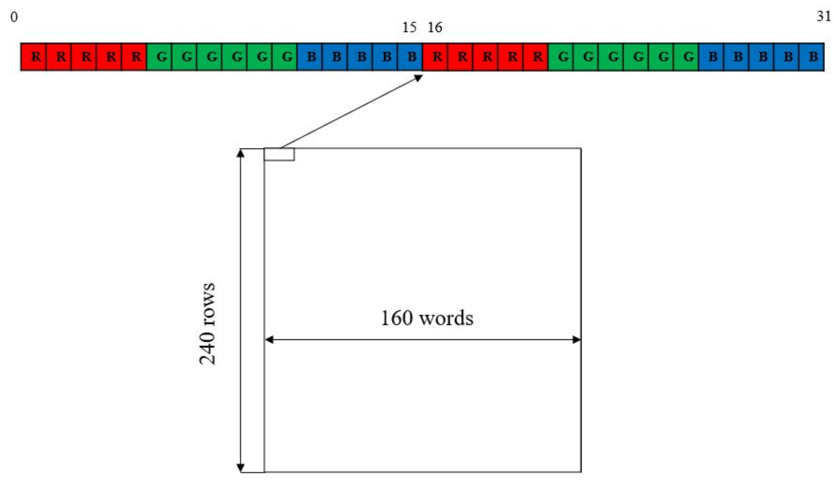

0
15 16
31
R R R R R G G G G G G B B B B B R R R R R G G G G G G B B B B B
240 rows
160 words

Figure 8: ***Organisation of the memory***

1.3.3 Camera configuration

In order to achieve the pixel reading and formatting explained before with the right resolution, we needed to configure the registers of the camera with particular values.

First, the datasheet gives us in Table 1 below, the values to put into the registers in order to achieve binning x4 and skipping x4 both on rows and columns with a final resolution of 640 * 480.

Resolution	Frame Rate	Sub-sampling Mode	Column _Size (R0x04)	Row_ Size (R0x03)	Shutter_ Width_ Lower (R0x09)	Row_ Bin (R0x22 [5:4])	Row_ Skip (R0x22 [2:0])	Column_ Bin (R0x23 [5:4])	Column_ Skip (R0x23 [2:0])
640 x 480 VGA	77.4	bining	2559	1919		3	3	3	3

Table 1: Datasheet of the camera for bining 4x and skipping x4 for both columns and rows

We also put the register R0x00A bit 15 (Invert Pixel Clock) at 1 in order to capture new data on the rising edge of the clock. We let the default values of register R0x001 Row_Start = 54 and register R0x002 Column_start = 16 because they already respect all the requirements for the binning and skipping that we will need, i.e they are both even and Column_start is a multiple of 16. Finally, to respect the minimal blanking requirements we wrote in the register R0x005 Horizontal Blank = 906 and in the register R0x006 Vertical Blank = 25.

1.3.4 Description of the Avalon Slave

Our Camera_Ctrl component has a register interface that can be configured by the NIOS CPU to start/stop the storage of new frames or to know how to access memory. We are going to use the following register interface :

- RegAdr - A location in the register interface where the start address of the new frame to write in memory is updated by the CPU.
- RegLength - A location in the register interface where the length of the frame to write in memory is given by the CPU.
- RegEnable - A location in the register interface where the status of the acquisition of data from the Camera by "Camera Interface" is enabled or stopped by the CPU.
- RegBurst - A location in the register interface where the length of the burst transfer in words to write in memory is given by the CPU.
- RegLight - A location in the register where we can choose if we take the less or most significant bits of pixels depending on the light conditions.

Since there are 5 registers in our register map, the smallest address bus width we could use to index all registers is 3 bits. Table 2 summarizes the write and read behavior of these registers.

In comparison with lab 3, we add here the RegLight register.

Address	Write register	Writedata[31...0]	Read register	Readdata[31...0]
0	RegAdr	→ iRegAdr	RegAdr	iRegAdr →
1	RegLength	→ iRegLength	RegLength	RegLength →
2	RegEnable	→ iRegEnable	RegEnable	iRegEnable →
3	RegBurst	→ iRegBurst	RegBurst	iRegBurst →
4	RegLight	→ iRegLight	RegLight	iRegLight →
5	-	Don't care	-	0x00
6	-	Don't care	-	0x00
7	-	Don't care	-	0x00

Table 2: Register map of the camera

1.3.5 Description of the Camera Interface

Each pixel data D[11 ... 0] is catched on the rising edge of the clock of PIXCLK. The two entry FIFO will both receive, one row of each pair of rows, and then when they both receive a full row, they will empty themselves in parallel in a component that will convert the color pixels sent by the Camera into the resolution and the format that is compatible with the LCD.

Figure 9 shows the state machine that explains the logic to write the informations from the camera to the entry FIFOS. Its states are the following ones:

- Idle : Stays idle till a frame ends and camera interface is enabled
- WaitLine 1 : Waits for the beginning of the first line
- ReadLine 1 : Reads the first line
- WaitLine 2 : Waits for the beginning of the second line
- ReadLine 2 : Reads the second line

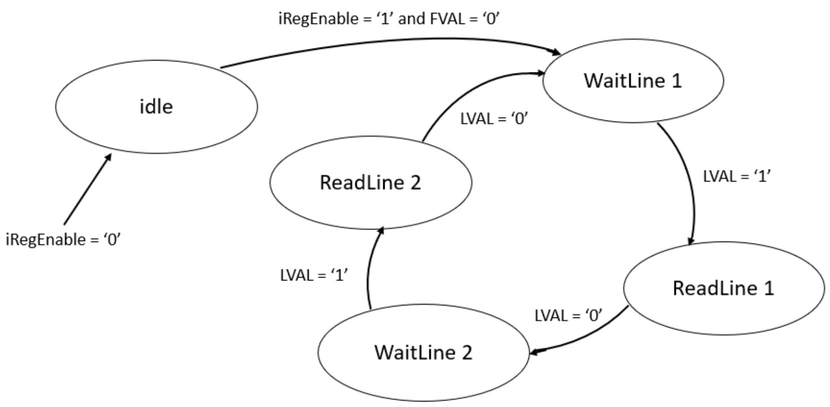


Figure 9: ***State machine explaining how we write the informations from the camera in the first two FIFOS (entry)***

Afterwards, Figure 10 explains how we empty the first two FIFOS and how we write to FIFO exit after executing some operations on these pixels. The operations consist in averaging the two green pixel (G1, G2) to form only one green pixel, and will store the R, G, B 12 bit bayer color pixels to form only one RGB pixel of 36 bit. This will be done with binary operation over both rows in one pair. Then the component will convert all the 36 bit RGB pixels into 16 bit RGB pixels with 5 bit for blue, 6 bits for green, and 5 bits for red. Thanks to the register iRegLight, we decide whether we keep the most or less significant bits.

The states defining this state machine are the following ones:

- Idle : Stays idle till the second FIFO is not empty
- WaitRead : Waits one clock cycle before reading the two first pixels of the pair of rows
- Read G1_B : Reads the pixels G1 and B
- Read G1_B : Reads the pixles R and G2
- Ops : Averages G1 and G2 to create G
- WritePixels : Writes the previous pixel RGB into FIFO_Exit

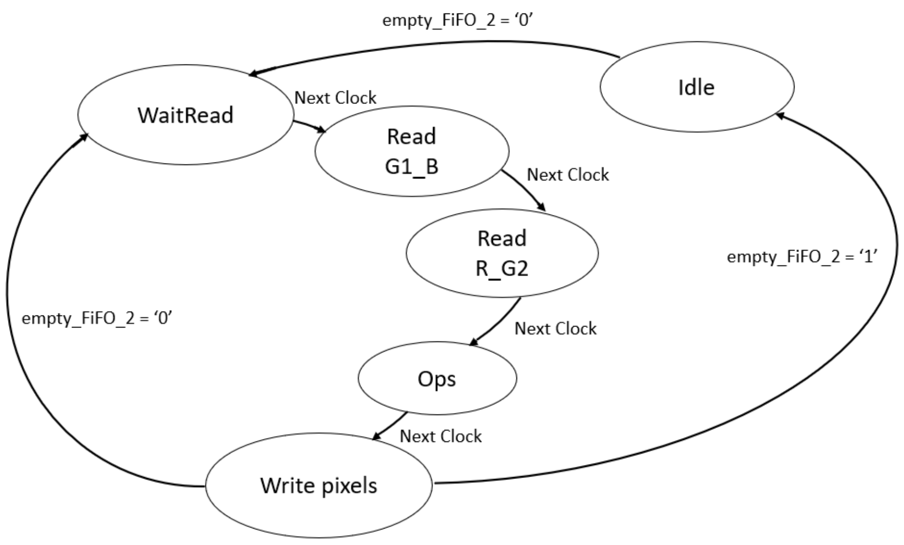


Figure 10: ***State machine explaining how we empty the first two FIFOS and write in FIFO exit***

Every time that two 16 bits RGB pixels are ready, they will be sent to the exit FIFO. For this report, we chose a burst count of 16, but we will tune this value in the real system. So when this FIFO will have at least 32 pixels ready (i.e 16 words of 32 bits ready to be sent) the signal NewData will be set to '1' thanks to usedw signal of FIFO_Exit (see Figure **??**) and will warn the Avalon Master that a burst transfer of 16 new 32 bit words of data is possible. The machine state that explains how we empty FIFO exit and we write to the DMA can be seen in Figure 11. Its states are:

- Idle : Stays idle till FIFO Exit has at least iRegBurst 32 bit words ready in its buffer

- SenData: Sends iRegBurst data to DMA and finishes by putting NewData to '0'

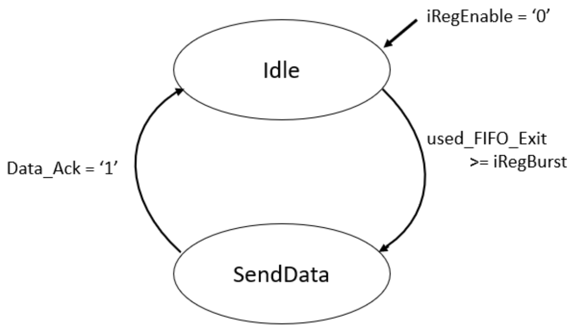


Figure 11: ***State machine explaining how we empty FIFO exit and write to the DMA.***

1.3.6 Description of the Avalon Master

When the Camera Interface is enabled, the exit FIFO will store the pixels that are ready to be sent, and every time that iRegBurst pixels are ready to be sent, i.e 16 words of 32 bits are ready, the DMA will send those words in a BurstCount of 16 (16 words of 32 bits) to the DDR3 memory through the Avalon Bus. We can see an example of transfer in Figure 12. Every time that a burst transfer is made, we will then need to increment the writing address by RegBurst * 4 units (i.e 16 * 4 = 64) before starting the next burst transfer. The state machine of the avalon master is represented in Figure 13 and its states are the following ones:

- Idle : Stays idle till iRegLength /=0

- WaitData : Waits for a new Burst transfer

- WriteData : Writes on memory

- AcqData : Waits end of request

- WaitCPU : Waits for CPU polling to start a new frame

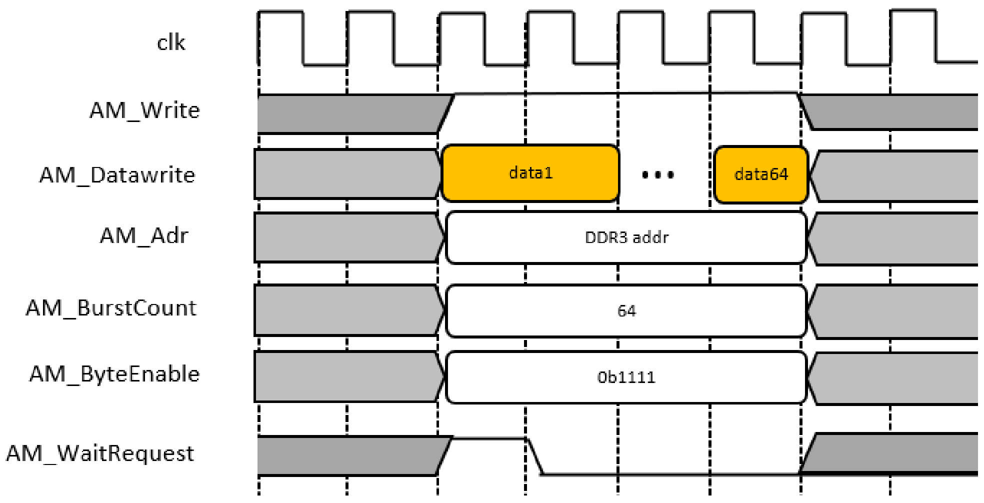


Figure 12: ***Signals on Avalon master***

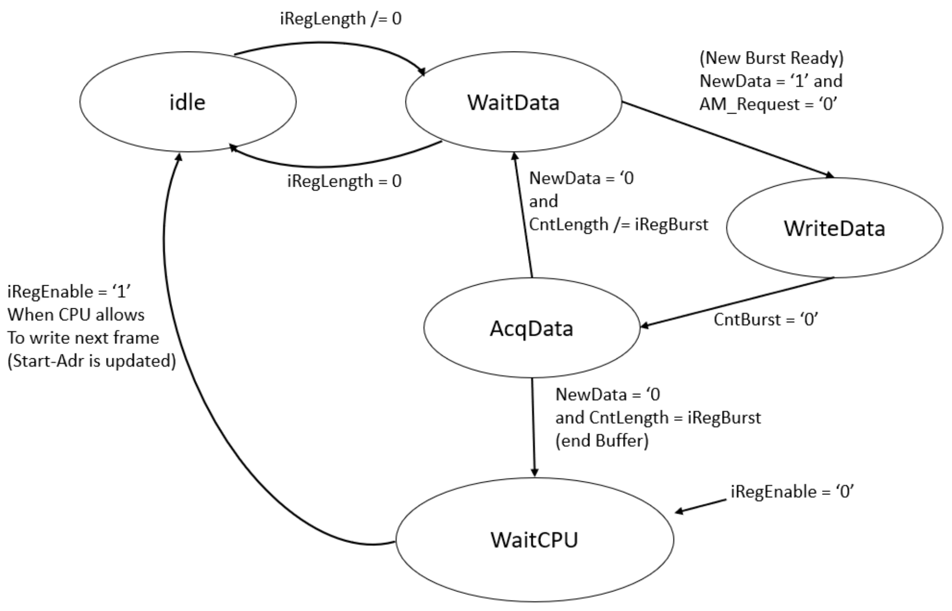


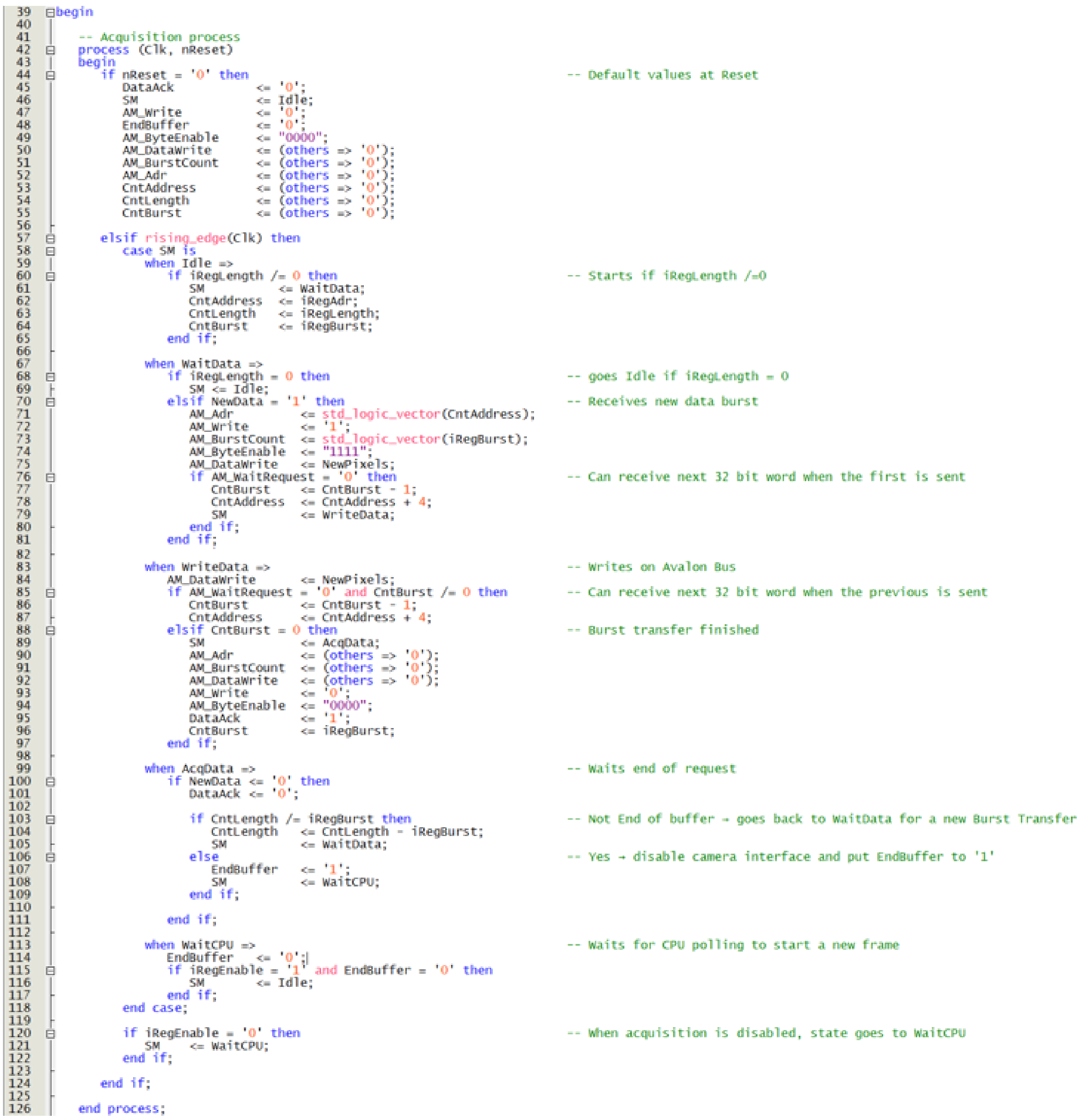
Figure 13: ***State machine of DMA***

When it finishes writing a frame in the memory, iRegEnable is set to '0' and the CPU who is continuously reading this register will know that it finishes writing a frame and that it wants the permission to write a new frame in the second buffer. Similarly, the LCD controller will also have the same behaviour when it finishes to read a frame in a buffer.

2 Writing VHDL code and testing on ModelSim

2.1 VHDL code

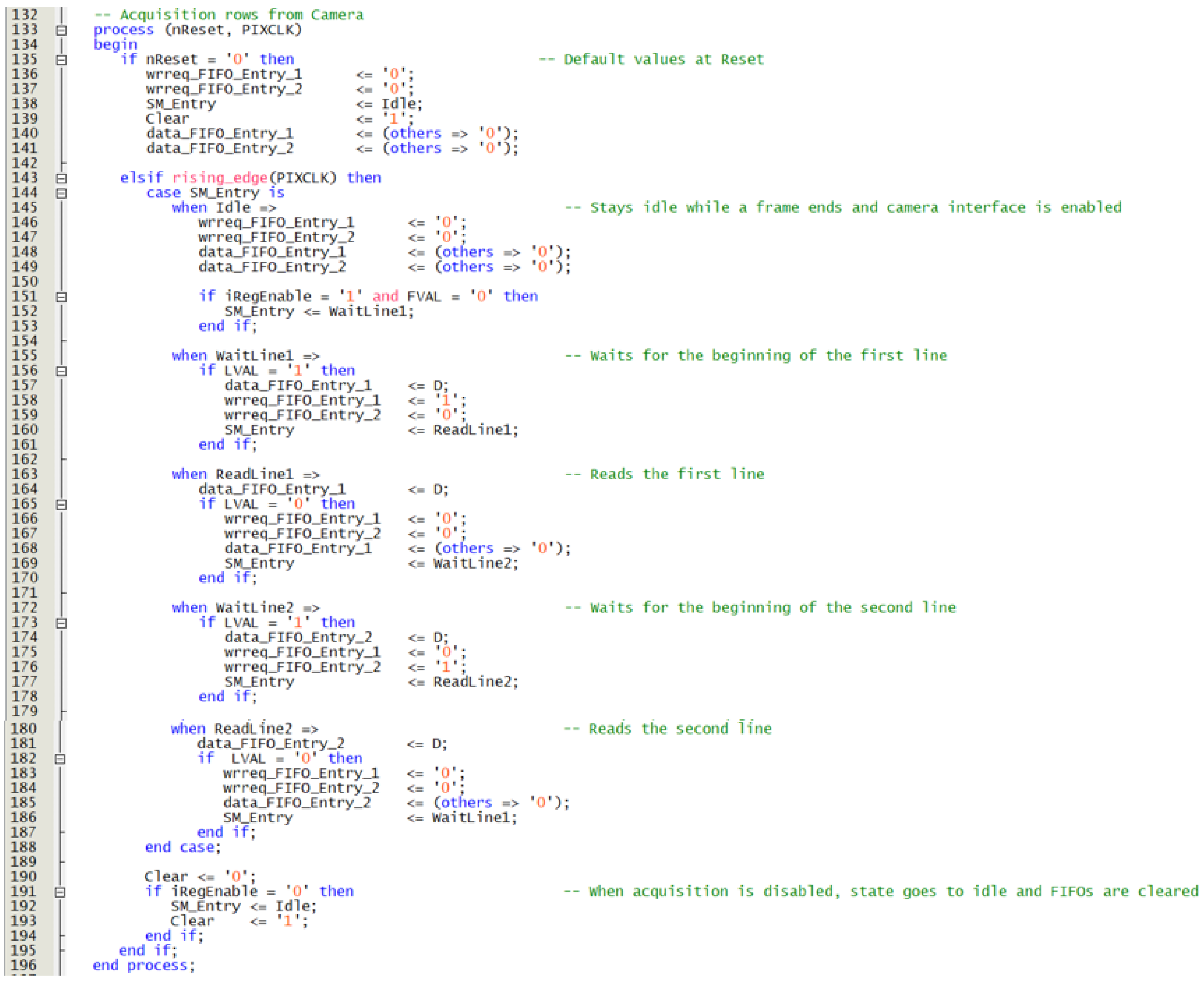
The VHDL code below illustrates the behavior of the state machine of Figure 13. We added a condition at line 120-121 that forces the CPU to enter "wait" mode whenever iRegenable is set to zero.



```vhdl
 39  begin
 40
 41      -- Acquisition process
 42      process (Clk, nReset)
 43      begin
 44          if nReset = '0' then                                            -- Default values at Reset
 45              DataAck             <= '0';
 46              SM                  <= Idle;
 47              AM_Write            <= '0';
 48              EndBuffer           <= '0';
 49              AM_ByteEnable       <= "0000";
 50              AM_DataWrite        <= (others => '0');
 51              AM_BurstCount       <= (others => '0');
 52              AM_Adr              <= (others => '0');
 53              CntAddress          <= (others => '0');
 54              CntLength           <= (others => '0');
 55              CntBurst            <= (others => '0');
 56
 57          elsif rising_edge(Clk) then
 58              case SM is
 59                  when Idle =>
 60                      if iRegLength /= 0 then                         -- Starts if iRegLength /=0
 61                          SM          <= WaitData;
 62                          CntAddress  <= iRegAdr;
 63                          CntLength   <= iRegLength;
 64                          CntBurst    <= iRegBurst;
 65                      end if;
 66
 67                  when WaitData =>
 68                      if iRegLength = 0 then                          -- goes Idle if iRegLength = 0
 69                          SM <= Idle;
 70                      elsif NewData = '1' then                        -- Receives new data burst
 71                          AM_Adr          <= std_logic_vector(CntAddress);
 72                          AM_Write        <= '1';
 73                          AM_BurstCount   <= std_logic_vector(iRegBurst);
 74                          AM_ByteEnable   <= "1111";
 75                          AM_DataWrite    <= NewPixels;
 76                          if AM_WaitRequest = '0' then                -- Can receive next 32 bit word when the first is sent
 77                              CntBurst    <= CntBurst - 1;
 78                              CntAddress  <= CntAddress + 4;
 79                              SM          <= WriteData;
 80                          end if;
 81                      end if;
 82
 83                  when WriteData =>                                   -- Writes on Avalon Bus
 84                      AM_DataWrite        <= NewPixels;
 85                      if AM_WaitRequest = '0' and CntBurst /= 0 then  -- Can receive next 32 bit word when the previous is sent
 86                          CntBurst        <= CntBurst - 1;
 87                          CntAddress      <= CntAddress + 4;
 88                      elsif CntBurst = 0 then                         -- Burst transfer finished
 89                          SM              <= AcqData;
 90                          AM_Adr          <= (others => '0');
 91                          AM_BurstCount   <= (others => '0');
 92                          AM_DataWrite    <= (others => '0');
 93                          AM_Write        <= '0';
 94                          AM_ByteEnable   <= "0000";
 95                          DataAck         <= '1';
 96                          CntBurst        <= iRegBurst;
 97                      end if;
 98
 99                  when AcqData =>                                     -- Waits end of request
100                      if NewData <= '0' then
101                          DataAck <= '0';
102
103                          if CntLength /= iRegBurst then              -- Not End of buffer → goes back to WaitData for a new Burst Transfer
104                              CntLength   <= CntLength - iRegBurst;
105                              SM          <= WaitData;
106                          else                                        -- Yes → disable camera interface and put EndBuffer to '1'
107                              EndBuffer   <= '1';
108                              SM          <= WaitCPU;
109                          end if;
110
111                      end if;
112
113                  when WaitCPU =>                                     -- Waits for CPU polling to start a new frame
114                      EndBuffer   <= '0';
115                      if iRegEnable = '1' and EndBuffer = '0' then
116                          SM      <= Idle;
117                      end if;
118              end case;
119
120              if iRegEnable = '0' then                                -- When acquisition is disabled, state goes to WaitCPU
121                  SM      <= WaitCPU;
122              end if;
123
124          end if;
125
126      end process;
```

Figure 14: ***Code master***

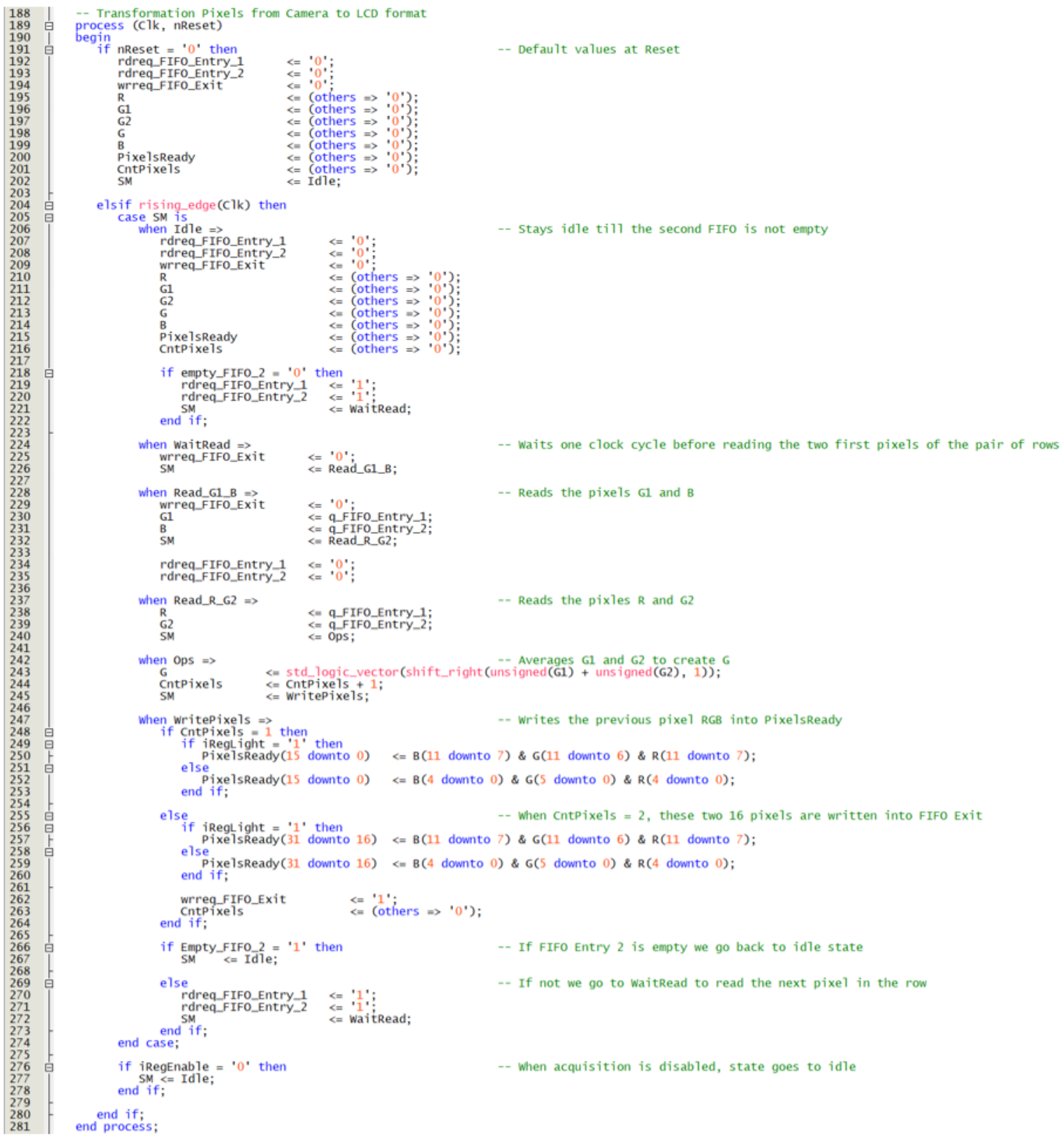
The following code illustrates the behaviour of the state machine in Figure 9 (entry FIFO):



```vhdl
-- Acquisition rows from Camera
process (nReset, PIXCLK)
begin
    if nReset = '0' then                                   -- Default values at Reset
        wrreq_FIFO_Entry_1      <= '0';
        wrreq_FIFO_Entry_2      <= '0';
        SM_Entry                <= Idle;
        Clear                   <= '1';
        data_FIFO_Entry_1       <= (others => '0');
        data_FIFO_Entry_2       <= (others => '0');

    elsif rising_edge(PIXCLK) then
        case SM_Entry is
            when Idle =>                                   -- Stays idle while a frame ends and camera interface is enabled
                wrreq_FIFO_Entry_1      <= '0';
                wrreq_FIFO_Entry_2      <= '0';
                data_FIFO_Entry_1       <= (others => '0');
                data_FIFO_Entry_2       <= (others => '0');

                if iRegEnable = '1' and FVAL = '0' then
                    SM_Entry <= WaitLine1;
                end if;

            when WaitLine1 =>                              -- Waits for the beginning of the first line
                if LVAL = '1' then
                    data_FIFO_Entry_1       <= D;
                    wrreq_FIFO_Entry_1      <= '1';
                    wrreq_FIFO_Entry_2      <= '0';
                    SM_Entry                <= ReadLine1;
                end if;

            when ReadLine1 =>                              -- Reads the first line
                data_FIFO_Entry_1       <= D;
                if LVAL = '0' then
                    wrreq_FIFO_Entry_1      <= '0';
                    wrreq_FIFO_Entry_2      <= '0';
                    data_FIFO_Entry_1       <= (others => '0');
                    SM_Entry                <= WaitLine2;
                end if;

            when WaitLine2 =>                              -- Waits for the beginning of the second line
                if LVAL = '1' then
                    data_FIFO_Entry_2       <= D;
                    wrreq_FIFO_Entry_1      <= '0';
                    wrreq_FIFO_Entry_2      <= '1';
                    SM_Entry                <= ReadLine2;
                end if;

            when ReadLine2 =>                              -- Reads the second line
                data_FIFO_Entry_2       <= D;
                if  LVAL = '0' then
                    wrreq_FIFO_Entry_1      <= '0';
                    wrreq_FIFO_Entry_2      <= '0';
                    data_FIFO_Entry_2       <= (others => '0');
                    SM_Entry                <= WaitLine1;
                end if;
        end case;

        Clear <= '0';
        if iRegEnable = '0' then                           -- When acquisition is disabled, state goes to idle and FIFOs are cleared
            SM_Entry <= Idle;
            Clear    <= '1';
        end if;
    end if;
end process;
```

Figure 15: ***Code camera acquisition***

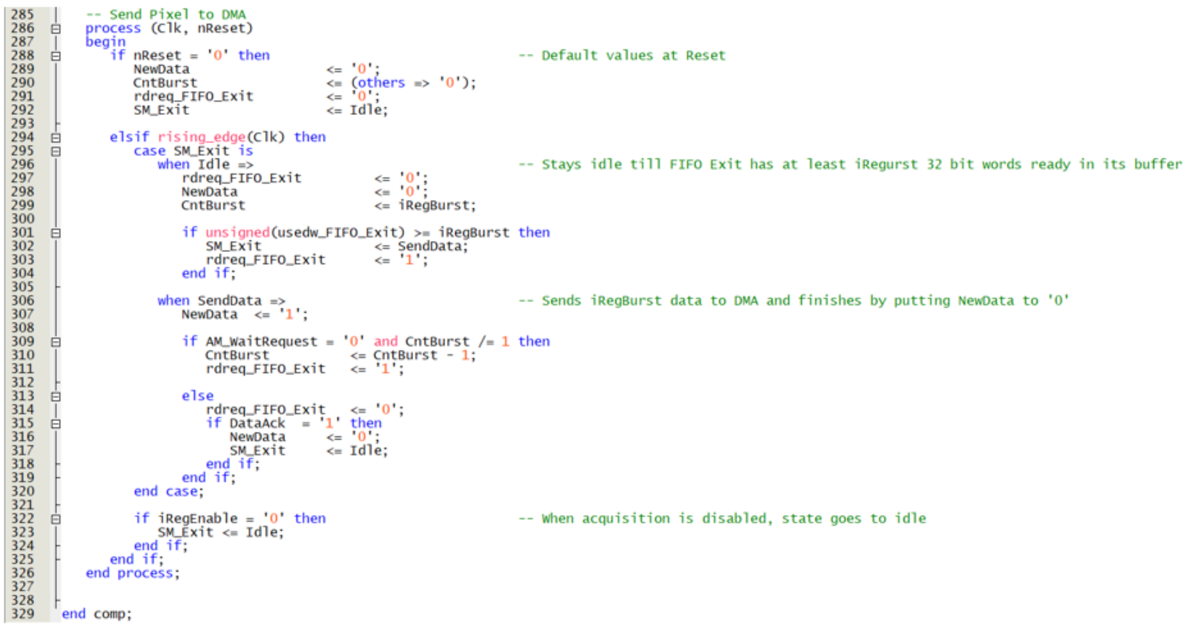
Below, the code represents the state machine of Figure 10, which transforms the pixels. The 4 pixels are read from line 220 to 239, transformed on lines 243 and written on lines 247-274.



```
-- Transformation Pixels from Camera to LCD format
process (Clk, nReset)
begin
   if nReset = '0' then                                     -- Default values at Reset
      rdreq_FIFO_Entry_1      <= '0';
      rdreq_FIFO_Entry_2      <= '0';
      wrreq_FIFO_Exit         <= '0';
      R                       <= (others => '0');
      G1                      <= (others => '0');
      G2                      <= (others => '0');
      G                       <= (others => '0');
      B                       <= (others => '0');
      PixelsReady             <= (others => '0');
      CntPixels               <= (others => '0');
      SM                      <= Idle;

   elsif rising_edge(Clk) then
      case SM is
         when Idle =>                                       -- Stays idle till the second FIFO is not empty
            rdreq_FIFO_Entry_1      <= '0';
            rdreq_FIFO_Entry_2      <= '0';
            wrreq_FIFO_Exit         <= '0';
            R                       <= (others => '0');
            G1                      <= (others => '0');
            G2                      <= (others => '0');
            G                       <= (others => '0');
            B                       <= (others => '0');
            PixelsReady             <= (others => '0');
            CntPixels               <= (others => '0');

            if empty_FIFO_2 = '0' then
               rdreq_FIFO_Entry_1   <= '1';
               rdreq_FIFO_Entry_2   <= '1';
               SM                   <= WaitRead;
            end if;

         when WaitRead =>                                   -- Waits one clock cycle before reading the two first pixels of the pair of rows
            wrreq_FIFO_Exit      <= '0';
            SM                   <= Read_G1_B;

         when Read_G1_B =>                                  -- Reads the pixels G1 and B
            wrreq_FIFO_Exit      <= '0';
            G1                   <= q_FIFO_Entry_1;
            B                    <= q_FIFO_Entry_2;
            SM                   <= Read_R_G2;

            rdreq_FIFO_Entry_1   <= '0';
            rdreq_FIFO_Entry_2   <= '0';

         when Read_R_G2 =>                                  -- Reads the pixles R and G2
            R                    <= q_FIFO_Entry_1;
            G2                   <= q_FIFO_Entry_2;
            SM                   <= Ops;

         when Ops =>                                        -- Averages G1 and G2 to create G
            G              <= std_logic_vector(shift_right(unsigned(G1) + unsigned(G2), 1));
            CntPixels      <= CntPixels + 1;
            SM             <= WritePixels;

         when WritePixels =>                                -- Writes the previous pixel RGB into PixelsReady
            if CntPixels = 1 then
               if iRegLight = '1' then
                  PixelsReady(15 downto 0)   <= B(11 downto 7) & G(11 downto 6) & R(11 downto 7);
               else
                  PixelsReady(15 downto 0)   <= B(4 downto 0) & G(5 downto 0) & R(4 downto 0);
               end if;

            else                                            -- When CntPixels = 2, these two 16 pixels are written into FIFO Exit
               if iRegLight = '1' then
                  PixelsReady(31 downto 16)  <= B(11 downto 7) & G(11 downto 6) & R(11 downto 7);
               else
                  PixelsReady(31 downto 16)  <= B(4 downto 0) & G(5 downto 0) & R(4 downto 0);
               end if;

               wrreq_FIFO_Exit         <= '1';
               CntPixels               <= (others => '0');
            end if;

            if Empty_FIFO_2 = '1' then                      -- If FIFO Entry 2 is empty we go back to idle state
               SM    <= Idle;

            else                                            -- If not we go to WaitRead to read the next pixel in the row
               rdreq_FIFO_Entry_1   <= '1';
               rdreq_FIFO_Entry_2   <= '1';
               SM                   <= WaitRead;
            end if;
      end case;

      if iRegEnable = '0' then                              -- When acquisition is disabled, state goes to idle
         SM <= Idle;
      end if;

   end if;
end process;
```

Figure 16: ***Code pixel transformation***

Finally, the code below handles the state machine on Figure 11, which sends the pixels to the DMA at each burst transfer.



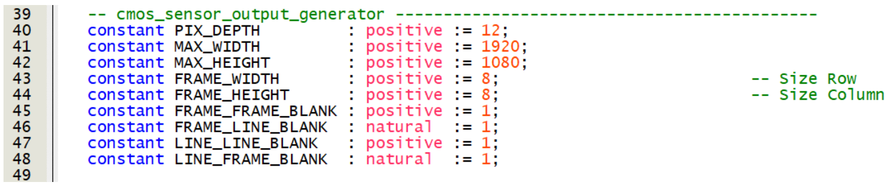
```vhdl
-- Send Pixel to DMA
process (Clk, nReset)
begin
   if nReset = '0' then                                   -- Default values at Reset
      NewData                 <= '0';
      CntBurst                <= (others => '0');
      rdreq_FIFO_Exit         <= '0';
      SM_Exit                 <= Idle;

   elsif rising_edge(Clk) then
      case SM_Exit is
         when Idle =>                                     -- Stays idle till FIFO Exit has at least iRegurst 32 bit words ready in its buffer
            rdreq_FIFO_Exit         <= '0';
            NewData                 <= '0';
            CntBurst                <= iRegBurst;

            if unsigned(usedw_FIFO_Exit) >= iRegBurst then
               SM_Exit              <= SendData;
               rdreq_FIFO_Exit      <= '1';
            end if;

         when SendData =>                                 -- Sends iRegBurst data to DMA and finishes by putting NewData to '0'
            NewData  <= '1';

            if AM_WaitRequest = '0' and CntBurst /= 1 then
               CntBurst          <= CntBurst - 1;
               rdreq_FIFO_Exit   <= '1';

            else
               rdreq_FIFO_Exit   <= '0';
               if DataAck  = '1' then
                  NewData     <= '0';
                  SM_Exit     <= Idle;
               end if;
            end if;
      end case;

      if iRegEnable = '0' then                            -- When acquisition is disabled, state goes to idle
         SM_Exit <= Idle;
      end if;
   end if;
end process;

end comp;
```

Figure 17: ***Code to send pixels to DMA***

2.2 ModelSim testing

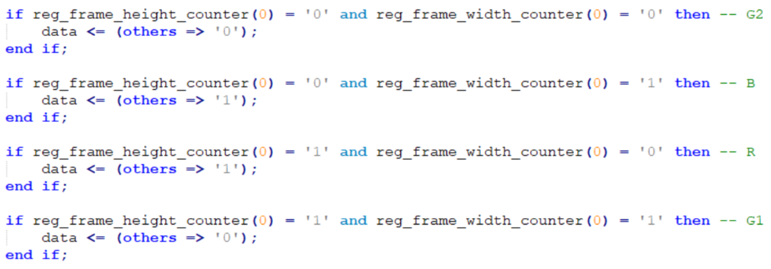
To test if the implementation of the architecture is correct we use a VHDL testbench, ModelSim.

To simulate, we generate 8x8 RGB frames with the cmos_sensor_output_generator (see Figures 18 and 19) which then leaves us, after doing the operations on the bayer pixels, with 16 RGB pixels of 16 bits. Finally, we have 8 words of 32 bits that we send with 2 Bursts of length 4.



```vhdl
-- cmos_sensor_output_generator -------------------------------------------
constant PIX_DEPTH          : positive := 12;
constant MAX_WIDTH          : positive := 1920;
constant MAX_HEIGHT         : positive := 1080;
constant FRAME_WIDTH        : positive := 8;                    -- Size Row
constant FRAME_HEIGHT       : positive := 8;                    -- Size Column
constant FRAME_FRAME_BLANK  : positive := 1;
constant FRAME_LINE_BLANK   : natural  := 1;
constant LINE_LINE_BLANK    : positive := 1;
constant LINE_FRAME_BLANK   : natural  := 1;
```

Figure 18: ***cmos settings for testbench***



```vhdl
if reg_frame_height_counter(0) = '0' and reg_frame_width_counter(0) = '0' then -- G2
    data <= (others => '0');
end if;

if reg_frame_height_counter(0) = '0' and reg_frame_width_counter(0) = '1' then -- B
    data <= (others => '1');
end if;

if reg_frame_height_counter(0) = '1' and reg_frame_width_counter(0) = '0' then -- R
    data <= (others => '1');
end if;

if reg_frame_height_counter(0) = '1' and reg_frame_width_counter(0) = '1' then -- G1
    data <= (others => '0');
end if;
```

Figure 19: ***RGB pattern for testbench***

The following Figure 20 shows how the bayer pixels are sent by the camera via the data signal to our Camera_Ctrl.

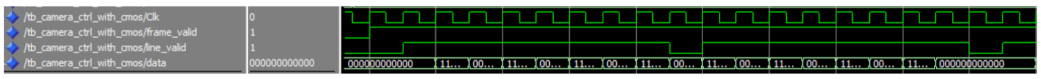
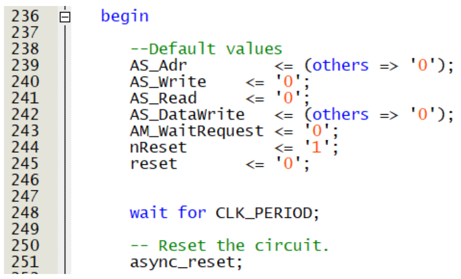


Figure 20: ***Testbench simulation: frame (8x8)***

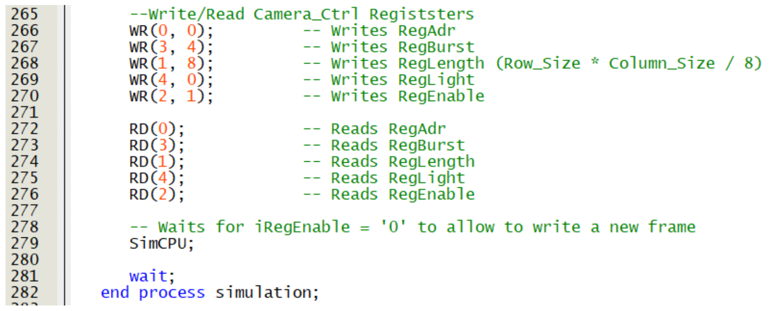
The code below (a snippet from "tb_Camera_Ctrl-with_cmos") shows how we start our simulation.



```
begin

    --Default values
    AS_Adr          <= (others => '0');
    AS_Write     <= '0';
    AS_Read      <= '0';
    AS_DataWrite    <= (others => '0');
    AM_WaitRequest <= '0';
    nReset          <= '1';
    reset        <= '0';


    wait for CLK_PERIOD;

    -- Reset the circuit.
    async_reset;
```

Figure 21: ***Code to initialize the simulation***

Then, the next code (Figure 22) shows how we write and read to the registers of Camera_Ctrl so that it is ready to receive 8x8 RGB frames with 2 bursts of length 4. We can visualize the results in Figure 23 (write) and Figure 24 (read).



```
    --Write/Read Camera_Ctrl Registsters
    WR(0, 0);          -- Writes RegAdr
    WR(3, 4);          -- Writes RegBurst
    WR(1, 8);          -- Writes RegLength (Row_Size * Column_Size / 8)
    WR(4, 0);          -- Writes RegLight
    WR(2, 1);          -- Writes RegEnable

    RD(0);             -- Reads RegAdr
    RD(3);             -- Reads RegBurst
    RD(1);             -- Reads RegLength
    RD(4);             -- Reads RegLight
    RD(2);             -- Reads RegEnable

    -- Waits for iRegEnable = '0' to allow to write a new frame
    SimCPU;

    wait;
end process simulation;
```

Figure 22: ***Code to write and read the main registers***

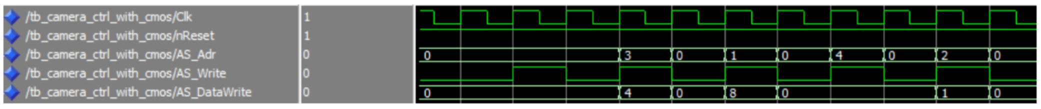


Figure 23: ***Testbench simulation: writedata***

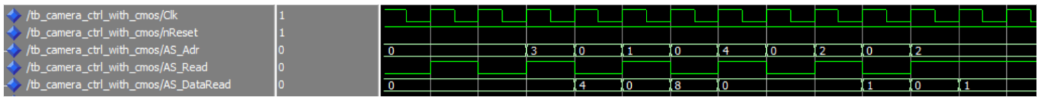
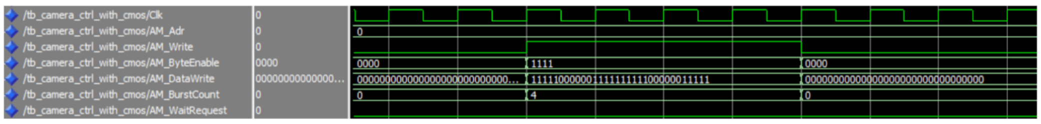


Figure 24: ***Testbench simulation: readdata***

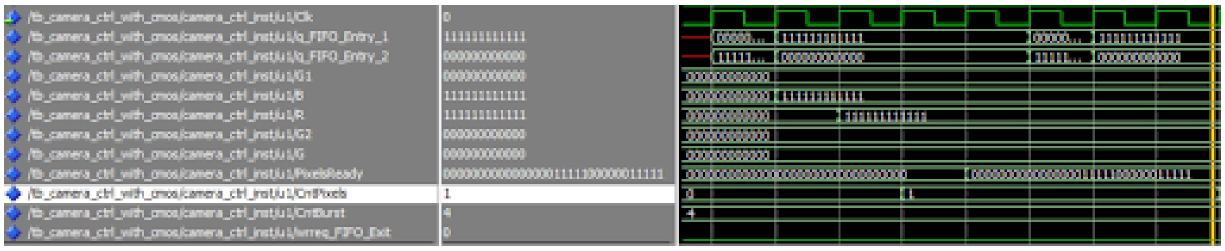
In Figure 25 one can see the waves signals of the Avalon Master to realise the first Burst transfer of 4 words of 32 bits.



```
/tb_camera_ctrl_with_cmos/Clk              0
/tb_camera_ctrl_with_cmos/AM_Adr           0
/tb_camera_ctrl_with_cmos/AM_Write         0
/tb_camera_ctrl_with_cmos/AM_ByteEnable    0000
/tb_camera_ctrl_with_cmos/AM_DataWrite     00000000000000...
/tb_camera_ctrl_with_cmos/AM_BurstCount    0
/tb_camera_ctrl_with_cmos/AM_WaitRequest   0
```

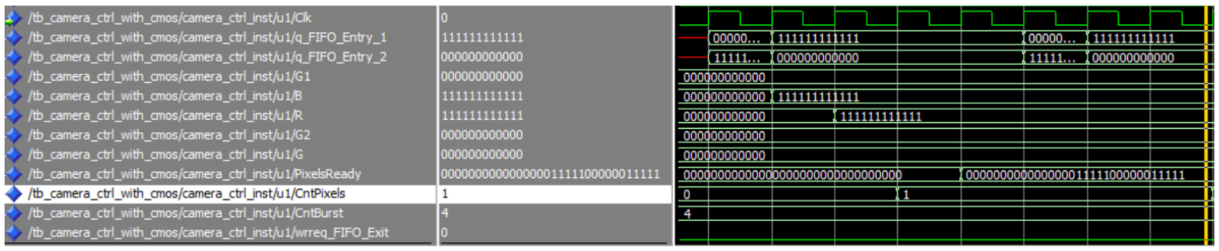
Figure 25: ***Testbench simulation: DMA first Burst transfer of 4 words of 32 bits***

Finally, in Figure 26 one can see in (a) how we store the first RGB pixel in PixelsReady (15 downto 0) and in (b) how we store the second RGB pixel in PixelsReady (31 downto 16). Then PixelsReady is written into FIFO_Exit to be sent after to the DMA.



```
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/Clk             0
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/q_FIFO_Entry_1  111111111111
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/q_FIFO_Entry_2  000000000000
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/G1              000000000000
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/B               111111111111
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/R               111111111111
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/G2              000000000000
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/G               000000000000
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/PixelsReady     00000000000000001111100000011111
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/CntPixels       1
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/CntBurst        4
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/wrreq_FIFO_Exit 0
```

(a)



```
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/Clk             0
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/q_FIFO_Entry_1  111111111111
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/q_FIFO_Entry_2  000000000000
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/G1              000000000000
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/B               111111111111
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/R               111111111111
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/G2              000000000000
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/G               000000000000
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/PixelsReady     00000000000000001111100000011111
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/CntPixels       1
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/CntBurst        4
/tb_camera_ctrl_with_cmos/camera_ctrl_inst/u1/wrreq_FIFO_Exit 0
```

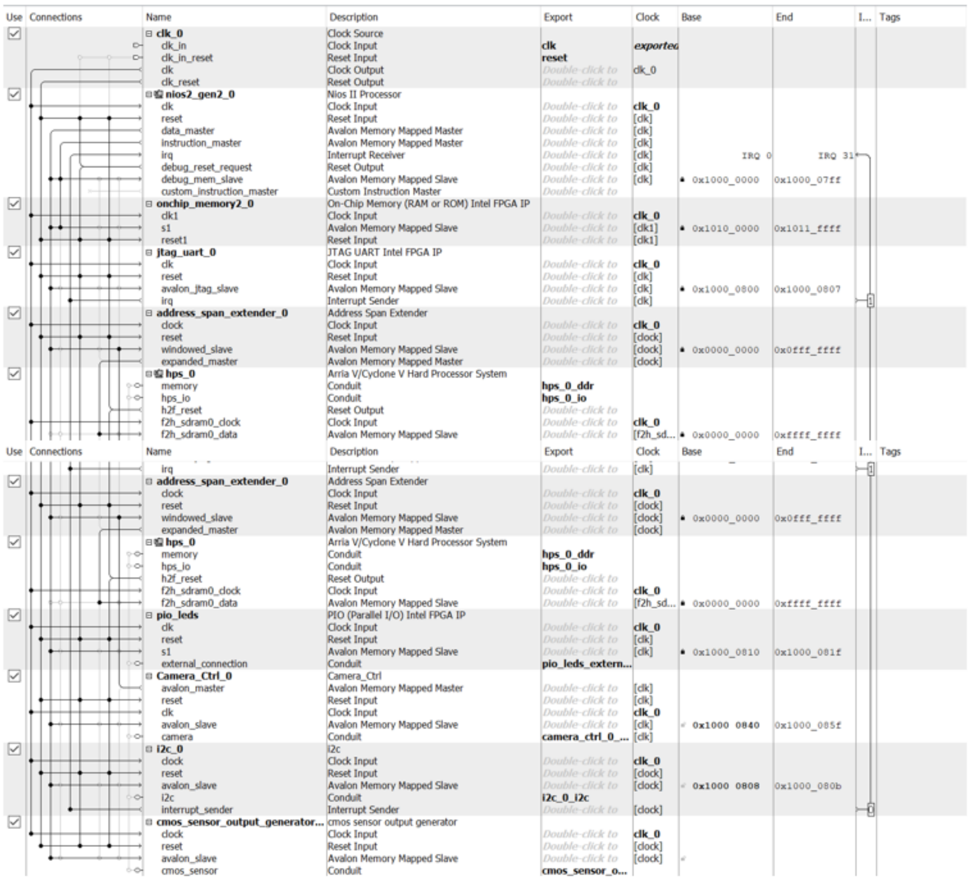
(b)

Figure 26: ***Operation on pixels before being written to FIFO_Exit (a) Pixel 1; (b) Pixel 2;***

3 Qsys implementation and execute C-code to FPGA board

3.1 Qsys implementation

After, designing and testing our components, we added them in Qsys (Figure27). This configuration is valid when we use the cmos_sensor_output_generator and when we use the real camera the only difference is to remove the cmos_sensor_output_generator from this Qsys architecture.



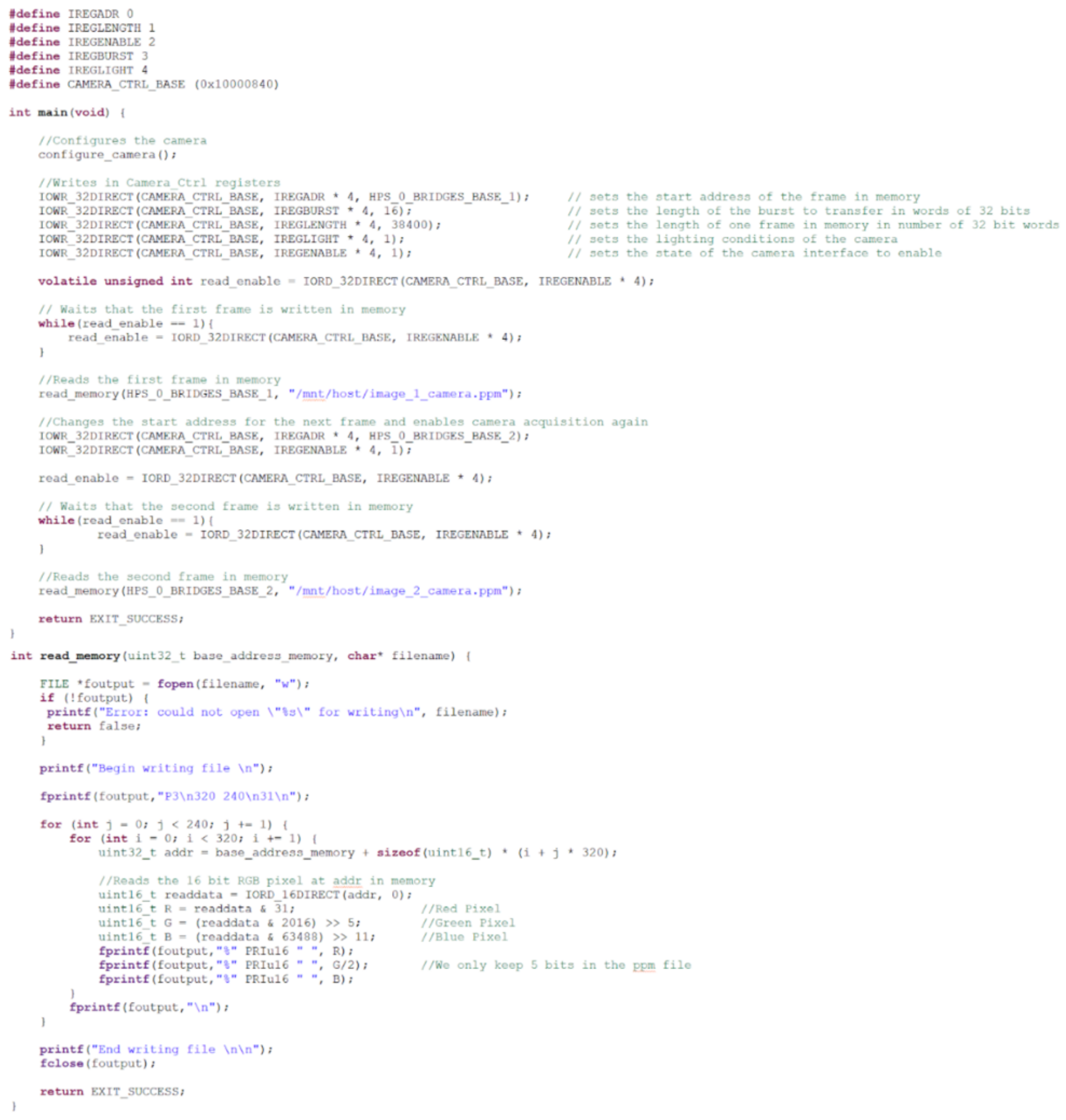
Use	Connections	Name	Description	Export	Clock	Base	End	I...	Tags
✓		⊟ clk_0	Clock Source						
		clk_in	Clock Input	clk	exportec				
		clk_in_reset	Reset Input	reset					
		clk	Clock Output	Double-click to	clk_0				
		clk_reset	Reset Output	Double-click to					
✓		⊟ nios2_gen2_0	Nios II Processor						
		clk	Clock Input	Double-click to	clk_0				
		reset	Reset Input	Double-click to	[clk]				
		data_master	Avalon Memory Mapped Master	Double-click to	[clk]				
		instruction_master	Avalon Memory Mapped Master	Double-click to	[clk]				
		irq	Interrupt Receiver	Double-click to	[clk]	IRQ 0	IRQ 31		
		debug_reset_request	Reset Output	Double-click to	[clk]				
		debug_mem_slave	Avalon Memory Mapped Slave	Double-click to	[clk]	0x1000_0000	0x1000_07ff		
		custom_instruction_master	Custom Instruction Master	Double-click to					
✓		⊟ onchip_memory2_0	On-Chip Memory (RAM or ROM) Intel FPGA IP						
		clk1	Clock Input	Double-click to	clk_0				
		s1	Avalon Memory Mapped Slave	Double-click to	[clk1]	0x1010_0000	0x1011_ffff		
		reset1	Reset Input	Double-click to	[clk1]				
✓		⊟ jtag_uart_0	JTAG UART Intel FPGA IP						
		clk	Clock Input	Double-click to	clk_0				
		reset	Reset Input	Double-click to	[clk]				
		avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to	[clk]	0x1000_0800	0x1000_0807		
		irq	Interrupt Sender	Double-click to	[clk]			1	
✓		⊟ address_span_extender_0	Address Span Extender						
		clock	Clock Input	Double-click to	clk_0				
		reset	Reset Input	Double-click to	[clock]				
		windowed_slave	Avalon Memory Mapped Slave	Double-click to	[clock]	0x0000_0000	0x0fff_ffff		
		expanded_master	Avalon Memory Mapped Master	Double-click to	[clock]				
✓		⊟ hps_0	Arria V/Cyclone V Hard Processor System						
		memory	Conduit	hps_0_ddr					
		hps_io	Conduit	hps_0_io					
		h2f_reset	Reset Output	Double-click to					
		f2h_sdram0_clock	Clock Input	Double-click to	clk_0				
		f2h_sdram0_data	Avalon Memory Mapped Slave	Double-click to	[f2h_sd...	0x0000_0000	0xffff_ffff		

Use	Connections	Name	Description	Export	Clock	Base	End	I...	Tags
		irq	Interrupt Sender	Double-click to	[clk]			1	
✓		⊟ address_span_extender_0	Address Span Extender						
		clock	Clock Input	Double-click to	clk_0				
		reset	Reset Input	Double-click to	[clock]				
		windowed_slave	Avalon Memory Mapped Slave	Double-click to	[clock]	0x0000_0000	0x0fff_ffff		
		expanded_master	Avalon Memory Mapped Master	Double-click to	[clock]				
✓		⊟ hps_0	Arria V/Cyclone V Hard Processor System						
		memory	Conduit	hps_0_ddr					
		hps_io	Conduit	hps_0_io					
		h2f_reset	Reset Output	Double-click to					
		f2h_sdram0_clock	Clock Input	Double-click to	clk_0				
		f2h_sdram0_data	Avalon Memory Mapped Slave	Double-click to	[f2h_sd...	0x0000_0000	0xffff_ffff		
✓		⊟ pio_leds	PIO (Parallel I/O) Intel FPGA IP						
		clk	Clock Input	Double-click to	clk_0				
		reset	Reset Input	Double-click to	[clk]				
		s1	Avalon Memory Mapped Slave	Double-click to	[clk]	0x1000_0810	0x1000_081f		
		external_connection	Conduit	pio_leds_extern...					
✓		⊟ Camera_Ctrl_0	Camera_Ctrl						
		avalon_master	Avalon Memory Mapped Master	Double-click to	[clk]				
		reset	Reset Input	Double-click to	[clk]				
		clk	Clock Input	Double-click to	clk_0				
		avalon_slave	Avalon Memory Mapped Slave	Double-click to	[clk]	0x1000 0840	0x1000_085f		
		camera	Conduit	camera_ctrl_0_...	[clk]				
✓		⊟ i2c_0	i2c						
		clock	Clock Input	Double-click to	clk_0				
		reset	Reset Input	Double-click to	[clock]				
		avalon_slave	Avalon Memory Mapped Slave	Double-click to	[clock]	0x1000 0808	0x1000_080b		
		i2c	Conduit	i2c_0_i2c					
		interrupt_sender	Interrupt Sender	Double-click to	[clock]			0	
✓		⊟ cmos_sensor_output_generator...	cmos sensor output generator						
		clock	Clock Input	Double-click to	clk_0				
		reset	Reset Input	Double-click to	[clock]				
		avalon_slave	Avalon Memory Mapped Slave	Double-click to	[clock]				
		cmos_sensor	Conduit	cmos_sensor_o...					

Figure 27: ***Qsys architecture***

3.2 C code

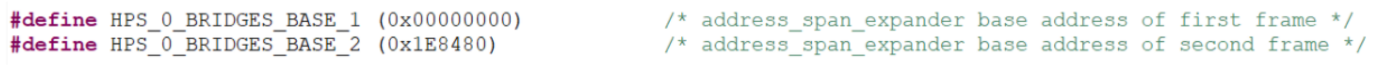
In Figure 28 one can see the main that we used to write two frames in the DDR3 memory and to read them with our host computer.



```c
#define IREGADR 0
#define IREGLENGTH 1
#define IREGENABLE 2
#define IREGBURST 3
#define IREGLIGHT 4
#define CAMERA_CTRL_BASE (0x10000840)

int main(void) {

    //Configures the camera
    configure_camera();

    //Writes in Camera_Ctrl registers
    IOWR_32DIRECT(CAMERA_CTRL_BASE, IREGADR * 4, HPS_0_BRIDGES_BASE_1);     // sets the start address of the frame in memory
    IOWR_32DIRECT(CAMERA_CTRL_BASE, IREGBURST * 4, 16);                     // sets the length of the burst to transfer in words of 32 bits
    IOWR_32DIRECT(CAMERA_CTRL_BASE, IREGLENGTH * 4, 38400);                 // sets the length of one frame in memory in number of 32 bit words
    IOWR_32DIRECT(CAMERA_CTRL_BASE, IREGLIGHT * 4, 1);                      // sets the lighting conditions of the camera
    IOWR_32DIRECT(CAMERA_CTRL_BASE, IREGENABLE * 4, 1);                     // sets the state of the camera interface to enable

    volatile unsigned int read_enable = IORD_32DIRECT(CAMERA_CTRL_BASE, IREGENABLE * 4);

    // Waits that the first frame is written in memory
    while(read_enable == 1){
        read_enable = IORD_32DIRECT(CAMERA_CTRL_BASE, IREGENABLE * 4);
    }

    //Reads the first frame in memory
    read_memory(HPS_0_BRIDGES_BASE_1, "/mnt/host/image_1_camera.ppm");

    //Changes the start address for the next frame and enables camera acquisition again
    IOWR_32DIRECT(CAMERA_CTRL_BASE, IREGADR * 4, HPS_0_BRIDGES_BASE_2);
    IOWR_32DIRECT(CAMERA_CTRL_BASE, IREGENABLE * 4, 1);

    read_enable = IORD_32DIRECT(CAMERA_CTRL_BASE, IREGENABLE * 4);

    // Waits that the second frame is written in memory
    while(read_enable == 1){
            read_enable = IORD_32DIRECT(CAMERA_CTRL_BASE, IREGENABLE * 4);
    }

    //Reads the second frame in memory
    read_memory(HPS_0_BRIDGES_BASE_2, "/mnt/host/image_2_camera.ppm");

    return EXIT_SUCCESS;
}
int read_memory(uint32_t base_address_memory, char* filename) {

    FILE *foutput = fopen(filename, "w");
    if (!foutput) {
     printf("Error: could not open \"%s\" for writing\n", filename);
     return false;
    }

    printf("Begin writing file \n");

    fprintf(foutput,"P3\n320 240\n31\n");

    for (int j = 0; j < 240; j += 1) {
        for (int i = 0; i < 320; i += 1) {
            uint32_t addr = base_address_memory + sizeof(uint16_t) * (i + j * 320);

            //Reads the 16 bit RGB pixel at addr in memory
            uint16_t readdata = IORD_16DIRECT(addr, 0);
            uint16_t R = readdata & 31;                 //Red Pixel
            uint16_t G = (readdata & 2016) >> 5;        //Green Pixel
            uint16_t B = (readdata & 63488) >> 11;      //Blue Pixel
            fprintf(foutput,"%" PRIu16 " ", R);
            fprintf(foutput,"%" PRIu16 " ", G/2);       //We only keep 5 bits in the ppm file
            fprintf(foutput,"%" PRIu16 " ", B);
        }
        fprintf(foutput,"\n");
    }

    printf("End writing file \n\n");
    fclose(foutput);

    return EXIT_SUCCESS;
}
```

Figure 28: ***C code main***

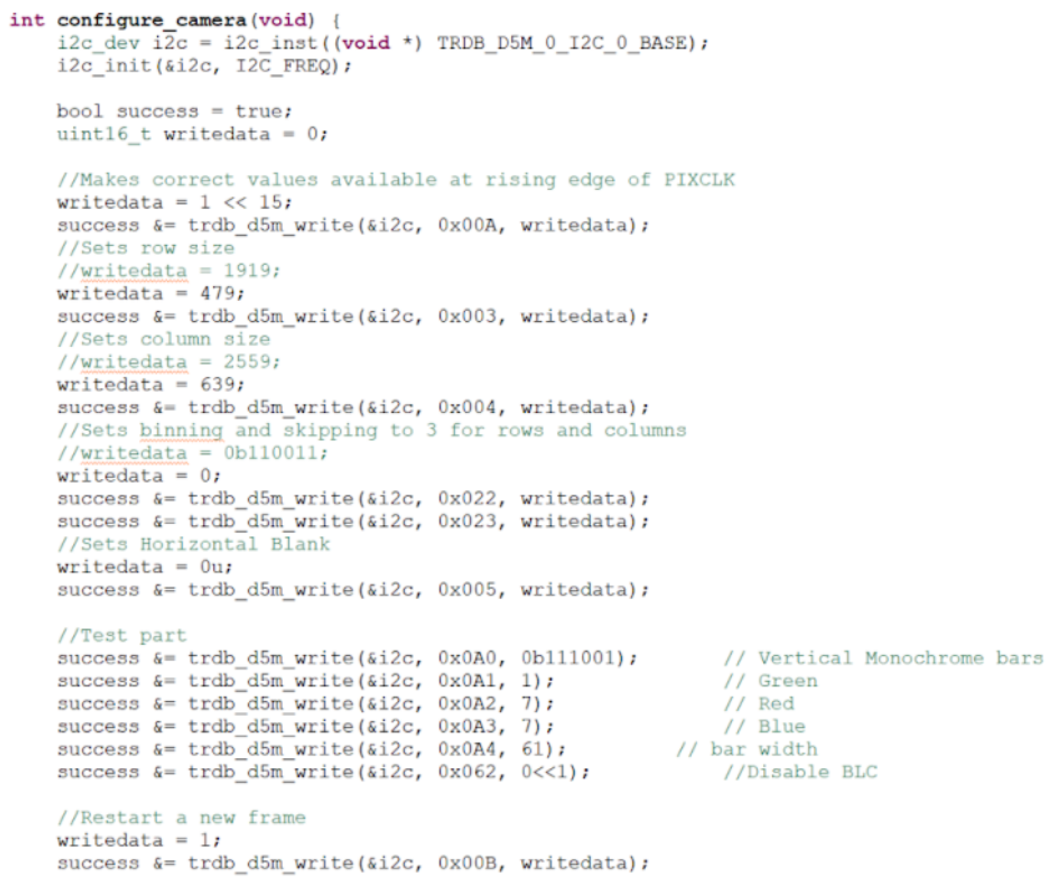
In Figure 29 one can see the base address that we chose to write the frames in memory. As the size of a frame is : 1 228 800 bits, we chose to start the second frame at address 2 000 000, to let enough space.



```c
#define HPS_0_BRIDGES_BASE_1 (0x00000000)          /* address_span_expander base address of first frame */
#define HPS_0_BRIDGES_BASE_2 (0x1E8480)            /* address_span_expander base address of second frame */
```

Figure 29: ***Memory start address of the two frames***

Firstly, we generated a test pattern of Monochrome vertical bars with the TRDB-D5M camera (see Figure 31) by using the following configuration for the camera :



```c
int configure_camera(void) {
    i2c_dev i2c = i2c_inst((void *) TRDB_D5M_0_I2C_0_BASE);
    i2c_init(&i2c, I2C_FREQ);

    bool success = true;
    uint16_t writedata = 0;

    //Makes correct values available at rising edge of PIXCLK
    writedata = 1 << 15;
    success &= trdb_d5m_write(&i2c, 0x00A, writedata);
    //Sets row size
    //writedata = 1919;
    writedata = 479;
    success &= trdb_d5m_write(&i2c, 0x003, writedata);
    //Sets column size
    //writedata = 2559;
    writedata = 639;
    success &= trdb_d5m_write(&i2c, 0x004, writedata);
    //Sets binning and skipping to 3 for rows and columns
    //writedata = 0b110011;
    writedata = 0;
    success &= trdb_d5m_write(&i2c, 0x022, writedata);
    success &= trdb_d5m_write(&i2c, 0x023, writedata);
    //Sets Horizontal Blank
    writedata = 0u;
    success &= trdb_d5m_write(&i2c, 0x005, writedata);

    //Test part
    success &= trdb_d5m_write(&i2c, 0x0A0, 0b111001);       // Vertical Monochrome bars
    success &= trdb_d5m_write(&i2c, 0x0A1, 1);              // Green
    success &= trdb_d5m_write(&i2c, 0x0A2, 7);              // Red
    success &= trdb_d5m_write(&i2c, 0x0A3, 7);              // Blue
    success &= trdb_d5m_write(&i2c, 0x0A4, 61);         // bar width
    success &= trdb_d5m_write(&i2c, 0x062, 0<<1);           //Disable BLC

    //Restart a new frame
    writedata = 1;
    success &= trdb_d5m_write(&i2c, 0x00B, writedata);
```

Figure 30: ***Camera Configuration for test pattern***

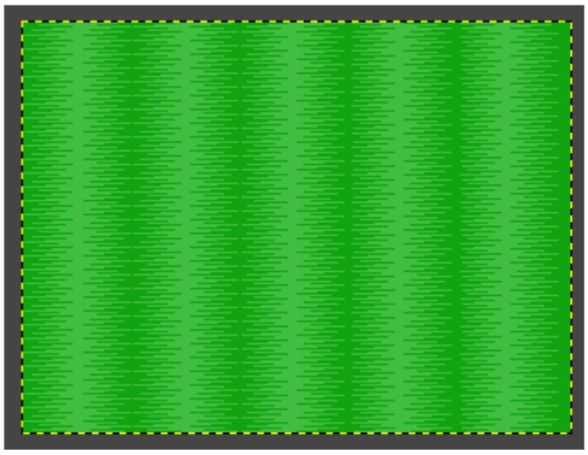
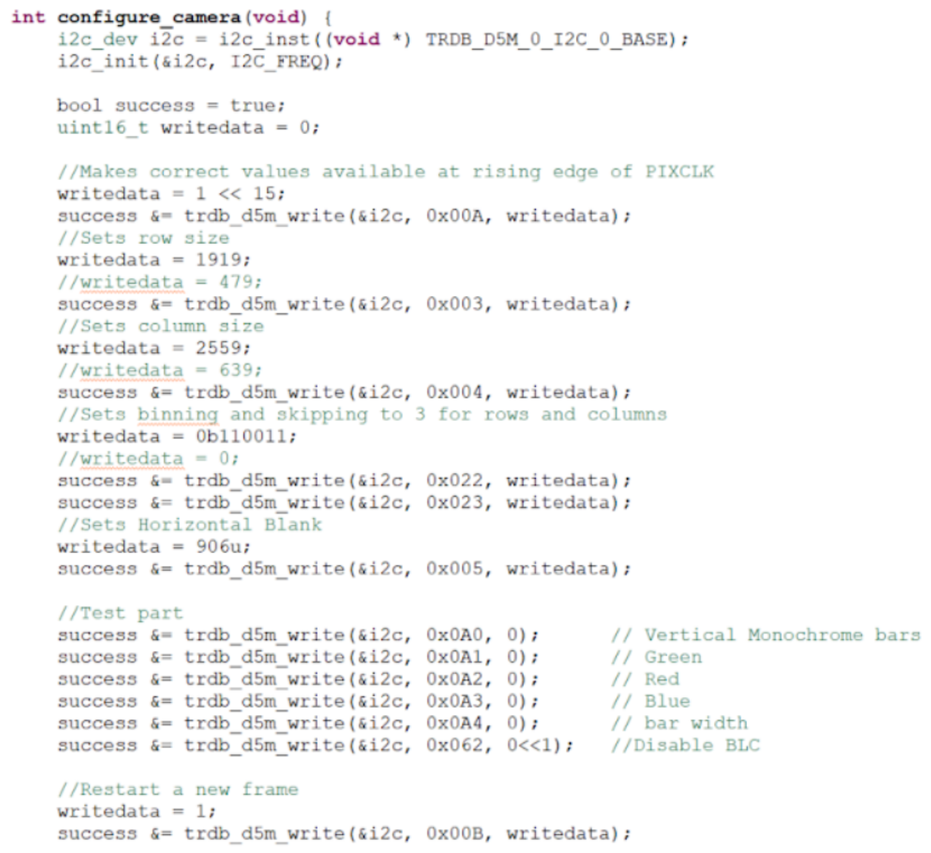


Figure 31: ***Monochrome vertical bars generated by the TRDB-D5M camera***

Then, we took a picture of a hand with the TRDB-D5M camera (see Figure 33) by using the following configuration for the camera :



```
int configure_camera(void) {
    i2c_dev i2c = i2c_inst((void *) TRDB_D5M_0_I2C_0_BASE);
    i2c_init(&i2c, I2C_FREQ);

    bool success = true;
    uint16_t writedata = 0;

    //Makes correct values available at rising edge of PIXCLK
    writedata = 1 << 15;
    success &= trdb_d5m_write(&i2c, 0x00A, writedata);
    //Sets row size
    writedata = 1919;
    //writedata = 479;
    success &= trdb_d5m_write(&i2c, 0x003, writedata);
    //Sets column size
    writedata = 2559;
    //writedata = 639;
    success &= trdb_d5m_write(&i2c, 0x004, writedata);
    //Sets binning and skipping to 3 for rows and columns
    writedata = 0b110011;
    //writedata = 0;
    success &= trdb_d5m_write(&i2c, 0x022, writedata);
    success &= trdb_d5m_write(&i2c, 0x023, writedata);
    //Sets Horizontal Blank
    writedata = 906u;
    success &= trdb_d5m_write(&i2c, 0x005, writedata);

    //Test part
    success &= trdb_d5m_write(&i2c, 0x0A0, 0);      // Vertical Monochrome bars
    success &= trdb_d5m_write(&i2c, 0x0A1, 0);      // Green
    success &= trdb_d5m_write(&i2c, 0x0A2, 0);      // Red
    success &= trdb_d5m_write(&i2c, 0x0A3, 0);      // Blue
    success &= trdb_d5m_write(&i2c, 0x0A4, 0);      // bar width
    success &= trdb_d5m_write(&i2c, 0x062, 0<<1);   //Disable BLC

    //Restart a new frame
    writedata = 1;
    success &= trdb_d5m_write(&i2c, 0x00B, writedata);
```

Figure 32: ***Camera Configuration for pictures***

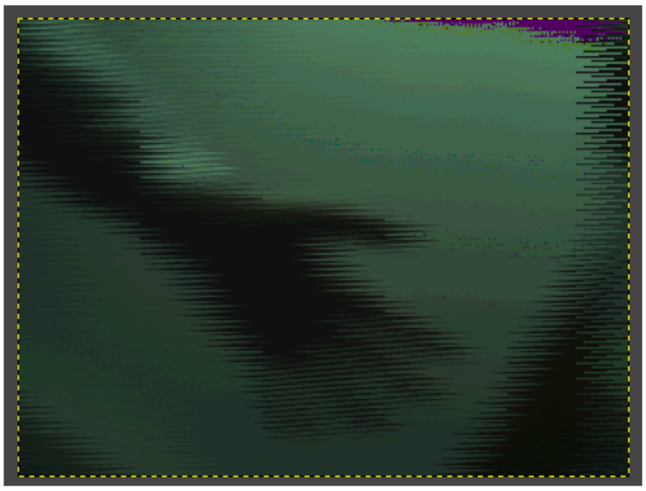


Figure 33: ***Image of a hand taken with the TRDB-D5M camera***

Finally, in Figure 34 one can see the full red, green and blue images generated with the cmos_sensor_output_generator.

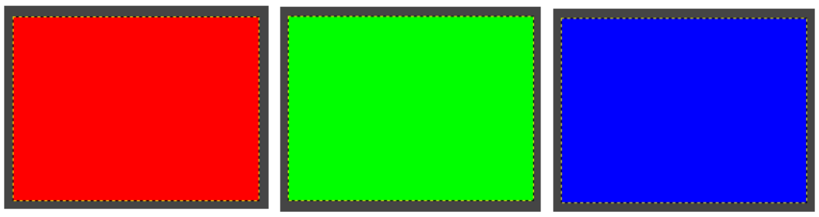


Figure 34: ***RGB***

4 Conclusion

To conclude, we were really eager to learn how to read pixels from a camera during this project. We tried to find the best strategies to accomplish this project. In the end, the results were successful: we were able to read the image from the camera, execute some operations on the pixels and finally send it in memory and verify the results. At this stage of the project, we could not manage to test in real life with the other group if the full system (camera + LCD) was working fine but the image that we sent in memory should be compliant with the LCD part. All will be tested for the presentation. We would like to thanks the professor and the assistants for their help throughout this project. This work was really challenging for us and their guidance was key to success.