Lab 4 - Custom IP implementation

TRDB-D5M camera - LT24 LCD

Augustin Mohr (289179)
Nicolas Cavedon (287879)
Iacopo Sprenger (284074)

Submitted on the 10 January 2021

Contents

Part I
LT24 LCD

1 Overview

In the previous lab, we did some theoretical work on a design for a custom IP component to interface directly an LT24 LCD with the memory. In this lab, we implemented it, and made some changes to make it work.
In the figure below, we can see the different sub-modules of our component. We have an Avalon Slave to communicate with the NIOS-2 processor by using software, an Avalon Master that will act as a Direct Memory Access to read the pixels stored in memory. Then we have the LCD Controller to interface with the LT24 LCD. And finally, to ensure the asynchronous transfer of the pixels between the Avalon Master and the LCD Controller, we use a FIFO buffer.

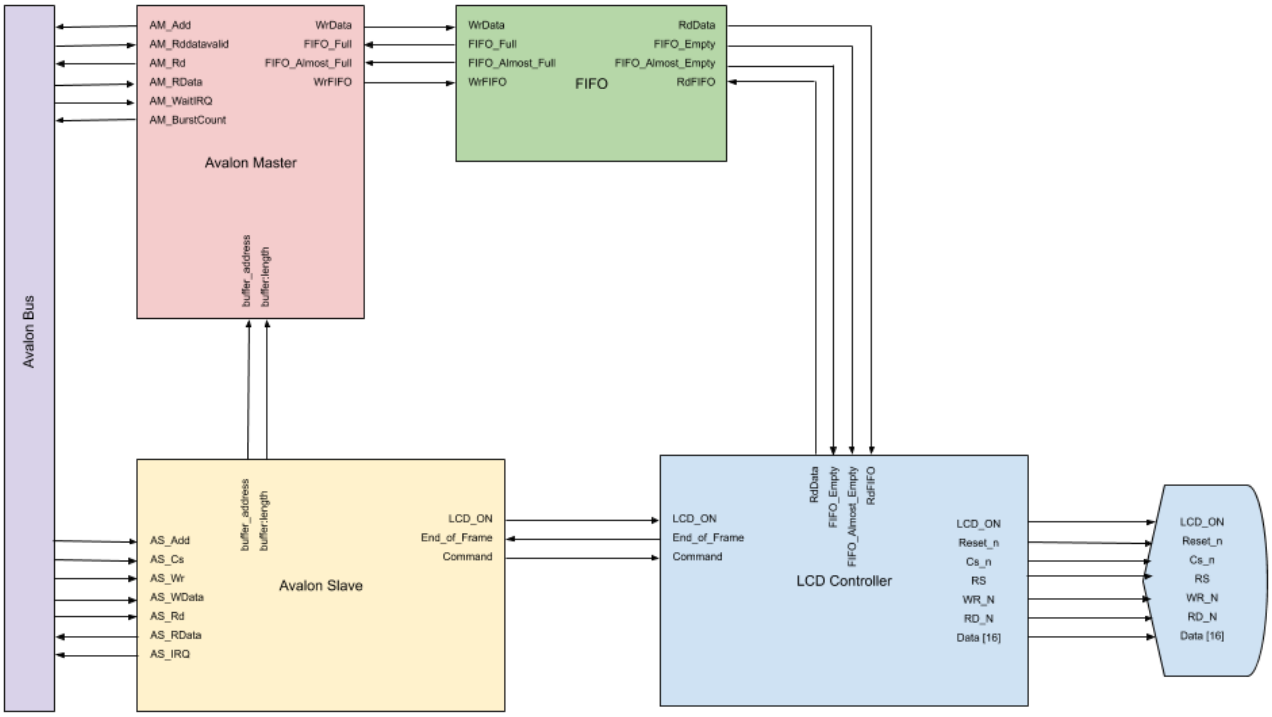


Figure 1: Custom IP Block Diagram

2 Avalon Slave

The Avalon Slave is used to communicate with the NIOS-2 processor. It's through the Avalon Slave that we are able to send commands and data to the LT24, as well as write/read in/from the different registers of our component.
We can see below in the register map the different registers that can be accessed by the NIOS-2 and can be programmed in the software implementation.

Name	Offset Interface	Offset uP	Data Width	Description
buffer_address	0	0	32bits	Base address of the frame stored in memory
buffer_length	1	4	32bits	Length of the frame stored in memory
LCD_command	2	8	8bits	Command sent to the ILI9341 by the NIOS-2
LCD_data	3	12	16bits	Data sent to the ILI9341 by the NIOS-2
burstcount	4	16	8bits	Size of each burst transfers for the Avalon Master read
finished	5	20	1bit	Read only : Status register indicating if last frame is finished
interrupt_enable	6	24	1bit	Enable the interrupt when the Master has finished reading a frame
LCD_on	7	28	1bit	Turn the LCD ON or OFF

We also use this sub-module to handle the interruption we send to the processor when the Avalon Master has finished reading a frame. Note that the interrupt is disabled by default.
Here is the vhdl code for the Avalon Slave write :

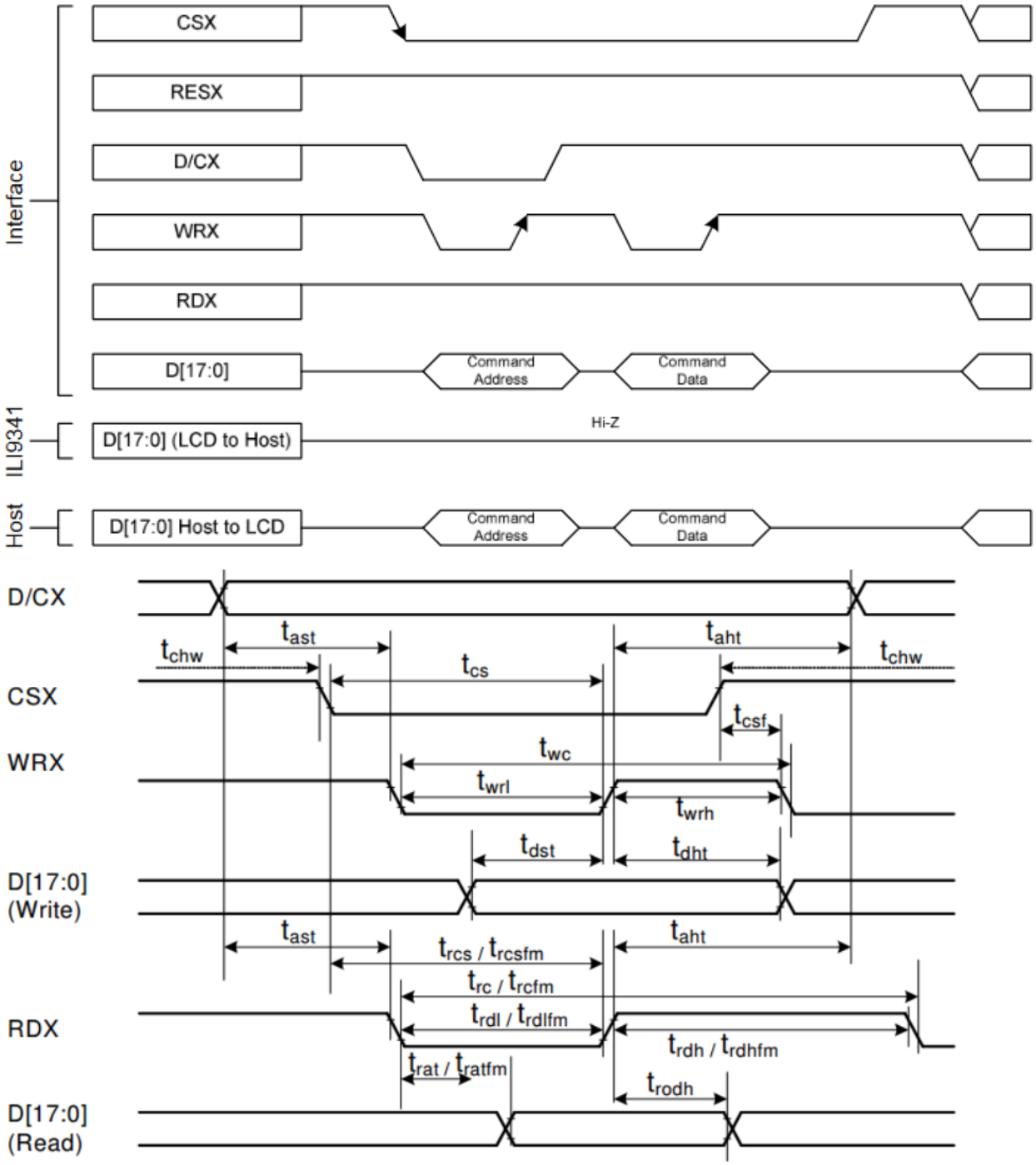
```
-- Avalon Slave write to registers
Avalon_slave_write : process(clk, nReset)
begin

  if nReset = '0' then
    buffer_address <= (others => '0');
    buffer_length  <= (others => '0');
    LCD_command  <= (others => '0');
    LCD_data  <= (others => '0');
    burst_count <= X"10";
    irq_buffer <= '0';
    interrupt_enable <= '0';
    LCDon <= '0';
  elsif rising_edge(clk) then
    if AS_CS = '1' and AS_write = '1' then
      case AS_address is
        when "0000" => buffer_address <= unsigned(AS_writedata);
        when "0001" => buffer_length  <= unsigned(AS_writedata);
        when "0010" => LCD_command    <= AS_writedata(7 downto 0);
        when "0011" => LCD_data     <= AS_writedata(15 downto 0);
        when "0100" => burst_count  <= unsigned(AS_writedata(7 downto 0));
        when "0101" => null; --Read only
        when "0110" => interrupt_enable <= AS_writedata(0);
        when "0111" => LCDon <= AS_writedata(0);
        when others => null;
      end case;
    end if;
   --Interrupt on finished state
    if finished_flag = '1' then
      buffer_length <= (others => '0');
      if interrupt_enable = '1' then
        AS_irq <= '1';
      end if;
    else
      AS_irq <= '0';
    end if;
  end if;
end process Avalon_slave_write;
```

Listing 1: vhdl code for Avalon_Slave_write.

3 LCD Controller

The LCD Controller sub-module is used to send the commands, data and pixels in the correct way and with the correct timings. We can see on the two images below the constraints we have to satisfy in order for the ILI9341 driver of the LT24 to recognize the commands, datas and pixels we will send.



WRX	twc	Write cycle	66	-	ns
	twrh	Write Control pulse H duration	15	-	ns
	twrl	Write Control pulse L duration	15	-	ns

Figure 2: Signals and Timings from ILI9341 Datasheet

To send the different information to the ILI9341, we use a simple Finite State Machine. Our LCD Controller FSM has 4 different states : idle, write_command, write_data and write_pixel. The FSM always goes back to the idle state when it finishes and operation as this state is the state that will direct the FSM to the next operation. The idle state looks like this :

```
when idle =>
        RESET_N <= '1';
        CS_N <= '1';
        D_C_N <= '1';        --Idle default state
        WR_N <= '1';
        RD_N <= '1';
        wait_LCD <= 0;
        DATA <= (others => 'Z');
        if AS_CS = '1' and AS_write = '1' and AS_address = "0010" then
          LCD_state <= write_command; --If a command has been sent to the AS by the processor
        elsif AS_CS = '1' and AS_write = '1' and AS_address = "0011" then
          LCD_state <= write_data; --If a data has been sent to the AS by the processor
        elsif FIFO_empty = '0' then
          LCD_state <= write_pixel;  --If there is no command and there are pixels to display
        end if;
```

Listing 2: idle state vhdl code in the LCD_controller FSM.

When a command is sent to the Avalon Slave by the processor, it makes our FSM go into write_command state. This state handles the procedure that must be used to send the command the Avalon Slave stored in the LCD command register. We use the wait_LCD counter as an internal FSM in order to apply sequentially the steps necessary to send the command. The write_command state looks like this :

```
when write_command =>
        wait_LCD <= wait_LCD + 1;
        case wait_LCD is
            when 0 =>
              CS_N <= '0';
              WR_N <= '0';
              D_C_N <= '0';
              DATA(15 downto 8) <= (others => '0'); --Set the data port with the command
              DATA(7 downto 0) <= LCD_command;
            when 1 =>
              WR_N <= '1';               --Write command to LCD

            when 2 =>
              D_C_N <= '1';
              DATA <= (others => 'Z');         --Negate the command on the data port
            when others =>
              LCD_state <= idle;
            end case;
```

Listing 3: write_command state vhdl code in the LCD_controller FSM.

When a data is sent to the Avalon Slave by the processor, the FSM goes in write_data state. This state is identical to the write_command state except for the D_C_N port. Here is the code for the write data state :

```
when write_data =>
        wait_LCD <= wait_LCD + 1;
        case wait_LCD is
            when 0 =>
              CS_N <= '0';
              WR_N <= '0';
              D_C_N <= '1';
              DATA <= LCD_data;
            when 1 =>
              WR_N <= '1';                        --Write parameter to LCD

            when 2 =>
              DATA <= (others => 'Z');               --Negate the data port
            when others =>
              LCD_state <= idle;
            end case;
```

Listing 4: write_data state vhdl code in the LCD_controller FSM.

The last state is used to send a pixel to the ILI9341. This states is identical to the write data state, the only difference is that it takes a few more steps because it starts by reading the FIFO. Here is the code for the write pixel state :

```
when write_pixel =>
        wait_LCD <= wait_LCD + 1;
```

```
        case wait_LCD is
            when 0 =>
              FIFO_read <= '1';                          --Request read from the FIFO
            when 1 =>
              CS_N <= '0';
              WR_N <= '0';
              D_C_N <= '1';
              DATA <= FIFO_readdata;                      --Read from FIFO
              FIFO_read <= '0';
            when 2 =>
              WR_N <= '1';                           -- Write to LCD
            when 3 =>
              DATA <= (others => 'Z');
              LCD_state <= idle;
            when others =>
              LCD_state <= idle;
            end case;
          when others =>
            LCD_state <= idle;
```

Listing 5: write_pixel state vhdl code in the LCD_controller FSM.

We are using a 50MHz clock, which means every step takes 20ns. Furthermore, our sequence lasts at least 4 cycles, which mean at least 80ns. With this we ensure that we respect the timings shown in Figure 2. As we can see in the image below, we are respecting the command sequence to send one command and its parameter to the ILI9341.

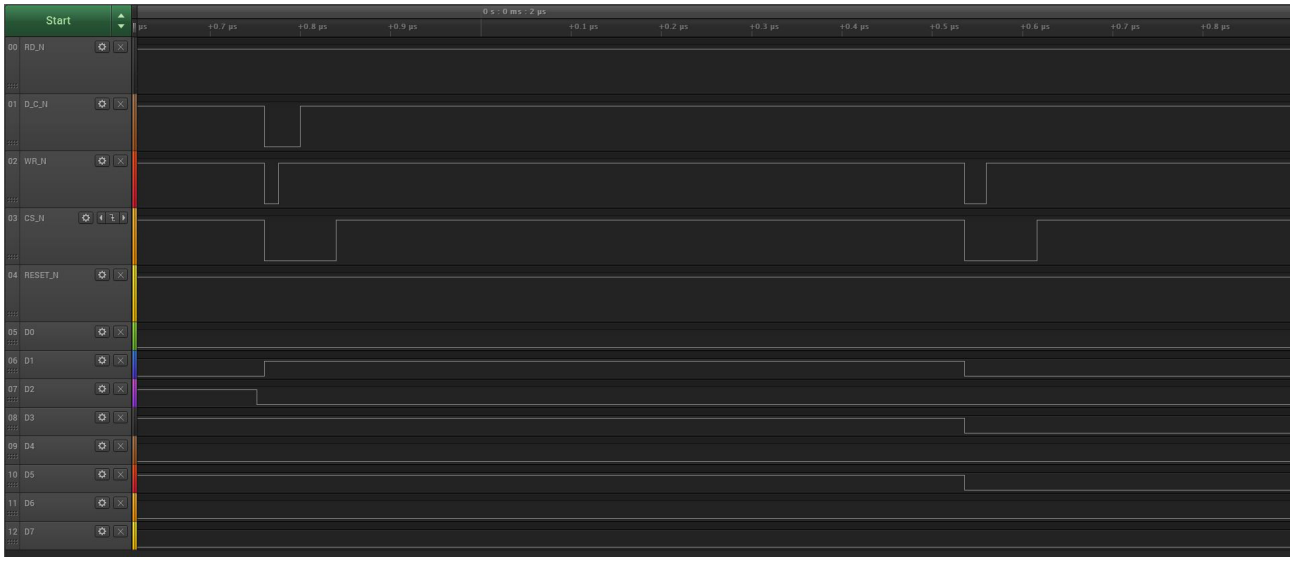


Figure 3: Screenshot of the Logic Analyser.

4 Avalon Master

The Avalon Master's role is to take information from the SDRAM and send it to the LCD Controller via a FIFO buffer. The Avalon Master FSM is detailed below:

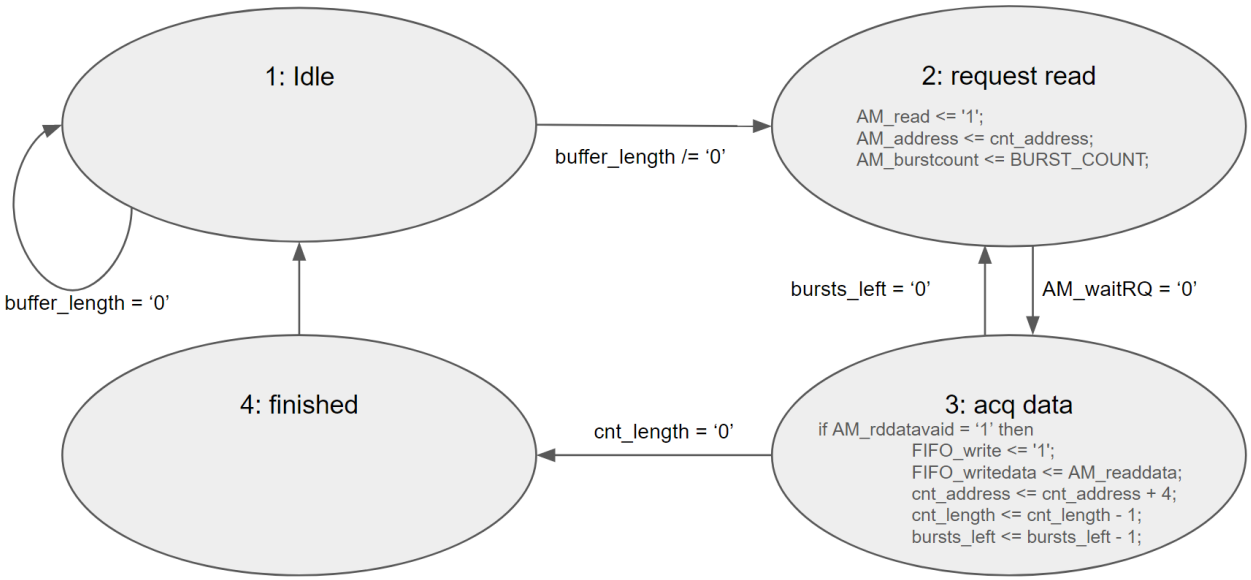


Figure 4: Avalon master finite state machine.

The default state of the Avalon master is the AM_idle state. In this state, it will wait to receive information on the buffer length from the processor through the Avalon slave. Once the information on the buffer length has been received, it will assume the information on the buffer address has also been updated accordingly. It is the responsibility of the processor to first send the buffer address, then the buffer length, which will trigger the read. Those will be stored in the cnt_address and cnt_length signals. The status "finished" will then be set to '0', to let the processor know it is currently reading information from the memory to the LCD_Controller.

```
when AM_idle => --Idle state, waiting to receive a buffer to fetch pixel from.
  if buffer_length /= X"0000_0000" then
    cnt_length <= buffer_length / 4; --Convert from number of bytes to number of transfers
    necessary
    cnt_address <= buffer_address;
    finished <= '0';
    AM_state <= AM_read_request;
  end if;
```

Listing 6: AM_idle state vhdl code in the Avalon master FSM.

The next state is AM_read_request. In this state, the Avalon master will send a request for a read to the address specified by cnt_address, which will be loaded into AM_address. AM_read will then be pulled to 1, and AM_burstcount will have the corresponding burst count, which can be specified by the processor. By default, the number of burst is set to 16. The read request will only be sent if there is enough space in the FIFO to write a full burst, otherwise it will wait for the LCD_Controller unit to read through the FIFO to free it up. If cnt_length, which represents the number of reads left to do from the SDRAM memory to the FIFO, is smaller than the burst count, the burst count will be set to that number. This means that the next read request will have the exact number of burst necessary to reach the end of the buffer.
Once AM_waitrq is pulled to 0 by the Avalon bus, signifying that the read can begin, the Avalon Master will proceed to the next state.

```
when AM_read_request => --Requesting a burst read from the memory through the Avalon Bus.
  if unsigned(FIFO_usedw) <= FIFO_size - burst_count and AM_rddatavalid = '0'then --Request
    the read only if there is enough room in the FIFO to receive it, and AM_rddatavalid = 0.
    AM_read <= '1';
    AM_address <= std_logic_vector(cnt_address);
    if cnt_length < burst_count then
      AM_burstcount <= std_logic_vector(cnt_length(7 downto 0));
      bursts_left <= cnt_length(7 downto 0);
    else
      AM_burstcount <= std_logic_vector(burst_count);
      bursts_left <= burst_count;
    end if;

    if AM_waitrq = '0' then
      AM_state <= AM_acq_data;
      AM_read <= '0';
      AM_address <= (others => '0');
      AM_burstcount <= (others => '0');
    end if;
  else
    AM_read <= '0';
  end if;
```

Listing 7: AM_read_request state vhdl code in the Avalon master FSM.

The next state is AM_acq_data. This part takes care of the burst read. The data can be read whenever AM_rddatavalid is set to 1. At that point, FIFO_write is set to 1, and the data seen on AM_readdata will be written to the FIFO via the FIFO_writedata line. cnt_address and cnt_length are then incremented and decremented respectively. burst_left is also decremented, to keep track of how many bursts are left.
Once the last burst read has been sent, the Avalon master will switch back to the AM_read_request state, and request another transfer. If we have reached the end of the buffer, the Avalon master will instead go to the AM_finished state.

```
when AM_acq_data => --Reading each valid data of the burst sent on the Avalon Bus.
  if AM_rddatavalid = '1' then
    FIFO_write <= '1';
    FIFO_writedata <= AM_readdata;
    cnt_address <= cnt_address + 1;
    cnt_length <= cnt_length - 1;
    bursts_left <= bursts_left - 1;
    if cnt_length <= 1 then --Checking End of Buffer (End of Frame) -> Finished.
      AM_state <= AM_finished;
      finished <= '1';
      finished_flag <= '1';
    elsif bursts_left <= 1 then --Checking End of Burst -> Request another burst read.
      AM_state <= AM_read_request;
    end if;
  end if;
```

Listing 8: AM_acq_data state vhdl code in the Avalon master FSM.

The last state of our master is AM_finished. This state acts as a buffer state, during which the buffer_length and the buffer_address will be reset to 0. Just before entering that state, the finished flag and the finish state will be set to 1. The aim of the finished flag is to trigger the interruption for one cycle, although we do not intend to use that feature during this project.

```
when AM_finished => --Going back to idle.
  finished_flag <= '0';
  AM_state <= AM_idle;
```

Listing 9: AM_finished state vhdl code in the Avalon master FSM.

Below is an example of the Avalon master performing a burst read. This image has been taken with the signal tap functionality of Quartus.

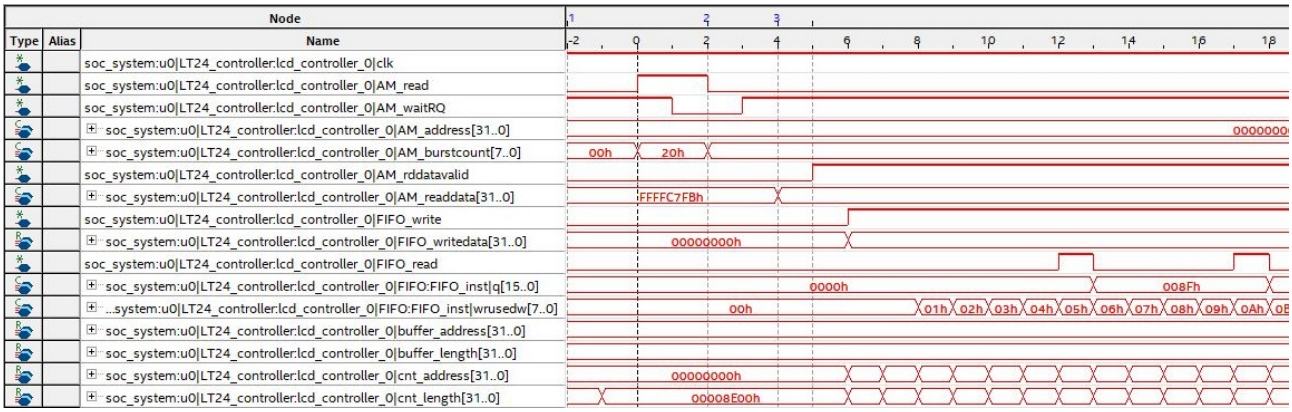


Figure 5: Avalon master burst read, taken with the signal tap.

5 FIFO buffer

In order to send the pixels from the Avalon Master to the LCD Controller, we use an asynchronous FIFO. Our FIFO has an input width of 32 bits, while having an output width of only 16 bits, which is the size of one pixel. This allows us to have the Avalon Master reading in bursts on a 32 bit wide Avalon Bus, as long as there is enough space left in the FIFO. To examine this condition, we use the used words (usedw) output of the FIFO to see if there is enough space to do one burst read before doing it. On the output size of the FIFO, our LCD Controller displays the pixel whenever it can, in other words, as long as the FIFO is not empty, the LCD Controller empties it by displaying the pixel on the LT24.

6 Software Implementation

6.1 LCD initialisation

In order to use our LCD, the very first thing we do is turn it on. We then need to configure it by sending a set of commands to the LCD_Controller. All of this is done in the LCD_init() function, in which we initialize the internal registers of our LCD. The basis of this function is directly taken from the LT24 painter demonstration. Some commands require parameters, which will be sent as data to the LCD directly after the command has been sent. We will not go through all the commands in LCD_init() as there are many, but we will go over the commands that set our LCD to be in landscape mode, which we have defined ourselves.

```
1  LCD_WR_REG(0x0036); // Memory access control (MADCTL B5 = 1)
2      LCD_WR_DATA(0x0028); // MY MX MV ML_BGR MH 0 0 -> 0b0010 1000
3
4  LCD_WR_REG(0x002A); // Column Address Set
5      LCD_WR_DATA(0x0000); // SC0-7
6      LCD_WR_DATA(0x0000); // SC8-15 -> 0x0000
7      LCD_WR_DATA(0x0001); // EC0-7
8      LCD_WR_DATA(0x003F); // EC8-15 -> 0x013F
9
10 LCD_WR_REG(0x002B); // Page Address Set
11     LCD_WR_DATA(0x0000); // SP0-7
12     LCD_WR_DATA(0x0000); // SP8-15 -> 0x0000
13     LCD_WR_DATA(0x0000); // EP0-7
14     LCD_WR_DATA(0x00EF); // EP8-15 -> 0x00EF
```

Listing 10: Memory access control, Column address set and Page address set commands in the LCD_init() function.

First, the Memory Access Control register is set to 0x28. This sets the MV and BGR bits to 1, which sets the landscape mode for the LCD and defines the pixel encoding as RGB. Then, the Column Address Set and the Page Address Set registers are defined. Those registers define the start and end of the column and page addresses of our LCD. They are set to 0 - 320 and 0 - 240 respectively.

6.2 Displaying an image in PPM format

To send an image in ppm format to the SDRAM, the upload_image() function is used. The code for this function is described below:

```
void upload_image(void) {
    char* filename = "/mnt/host/image.ppm";
    int r1, g1, b1, r2, g2, b2, r, w, h, maxComp1, maxComp2, maxComp3, i = 0;
    char format[3];

    FILE *file = fopen(filename, "r");
    if (!file) {
     printf("Error: could not open \"%s\" for reading\n", filename);
     return;
    }
    printf("Reading file...\n");

    fscanf(file, "%2s", &format);
    fscanf(file, "%d", &w);
    fscanf(file, "%d", &h);
    fscanf(file, "%d", &maxComp1);
    fscanf(file, "%d", &maxComp2);
    fscanf(file, "%d", &maxComp3);
    printf("format: %2s\nsize: %d x %d \n%d\n",format, w, h, maxComp1, maxComp2, maxComp3);
    printf("Sending info to SDRAM...\n");

    for(i = 0; i < BUFFER_LENGTH; i = i + 4) {
        r1=((fgetc(file) >> 3) & 0x1f) << 11;
        g1=((fgetc(file) >> 2) & 0x3f) << 5;
        b1=(fgetc(file) >> 3) & 0x1f;
        r2=((fgetc(file) >> 3) & 0x1f) << 11;
        g2=((fgetc(file) >> 2) & 0x3f) << 5;
        b2=(fgetc(file) >> 3) & 0x1f;
        r = (int)(r1  + g1 + b1 + ((r2 + g2 + b2) << 16));
        MEM_WR(BUFFER1_OFFSET + i, r);
    }
    fclose(file);
    printf("Sent image to memory after %d iterations\n", i);

    BUFF_ADD_WR(BUFFER1_OFFSET); // buffer address
    BUFF_LEN_WR(BUFFER_LENGTH); // buffer length
}
```

Listing 11: upload_image() function.

The reading of the image file by the processor is done using the altera_hostfs package, which allows it to access files from the computer's hard disk. Once the file has been correctly opened, we read the first 4 lines and print them out in the console. Those lines are the header of the ppm file, and contain information about the image, like the format and the size. The function then enters a loop, in which it extracts the pixels from the image and sends them to the SDRAM.
Because the pixels in a ppm image are encoded in RGB888, we must convert it to the RGB565 format. Since we are sending 32 bits of information at a time, we also need to combine two pixels together. This is done using bitwise operations on the corresponding colours, and merging them together to form a pair of pixel readable by the LCD.
Once the image has been sent to the SDRAM, we close the file and send the information on the buffer address and length where the image is stored. The image will then be displayed on the LCD.
Note that only the code for a P6 (regular ppm) format has been detailed here. In the code provided with this assignment, modifications have been made to support the sending of an image in a P3 (plain ppm) format.

7 Results

We are now able to display a .ppm image stored in the memory of our board using Direct Memory Access. See example below :

Figure 6: image.ppm displayed on the LCD.

8 Conclusion

To conclude this part of the lab, we can see that the objective has been achieved. We are able to display images to the LT24 using our custom IP component.

Part II
Camera

9 Overview

In the previous lab, we have designed a custom IP to acquire and store the pixels of an image from the TRDB-D5M camera. In this lab, we will implement the designed IP. The camera module provides us with pixels sequentially line by line. The camera sees the world through a Bayer color filter and we receive single color pixels alternating between the three primary colors. In order to construct a color image, we assemble the pixels four by four. Once the color pixels are reconstructed, we transmit them through an Avalon master interface to the memory.

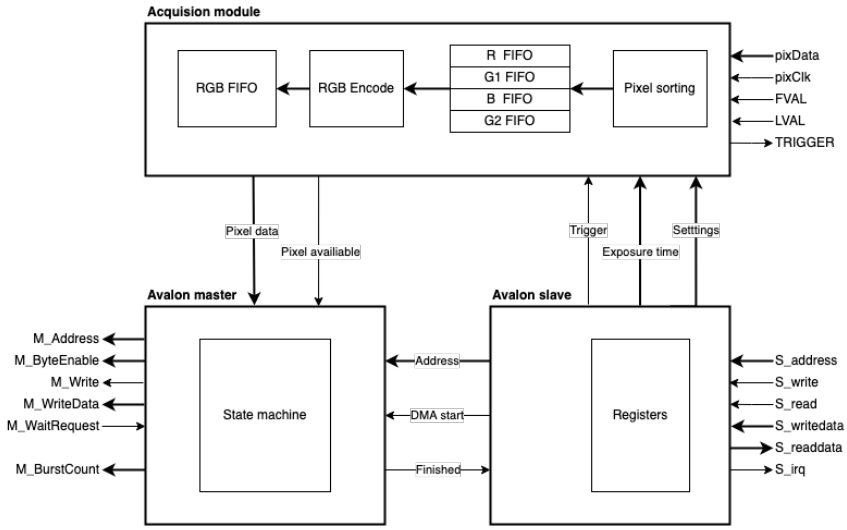


Figure 7: Block diagram of our IP showing the three parts and how they interact with each other.

10 Acquisition module

10.1 Trigger signal

We decided to use the camera module in bulb exposure mode, this means we have to provide a trigger pulse that has the length of the desired exposure.

The user can control the exposure time by setting the exposure time in the control register. In order to get a reasonable exposure length, this value is multiplied by $2^8 = 256$ to get the exposure number. As the exposure value is stored on 16 bits, the exposure number will be at most $(2^{16} - 1) * 2^8 = 1677696$ clock cycles with an incremental step of 256 clock cycles. This gives us a maximal exposure time of $0.33s$ and a minimal exposure time of $0.00001s$ with a clock frequency of $50MHz$.

```
exposure_number <= unsigned(control_reg(17 downto 2) & "00000000");
```

The `exposure_active` signal is set as soon as a write operation is done on the `control_reg` with the trigger bit set. Whenever the `exposure_active` is set, we count the clock cycles until `exposure_counter` reaches `exposure_number`. At this moment, we clear the `exposure_active`.

```
if exposure_active = '1' then
  exposure_counter <= exposure_counter + 1;
  restart <= '0';
  if exposure_counter = exposure_number then
    exposure_active <= '0';
    exposure_counter <= (others => '0');
  end if;
end if;
```

Listing 12: Located in a sequential block driven by the main clock, this describes the exposure counter. Once the counter reaches the exposure number, we clear the exposure active signal.

The `exposure_active` signal is connected to the trigger pin of the camera through an inverter as the camera trigger is active low.

10.2 Pixel sorting

Because the sensor array is covered by a bayer color filter, the pixels sent by the camera need to be sorted according to their color upon arrival. We know the order in which the pixels will arrive. On the even lines, the pixels will alternate between green and red. On the odd lines, the pixels will alternate between blue and green.

In order to sort the pixels, we keep track of which pixel on the line is arriving as well as which line in the frame using two counters. To know how many pixels there are in a line, we get the width from the settings register and to know how many lines there are in one frame, we get the height from the settings register.

```
elsif rising_edge(C_pixclk) then
  if pix_valid = '1' then
    pix_counter <= pix_counter + 1;
    if pix_counter = unsigned(settings_reg(11 downto 0))-1 then
      line_counter <= line_counter + 1;
      pix_counter <= (others => '0');
      if line_counter = unsigned(settings_reg(23 downto 12))-1 then
        line_counter <= (others => '0');
      end if;
    end if;
  end if;
end if;
```

Listing 13: The counting logic for the pixel and line counters. We increment the `pix_counter` when a new pixel arrives and we increment the `line_counter` when an entire line has arrived.

We use the least significant bit of each of these two counters to drive the write signals of the individual color FIFO buffers. The pixel data bus is connected to the four FIFO buffers, but only one of the buffer's write signal can be active such that the incoming pixel data gets stored in the right buffer.

```
mux_input(1) <= pix_counter(0);
mux_input(0) <= line_counter(0);

g1_write <= '1' when mux_input = "00" and pix_valid = '1' else '0';
r_write <= '1' when mux_input = "10" and pix_valid = '1' else '0';
b_write <= '1' when mux_input = "01" and pix_valid = '1' else '0';
g2_write <= '1' when mux_input = "11" and pix_valid = '1' else '0';
```

Listing 14: How the write signals are generated for the four individual color FIFO buffers.

10.3 Buffers

Our IP uses five FIFO buffers. The first four are the individual color FIFO buffers that store the arriving newly sorted pixels. These buffers make the connection between the pixel clock space and the main clock space. The input is driven by the pixel clock coming from the camera module and the output is driven by the main clock. This bridge is important as the pixel clock is not guaranteed to be synchronized with the main clock.

```
fifo_r : entity work.fifo_12 PORT MAP (
  aclr   => reset,
  data   => C_pixdata,
  rdclk  => clock,
  rdreq  => color_available,
  wrclk  => C_pixclk,
  wrreq  => r_write,
  q  => color_r,
  rdempty  => r_empty,
  wrfull   => open
);
```

Listing 15: Example connection of one of the four individual color FIFO buffers.

The last buffer is used to store the RGB565 pixels once they are computed. The buffer also serves to accumulate pixels awaiting a burst transfer. This fifth FIFO buffer works entirely in the main clock space and thus only has one clock input.
The input comes directly from the color assembler and the write signal is set as soon as one of each color is available in the single color FIFO buffers.

```
fifo_rgb : entity work.fifo_16 PORT MAP (
    aclr   => reset,
    clock  => clock,
    data   => color_rgb,
    rdreq  => read_pix,
    wrreq  => color_available,
    empty  => pixel_empty,
    full   => open,
    q  => rgb_out
  );
```

Listing 16: Example connection of one of the four individual color FIFO buffers.Connection of the RGB FIFO buffer.

All our buffers have an asynchronous clear that is connected to the `reset` signal. This signal is connected to the global reset network as well as the `restart` signal which is generated for the first clock cycle of the trigger. This allows to flush the buffers each time we capture a new image to avoid having some residual data from earlier images in the FIFO buffers.
Additionally, the buffers must be configured in the show ahead mode so that our combinational logic works.

10.4 RGB pixels assembly

The assembly of the individual pixels into the RGB565 format is done in a purely combinational manner. The outputs of the four individual color FIFO buffers are connected through the color assembly logic to the input of the RGB FIFO buffer. The read signals for the four FIFO buffers and the write signal for the RGB FIFO buffer are generated from the empty signals of the four FIFO buffers. For this reason, it is very important that the data is already available on the output before the read signal is set such that an RGB565 pixel is already available when the write signal to the RGB buffer is set.

```
color_available <= ((not r_empty) and (not g1_empty) and (not b_empty) and (not g2_empty));

color_g <= std_logic_vector(unsigned('0' & color_g1)+unsigned('0' & color_g2));

color_rgb(15 downto 11) <= color_r(11 downto 7);
color_rgb(10 downto 5) <= color_g(12 downto 7);
color_rgb(4 downto 0) <= color_b(11 downto 7);
```

Listing 17: Combinational logic to assemble four individual color pixels into one RGB565 pixel.

11 Avalon Interface

11.1 Registers

Our IP can be configured and controlled using memory mapped registers.
All the register access is done in a sequential block driven by the main clock. Whenever the user sets the `S_write` signal, we copy the value of the `S_writedata` bus to the register selected by the address bus. When a write access to the control register is performed, we set the `exposure_active` and the `restart` signal if the trigger bit is set. The `exposure_active` signal will lead to an image acquisition and the `restart` signal will flush the FIFO buffers and reset the counters to discard any residual data in the system.

When writing to the settings register, there is a condition that ensures that the burst length is at least one.

```
if S_write = '1' then
  case S_address is
    when "00" =>
      control_reg <= S_writedata;
      if S_writedata(0) = '1' then
        restart <= '1';
        exposure_active <= '1';
      end if;
    when "01" =>
      settings_reg <= S_writedata;
      if unsigned(S_writedata(31 downto 24)) = 0 then
        settings_reg(24) <= '1';  -- minimum burst length is 1
      end if;
    when "10" =>
      address_reg <= S_writedata;
    when others => null;
  end case;
```

Listing 18: Located in a sequential block driven by the main clock, this allows writing to the registers.

Reading from the registers works in a similar way but this time, we write from the registers to the `S_readdata` bus.

```
elsif S_read = '1' then
  case S_address is
    when "00" =>
      S_readdata <= control_reg;
    when "01" =>
      S_readdata <= settings_reg;
    when "10" =>
      S_readdata <= address_reg;
    when others => null;
  end case;
end if;
```

Listing 19: Located in a sequential block driven by the main clock, this allows reading from the registers.

11.2 Register map

Control [0x00] [RW]					
31	30	29:18	17:2	1	0
IF	IE	Reserved	Exposure time	Start	Trigger

Bits	Name	Description
0	Trigger	Trigger a frame acquistion
1	Start	Enable DMA transfer
17:2	Exposure time	Length of the trigger signal
30	IE	DMA transfer complete interrupt enable
31	IF	DMA transfer finished interrupt flag

Settings [0x01] [RW]		
31:24	23:12	11:0
Burst length	Height	Width

Bits	Name	Description
11:0	Width	Expected frame width
23:12	Height	Expected frame height
31:24	Burst length	Burst transfer length

Address [0x02] [RW]
31:0
Destination address

Bits	Name	Description
31:0	Address	Destination address for the DMA transfer

11.3 Interrupts

The interrupt request line will be high as long as the interrupt flag is set and the interrupts are enabled. This allows us to control whether the interrupt is active by setting or clearing the interrupt enable in the control register. The interrupt flag of the control register can be set by the IP and must be cleared by the user when the interrupt was processed.

```
S_irq <= control_reg(31) and control_reg(30);
```

In order to control the interrupt flag, the Avalon master process of our IP will set the `finished` signal once a frame has been successfully transmitted to memory. The Avalon slave process will set the interrupt flag as soon as the `finished` signal is set.

```
if finished = '1' then
  control_reg(31) <= '1';
end if;
```

The interrupt flag can only be cleared by the user through the control register. It is supposed to be cleared when the user handles the interrupt. However, the interrupt flag will be set even if the interrupts are disabled. This allows the user to check whether the job is done by polling the control register.

11.4 Avalon Master

The avalon master interface was the main change in design since the lab 3. We initially decided to send one burst of data for the entire frame. However, this is a very bad idea as it will lock the Avalon bus for the entire transfer which is really not desirable.
Instead, we have now added a parameter in the settings register. The burst length represents the size of each burst transfers we will make.

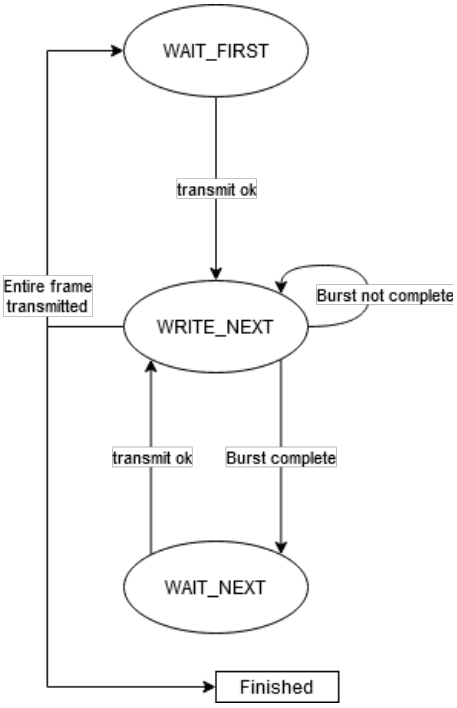


Figure 8: Diagram of our Avalon master finite state machine behaviour.

The finite state machine starts in the `WAIT_FIRST` state. In this state, we will wait for the first burst of pixels to be available in the rgb fifo. When the signal `transmit_ok` is set, this means there are the right number of pixels available and the DMA start bit is set in the control register.

```
when WAIT_FIRST =>
  if transmit_ok = '1' then
    master_state <= WRITE_NEXT;
    M_burstcount <= settings_reg(31 downto 24);
    M_address <= address_reg;
    address_bfr <= unsigned(address_reg);
    read_pix <= '1';
    out_bfr <= rgb_out;
    M_write <= '1';
    send_counter <= to_unsigned(1, send_counter'length);
    burst_counter <= to_unsigned(1, burst_counter'length);
  end if;
```

Listing 20: Description of the `WAIT_FIRST` state in the Avalon master finite state machine.

We can see that in this first state, we write the first address and the first burst count. We also write the first pixel to the Avalon master data bus. Then we initialize the burst counter and the send counters, these will allow us to detect the end of a burst and the end of a frame transfer.

The next state is the `WRITE_NEXT` state. In this state, we will write an entire burst of the correct length to the avalon master data bus. We assume that the rgb FIFO buffer is filled enough as this was checked during the previous state. We wait in this state until the waitrequest is cleared and then there are three options.

If the send counter is reached, this means an entire frame was transmitted and thus we trigger the finished signal and get back to the initial state.

If the burst counter is reached, this means the current burst is complete and thus we increment the address buffer aand move to the `WAIT_NEXT` state.

If none of the above is true, we transmit the next pixel in the current burst. And increment the counters.

```
when WRITE_NEXT =>
  read_pix <= '0';
  if M_waitrequest = '0' then
    M_write <= '0';
    if send_counter = send_number then
      master_state <= WAIT_FIRST;
      finished <= '1';
    elsif burst_counter = unsigned(settings_reg(31 downto 24)) then
      master_state <= WAIT_NEXT;
      address_bfr <= address_bfr + unsigned(settings_reg(31 downto 24) & "0"); --double
    because we set word addresses
    else
      read_pix <= '1';
      M_write <= '1';
      send_counter <= send_counter + 1;
      burst_counter <= burst_counter + 1;
      out_bfr <= rgb_out;
    end if;
  end if;
```

Listing 21: Description of the `WRITE_NEXT` state in the Avalon master finite state machine.

The last state in our finite state machine is the `WAIT_NEXT` state. the machine will be in this state when it waits for enough pixels to arrive in order to start a burst transfer. So aas soon as `transmit_ok` is set, we write the address and the burst count as well as the first pixel to the avalon master. Then we initialize the burst counter and we move to the `WRITE_NEXT` state.

```
when WAIT_NEXT =>
  if transmit_ok = '1' then
    master_state <= WRITE_NEXT;
    read_pix <= '1';
    M_write <= '1';
    send_counter <= send_counter + 1;
    burst_counter <= to_unsigned(1, burst_counter'length);
    M_burstcount <= settings_reg(31 downto 24); -- set the right burstcount
    M_address <= std_logic_vector(address_bfr);
    out_bfr <= rgb_out;
  end if;
```

Listing 22: Description of the `WAIT_NEXT` state in the Avalon master finite state machine.

12 Software Implementation

12.1 Camera configuration

At the beginning of the software, we start by configuring the camera module using the I^2C interface. We use the library provided with the I^2C IP as well as the `trdb_d5m_write` and `trdb_d5m_read` functions to send configuration commands to the camera.

We send configuration commands to get an output image of 640 x 480 pixels. We also want an inverted pixel clock so that the pixel values are valid on the rising edge. Finally, we want the camera to be in snapshot mode with bulb exposure and an electronic rolling shutter. Snapshot because we take the pictures one by one, bulb exposure so the exposure time is controlled by the trigger signal and electronic rolling shutter because we do not have a mechanical shutter and wish to have the same exposure time on each pixel.

We also configure skip and bin at 3 for both axis, which means the camera will average an area to get the lower resolution instead of taking just a smaller rectangle on the sensor.

```
bool camera_initial_setup(camera_dev * cam) {
  bool success = true;
  success &= trdb_d5m_write(&cam->i2c, 0x00D, 0x0001);
  success &= trdb_d5m_write(&cam->i2c, 0x00D, 0x0000);
  success &= trdb_d5m_write(&cam->i2c, 0x001, 54);
  success &= trdb_d5m_write(&cam->i2c, 0x002, 16);
  success &= trdb_d5m_write(&cam->i2c, 0x003, 1919);
  success &= trdb_d5m_write(&cam->i2c, 0x004, 2559);
  success &= trdb_d5m_write(&cam->i2c, 0x022, 0x0033);
  success &= trdb_d5m_write(&cam->i2c, 0x023, 0x0033);
  success &= trdb_d5m_write(&cam->i2c, 0x00A, 0x0080);
  success &= trdb_d5m_write(&cam->i2c, 0x01E, 0x0140);
```

```
13   return success;
14 }
```

Listing 23: Sequence of I^2C write operations to configure the TRDB D5M camera.

12.2 Module configuration

In order to configure our IP, we have written a few C functions that simplyfy this task.

```
1 void camera_settings(camera_dev * cam, uint8_t burst_size, uint16_t width, uint16_t height) {
2   uint32_t settings = (burst_size << CAMERA_SETTINGS_BURST) | (height <<
      CAMERA_SETTINGS_HEIGHT) | (width << CAMERA_SETTINGS_WIDTH);
3   IOWR_32DIRECT(cam->base, CAMERA_SETTINGS_BASE, settings);
4 }
```

Listing 24: Settings register access to configure the size of the image

In the address register we place the address at which we want the Avalon master to store the image. We decided to use the beginning of the sdram address space.

```
1 void camera_address(camera_dev * cam, uint32_t address) {
2   IOWR_32DIRECT(cam->base, CAMERA_ADDRESS_BASE, address);
3 }
```

Listing 25: Address register access to configure the destination address for the DMA write operations.

Finally, in the control register, we set the exposure time and the start flags. We put the maximal exposure time for now.

```
1 void camera_capture(camera_dev * cam, uint16_t exposure) {
2   uint32_t control = (exposure << CAMERA_CONTROL_EXP) | (1 << CAMERA_CONTROL_START) | (1 <<
      CAMERA_CONTROL_TRIGGER);
3   IOWR_32DIRECT(cam->base, 0, control);
4 }
```

Listing 26: Control register access to configure and start the image acquisition.

12.3 Image saving

In order to inspect the captured image, we use the altera hostfs debugging functions to store the image as a ppm file on the connected computer. We start by writing the very simple ppm header. Then, we iterate over each pixel, decode them and write them to the computer using the altera hostfs interface.

```
1  void camera_save_ppm(camera_dev * cam, uint32_t addr) {
2    FILE * f = fopen("/mnt/host/image.ppm", "w");
3    fprintf(f, "P3\n320 240\n255 255 255\n");
4    for(uint16_t i = 0; i < 240; i += 1) {
5      for(uint16_t j = 0; j < 320; j+= 1) {
6        uint16_t color = IORD_16DIRECT(addr+sizeof(uint16_t)*(j+320*i), 0);
7        uint8_t red =   (color & 0b1111100000000000)>>8;
8        uint8_t green = (color & 0b0000011111100000)>>3;
9        uint8_t blue =  (color & 0b0000000000011111)<<3;
10       //printf("%x color: %x | %d %d %d \n", HPS_0_BRIDGES_BASE+sizeof(uint16_t)*(j+320*i),
     color, red, green, blue);
11       fprintf(f, "%d %d %d \n", red, green, blue);
12     }
13     printf("line %d\n", i);
14   }
15   fclose(f);
16 }
```

Listing 27: The code to store the image from the sdram to a connected computer. It is important to take note that this only works in debug mode.

13 Results

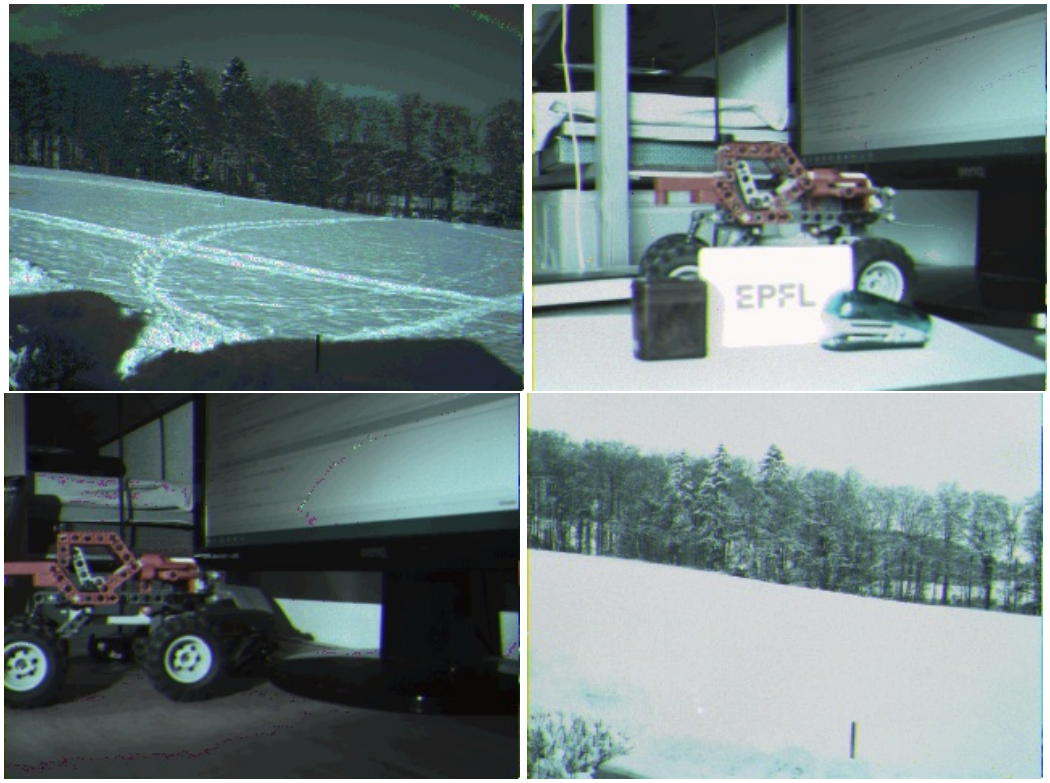


Figure 9: Four pictures taken with the camera and our IP implementation.

On the figure 9 we can see some pictures taken with our system. Sadly, the colors are not totally right. This is due to the fact that we so not perform any color correction after capturing the image. Yet a CMOS sensor requires a lot of post processing to yield a decent image.

In an effort to increase the image quality, we still provide functions to increase the individual gain of the three colors, but just meddling with those did not improve the colors.

14 Conclusion

The purpose of this part of the mini project was to capture a picture with the camera and store it to memory. We managed to do this as we showed with the images transmitted to the computer. However the poor color quality of the images is not ideal and we would have liked to try to implement some simple color post processing. Sadly, we did not have the time to do that.

Part III
Combined camera with display

15 Communication

In this last part, we will combine the systems of the last two part in order to display directly the image from the camera on the LCD. In order to do that, we implemented two options : using one or two buffers. As using only one buffer was clearly enough performance-wise and that our tests with double buffering resulted on strange behavior from the memory, we decided to use the single buffering option. The process is simple : The camera writes the image in the buffer and when it's finished, the LCD reads and displays it, then the camera starts writing again. This sequential access allow us for a smaller frame rate than a double buffering method where the two components would work in parallel and then swap the buffers when both are done. However, even with the sequential one buffer method, we obtained a very high frame rate, so we added a small waiting time between to frames. This allows us to have a better stability with the memory.

The C code for this part can be read on the next page.

16 Results

The combined implementation works very well, and seems really stable as it managed to display the video feed from the camera on the LCD for several minutes without any problem. A video demonstration can be found by clicking this link.

17 Code

```
int main()
{
  printf("start\n");

  //framebuffer initialisation
  uint32_t fb[] = {FB0, FB1};
  uint16_t exposure = 0x00ff;
  uint8_t ord = 0;

  //camera initialisaion
  camera_dev camera;
  camera_create(&camera, I2C_0_BASE, CAMERA_0_BASE);
  camera_initial_setup(&camera);
  camera_settings(&camera, BURST_LEN, WIDTH*2, HEIGHT*2);
  camera_address(&camera, fb[ord]);
  camera_red_gain(&camera, 8, 1, 0);
  camera_green_gain(&camera, 8, 1, 0);
  camera_blue_gain(&camera, 8, 1, 0);

  //display initialisation
  LCD_Init();
  BURST_COUNT_WR(BURST_LEN);
  LCD_Clear(0x0000);

  //first picture
  camera_capture(&camera, exposure);
  while(!camera_is_finished(&camera));

  while(1) {
#ifdef DOUBLE_BUFFERING
    ord = !ord;
    printf("swapping_buffers\n");

    display_buffer_addr(fb[!ord]);
    display_buffer_len(BUFFER_LEN); //launch display

    camera_address(&camera, fb[ord]);
    camera_capture(&camera, exposure);

    while(!camera_is_finished(&camera));
    while(!display_is_finished());
#else
    camera_address(&camera, 0);
    camera_capture(&camera, exposure);
    while(!camera_is_finished(&camera));

    display_buffer_addr(0);
    display_buffer_len(BUFFER_LEN); //launch display
    while(!display_is_finished());

    waitms(33);
#endif
  }
  return 0;
}
```

Listing 28: The C code driving the camera in combination with the LCD display.

References

[1] Intel, *NIOS II Processor Reference Guide*, 2020. NII-PRG.

[2] Intel, *Avalon Interface Specifications*, 2020. vers. 20.1.

[3] Intel, *Intel Quartus Prime Standard Edition User Guide: Platform Designer*, 2019. UG-20174.

[4] TerasIC, *Terasic TRDB-D5M Hardware specification*, 2009. Version 0.2.

[5] ILI Technology corp., *a-Si TFT LCD Single Chip Driver 240RGBx320 Resolution and 262K color*. Version 1.11.