Embedded Systems: Lab 4 Report

Aymen Bahroun, Thales Mendes Sampaio, Abhi Kamboj & Aurélien Morel

October 13, 2020

1 Introduction

FPGA custom components are particularly suitable for the design of interfaces with specific hardware components. In this project, our ultimate goal is to design such an interface between a TRDB-D5M camera and a LT24 ILI9341 LCD displayer through a two parts interface with our FPGA board. This report's goal is to present the results of our implementation.

As a reminder on the previous report about the design of the interface, we agreed to use the following memory organisation presented in figure 1. The camera interface acquires the pixels from the camera sensor on a progressive scan, and puts 4 consecutive pixels in each memory address. From the display point of view, the first 320 buffer values will form the first line of the frame.

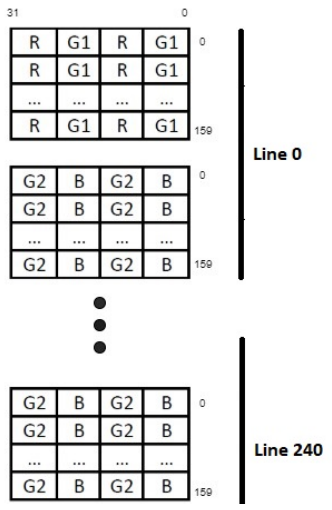

Figure 1: Memory organisation for one frame

2 Register maps

2.1 Camera

Offset	Name	Access	Description
000	CamAddress	R/W	32 bits width, Memory address where to store image data
001	CamLength	R/W	32 bits width, Buffer size
010	CamComm	R/W	8 bits width, Command to the camera interface
011	CamStatus	R/W	8 bit width, Status of the interface
100	CamStart	R/W	8 bits width, Start acquisition
101	CamStop	R/W	8 bits width, Stop acquisition
110	CamSnap	R/W	8 bits width, Snapshot mode

2.2 LCD

Offset	Name	Access	Description
00000	BASE_ADDRESS	R/W	32 bits width, base address in memory of current frame
00001	COMMAND	R/W	8 bits width, command to send
00010	NBR_DATA	R/W	4 bits width, number of data signals to send after the command
00011	CTRL	R/W	1 bit width, 1 to send the command
00100	DATA_1	R/W	8 bits width, data 1
	...	...	...
10010	DATA_15	R/W	8 bits width, data 15
10011	FRAME_OVER	R	1 bit width, new base address is required

3 Simulation using ModelSim

In order to implement, debug and test our code with ease, we use the ModelSim software. With a simple **.do** file, we force our port signals to typical values that would be provided either by the slave interface, the master interface, the FIFO component or the hardware.

3.1 Camera Interface

By simulating the Camera Interface in isolation, we can assure the image data is being stored correctly on the memory buffer. We set the clock of the camera (**PixClk**) to be about 7 times slower than the system clock. The overall camera interface simulation can be seen on Figure 2.

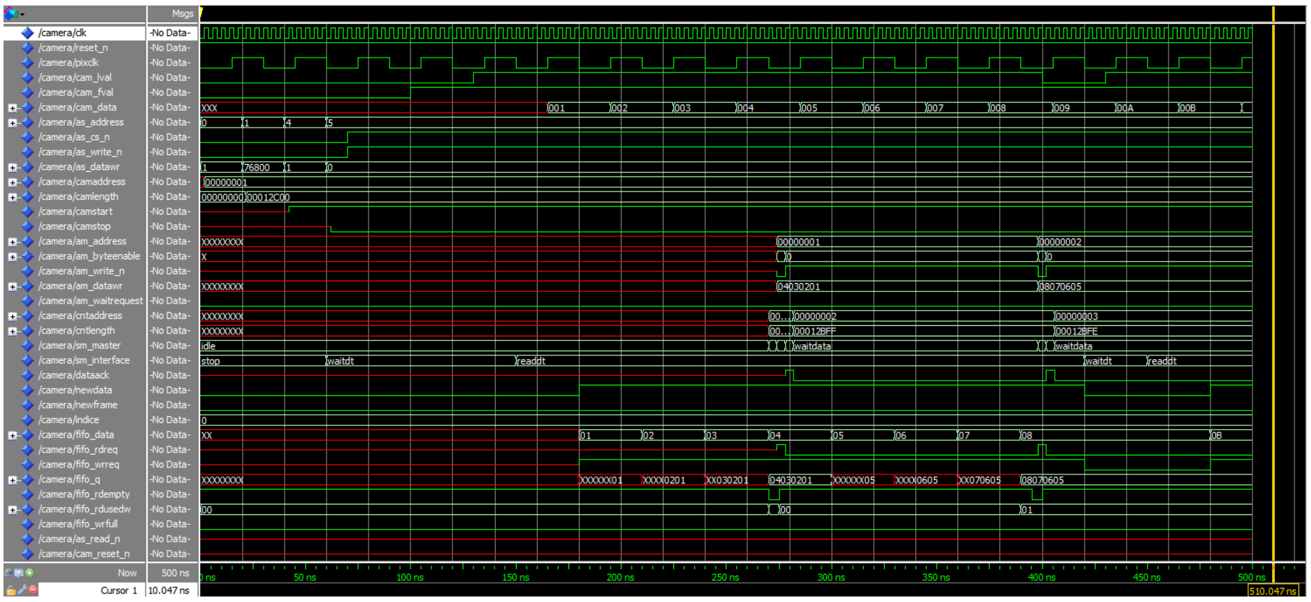

Figure 2: ModelSim results for the camera interface

3.1.1 Avalon Slave

After initialization, we simulate the functionality of the Avalon Slave to set the internal registers such as base memory address and length, and start of acquisition. Figure 3 shows this step of the simulation.

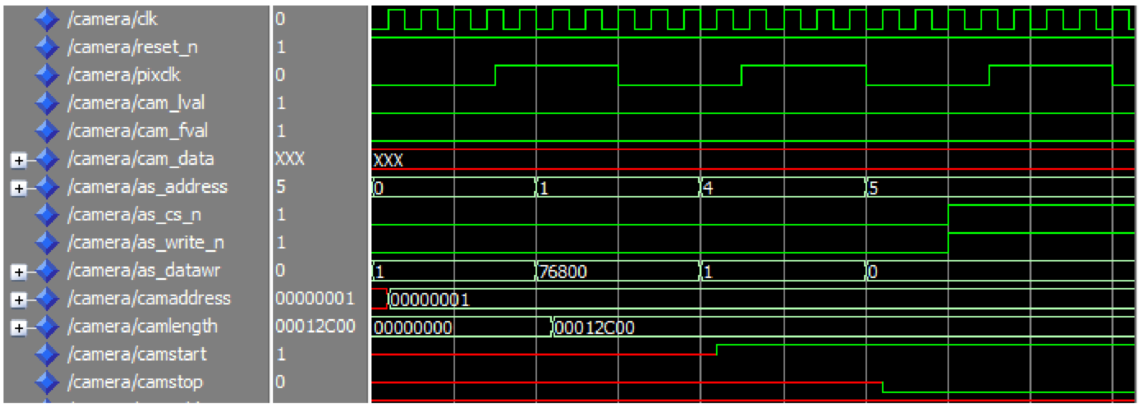

Figure 3: Avalon Slave setting the internal registers of the camera interface

3.1.2 Camera Controller

In this sub-module the data from the camera is acquired and stored on a FIFO for synchronization purpose. The logic runs based on a FSM composed of the states **stop**, **waitdt**, and **readdt**. The Avalon Master will use the FIFO's registers **rdempty** and **rdusedw**, and **NewData** value to read the data from the FIFO's output. The camera data is stored into the FIFO only when the image is valid, being when **Cam_Fval** and **Cam_Lval** are set. Figure 4 shows an example of image data being stored into the FIFO and the Master collecting that data when the output of the FIFO is ready. The data from the camera is checked on the falling edge of **PixClk**.

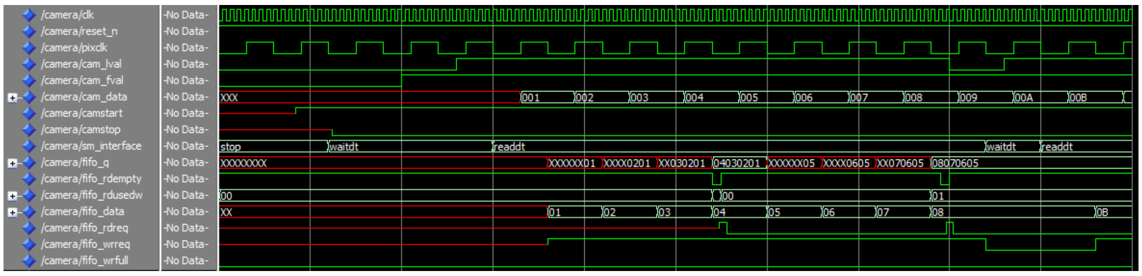

Figure 4: ModelSim results focusing on the interface with camera and FIFO functionality

3.1.3 Avalon Master

The Avalon Master Interface is responsible for acquiring the values from the FIFO and store them on the memory buffer via Direct Memory Access. The Master operates based on a FSM composed of 4 states: **idle**, **waitdata**, **writedata**, and **acqdata**.

The Master stays on **idle** until the FIFO builds its fist output value (32-bit data composed of 4 pixels), then it reads from the internal registers the address and length of the memory space to be used and store these values into its registers **cntaddress** and **cntlength**. During the state **writedata**, the Master requests the FIFO its output value, and writes it into memory. Next state is **acqdata**, where an acknowledgement signal is set and the memory address value is incremented. The following state is **waitdata**, and the process is repeated. These steps are shown on Figure 5.

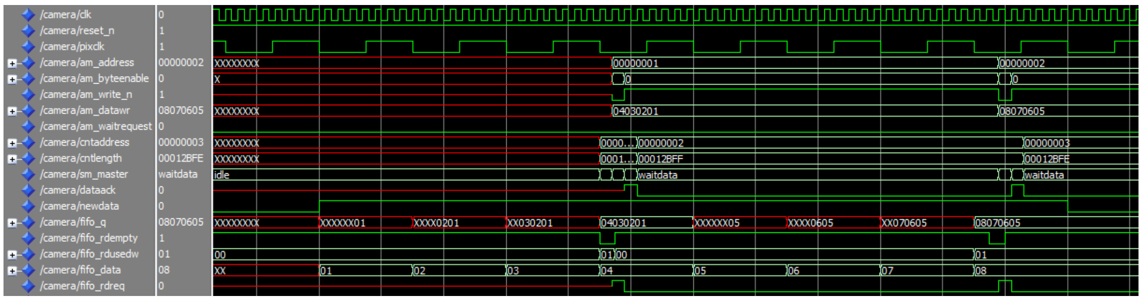

Figure 5: ModelSim results of the Avalon Master Interface during data acquisition

3.2 LCD Interface

To begin with the simulation, we create a simple oscillating clock for the *clk* signal. The operation of the Avalon Slave interface is similar to the camera's slave interface.

3.2.1 Master controller

In order to simulate the reading process from the memory, we set the *readdataM* signal to a constant value, simulating constant memory words. Also, we force the *wait_request* to 0, making the hypothesis that the Avalon Master is always free.

Figure 6 shows the result of the reading simulation. First of all, we see that the state machine cycles through the states **mt1**, **mt2** and **send** (a cycle starts at the yellow mark). As explained in the report for lab 3, one reading cycle reads two pixel values and sends them to the FIFO. In

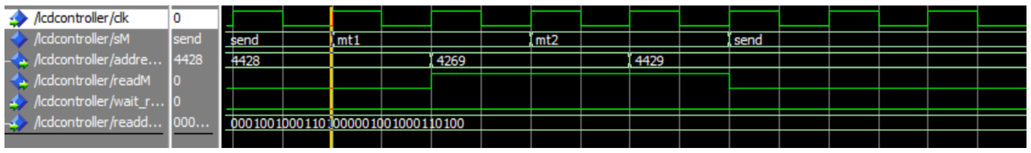

Figure 6: Master controller reading from memory

that sense, two memory words are read. We see that the address is increased by 160 after reading the first word: this is to get the complementary values for the two pixels.

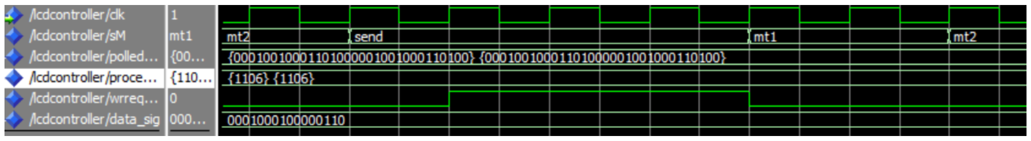

Figure 7: Master controller sending to FIFO

Figure 7 details what happens during the **send** state: the two polled memory words are processed in two pixel values and sent to the FIFO. Polled and processed data do not change here as we simulate a constant memory.

In Figure 8, the controller has read all the values from the frame, and therefore stays in the send state until a new base address is provided.

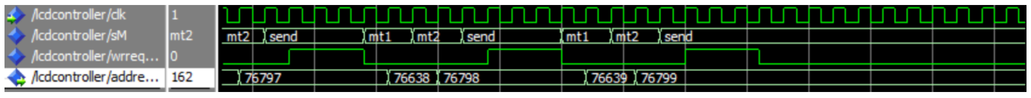

Figure 8: End of the frame

3.2.2 LCD Controller

The first process we simulate is the configuration of the LCD displayer. The first step in this process is to use the Avalon Slave to describe the command and write its parameters in the required registers. Then, figure 9 depicts the process that transfers the command to the LCD displayer. The important detail to notice is that the *dcx* signal moves between command and data: indeed, in that case, we simulate an hypothetical command numbered 1E that requires 3 data words after the command. In that case, *dcx* signal is low when the command is sent, and then goes back to high when the 3 data words are sent.

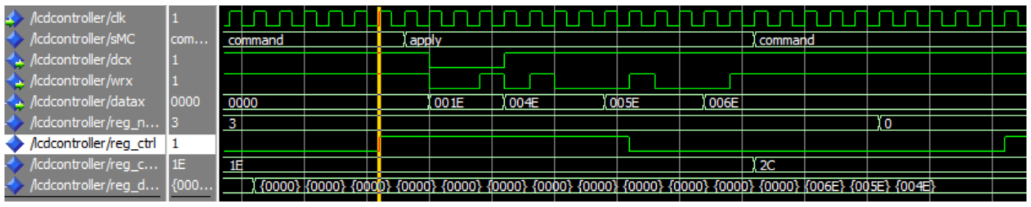

Figure 9: Configuration commands

After the series of configuration commands, the LCD is shifted in the memory write mode, as shown in figure 10. When the command **2C** is applied, the controller switches to data mode.

The next step in our simulation is to read data from the FIFO and transfer to the displayer. Again, for simplicity, we simulate a constant queue in the FIFO. We force it to be not empty at all times. Figure 11 shows the result: we observe a cyclic behavior (reading one value from the FIFO, and then writing the data on the hardware).

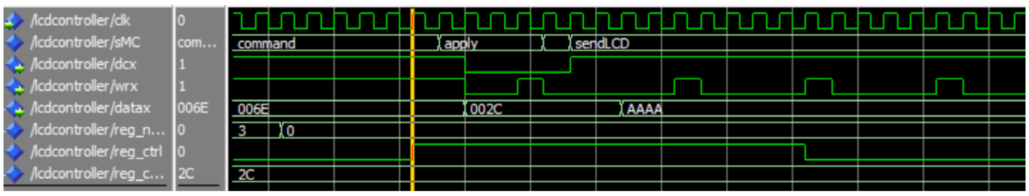

Figure 10: End of configuration and change to display mode

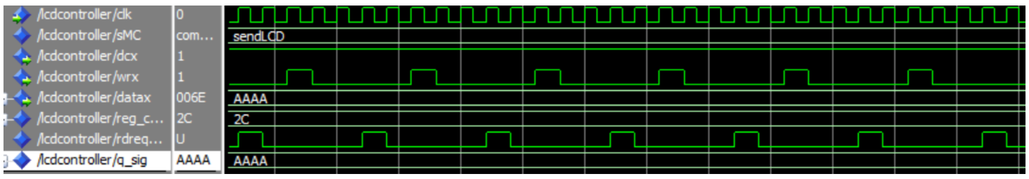

Figure 11: Read from the FIFO and transfer to the displayer

Finally, the end of frame is reached and the controller switches back to command mode (when the number of sent lines equals the number of display lines, see figure 12).

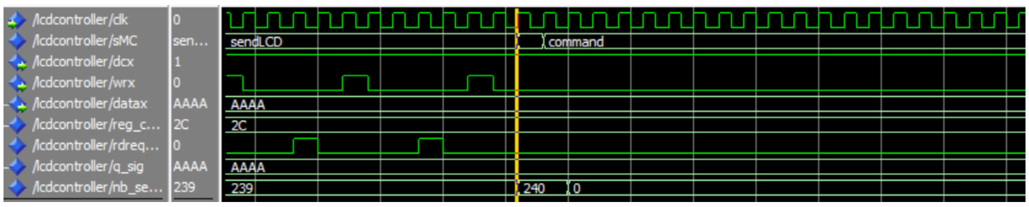

Figure 12: End of frame

4 Implementation

Both Master controllers were combined into one FPGA system and compiled successfully. Then the C code for the software running on the board was created to initialize the hardware (TRDB-D5M and LT24) and the the controllers were configured. Unfortunately, after this point we did not manage to obtain a visualization of the camera image, perhaps due to incomplete initialization/configuration of the device/controllers.

5 Annexes

5.1 Camera hdl code

```
    -- ##############################################################################
-- camera.vhd
--
-- Camera interface for Lab4 - Embedded Systems
--
-- BOARD          : DE0-Nano-SoC from Terasic
-- Author         : Thales Mendes Sampaio / Abhi Kamboj
-- Revision       : 1.1
-- Creation date : 14/12/2019
--
-- ##############################################################################

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

entity camera is

        Port(

                Clk             : IN STD_LOGIC;
                Reset_n         : IN STD_LOGIC;

                -- Camera Interface
                PixClk          : IN STD_LOGIC;
                Cam_Lval                : IN STD_LOGIC;
                Cam_Fval                : IN STD_LOGIc;
                Cam_data                : IN STD_LOGIC_VECTOR(11 downto 0);
                Cam_Reset_n : OUT STD_LOGIC;
                Strobe          : IN STD_LOGIC;
                Trigger         : OUT STD_LOGIC;
                XCLKIN          : OUT STD_LOGIC;

                -- Avalon Slave :
                AS_Address      : IN STD_LOGIC_VECTOR(2 downto 0);
                AS_CS_n                 : IN STD_LOGIC;
                AS_Write_n      : IN STD_LOGIC;
                AS_Read_n       : IN STD_LOGIC;
                AS_DataWr       : IN STD_LOGIC_VECTOR(31 downto 0);
                AS_DataRd       : OUT STD_LOGIC_VECTOR(31 downto 0);
                AS_IRQ_n        : OUT STD_LOGIC;

                -- Avalon Master :
                AM_Address              : OUT STD_LOGIC_VECTOR(31 downto 0);
                AM_ByteEnable   : OUT STD_LOGIC_VECTOR(3 downto 0);
                AM_Write_n              : OUT STD_LOGIC;
                AM_DataWr               : OUT STD_LOGIC_VECTOR(31 downto 0);
                AM_WaitRequest : IN STD_LOGIC

        );

end entity camera;


Architecture Comp of camera is

component fifo
        PORT
        (
                data                    : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                rdclk                   : IN STD_LOGIC ;
                rdreq                   : IN STD_LOGIC ;
                wrclk                   : IN STD_LOGIC ;
                wrreq                   : IN STD_LOGIC ;
                q                               : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
                rdempty         : OUT STD_LOGIC ;
                rdusedw         : OUT STD_LOGIC_VECTOR (5 DOWNTO 0);
                wrfull          : OUT STD_LOGIC
        );
end component;

TYPE AM_State is (Idle, WaitData, WriteData, AcqData);
TYPE CI_State is (Stop, WaitDt, ReadDt);

Signal CamAddress               : STD_LOGIC_VECTOR(31 downto 0);
Signal CamLength                : STD_LOGIC_VECTOR(31 downto 0) := X"0000_0000";
Signal CamComm                  : STD_LOGIC_VECTOR(7 downto 0);
Signal CamStatus                : STD_LOGIC_VECTOR(7 downto 0);
Signal camStart         : STD_LOGIC;
Signal CamStop                  : STD_LOGIC;
Signal CamSnap                  : STD_LOGIC;

Signal CntAddress               : STD_LOGIC_VECTOR(31 downto 0);
Signal CntLength                : STD_LOGIC_VECTOR(31 downto 0);
Signal SM_Master                : AM_State;
Signal SM_Interface  : CI_State;

Signal DataAck                  : STD_LOGIC;
Signal NewData                  : STD_LOGIC := '0';
Signal NewFrame         : STD_LOGIC := '0';

Signal Indice                   : Integer Range 0 to 3;

Signal FIFO_data                : STD_LOGIC_VECTOR (7 DOWNTO 0);
Signal FIFO_rdreq               : STD_LOGIC ;
Signal FIFO_wrreq               : STD_LOGIC ;
Signal FIFO_q                   : STD_LOGIC_VECTOR (31 DOWNTO 0);
Signal FIFO_rdempty     : STD_LOGIC ;
Signal FIFO_rdusedw     : STD_LOGIC_VECTOR (5 DOWNTO 0);
Signal FIFO_wrfull      : STD_LOGIC;


Begin

fifo_inst : fifo PORT MAP (
                data            => FIFO_data,
        rdclk           => PixClk,
        rdreq           => FIFO_rdreq,
        wrclk           => Clk,
        wrreq           => FIFO_wrreq,
        q                       => FIFO_q,
        rdempty => FIFO_rdempty,
        rdusedw => FIFO_rdusedw,
        wrfull  => FIFO_wrfull
    );

-- Camera Interface
pCamera_Interface:
Process(PiXClk, Reset_n)
Begin
    if Reset_n = '0' then
        cam_Reset_n <= '0';
        SM_Interface <= Stop;
        FIFO_wrreq <= '0';
    elsif falling_edge(PixClk) then         -- Lval, Fval, and data should be captured on the falling edge of PixClk
        case SM_Interface is
            when Stop =>
                if CamStart = '1' then
                    SM_Interface <= WaitDt;
                end if;
            when WaitDt =>
                if (Cam_Lval = '1') and (Cam_Fval = '1') then
                    SM_Interface <= ReadDt;
                end if;
            when ReadDt =>
                if CamStop = '0' then
                    if (Cam_Lval = '1') and (Cam_Fval = '1') then
                            FIFO_wrreq <= '1';
                            FIFO_data <= std_logic_vector(resize(unsigned(Cam_data), FIFO_data'len
--scale 12 bits down to 8
                            NewData <= '1';
                    else
                        SM_Interface <= WaitDt;
                        FIFO_wrreq <= '0';
                        NewData <= '0';
                    end if;
                elsif CamStop = '1' then
                    SM_Interface <= Stop;
                    FIFO_wrreq <= '0';
                    NewData <= '0';
                end if;
        end case;
    end if;

End Process pCamera_Interface;

-- Avalon Slave
pAvalon_Slave:
Process(Clk, Reset_n)
Begin
    if Reset_n = '0' then
        CamAddress <= (others => '0');
        CamLength <= (others => '0');
    elsif rising_edge(Clk) then
        if AS_CS_n = '0' then
            if AS_Write_n = '0' then
                case AS_Address is
                    when "000" => CamAddress <= AS_DataWr;
                    when "001" => CamLength <= AS_DataWr;
                    when "010" => CamComm <= AS_DataWr(7 downto 0);
                    when "011" => CamStatus <= AS_DataWr(7 downto 0);
                    when "100" => CamStart <= AS_DataWr(0);
                    when "101" => CamStop <= AS_DataWr(0);
                    when "110" => CamSnap <= AS_DataWr(0);
                    when others => null;
                end case;
            elsif AS_Read_n = '0' then
                case AS_Address is
                    when "000" => AS_DataRd <= CamAddress;
                    when "001" => AS_DataRd <= CamLength;
                    when "010" => AS_DataRd(7 downto 0) <= CamComm;
                    when "011" => AS_DataRd <= CamAddress;
                    when "100" => AS_DataRd(0) <= CamStart;
                    when "101" => AS_DataRd(0) <= CamStop;
                    when "110" => AS_DataRd(0) <= Camsnap;
                    when others => null;
                end case;
            end if;
        end if;
    end if;
End Process pAvalon_Slave;


-- Avalon Master
pAvalon_Master:
Process(Clk, Reset_n)
Begin
    if Reset_n = '0' then
        DataAck <= '0';
        SM_Master <= Idle;
        AM_Write_n <= '1';
        AM_ByteEnable <= "0000";
        CntAddress <= (others => '0');
        CntLength <= (others => '0');
        FIFO_rdreq <= '0';
    elsif rising_edge(Clk) then
        case SM_Master is
            when Idle =>
                if FIFO_rdusedw >= "000001" then                -- This checks if the first 4 pixels were sto
                    SM_Master <= WaitData;
                    CntAddress <= CamAddress;
                    CntLength <= CamLength;
                end if;
            when WaitData =>
                if NewData = '1' and FIFO_rdempty /= '1' then
                    SM_Master <= WriteData;
                    AM_Address <= CntAddress;
                    AM_Write_n <= '0';
                    FIFO_rdreq <= '1';
                    AM_DataWr <= FIFO_q;
                    AM_ByteEnable <= "1111";
                end if;
```

```
                        when WriteData =>
                                if AM_WaitRequest = '0' then
                                        SM_Master <= AcqData;
                                        AM_Write_n <= '1';
                                        FIFO_rdreq <= '0';
                                        AM_ByteEnable <= "0000";
                                        DataAck <= '1';
                                end if;
                        when AcqData =>
                                SM_Master <= WaitData;
                                DataAck <= '0';
                                if CntLength /= std_logic_vector(to_unsigned(1,32)) then
                                        CntAddress <= std_logic_vector(unsigned(CntAddress) + 1);
                                        CntLength <= std_logic_vector(unsigned(CntLength) - 1);
                                else
                                        CntAddress <= CamAddress;
                                        CntLength <= CamLength;
                                end if;
                end case;
        end if;
End Process pAvalon_Master;
End Comp;
```

5.2 LCD controller hdl code

```
    -- #############################################################################
-- lcdController.vhd
--
-- LCD interface for Lab4 - Embedded Systems
--
-- BOARD         : DE0-Nano-SoC from Terasic
-- Author        : Aymen Bahroun / Aur lien Morel
-- Revision      : 1.2
-- Creation date : 14/12/2019
--
-- #############################################################################


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity lcdController is
  port(
    -- Avalon Clock Interface
    clk : in std_logic;

    -- Avalon Reset Interface
    reset : in std_logic;

    -- Avalon Conduit Interface
    csx : out std_logic;
    dcx : out std_logic;
    wrx : out std_logic;
    rdx : out std_logic;
    datax : out std_logic_vector(15 downto 0);




    -- Avalon Master Interface
    addressM : out std_logic_vector(31 downto 0);
--    burst_count : out std_logic_vector(8 downto 0);
    readdataM : in std_logic_vector(31 downto 0);
--    readdatavalid : in std_logic;
    readM : out std_logic;
    wait_request : in std_logic;

    -- Avalon Slave Interface
    address : in std_logic_vector(4 downto 0);
    read    : in std_logic;
    write   : in std_logic;
    readdata: out std_logic_vector(31 downto 0);
    writedata: in std_logic_vector(31 downto 0)


  );
end lcdController;

architecture logic of lcdController is
  component fifo
        PORT
        (
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
                rdreq           : IN STD_LOGIC ;
                wrreq           : IN STD_LOGIC ;
                almost_empty            : OUT STD_LOGIC ;
                full  : OUT STD_LOGIC;
                q               : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
        );
  end component;


  type State is (wt, mt1, mt2, send);
  type StateC is (apply, command, data, sendLCD);
  type Polled is array (1 downto 0) of std_logic_vector(31 downto 0);
  type Processed is array (1 downto 0) of std_logic_vector(15 downto 0);
  type CommandData is array (14 downto 0) of std_logic_vector(15 downto 0);

  signal reg_base_address : std_logic_vector(31 downto 0);
  signal reg_command : std_logic_vector(7 downto 0);
  signal reg_nbr_data : std_logic_vector(3 downto 0);
  signal reg_ctrl : std_logic;
  signal reg_data : CommandData;
  signal reg_frame_over : std_logic;
  signal sM : State;
  --signal newitem : std_logic;
  signal nb_received_bursts : integer range 0 to 511;
  signal nb_command_data_sent : integer range 0 to 14;
  signal nb_sent_data : integer range 0 to 511;
  signal nb_sent_data_lcd : integer range 0 to 511;
  signal nb_sent_lines : integer range 0 to 511;
  signal polledData : Polled;
  signal processedData : Processed;
  signal sMC : StateC;
  signal reg_ctrl_bis : std_logic;
  signal        data_sig                : STD_LOGIC_VECTOR(15 DOWNTO 0);
  signal        rdreq_sig               : STD_LOGIC;
  signal        wrreq_sig               : STD_LOGIC;
  signal        almost_empty_sig                : STD_LOGIC;
  signal        q_sig           : STD_LOGIC_VECTOR(15 DOWNTO 0);
  signal full_sig : std_logic;
  signal addressIncrement : natural range 0 to 200000;

  constant REG_BASE_ADDRESS_OFST : std_logic_vector(4 downto 0) := "00000";
  constant REG_COMMAND_OFST   : std_logic_vector(4 downto 0) := "00001";
  constant REG_NBR_DATA_OFST    : std_logic_vector(4 downto 0) := "00010";
  constant REG_CTRL_OFST   : std_logic_vector(4 downto 0) := "00011";
  constant REG_DATA_1_OFST   : std_logic_vector(4 downto 0) := "00100";
  constant REG_DATA_2_OFST   : std_logic_vector(4 downto 0) := "00101";
  constant REG_DATA_3_OFST   : std_logic_vector(4 downto 0) := "00110";
  constant REG_DATA_4_OFST   : std_logic_vector(4 downto 0) := "00111";
  constant REG_DATA_5_OFST   : std_logic_vector(4 downto 0) := "01000";
  constant REG_DATA_6_OFST   : std_logic_vector(4 downto 0) := "01001";
  constant REG_DATA_7_OFST   : std_logic_vector(4 downto 0) := "01010";
  constant REG_DATA_8_OFST   : std_logic_vector(4 downto 0) := "01011";
  constant REG_DATA_9_OFST    : std_logic_vector(4 downto 0) := "01100";
  constant REG_DATA_10_OFST    : std_logic_vector(4 downto 0) := "01101";
  constant REG_DATA_11_OFST    : std_logic_vector(4 downto 0) := "01110";
  constant REG_DATA_12_OFST    : std_logic_vector(4 downto 0) := "01111";
  constant REG_DATA_13_OFST    : std_logic_vector(4 downto 0) := "10000";
  constant REG_DATA_14_OFST    : std_logic_vector(4 downto 0) := "10001";
  constant REG_DATA_15_OFST    : std_logic_vector(4 downto 0) := "10010";
  constant REG_FRAME_OVER_OFST    : std_logic_vector(4 downto 0) := "10011";
  constant BUFFER_LENGTH : std_logic_vector(8 downto 0) := "101000000";
  constant MEMORY_WRITE : std_logic_vector(7 downto 0) := "00101100";
  constant NB_LINES : std_logic_vector(7 downto 0) := "11110000";
  constant NB_MEMORY_WORDS : std_logic_vector(16 downto 0) := "10010110000000000";



begin
  fifo_inst : fifo PORT MAP (
                clock     => clk,
                data      => data_sig,
                rdreq     => rdreq_sig,
                wrreq     => wrreq_sig,
                almost_empty      => almost_empty_sig,
                full => full_sig,
                q         => q_sig
        );

  -- Avalon-MM slave write
  process(clk, reset)
  begin
    if reset = '1' then
                reg_base_address <= (others => '0');
                reg_command <= (others => '0');
                reg_nbr_data <= (others => '0');
                reg_ctrl <= '0';
                for i in reg_data' range loop
                        reg_data(i) <= (others => '0');
                end loop;




    elsif rising_edge(clk) then
                if write = '1' then
                        case address is
                                when REG_BASE_ADDRESS_OFST =>
                                        reg_base_address <= writedata;
                                when REG_COMMAND_OFST =>
                                        reg_command <= writedata(7 downto 0);
                                when REG_NBR_DATA_OFST =>
                                        reg_nbr_data <= writedata(3 downto 0);
                                when REG_CTRL_OFST =>
                                        reg_ctrl <= writedata(0);
                                when REG_DATA_1_OFST =>
                                        reg_data(0) <= writedata(15 downto 0);
                                when REG_DATA_2_OFST =>
                                        reg_data(1) <= writedata(15 downto 0);
                                when REG_DATA_3_OFST =>
                                        reg_data(2) <= writedata(15 downto 0);
                                when REG_DATA_4_OFST =>
                                        reg_data(3) <= writedata(15 downto 0);
                                when REG_DATA_5_OFST =>
                                        reg_data(4) <= writedata(15 downto 0);
                                when REG_DATA_6_OFST =>
                                        reg_data(5) <= writedata(15 downto 0);
                                when REG_DATA_7_OFST =>
                                        reg_data(6) <= writedata(15 downto 0);
                                when REG_DATA_8_OFST =>
                                        reg_data(7) <= writedata(15 downto 0);
                                when REG_DATA_9_OFST =>
                                        reg_data(8) <= writedata(15 downto 0);
                                when REG_DATA_10_OFST =>
                                        reg_data(9) <= writedata(15 downto 0);
                                when REG_DATA_11_OFST =>
                                        reg_data(10) <= writedata(15 downto 0);
                                when REG_DATA_12_OFST =>
                                        reg_data(11) <= writedata(15 downto 0);
                                when REG_DATA_13_OFST =>
                                        reg_data(12) <= writedata(15 downto 0);
                                when REG_DATA_14_OFST =>
                                        reg_data(13) <= writedata(15 downto 0);
                                when REG_DATA_15_OFST =>
                                        reg_data(14) <= writedata(15 downto 0);



                                -- Remaining addresses in register map are unused.
                                when others => null;
                        end case;
                else
                        if reg_ctrl_bis = '0' then
                                reg_ctrl <= '0';
                        end if;
                end if;

    end if;
  end process;


  -- Avalon-MM slave read
        process(clk, reset)
                begin
                if rising_edge(clk) then
                        readdata <= (others => '0');
                        if read = '1' then

                                case address is

                                        when REG_BASE_ADDRESS_OFST =>
                                                readdata <= reg_base_address;
                                        when REG_COMMAND_OFST =>
                                                readdata(7 downto 0) <= reg_command;
                                        when REG_NBR_DATA_OFST =>
```

```
                                            readdata(3 downto 0) <= reg_nbr_data;
                                    when REG_CTRL_OFST =>
                                            readdata(0) <= reg_ctrl;
                                    when REG_FRAME_OVER_OFST =>
                                            readdata(0) <= reg_frame_over;
                                    -- Remaining addresses in register map are unmapped => return 0.
                                    when others =>
                                    readdata <= (others => '0');
                            end case;
                    end if;
            end if;
    end process;
-- Avalon Master State Machine --
    process(clk, reset)
            variable G1 : natural range 0 to 255;
            variable R : natural range 0 to 255;
            variable B : natural range 0 to 255;
            variable G2 : natural range 0 to 255;
            variable Breal : natural range 0 to 31;
            variable Greal : natural range 0 to 63;
            variable Rreal : natural range 0 to 31;

            variable readTimer : natural range 0 to 10;
            variable currentBaseAddress : std_logic_vector(31 downto 0);


            begin
            if reset = '1' then
                    sM <= wt;
            elsif rising_edge(clk) then
                    case sM is
                            when wt =>
                                    readTimer := 0;
                                    addressIncrement <= 0;
                                    currentBaseAddress := reg_base_address;
                                    if full_sig /= '1' then
                                            sM <= mt1;
                                    end if;
                                    if reg_frame_over = '1' then
                                            sM <= wt;
                                    end if;
                            when mt1 =>
                                    if readTimer = 0 then
                                            readM <= '1';
                                            nb_sent_data <= 0;
                                            addressM <= std_logic_vector(to_unsigned(to_integer(unsigned(reg_base_address)
                                            if wait_request = '0' then
                                                    readTimer := readTimer + 1;
                                            end if;
                                    else
                                            polledData(0) <= readDataM;
                                            readTimer := 0;
                                            sM <= mt2;
                                    end if;

                            when mt2 =>
                                    if readTimer = 0 then
                                            readM <= '1';
                                            addressM <= std_logic_vector(to_unsigned(to_integer(unsigned(reg_base_address)
                                            if wait_request = '0' then
                                                    readTimer := readTimer + 1;
                                            end if;
                                    else
                                            polledData(1) <= readDataM;
                                            readTimer := 0;
                                            readM <= '0';
                                            addressIncrement <= addressIncrement + 1;
                                            if addressIncrement mod 160 = 159 then
                                                  addressIncrement <= addressIncrement + 161;
                                            end if;
                                            sM <= send;
                                    end if;

                            when send =>
                                    if nb_sent_data = 0 then
                                            for i in processedData' range loop

                                                    if i = 0 then
                                                            G1 := to_integer(unsigned(polledData(0)(7 downto 0)));
                                                            R := to_integer(unsigned(polledData(0)(15 downto 8)));
                                                            B := to_integer(unsigned(polledData(1)(7 downto 0)));
                                                            G2 := to_integer(unsigned(polledData(1)(15 downto 8)));
                                                            Breal := integer(B * 31 / 255);
                                                            Greal := integer((G1 + G2) * 63 / (2 * 255));
                                                            Rreal := integer(R * 31 / 255);
                                                            processedData(0)(4 downto 0) <= std_logic_vector(to_unsigned(
                                                            processedData(0)(10 downto 5) <= std_logic_vector(to_unsigned(
                                                            processedData(0)(15 downto 11) <= std_logic_vector(to_unsigned

                                                    else
                                                            G1 := to_integer(unsigned(polledData(0)(23 downto 16)));
                                                            R := to_integer(unsigned(polledData(0)(31 downto 24)));
                                                            B := to_integer(unsigned(polledData(1)(23 downto 16)));
                                                            G2 := to_integer(unsigned(polledData(1)(31 downto 24)));
                                                            Breal := integer(B * 31 / 255);
                                                            Greal := integer((G1 + G2) * 63 / (2 * 255));
                                                            Rreal := integer(R * 31 / 255);
                                                            processedData(1)(4 downto 0) <= std_logic_vector(to_unsigned(
                                                            processedData(1)(10 downto 5) <= std_logic_vector(to_unsigned(
                                                            processedData(1)(15 downto 11) <= std_logic_vector(to_unsigned



                                                    end if;

                                            end loop;
                                            nb_sent_data <= nb_sent_data + 1;
                                    end if;

                                    wrreq_sig <= '1';
                                    if nb_sent_data > 0 then
                                            if nb_sent_data < 3 then
                                                    data_sig <= processedData(nb_sent_data - 1);
                                                    nb_sent_data <= nb_sent_data + 1;
                                            end if;
                                            if nb_sent_data = 3  then
                            if addressIncrement >= unsigned(nb_memory_words) then
                                if reg_base_address /= currentBaseAddress then
                                    sM <= wt;
                                end if;
                            elsif full_sig /= '1'then
                                sM <= mt1;
                            end if;
                            wrreq_sig <= '0';
                        end if;
                    end if;
                when others =>

            end case;
        end if;
    end process;


-- LCD Controller State Machine --
    process(clk, reset)
        variable timerForCommand : natural range 0 to 159;
        variable timerForData : natural range 0 to 159;

        begin
        if reset = '1' then
            sMC <= command;
            csx <= '1';
            dcx <= '1';
            wrx <= '1';
            rdx <= '1';
            datax <= (others => '0');
            reg_ctrl_bis <= '1';
            reg_frame_over <= '0';
        elsif rising_edge(clk) then
            case sMC is
                when command =>
                    nb_sent_lines <= 0;
                    timerForCommand := 0;
                    nb_command_data_sent <= 0;
                    if reg_ctrl = '1' then
                        reg_ctrl_bis <= '1';
                        smC <= apply;
                    end if;
                    if reg_frame_over = '1' then
                        sMC <= command;
                    end if;


                when apply =>
                    csx <= '0';
                    if timerForCommand <= 3 then
                        dcx <= '0';
                        datax(7 downto 0) <= reg_command;
                        wrx <= '0';
                        if timerForCommand = 2 then
                            wrx <= '1';
                            reg_ctrl_bis <= '0';
                        end if;
                        timerForCommand := timerForCommand + 1;
                    end if;
                    if timerForCommand > 3 then
                        if nb_command_data_sent = to_integer(unsigned(reg_nbr_data)) then
                            if reg_command = MEMORY_WRITE then
                                smC <= data;
                            else
                                sMC <= command;
                            end if;
                        else
                            dcx <='1';
                            wrx <= '0';
                            if timerForCommand mod 4 = 0 then
                                datax(15 downto 0) <= reg_data(nb_command_data_sent);
                            end if;
                            if timerForCommand mod 4 = 1 then
                                wrx <= '1';
                                nb_command_data_sent <= nb_command_data_sent + 1;
                            end if;
                            timerForCommand := timerForCommand + 1;
                        end if;
                    end if;
                when data =>
                    dcx <= '1';
                    nb_sent_data_lcd <= 0;
                    timerForData := 0;
                    if reg_command /= MEMORY_WRITE then
                        sMC <= command;
                    end if;
                    if almost_empty_sig = '0' then
                        sMC <= sendLCD;
                    end if;
                    if nb_sent_lines = unsigned(NB_LINES) then
                        sMC <= command;
                        reg_frame_over <= '1';
                    end if;

                when sendLCD =>
                    rdreq_sig <= '0';
                    wrx <= '0';
                    if timerForData = 1 then
                        rdreq_sig <= '1';
                        datax <= q_sig;
                    end if;
                    if timerForData = 3 then

                        wrx <= '1';

                    end if;
                    if timerForData = 4 then

                        nb_sent_data_lcd <= nb_sent_data_lcd + 1;
                    end if;
                    if timerForData = 5 then
                        timerForData := 0;
                    end if;

                    timerForData := timerForData + 1;
                    if nb_sent_data_lcd = unsigned(BUFFER_LENGTH) then
                        sMC <= data;
                        nb_sent_lines <= nb_sent_lines + 1;
                    end if;


                when others =>
            end case;
        end if;

    end process;



end logic;
```

5.3 C Code

```
#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>

#include "system.h"
#include "io.h"
#include "i2c/i2c.h"

#define I2C_FREQ                (50000000) /* Clock frequency driving the i2c core: 50 MHz in this example */
#define TRDB_D5M_I2C_ADDRESS    I2C_0_BASE

// Addresses for the LCD Interface
#define REG_LCD_BASE            LCD_CONTROLLER_0_BASE
#define REG_BASE_ADDRESS_OFST   0 * 5
#define REG_COMMAND_OFST        1 * 5
#define REG_NBR_DATA_OFST       2 * 5
#define REG_CTRL_OFST                   3 * 5

// Addresses for the Camera Interface
#define REG_CAM_BASE            CAMERA_0_BASE
#define REG_CAM_ADDRESS_OFST    0
#define REG_CAM_LENGTH_OFST     1 * 3
#define REG_CAM_COMM_OFST       2 * 3
#define REG_CAM_STATUS_OFST     3 * 3
#define REG_CAM_START_OFST      4 * 3
#define REG_CAM_STOP_OFST       5 * 3
#define REG_CAM_SNAP_OFST       6 * 3

#define LCD_WR_REG(command) ({\
            IOWR_32DIRECT(REG_LCD_BASE, REG_COMMAND_OFST, command);\
                })
#define LCD_WR_DATA(num, data) ({\
                        int REG_DATA_OFST = (num+3) * 5; \
            IOWR_32DIRECT(REG_LCD_BASE, REG_DATA_OFST, data);\
                })
#define LCD_Control() ({\
            IOWR_32DIRECT(REG_LCD_BASE, REG_CTRL_OFST, 1);\
                })
#define LCD_NBR_data(num) ({\
            IOWR_32DIRECT(REG_LCD_BASE, REG_NBR_DATA_OFST, num);\
                })

void lcd_init();
void cam_init();
bool trdb_d5m_write(i2c_dev *i2c, uint8_t register_offset, uint16_t data);
bool trdb_d5m_read(i2c_dev *i2c, uint8_t register_offset, uint16_t *data);

int main()
{
        cam_init();
        lcd_init();

        while(1)
        {

        }

        return 0;
}

void cam_init()
{
        // i2c instantiation
    i2c_dev i2c = i2c_inst((void *) TRDB_D5M_I2C_ADDRESS);
    i2c_init(&i2c, I2C_FREQ);

    // Camera configuration
    trdb_d5m_write(&i2c, 0x0D, 1);      // Reset for configuration
        trdb_d5m_write(&i2c, 4, 639);   // Column Size
        trdb_d5m_write(&i2c, 3, 479);   // Row Size
        trdb_d5m_write(&i2c, 9, 479);   // Shutter Width Lower
        trdb_d5m_write(&i2c, 34, 0);    // Row Bin / Row Skip
        trdb_d5m_write(&i2c, 35, 0);    // Column Bin / Column Skip
        trdb_d5m_write(&i2c, 98, 1);    // No dark rows/columns are read
        trdb_d5m_write(&i2c, 0x0D, 0);  // Normal operation

        // Set Base Memory
        IOWR_32DIRECT(REG_CAM_BASE, REG_CAM_ADDRESS_OFST, 0x00f6);
        // Set Frame Length
        IOWR_32DIRECT(REG_CAM_BASE, REG_CAM_LENGTH_OFST, 76800);
        // Start Acquisition
        IOWR_32DIRECT(REG_CAM_BASE, REG_CAM_START_OFST, 1);
        IOWR_32DIRECT(REG_CAM_BASE, REG_CAM_STOP_OFST, 0);
}

void lcd_init()
{
        //Set_LCD_RST
        //Clr_LCD_RST
        //Delay_Ms(10)


        //exit sleep mode
        LCD_WR_REG(0x0011);
        LCD_Control();

        //Power control B
        LCD_WR_REG(0x00CF);
        LCD_NBR_data(3);
        LCD_WR_DATA(1, 0x0000);
        LCD_WR_DATA(2, 0x0081);
        LCD_WR_DATA(3, 0X00c0);
        LCD_Control();

        //Driver timing control A
        LCD_WR_REG(0x00E8);
        LCD_NBR_data(3);
        LCD_WR_DATA(1, 0x0085);
        LCD_WR_DATA(2, 0x0001);
        LCD_WR_DATA(3, 0X0798);
        LCD_Control();

        //Power control A
        LCD_WR_REG(0x00CB);
        LCD_NBR_data(5);
        LCD_WR_DATA(1, 0x0039);
        LCD_WR_DATA(2, 0x002C);
        LCD_WR_DATA(3, 0X0000);
        LCD_WR_DATA(4, 0X0034);
        LCD_WR_DATA(5, 0X0002);
        LCD_Control();

        //Pump ratio control
        LCD_WR_REG(0x00F7);
        LCD_NBR_data(1);
        LCD_WR_DATA(1, 0x0020);
        LCD_Control();

        //Driver timing control B
        LCD_WR_REG(0x00EA);
        LCD_NBR_data(2);
        LCD_WR_DATA(1, 0x0000);
        LCD_WR_DATA(2, 0x0000);
        LCD_Control();

        //Frame control : normal mode
        LCD_WR_REG(0x00B1);
        LCD_NBR_data(2);
        LCD_WR_DATA(1, 0x0000);
        LCD_WR_DATA(2, 0x001b);
        LCD_Control();

        //Display function control
        LCD_WR_REG(0x00B6);
        LCD_NBR_data(2);
        LCD_WR_DATA(1, 0x000A);
        LCD_WR_DATA(2, 0x00A2);
        LCD_Control();

        //Power control 1
        LCD_WR_REG(0x00C0);
        LCD_NBR_data(1);
        LCD_WR_DATA(1, 0x0005);
        LCD_Control();

        //Power control 2
        LCD_WR_REG(0x00C1);
        LCD_NBR_data(1);
        LCD_WR_DATA(1, 0x0011);
        LCD_Control();

        //VCM control 1
        LCD_WR_REG(0x00C5);
        LCD_NBR_data(2);
        LCD_WR_DATA(1, 0x0045);
        LCD_WR_DATA(2, 0x0045);
        LCD_Control();

        //VCM control 2
        LCD_WR_REG(0x00C7);
        LCD_NBR_data(1);
        LCD_WR_DATA(1, 0x00a2);
        LCD_Control();


        //MADCTR : 100 to MV MX MY
        //Memory access control
        LCD_WR_REG(0x0036);
        LCD_NBR_data(1);
        //LCD_WR_DATA(1, 0x0008); //BGR order
        LCD_WR_DATA(1, 0x0028); //BGR order display data direction X-Y exchange
        LCD_Control();

        //Disable 3Gamma function
        LCD_WR_REG(0x00F2);
        LCD_NBR_data(1);
        LCD_WR_DATA(1, 0x0000);
        LCD_Control();

        //Gamma set
        LCD_WR_REG(0x0026);
        LCD_NBR_data(1);
        LCD_WR_DATA(1, 0x0001);  //select gamma curve
        LCD_Control();

        //Positive gamma correction set gamma
        LCD_WR_REG(0x00E0);
        LCD_NBR_data(15);
        LCD_WR_DATA(1, 0x000F);
        LCD_WR_DATA(2, 0x0026);
        LCD_WR_DATA(3, 0X0024);
        LCD_WR_DATA(4, 0X000b);
        LCD_WR_DATA(5, 0X000E);
        LCD_WR_DATA(6, 0x0008);
        LCD_WR_DATA(7, 0x004b);
        LCD_WR_DATA(8, 0X00a8);
        LCD_WR_DATA(9, 0X003b);
        LCD_WR_DATA(10, 0X000a);
        LCD_WR_DATA(11, 0x0014);
        LCD_WR_DATA(12, 0x0006);
        LCD_WR_DATA(13, 0X0010);
        LCD_WR_DATA(14, 0X0009);
        LCD_WR_DATA(15, 0X0000);
        LCD_Control();

        //Negative gamma correction set gamma
        LCD_WR_REG(0x00E1);
        LCD_NBR_data(15);
        LCD_WR_DATA(1, 0x0000);
        LCD_WR_DATA(2, 0x001c);
        LCD_WR_DATA(3, 0X0020);
        LCD_WR_DATA(4, 0X0004);
        LCD_WR_DATA(5, 0X0010);
        LCD_WR_DATA(6, 0x0008);
        LCD_WR_DATA(7, 0x0034);
```

```
        LCD_WR_DATA(8, 0X0047);
        LCD_WR_DATA(9, 0X0044);
        LCD_WR_DATA(10, 0X0005);
        LCD_WR_DATA(11, 0x000b);
        LCD_WR_DATA(12, 0x0009);
        LCD_WR_DATA(13, 0X002f);
        LCD_WR_DATA(14, 0X0036);
        LCD_WR_DATA(15, 0X000f);
        LCD_Control();

        //Column address set
        LCD_WR_REG(0x002A);
        LCD_NBR_data(4);
        LCD_WR_DATA(1, 0x0000);
        LCD_WR_DATA(2, 0x0000);
        LCD_WR_DATA(3, 0x0000);
        LCD_WR_DATA(4, 0x00ef);
        LCD_Control();

        //Page address set
        LCD_WR_REG(0x002A);
        LCD_NBR_data(4);
        LCD_WR_DATA(1, 0x0000);
        LCD_WR_DATA(2, 0x0000);
        LCD_WR_DATA(3, 0x0000);
        LCD_WR_DATA(4, 0x00ef);
        LCD_Control();

        //COLMOD : pixel format set
        LCD_WR_REG(0x003A);
        LCD_NBR_data(1);
        LCD_WR_DATA(1, 0x0055);
        LCD_Control();

        //Interface control
        LCD_WR_REG(0x00f6);
        LCD_NBR_data(3);
        LCD_WR_DATA(1, 0x0001);
        LCD_WR_DATA(2, 0x0030);
        LCD_WR_DATA(3, 0x0000);
        LCD_Control();

        //display on
        LCD_WR_REG(0x0029);
        LCD_Control();

        //Memory write
        LCD_WR_REG(0x002c);
        LCD_Control();

}
bool trdb_d5m_write(i2c_dev *i2c, uint8_t register_offset, uint16_t data) {
    uint8_t byte_data[2] = {(data >> 8) & 0xff, data & 0xff};

    int success = i2c_write_array(i2c, TRDB_D5M_I2C_ADDRESS, register_offset, byte_data, sizeof(byte_data));

    if (success != I2C_SUCCESS) {
        return false;
    } else {
        return true;
    }
}

bool trdb_d5m_read(i2c_dev *i2c, uint8_t register_offset, uint16_t *data) {
    uint8_t byte_data[2] = {0, 0};

    int success = i2c_read_array(i2c, TRDB_D5M_I2C_ADDRESS, register_offset, byte_data, sizeof(byte_data));

    if (success != I2C_SUCCESS) {
        return false;
    } else {
        *data = ((uint16_t) byte_data[0] << 8) + byte_data[1];
        return true;
    }
}
```