École polytechnique fédérale de Lausanne

CS-473 Embedded Systems
Mini-project
January 2022

LCD controller and camera implementation on Terasic DE0-Nano-SoC

LCD
Nguyen Thanh Vinh Vincent
Bakkali Yassine

Camera
Simik Martin
Pavliv Taras

Contents

1 Introduction

In this mini-project, we present the implementation of our conceptual design that we developed in the lab 3.0. This project consists in the design of a custom LCD controller for the TerasIC LT24 - 2.4" LCD touch module, the design of a camera controller for the TRDB-D5M camera and the interfacing of those two systems. This report will detail the design of our own custom IP components, whose aim are to store the data from the camera in RGB565 in the SDRAM, and then to read and display it. Both controllers are interfaced with an Avalon bus and are programmable through an Avalon slave interface. Each group designed a master interface (DMA) to enable fast data transfer with the SDRAM.

2 System overview

The overall system is as follows: a nios II processor and its on-chip memory will be used to run the system and can be programmed through the JTAG UART interface. The Avalon bus is also interfaced with an address span extender, itself connected to a memory controller to control the external SDRAM (see lab 4 assignment). The FPGA is connected to the D5M Camera through its GPIO1 pins (see section 3.5) and to the LT24 Screen through its GPIO0 pins (see section 4.6). The I2C module was provided to us, so the IPs we will develop in this lab are the Camera controller and the LCD controller.

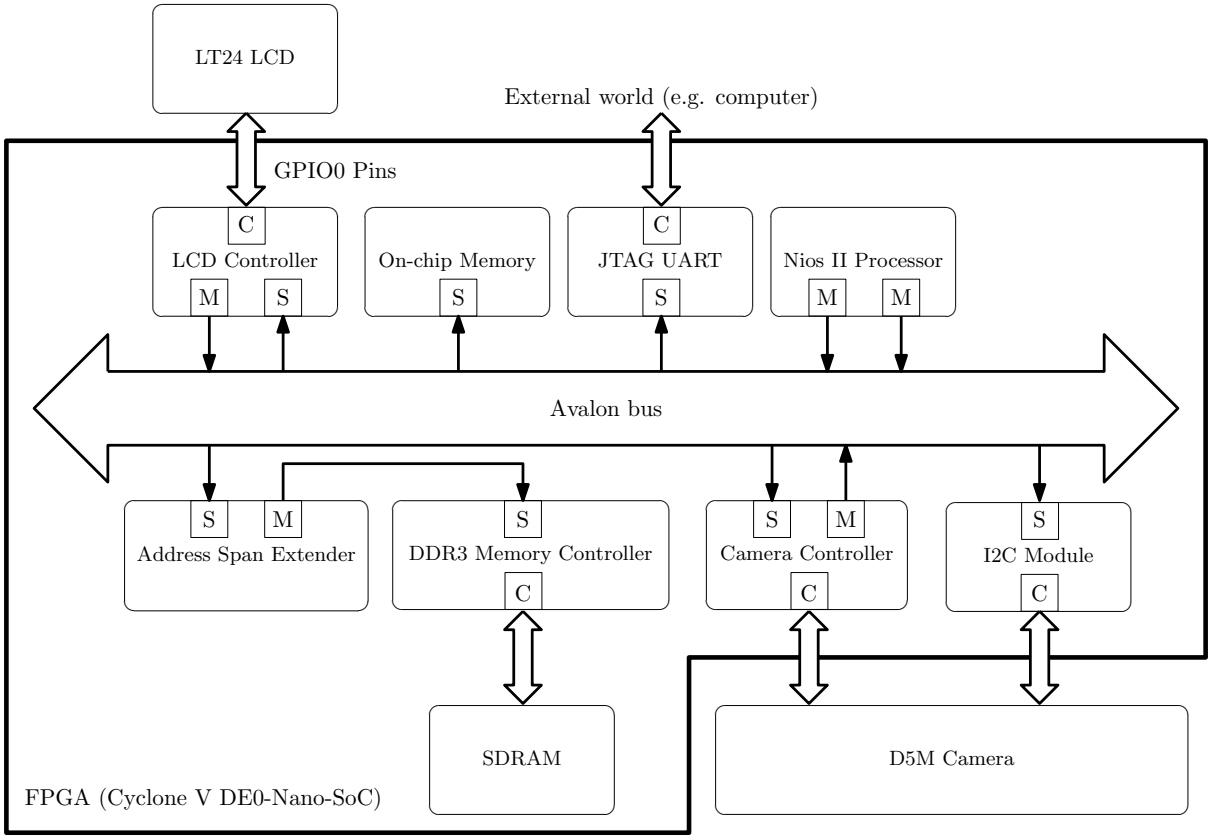


Figure 1: General system overview.

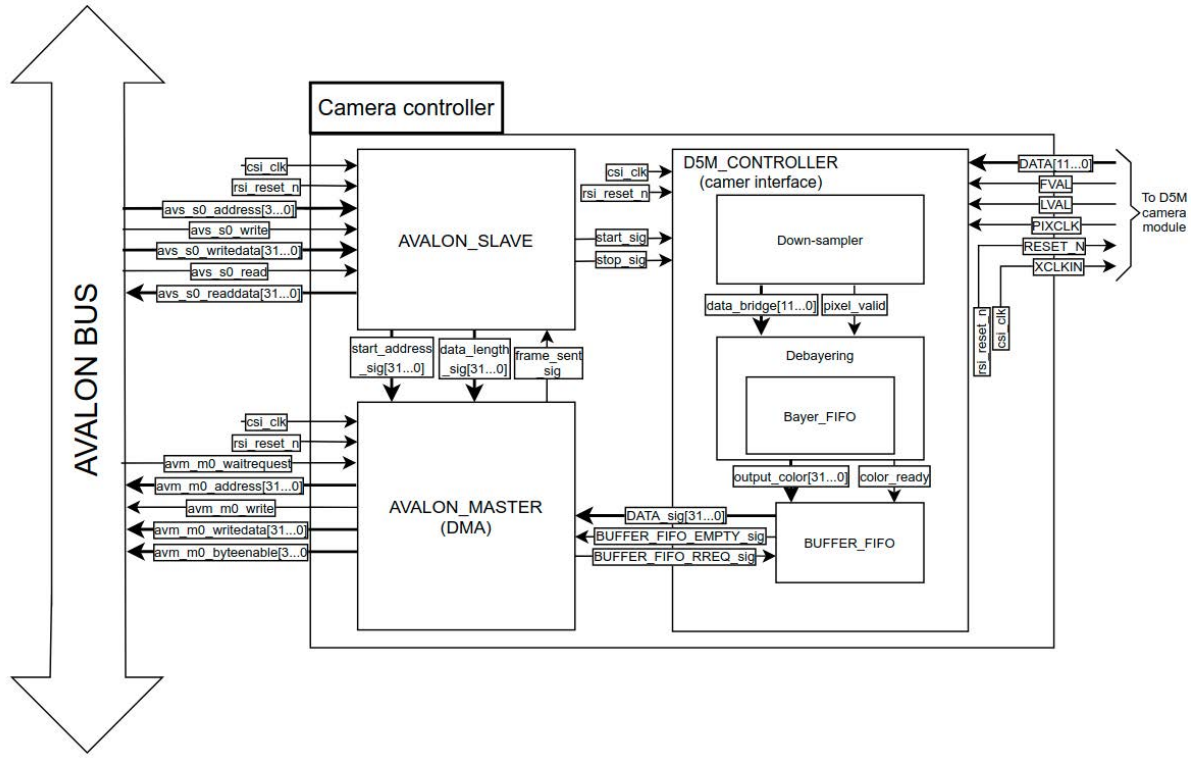


Figure 2: Block diagram of the camera controller. Note that inside the D5M_CONTROLLER not all signals are shown for clarity purposes.

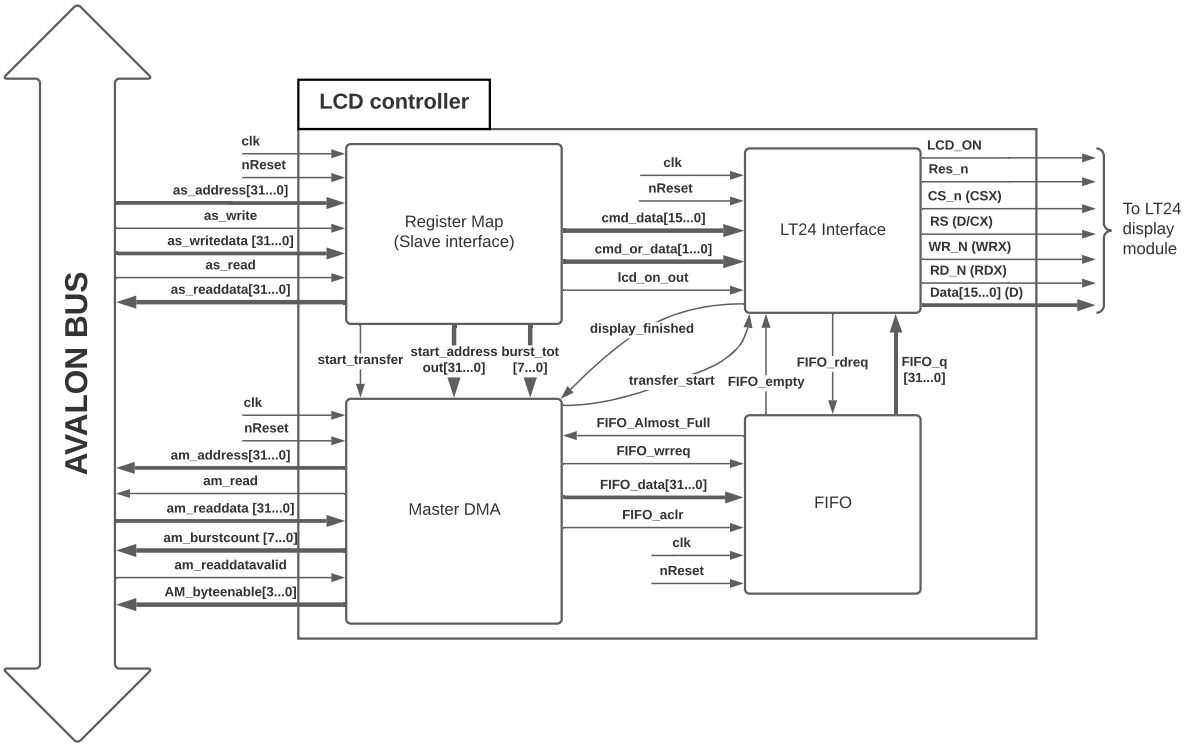


Figure 3: Block diagram of the LCD controller.

3 Camera Hardware

3.1 Difference with conceptual design

1. The memory organization changed, it is less space efficient but easier to implement. It now takes 4 bytes per color (instead of 3).
2. A software trigger is used instead of the trigger input pin.
3. The snapshot mode signal = 1 and is a constant.
4. The length of the picture data is equal to 4×320×240 by default, it changed due to the memory organization change.
5. We don't do burst write, so the finite state machine of the DMA changed accordingly, and the writing is done in the "Waiting Write" state.

3.2 Slave interface

In the table 1 is the register map of the camera Avalon Slave Interface. It consists of basic commands to control the acquisition of the camera, as well as commands to control the Master Interface, such as the address where it should store the image and the size of the image (the size is normally constant in this project). It also features a register to read if a full image was sent to memory, which can be used to know when another device can safely read the image. Note that this register is read-only, and will reset automatically when someone reads it.

3.2.1 Register map

Address	Register	Reset Value	Size	Description	Access
00h	`CamAddr`	0h	32 bits	Destination Address	R/W
04h	`CamLength`	320×240×4	32 bits	Buffer size in Bytes	R/W
08h	`CamStart`	0h	8 bits	Acquisition Enabled	R/W
0Ch	`CamStop`	0h	8 bits	Acquisition Disabled	R/W
10h	`CamSnapshot`	0h	8 bits	Snapshot Mode	R/W
14h	`FrameSent`	0h	8 bits	The bit is set when a whole image is sent to the SDRAM	R

Table 1: Camera Avalon slave register map.

3.3 TRDB-D5M Interface

3.3.1 Finite state machine

The Finite State Machine for the camera interface (also named the D5M_CONTROLLER in our project) is unchanged from the Lab 3.0, except for the names of the signals and states. It is controlled by the registers `CamStart` and `CamStop` for the signals `start_sig` and `start_sig` respectively. Being in the `IDLE` state is equivalent to continuously resetting the camera interface, so no pixels are processed.

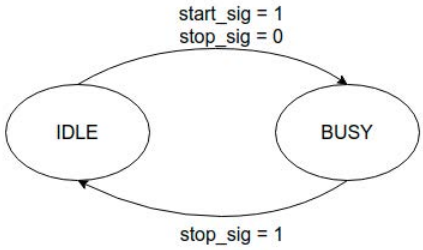


Figure 4: Camera interface FSM.

Pixel reading In the Figure 5, we see that the origin is at the top right corner, and from Figure 7 we know that it is read row by row, so from right to left and top to bottom. From Figure 7 we also notice that the first row we read will be composed of red and green (R and G1). That means that when reading the first row we first read the G1 pixel (in 12 bits, send over a bus of width 12, with the D11 pin carrying the MSB and D0 the LSB). Then we read the R color, then the G1 again etc. The speed at which the pixels will be read is dictated by the `PXL_CLK` signal that is outputted by the camera.

We configure (by software, see the subsection 6.1.2) the camera so that the colors are given on the rising edge of `PXL_CLK`. Because the downsampling process, the debayering process and the writing to the Buffer FIFO are all dependant on the speed at which the pixels are read, they will all be doing operations on the rising edge of the `PXL_CLK` signal, as opposed to the rest of the Camera Controller which works with the `csi_clk` given by the FPGA. From the Figure 5 we also notice that there are some padding with dark and boundaries pixels. We don't want to read them and only want pixels from the "Active Image" zone. The image acquisition is only enabled when the signals `LVAL` and `FVAL` are asserted by the camera.

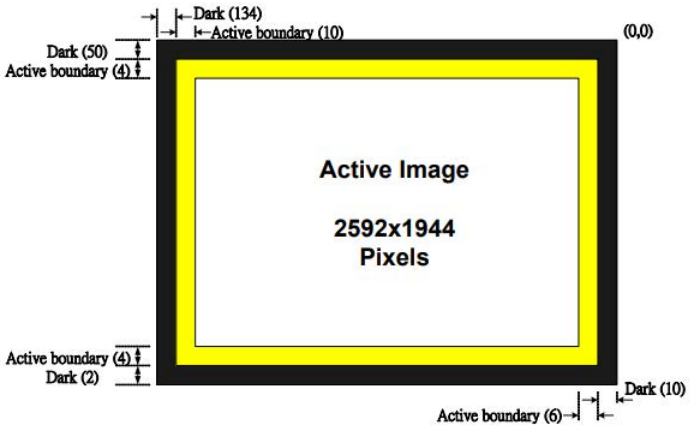


Figure 5: Pixel Array Description.[6]

Downsampling The minimum configurable resolution on the camera is 640x480, but a picture of 320x240 colors is needed to match the LCD screen resolution. We decided to skip pixels in order to achieve that resolution. When reading the first row, only the first G1 and R values are taken, the next R and G1 are skipped, the following ones are saved and so on. The downsampling process is done before the de-bayering.

Obviously, when reading the following row with B and G2, we need to skip the same corresponding values. When finished with reading the row two, the row 3 and 4 completely ignored, as a vertical downsampling is also needed. The skipped values can be summarized with Figure 6. To achieve this, we will use the signal `pixel_valid` which will be 0 if the pixel should be skipped and 1 otherwise.

For this we use three signals : `downsampling_col_counter`, `downsampling_row_counter` and `valid_row`. `downsampling_col_counter` counts from 0 to 3, but will say the pixel is valid only if the counter is 0 or 1. `downsampling_row_counter` counts from 0 to 2559, because that is the amount of pixels in one row of colors and will say that they are in a `valid_row`, then invert the `valid_row` and count the next 2560 pixels and say they are invalid, then invert the `valid_row` again and so on.

Figure 6: In dark are the pixels values that will be skipped over with the down-sampler.

Bayer Pattern Here, we work only when `pixel_valid` = 1. We want to save the pixels in an RGB565 encoding, as described in the section 5. But because of the Bayer pattern used, shown in Figure 7, we need to read the whole first row before reading the blue value of the first color. A `Bayer_FIFO` is used for this purpose. It needs to be at least 640 deep (320 for red pixels and 320 for green 1 pixels). Because we can only choose the depth to be a power of 2, we chose the FIFO to have a depth of 1024, for data of width 12 (the same width as given by the camera).

When reading the second row, the value of G1 is popped when receiving the B value. We truncate them to only keep the 5 most significant bits, and store them in `output_color`. The next clock, R value is popped when receiving the G2. We truncate them too to 5 bits, but the 5-bit G2 value is added to the already stored 5-bit G1 value, so the result is in 6-bit. At the same time, we set the signal `color_ready` to 1 to push the `output_color` in the `Buffer_FIFO`.

To know when we should fill the `Bayer_FIFO` and when we should empty it (by popping values), we introduce the signal `push_state`. For `push_state` = 1 we will be pushing in the FIFO and for `push_state` = 0 we will be popping from it. It will know when to change its state by keeping track of how much the FIFO is full, with the `usedw` signal provided by the FIFO. If it reaches 640, it means we should start popping from it, then when it reaches 0 we should start filling it again.

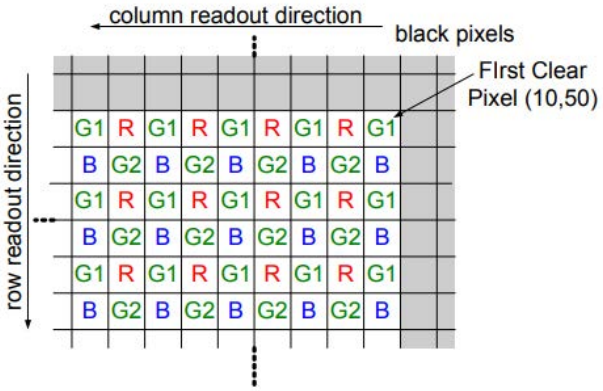


Figure 7: Pixel Color Pattern Detail (Top Right Corner).[6]

Buffer FIFO The de-bayering filter gives us 32-bits for the RGB value of a color (only the 16 last bits are used), and we need to send it to the SDRAM, as will be discussed in the subsection 3.4. We know that the DMA will send this data, but we don't know when, as it depends on when the Avalon bus will be free to use (when `avm_m0_waitrequest` is 0), which is out of our control. A `Buffer_FIFO` is used to store data that comes out of the de-bayering filter. It has a width of 32 bits, because each color is stored in 32 bits, and a depth of 512, which is too much for our application but we wanted to be sure that's not the problem when debugging.

As soon as the FIFO is not empty, the DMA will try to send data to through the Avalon bus, and will continue until the FIFO is empty again. It is described more precisely in the subsection 3.4.

3.3.2 Simulations

Figure 8 shows the Modelsim simulated signals of the camera interface IP. `FVAL` and `LVAL` are asserted, the camera gives a portion of the image that the interface must process. All the signals are timed by the camera clock `PIXCLK`. The signal `usedw_sig` shows the `Bayer_FIFO` utilization. When the FIFO utilization is under 640 the FIFO is in push state and accumulates only pixels when `pixel_valid` is asserted. When it reached 640 it starts to empty as described in the **Bayer Pattern** paragraph. `color_ready` is asserted when an `output_color` is ready to be sent to the `Buffer_FIFO`.

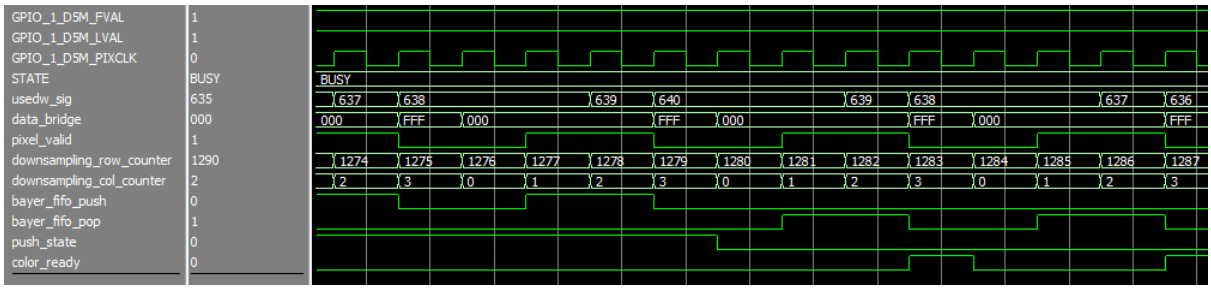


Figure 8: Camera interface signals simulated in Modelsim.

3.4 Master DMA

3.4.1 Finite state machine

The Avalon master interface works as follow: if the picture length in the register `CamLength` of the Avalon Slave is equal to zero it stays in `IDLE`. If the length is larger than zero, the FSM change its state to `WAITING_DATA`. When the Camera Interface Bayer FIFO is not empty anymore, the DMA changes state to `WAITING_WRITE` and asserts `avm_m0_write`, where it waits until the Avalon bus becomes unused i.e. `avm_m0_waitrequest` = 0. As soon as it is the case, it means it have written the data from the Avalon Master so the data is acknowledged to the `Buffer_FIFO` and it loads the next data (color). It also increases the next address to write to by 4, as we have written 32 bits so 4 bytes, and decreases amount of data that is left to write by 4, for the same reason. Those values are stored in signals `startAddr_bridge` and `datalength_bridge` respectively. As soon as the FIFO is empty again returns to the `WAITING_DATA` state.

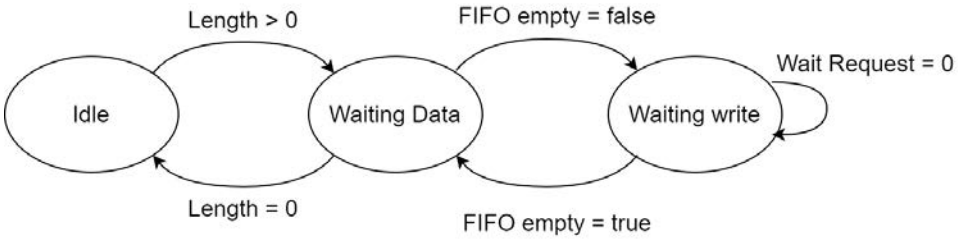
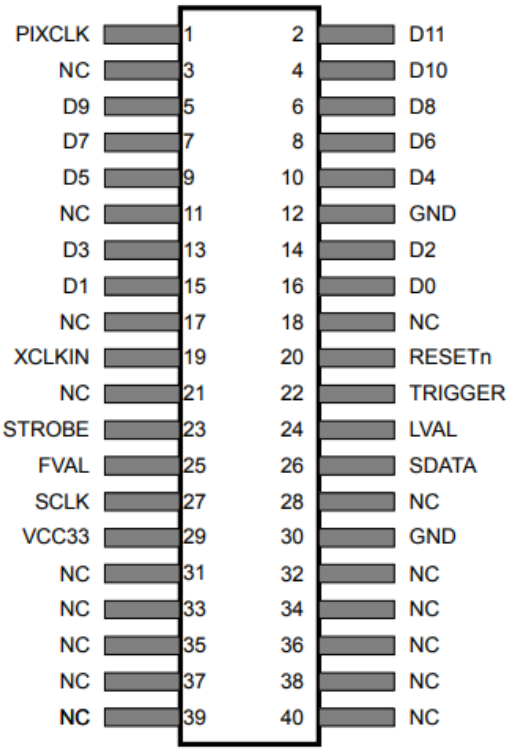


Figure 9: Camera master interface FSM.

When `datalength_bridge` reaches 0, that means a full image was sent, so the corresponding register is set in the Avalon Slave. It also means we can load the new address and the new length (which should still be 320x240x4) from the corresponding Avalon Slave's registers and be ready to store the next image in the new address.

3.5 Pinout

The only requirement imposed for the camera-FPGA interface is to use the GPIO 1 connector for the camera. The pinout is the following one:



FPGA Pin	Physical Pin	I/O	Signal Name
GPIO_1(0)	1	O	PIXCLK
GPIO_1(16)	19	I	XCLKIN
GPIO_1(17)	20	I	RESETn
GPIO_1(22)	25	O	FValid
GPIO_1(21)	24	O	Lvalid
GPIO_1(23)	26	I	SDATA
GPIO_1(24)	27	I	SCLK
GPIO_1(13)	16	I	Data(0)
GPIO_1(12)	15	I	Data(1)
GPIO_1(11)	14	I	Data(2)
GPIO_1(10)	13	I	Data(3)
GPIO_1(9)	10	I	Data(4)
GPIO_1(8)	9	I	Data(5)
GPIO_1(7)	8	I	Data(6)
GPIO_1(6)	7	I	Data(7)
GPIO_1(5)	6	I	Data(8)
GPIO_1(4)	5	I	Data(9)
GPIO_1(3)	4	I	Data(10)
GPIO_1(1)	2	I	Data(11)

Table 2: Camera pinout.

4 LCD Hardware

4.1 Difference with conceptual design

The main difference with the conceptual design is that we have added more signals between the different components of our LCD controller, which were necessary to ensure proper communication between the components. The finite state machines of both Master DMA and LT24 Interface components were slightly changed accordingly, and more registers were added to the Avalon slave interface.

4.2 Slave interface

This interface is used to program the LCD controller. It is used to specify the start address of our image stored in the SDRAM, the total number of bursts per burst transfer and also to send commands and data to the LCD.

4.2.1 Register map

Address	Write register	Bits	Description	Access
000	`RegStartAddress`	`[0-31]`	Start address in the SDRAM of the frame to display (address of top left pixel).	R/W
001	`RegCmdOrData`	`[0-1]`	`'10'`: data, `'01'`: command `'00'`: no op	R/W
010	`RegCmdData`	`[0-15]`	Command or data to send to LCD (e.g. '2C' to start the transfer).	R/W
011	`RegBurstTot`	`[0-7]`	Number of bursts for each transfer.	R/W
100	`RegLCDON`	`0`	Turn the LCD ON/OFF (`'1'` = ON, `'0'` = OFF).	R/W

Table 3: LCD Avalon slave register map. All registers can be read or written to.

4.2.2 Simulation

The LT24 interface sends back a `cmd_data_ready` signal when a write sequence is completed, which is used to reset the `RegCmdOrData` register (to avoid having it trigger another write sequence). This behavior can be seen in the simulation below.

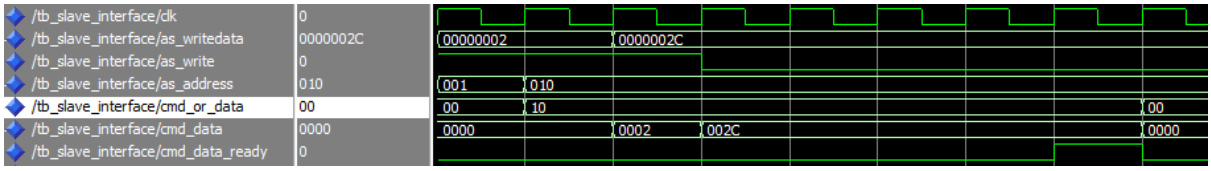


Figure 10: Example of a write command request from the slave to the LT24.

4.3 FIFO

We use for this component a single clock FIFO, with a data width of 16 bits (which is the same width as what the LT24 expects for 1 pixel of data). The design of the FIFO will not be detailed here, as we will use an automatically generated one in Quartus. The FIFO's data width is of 256 words. To synchronize the data transfer from the Master DMA to the FIFO and finally to the LT24 interface, we used the `FIFO_empty` and `FIFO_almost_full` signals, the latter having a word limit of 224. We can thus store up to 224 pixels before the `almost_full` signal activates and stalls the transfer. As we use the FIFO in normal mode, the `FIFO_wrreq` and `FIFO_rdreq`

signals both have a latency of one clock period whenever it respectively writes data within the FIFO or we want to read data from it.

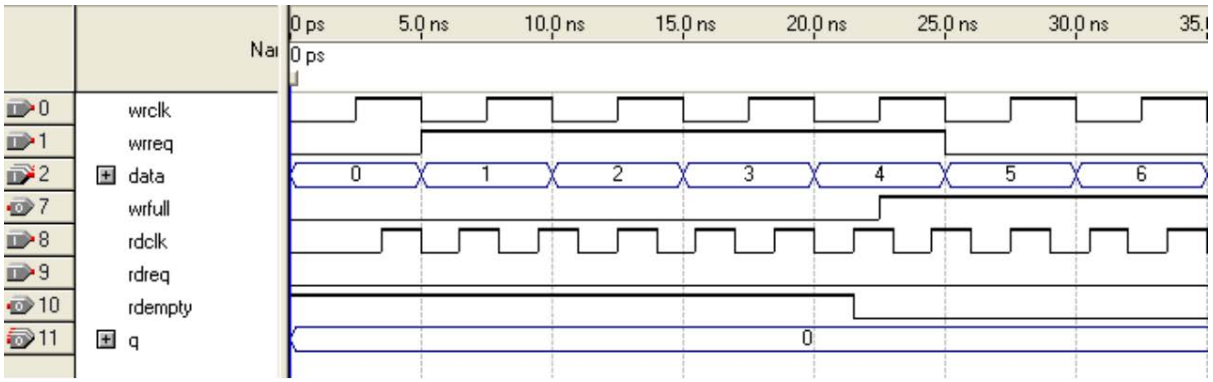


Figure 11: Detailed timing of the writing process of the SCFIFO [2].

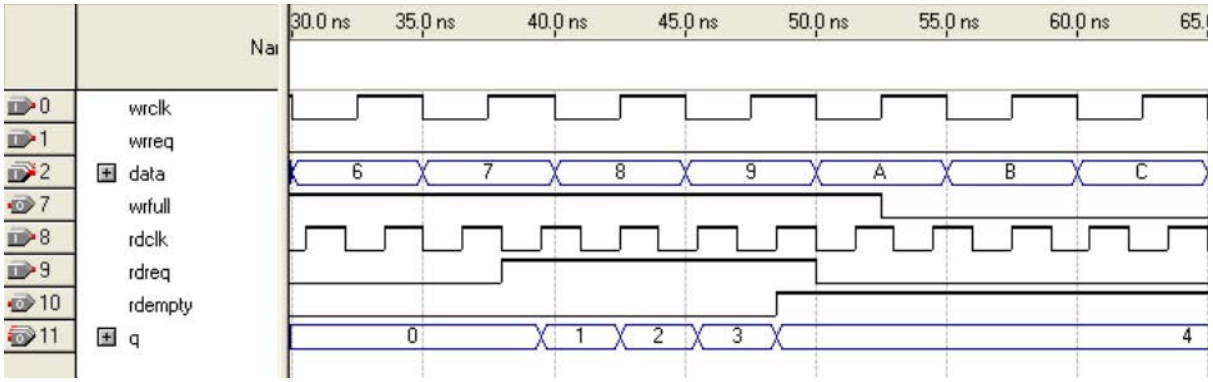


Figure 12: Detailed timing of the reading process of the SCFIFO [2].

4.4 Master DMA

The master interface is in our case the DMA (Direct Memory Access) unit. It is used to transfer large quantities of data rapidly from the SDRAM to the FIFO.

4.4.1 Finite state machine

When enabled, it first starts initializing its registers before reaching the `IDLE` state. In this state, the DMA waits for the `LT24_interface` to send the `start_in` signal, which will initiate the data transfer, the starting address of the memory `start_address` and the transfer burst count before switching to the `READ_REQUEST` state. This state will monitor the `wait_request` signal and the state of the FIFO before reading data. If the FIFO happens to be almost full then the transfer will stop and wait for the FIFO to empty itself before resuming. Otherwise, if the FIFO is empty and there is no wait request, then we will reach the `READ_DATA` state.

The `READ_DATA` state ensures that the right number of bursts is applied during the transfer. Before the beginning of each burst, it checks if the `am_readdatavalid` signal is set which infers that the data on the `am_readdata` lines are inline with the address we want to access, hence we can safely read the data. After `burst_tot` = 8 bursts. After the first bursts of data has been successfully written onto the FIFO, we set the `transfer_start` signal which will inform the `LT24_interface` that it can start reading data. We then update the `current_address` the DMA reads from in the `VERIFY_ADDRESS` state and return to `READ_REQUEST` until we finish reading all the stored pixels in the SDRAM.

In this later state, the unit keeps track of the current address we are reading from. After every burst transfer, it increments its address by `inc_add` = `address_width_byte` × `burst_tot` = `32` with `address_width_byte` = `4` the width of an address. The transfer is terminated once the current address has reached the final address and we return to the `INIT` state to prepare for the next frame. We can calculate the last address to reach given the burst length with the following formula:

$$\begin{aligned} \texttt{end\_add} &= \texttt{start\_add} + \frac{\texttt{IMG\_WIDTH} \cdot \texttt{IMAGE\_HEIGHT} \cdot \texttt{inc\_add}}{\texttt{burst\_tot}} - \texttt{BITS\_PER\_TRANSFER} \\ &= \texttt{start\_add} + 307168 \end{aligned}$$

With `IMG_WIDTH` = 320 and `IMG_HEIGHT` = 240 being the resolution of the displayed image in pixel, `start_add` the address of the first pixel to read and `BITS_PER_TRANSFER` the number of bits per transfer (= `inc_add` × `burst_tot` = 32).

When the transfer is finished, the master will stay in the `WAIT_DISPLAY` state until the `LT24_interface` finishes reading the FIFO to avoid flushing it in the next state. Once it is done, we return to the `INIT` state.

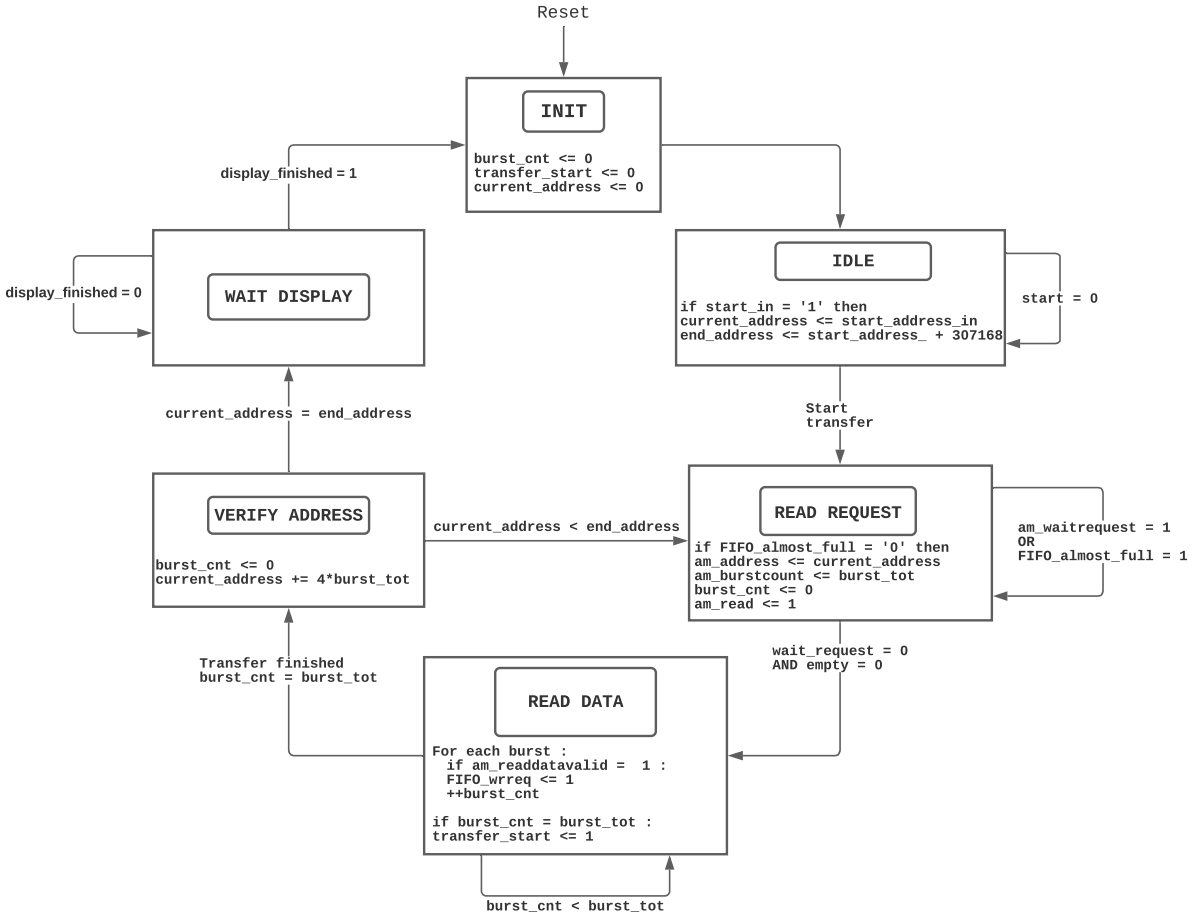


Figure 13: Master DMA interface finite state machine.

4.4.2 Simulation

Below will be presented some simulations that were conducted using ModelSim.

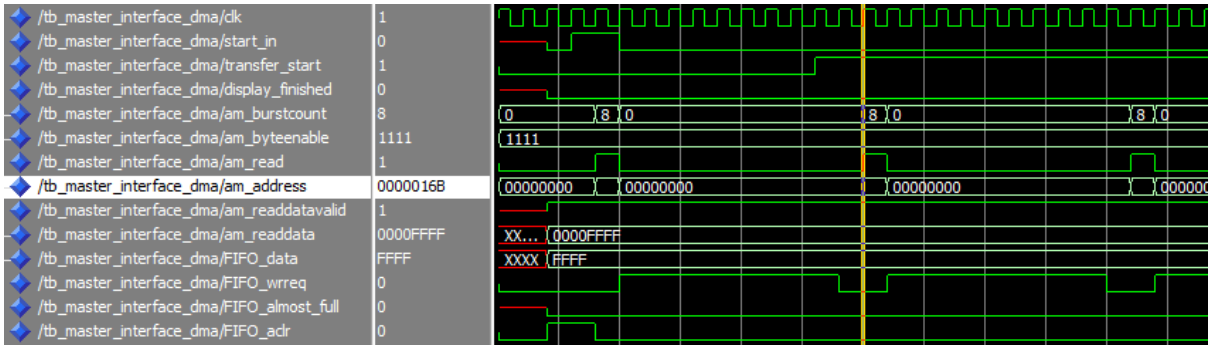


Figure 14: Simulation of start of transfer.

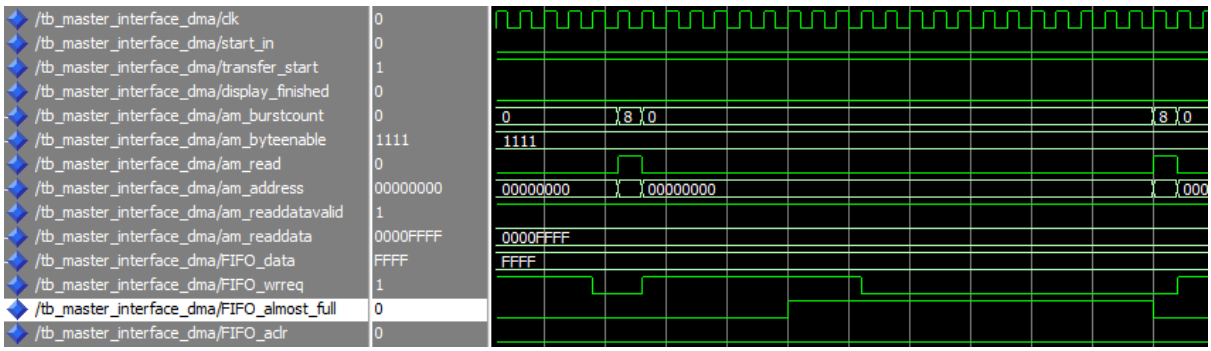


Figure 15: Simulation of almost full FIFO during transfer.

4.5 LT24 interface

This component is designed to follow the 8080 I timing specifications. Below are tables describing in more details the different functions of the 8080 protocol, and its timing constraints [4, p.27].

0	0	0	1	8080 MCU 16-bit bus interface I	"L"	↑	"H"	"L"	Write command code.
					"L"	"H"	↑	"H"	Read internal status.
					"L"	↑	"H"	"H"	Write parameter or display data.
					"L"	"H"	↑	"H"	Reads parameter or display data.

Table 4: Different functions of the 8080 MCU 16-bit bus interface.

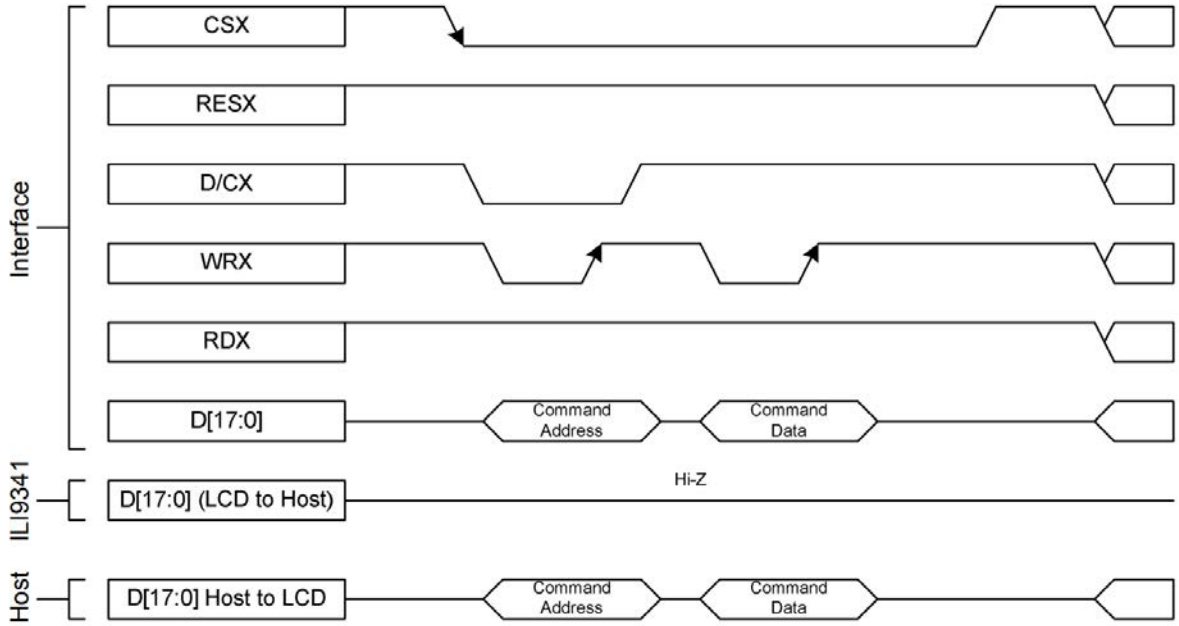
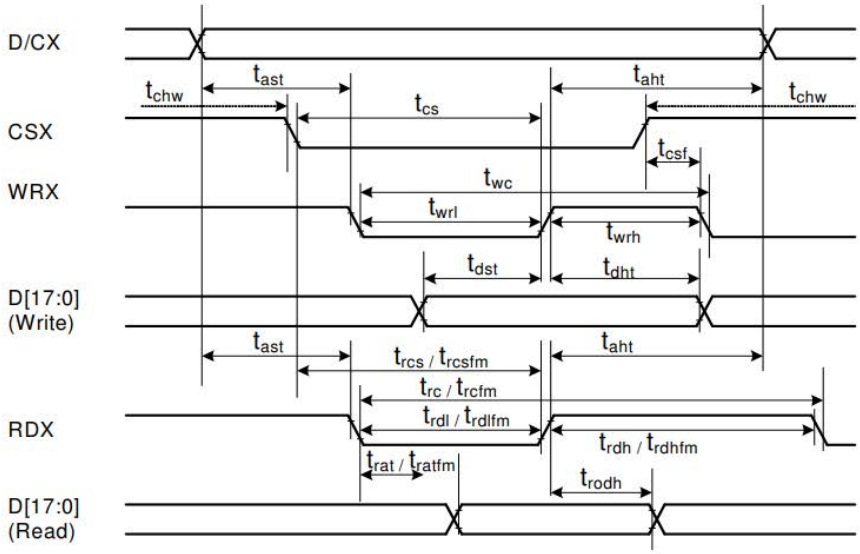


Figure 16: Example of a write sequence.



Signal	Symbol	Parameter	min	max	Unit	Description
DCX	tast	Address setup time	0	-	ns	
	taht	Address hold time (Write/Read)	0	-	ns	
CSX	tchw	CSX "H" pulse width	0	-	ns	
	tcs	Chip Select setup time (Write)	15	-	ns	
	trcs	Chip Select setup time (Read ID)	45	-	ns	
	trcsfm	Chip Select setup time (Read FM)	355	-	ns	
	tcsf	Chip Select Wait time (Write/Read)	10	-	ns	
WRX	twc	Write cycle	66	-	ns	
	twrh	Write Control pulse H duration	15	-	ns	
	twrl	Write Control pulse L duration	15	-	ns	
RDX (FM)	trcfm	Read Cycle (FM)	450	-	ns	
	trdhfm	Read Control H duration (FM)	90	-	ns	
	trdlfm	Read Control L duration (FM)	355	-	ns	
RDX (ID)	trc	Read cycle (ID)	160	-	ns	
	trdh	Read Control pulse H duration	90	-	ns	
	trdl	Read Control pulse L duration	45	-	ns	
D[17:0], D[15:0], D[8:0], D[7:0]	tdst	Write data setup time	10	-	ns	For maximum CL=30pF For minimum CL=8pF
	tdht	Write data hold time	10	-	ns	
	trat	Read access time	-	40	ns	
	tratfm	Read access time	-	340	ns	
	trod	Read output disable time	20	80	ns	

Figure 17: Detailed timings of the 8080-I protocol.

In order to satisfy these timing constraints, we will need to pay attention to the following parameters: t_{wrl}, t_{wrh}, t_{wc}. We will for this project use the 50 MHz clock (`FPGA_CLK1_50` [3, p.22]). Consequently, we will need to use at least 1 clock cycle for t_{wrl} and t_{wrh}, and 4 clock cycles for t_{wc}. Because the latter is at least the sum of the two former, we choose to have a write cycle executed in 4 clock cycles, i.e. 80 ns. The total time taken for a write sequence is however 5 cycles, as there are additional signals that need to be asserted once the write sequence is completed. A simulation of a write sequence can be seen in figure 19 and the actual signals can be seen in section 8.

4.5.1 Finite state machine

The LT24 Interface component stays in the `INIT` state as along as it hasn't received a command from the Avalon slave (`cmd_or_data`).

Once it has received the correct command, it transitions to the corresponding state. If it is a command other than `'2C'` or if it is a write data command, it goes back to the `INIT` state

after completion.

If a write command is requested and it corresponds to the display command `'2C'`, the component sends a `start_out` signal to the master DMA and then goes to the `WAIT_TRANSFER` state (see figure 20). In that state, the component waits for a `transfer_start` signal from the master DMA which indicates that it is ready (the DMA has to send a write request to the FIFO).

When both the master DMA and the LT24 interface are ready to display, the component goes to the `CHECK_FIFO` state. This state checks if the FIFO is empty (stays in that state if it is, see figure 22), and sends a read request if it is not. When the FIFO is not empty, we can display its content to the LT24 (`DISPLAY` state). Once a pixel is displayed, the component goes to the `CHECK_TRANSFER` state which checks if the whole image was displayed and increments a counter that keeps track of how many pixels are displayed. If the full image is not yet displayed, there are two cases: if the FIFO is not empty, it goes back to the `DISPLAY` state. If the FIFO is empty, it goes to the `CHECK_FIFO` state. If the full image is displayed, the component goes back to the `INIT` state after sending a `display_finished` signal pulse (see figure 21).

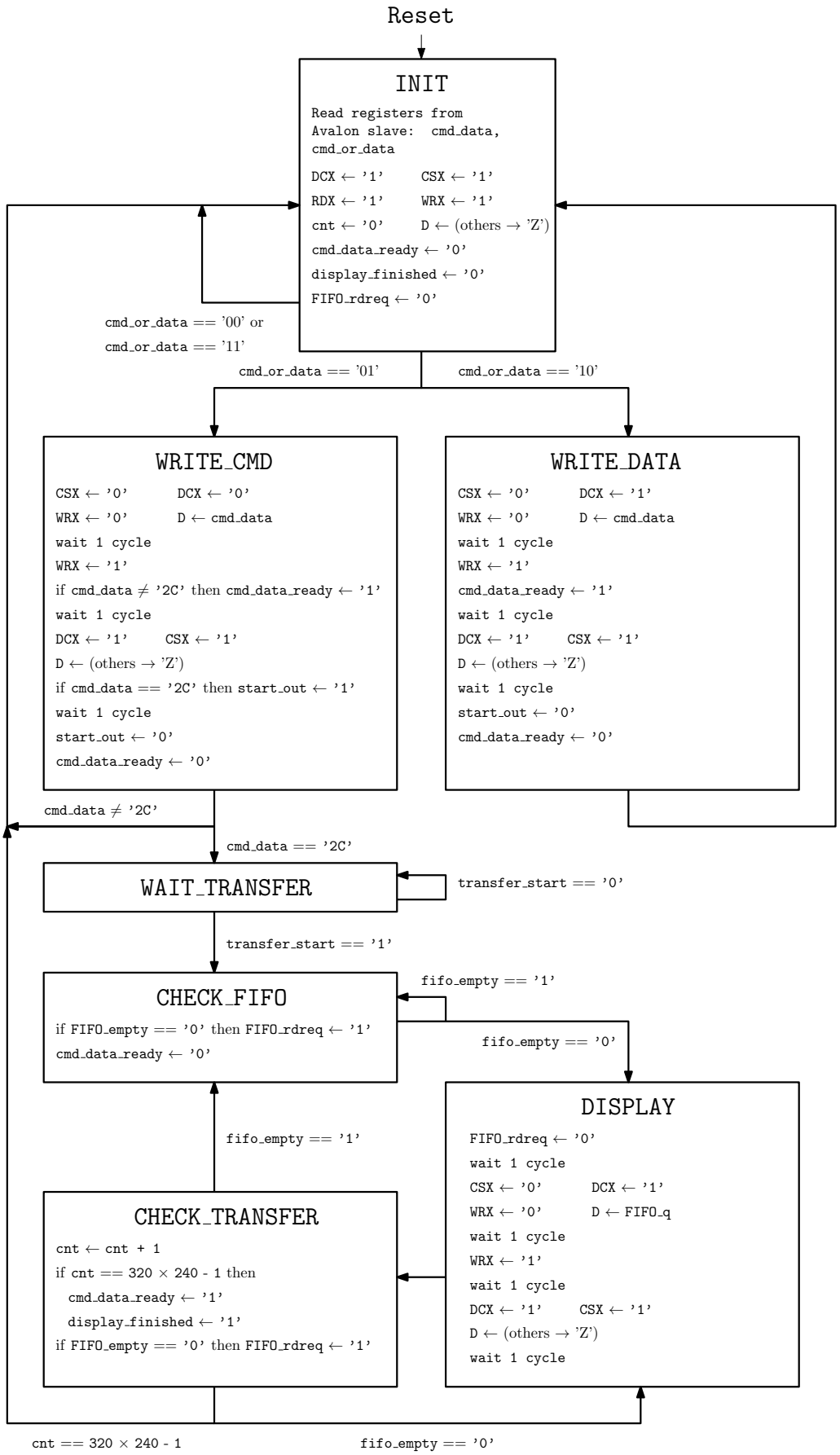


Figure 18: LT24 interface finite state machine.

4.5.2 Simulations

Below will be presented some simulations that were conducted using ModelSim.

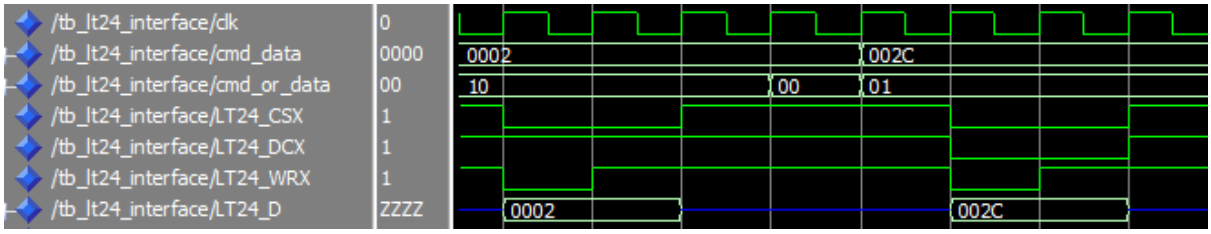


Figure 19: Simulation of a write sequence. Left: write data (0x002). Right: write command (0x02C).

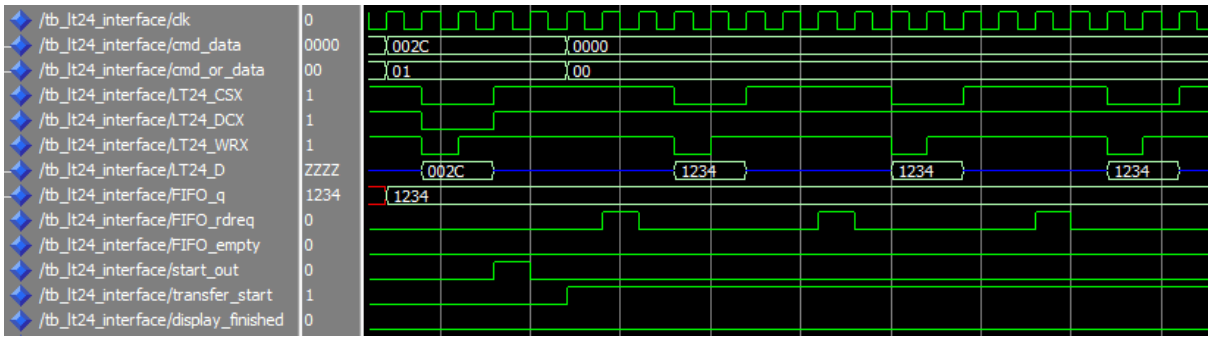


Figure 20: Simulation of start of transfer.

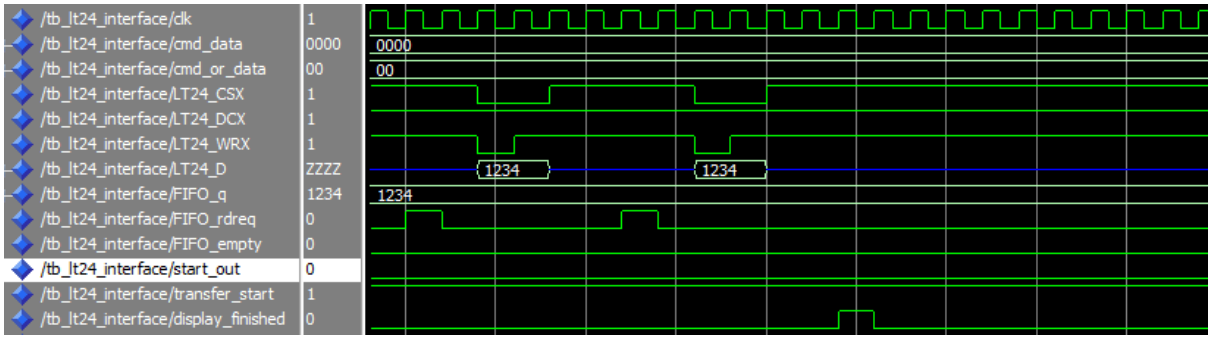


Figure 21: Simulation of end of transfer.

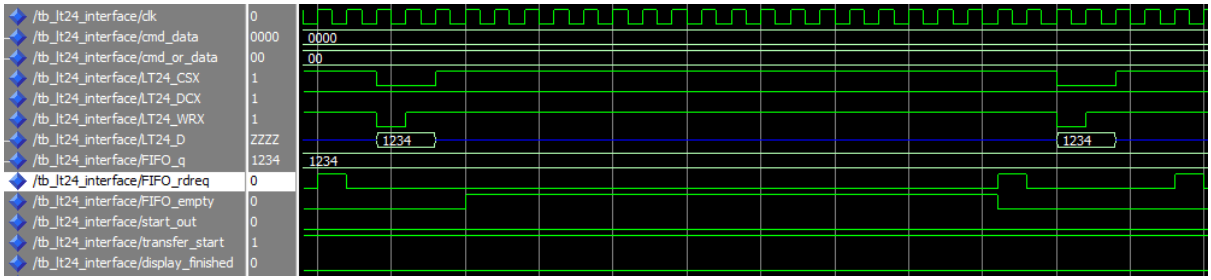
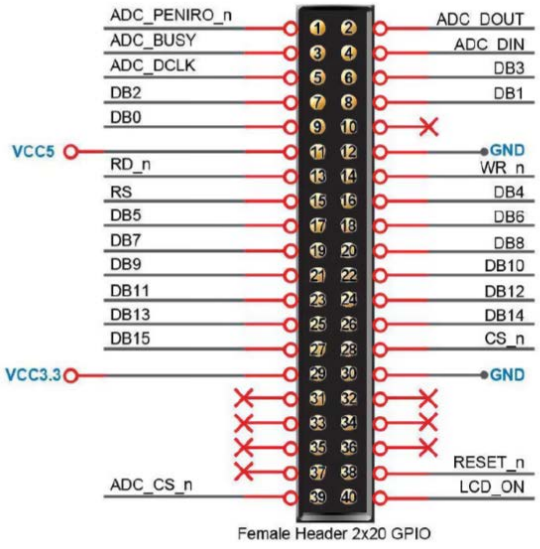


Figure 22: Simulation of empty FIFO during transfer.

4.6 Pinout

The LT24 is connected to the GPIO0 port of the DE0-Nano-SoC, as specified in the user manual [5, p.10].



FPGA Pin	Physical pin	I/O	Signal Name
GPIO_0(34)	39	O	CS_n
GPIO_0(35)	40	O	LCD_ON
GPIO_0(10)	13	O	RD_n
GPIO_0(33)	38	O	Reset_n
GPIO_0(12)	15	O	RS
GPIO_0(11)	14	O	WR_n
GPIO_0(8)	9	O	DB0
GPIO_0(7)	8	O	DB1
GPIO_0(6)	7	O	DB2
GPIO_0(5)	6	O	DB3
GPIO_0(13)	16	O	DB4
GPIO_0(14)	17	O	DB5
GPIO_0(15)	18	O	DB6
GPIO_0(16)	19	O	DB7
GPIO_0(17)	20	O	DB8
GPIO_0(18)	21	O	DB9
GPIO_0(19)	22	O	DB10
GPIO_0(20)	23	O	DB11
GPIO_0(21)	24	O	DB12
GPIO_0(22)	25	O	DB13
GPIO_0(23)	26	O	DB14
GPIO_0(24)	27	O	DB15

Table 5: LT24 LCD pinout.

5 Frame Buffer Structure

As we misunderstood each other while preparing the lab 3.0, the buffer structure has slightly changed from the conceptual design. We now use the following format: the image is in RGB565 format (red in MSB) instead of BGR565 format, one color per 32 bits, stored in the bits 15...0.

(Row, Column)	Bit				Address in SDRAM
	[0...4]	[5...10]	[11...15]	[16...31]	
(0, 0)	B0	G0	R0	Empty	0
(0, 1)	B1	G1	R1	Empty	4
...	...	...	...	Empty	...
(0, 319)	B319	G319	R319	Empty	1276
(1, 0)	B320	G320	R320	Empty	1280
(1, 1)	B321	G321	R321	Empty	1284
...	...	...	...	Empty	...
(1, 319)	B639	G639	R639	Empty	2556
...	...	...	...	Empty	...
(239, 0)	B76480	G76480	R76480	Empty	305920
...	...	...	...	Empty	...
(239, 319)	B76799	G76799	R76799	Empty	307196

Figure 23: Pixel organization inside of the SDRAM buffer.

6 Camera Software

The software to control our camera is divided in two parts: configuration and capture of the image.

6.1 Configuration

6.1.1 Configuration of the Camera Controller

As we have left configurable the address and the length of the image, we assign them the values as discussed with the group designing the LCD controller. Actually the data length shouldn't change and is set right by default, but we left it configurable to help in the case we had problems interfacing the camera with the LCD screen.

```
1 //Avalon configure image storage, address and length
2 IOWR_32DIRECT(0x10000840, 0x00, HPS_0_BRIDGES_BASE); // address
3 IOWR_32DIRECT(0x10000840, 0x04, 320*240*4); // data length
```

6.1.2 Configuration by I2C of the Camera

We need to configure the camera to match the hypothesis we emitted when designing the camera controller, that is to have a resolution of 640x480, with the data sampled on the rising edge of PIX_CLK.

One thing we didn't anticipate in the lab 3.0 is that while the C code initializes the camera, the acquisition will already start and when we will command our controller to start processing the pixels, the camera will already be giving the pixels of somewhere inside the image. That's why we decided to configure it to the Snapshot Mode and to use the software trigger to start the acquisition at almost the same time as we start processing with our camera controller.

```
//configure resolution
success &= trdb_d5m_write(&i2c, 0x003, 1919); //1920 colors rows
success &= trdb_d5m_write(&i2c, 0x004, 2559); //2560 colors columns
//sample colors on rising edge and divide clock by 2
success &= trdb_d5m_write(&i2c, 0x00A, INV_PX_CLK | DIV2_PX_CLK);
success &= trdb_d5m_write(&i2c, 0x022, COLOR_BINNING4); //row binning by 4
success &= trdb_d5m_write(&i2c, 0x023, COLOR_BINNING4); //column binning by 4

//snapshot mode and invert trigger
success &= trdb_d5m_write(&i2c, 0x01E, 0x4000 | INV_TRIG | SNAPSHOT_MODE);
```

We also tried to balance the colors, but as it doesn't give any valuable information about the design, we don't include it here. It can be read in the "camera_utils.c" file.

6.2 Starting the capture

To start the capture of the camera we first start the software trigger, then we configure the registers of the Camera Controller so that the Camera Interface is in the `BUSY` state. We do it in this order because after the software trigger we still have some time where `FVAL` or `LVAL` are low (as seen in the figure 5, we don't start in the "Active Image" zone), so the pixels are not valid anyway.

Here we use a while loop where we check every millisecond if the acquisition of the image is finished, but in a "real-case" scenario we would configure a timer to generate an interrupt after about 1.560 seconds (that's the time we measured with the variable `wait_t`), so that the CPU is free to do something else during this time.

```
void camera_capture(i2c_dev i2c){
        //start the capture
        trdb_d5m_write(&i2c, 0x00B, TRIGGER); //triggers the start of capture
        IOWR_32DIRECT(0x10000840, 0x0C,0); // stop = 0
        IOWR_32DIRECT(0x10000840, 0x08, 1); // start = 1

        int wait_t= 0;
        while(IORD_32DIRECT(0x10000840,0x14) == 0){
                usleep(1000);
                ++wait_t;
        }

        printf("waited for [ms]: %d ", wait_t);

        IOWR_32DIRECT(0x10000840, 0x08, 0); // start = 0
        IOWR_32DIRECT(0x10000840, 0x0C, 1); // stop = 1
}
```

7 LCD Software

To test our LCD controller design, we have developed some useful functions, which are divided into 3 categories: functions related to the LT24, functions related to the SDRAM and functions related to a test image generation.

7.1 LT24 LCD

7.1.1 Writing commands and data

To write to the LCD, the `RegCmdOrData` needs to be set to `'01'` for a data write and `'10'` for a command write. The desired command/data is written inside the `RegCommandData` register. A special command is used to start the display, which is the `'0x2C'` command. Because our Avalon slave width is 32-bits, we need to pad the 16-bits commands/data with zeros.

```
void LCD_write_data(uint32_t data) {
    // Write to RegCommandData
    IOWR_32DIRECT(LCD_CONTROLLER_0_BASE, ADDR_COMMAND_DATA, data);
    // RegCommandOrData = 10b to write data
    IOWR_32DIRECT(LCD_CONTROLLER_0_BASE, ADDR_COMMAND_OR_DATA, 0x00000002);
}

void LCD_write_command(uint32_t command) {
    // Write to RegCommandData
    IOWR_32DIRECT(LCD_CONTROLLER_0_BASE, ADDR_COMMAND_DATA, command);
    // RegCommandOrData = 01b to write data
    IOWR_32DIRECT(LCD_CONTROLLER_0_BASE, ADDR_COMMAND_OR_DATA, 0x00000001);
}

void LCD_display(void) {
    // 0x2C Display Command
    LCD_write_command(0x0000002C);
}
```

7.1.2 LT24 configuration

We use the configuration that was proposed in the lecture slides, with a few modifications to cater to our buffer organization and desired pixel order. The commands that differ from the lecture slides are listed below. These commands are used to define area of frame memory where MCU can access, to define pixel format and row/column order [4, p.111, 112, 127].

```
// Memory Access Control
LCD_write_command(0x00000036);
// BGR Order
// Change Row Address Order
// Exchange Row/Column
LCD_write_data(0x000000A8);

// Column Address Set
LCD_write_command(0x0000002A);
// SC[15:0] = 0x0000 = 0
LCD_write_data(0x00000000);
LCD_write_data(0x00000000);
```

```
// EC[15:0] = 0x013F = 319
LCD_write_data(0x00000001);
LCD_write_data(0x0000003F);

// Page Address Set
LCD_write_command(0x0000002B);
// SP[15:0] = 0x0000 = 0
LCD_write_data(0x00000000);
LCD_write_data(0x00000000);
// EP[15:0] = 0x00EF = 239
LCD_write_data(0x00000000);
LCD_write_data(0x000000EF);
```

It is also necessary to configure the registers of our LCD controller. This can be done by using the `IOWR_32DIRECT` macro.

```
void LCD_write_registers(void) {
    // Provide start address (RegStartAddress)
    IOWR_32DIRECT(LCD_CONTROLLER_0_BASE, ADDR_START_ADDRESS, REG_START_ADDRESS);

    // Provide number of bursts per transfer (RegBurstTot)
    IOWR_32DIRECT(LCD_CONTROLLER_0_BASE, ADDR_BURST_TOT, REG_BURST_TOT);
}
```

7.2 SDRAM

7.2.1 SDRAM memory access

Since the Avalon master has a width of 32-bits, we save 32 bits pixels on each address of the SDRAM with the same organization as described in 5. For testing purposes, we have added the possibility to import an RGB565 image directly into the SDRAM.

```
void SD_fill_range(uint32_t start_addr, uint32_t end_addr, uint16_t color) {

    if (end_addr <= END_ADDRESS) {
        uint32_t addr;
        for (addr = start_addr; addr < end_addr; addr += 4) {
            IOWR_32DIRECT(HPS_0_BRIDGES_BASE, addr, color);
        }
    } else {
        printf("Error: end address is outside of allowed zone.\n");
    }
}
```

7.3 RGB565 image generation

A simple program was developed to convert any PNG image to a binary RGB565 image with the correct dimensions. This program extracts the 5 MSB of the red and blue channels for each pixels, and the 6 MSB for the green channels. The bits are then arranged in the desired format for each pixel (format described in figure 23). This program can be found in the `".\sw\nios\application\img"`.

8 Results

8.1 Camera

8.1.1 Measured signals

The signals were measured using the Signal Tap Logic Analyzer in Quartus.

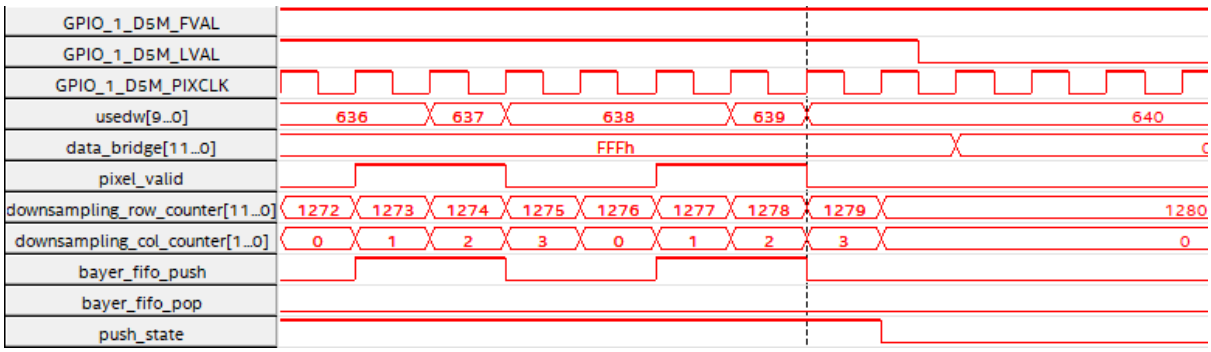


Figure 24: Measured signals at the end of the first line of pixels.

As we see in the figure 24, for the values of `usedw` up to 640, it is the same as in the simulation in the figure 8. However after that it stops. That's because it's the end of the line of the camera, `usedw` = 640 means that the Bayer FIFO has 320 red pixels and 320 green pixels stored in it. The difference comes from the fact that in our simulation, we left the `FVAL` and `LVAL` values at 1, so when the first line is finished it starts the next one right away, whereas in the real world that's not the case and `LVAL` goes low. As we see our system manages it correctly by stopping counting and waiting for the `LVAL` and `FVAL` values to be set again.

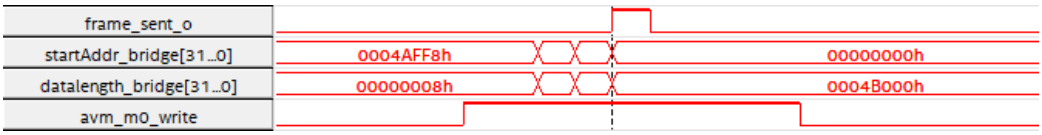


Figure 25: Measured signals when a frame is fully sent.

We didn't simulate the DMA as it would require to simulate the Avalon bus, so we were happy to measure that when a frame is sent, the system works as expected: the Avalon master stops writing, the address and the data length are updated with what is stored in the registers of the Avalon slave module and it's ready to capture again.

8.1.2 Example of images taken by the camera

(a) "Krtek" photographed with a phone camera.

(b) "Krtek" photographed with the TRDB-D5M camera.

Figure 26: Example of picture taken with the camera.

8.2 LCD

8.2.1 Measured signals

The signals of our design were measured using Quartus' Signal Tap Logic Analyzer.

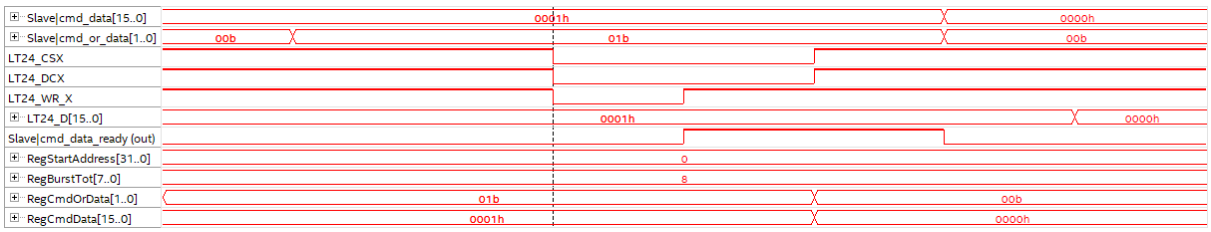


Figure 27: Measured signals of a write command sequence.

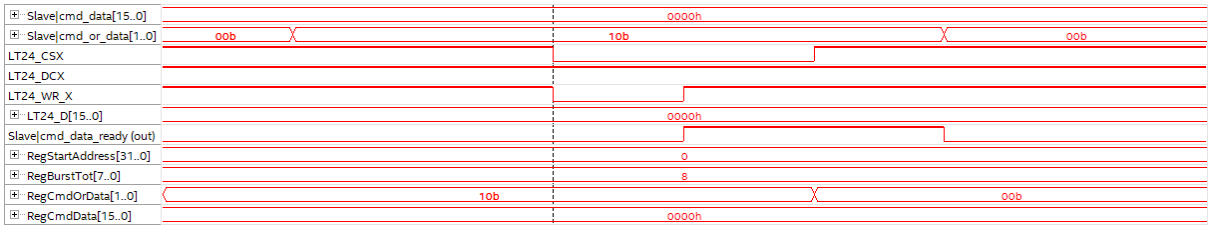


Figure 28: Measured signals of a write data sequence.

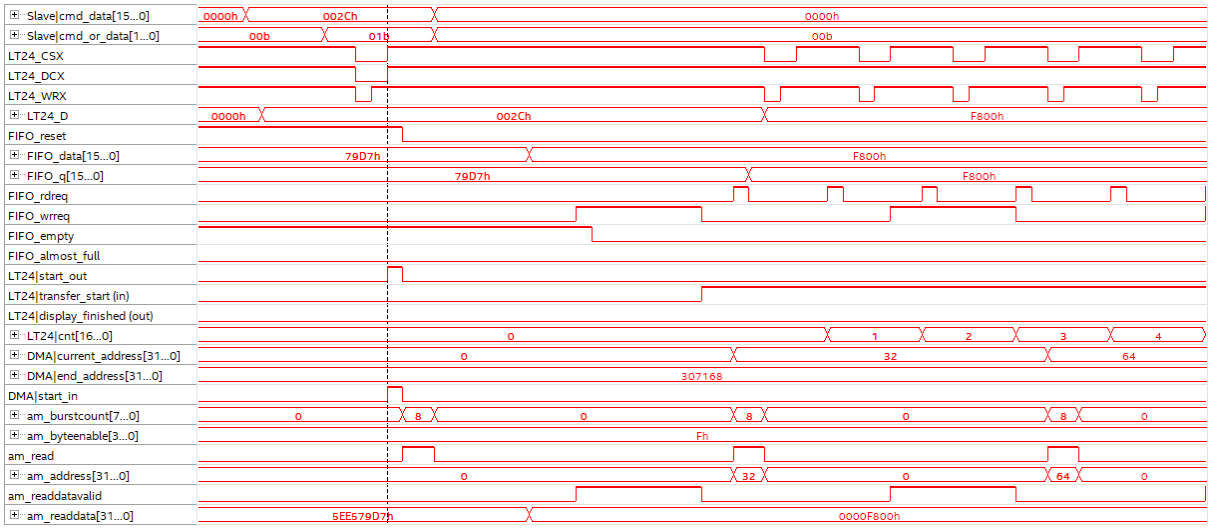


Figure 29: Measured signals of start of transfer sequence.

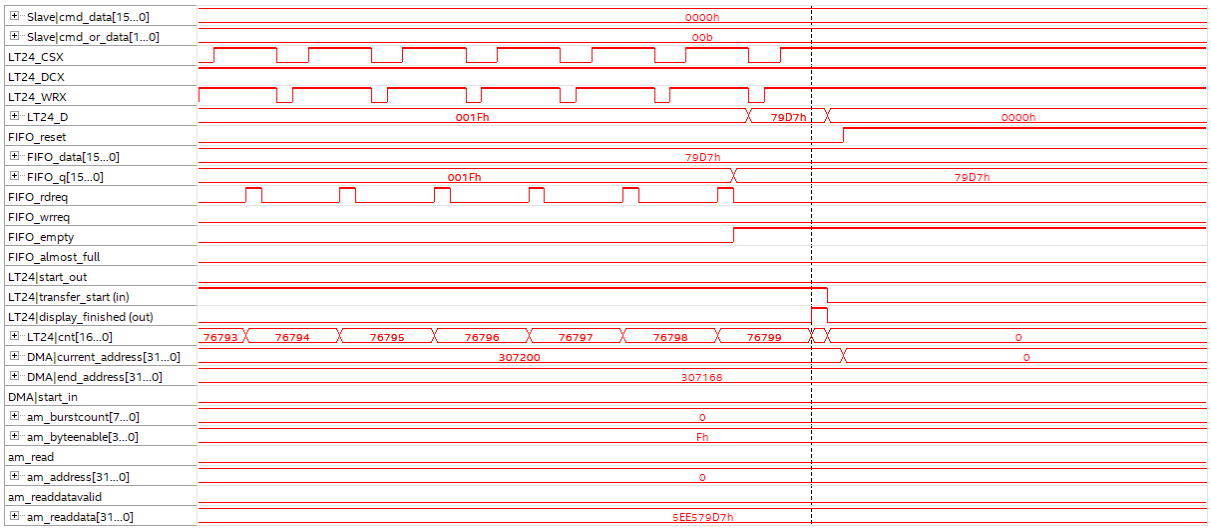


Figure 30: Measured signals of end of transfer sequence.

8.2.2 Example of displayed images on the LCD

(a) Displayed image 1.

(c) Displayed image 2.

(b) Reference image 1.

(d) Reference image 2.

Figure 31: Example of images displayed on the LCD and their respective reference images.

8.3 Example of picture taken with the camera and displayed on the LCD

(a) Picture of the whole setup.

(b) The object that we will take a picture of.

(c) The displayed image after taking the picture.

9 Conclusion

In conclusion, we were successfully able to design both systems, namely an LCD controller and a camera controller. The interfacing between the two systems was pretty straightforward and yielded good results. However, some improvements could be considered for future versions, such as video support and a more efficient frame buffer structure inside of the SDRAM.

References

[1] *5 Mega Pixel Digital Camera Development Kit for TRDB D5M*. [online; accessed 10 December 2021]. terasIC. URL: http://venividiwiki.ee.virginia.edu/mediawiki/images/a/ae/TRDB_D5M_UserGuide.pdf.

[2] *Avalon® Interface Specifications*. [online; accessed 09 December 2021]. Intel. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf.

[3] *DE0-Nano-Soc User Manual*. V.1.5. Terasic Inc., Altera. Dec. 2015. URL: https://www.terasic.com.tw/attachment/archive/941/DE0-Nano-SoC_User_manual.pdf.

[4] *ILI9341. a-Si TFT LCD Single Chip Driver 240RGBx320 Resolution and 262K color*. [online; accessed 05 December 2021]. ILITEK. URL: https://www.newhavendisplay.com/app_notes/ILI9341.pdf.

[5] *LT24 User Manual*. [online; accessed 01 December 2021]. Terasic Inc., Altera. URL: https://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=892&FID=527f33a451f2c9a404934446366f5342.

[6] *Terasic TRDB-D5M Hardware specification*. [online; accessed 8 December 2021]. terasIC. URL: https://moodle.epfl.ch/pluginfile.php/2536192/mod_resource/content/1/TRDB-D5M_Hardware%20specification_V0.2.pdf.