Embedded Systems

Lab 4.0

Richard Ismer

Dimitri Jacquemont

Sina Schaeffler

Nicolas Barker

August 31, 2023

1 Problem Statement

The purpose of this project was to design and implement LCD and Camera controllers for a FPGA. The LCD controller was designed to display video data on a screen, while the Camera controller was designed to capture and transmit this data. The camera and LCD controller need to be separate modules, and will therefore exchange data using the development board's memory.

The project began with the creation of a detailed report outlining the memory requirements and performance goals. From there, the VHDL code was written and simulated to ensure correct functionality.

For this project are used the DEO-Nano-SoC board, the LT24 LCD and the TRDB-D5M camera module. This report will provide a detailed overview of the design and implementation process for the LCD and Camera controllers, as well as the results of the testing and integration.

2 High-level Description

In order to run our custom slaves on the DEO-Nano-SoC, we are making use of the Nios 2 processor, a 32-bit embedded processor architecture designed specifically for the Altera family of FPGA integrated circuits. This IP component will be in charge of the communication between our two components.

All the high-level description is further discussed in the Lab 3.0 report, part of the mini project.

3 TRDB-D5M Camera Controller

3.1 Low-level Description

The exact functioning of some of the sub-components needed small updates. However, the overall architecture is still the same, and most components did not get significantly changed.

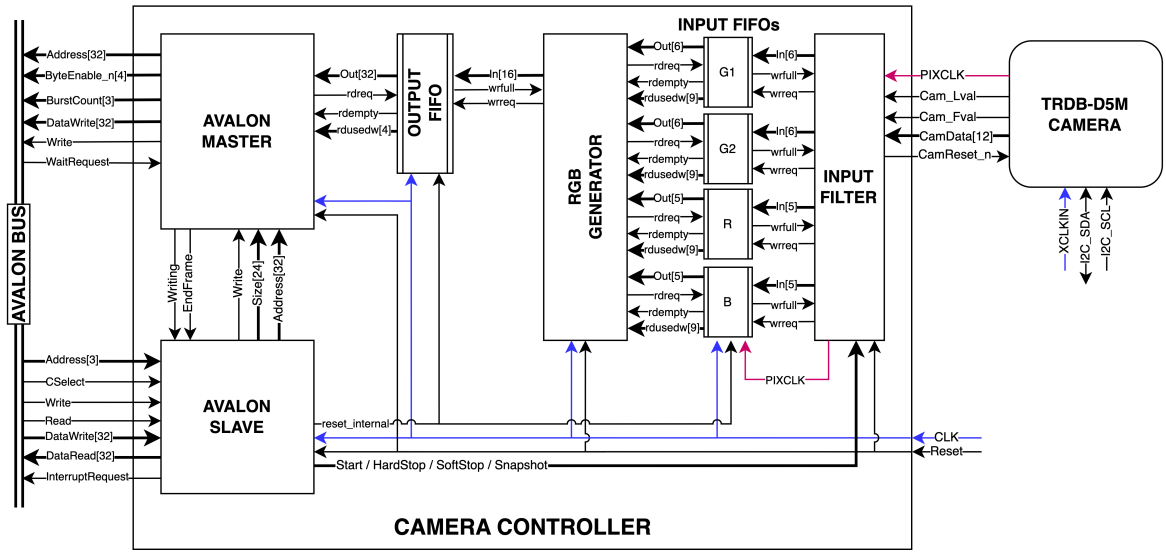

Figure 1: Reminder: The overall structure of the Camera_Controller

Here are the corresponding state-machine-like descriptions for the master and slave, since these are the ones with which the user will likely interact most.

Avalon Slave

The avalon slave knows 4 states:

running: This start is reached when starting is terminated without stopping so that start and stop transitions are possible. During this state, the master makes most of its writes to memory.

- not writing: all fifos are cleared until next start
- when writing: fifos are not cleared
- irq are generated when transitioning from writing to not writing

stopping: This state is triggered by a stop when the Comm register is set to 1 so that soft stops are enabled. While stopping via a soft stop, memory writes can continue until end of frame is reached. No signal to indicate this is generated.

- no irq can be generated
- at end of frame, the fifos are empty, and get cleared, then the state is stopped

stopped: This state is active after stopping finished, or when a hard stop is triggered

Only in this state snapshot is relevant.

without snapshot set:

- all fifos are constantly cleared when stopped without snapshot activated
- on start, after starting the running state will be reached

with snapshot: when snapshot is 1 while stopped

- no fifo is permanently cleared
- if snapshot is set before stopped, the fifos are still cleared once in the beginning
- possibility to generate irqs if a one-frame buffer is full
- start while snapshot takes a snapshot, but does not change the state to running.

(re)starting: This state is triggered by a write of 1 to start, whatever value it had before, except when stopping

On a start, for 1 cycle all fifos are cleared. The next cycle, Input_Filter and RGB_transformer are restarted via the internal_start and write signals with 1 cycle delay, while the fifos are released. The next state is running if snapshot is not set or if the previous state was running, and stopped with snapshot otherwise.

Avalon Master

The avalon master knows 4 states

initialization: This state is triggered by a reset or an empty fifo in the middle of a burst-Transfer (which triggers a reset state as it happens only on hard stops). On write it transitions to write_received.

- writing is set to false
- all signals are set to 0 (or 1 if active-low)

await_write: This state is triggered when a buffer gets full. On write it transitions to write_received on a reset to initialization. This state is functionally equivalent to initialization, since the component is inactive and can be reactivated only via a write signal which resets most signals.

- writing is set to false
- some counters are reinitialized
- any ongoing burst transfer is stopped (even if there should be none)

write_received: This state is triggered when the write signal from the slave is set. It takes only 1 cycle, then transitions to initialization if reset, wait_for_fifo otherwise.

- buffer information is taken into account
- all counters are reinitialized
- writing is asserted

wait_for_fifo: Once writing is asserted, the new state is wait_for_fifo as long as the burst signal stays at 0.

When the output fifo contains 12 words, it sets the burst signal to 1 and transitions to the burstTransfer state. On reset and write signals it reacts by moving to initialization or write_received states respectively.

burstTransfer: This state is reached when writing and burst are enabled.

- each cycle when not waiting: send value from fifo, request new value from fifo, decrement burst_size_counter
- while waitRequest is set: not sending nor requesting new data from fifo nor count
- if fifo empty, reset
- if count is 0, go to the await_write state

3.2 Firmware

We provide a small library, similar to the one for the components we were given. It mainly contains an initialization function which takes care of the correct setup of the i2c registers of the camera, and then starts the Camera_Controller, exactly as described in the report of lab 3.0.

The header file also exports a structure *Context* which the user has to instantiate and give as an argument to the initialization function. The three fields of this structure have to contain, from the last to the first, a buffer filled with valid memory addresses, at least frame_size apart

one from each other, the number of addresses in that buffer minus one, a number between 0 and the preceding value.

A main.c example file is provided.

3.3 Results

We were sadly unable to test on the real hardware, mainly because we read too late through some essential documents. We therefore also could also not test our code. However, we tested our design via simulation (testbench files are provided), and are confident that it will work correctly on hardware also, given a suitable initialization.

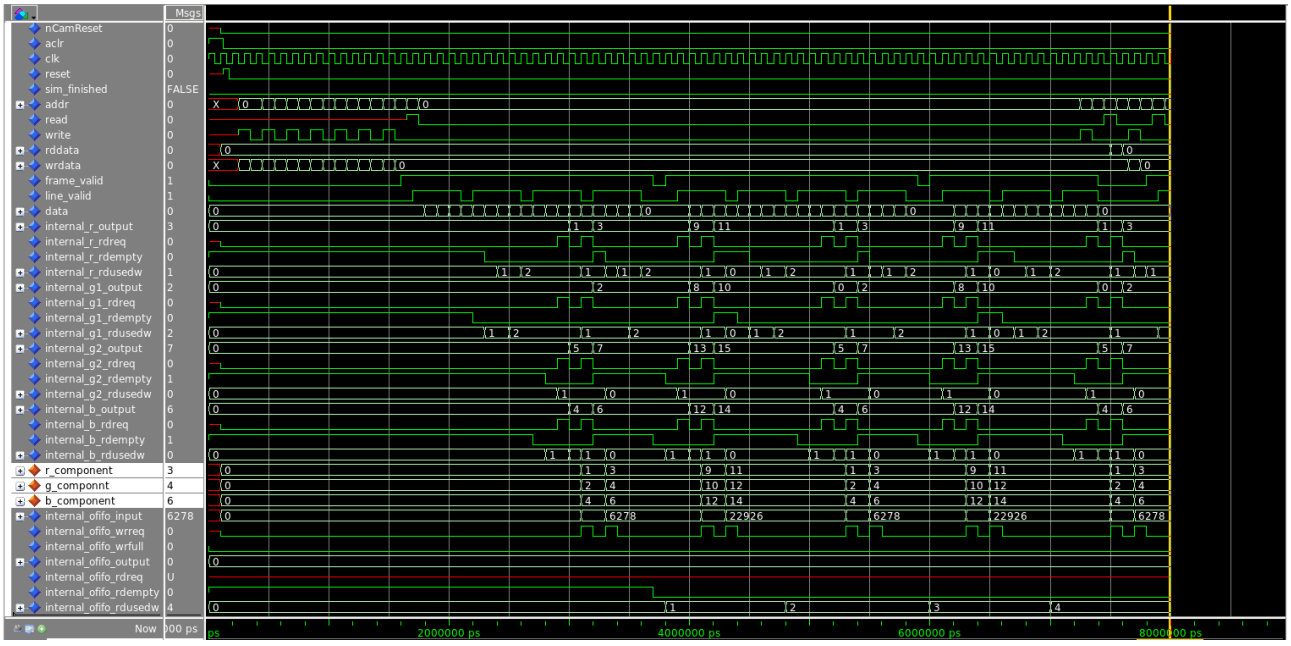

Figure 2: Simulation Results of Camera, Input Filter, Input FIFOs, RGB Generator, and Output FIFO. From these results it can be seen that this section is functioning properly.

4 LT24 Driver

4.1 Low-level Description

4.1.1 State Machine

To be able to send different commands and data to the LCD we are using the following state machine.

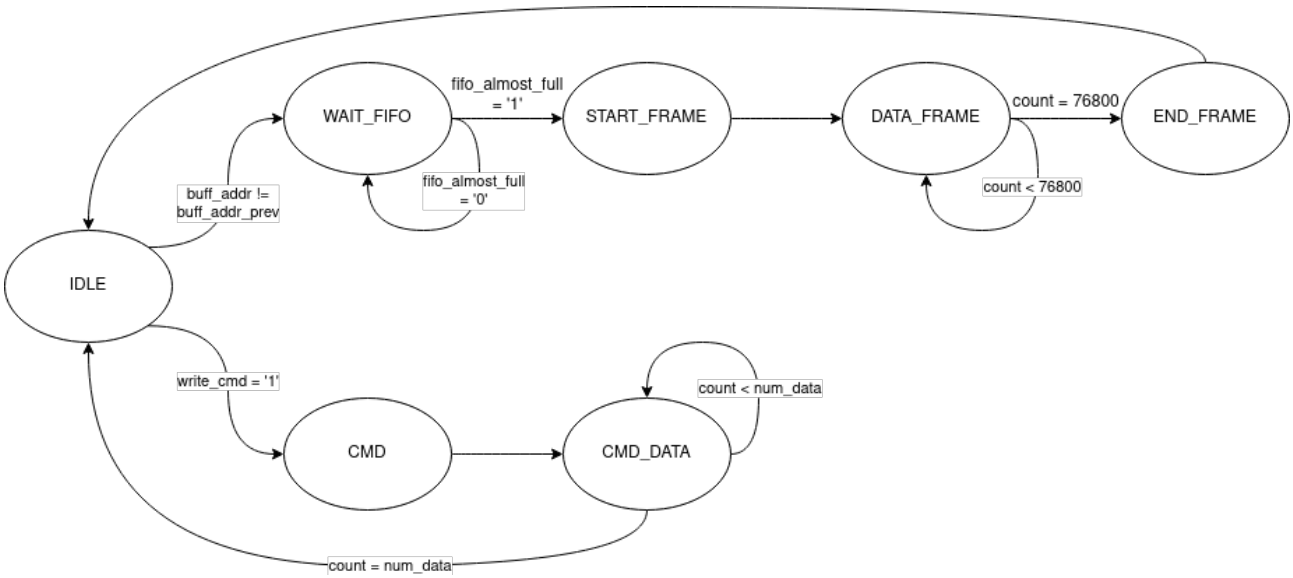

Figure 3: State Machine LCD controller

IDLE The controller is waiting either for a command to be send from the processor or for the buffer address to change, meaning a new frame is available.

WAIT_FIFO The controller waits for the FIFO to be almost full. This allows a continuous data flow from the FIFO to the LCD.

START_FRAME The controller sends a start of frame command. This state takes 4 cycles to execute.

DATA_FRAME The controller sends a data from the FIFO while there is some in it. This state takes 4 cycles to execute.

END_FRAME The controller sends a nop command to the LCD to finish the sending of the current frame. This state takes 4 cycles to execute.

CMD The controller sends a command located in the CMD_command register to the LCD. This state takes 4 cycles to execute.

CMD_DATA In this state the controller sends the data associated to the command to the LCD. This state is repeated CMD_numData times and takes 4 cycles to execute.

4.1.2 Internal connections of the FPGA with the LCD

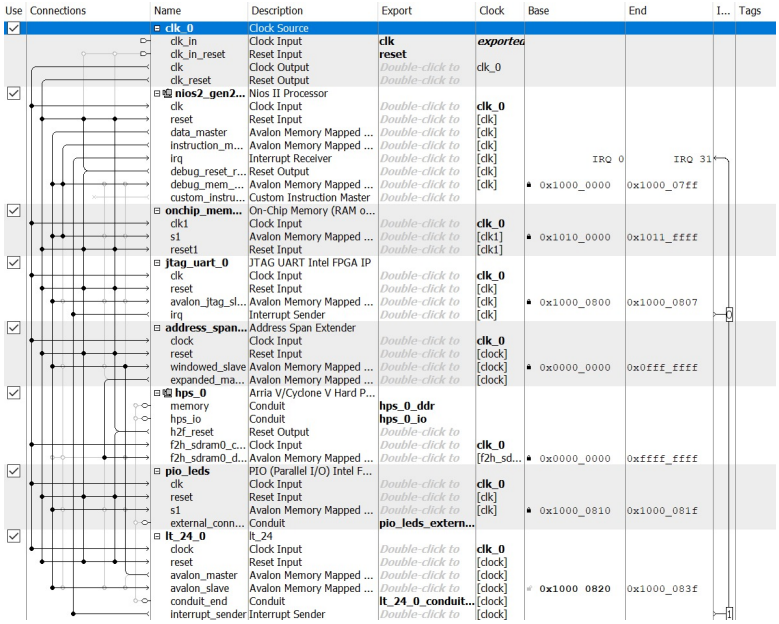

Figure 4: Internal connection of the FPGA with only the LCD

4.1.3 Modelsim simulation

To validate our design, testbench were created for each of the sub-modules and tested on ModelSim.

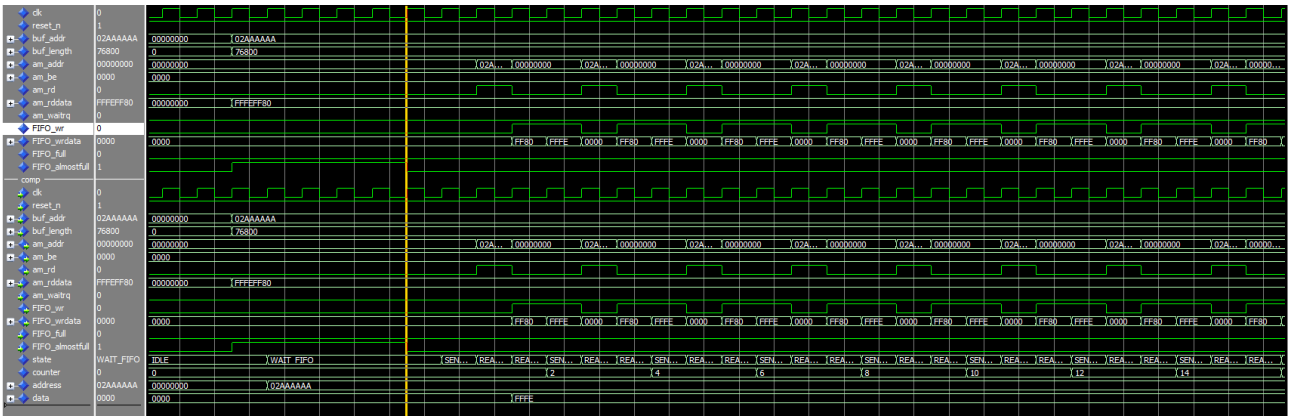

Figure 5: Simulation sub-module Master in Modelsim

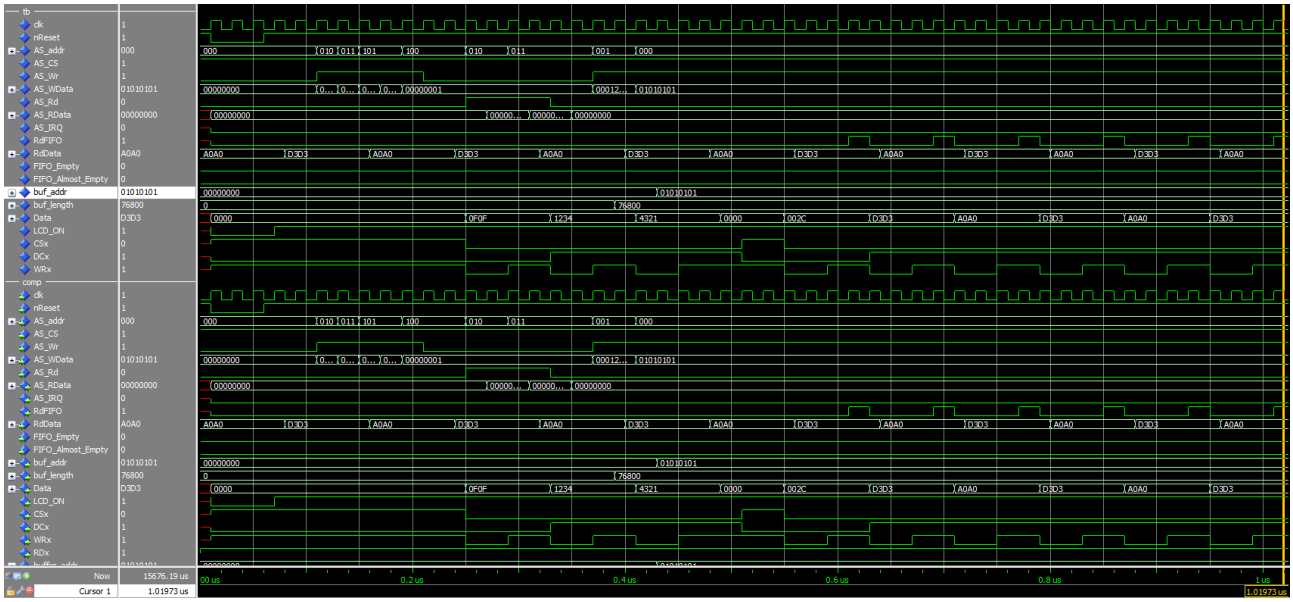

Figure 6: Simulation sub-module Slave/Controller in Modelsim

4.2 Firmware

All the functions to communicate with the component LT24 driver and the commands to be sent are to be found in the file *lt_24_utils.c*.

A structure *command* is used to manipulate the command and associated data. This structure contains the command, the number of data, and a pointer pointing to an array of data. All the commands and their associated data are stored in the command structures (example bellow), which are pointed by the entries of a table named *command_set* for ease of use.

```
// table with the data associated to the command 0x00CF
int pwr_ctrl_b_table[3]    = {0x0000,0x0081,0x00C0};
// command structure with the command, data nbr, and table of data
struct command pwr_ctrl_b = {0x00CF, 3, pwr_ctrl_b_table};
```

This code organisation let us iterate through all the command and their respective data. The LCD's initialization is cared for in function *lcd_setup*.

```
void lcd_setup(void){
  for(int i = 0; i<CMD_SET_SIZE; i++){
    // iterating through each command
    struct command *cmd = command_set[i];
    // sending the command to the slave LCD controller
    IOWR_32DIRECT(LT_24_0_BASE, CMD_CMD, cmd->cmd);
    // sending the number of data to the slave LCD controller
    IOWR_32DIRECT(LT_24_0_BASE, CMD_NUM, cmd->length);

    // sending all the command's associated data to the slave LCD controller
    if(cmd->length != 0) {
```

```
12        for(int j = 0; j < (cmd->length); j++){
13          IOWR_32DIRECT(LT_24_0_BASE, CMD_DATA, cmd->table[j]);
14        }
15      }
16
17      // ordering the LCD controller to send the current command and data
18      IOWR_32DIRECT(LT_24_0_BASE, CMD_WRITE, 1);
19      int running = 1;
20      do{
21        running = IORD_32DIRECT(LT_24_0_BASE, CMD_WRITE);
22      // while CMD_WRITE is equal to 1, the LCD controller is writing to the LCD
23      }while(running);
24    }
25    return;
26  }
```

An interrupt signal to inform that the buffer is empty, and needs to be changed, is sent by the LCD Controller to the Nios 2 processor with the following line:

```
1 // setup interrupt routine
2 alt_ic_isr_register(LT_24_0_IRQ_INTERRUPT_CONTROLLER_ID, LT_24_0_IRQ, lcd_isr,
    NULL, NULL);
```

This line redirects the PC of the processor to an interrupt handler function, which sends a new buffer address.

4.3 Results

Despite extensive debugging and effort, our component was unable to be integrated into a larger system for now. We will continue to work on it to attempt to have a working demo for the presentation. Even though the code was unable to display a video feed, we managed to display one image on the LCD, and our components worked in the simulation (figure 5 and 6). As a result, the project was unable to meet its performance goals and did not produce the desired results.

This outcome was a bit disappointing and frustrating, as we had put in significant time and effort into the design, implementation and testing process.

5 Conclusion

Throughout the project, we faced a number of challenges, including the complexity of the VHDL code and the need to carefully test and debug our designs. Overall, the project has been a valuable learning experience and has allowed for the development of new skills in the design and implementation of digital systems using VHDL.