École Polytechnique fédérale de Lausanne

CS-473 : Embedded Systems
MA1 - Fall 2022

Lab 4.0 - Camera, LCD & VGA - implementation

Professeur : René Beuchat

DARGENT Maxime - 312620
SILVEIRA Joaquim - 311112

Table des matières

1 Introduction

This lab aims to implement the structure of an LCD controller custom component, which gets image data in a DMA fashion from the On-Chip memory and displays it on the LT24 LCD, following the conceptual design explained in Lab 3.0. This project aims to pair our design to a custom camera component controller, which saves the image in the memory we will be fetching. Unfortunately, even though we tried to find a group and even posted a message on moodle, we couldn't find a camera group.

Initially, we laid out the top-level system implementation, describing each component's FSM and functionality and explaining how we updated our design from lab 3.0. Then, we describe the implementation of the programmable interface in VHDL, both regarding the register map exposed to interact with it and the internal structure. We then describe briefly the results obtained with the modelsim testbench. Finally, we provide the results of the deployment in the FPGA with the LT24 LCD and its display.

2 High-Level Description

2.1 Updated DMA FSM

There was a small error in the FSM : in the state ReceiveData (now renamed AcqData), when the length of CntLenght = 0, the state switches to IDLE. Indeed, the goal is to track when a line of pixels is finished, so we have to monitor the size of CntLenght, which is a variable where AcqLenght isn't as it is a fixed input. We also decided to add flags in the DMA module, as it is convenient to synchronize both the DMA and the LCD controller : a flag equivalent to an acquisition enable was added as a condition for the FSM to get out of the IDLE state. We also added two reset flags to handle the register file in the DMA, but that has no link with the acquisition FSM.

During the process of the implementation of the DMA, we realized many problems with our initial design.

First, we inspired our design from the acquisition module of the camera, which is inconsistent with what we tried to do. Indeed, we prepared our design to read and write to the memory, whereas we only needed to read from memory.

This caused many problems, especially during the deployment ; we had a latency when writing in the Avalon bus to send commands to the LCD. We believe the cause was the following : in a bus-based system, the DMA controller is typically treated as a master on the bus and can initiate transactions to read and write data from and to slaves on the bus. If the DMA controller is configured to access the bus simultaneously with another master, this can cause bus contention. As we were trying to write commands to the Avalon Bus, the DMA had the AM_Write bit set to '1', causing a contention. Another cause for this contention might also be that we kept almost constantly our Read bit to '1', meaning that the Avalon Bus thought that we were trying to read from the Avalon bus most of the time. To correct this mistake, we decided to rearrange our FSM such that the reading time of the DMA is as tiny as possible. To do that, we inverted the orders of our checks : we first check that the FIFO has enough space to fit an entire line of pixels, then we check if the Avalon Bus is read with the signal AM_request, and finally activate the read bit and retrieve the data of a pixel, and so on. The final FSM is shown in Fig.1.

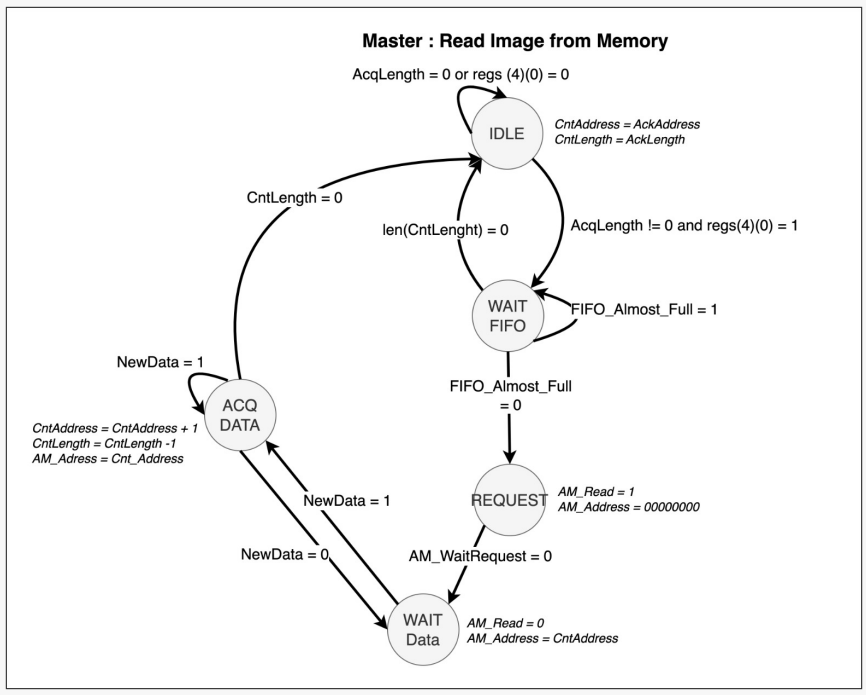

Figure 1 – Updated FSM for the DMA component.

2.2 LCD controller FSM

The general FSM for the LCD controller did not change. We recall it's structure in Fig.2

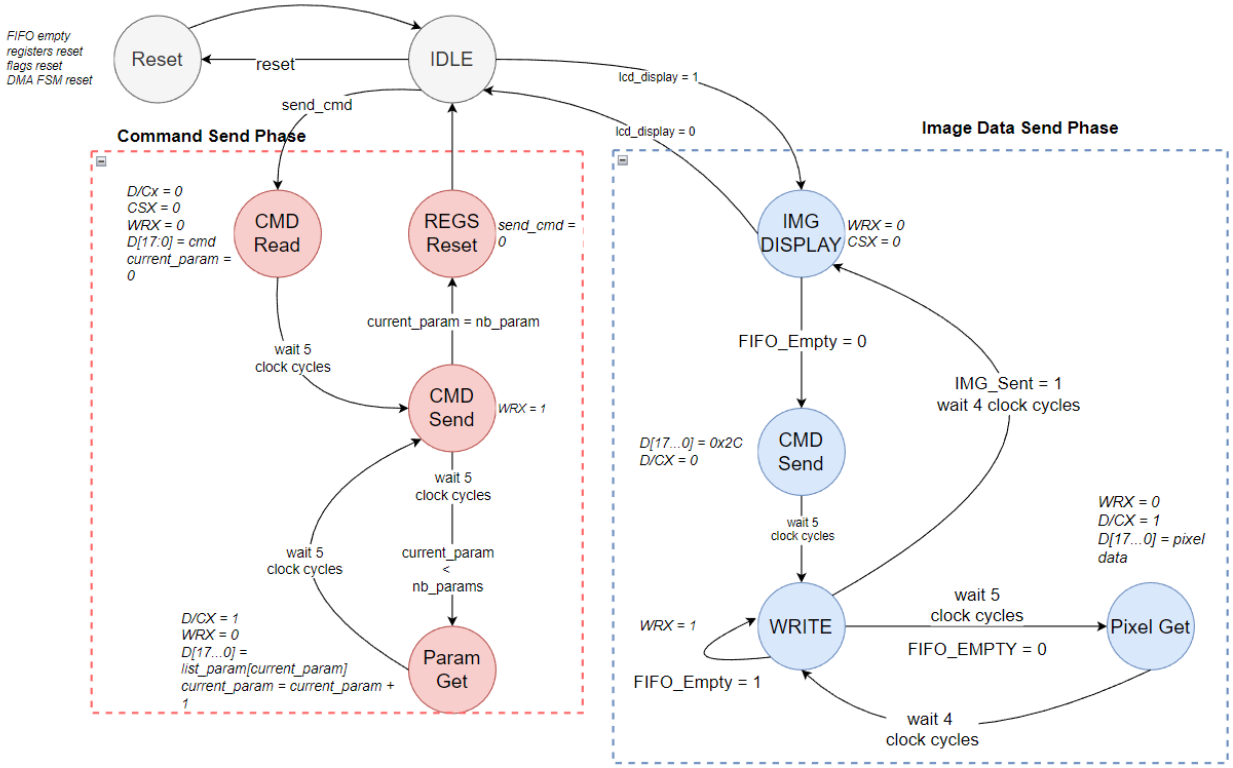

Figure 2 – LCD Controller FSM

3 Low-Level Description

3.1 Programming Interface and Register Map

We have made an addition to the register map by including a Flag Register. This register allows us to control the LCD enable and reset bits, which enables us to initiate the display of an image on the LCD and reset the LCD, respectively. The updated register map is shown in Fig.3

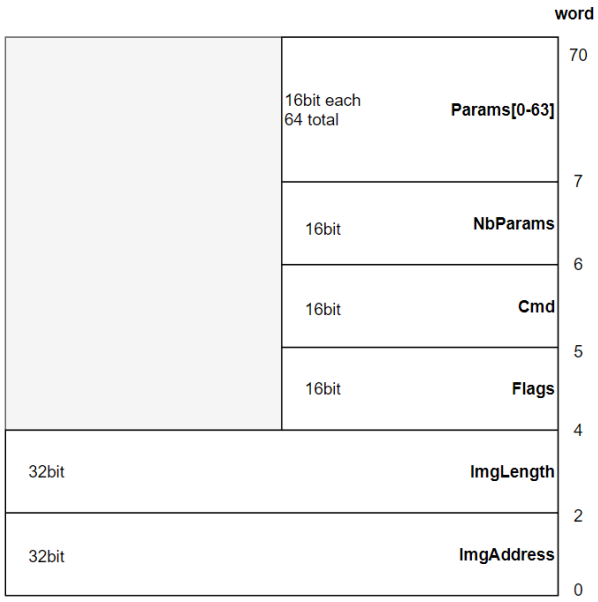

Figure 3 – Updated Register Map.

4 VHDL Implementation

4.1 Avalon Slave Interface - Register Interface

The DMA controller is responsible for handling the registers interface. In that case, we use an intermediate variable temp_regs containing the value of all the 70 registers of 16 bits each. We then set the value of the different commands used to write in multiple registers with the AS_DataWrite signal, depending on the Address sent to the slave side of the component AS_Address. We then set in parallel the values of each register to their dedicated signals :

- Image Address : registers 0 and 1
- Image Length : registers 2 and 3
- Flags Registers : register 4
- Commands Register : register 5
- Number of Parameters : register 6
- Parameters Register : register 7 to 70

The VHDL code is shown below :

```
    process (clk, nReset)
    variable temp_regs : RF(0 to 70) := (others => (others => '0'));
    variable written : std_logic := '0';
    variable prev_addr : integer;
Begin

    if     nReset ='0' THEN
        temp_regs(0) := (others => '0');
        temp_regs(1) := (others => '0');
        temp_regs(2) := (others => '0');
```

```
        temp_regs(3) := (others => '0');
        temp_regs(4) := (others => '0');
        temp_regs(5) := (others => '0');
        temp_regs(6) := (others => '0');
        temp_regs(7 to 70) := (others => (others => '0'));
        regs <= temp_regs;
    elsif rising_edge(clk) then
        -- decode address
        temp_regs := regs;

        addr <= to_integer(unsigned(AS_Address));
        -- write operation
        if (AS_Write = '1') then
            if written = '0' then
                temp_regs(addr) := AS_DataWrite;
                prev_addr := addr;
                written := '1';
            elsif prev_addr /= addr then
                written := '0';
            end if;
        end if;

        if ResetFlagCMD = '0' then
            temp_regs(4)(1) := '0';
        end if;
        if ResetFlagReset = '0' then
            temp_regs(4)(2) := '0';
        end if;
        if reset_lcd_enable = '0' then
            temp_regs(4)(0) := '0';
        end if;

        regs <= temp_regs;

    end if;

end process;

AcqAddress <= regs(0) & regs(1);
AcqLength <= regs(2) & regs(3);
Reg_Flags <= regs(4);
Reg_Cmd <= regs(5);
Reg_NbParam <= regs(6);
Reg_Param <= regs(7 to 70);
debug <= regs(4);
```

We can see in the code above a particular set of conditions influencing the Flag register :

```
if ResetFlagCMD = '0' then
    temp_regs(4)(1) := '0';
end if;
if ResetFlagReset = '0' then
    temp_regs(4)(2) := '0';
end if;
if reset_lcd_enable = '0' then
    temp_regs(4)(0) := '0';
end if;
```

This is because these flags are modified by either the DMA controller or the LCD controller in some cases :

the reset_lcd_enable is set to 0 by the DMA when it has finished reading an image and switches back to the IDLE state. This is useful to ensure that the Acquisition process stops when it has nothing more to read and prevents it from trying to acquire corrupted or incorrect data. The ResetFlagReset is controlled by the RESET routine in the LCD controller, and it signals the completion of the routine by clearing the bit set by the Nios II. Similarly, the ResetFlagCMD is activated by the LCD controller once all necessary commands and their corresponding parameters have been sent, indicating to the higher level that they can be cleared.

Another particular aspect is the following :

```
if (AS_Write = '1') then
    if written = '0' then
        temp_regs(addr) := AS_DataWrite;
        prev_addr := addr;
        written := '1';
    elsif prev_addr /= addr then
        written := '0';
    end if;
end if;
```

We used two intermediate variables to fix a timing problem which caused AS_Datawrite to be copied in the wrong register because the process didn't have time to get the new value on the Avalon bus whereas it already had the new address. The tradeoff here is that it is impossible to write twice in a row in the same registers, but this situation never happens in a general case.

4.2 Avalon Master Interface - DMA Controller

As said before, we took inspiration from the code given in class for the VHDL implementation of this module. We had to change a few elements :

A particular input of the DMA controller to write in the memory is the data acquisition bus and NewData. These inputs transmit data from the camera to the Acquisition module. A particular module is used in this case, with the following process : The module receives the data on the Datacquisition bus [7...0]. When new data is received on the bus, the NewData bit is set to '1' until it is acknowledged by the Acquisition module, which sets a signal called DataAck to '1'. When it is set, NewData is cleared at the next rising edge of the clock, and then DataAck can be set to '0'.

We decided to keep similar names to understand the acquisition process better : we connected AM_readdatavalid to NewData and AM_readdata to DataAcquisition. The process is then the following : as soon as the FIFO is not almost complete and the wait request is 0, we retrieve data from memory at the address given by AM_Address. We then subtract 2 from the length of the data we need to acquire and add 2 to the following address where we need to fetch the data. This is because we send the FIFO packets of 16 bits of data (equivalent to 1 pixel).

This part o the code is shown below :

```
when Request =>
    AM_Read <= '1';
    AM_Address <= CntAddress;
    if AM_WaitRequest = '0' then
        SM <= WaitData;
    end if;
when WaitData =>
    AM_Read <= '0';
    AM_Address <= x"00000000";
    if NewData = '1' then
        SM <= AcqData;
    end if;
when AcqData =>
    if to_integer(unsigned(CntLength)) = 0 then
        reset_lcd_enable <= '0';
        SM <= Idle;
        AM_Address <= (others => '0');
```

```
    elsif NewData = '1' then
        CntAddress <= std_logic_vector(to_unsigned(to_integer(unsigned(CntAddress)) +
        2,CntAddress'length));
        CntLength <=  std_logic_vector(to_unsigned(to_integer(unsigned(CntLength)) -
        2,CntLength'length));
    else
        SM <= WaitData;
    end if;
```

We can also see in the code above that if the lenght of the data we want to display on the LCD is = to 0, we go back to IDLE, as said in the section on the updated FSM.

4.3 LCD Controller

The LCD controller subsystem is responsible for managing the communication between the FPGA (employing a Nios II CPU) and the physical LCD screen, the LT24. This is achieved through the use of GPIOs to send and receive signals such as RESX, D, DCX, WRX, and CSX. The subsystem operates in three main phases : reset, command transmission, and image data transmission. These phases are described in more detail below.

4.3.1 Image Data Transfer

When the bit 0 in the Flag Register is set to 1, the DMA begins transferring data from the previously specified memory address and fills up the FIFO buffer, which is connected to the LCD controller. A FIFO is used to smooth out any differences in the data transfer rate between the DMA and the LCD controller. The DMA may be able to transfer data at a flengthrof than the LCD controller can process it, in which case the FIFO acts as a buffer to store the excess data until the LCD controller is ready to process it, preventing data loss or overwrite.

When the LCD controller FSM is in the IDLE state and the bit 0 in the Flag Register is set, it transitions to the IMG_DISPLAY state, which waits for valid data from the FIFO buffer. We indicate to the LCD that the data being sent from the controller is pixel data by sending the $0x2C$ command, setting the *DCX* signal to low (indicating it is a command), and setting the *WRX* signal to low to send it.

After a delay, the controller transitions to the PIXEL_WRITE state, in which the *WRX* signal is set to high and the state of the transmitted image, the remaining length, and the FIFO empty signal are checked. The controller then alternates between the PIXEL_GET and PIXEL_WRITE states. In the PIXEL_GET state, the *DCX* signal is set to high and the *WRX* signal is set to low, indicating that pixel data will be sent to the LCD. Care must be taken to properly count the cycles before requesting more data from the FIFO, as the communication is asynchronous and data may be lost if a pixel package is skipped.

```
when PIXEL_GET =>
    DCX <= '1';
    WRX <= '0';

    if clock_cycles < 2 then
        LCD_ReadReq <= '0';
        clock_cycles <= clock_cycles + 1;
        current_state <= PIXEL_GET;
    elsif clock_cycles = 2 then
        LCD_ReadReq <= '1';
        clock_cycles <= clock_cycles + 1;
        current_state <= PIXEL_GET;
    elsif clock_cycles = 3 then
        clock_cycles <=  clock_cycles + 1;
        current_state <= PIXEL_GET;
        LCD_ReadReq <= '0';
    else
        D <= LCD_q;
        clock_cycles <= "000";
```

```
        current_state <= PIXEL_WRITE;
        LCD_ReadReq <= '0';
    end if;
```

The PIXEL_WRITE state rises the *WRX* signal, which sends the value that was placed on the data bus. If there are any pixels left to send and the FIFO is empty, we wait here until it fills up again.

```
when PIXEL_WRITE =>
    WRX <= '1';
    if bytes_remaining = 0 then
        LCD_ReadReq <= '0';
        if (clock_cycles) < 4 then
            current_state <= PIXEL_WRITE;
            clock_cycles <= clock_cycles + 1;
        else
            current_state <= IMG_DISPLAY;
            clock_cycles <= "000";
        end if;
        elsif LCD_Empty = '0' then
            if (clock_cycles) < 4 then
                clock_cycles <= clock_cycles + 1;
                current_state <= PIXEL_WRITE;
            else
                current_state <= PIXEL_GET;
                clock_cycles <= "000";
                bytes_remaining <= bytes_remaining - 2;
            end if;
        else
            current_state <= PIXEL_WRITE;
            clock_cycles <= "000";
    end if;
```

4.3.2 Command Transmission

4.3.3 Reset

The reset protocol implemented follows the description provided in the ILI9341 datasheet and in Lab 3.0. It is initiated when the Nios II sets the reset bit (bit 2) in the Flag Register. An internal counter is used to keep track of the timing of the process, as specified in the datasheet. The *RESX* signal is set to 1 for 1ms (equivalent to 50,000 clock cycles using the *FPGA_1_50* clock source), then set to 0 for $10ms$, and finally set to 1 again for $120ms$. After the reset process is complete, the reset flag is cleared by the LCD controller. The code for the reset implementation is provided below :

```
when RESET  =>
    WRX <= '0';
    if reset_cnt < 50000  then
        RESX <= '1';
        reset_cnt <= reset_cnt + 1;
        reset_flag_reset <= '1';
        current_state <= RESET;

    elsif reset_cnt <550000 then
        RESX <= '0';
        reset_cnt <= reset_cnt + 1;
        current_state <= RESET;
        reset_flag_reset <= '1';

    elsif reset_cnt < 6550000 then
        RESX <= '1';
```

```
        reset_cnt <= reset_cnt + 1;
        current_state <= RESET;
        reset_flag_reset <= '1';


    elsif reset_cnt = 6550000 then
        RESX <= '1';
        reset_flag_reset <= '0';
        reset_cnt <= reset_cnt + 1;
        current_state <= RESET;

    else
        RESX <= '1';
        reset_flag_reset <= '0';
        current_state <= IDLE;
        reset_cnt <= x"00000000";
    end if;
```

It's behaviour has been verified in ModelSim as shown in Fig.4.

5 ModelSim Simulation

To test our design, we decided to implement a top-level test bench. A general view of the simulation is shown in Fig.4.

We can see that overall the testbench shows a successful implementation of the design, as seen on the D bus connected to the controller; it sends the data as expected from the testbench (code in annex), same goes for the other signals connected to the LCD. We can also see that we implemented many debug signals to make sure that every process works as it should. These signals will be explained in more detail below.

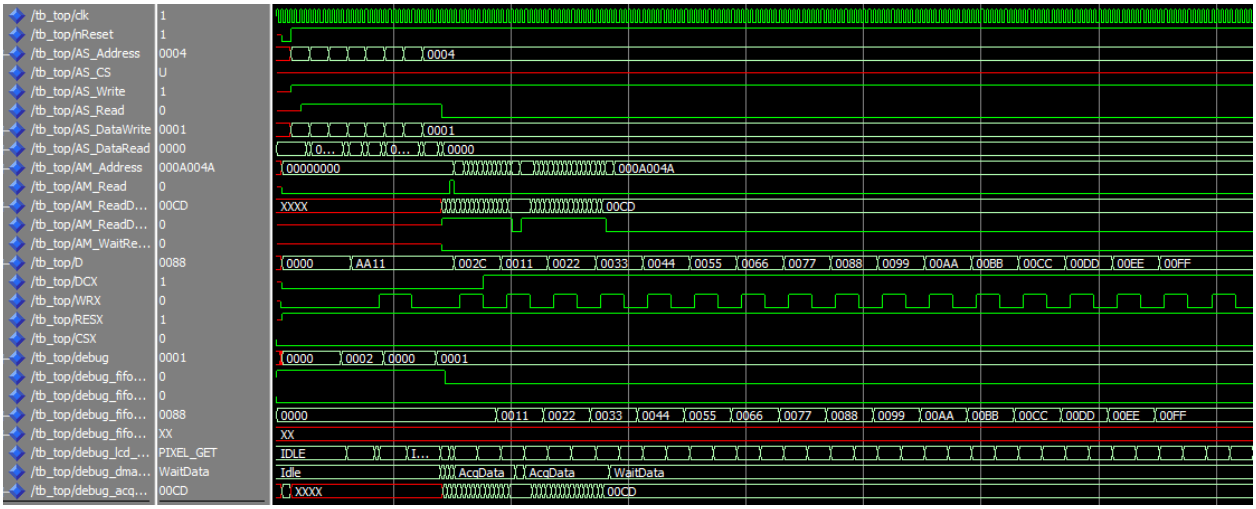

Figure 4 – Top level Testbench on ModelSim.

We tested three elements in our testbench : a reset protocol, sending commands, and a simple display on the LCD.

5.1 Reset

Here we tested the Reset sequence by setting the reset flag in the Flag Register to 1. As we can see by the *RESX* signal, it works as expected. The simulation is shown in Fig.5.

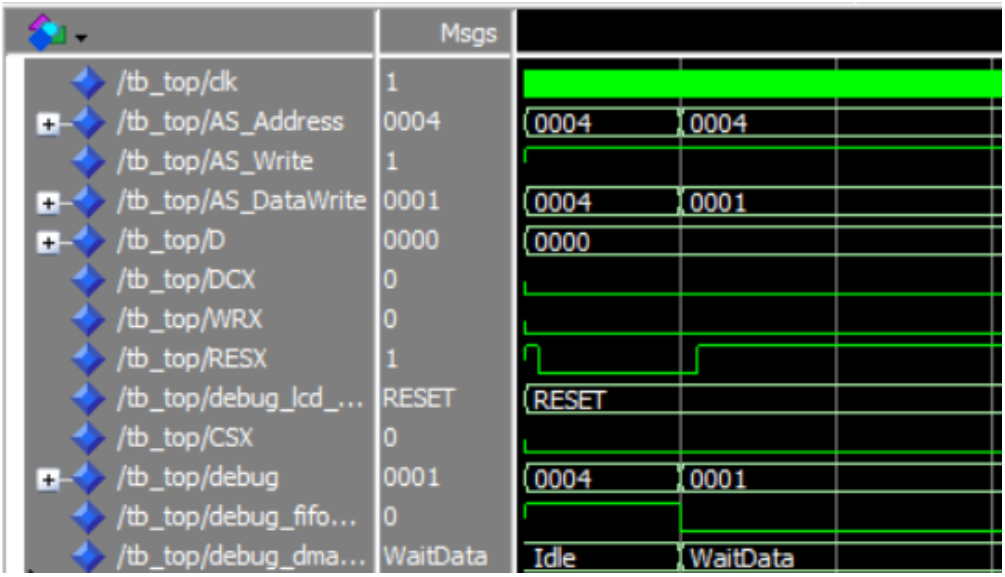

Figure 5 – Testbench on ModelSim : Reset

5.2 Commands

Here we show the command sending simulation. At the begining, we set the command register to the command to be sent value, $0xAA11$. This value is then ported to the databus D, and the CSX signal is lowered, to indicate that it is a command, and not data. We then raise the WRX signal to output it to the LCD. The commands are sent by setting the second bit on the Flag Register. The simulation is shown in Fig.6.

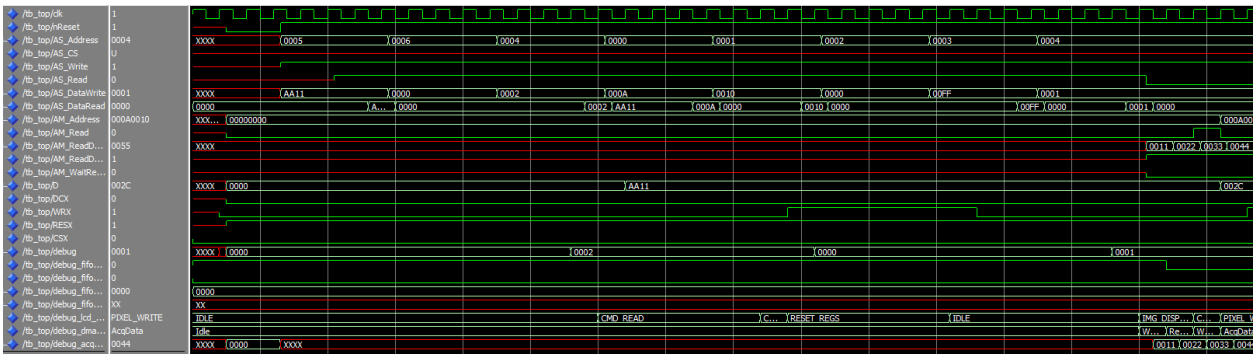

Figure 6 – Testbench on ModelSim : Sending commands

5.3 Display

For the display simulation, we sent two bursts of 15 pixels ranging from 0x11 to 0xFF. The goal here was to make sure that we didn't have packet loss, as well as to check that the FSM works according to the initial design. When the flag(0) is set to 1, the acquisition process changes state to check the FIFO and switches to the request state. The debug signal is located on the lowest part of Fig.7. This signal shows the output of the DMA, and confirms that the Acquisition process works as expected. It also transitions back to the WaitData state when no data is left on the AM_DataRead.

6 Deployment

To deploy our design on the LCD, we have to configure the pins of the De0-nano-SoC to connect to the LCD pins. Fig.8 shows an image of the said pins.

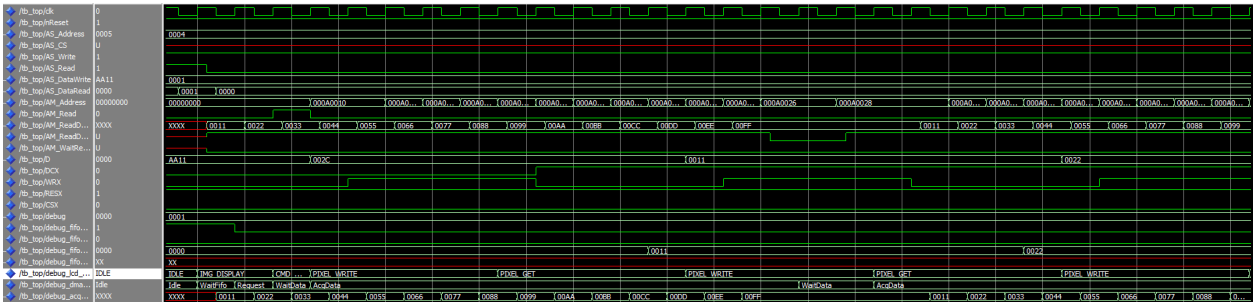

Figure 7 – Testbench on ModelSim : Display

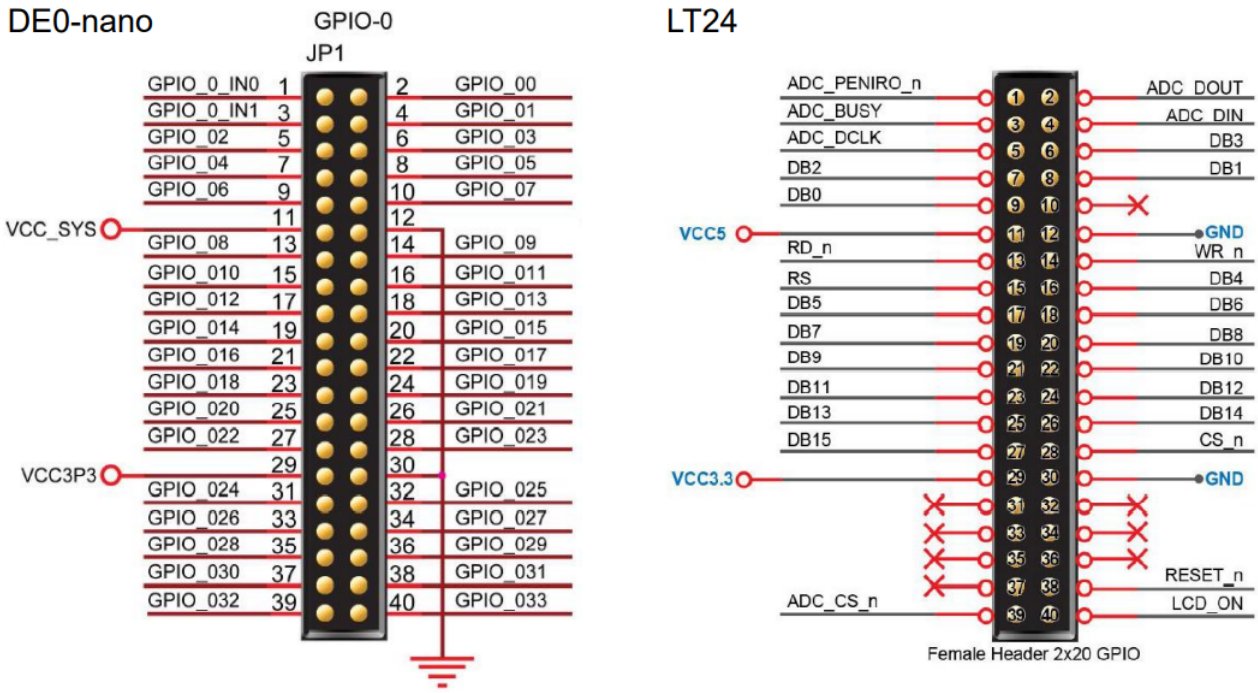

Figure 8 – DE0-nano and LT24 pinout

To successfully implement our design for the LT24 LCD, we must establish connections between several of its pins and our board. These pins include RESX, WRX, RDX, D/CX, and D[15...0]. It is important to note that, even if certain pins are not being used in our design, it is still necessary to properly configure them according to the specifications outlined in the datasheets in order for the LCD to operate correctly. These pin assignments can be found in the *DE0-nano-SoC_top-level file*, and it is crucial to properly set them up in order to ensure the proper functioning of the LCD.

```
GPIO_0_LT24_ADC_CS_N  <= '1';       --active low, therefore 1 should activate LCD
GPIO_0_LT24_LCD_ON    <= '1';
GPIO_0_LT24_RD_N      <= '1';

u0 : component soc_system

port map(
    clk_clk                           => FPGA_CLK1_50,
    --reset_reset_n                     => '1',
    reset_reset_n => KEY_N(0),
    ...
    lcd_0_output_csx    =>      GPIO_0_LT24_CS_N,         -- chip select active low
    lcd_0_output_d      =>      GPIO_0_LT24_D,            -- data bus
```

```
lcd_0_output_resx   =>      GPIO_0_LT24_RESET_N,      -- lcd reset active low
lcd_0_output_wrx    =>      GPIO_0_LT24_WR_N,         -- write signal active low
lcd_0_output_dcx     =>      GPIO_0_LT24_RS           -- data/command select
```

6.1 Qsys Topology

To implement our component, we follow the instructions provided in the Lab's notice. First, we connect the Avalon Master interface to the memory where the image will be stored. The slave interface is then connected to the Nios II CPU. The signals that control the LCD hardware are exported as conduits and connected to the appropriate GPIOs, as previously described.

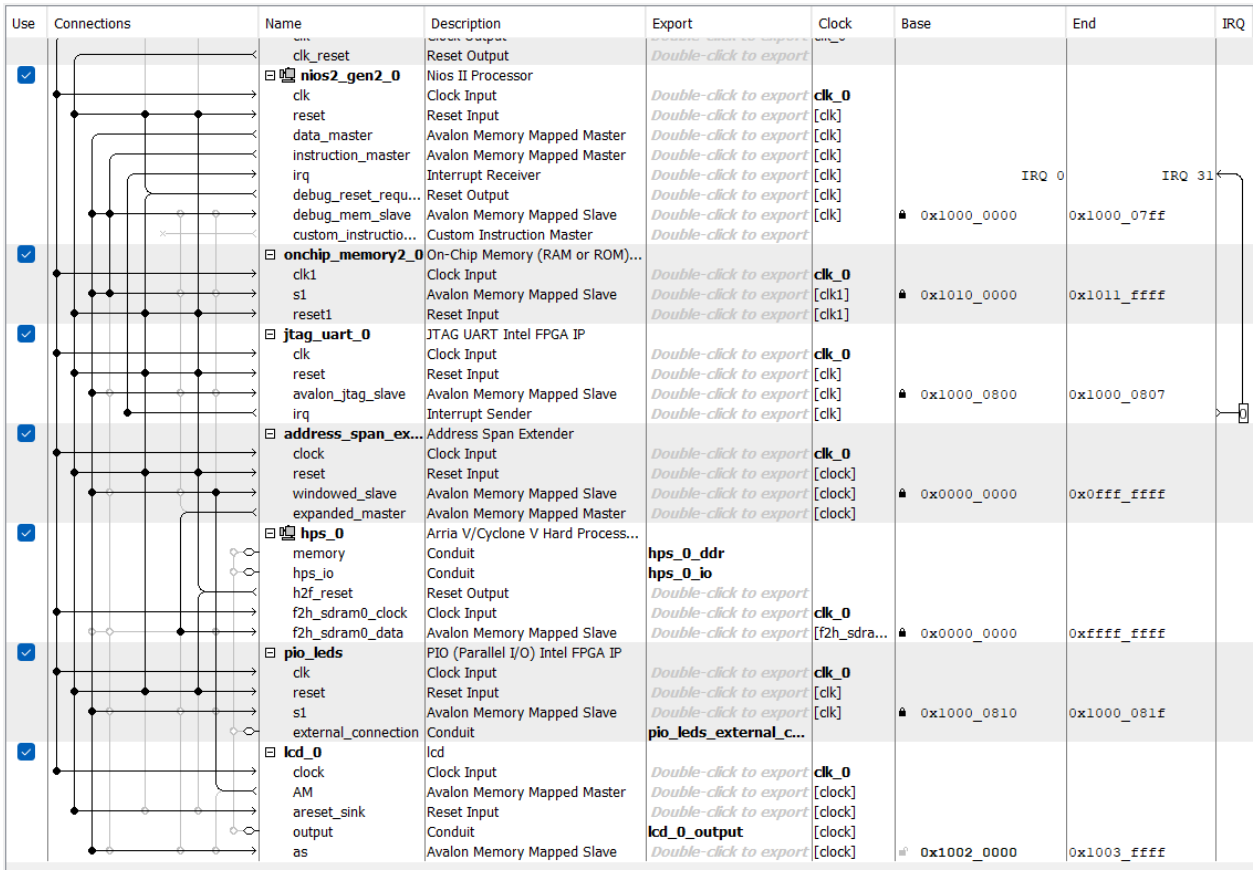

Figure 9 – Qsys topology implementing our LCD controller.

6.2 C code

To communicate with our component from the Nios II CPU, we developed a simple library that includes all the necessary functions for manipulating bits and registers. The first step in using this library is to reset the LCD using the reset routine described earlier. Next, the LCD is initialized with the desired configuration, as learned in class. We discovered that some commands were occasionally skipped or lost during the initialization process, even though we had implemented a polling method. As a result, we added delays between commands to improve the reliability of the communication. Finally, we set the image length and address in the appropriate registers and activate the LCD display by setting the corresponding bit in the Flag Register.

```
int main(){
    uint32_t img_address = HPS_0_BRIDGES_BASE;
    uint32_t img_length = 240 * 320 * 2;

    printf("Hello from Nios II!");
    // reset lcd
```

```
    lcd_reset();
    // init lcd
    init_lcd();
    // set image in memory
    init_image(img_address, 320, 240);
    // configure image
    configure_image(img_address, img_length);
    // start lcd
    start_lcd();
    return 0;
}
```

The whole code can be studied in the annexes.

7 Conclusion

While we have gained valuable experience in the design and integration of components through this lab, we have not yet been able to fully complete our custom component for the LT24 LCD. During the development process, we encountered numerous challenges and difficulties that slowed our progress. These issues were particularly impactful as this was our first time working with Hardware Description Language and hardware development, and we struggled to effectively identify and resolve the problems we faced.

One specific issue that caused us significant delays was the read timing parameter of the Qsys component, which was set to 1, by default. This led to difficulties with the AS_Write function, as it would not fire when we attempted to write to one of the registers due to a timing protocol issue shown in Figure 10. We encountered a number of other bugs and challenges throughout the development process, and it took longer than we had anticipated to effectively resolve these issues. Despite these difficulties, we were able to gain valuable knowledge and experience in the design and integration of components through this lab.

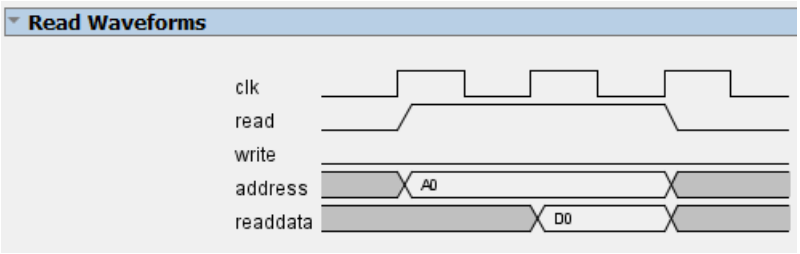

Figure 10 – Timing protocol to be able to write to our slave component.

During the development process, we encountered several issues that slowed our progress and required us to troubleshoot and find solutions. Another problem we encountered was that, after setting the read timing parameter to 0, we were able to write to the registers but the written values seemed to be corrupted when we tried to read them back. We eventually discovered that this issue was caused by the Avalon Master side of our component constantly requesting access to the bus due to a state mismatch in the FSM, which was caused by the corresponding signals not being correctly set at the start of the system. To resolve this issue, we had to modify the template of the project so that the general system reset was linked to one of the buttons on the board rather than being constantly driven to 1.

These are just a few examples of the challenges we faced during the development process. Despite these obstacles, we were able to make progress through careful study and problem-solving. Unfortunately, we were unable to test the whole system with the camera module as we were unable to find another group that had implemented it. However, we are confident that everything would have worked smoothly, as the ModelSim simulations and testbenches were functioning as expected. We simply needed more time to complete the project.

8 Annexes

8.1 C Code

```
    /*
 * "Hello World" example.
 *
 * This example prints 'Hello from Nios II' to the STDOUT stream. It runs on
 * the Nios II 'standard', 'full_featured', 'fast', and 'low_cost' example
 * designs. It runs with or without the MicroC/OS-II RTOS and requires a STDOUT
 * device in your system's hardware.
 * The memory footprint of this hosted application is ~69 kbytes by default
 * using the standard reference design.
 *
 * For a reduced footprint version of this template, and an explanation of how
 * to reduce the memory footprint for a given application, see the
 * "small_hello_world" template.
 *
 */

#include "stdio.h"
#include <inttypes.h>
#include "system.h"
#include "io.h"
#include "stdarg.h"

// Register map offsets
#define LCD_REG_IMG_ADDRESS_HIGH 0x00
#define LCD_REG_IMG_ADDRESS_LOW 0x01
#define LCD_REG_IMG_LENGTH_HIGH 0x02
#define LCD_REG_IMG_LENGTH_LOW  0x03
#define LCD_REG_FLAGS  0x04
#define LCD_REG_CMD_REG 0x05
#define LCD_REG_NB_PARAM 0x06
#define LCD_REG_PARAM(index)+ index


void delay();
void small_delay();
void lcd_flag_set (uint16_t flag);
void lcd_reset();
void cmd_send(uint16_t cmd, uint16_t n, uint16_t* params);
void start_lcd();
void init_lcd();


int main(){
    uint32_t img_address = HPS_0_BRIDGES_BASE;
    uint32_t img_length = 240 * 320 * 2;

    printf("Hello from Nios II!");
    // reset lcd
    lcd_reset();
    // init lcd
    init_lcd();
    // set image in memory
    init_image(img_address, 320, 240);
```

```
    // configure image
    configure_image(img_address, img_length);
    // start lcd
    start_lcd();
    return 0;
}



void delay(){
    uint32_t c;
    for (c = 1; c <= 3000000; c++){}
}

void small_delay(){
    uint32_t c;
    for (c = 1; c <= 1000; c++){}
}


void lcd_flag_set (uint16_t flag){
    uint16_t reg_flag_val =     IORD_16DIRECT(LCD_0_BASE, LCD_REG_FLAGS);
    delay();
    IOWR_16DIRECT(LCD_0_BASE, LCD_REG_FLAGS,    (reg_flag_val | flag));
}

void lcd_reset(){
    lcd_flag_set(0x04);

    delay();

    uint16_t reg_flag_val = IORD_16DIRECT(LCD_0_BASE, LCD_REG_FLAGS);

    printf("%lu  \n", (unsigned long)reg_flag_val);

    delay();

    while( reg_flag_val & 0x04){
        reg_flag_val = IORD_16DIRECT(LCD_0_BASE, LCD_REG_FLAGS);
    }
    printf("reset finished \n");

}

void cmd_send(uint16_t cmd, uint16_t n,    uint16_t* params){
    uint16_t index = 0;

    IOWR_16DIRECT(LCD_0_BASE, LCD_REG_CMD_REG, cmd);
    small_delay();
    IOWR_16DIRECT(LCD_0_BASE, LCD_REG_NB_PARAM, n);


    while(index < n){
        IOWR_16DIRECT(LCD_0_BASE, LCD_REG_PARAM(index), params[index]);
        small_delay();
        ++index;
    }
```

```
    IOWR_16DIRECT(LCD_0_BASE, LCD_REG_FLAGS, 0x2);
    uint16_t reg_flags = IORD_16DIRECT(LCD_0_BASE, LCD_REG_FLAGS);
    while(reg_flags & 0x2){
        reg_flags = IORD_16DIRECT(LCD_0_BASE, LCD_REG_FLAGS);
    }
    printf("command sent \n", (unsigned long)reg_flags,"\n");
}

void start_lcd(){
    lcd_flag_set(0x01);
    uint16_t reg_flags_val = IORD_16DIRECT(LCD_0_BASE, LCD_REG_FLAGS);

    while(reg_flags_val & 0x1){
        reg_flags_val = IORD_16DIRECT(LCD_0_BASE, LCD_REG_FLAGS);
    }
}

void init_lcd(){
    cmd_send(0x11, 0, (uint16_t []){ 0x09, 0x0a});
    small_delay();
    cmd_send(0xcf, 3, (uint16_t []){ 0x0, 0x81, 0xc0});
    small_delay();
    cmd_send(0xed, 4, (uint16_t []){ 0x64, 0x03, 0x12, 0x81});
    small_delay();
    cmd_send(0xe8, 3, (uint16_t []){ 0x85, 0x01, 0x0798});
    small_delay();
    cmd_send(0xcb, 5, (uint16_t []){ 0x39, 0x2c, 0x00, 0x34, 0x02});
    small_delay();
    cmd_send(0xf7, 1, (uint16_t []){ 0x20});
    small_delay();
    cmd_send(0xea, 2, (uint16_t []){ 0x00, 0x00});
    small_delay();
    cmd_send(0xb1, 2, (uint16_t []){ 0x00, 0x1b});
    small_delay();
    cmd_send(0xb6, 2, (uint16_t []){ 0x0a, 0xa2});
    small_delay();
    cmd_send(0xc0, 1, (uint16_t []){ 0x05});
    small_delay();
    cmd_send(0xc1, 1, (uint16_t []){ 0x11});
    small_delay();
    cmd_send(0xc5, 2, (uint16_t []){ 0x45, 0x45});
    small_delay();
    cmd_send(0xc7, 1, (uint16_t []){ 0xa2});
    small_delay();
    cmd_send(0x36, 1, (uint16_t []){ 0x08}); //BGR  (originally rgb with 0x48)
    small_delay();
    cmd_send(0xf2, 1, (uint16_t []){ 0x00});
    small_delay();
    cmd_send(0x26, 1, (uint16_t []){ 0x01});
    small_delay();
    cmd_send(0xe0, 15, (uint16_t []){ 0xf, 0x26,    0x24, 0xb, 0xe, 0x8, 0x4b, 0xa8, 0x3b, 0x0a,   0x1
    small_delay();
    cmd_send(0xe1, 15, (uint16_t []){ 0x0, 0x1c,    0x20, 0x4, 0x10, 0x8, 0x34, 0x47, 0x44, 0x05,  0xb
    small_delay();
    cmd_send(0x2a, 4, (uint16_t []){ 0x0, 0x0,  0x0, 0xef});
    small_delay();
```

```
    cmd_send(0x2b, 4, (uint16_t []){ 0x0, 0x0,  0x01, 0x3f});
    small_delay();
    cmd_send(0x3a, 1, (uint16_t []){ 0x55}); //RGB
    small_delay();
    cmd_send(0xf6, 3, (uint16_t []){ 0x01, 0x30,    0x0});
    small_delay();
    cmd_send(0x29, 0, (uint16_t []){ 0x09, 0x0a});
    small_delay();
    cmd_send(0x2c, 0, (uint16_t []){ 0x09, 0x0a});
    small_delay();
}

void configure_image(uint32_t image_address, uint32_t image_size){
    uint16_t image_address_low = image_address & 0xFFFF;
    uint16_t image_address_high = (image_address >> 16) & 0xFFFF;
    IOWR_16DIRECT(LCD_0_BASE, LCD_REG_IMG_ADDRESS_LOW,image_address_low);
    delay();
    IOWR_16DIRECT(LCD_0_BASE, LCD_REG_IMG_ADDRESS_HIGH,image_address_high);
    delay();
    uint16_t image_size_low = image_size & 0xFFFF;
    uint16_t image_size_high = (image_size >> 16) & 0xFFFF;
    IOWR_16DIRECT(LCD_0_BASE, LCD_REG_IMG_LENGTH_LOW, image_size_low);
    delay();
    IOWR_16DIRECT(LCD_0_BASE, LCD_REG_IMG_LENGTH_HIGH, image_size_high);
    delay();
}


void init_image(uint32_t image_address, uint32_t rows, uint32_t cols){
    uint32_t i = 0;
    uint16_t color = 0xA8F0;

    while(i < (rows * sizeof(uint16_t))){
        uint32_t j = 0;

        if(i % 30 == 0){
            color = color == 0xA8F0 ? 0x00AA : 0xA8F0;
        }

        while(j < cols * sizeof(uint16_t)){
            IOWR_16DIRECT(image_address, i * cols + j, color);
            small_delay();
            j += 2;
        }
        i += 2;
    }
}
```

8.2 Testbench Code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.lcd_package.all;
```

```
entity tb_top is

end tb_top;

architecture test of tb_top is

    CONSTANT CLK_PERIOD : time := 20 ns;

--global signals
    SIGNAL clk : std_logic;
    SIGNAL nReset : std_logic;

-- avalon slave terface
    SIGNAL AS_Address :  std_logic_vector(15 downto 0);
    SIGNAL AS_CS :  std_logic;
    SIGNAL AS_Write :  std_logic;
    SIGNAL AS_Read :  std_logic;
    SIGNAL AS_DataWrite :  std_logic_vector(15 downto 0);
    SIGNAL AS_DataRead :  std_logic_vector(15 downto 0);

-- avalon master terface(DMA)
    SIGNAL AM_Address :  std_logic_vector(31 downto 0);
    SIGNAL AM_Read :  std_logic;
    SIGNAL AM_ReadData :  std_logic_vector(15 downto 0);
    SIGNAL AM_ReadDataValid :  std_logic;
    SIGNAL AM_WaitRequest :  std_logic;

--lcd signals(gpio)
    SIGNAL D :  std_logic_vector(15 downto 0);
    SIGNAL DCX :  std_logic;
    SIGNAL WRX :  std_logic;
    SIGNAL RESX :   std_logic;
    SIGNAL CSX :  std_logic;

-- debug
    signal debug : std_logic_vector(15 downto 0);
    SIGNAL debug_fifo_out_empty : std_logic;
    SIGNAL debug_fifo_out_full : std_logic;
    signal debug_fifo_out_q : std_logic_vector(15 downto 0);
    signal debug_fifo_usedw : std_logic_vector(7 downto 0);
    SIGNAL debug_lcd_state : LCDFSM;
    SIGNAL debug_dma_state : AcqState;
    signal debug_acq_data_transfer : std_logic_vector(15 downto 0);

begin

    top : entity work.top
        port map(
            clk => clk,
            nReset => nReset,

            AS_Address => AS_Address,
            AS_Write => AS_Write,
            AS_Read => AS_Read,
            AS_DataWrite => AS_DataWrite,
            AS_DataRead => AS_DataRead,
```

```
            AM_Address => AM_Address,
            AM_Read => AM_Read,
            AM_ReadData => AM_ReadData,
            AM_ReadDataValid => AM_ReadDataValid,
            AM_WaitRequest => AM_WaitRequest,

            D => D,
            DCX => DCX,
            WRX => WRX,
            RESX => RESX,
            CSX => CSX,
            debug => debug,
            debug_fifo_out_empty => debug_fifo_out_empty,
            debug_fifo_out_full => debug_fifo_out_full,
            debug_lcd_state => debug_lcd_state,
            debug_dma_state => debug_dma_state,
            debug_fifo_out_q => debug_fifo_out_q,
            debug_fifo_usedw => debug_fifo_usedw,
            debug_acq_data_transfer => debug_acq_data_transfer

);

clk_gen : process
    begin
        clk <= '1';
        wait for CLK_PERIOD / 2;
        clk <= '0';
        wait for CLK_PERIOD / 2;
    end process clk_gen;

simulation : process
    procedure async_reset is
        begin
            wait until rising_edge(clk);
            wait for CLK_PERIOD / 4;
            nReset <= '0';
            wait for CLK_PERIOD * 2;
            nReset <= '1';

    end procedure async_reset;

begin

wait for CLK_PERIOD;


async_reset;


AS_Address <= x"0004";      -- 0b0000000000001010
AS_Write <= '1';
AS_DataWrite <= x"0004";
wait for 10000us;

-- send command
--set cmd reg to cmd 0x11
AS_Address <= x"0005";      -- 0b0000000000001010
```

```
AS_Write <= '1';
AS_DataWrite <= x"AA11";
wait for CLK_PERIOD * 2;
AS_Read <= '1';
wait for CLK_PERIOD * 2;

-- set cmd nb param to 0
AS_Address <= x"0006";      -- 0b0000000000001010
AS_Write <= '1';
AS_DataWrite <= x"0000";    -- 0b0000000000000000
wait for CLK_PERIOD * 2;
AS_Read <= '1';
wait for CLK_PERIOD * 2;

AS_Address <= x"0004";      -- 0b0000000000001000
AS_Write <= '1';
AS_DataWrite <= x"0002";    -- 0b0000000000000010

wait for CLK_PERIOD * 2;
AS_Read <= '1';
wait for CLK_PERIOD * 2;


-- Image address register
AS_Address <= x"0000";      -- 0b0000000000000100
AS_Write <= '1';
AS_DataWrite <= x"000A";    -- 0b0000000000001010
wait for CLK_PERIOD * 2;
AS_Read <= '1';
wait for CLK_PERIOD * 2;

-- Image length register
--set to 10 (bytes ? )
AS_Address <= x"0001";      -- 0b0000000000000010
AS_Write <= '1';
AS_DataWrite <= x"0010";
wait for CLK_PERIOD * 2;
AS_Read <= '1';
wait for CLK_PERIOD * 2;

-- Image length register
AS_Address <= x"0002";      -- 0b0000000000000010
AS_Write <= '1';
AS_DataWrite <= x"0000";
wait for CLK_PERIOD * 2;
AS_Read <= '1';
wait for CLK_PERIOD * 2;

-- AS flag register
-- set lcd enable bit( = flag_reg(0)) to 1 in AS
AS_Address <= x"0003";      -- 0b0000000000001000
AS_Write <= '1';
AS_DataWrite <= x"00FF";    -- 0b0000000000000001

wait for CLK_PERIOD * 2;
AS_Read <= '1';
wait for CLK_PERIOD * 2;
```

```
AS_Address <= x"0004";      -- 0b0000000000001000
AS_Write <= '1';
AS_DataWrite <= x"0001";    -- 0b0000000000000010

wait for CLK_PERIOD * 2;
AS_Read <= '1';
wait for CLK_PERIOD * 2;
AS_Read <= '0';

--  -- Simulating Slave response to DMA request
AM_ReadDatavalid <= '1';
AM_WaitRequest <= '0';
AM_ReadData <= x"0011";       -- 0b00010001
wait for CLK_PERIOD;
AM_ReadData <= x"0022";       -- 0b00010010
wait for CLK_PERIOD;
AM_ReadData <= x"0033";       -- 0b00010011
wait for CLK_PERIOD;
AM_ReadData <= x"0044";       -- 0b00010100
wait for CLK_PERIOD;
AM_ReadData <= x"0055";       -- 0b00010101
wait for CLK_PERIOD;
AM_ReadData <= x"0066";       -- 0b00010110
wait for CLK_PERIOD;
AM_ReadData <= x"0077";       -- 0b00010111
wait for CLK_PERIOD;
AM_ReadData <= x"0088";       -- 0b00011000
wait for CLK_PERIOD;
AM_ReadData <= x"0099";       -- 0b00011001
wait for CLK_PERIOD;
AM_ReadData <= x"00AA";       -- 0b00011010
wait for CLK_PERIOD;
AM_ReadData <= x"00BB";       -- 0b00011011
wait for CLK_PERIOD;
AM_ReadData <= x"00CC";       -- 0b00011100
wait for CLK_PERIOD;
AM_ReadData <= x"00DD";       -- 0b00011101
wait for CLK_PERIOD;
AM_ReadData <= x"00EE";       -- 0b00011110
wait for CLK_PERIOD;
AM_ReadData <= x"00FF";       -- 0b00011111
wait for CLK_PERIOD;
AM_ReadDatavalid <= '0';
wait for CLK_PERIOD*2;
AM_ReadDatavalid <= '1';
--  -- Second burst of pixels
wait for CLK_PERIOD * 2;
AM_WaitRequest <= '0';
AM_ReadData <= x"0011";
wait for CLK_PERIOD;
AM_ReadData <= x"0022";
wait for CLK_PERIOD;
AM_ReadData <= x"0033";
wait for CLK_PERIOD;
AM_ReadData <= x"0044";
wait for CLK_PERIOD;
```

```
AM_ReadData <= x"0055";
wait for CLK_PERIOD;
AM_ReadData <= x"0066";
wait for CLK_PERIOD;
AM_ReadData <= x"0077";
wait for CLK_PERIOD;
AM_ReadData <= x"0088";
wait for CLK_PERIOD;
AM_ReadData <= x"0099";
wait for CLK_PERIOD;
AM_ReadData <= x"00AA";
wait for CLK_PERIOD;
AM_ReadData <= x"00BB";
wait for CLK_PERIOD;
AM_ReadData <= x"00CC";
wait for CLK_PERIOD;
AM_ReadData <= x"00DD";
wait for CLK_PERIOD;
AM_ReadData <= x"00EE";
wait for CLK_PERIOD;
AM_ReadData <= x"00FF";
wait for CLK_PERIOD;
AM_ReadData <= x"00CD";
wait for CLK_PERIOD;
AM_ReadDatavalid <= '0';

wait;

end process simulation;
end architecture test;
```