Implementation of an LCD Programmable Interface - Lab 4.0

David Spielmann, Saverio Nasturzio

January 5, 2022

1 Introduction

The goal of this final lab is to implement the conceptual design which has been defined in lab 3.0. In a first step, we implement an LCD controller that fetches in a direct memory access (DMA) fashion an image from memory and displays it onto the LT24 LCD. Secondly, we team up with a group that implemented the camera part of this project. The two implementations should then be merged such that a picture taken by the camera can be displayed on the LCD. More precisely, the implementation of the camera module saves a picture to memory which in turn should then be fetched and displayed to the LCD by our implementation. The final goal will be to interface to the Camera Module implemented by Donzé Léane Marie Josée and Alice Guntli, shared with the other LT24 group of Gianluca Radi and Gaspard Pierre Leroy.

2 Register Map

We recall the register map from lab 3.0 in Fig. 1. Furthermore, we decided to make a small change in the flag register. To be precise, we switched the position of the reset and the send_command flags. For the rest of the register map, we stuck to the defined register map from lab 3.0.

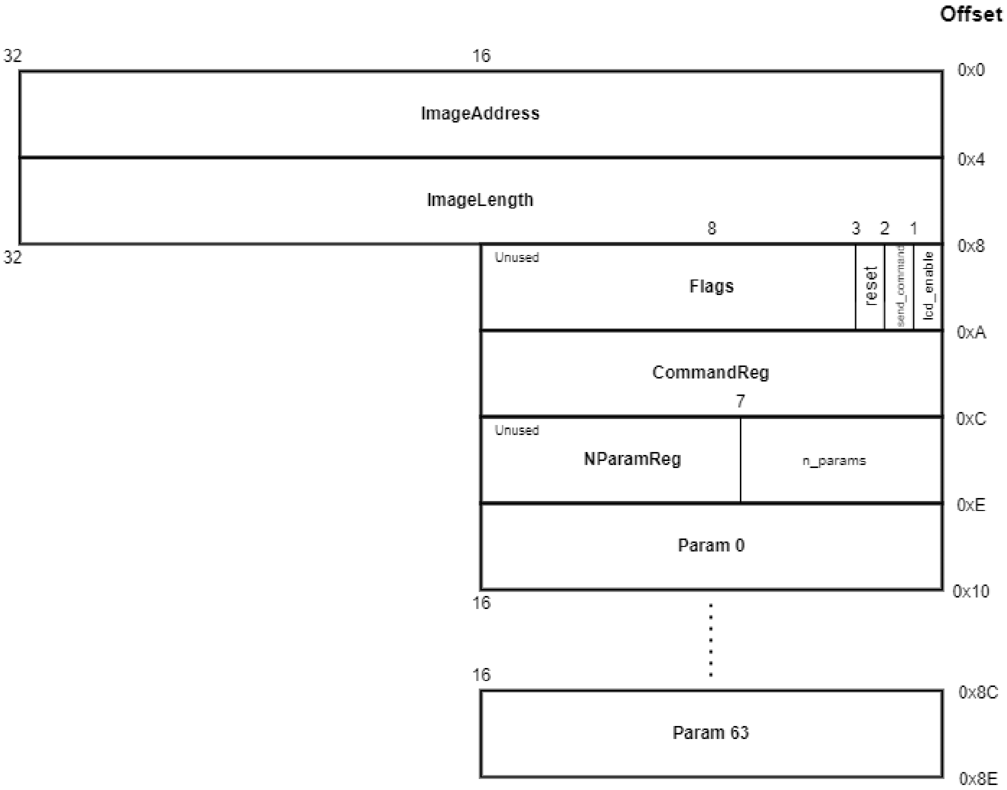


Figure 1: Register Map of the Programmable Interface.

3 Updated Finite State Machines (FSM)

During the implementation of the defined finite state machines in lab 3.0, we decided to make some small changes and improvements in the FSMs. For the sake of completeness, the updated FSMs are depicted in Fig. 2 and Fig. 3. In the FSM of the LCD controller, we detected a mistake in the reset procedure. In the FSM of the DMA Controller, shown in Fig. 3, the states WAIT DATA and CHECK DATA slighty changed compared to the FSM from lab 3.0.

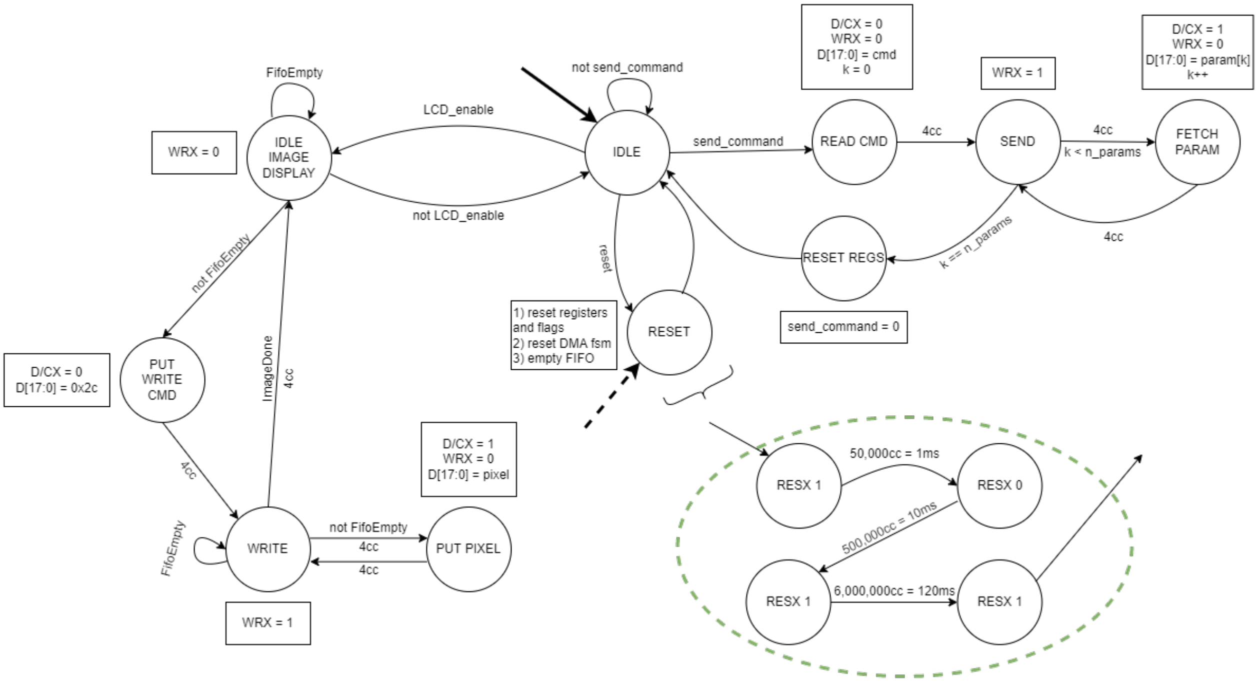


Figure 2: Updated FSMs of our LCD Controller.

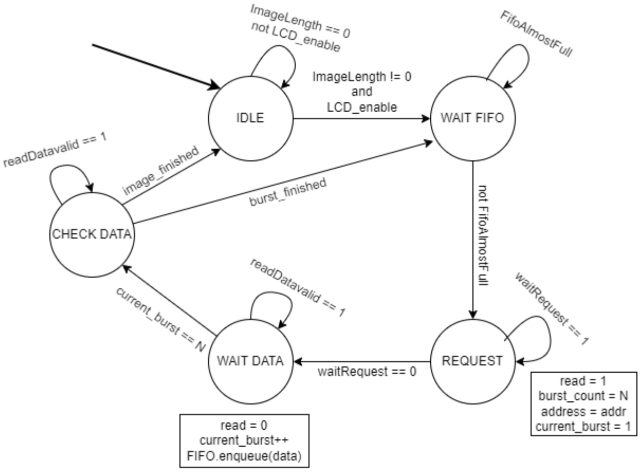


Figure 3: Updated FSMs of our DMA Controller.

4 VHDL Implementation

In this section, we highlight the most important parts of our implementation. As shown in lab 3.0, our top-level component is made of 4 main blocks. We have a DMA controller that is in charge of reading the pixel data from memory at a specified address and pass them to the LCD controller through an intermediary FIFO module. The final component is a Register File (Slave) that can be programmed by a C file that runs on the NIOS processor and specifies the values of the registers. It allows to customize the behavior of our programmable interface.

Due to the size of the image, we must externally provide a DDR3 module to store the image. To avoid accessing reserved memory regions, we are provided with an extender through which enough memory is reserved for our use.

4.1 Register File

Our Register File interfaces through the Avalon Bus as a Slave interface with the usual protocol and signals. It provides as output to the other components the values of the registers. In addition, it receives the reset signals for the Flags.

The Slave interface is implemented according to specification by a write and a read process that handle the incoming requests from a Master. We employ the `registers` signal to store the 71 words of the registers and allow for an easy decoding from the process. The write process is also in charge of updating the register file in case the reset signals from the other components are LOW[1]. Finally, the outgoing signals to the other components are set outside of any process Register File component, as shown in Listing 1. **NOTE:** since our slave is 16-bit and both `ImageAddress` and `ImageLength` are 32-bit we must be careful to properly construct the two 32-bit registers according to the system's endianness to avoid providing wrong values to the DMA component.

```
ImageAddress <= registers(1) & registers(0);
ImageLength  <= registers(3) & registers(2);
Flags        <= registers(4);
CommandReg   <= registers(5);
NParamReg    <= registers(6);
Params       <= registers(7 TO 70);
```

Listing 1: Assignments of the outgoing signals to the other components.

4.2 LCD Controller

As pointed out in the previous lab, we have decided to wait for 4 clock cycles in each state before proceeding in order to meet the timing constraints[2]. To this end, we have employed the `cc4` signal throughout the states to be used as a counter.

4.2.1 Reset

We implemented the Reset protocol for the LT24 as explained in lab 3.0, which is invoked when the user sets bit 2 of the Flags register to HIGH. The vhdl code of our reset is shown in Listing 2. We employ `reset_counter` to control the timings of the reset procedure. Before entering the `Reset` state, the `reset_counter` is initialized to 0. During the reset procedure, the `reset_counter` is incremented by 1 at each clock cycle. The LT24 reset is controlled by the `RESX` signal:

- The `RESX` signal is first set to 1 for 50,000 clock cycles[3].
- Next, `RESX` is set to 0 for 10ms.
- Finally, we set `RESX` to 1 again for 120ms. Moreover, we reset the reset flag in the register file using `reset_flag_reset` and transition back to the state `IDLE`.

[1] They are active-low signals.

[2] In reality, due to the way we implemented the state transitions, we employ more than 4 cycles.

[3] This is equivalent to 1ms, given that our clock runs at a frequency of 50Mhz.

```
WHEN Reset =>
        WRX <= '0';
        IF (reset_counter) < 50000 THEN
                -- Wait for 1ms (50 MHz clock) with RESX HIGH
                RESX             <= '1';
                reset_counter    <= reset_counter + 1;
                reset_flag_reset <= '1';
                state            <= Reset;
        ELSIF (reset_counter) < 550000 THEN
                -- Wait for 10ms with RESX LOW
                RESX             <= '0';
                reset_counter    <= reset_counter + 1;
                reset_flag_reset <= '1';
                state            <= Reset;
        ELSIF (reset_counter) < 6550000 THEN
                -- Wait for 120ms with RESX HIGH
                RESX             <= '1';
                reset_counter    <= reset_counter + 1;
                reset_flag_reset <= '1';
                state            <= Reset;
        ELSIF (reset_counter) = 6550000 THEN
                RESX             <= '1';
                reset_flag_reset <= '0'; -- Reset the reset flag
                ↪   (Flags(2))
                state            <= Reset;
                reset_counter    <= reset_counter + 1;
        ELSE
                RESX             <= '1';
                reset_flag_reset <= '0';
                reset_counter    <= x"00000000";
                state            <= Idle;
        END IF;
```

Listing 2: Reset routine implemented in VHDL.

We test our reset procedure in ModelSim, shown in Fig. 4, and through the Logic Analyzer we verify the correctness of our implementation. We test the reset routine within the testbench by

- writing to the flag address in the register file: `address_slave` = 0000100.
- setting the third bit to 1: `writedata` = 0000000000000100.
- setting the write signal to 1: `write` = 1.

The reset routine is defined in the ILI9341 datasheet. The screenshot of the simulation is depicted in Fig. 4. The `RESX` signal is first set to HIGH for 1ms. Then the signal is set to LOW for 10ms followed by a 120ms period where the signal is HIGH again. After this procedure, the LCD is reset, and the RESX signal stays HIGH.

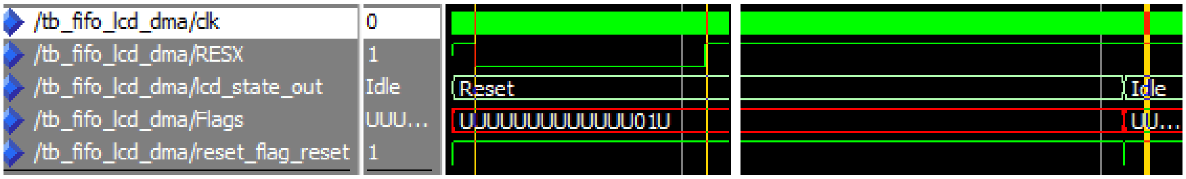


Figure 4: Simulation of the reset routine in ModelSim.

NOTE: To make it more readable, we have cropped the testbench wave forms to reduce the 120ms wait. All transitions respect the required timings.

4.2.2 Command Transmission

The command transmission phase starts when the user sets bit 1 of the Flags register to HIGH after having written the values to the corresponding registers (CommandReg, NParamReg, and Params).

From the IDLE state, when the flag is read HIGH, we transition to the ReadCmd state in which D_CX is set to LOW (indicating a command), WRX is reset, and the command value is placed onto the bus. After the usual delay we transition to Send to commit the value. If we have sent all parameters of the command, then the condition `k = unsigned(NParamReg)` becomes true and hence we can transition back to IDLE by going through the ResetRegs state and clearing the flag. Otherwise, we go to the state `FetchParam`. In this state, we raise the `D_CX` signal to 1, set `WRX` to 0 and assign the parameter stored at index k of the Params Register File to the data bus `D`. After the usual delay, we transition back to `Send` and increment the index. This procedure continues until all parameters have been sent.

```
WHEN Send =>
        WRX <= '1';
        -- We check if we have any parameter left to send
        IF k = unsigned(NParamReg) THEN
                state          <= ResetRegs;
                cc4            <= "000";
                reset_flag_cmd <= '0';
        ELSIF (cc4) < 4 THEN
                cc4   <= cc4 + 1;
                state <= Send;
        ELSE
                cc4   <= "000";
                state <= FetchParam;
        END IF;
WHEN FetchParam =>
        D_CX <= '1';
        WRX  <= '0';
        IF (cc4) < 4 THEN
                D     <= Params(to_integer(k)); -- Extract the parameter
                ↪  to be sent
                cc4   <= cc4 + 1;
                state <= FetchParam;
        ELSE
                state <= Send;
                k     <= k + 1; -- Increment the param index
                cc4   <= "000";
        END IF;
```

Listing 3: VHDL implementation of how we keep alternating between the states `Send` and `FetchParam` in order to transmit commands and its parameters to the LCD.

4.2.3 Pixel Data Transmission

When the user sets the bit 0 of the Flags register to 1, the DMA unit starts reading ImageLength pixels from the address provided in ImageAddress. It inserts the pixel data into the FIFO such that the LCD controller can read from the FIFO and relay them to the LT24.

When the LCD controller is in the Idle state and reads the lcd_enable flag as HIGH, it transitions to IdleImageDisplay where it waits for the FIFO to contain valid data. The first step is to transition to the PutWriteCmd state in which the command 0x2C is sent to the LCD to indicate that pixels are about to be sent. In this state the command is placed onto the `D` bus[4], the `D_CX` bit is set to LOW to indicate a command, and `WRX` is set to LOW to allow for the upwards transition. After the usual delay we transition to WritePixel state, where `WRX` is set to HIGH and the state of the image is checked. After having sent the command we proceed to send the pixels by alternating between the WritePixel and PutPixel states.

- `PutPixel`. In this state, we raise `D_CX` to 1 to indicate that we are about to send data, and `WRX` is set to LOW. In order to correctly extract the pixel from the FIFO, we must raise the `rdreq` signal to HIGH after two cycles so that in the third cycle the FIFO asserts it and provides on the `q` bus the pixel. Finally, in the last cycle before transitioning to WritePixel, we can place on the `D` bus the correct value received from the FIFO.

[4] Interfaced through the GPIO pins to the LT24 display.

- `WritePixel`. We first raise the `WRX` signal to commit the value that is placed on the bus. Moverover, we check if there are any pixels left that are need to be sent (`BytesLeft = 0`). If there are still pixel data that need to be sent, but the FIFO is empty (`empty = 0`), we keep waiting in this state until the FIFO contains some data. Finally, if we have not yet sent the complete image and the FIFO contains data, we have some data to transmit to the LCD and hence can transition to the state `PutPixel` after having decremented the `BytesLeft` counter.

```
WHEN WritePixel =>
        WRX <= '1';
        -- Check if we have sent the whole image
        IF BytesLeft = 0 THEN
                rdreq <= '0';
                IF (cc4) < 4 THEN
                        state <= WritePixel;
                        cc4   <= cc4 + 1;
                ELSE
                        -- Transition back to IdleImageDisplay after
                        ↪  usual delay
                        state <= IdleImageDisplay;
                        cc4   <= "000";
                END IF;
        ELSIF empty = '0' THEN
                -- We have pixels ready to be sent
                IF (cc4) < 4 THEN
                        cc4   <= cc4 + 1;
                        state <= WritePixel;
                ELSE
                        state     <= PutPixel;
                        cc4       <= "000";
                        BytesLeft <= BytesLeft - 2;
                END IF;
        ELSE
                state <= WritePixel;
                cc4   <= "000";
        END IF;
WHEN PutPixel =>
        D_CX <= '1';
        WRX  <= '0';
        IF (cc4) < 2 THEN
                cc4   <= cc4 + 1;
                state <= PutPixel;
                rdreq <= '0';
        ELSIF (cc4) = 2 THEN
                -- At the third cycle we request a word from the FIFO
                cc4   <= cc4 + 1;
                state <= PutPixel;
                rdreq <= '1';
        ELSIF (cc4) = 3 THEN
                -- In this cycle the FIFO has read the request and
                ↪  provides the value onto the bus
                cc4   <= cc4 + 1;
                state <= PutPixel;
                rdreq <= '0';
        ELSE
                -- In this final cycle we have the value ready on the q
                ↪  bus
                cc4   <= "000";
                D     <= q;
                state <= WritePixel;
                rdreq <= '0';
        END IF;
```

Listing 4: VHDL implementation of how we keep alternating between the states `WritePixel` and `PutPixel` in order to transmit pixel data to the LCD.

4.3 DMA Controller

The DMA controller has the task of reading pixels from the address specified in the `ImageAddress` register in bursts of 16 words of 16 bits (1 pixel), and transmitting them to the LCD controller through a FIFO. As depicted in Listing 5, the DMA starts off in state Idle, where it waits for bit 0 of the Flags register to be set to HIGH, and a valid length to be set into register `ImageLength`.

At this point, we transition to WaitFifo to wait until the FIFO as enough space to accommodate an entire burst of 16 pixels. When `almost_full` is 0 (meaning that there is enough space), we initiate the Master read protocol of the Avalon Bus by setting `read` to HIGH, selecting a `burstcount` of 16 words, and providing the correct address onto the `address` bus.

We then transition to the Request state, in which the previous information are expected to be captured by the Slave (memory) and we wait until the `waitRequest` signal is lowered and the bus is granted to our controller. We initialize the `current_burst` signal to 0x1, and it will be used to keep track of the number of received pixels of the current burst.

When the bus is granted, we transition to state WaitData, where we keep increasing `CurrentAddr`[5], decreasing `CurrentLen`[6], and increasing `current_burst`. The writing mechanism to the FIFO is handled combinationally by setting the `wrreq` signal to `readdatavalid`, given that any valid word must be written contextually to the FIFO, and the `data` bus is set to the `readdata` bus that contains the received pixel from memory.

Whenever we terminate an entire burst we transition to state CheckData, in which we check whether we have read the entire image (`CurrentLen` = 0) or we need to request another burst. If we are done with the whole image, we can reset the `lcd_enable` flag and transition to Idle. Otherwise, we transition back to WaitFifo to proceed with another Master burst read.

```vhdl
WHEN Idle =>
        reset_flag_lcdenable <= '1';
        CurrentAddr          <= unsigned(ImageAddress);
        CurrentLen           <= unsigned(ImageLength);
        -- Flags(0) is LCD_enable
        IF Flags(0) = '1' AND ImageLength /= x"00000000" THEN
                state <= WaitFifo;
        END IF;
WHEN WaitFifo =>
        -- We check that there is enough space to fit an entire burst of
        ↪   16 words
        IF almost_full = '0' THEN
                state      <= Request;
                read       <= '1';
                burstcount <= std_logic_vector(N);
                address    <= std_logic_vector(CurrentAddr);
        END IF;
WHEN Request =>
        -- We initialize the counter of the current number of words read
        ↪   within the burst
        current_burst <= x"0001";
        -- When the bus is granted we start waiting for data
        IF waitRequest = '0' THEN
                state <= WaitData;
                read  <= '0';
        END IF;
```

[5] Address at which we are currently reading, initialized at the value provided in ImageAddress.
[6] Number of bytes left to receive, initialized at the value provided in ImageLength.

```
        WHEN WaitData =>
                IF readdatavalid = '1' THEN
                        -- We read words of 2 bytes, so we must increment the
                        ↪  address (byte-addressed) by 2
                        CurrentAddr   <= CurrentAddr + 2;
                        CurrentLen    <= CurrentLen - 2;
                        current_burst <= current_burst + 1;
                END IF;
                -- When we finish an entire burst we check if the image is done
                ↪  in CheckData
                IF current_burst = unsigned(N) THEN
                        state <= CheckData;
                END IF;
        WHEN CheckData =>
                -- Check if image is finished
                IF CurrentLen = 0 THEN
                        state <= ResetFlag;
                        -- Resetting the lcdenable flag
                        reset_flag_lcdenable <= '0';
                ELSE
                        -- We still have more bursts to go, so we transition back
                        ↪  to WaitFifo
                        state <= WaitFifo;
                END IF;
        WHEN ResetFlag =>
                state <= Idle;
END CASE;
```

Listing 5: VHDL implementation of our DMA Controller.

5 Complete Testbench and Evaluation of the System

We employed several testbenches to verify the correct behavior of components by testing them by themselves and incrementally attaching them together to form the entire system. For the sake of conciseness we shall present only the final version that simulates the behavior of the entire subsystem in charge of displaying the image (DMA + FIFO + LCD Controller). To precisely follow the explanation, we suggest to refer to Appendix A for the source code.

Our top level exposes the Master signals to the Avalon bus used by the DMA to interface with the external world, the outgoing signals of the LCD controller towards the LT24, and the input registers from the Register File[7].

The testbench tries to verify the correctness of all parts of the system by dividing the simulation in 3 main phases:

- We first invoke the reset routing by setting Flags(2) to HIGH and wait for its completion[8]
- We then test the Command Transmission routine by sending two commands to the LT24.
- Finally, we set the ImageAddress and ImageLength fields and start the image display by setting Flags(0) to HIGH in order to test the DMA + LCD controller interaction.

5.1 Command Transmission

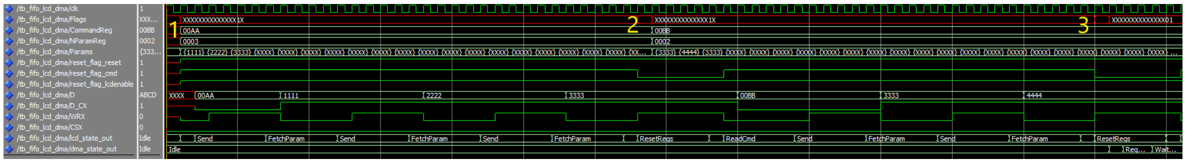
We populate the CommandReg, NParamReg and Params registers with values pertaining to the first command. We then instruct to send the command by setting Flags(1) to HIGH, and repeat the same procedure for a second command.

As depicted in Fig. 5, the command transmission implementation is deemed correct. In point 1, we see that `CommandReg`, `NParamReg`, and `Params` are set with the values of the first command to be sent. Flags(1) is raised to HIGH and our LCD controller transitions to state ReadCmd [9]. At this point, we place on the `D` bus the command and transition to the Send state in which WRX is raised to HIGH.

[7] The slave register file was tested separately and is left out of this simulation to simplify the testbench.

[8] The evaluation of the reset routine can be found in section 4.2

[9] Due to the size of the image, the text was omitted.



/tb_fifo_lcd_dma/clk 1
/tb_fifo_lcd_dma/Flags XXX...
/tb_fifo_lcd_dma/CommandReg 00BB
/tb_fifo_lcd_dma/NParamReg 0002
/tb_fifo_lcd_dma/Params {333...
/tb_fifo_lcd_dma/reset_flag_reset 1
/tb_fifo_lcd_dma/reset_flag_cmd 1
/tb_fifo_lcd_dma/reset_flag_lcdenable 1
/tb_fifo_lcd_dma/D ABCD
/tb_fifo_lcd_dma/D_CX 1
/tb_fifo_lcd_dma/WRX 0
/tb_fifo_lcd_dma/CSX 0
/tb_fifo_lcd_dma/lcd_state_out Idle
/tb_fifo_lcd_dma/dma_state_out Idle

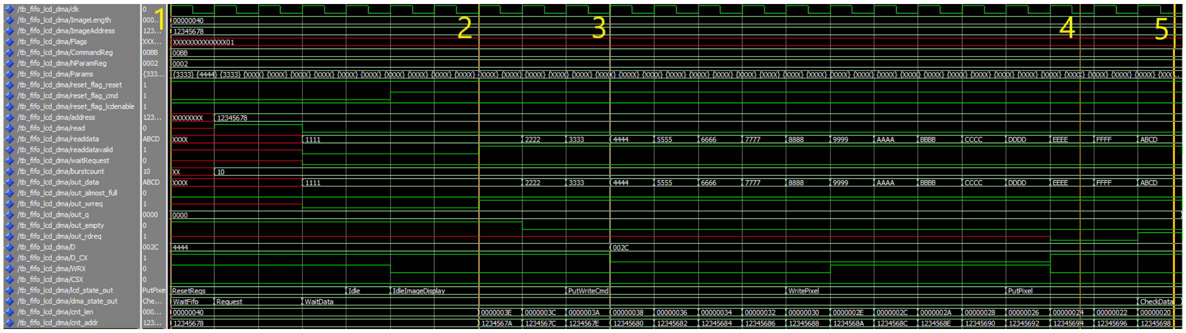
Figure 5: Simulation of the command transmission routine in ModelSim.

For each parameter we load it from the `Params` registers, place it on the but, raise D_CX to HIGH to indicate a command, and proceed to the Send state to commit the value. After the three parameters have been sent, we transition to ResetRegs, set the `reset_flag_cmd` to LOW to reset Flags(1), and transition back to Idle to wait for a new command.

The same procedure is repeated for the second command from point 2.

5.2 Image Display

In order to display the image, we set the ImageAddress and ImageLength registers and set Flags(0) to HIGH to start the LCD controller. We then wait for the DMA to issue the request on the Avalon bus and manually reply with dummy values to fill 2 burst requests[10]. We also simulate the Slave leaving `readdatavalid` to LOW for a few cycles to make sure we respect the correct protocol.



/tb_fifo_lcd_dma/clk 0
/tb_fifo_lcd_dma/ImageLength 000...
/tb_fifo_lcd_dma/ImageAddress 123...
/tb_fifo_lcd_dma/Flags XXX...
/tb_fifo_lcd_dma/CommandReg 00BB
/tb_fifo_lcd_dma/NParamReg 0002
/tb_fifo_lcd_dma/Params {333...
/tb_fifo_lcd_dma/reset_flag_reset 1
/tb_fifo_lcd_dma/reset_flag_cmd 1
/tb_fifo_lcd_dma/reset_flag_lcdenable 1
/tb_fifo_lcd_dma/address 123...
/tb_fifo_lcd_dma/read 0
/tb_fifo_lcd_dma/readdata ABCD
/tb_fifo_lcd_dma/readdatavalid 1
/tb_fifo_lcd_dma/waitRequest 0
/tb_fifo_lcd_dma/burstcount 10
/tb_fifo_lcd_dma/out_data ABCD
/tb_fifo_lcd_dma/out_almost_full 0
/tb_fifo_lcd_dma/out_wrreq 1
/tb_fifo_lcd_dma/out_q 0000
/tb_fifo_lcd_dma/out_empty 0
/tb_fifo_lcd_dma/out_rdreq 1
/tb_fifo_lcd_dma/D 002C
/tb_fifo_lcd_dma/D_CX 1
/tb_fifo_lcd_dma/WRX 0
/tb_fifo_lcd_dma/CSX 0
/tb_fifo_lcd_dma/lcd_state_out PutPixe
/tb_fifo_lcd_dma/dma_state_out Che...
/tb_fifo_lcd_dma/cnt_len 000...
/tb_fifo_lcd_dma/cnt_addr 123...

Figure 6: DMA access and FIFO insertion.

As depicted in Fig. 6 in point 1, as soon as Flags(0)[11] is set to HIGH, the DMA controller transitions to WaitFifo, in which it asserts the `almost_full` signal from the FIFO to make sure an entire burst can fit inside. The LCD controller is still finishing up the previous command transmission and does not react immediately to the command. Since the FIFO is not full, the DMA controller transitions to Request, where it prepares the request to the Slave (memory) according to the Avalon Specification. In particular, it places on the `address` bus the content of `ImageAddress`, it places 0x10 (dec 16) on the `burst_count` bus, and the `read` signal is set to HIGH. As soon as the bus is granted (`waitRequest` = 0), it transitions to WaitData, where it keeps sampling the `readdatavalid` line for valid words to arrive.

Since we are simulating a delay in the data from the Slave, we keep the `readdatavalid` to LOW for a few cycles, after which it is set to HIGH in point 2 and the DMA controller starts to insert the data into the FIFO. For this to happen, signal `wrreq` follows the behavior of `readdatavalid`, and the received data in `readdata` is placed onto the FIFO's bus `data`. At each cycle, we see that the subsequent value is en-queued and both the `cnt_addr`[12], and `cnt_len`[13] are properly updated.

Finally, in point 5 the DMA has completely received and en-queued the first burst of 16 words. In CheckData, it sees that a new burst is required to complete the image, and will transition back to WaitFifo.

In parallel, as soon as the command was completely transmitted, the LCD controller transitions back to Idle and, since Flags(0) is seen HIGH, directly to IdleImageDisplay. In this state it waits for the

[10] ImageLength was set to 64 bytes, which requires 2 bursts of 16 words (2 bytes) to complete.

[11] lcd_enable flag

[12] Current address at which it is reading.

[13] Number of bytes left to receive.

first pixel to be available in the FIFO. After `empty` is read LOW, it transitions to the PutWriteCmd state, in which the 0x2C command is placed onto the `D` bus (point 3) and sent on the following WritePixel[14].

After the usual delay, the LCD controller transitions to state PutPixel, in which it starts fetching pixels from the FIFO as explained in section 4.2.2.

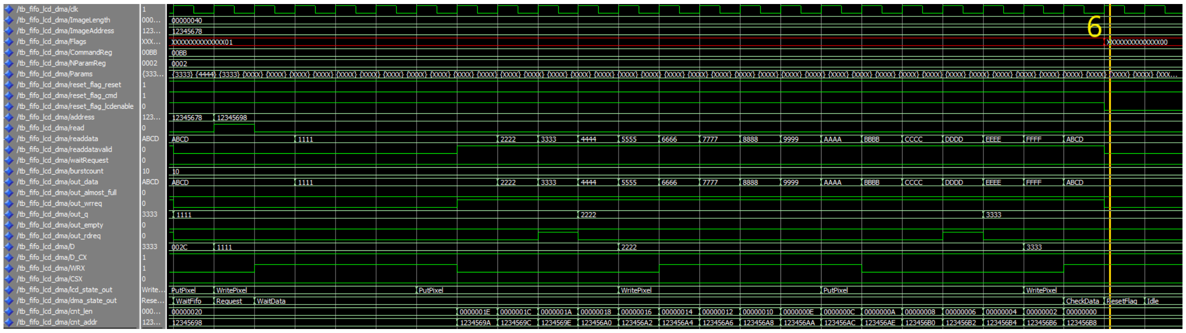

/tb_fifo_lcd_dma/clk
/tb_fifo_lcd_dma/ImageLength
/tb_fifo_lcd_dma/ImageAddress
/tb_fifo_lcd_dma/Flags
/tb_fifo_lcd_dma/CommandReg
/tb_fifo_lcd_dma/NParamReg
/tb_fifo_lcd_dma/Params
/tb_fifo_lcd_dma/reset_flag_reset
/tb_fifo_lcd_dma/reset_flag_cmd
/tb_fifo_lcd_dma/reset_flag_lcdenable
/tb_fifo_lcd_dma/address
/tb_fifo_lcd_dma/read
/tb_fifo_lcd_dma/readdata
/tb_fifo_lcd_dma/readdatavalid
/tb_fifo_lcd_dma/waitRequest
/tb_fifo_lcd_dma/burstcount
/tb_fifo_lcd_dma/out_data
/tb_fifo_lcd_dma/out_almost_full
/tb_fifo_lcd_dma/out_wrreq
/tb_fifo_lcd_dma/out_q
/tb_fifo_lcd_dma/out_empty
/tb_fifo_lcd_dma/out_rdreq
/tb_fifo_lcd_dma/D
/tb_fifo_lcd_dma/D_CX
/tb_fifo_lcd_dma/WRX
/tb_fifo_lcd_dma/CSX
/tb_fifo_lcd_dma/lcd_state_out
/tb_fifo_lcd_dma/dma_state_out
/tb_fifo_lcd_dma/cnt_len
/tb_fifo_lcd_dma/cnt_addr
6

Figure 7: Pixel transmission to LT24.

Figure 7, shows the final part of the simulation, in which the DMA is requesting the second burst according to the same specification and behavior explained above, and the LCD controller transmits the pixels to the LT24 by alternating the PutPixel and WritePixel command.

In point 6, the DMA has concluded the second burst and reverts back to Idle after having reset the Flags(0) bit to indicate that the image has been inserted into the FIFO. The LCD controller keeps fetching 1 pixel at a time from the FIFO by setting HIGH signal `rdreq` and sends it to the LCD through the same protocol, respecting the required timing constraints.

When the image is done, the LCD controller transitions back to IdleImageDisplay and then, since Flags(0) is now LOW, back to Idle for the remaining of the simulation.

6 Deployment and Usage

To deploy the system on our FPGA we undergo the same procedure carried out for lab 2.2. Through Platform Designer we define the entire structure of the system, connecting our DMA as a master interface, our Register File as a slave interface, and exposing the LCD controller's outputs as conduits.

In the top-level file, we attach such outputs to the GPIO_0 pins that are connected to the LT24 display.

After having programmed our FPGA, we can use a simple C program, which is run on the NIOS processor, to interact with our LCD controller. The main function of our C code can be found in Listing 6. For the whole C source code, please refer to the fully commented file attached in the submission. As mentioned in the lecture, the first thing we do is to reset the LCD. After having executed the reset procedure in `reset_lcd()`, we continue to initialize the LCD in the `init_lcd()` function. In this function, we send all the commands including their parameters to the LCD[15]. The next function is `init_image` where write an image to memory. To this end, the function needs to know the start of the address (`image_address`) and the size of the image (`image_size`). The next step is then to write these information to the register file. In `configure_image`, we write the image address (`image_address`) and the image size (`image_size`) to their corresponding positions in the register file. Finally, by setting the `lcd_enable` flag in the `start_lcd` function, we can start displaying the image from memory to the LCD.

As previously mentioned, all memory accesses must go through the DDR3 extender, which is accessed through the Avalon bus like any other memory mapped device. Accesses to memory employ the usual `IOWR` and `IORD` macros provided and use as base address HPS_0_BRIDGES_BASE.

[14] The naming is not very accurate, since it is also used to send the first command as well as all the pixel data.

[15] These commands differ from the snippet provided in the lecture to accomodate the format agreed upon between camera and lcd as defined in lab 3.0.

```
int main()
{
        reset_lcd();
        init_lcd();
        uint32_t image_size = 320 * 240 * sizeof(uint16_t);
        uint32_t image_address = HPS_0_BRIDGES_BASE;
        init_image(image_address, 320, 240);
        configure_image(image_address, image_size);
        start_lcd();
        return 0;
}
```

Listing 6: Main function of our C code.

NOTE: in our implementation the `lcd_enable` flag is reset by the DMA controller when it finished fetching the image from memory, but it does not correspond to the completion of the pixel transmission to the LT24. In theory, a problem could arise if as soon as the `lcd_enable` is set LOW, another image is displayed by immediately setting the flag to HIGH. This is because the DMA, which is faster than the LCD controller due to less stringent timing requirements, would insert the new pixels into the FIFO before the previous image had finished displaying. The LCD controller would treat the new pixels as pixels of the old image and mess up the visualization.

In practice this bug would not arise because, since every flag access is has to go through the NIOS and Avalon bus, the timing required to perform that should surpass the time taken to display the whole image since the display has a low resolution and we do not have many pixels.

A simple solution would be to extend the DMA finite state machine to wait before requesting new pixels if the FIFO is not empty.

7 Final Results

Testing on the FPGA the final result, we can see the correct implementation of our controllers. In particular, we are able to display custom images written to memory[16] to the LCD as shown in Figure 11. We have also checked that the signals are correctly output onto the GPIO pins with the help of the Logic Analyzer as depicted in the following Figures.

Figure 8 shows the entire process of displaying an image and includes the initial reset procedure, the command transmission phase, and finally the pixel transmission. This examples displays an image of alternating stripes of 50 pixels, and explains the variation in data values during the pixel transmission phase.

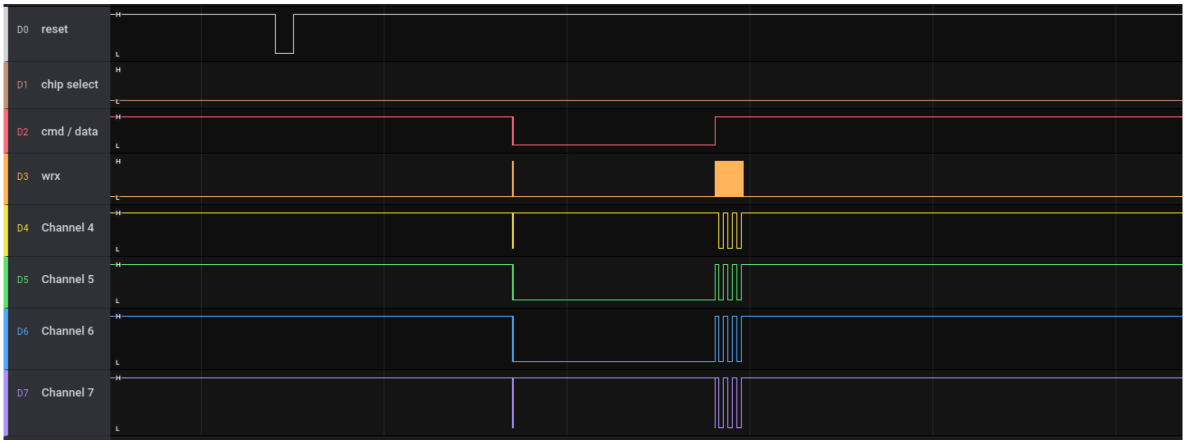


Figure 8: GPIO output of complete image display.

[16] As of the submission of this report, we have only tested the whole image on solid colored or striped backgrounds. Further testing will be carried out by loading from file subsequently.

Figure 9 zooms in on the command transmission phase. Between consecutive commands we spot longer delays and latencies that are probably due to the NIOS involvement in setting the new registers and flags. They do not affect the behavior of the component.

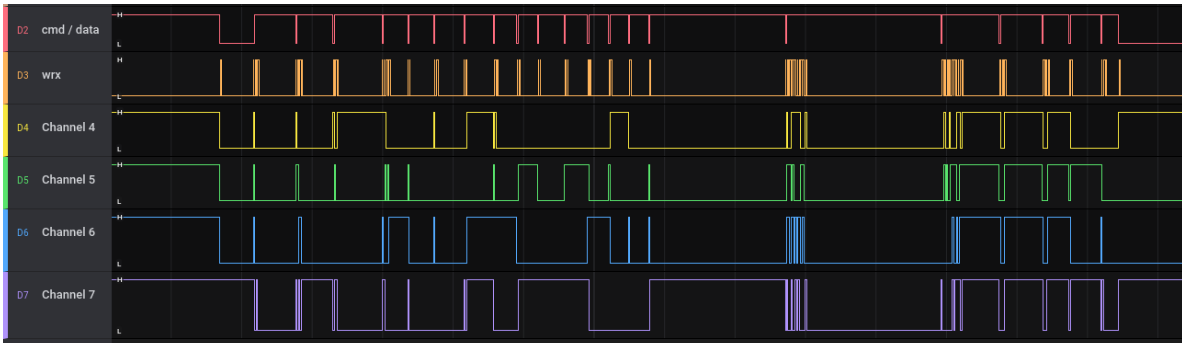


Figure 9: GPIO output of command transmission phase.

Figure 10 zooms in on the pixel transmission phase. We can see the initial 0x2C command being sent and subsequently a bunch of pixels of the same color being transmitted[17].

NOTE: due to a lack of channels in the Logic Analyzer, we are only showing the 4 LSB of the D bus to the LT24.

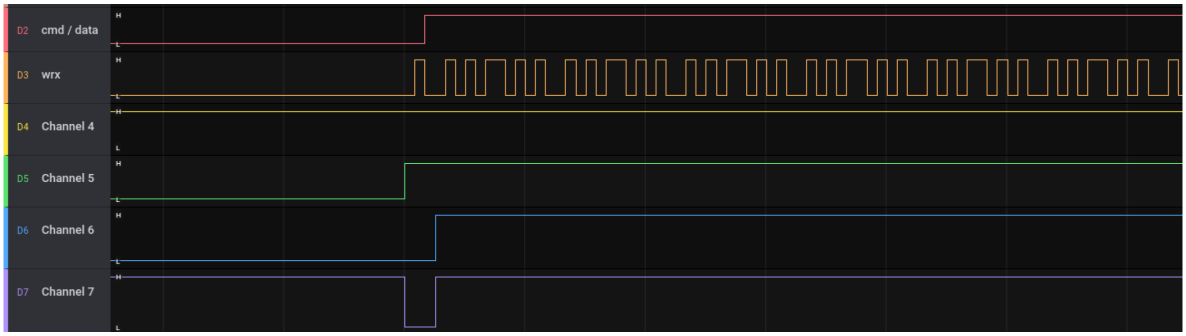


Figure 10: GPIO output of pixel transmission phase.

NOTE: due to a low sampling rate of the Logic Analyzer, not all WRX transitions are shown correctly and it could appear that we are skipping some values. In reality this is not happening since we also checked on a 1 GSample / second oscilloscope, and WRX behaves as expected.

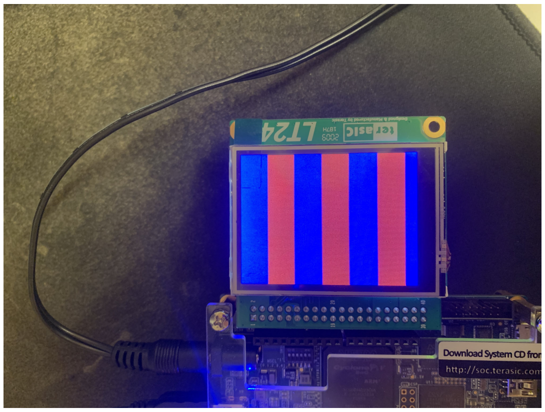


Figure 11: Final Result.

[17] This example displayed an alternating stripe pattern of 50 pixels each.

A Testbench Code

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

entity tb_fifo_lcd_dma is
end tb_fifo_lcd_dma;

architecture test of tb_fifo_lcd_dma is
                CONSTANT CLK_PERIOD : time := 20 ns;

                SIGNAL clk : std_logic;
                SIGNAL nReset :  std_logic;

                signal ImageLength : std_logic_vector(31 downto 0);
                signal ImageAddress : std_logic_vector(31 downto 0);
                signal Flags : std_logic_vector(15 downto 0);
                signal CommandReg : std_logic_vector(15 downto 0);
                signal NParamReg : std_logic_vector(15 downto 0);
                signal Params : RF(0 to 63);

                signal reset_flag_reset : std_logic;
                signal reset_flag_cmd : std_logic;
                signal reset_flag_lcdenable : STD_LOGIC;

                signal address : STD_LOGIC_VECTOR(31 DOWNTO 0);
                signal read : STD_LOGIC;
                signal readdata : STD_LOGIC_VECTOR(15 DOWNTO 0);
                signal readdatavalid : STD_LOGIC;
                signal waitRequest : STD_LOGIC;
                signal burstcount : STD_LOGIC_VECTOR(4 DOWNTO 0);

                signal out_data : std_logic_vector(15 downto 0);
                signal out_almost_full : std_logic;
                signal out_wrreq : std_logic;

                signal out_q : std_logic_vector(15 downto 0);
                signal out_empty : std_logic;
                signal out_rdreq : std_logic;

        -- Outputs to GPIO
        signal D :  STD_LOGIC_VECTOR(15 DOWNTO 0);
        signal D_CX :  STD_LOGIC;
        signal WRX :  STD_LOGIC;
        signal CSX :  STD_LOGIC;
        signal RESX :  STD_LOGIC;

                signal lcd_state_out : LCDState;
                signal dma_state_out : DMAState;
                signal cnt_len : unsigned(31 downto 0);
                signal cnt_addr : unsigned(31 downto 0);

begin

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

                        ImageLength => ImageLength,
                        ImageAddress => ImageAddress,
                        Flags => Flags,
                        CommandReg => CommandReg,
                        NParamReg => NParamReg,
                        Params => Params,

                        reset_flag_reset => reset_flag_reset,
                        reset_flag_cmd => reset_flag_cmd,
                        reset_flag_lcdenable => reset_flag_lcdenable,
                        address => address,
                        read => read,
                        readdata => readdata,
                        readdatavalid => readdatavalid,
                        waitRequest => waitRequest,
                        burstcount => burstcount,

                        out_data => out_data,
                        out_almost_full => out_almost_full,
                        out_wrreq => out_wrreq,
                        out_q => out_q,
                        out_rdreq => out_rdreq,
                        out_empty => out_empty,


                        D => D,
                        D_CX => D_CX,
                        WRX => WRX,
                        CSX => CSX,
                        RESX => RESX,

                        lcd_state_out => lcd_state_out,
                        cnt_len => cnt_len,
                        cnt_addr => cnt_addr,
                        dma_state_out => dma_state_out
                );

        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';
                        Flags <= x"0000";
                        CommandReg <= x"0000";
                        NParamReg <= x"0000";
                        Params <= (others => x"0000");
                end procedure async_reset;
        begin

                wait for CLK_PERIOD;

                -- Reset procedure
                Flags(2) <= '1';

                wait until reset_flag_reset = '0';
                Flags(2) <= '0';

                -- Sending 2 commands to LT24
                CommandReg <= x"00aa";
                NParamReg <= x"0003";
                Params(0) <= x"1111";
                Params(1) <= x"2222";
                Params(2) <= x"3333";
                Flags(1) <= '1';
                wait until reset_flag_cmd = '0';
                Flags(1) <= '0';
                wait for CLK_PERIOD;
                CommandReg <= x"00bb";
```

```
NParamReg <= x"0002";
Params(0) <= x"3333";
Params(1) <= x"4444";
Flags(1) <= '1';
wait until reset_flag_cmd = '0';
Flags(1) <= '0';
wait for CLK_PERIOD;


-- Starting display
ImageAddress <= x"12345678";
ImageLength <= x"00000040";

Flags(0) <= '1';

-- Simulating Slave response to DMA request
wait until read = '1';
wait for CLK_PERIOD * 2;
waitRequest <= '0';
readdata <= x"1111";
readdatavalid <= '0';
wait for CLK_PERIOD * 4;
readdatavalid <= '1';
wait for CLK_PERIOD;
readdata <= x"2222";
wait for CLK_PERIOD;
readdata <= x"3333";
wait for CLK_PERIOD;
readdata <= x"4444";
wait for CLK_PERIOD;
readdata <= x"5555";
wait for CLK_PERIOD;
readdata <= x"6666";
wait for CLK_PERIOD;
readdata <= x"7777";
wait for CLK_PERIOD;
readdata <= x"8888";
wait for CLK_PERIOD;
readdata <= x"9999";
wait for CLK_PERIOD;
readdata <= x"AAAA";
wait for CLK_PERIOD;
readdata <= x"BBBB";
wait for CLK_PERIOD;
readdata <= x"CCCC";
wait for CLK_PERIOD;
readdata <= x"DDDD";
wait for CLK_PERIOD;
readdata <= x"EEEE";
wait for CLK_PERIOD;
readdata <= x"FFFF";
wait for CLK_PERIOD;
readdata <= x"ABCD";
wait for CLK_PERIOD;
readdatavalid <= '0';

-- Second burst of pixels
wait until read = '1';
wait for CLK_PERIOD * 2;
waitRequest <= '0';
readdata <= x"1111";
readdatavalid <= '0';
wait for CLK_PERIOD * 4;
readdatavalid <= '1';
wait for CLK_PERIOD;
readdata <= x"2222";
wait for CLK_PERIOD;
readdata <= x"3333";
wait for CLK_PERIOD;
readdata <= x"4444";
wait for CLK_PERIOD;
```

```
                readdata <= x"5555";
                wait for CLK_PERIOD;
                readdata <= x"6666";
                wait for CLK_PERIOD;
                readdata <= x"7777";
                wait for CLK_PERIOD;
                readdata <= x"8888";
                wait for CLK_PERIOD;
                readdata <= x"9999";
                wait for CLK_PERIOD;
                readdata <= x"AAAA";
                wait for CLK_PERIOD;
                readdata <= x"BBBB";
                wait for CLK_PERIOD;
                readdata <= x"CCCC";
                wait for CLK_PERIOD;
                readdata <= x"DDDD";
                wait for CLK_PERIOD;
                readdata <= x"EEEE";
                wait for CLK_PERIOD;
                readdata <= x"FFFF";
                wait for CLK_PERIOD;
                readdata <= x"ABCD";
                wait for CLK_PERIOD;
                readdatavalid <= '0';

                wait until reset_flag_lcdenable = '0';
                Flags(0) <= '0';


                wait;



        end process simulation;
end architecture test;
```