LCD Interface Implementation

1. About the project

The aim of the project is to develop an LCD controller, which displays images stored in the SDRAM. As LCD we use the LT24 module by Terasic, which includes the ILI9341 LCD driver and a 240x320 display. Incorporating a DMA, our IP reads the data on the SDRAM through the Avalon Bus, then directly communicates with the LCD driver following an 8080 I protocol. The image data is provided by a camera module, following the format we present in this document. The design has been developed for the DE0-Nano-Soc Kit. Quartus, Eclipse, ModelSim and SignalTap were used for development and debugging.

We could not merge our LCD_controller with the camera system because the camera group did not have a fully working implementation. Thus, the presentation of the project was done with the LCD system only, running the demo presented in the last section of this document.

2. Feature Summary

- Capability of sending instructions to the ILI9341 by the CPU
- Minimum involvement of the CPU in data routines
- Reconfigurable frame buffer address and burst count
- Synchronization with the camera implicitly achieved
- Possibility of loading static or dynamic images via software in order to simulate the acquisition from the camera

3. Differences with the conceptual design

The overall system structure such as the general architecture and the custom IP diagram are unchanged.

The system was designed in the QSYS environment. It is composed by an embedded NIOS II processor, its basic requirements (memory and JTAG bridge for debugging) and an LCD controller custom IP. There is also the HPS bridge component, which include the memory controller, since the DDR3 is only available on the HPS side. The address span extender is used to prevent the access to any reserved region of the DDR3 memory. Accesses through the bridge are redirected with a specific offset to a block of continuous 256 MB within the DDR3 memory.

In Fig. 3.2 we report the system contents view generated in QSYS. Here we show detailed system components, base addresses of interfaces and interrupt numbers.

3.1. System Diagram

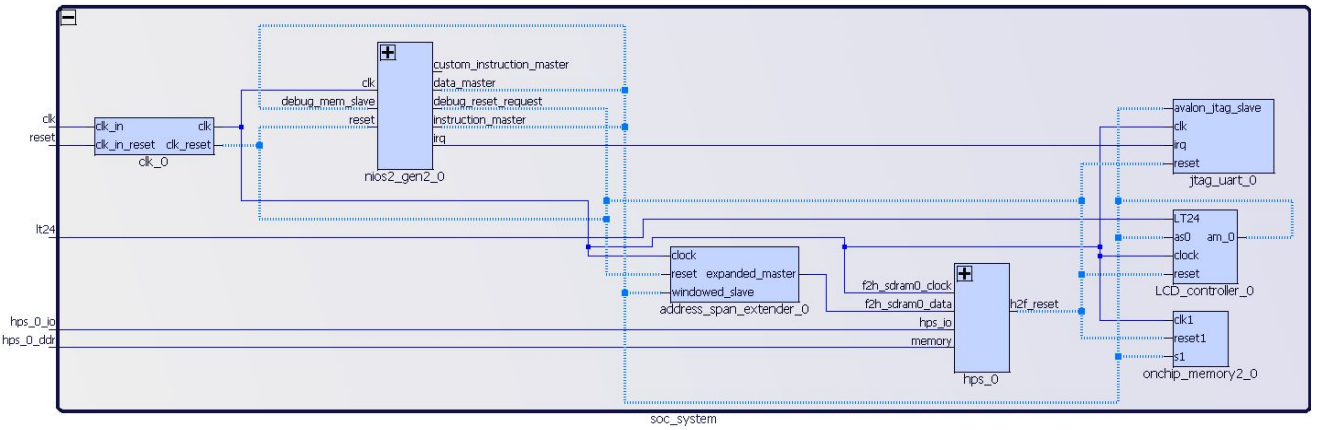


Fig. 3.1

3.2. System Contents

Connections	Name	Description	Export	Clock	Base	End	IRQ
	⊟ **clk_0**	Clock Source					
	clk_in	Clock Input	**clk**	***exported***			
	clk_in_reset	Reset Input	**reset**				
	clk	Clock Output	*Double-click to export*	clk_0			
	clk_reset	Reset Output	*Double-click to export*				
	⊟ **nios2_gen2_0**	Nios II Processor					
	clk	Clock Input	*Double-click to export*	**clk_0**			
	reset	Reset Input	*Double-click to export*	[clk]			
	data_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]			
	instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]			
	irq	Interrupt Receiver	*Double-click to export*	[clk]	IRQ 0	IRQ 31	
	debug_reset_requ...	Reset Output	*Double-click to export*	[clk]			
	debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x1000_0000	0x1000_07ff	
	custom_instructio...	Custom Instruction Master	*Double-click to export*				
	⊟ **onchip_memory2_0**	On-Chip Memory (RAM or ROM)					
	clk1	Clock Input	*Double-click to export*	**clk_0**			
	s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x1010_0000	0x1011_ffff	
	reset1	Reset Input	*Double-click to export*	[clk1]			
	⊟ **jtag_uart_0**	JTAG UART					
	clk	Clock Input	*Double-click to export*	**clk_0**			
	reset	Reset Input	*Double-click to export*	[clk]			
	avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x1000_0800	0x1000_0807	
	irq	Interrupt Sender	*Double-click to export*	[clk]			0
	⊟ **address_span_ex...**	Address Span Extender					
	clock	Clock Input	*Double-click to export*	**clk_0**			
	reset	Reset Input	*Double-click to export*	[clock]			
	windowed_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	0x0000_0000	0x0fff_ffff	
	expanded_master	Avalon Memory Mapped Master	*Double-click to export*	[clock]			
	⊟ **hps_0**	Arria V/Cyclone V Hard Process...					
	memory	Conduit	**hps_0_ddr**				
	hps_io	Conduit	**hps_0_io**				
	h2f_reset	Reset Output	*Double-click to export*				
	f2h_sdram0_clock	Clock Input	*Double-click to export*	**clk_0**			
	f2h_sdram0_data	Avalon Memory Mapped Slave	*Double-click to export*	[f2h_sdra...	0x0000_0000	0xffff_ffff	
	⊟ **LCD_controller_0**	LCD_controller					
	clock	Clock Input	*Double-click to export*	**clk_0**			
	reset	Reset Input	*Double-click to export*	[clock]			
	am_0	Avalon Memory Mapped Master	*Double-click to export*	[clock]			
	as0	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	**0x1000_0810**	0x1000_081f	
	LT24	Conduit	**lt24**	[clock]			

Fig. 3.2

3.3. Custom IP Register Map

Base address: 0x10000810					**Len**: 32b
NAME	**BIT WIDTH**	**BITS**	**DEFAULT**	**MODE**	**ADDRESS OFFSET**
Image start address	32	31 downto 0	0x00000000	R/W	0
Transfer length	21	28 downto 8	0x000000	R/W	4
Burst number	8	7 downto 0	0x00	R/W	4
Start transfer	1	4	0	W	8
Restart	1	0	0	W	8
Busy	1	1	0	R	8
Command/data	16	15 downto 0	0x0000	W	12
DCX (RS)	1	16	1	W	12
continue	1	20	0	W	12
LCD_on	1	24	1	W	12
LCD_resn	1	25	1	W	12
regs_write	1	28	-	W	12

Table 3.1

In *Table 3.1*, we present the register map of our IP.

- ***Image start address*** provides the initial address of the first buffer in memory where the picture is stored.
- ***Transfer length*** provides the number of burst transfer we need in order to load the full image.
- ***Burst number*** : number of burst for each transfer.
- ***Start transfer*** : start the transfer of data from memory to LCD.
- **Restart** : soft-reset of the DMA. Used to flush the FIFO and stop the DMA when the LCD controller has to be reconfigured during the transfer of a frame.
- **Busy** : status bit, stays high when the DMA is performing a transfer.
- ***Command/data*** : a port that can be accessed to send command/data to the LCD.
- ***DCX*** (RS) : must be low when we send a command and high when we send data.
- **continue**: high when the current command/data expect a following instruction.

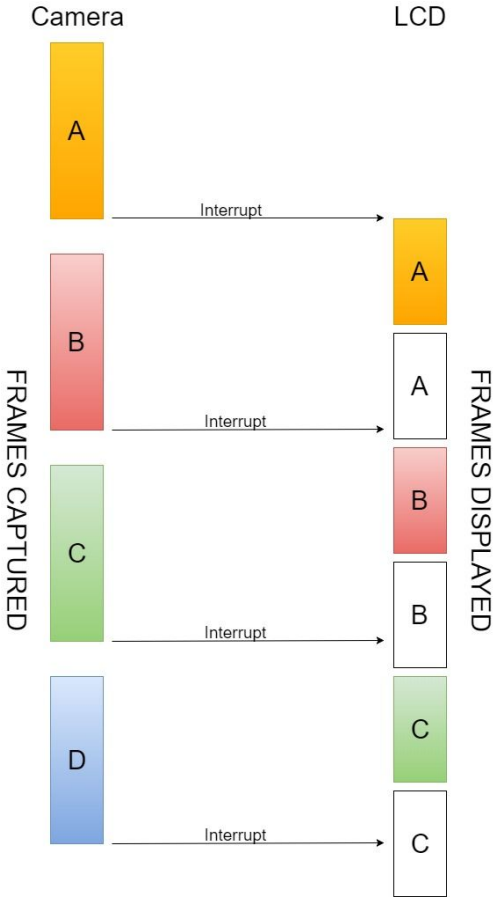
- ***LCD_on***: registered output.
- ***LCD_resn***: registered output.
- ***regs_write***: not a register, but a control bit to differentiate between commands to the LT24 and *LCD_on / LCD_resn* writings.

We deleted the *image 2 start address* and consequently the *buffer selection* bit. The new functional flow, indeed, does not require these registers, as explained in the following section.

3.4. Functional Flow

At first we decided to implement two different modes: photo mode and video mode. At the final stage of the development, the differences between the two modes became only conceptual. Each time a new address is written inside the image start address register, the start transfer bit is asserted and the DMA starts loading the frame from memory. Two possible cases can occur:

LCD faster than camera



LCD slower than camera

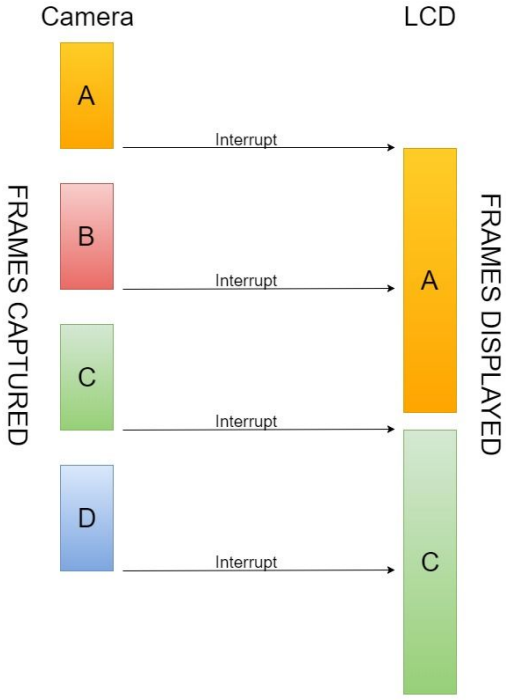


Figure 3.2 - Camera and LCD_controller synchronization. (Frames with white background are loaded from the graphic RAM of the LCD)

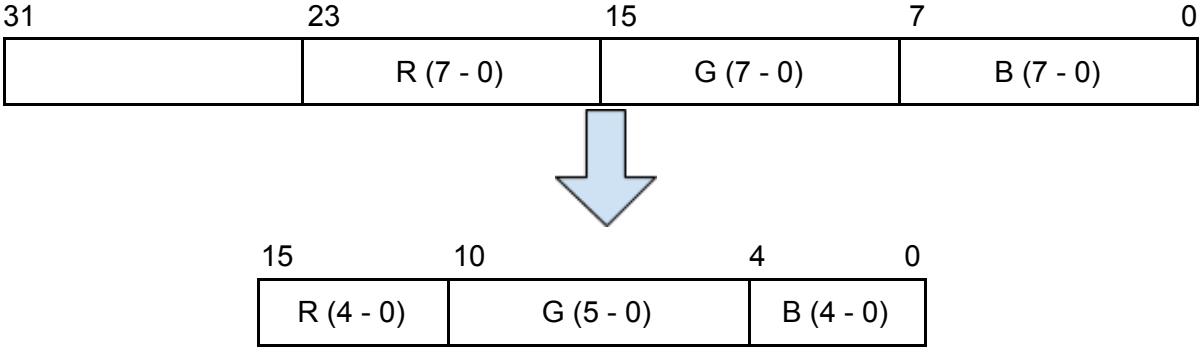
- **LCD faster than camera**
  The camera takes a picture, stores the frame in memory and then sends an interrupt

to the CPU. In the interrupt routine the DMA is started, then it loads the frame from memory and send it to the FIFO. If the DMA finishes loading the frame before the camera takes another picture, there is no need to do another access in memory thanks to the graphical RAM inside the LCD. When the camera finishes to load the second frame in memory a new interrupt is sent and the DMA can load the new image.

- **LCD slower than camera**
  Same routine as before but this time the camera sends two or more interrupt while the DMA is performing a transfer. This does not affect the current transfer. The DMA automatically skip the frames which can't be handled, so some frames may not be displayed[1]. The only bond in this implementation is that the number of frame buffers in the SDRAM must be higher than the expected skipped frames. Since we presume the LCD not to be much slower than the camera, a buffer of 3-4 frames should be enough.

3.5. Data Format

Data stored in the SDRAM is organized in frames. Each frame contains 320*240 pixels. Each pixel is 24 bits of data[2] (RGB: [8,8,8]) and is stored in one 32 bits location (8 bits are unused).



Organization of the data in row-wise or column-wise doesn't influence the data format, thanks to the ability of the ILI9341 LCD controller to easily flip and reflect frames through software configuration.

Since the LCD can only accept a 16 bit wide RGB data [5,6,5], in the master_dma unit we select 16 bits (most significant bits for each color) among the 32 of the SDRAM location we loaded and we send them to the FIFO. Thus the FIFO is organized with a 16 bit data width (1 pixel per location), to fit the LCD data format.

[1] The skipped frames are not visible in a real time video mode, since they are much faster than the human eye response (~20ms). Thus, no flicker or artefact are produced.
[2] Reported by the camera group.

3.6. FIFO Buffer

We implemented the FIFO buffer with the characteristics described in the conceptual design report. The FIFO was created using the *IP MegaWizard* in Quartus, which can be opened through the IP Catalog.

Observing the access time of the DDR3 memory with SignalTap, we noticed a high read latency even greater than 8 clock cycles.

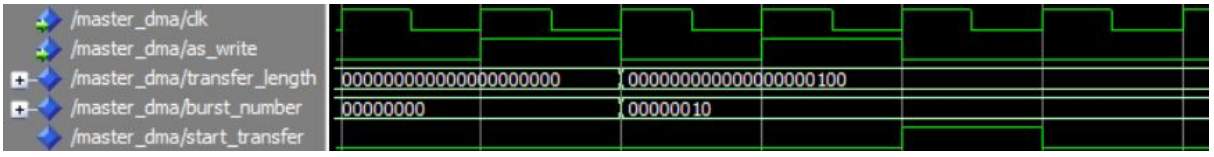
Because of this, a burst number greater or equal than 8 must be used to prevent the LCD from taking data faster than the DMA loads it, which would result in corrupted images on the screen. Since we have implemented the system with a reconfigurable burst number (8, 16 or 32), we set the almost full parameter of the FIFO so that it goes high when 31 locations from completeness are left[3]. In our code, we set the burst number to 16 to not hold the bus busy for too long, even though this means that 16 locations of the FIFO will be unused.

3.7. Avalon master unit

The functional flow of the FSM is unchanged. We added the status bit busy which stays high during the transfer of a frame. This bit can be polled in order to know when the transfer of an entire frame is completed. The *start transfer* bit, which triggers the DMA, now goes high when a new address is written in the image start address register and it is deasserted when the DMA goes from the IDLE state to the READ REQUEST state. The synchronous updating of the memory address is done in the last state CHECK TRANSFER.

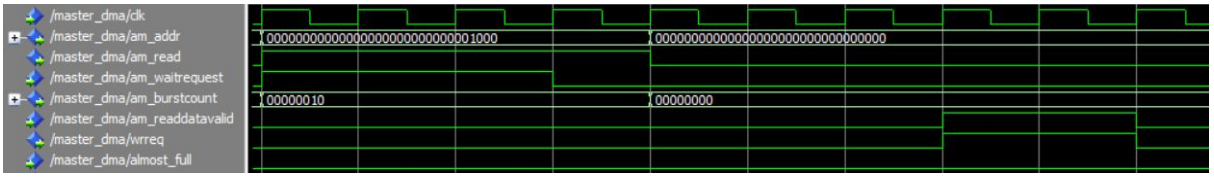
The operating principle of the DMA has already been described in the conceptual design report so here we only illustrate the timing diagrams obtained with ModelSim (burstcount = 2) and SignalTap (burstcount = 32).

Setting and starting the DMA



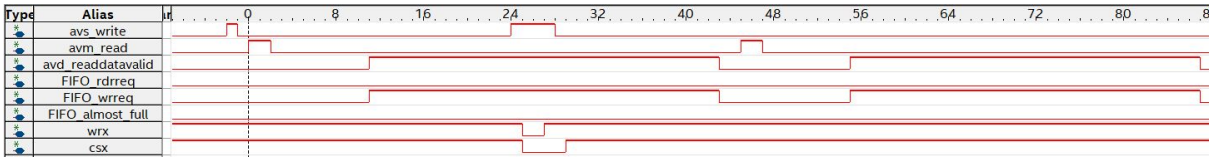
After the configuration of the *burst_number* and *transfer_length* registers, the image start address is written and the DMA is triggered by the *start_transfer* bit.

[3] The FIFO is completely filled only when using 32 as burst number. However, in order to avoid data misalignment when using different burst numbers, we had to configure the FIFO to fit the biggest burst number.



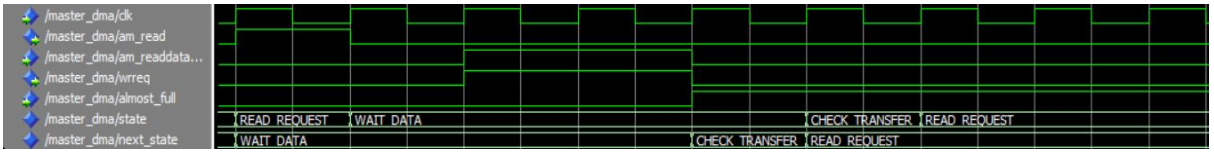
The DMA asserts *address*, *burstcount*, and *read* after the rising edge of clk. The slave asserts *waitrequest*, causing all inputs to be held constant through another clock cycle after the waitrequest is deasserted. The DMA then waits for the *readdatavalid* signal which is directly connected to the *wrreq* signal of the FIFO and allows the writing of valid data.

The right operation of the DMA, observed in the two simulations above, is confirmed by the timing diagram from SignalTap:



Here the second assertion of the avalon slave write signal is used to send the write command to the LT24 interface.

DMA paused

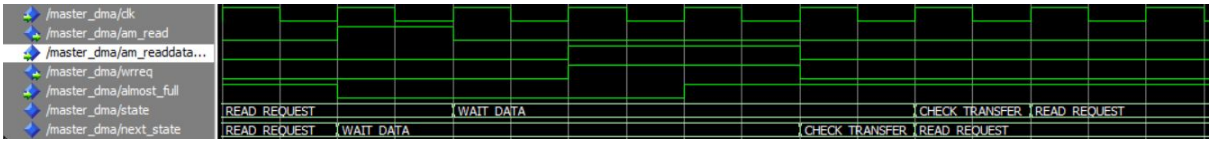


As soon as the *almost_full* signal goes high the DMA is paused and stays in the READ REQUEST state until the signal is deasserted.

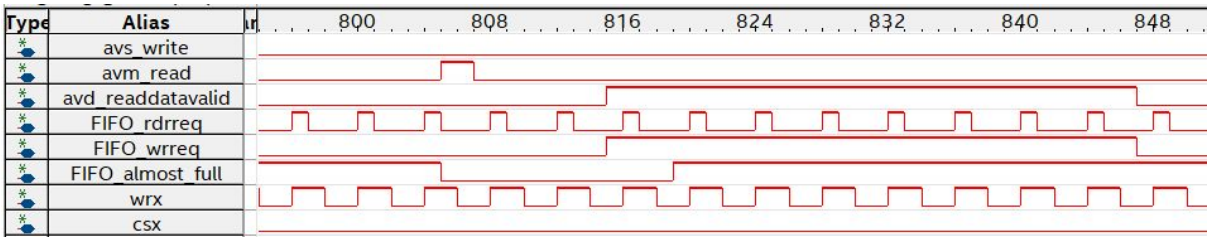
If the *almost_full* signal is asserted during a burst transfer from memory, the DMA at first finishes the current burst transfer and then is paused.

Here we can see how the *almost_full* signal is also used to trigger the LT24 controller.

Almost full transition



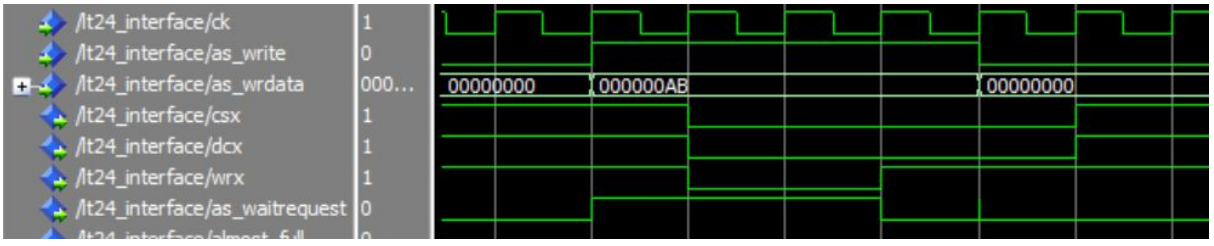
The DMA starts again the loading when the *almost_full* signal goes low. Thanks to the alignment between the FIFO and the burst number, there is an alternation of pausing and loading that allows not to lose any data. This alternation can be observed in the transitions of the *almost_full* signal.



3.8. LT24 Interface

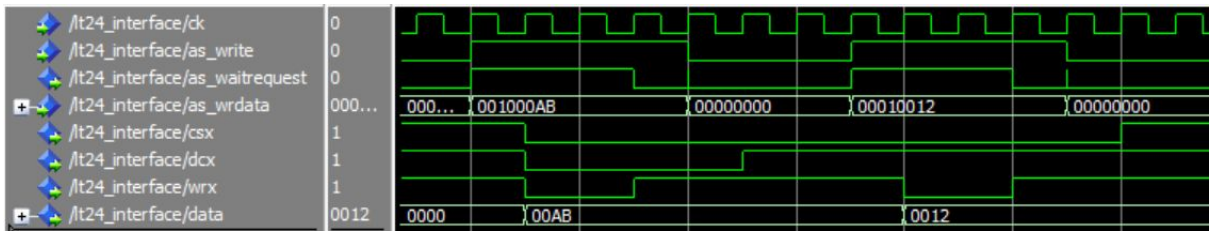
The operating scheme of the LT24 interface was completely unchanged from its conceptual design. The video mode is implicitly achieved with the logic defined in the conceptual FSM, because the DMA controls the *almost_full* signal of the FIFO. Here we present some timing diagrams from Modelsim, which show the correct behaviour of the unit.

Send command/data



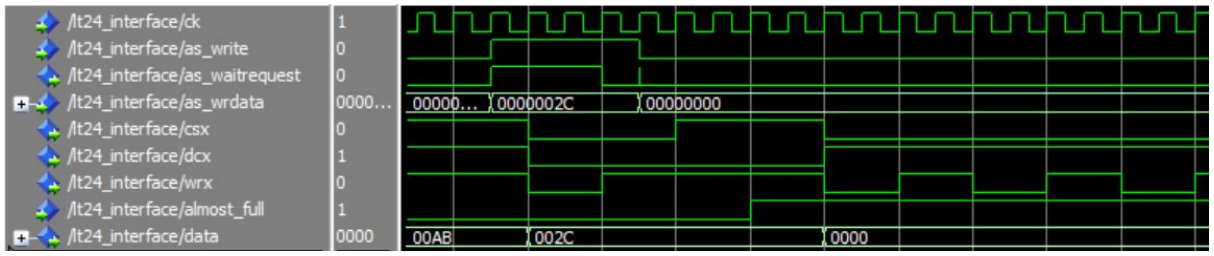
A single command is sent to the LCD. The waitrequest signal is raised for three cycles to keep the bus busy for 4 cycles. The outputs follows the correct timing but delayed of one cycle because they are registered.

Write continuous command followed by data



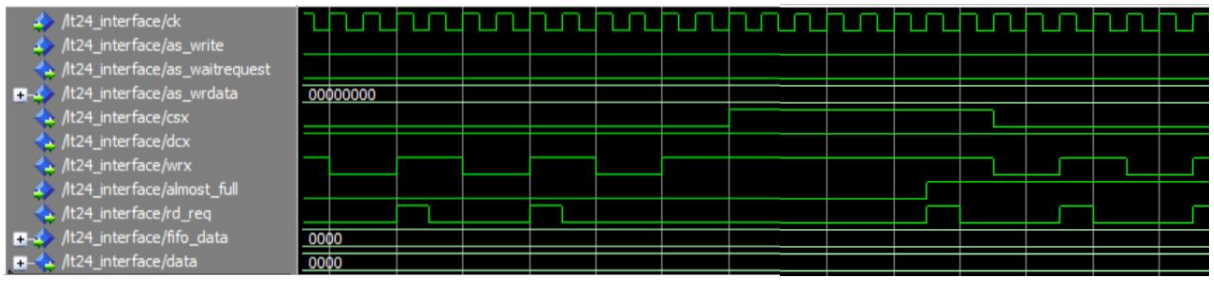
If the command/data is continuous, the *csx* signal stays activated until the continue bit (16th) is asserted.

Start writing into the LCD GRAM



A write command (0x2C) triggers the loading of the frame from the FIFO. The loading actually starts only when almost_full goes high.

Stop writing at the end of the frame and wait for the FIFO to be almost full again



At the end of a frame, the loading pauses (*csx* goes high) until *almost_full* is activated again. This happens only if the DMA has started the loading of a new frame.

4. IP VHDL Code

4.1. LCD controller (Top Level)

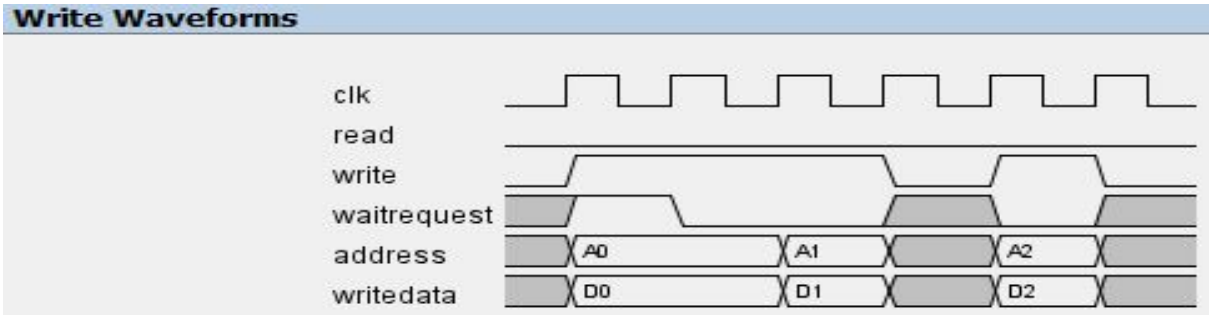
```
entity LCD_controller is
        port(   -- System signals
                clk                     : in std_logic;
                reset_n                 : in std_logic;

                --Avalon Slave signals
                avs_address             : in  std_logic_vector(1 downto 0);
                avs_write               : in  std_logic;
                avs_writedata           : in  std_logic_vector(31 downto 0);
                avs_read                : in  std_logic;
                avs_readdata            : out std_logic_vector(31 downto 0);
                avs_waitrequest         : out std_logic;

                --Avalon Master signals
                avm_address             : out std_logic_vector(31 downto 0);
                avm_read                : out std_logic;
                avm_readdata            : in  std_logic_vector(31 downto 0);
                avm_waitrequest         : in  std_logic;
                avm_byteenable          : out std_logic_vector(3 downto 0);
                avm_burstcount          : out std_logic_vector(7 downto 0);
                avm_readdatavalid       : in  std_logic;

                -- ILI9341 communication signals    (8080 I interface)
                csx                     : out std_logic;
                dcx                     : out std_logic;
                wrx                     : out std_logic;
                data                    : out std_logic_vector(15 downto 0);

                -- LT24 global signals, registered outputs
                LCD_on                  : out std_logic;
                LCD_resn                : out std_logic
                );
end LCD_controller;
```

This is a structural VHDL which connects the components of the IP. Only the Avalon Slave decoder logic is specified here. It selects where to send the avs_write signal, depending on

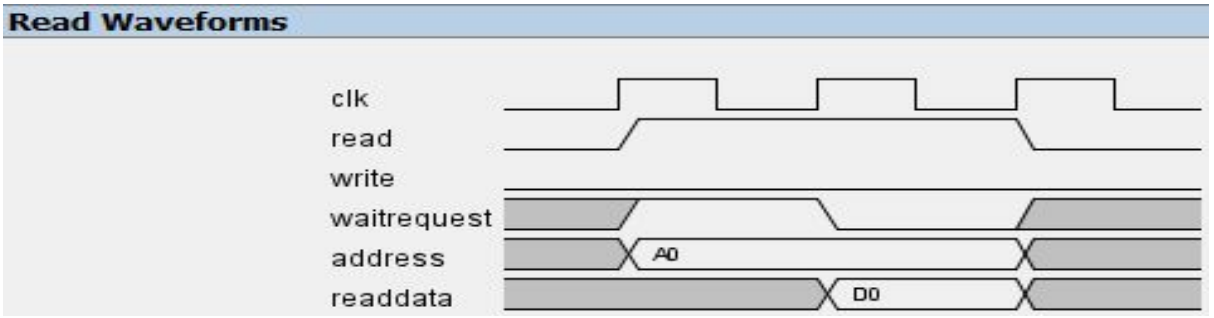
the requested address. It goes to the LT24 only if the slave side address is 3, which corresponds to an offset of 12 on the master side[4].

```
-- Avalon slave decoder
iwrite_to_DMA       <= avs_write when avs_address /= "11" else '0';
iwrite_to_LT24      <= avs_write when avs_address = "11" else '0';
avs_waitrequest     <= iwait_from_DMA or iwait_from_LT24;
```

The Avalon slave interface allows the software configuration of the Master unit and LT24 interface. We report here its timing diagram, as produced by the Qsys wizard:



When *write* is high new values are written in the internal registers which are updated at the next clock cycle.



When *read* is high, data is transferred to the output bus and is available at the next clock cycle.

4.2. Master DMA

```
entity Master_DMA is
        port(
                -- System signals
                clk                     : in  std_logic;
                reset_n                 : in  std_logic;

                -- Avalon Slave signals
                as_addr                 : in  std_logic_vector(1 downto 0);
                as_write                : in  std_logic;
                as_writedata            : in  std_logic_vector(31 downto 0);
```

[4] Slave addresses are word based (32 bit each), while master addresses are Byte based.

```
        as_read             : in  std_logic;
        as_readdata         : out std_logic_vector(31 downto 0);
        as_waitrequest      : out std_logic;

        -- Avalon Master signals
        am_addr             : out std_logic_vector(31 downto 0);
        am_read             : out std_logic;
        am_readdata         : in  std_logic_vector(31 downto 0);
        am_waitrequest      : in  std_logic;
        am_byteenable       : out std_logic_vector(3 downto 0);
        am_burstcount       : out std_logic_vector(7 downto 0);
        am_readdatavalid    : in  std_logic;

        --FIFO signals
        data                : out std_logic_vector(15 downto 0);
        wrreq               : out std_logic;
        almost_full         : in  std_logic;
        sclr                : out std_logic
    );
end entity Master_DMA;
```

The FSM of the DMA is implemented with a three-process template:

- **fsm_states**: a combinatorial process which compute the next state and assigns the combinatorial outputs.
- **DMA_routine**: a synchronous process, sensitive to the rising edge of the clock, which assigns the registered outputs.
- **state_register**: a synchronous process, sensitive to the rising edge of the clock, which assigns the next state to the present state.

In order to set the *start_transfer* bit to 0 inside the *fsm_states* process, we needed a second bit, called *transfer_accepted*, since it is not possible to drive the same signal from two different processes. The *transfer_accepted* bit goes high only for one cycle and triggers the write of the *start_transfe*r bit inside the avalon slave write process.

4.3. LT24 interface

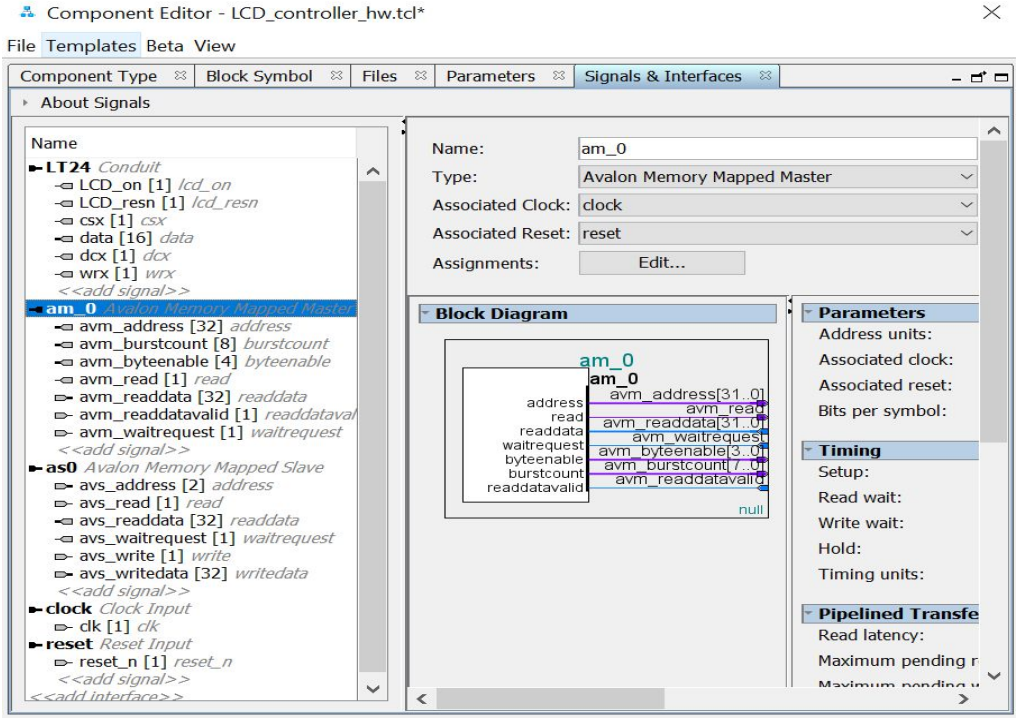
```
entity LT24_interface is
    port(
        -- System signals
        ck              : in std_logic;
        res_n           : in std_logic;

        -- Avalon Slave signals
        as_wrdata       : in std_logic_vector(31 downto 0);
        as_write        : in std_logic;
        as_waitrequest  : out std_logic;

        -- FIFO interface signals
        fifo_data       : in std_logic_vector(15 downto 0);
        almost_full     : in std_logic;
        rd_req          : out std_logic;
```

```
                -- ILI9341 communication signals, registered outputs (8080 I interface)
                csx             : out std_logic;
                dcx             : out std_logic;
                wrx             : out std_logic;
                data            : out std_logic_vector(15 downto 0);

                -- LT24 global signals, registered outputs
                LCD_on          : out std_logic;
                LCD_resn        : out std_logic
        );
end entity LT24_interface;
```

As in the DMA, here we used a three-process template. Respectively: *state_controller, register_controller, state_regs*. The unit is composed of a unique state machine, which also includes the Avalon Slave logic. The implementation follows the FSM proposed in the conceptual design straightforward. Other comments and explanations are included in the source code.

5. Creation of the IP in Quartus

We used Qsys to create the LCD_controller IP as a new component. In the wizard we specified the path of the IP vhdl files (“FIFO.vhd”, “Master_Dma.vhd”, “LT24_interface.vhd”, “LCD_controller.vhd”) and the structural vhdl as the top level. In the last section of the wizard, we grouped the signals and we created conduits for the external signals to the LCD.



After its creation, we imported the component into the Qsys system and we connected it to the other units.

6. Software

We used *Nios II Software Build Tools for Eclipse* to develop a demo application (*demo.c)* and a software package with helper macros and functions (*lcd_controller_utils.h*). The BSP implemented is the one automatically generated by the tool when creating a new project with the wizard.

6.1. C Code

6.1.1. lcd_controller_utils.h

This header file contains macros to easily access the registers of the LCD controller. Accesses inside the macros are based on the functions *IOWR_32DIRECT* and *IORD_32DIRECT* from the library *io.h*, which directly call an hardware routine that skips the cache. When accessing an I/O hardware unit as our LCD_controller, it is strongly recommended to bypass the cache, in order to avoid unwanted delays or desynchronization. Here we highlight some details that could be unclear reading the code:

- RESET_DMA clears the FIFO and restore default values of the DMA registers. It has to be called each time the LCD or the DMA stops loading before finishing the current frame (for example when a configuration command is sent to LCD while it is loading data). It must be also called before the first start of the DMA.
- WR / WRC (continuous): the writing functions of the LT24 interface can be continuous or not. A command/data is continuous when it expects additional following data to be sent. Usually, when configuring a register of the LCD, all the writings should be continuous except for the last one. We point out that ignoring this difference causes no troubles. If only WR functions are called, the ILI9341 pauses between each part of a register configuration, producing no side effect or delay.

The second part of the header file contains declarations of C functions used in the demo. The source file “LCD_controller_utils.c” contains functions implementation and explanation. Here we underline few aspects:

- Two HAL API[5] functions are used inside the functions:
  - *usleep(us)* imported by *<unistd.h>* allows to use low level resources (system clock) to wait the desired number of microseconds.
  - *alt_dcache_flush_all()* imported by *<sys/alt_cache.h>* is used in the function *load_image(int*, char*, int)* to make sure that all the pixels of an image have been correctly written in the DRAM and no data is kept inside the cache.
- The LCD initialization template provided in the course slides was modified to allow the landscape aspect ratio, since the default resolution is 240x320 and not 320x240, as the camera group required.

[5] Hardware Abstraction Layer Application Program Interface. See ***[ref.1]*** for details.

```
LCD_WRC_REG(0x0036); // Memory Access Control
LCD_WR_DATA(0x0028); // Invert XY axes to represent 320x240 images,
                        BGR order
```

6.1.2. demo.c

This file contains the main function of the demo, which is called when the program starts. *BURST_LENGTH* and *FRAME_PIXEL* must be defined by the user as constants, while the other quantities are extracted automatically. BURST_LENGTH is the burst number already discussed before, while FRAME_PIXEL is the number of pixels in the display and it is usually set to 76800.

The demo follows the following flow:

- 8 images extracted by a GIF animation are written to the DRAM in consecutive memory slots.
- The DMA and the LT24_interface are initialized and configured properly.
- The frames are loaded and displayed by the LCD in a loop, in order to create the effect of a video (1 frame each 100ms).

6.2. Demo

To run the demo, *readme* files can be referred. At first, follow the instructions in the *hw* folder to compile the system and program the device, then follow the *readme* in the *sw/nios/application* folder to build and run the demo software.

7. References

1. HAL API Reference, Nios II Software Developer's Handbook
2. Avalon® Interface Specifications
3. SCFIFO and DCFIFO IP Cores User Guide
4. ILI9341
5. LT24 User Manual
6. LT24 Schematic (LT24_v.1.0.2_SystemCD)