CS473 Embedded Systems – Lab 4

LCD & Camera mini project

Antoine Albertelli (205323, LCD)
Atri Bhattacharyya (269369, Camera)

This report presents the implementation of an embedded system acquiring pictures on a TRDM-D5M camera module and send them to an LT24 LCD module. The pictures are stored on the HPS external RAM. They are copied from the camera and to the screen without CPU intervention using Direct Memory Access (DMA) techniques.

Contents

Hardware architecture **2**
- Full system architecture . . . 2
  - Data format . . . 2
- LT24 (LCD) subsystem . . . 3
  - Changes from initial design . . . 3
- TRDM-D5M (Camera) subsystem . . . 3
  - Camera controller . . . 3
  - State machine . . . 5
  - Top level connections . . . 7
  - Timing diagrams . . . 7
  - Changes from initial design . . . 9

Programmer's manual **9**
- LCD module . . . 9
  - Software API . . . 9
  - Register map . . . 10
  - Usage . . . 11
- Camera module . . . 14
  - Software API . . . 14
  - Register map . . . 14
  - Usage . . . 15
  - Test setup . . . 15

Hardware architecture

Full system architecture

The full system (figure 1) contains a Nios processor (used to configure the peripherals and start / stop transfers), our custom LCD screen controller and our custom camera module controller. It also contains two memories: One small on chip RAM will hold the code for the Nios and one bigger external RAM will be used to contain the pictures transferred from camera to LCD. The external RAM will be used through the HPS DDR controller.

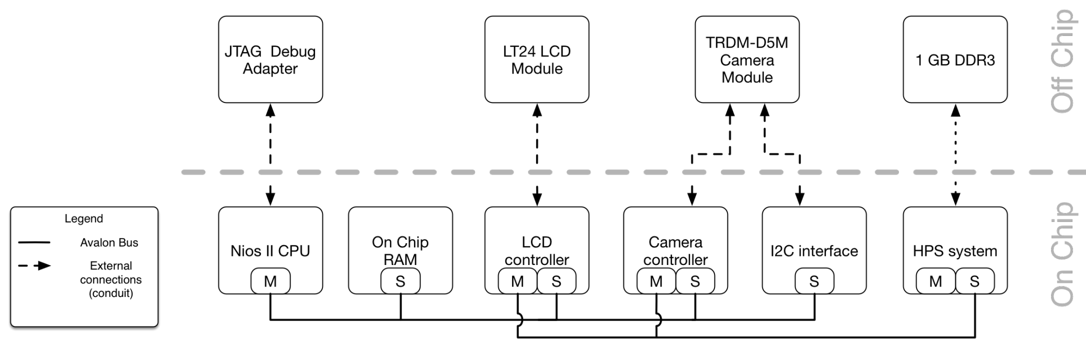

Figure 1: System architecture. The Hard Processor System (HPS) is only used as a gateway to the external RAM.

Data format

The display data are formated in the “R5G6B5” format: each pixel is made of 16 bits: 5 for red, 6 for green and 5 for blue. This is the native format for the LCD screen, meaning less conversion has to be done.

Figure 2: Pixel format used through the system.

Each frame is treated as a 320row x 240column array of 2bytes each. The pixels are stored row-majorwise. Hence, each frame requires 150KiB of memory.

LT24 (LCD) subsystem

The LCD controller architecture (figure 3) is made of several interconnected parts:

1. The LCD interface adapts the timing and signals coming from other parts of the system to be compatible with the LCD interface.
2. The Avalon slave interface exposes some registers used to configure the system from the NIOS2 processor. It can generate an interrupt on the end of frame if requested.
3. The Avalon master interface is used to fetch pixel data from the memory without the use of the NIOS2 processor. It makes use of the burst read functionality to increase throughput.
4. In order to use the burst read, a buffer must be placed between the LCD interface and the master interface. This is handled by a FIFO memory generated using Quartus' MegaWizard. The "almost full" signal is used to tell the master interface that there is not enough room to store a burst yet.

Changes from initial design

No significant hardware architecture changes were needed from initial design. The LCD's read enable pin is now driven instead of left open. This is required by the LCD but was left out of the design document. The FIFO control signals are inverted between Quartus' MegaWizard and my controller so this was changed as well.

TRDM-D5M (Camera) subsystem

The architecture for capturing frames from the camera consist of an I2C module for setting and reading the registers in the TRDM camera. A PLL is used to provide XCLKIN at a frequency of 25MHz. This is used by the camera for it's internal clock and ultimately for the PIXCLK signal that is sent to the camera controller. Finally, there is the camera controller that is used for configuration of the DMA module, recieving the frame from the camera and transferring that frame to memory. The internals of the camera controller are discussed in the next section.

Camera controller

The camera controller has 3 main components:

- Avalon slave module
- Avalon master module
- Camera interface module

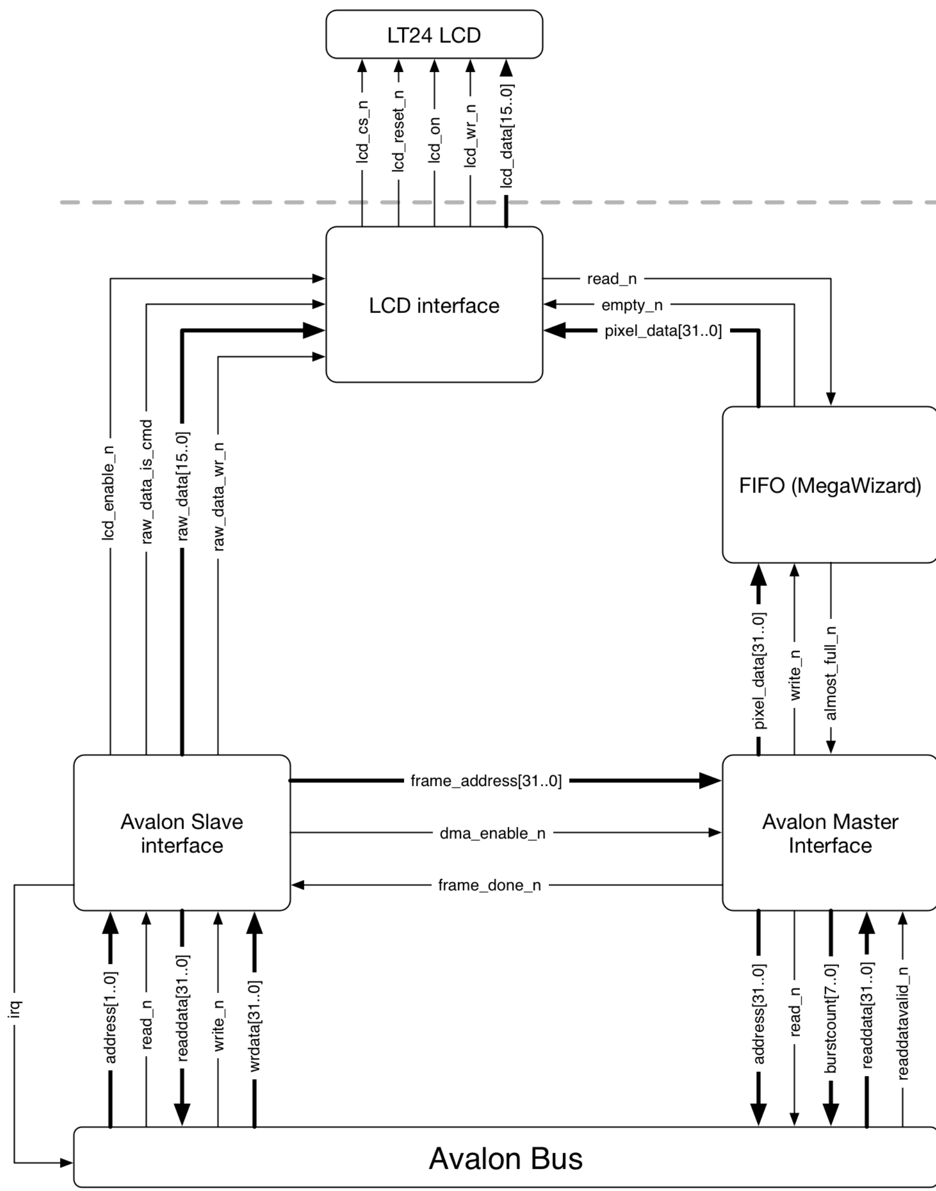

Figure 3: Architecture of the LCD controller

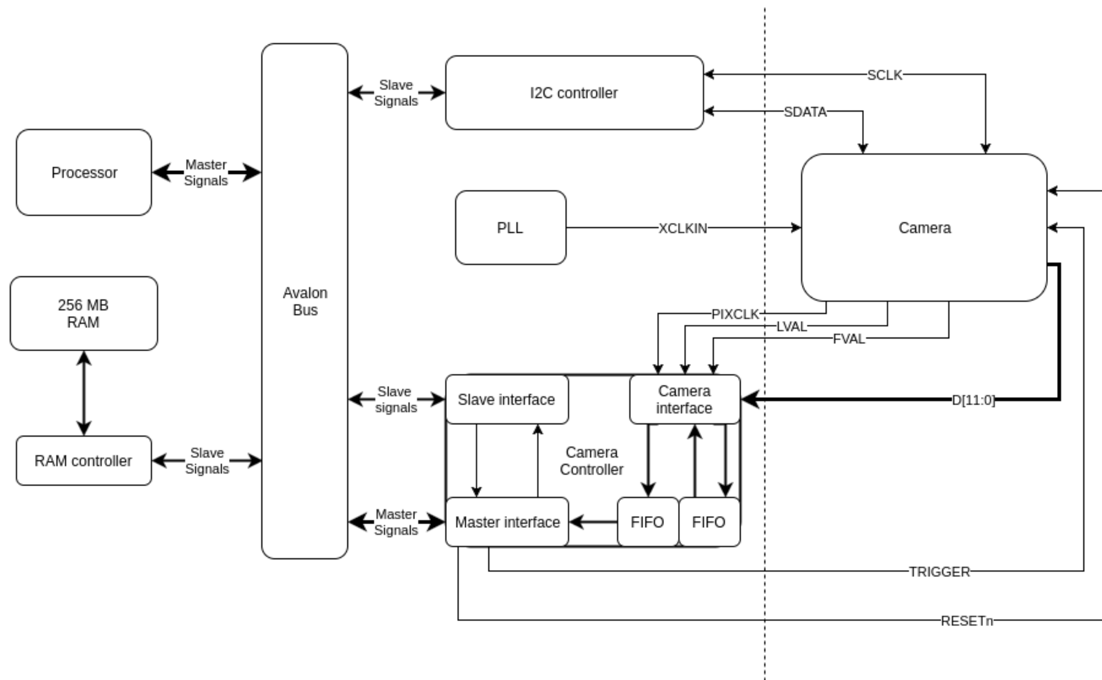

Figure 4: Camera subsystem architecture

Besides these, there are a couple of FIFO's, an odd-row FIFO which is used by the camera interface to store every alternate row and a FIFO between the master and interface modules that run on different clocks.

The avalon master module writes 16-bit half-words to memory one at a time. This is to accomodate the height-majorwise storage of the 2D pixel array in memory, and to enable the LCD DMA module to burst-read from the RAM. This means that successive pixels recieved from the camera are not adjacent in the memory.

State machine

The avalon master and slave modules have state machines as shown in the following diagrams.

In the WAITING state, the slave module allows the buffer address to be set and interrupts to be enabled or disabled. On receiving a write that triggers the camera, it transistions to the CLICKING state and sends the Start Frame signal to the master. Here, it waits for acknowledgement from the master that it has finished writing the frame to memory using DMA. Depending on whether interrupts are enabled or disabled, it transisions to the state INTERRUPT or WAITING respectively. In the INTERRUPT state, the interrupt is asserted. This allows the processor to know the end of frame transfer. The processor writes Reg0x10[1] = 1 to de-assert the interrupt. Then it goes back to the waiting state.

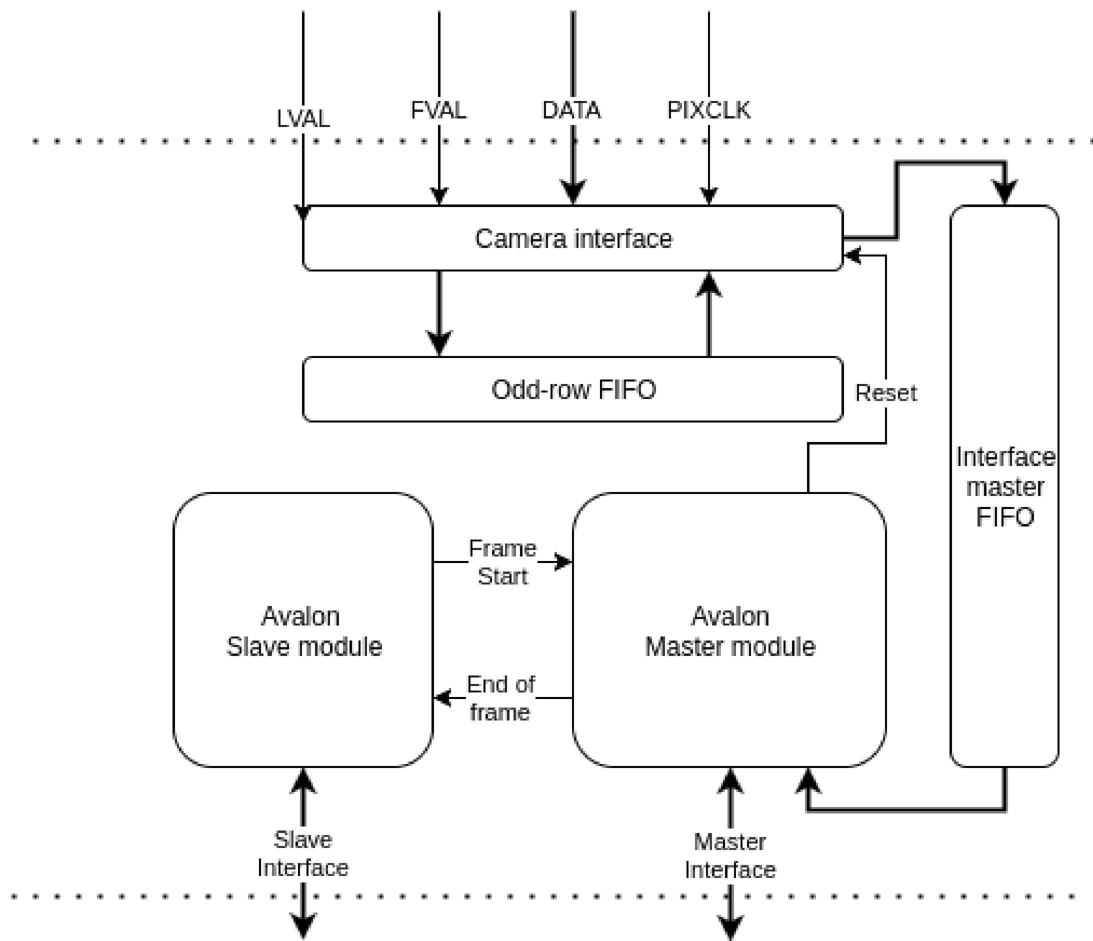

Figure 5: Camera controller architecture

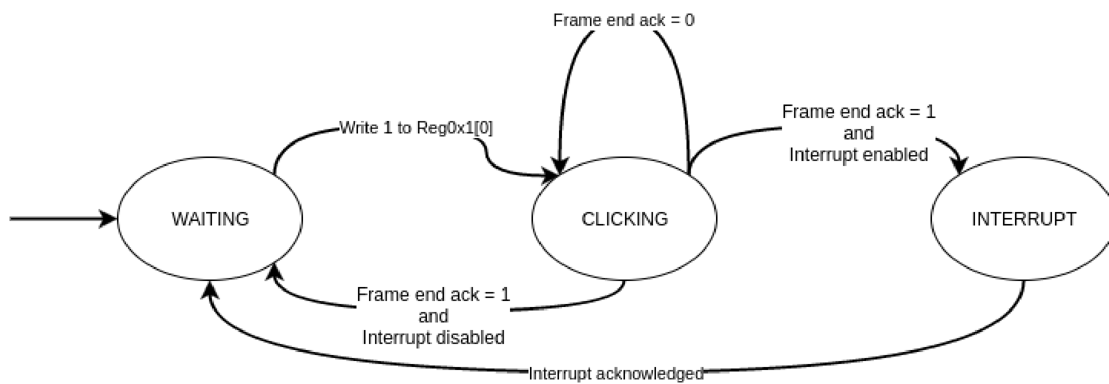

Figure 6: Slave state machine

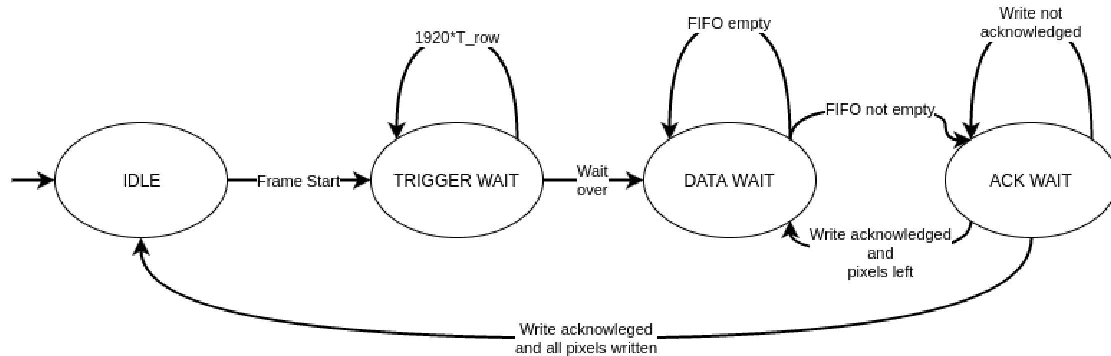

Figure 7: Master state machine

The Avalon master module starts in the IDLE state. Here, the interface module is held in reset. On recieving the Start Frame signal from the slave module, it transistions to a period where the camera is TRIGGERED. The camera interface is removed from reset and the camera then transistions to a DATA WAIT state. In this state, it waits for data coming from the camera interface module via the FIFO. When data is available, it initiates a write on the Avalon bus and moves to ACK WAIT state. In this state, it waits until the write is acknowleged using the wait_request signal. Then, depending on whether all pixels have been written, it transistions back to the IDLE or the DATA WAIT states.

Top level connections

Camera Controller Port	FPGA Pin
PIXCLK	GPIO_1_D5M_PIXCLK
DATA	GPIO_1_D5M_D
XCLKIN	pll_25mhz
RESETn	GPIO_1_D5M_RESET_N
TRIGGER	GPIO_1_D5M_TRIGGER
STROBE	unused
LVAL	GPIO_1_D5M_LVAL
FVAL	GPIO_1_D5M_FVAL
SDATA	GPIO_1_D5M_SDATA
SCLK	GPIO_1_D5M_SCLK

Timing diagrams

The following are simulations showing specific signals at different stages of a click. The clock is shown in yellow. Signals from the slave module are in indigo and those for the master module are in green.

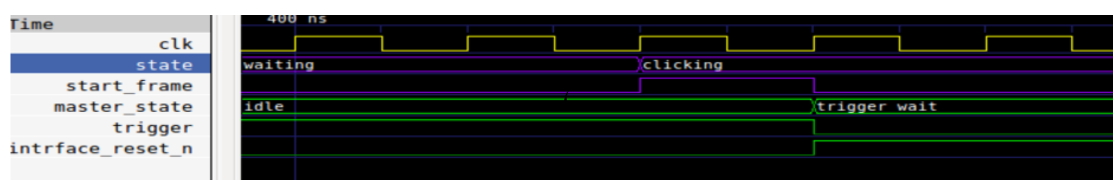

Figure 8: Triggering and associated state changes

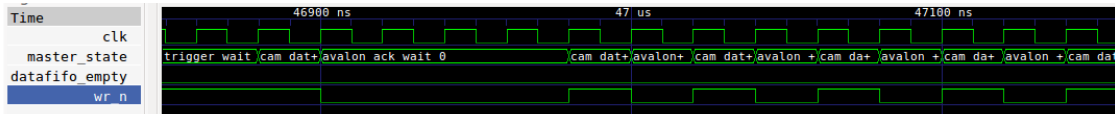

Figure 9: Beginning of data acquisition and DMA

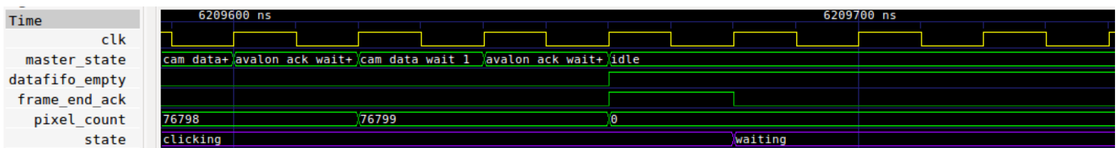

Figure 10: Signals at the end of a frame

Changes from initial design

The major changes from the design in Lab 3 to the final implementation are as follows:

- The startup procedure for the camera has been changed. This is mainly to overcome incorrect configurations caused by ambiguities or misunderstanding of the TRDB documentation. The PLL setup for the PIXCLK was abandoned as the camera is ultimately running at a much lower frequency than the master module and the FIFO internal to the camera is unnecessary.
- A FIFO stores the odd row pixels instead of an array of `std_logic_vector` after discussion with the TA. An array of that size would not fit inside the FPGA A FIFO, however, does fit and introduces a negligible amount of complexity to the design of the camera interface.
- A Dual-clock FIFO was introduced between the interface and master modules. This is necessary as the two modules run on unsynchronized clocks.

Programmer's manual

LCD module

Software API

API function	Description
`void lcd_reset(void *lcd_base)`	Resets the LCD controller
`void lcd_configure(void *lcd_base)`	Configures the LCD screen using the sequence described below.
`void lcd_enable(void *lcd_base)`	Turns on the screen.
`void lcd_interrutp_enable(void *lcd_base)`	Enables the interrupt for the LCD screen.
`void lcd_interrupt_disable(void *lcd_base)`	Disables the interrupt for the LCD screen.
`void lcd_interrupt_clear(void *lcd_base)`	Resets the interrupt pending flag. Must be called before returning from interrupt handler.
`void lcd_send_command(void *lcd_base, uint16_t cmd)`	Sends a command to the LCD.
`void lcd_send_data(void *lcd_base, uint16_t data)`	Sends raw data to the LCD.
`void lcd_send_frame(void *lcd_base, void *frame)`	Sends a frame via DMA to the LCD.
`void lcd_dma_is_busy(void *lcd_base)`	Returns 1 if a DMA transfer is in progress, 0 otherwise.

Register map

Table 3: LCD controller register map. All offsets are in bytes as seen from an Avalon Master.

Register	Offset	Description
`FRAMEADR`	0x00	32 bit base address of the display data.
`CTRL`	0x04	LCD interface control and status register.
`LCDCMD`	0x08	Raw data to be sent to the LCD as command.
`LCDDATA`	0x0C	Raw data to be sent to the LCD as data.

FRAMEADR register

This register holds the LCD frame data base address. The address is stored as a 32 bit Avalon byte address. When the `DMASTART` bit in the `CTRL` register is written, the LCD controller will start reading from this address.

This register is can be read and write.

CTRL register

Table 4: LCD control register organization

Field	Bit	Description
`LCDRESET`	5	Controls the reset line for the LCD. Writing a “1” here enables the reset.
`LCDON`	4	Controls wether the LCD is turned on or not.
`DMABUSY`	3	Indicates that a frame transfer is in progress.
`DMASTART`	2	Writings a “1” starts a DMA transfer to LCD. It is always immediately reset to zero.
`IE`	1	End-of-frame interrupt enable.
`IP`	0	End-of-frame interrupt pending. Writing zero clears interrupt flag.

LCDCMD register

This register is used to send raw commands to the LCD, e.g. during setup. Data written to this register will be sent to the LCD and marked as a command. Only the 16 bits of data are sent, the 16 most significant bits are ignored. This register is write only.

Note: Write to this register result in an Avalon wait state long enough for the LCD. This means that no delay is required between consecutives write to the `LCDCMD` register.

LCDDATA register

This register is used to send raw data to the LCD after a write to `LCDCMD`. Data written to this register will be sent to the LCD and marked as data. Only the 16 bits of data are sent, the 16 most significant bits are ignored. This register is write only.

Note: Writes to this register result in an Avalon wait state long enough for the LCD. This means that no delay is required between consecutives write to the `LCDDATA` register.

Usage

Before using the LCD interface in DMA mode, the LCD screen itself must be configured. In order to do so an interface to send configuration data to the LCD is exposed via the `LCDCMD` and `LCDDATA` registers. The LCD protocol made of commands followed by zero or more data.

In order to prevent invalid LCD state, the screen must first be reset by writing a '1' to the `LCDRESET` bit of the `CTRL` register, waiting at least 100 ms then writing back a '0'.

Once all configuration data has been written (table 5), the screen can be enabled by writing a '1' to the `LCDON` bit of the `CTRL` register.

Before transmitting a frame, the data address must be written to the `FRAMEADR` register. The transfer can then be started by writing a '1' to the `DMASTART` bit in the CTRL register.

If the `IE` bit of the `CTRL` register is set to '1' an interrupt is generated once a frame has been fully transmitted. The interrupt handler should write a '0' to the `IP` bit of the `CTRL` register in order to clear the interrupt flag. The interrupt handler can start the transfer of a new frame by writing a '1' to the `DMASTART` bit.

Table 5: LCD initialization sequence commands. See ILI9341 datasheet for details.

Command	Data	Description
0x0011		Leave sleep mode
0x00CF		Power control B
	0x0000	
	0x0081	
	0x00C0	
0x00ED		Power on control sequence
	0x0064	
	0x0003	
	0x0012	
	0x0081	
0x00E8		Driver timing control A
	0x0001	

Command	Data	Description
	0x0798	
0x00CB		Power control A
	0x0039	
	0x002C	
	0x0000	
	0x0034	
	0x0002	
0x00F7		Pump ratio control
	0x0020	
0x00EA		Driver timing control B
	0x0000	
	0x0000	
0x00B1		Frame control (normal mode)
	0x0000	
	0x001B	
0x00B6		Display function control
	0x000A	
	0x00A2	
0x00C0		Power control 1
	0x0005	
0x00C1		Power control 2
	0x0011	
0x00C5		VCM control 1
	0x0045	
	0x0045	
0x00C7		VCM control 2
	0x00A2	
0x0036		Memory access control
	0x0008	BGR order
0x00F2		Enable 3Gamma
	0x0000	3Gamma disable
0x0026		Gamma set
	0x0001	
0x00E0		Gamma correction curve (positive)
	0x000F	
	0x0026	
	0x0024	
	0x000B	
	0x000E	
	0x0008	

Command	Data	Description
	0x004B	
	0x00A8	
	0x003B	
	0x000A	
	0x0014	
	0x0006	
	0x0010	
	0x0009	
	0x0000	
0x00E1		Negative Gamma Correction, Set Gamma
	0x0000	
	0x001C	
	0x0020	
	0x0004	
	0x0010	
	0x0008	
	0x0034	
	0x0047	
	0x0044	
	0x0005	
	0x000B	
	0x0009	
	0x002F	
	0x0036	
	0x000F	
0x002A		Column Address Set
	0x0000	
	0x0000	
	0x0000	
	0x00EF	
0x002B		Page Address Set
	0x0000	
	0x0000	
	0x0001	
	0x003F	
0x003A		Pixel format
	0x0055	R5G6B5
0x00F6		Interface Control
	0x0001	
	0x0030	
	0x0000	

Command	Data	Description
0x0029		display on

Camera module

Software API

API function	Description
`void camera_setup()`	Setup camera for normal operations
`void click(void *address)`	Starts taking a picture, storing the result at the provided address.
`void setup_interrupt(void (*f)(int))`	Setup interrupts and enable them. f is a function that is called when the interrupt happens. f is called with the address of the frame buffer as the argument.
`void enable_interrupt()`	Enable interrupts.
`void disable_interrupt()`	Disable interrupts.
`void set_register(int register_no, uint16_t value)`	Set a camera register.
`void test_setup()`	Setup the camera to output the test pattern.
`void print_camera_setup()`	Print the camera registers for debuging.

Register map

Table 7: Register map for our camera controller. All offsets are in bytes, as seen from an Avalon master.

Register	Offset	Description
`BUFFADDR`	0x0	Address of the frame buffer. Writes are ignored when transfer is active
`TRIGGER`	0x04	Bit 0: Assert this bit to start a new transfer if idle. Write ignored if active. It is deasserted on completion of frame transfer
`CNTRL`	0x08	Bit 0: Enable interrupts on completion if set. Bit 1: Set on interrupt. Clear to deassert interrupt.

Usage

The provided setup function configures the internal registers of the camera using the provided I2C IP core. Here is the startup procedure:

Startup procedure (using I2C commands):

1. Set the Reset register : `Reg0x0D[0] = 1`
2. Wait 1ms
3. Reset the Reset register: : `Reg0x0D[0] = 0`
4. Wait 1ms
5. Set register values as given below.
6. Delay 1ms
7. Set `Reg0x1E = 0x4106` to use snapshit mode.
8. Delay 1ms

Table 8: Camera register values

Register	Value	Description
3	0x077F	1920 pixels, binned by 4, later subsampled by 2 = 240 pixels
4	0x09FF	2560 pixels, binned by 4, later subsampled by 2 = 320 pixels
9	0x077F	Shutter width same as width
10	0x0001	Divide XCLKIN to get a PIXCLK of 12.5MHz. Conceptually, other frequencies should not affect the correct functioning of the controller.
34	0x0033	Set Row bin and Row skip to 3
35	0x0033	Set Column bin and skip to 3
160	0x0000	Disable the test pattern if active

Test setup

Setting the test pattern from the camera:

Register number	Value	Explaination
160	0x0039	Sets monochromatic vertical bars as test pattern and activates testing
164	0x003F	Set width of bars as divisor of frame width
161	0x. . . .	Set arbitrary value for pixel values in even bars
163	0x. . . .	Set arbitrary value for pixel values in odd bars

Performance

The whole setup was able to transfer images between camera and LCD at a rate of about 2 frames per second. We believe this is mostly limited by the camera exposure time, as the DMA transfers are really fast. Therefore it should be possible to decrease the exposure time and use the lower bits of each pixel instead of the higher ones to gain some sensitivity. This could also improve color response, as our images were quite washed out.

Another performance issue with the current design is that LCD read bursts are extremely long (120 32-bit words). This means that the Avalon slave serving the request (the external RAM in our case) cannot be accessed for long period of time. If the memory contains code for the CPU, IRQ will be delayed, which might not always be acceptable.