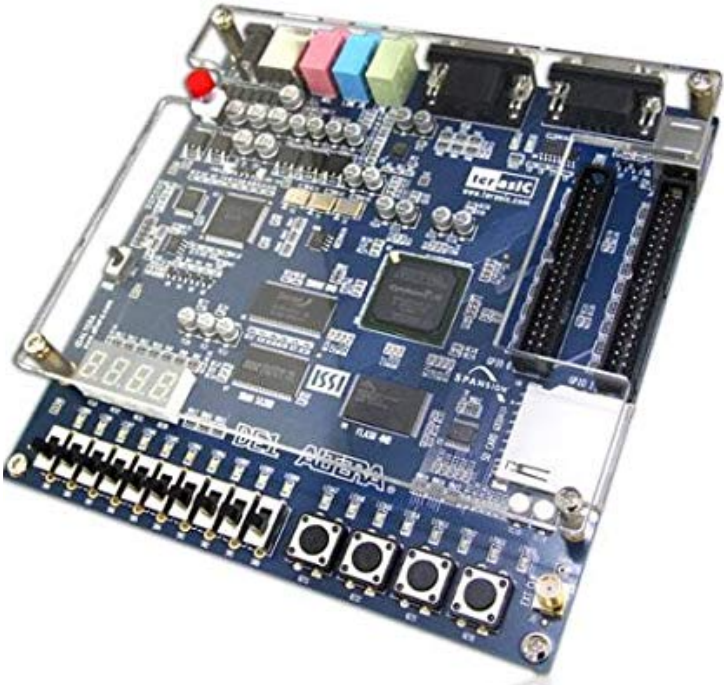
CSEE 4840 EMBEDDED SYSTEM DESIGN

SECURITY CAMERA

Final Project Report

Columbia University



Team Members:

Noe Silva - **ns3567**

Elliot Flrores Portillo - **esf2150**

Mir Naveen Alam - **ma4310**

Carlos Eduardo Cruz - **cec2274**

Shifeng Zhang - **sz3104**

Table of Contents

1. Overview

The core goal of this project is to implement a Security Video Camera capable of live-streaming video in VGA format. Complementary, a PIR sensor executes motion detection that triggers a signal to capture video frames and stores them on an SD Card in bitstream format. The target device is a cyclone V FPGA embedded on the De1-Soc Development Kit board that also contains a dual-core Arm Cortex A9 processor on it. Furthermore, five peripherals are connected to the board to achieve our purpose; Video Camera, Motion Sensor, Sdram, Sd-card, and VGA Monitor.

2. System Architecture

The figure below shows a top-level block diagram with the main components and their dependencies.

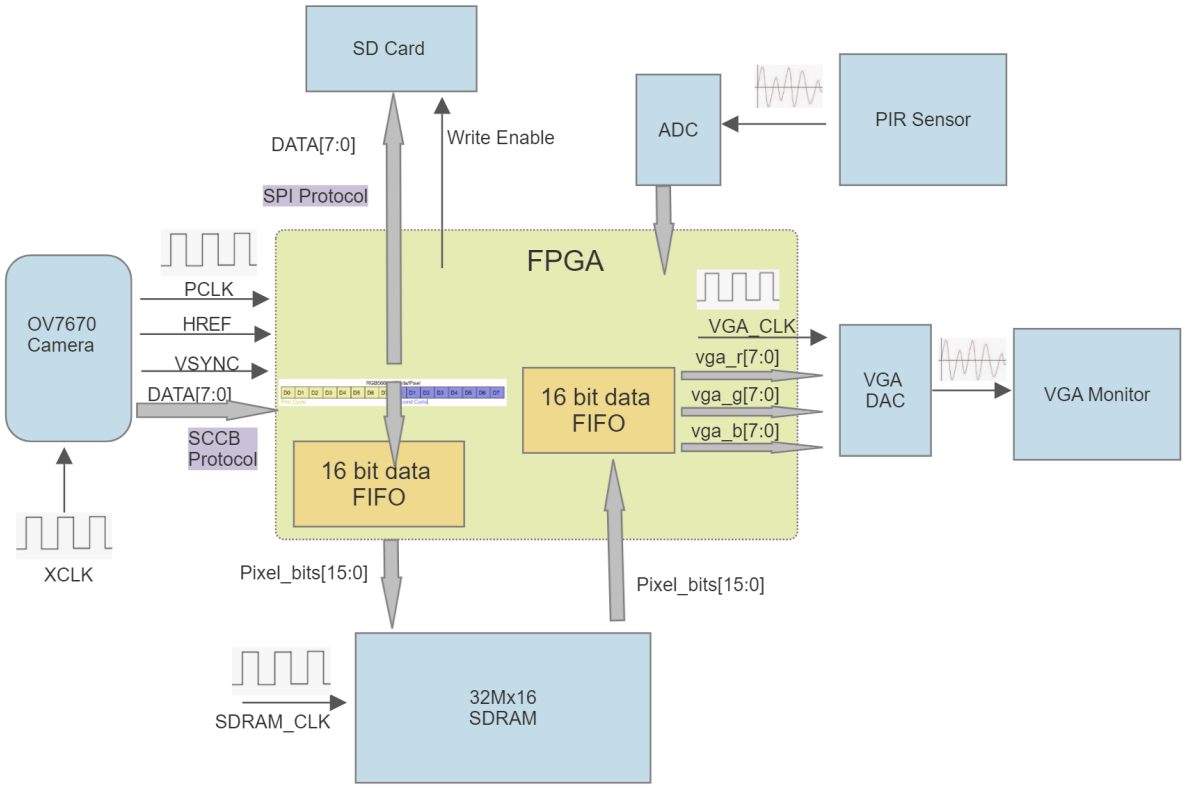


Figure 1. System Architecture Diagram

2.1. Pixel's bits Processing

A 24MHz clock is driving the OV7670 Camera which outputs 8 bits in parallel on each falling edge of the clock. The camera talks to the FPGA through the SCCB protocol. The FPGA captures 8 bits in the first cycle, then waits for the second cycle and captures 8 more bits. A total of 16 bits (one pixel) are sent to the Asynchronous FIFO. The output of the FIFO is connected to the onboard SDRAM through an SDRAM driver, The SDRAM is similarly connected to a second FIFO. From the output of the second FIFO, bits are padded to the vga_outputs to resize them to 8 bits each and then normalized. Three bytes are sent to the VGA DAC to finally display the pixels at 25MHz frequency.

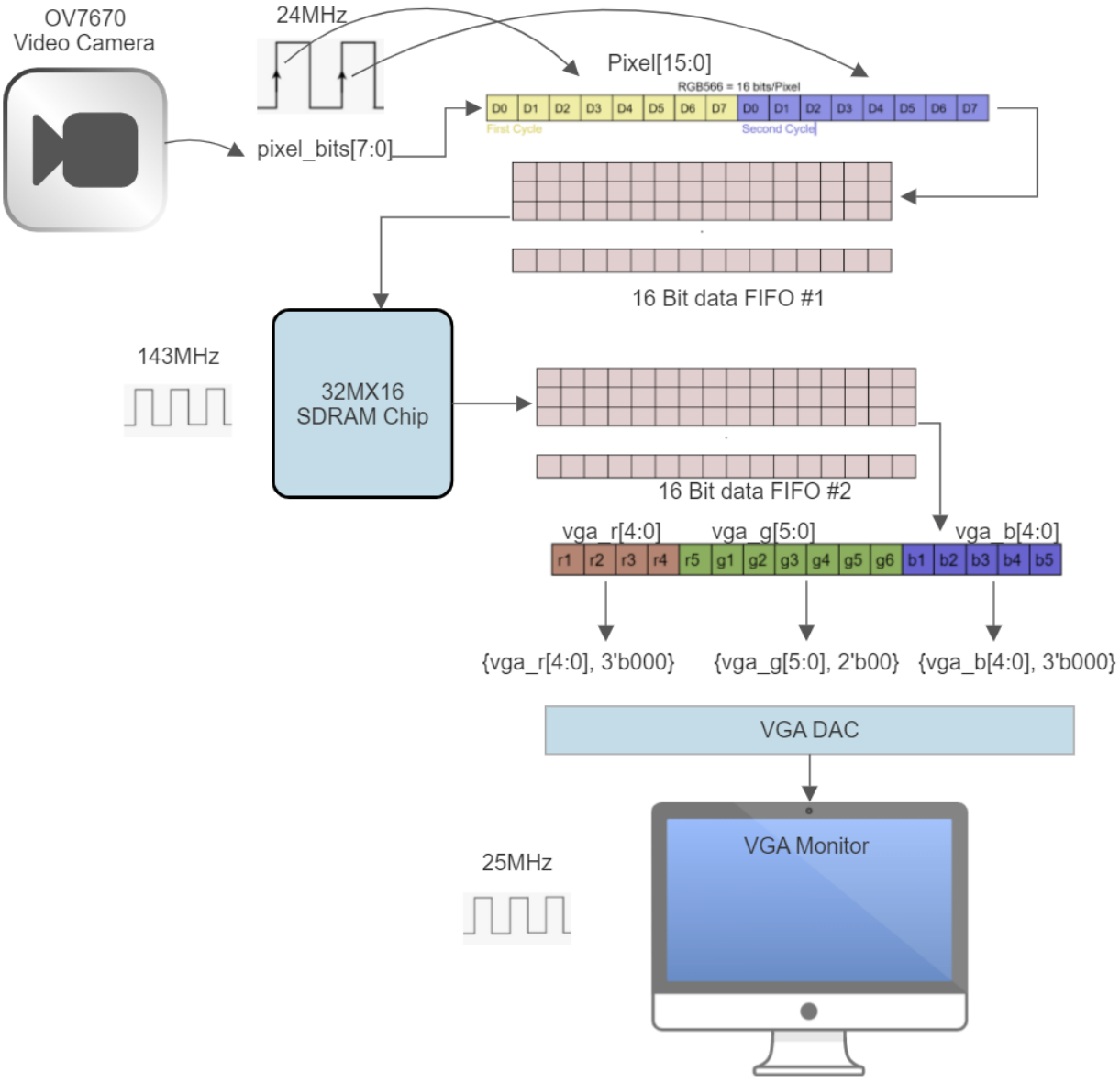


Figure 2. Pixel's bits processing

2.2. RTL Schematic

Top Level Schematic, connected to the five peripherals, VGA monitor, OV7670 video camera, and 64 MB SDRAM (32Mx16) chip, SD Card and PIR Sensor. We used three clock domains taking as reference the onboard 50MHz clock.

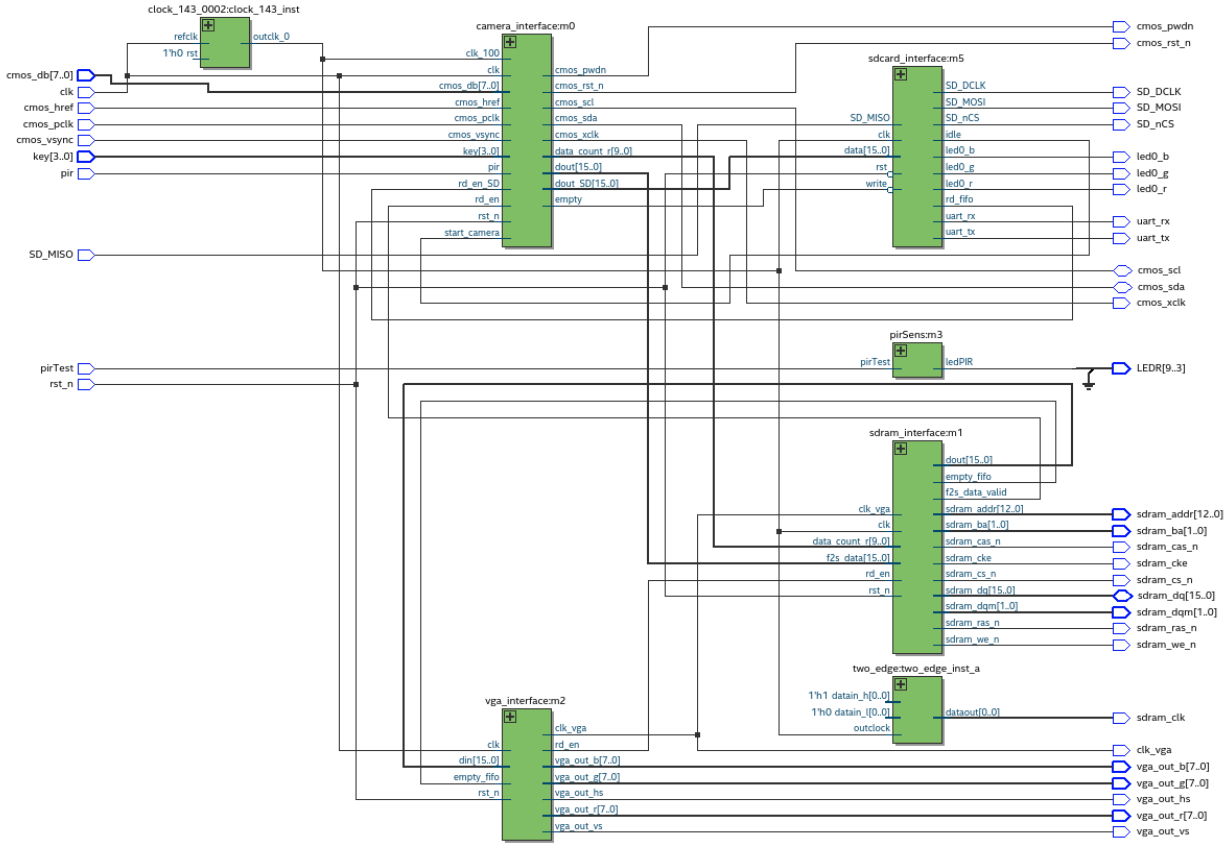


Figure 3. RTL Schematic

3.1 OV7670 Camera Module

The OV7670 Camera module has a resolution of 640x480 pixels. It's capable of displaying up to 30 frames per second. The OV7670 uses the Serial Camera Control Bus (SCCB) protocol to communicate with external hardware. There are two versions, one with 16 bits and the other with 18 pins. In this project, the 18-pin module was used and it's shown below.

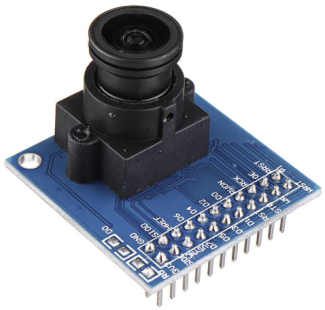


Figure 4. OV7670 Module

On the chart below there is a description of the camera's pins. The module is powered on with 3.3 V, it receives an input clock (XCLK) and produces an output pixel clock (PCLK). The falling edge of the pixel clock is used to output the parallel 7 bits. The maximum XCLK frequency is 25MHz, for this project, we used a clock frequency of 24MHz just to make sure we did not drive the camera to its limit.

Pin	Type	Description
VDD**	Supply	Power supply
GND	Supply	Ground level
SDIOC	Input	SCCB clock
SDIOD	Input/Output	SCCB data
VSYNC	Output	Vertical synchronization
HREF	Output	Horizontal synchronization
PCLK	Output	Pixel clock
XCLK	Input	System clock
D0-D7	Output	Video parallel output
RESET	Input	Reset (Active low)
PWDN	Input	Power down (Active high)

Chart 1. OV7670 Pins

All 18 of the camera's pins were connected to the GPIO expansion headers of the FPGA.

The first step before starting the pixel's bits transmission is to configure the camera mode operation by sending commands to set its registers. We set the camera to operate in the RGB565 format. Each pixel is represented by 5 bits for red, 6 bits for green, and 5 bits for blue. Due to each pixel being 16 bits(2 Bytes), two clock cycles are necessary to capture a single pixel.

3.2 SDRAM

The amount of memory necessary to store a frame is 16 bits X 480 X 680 = 4, 915, 200 / 8 = 614 KB. The on-chip memory provided by the DE1-SoC is 256KB (BRAM), therefore there is not enough memory to store a single frame. We used the 64 MB synchronous dynamic RAM (SDRAM) on the DE1-SoC board, which is organized as 32M x 16 bits, and used the BRAM as a pixel buffer.

Figure 6. Shows the connections between the FPGA and the 64 MB SDRAM chip. A 143MHz clock frequency was used to read/write data onto the SDRAM.

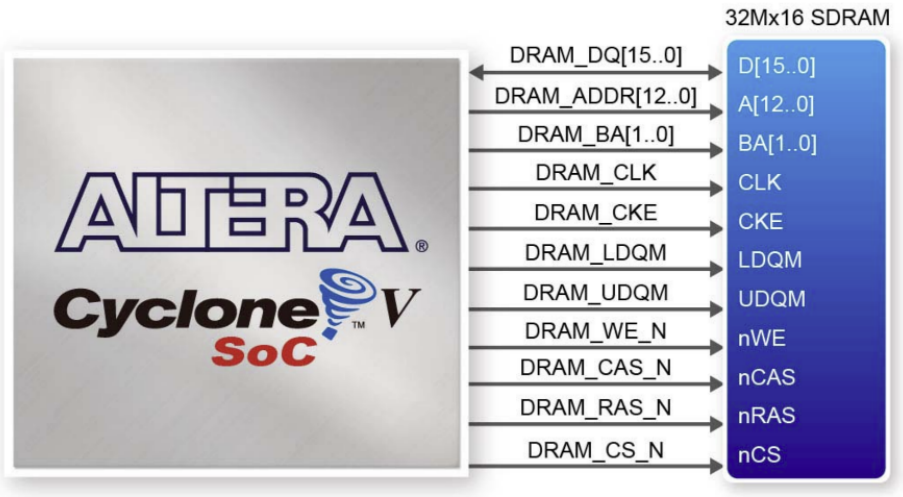


Figure 5. Connections between the SDRAM and the FPGA

3.3 VGA Monitor

As mentioned above, the images captured and stored in the SD Card will be displayed on a VGA (Video Graphics Array) monitor. The DE1-SoC board has a 15-pin D-SUB connector populated for VGA output. The VGA synchronization signals are generated directly from the Cyclone V SoC FPGA, and the Analog Devices ADV7123 triple 10-bit high-speed video DAC (only the higher 8-bits are used) transforms signals from digital to analog to represent three fundamental colors (red, green, and blue). The board can support up to 1280X1024 pixels resolution. For this project our pixel resolution is dictated by the video camera resolution; in this case 640X480 pixels.

Figure 6 shows the connections between the FPGA board and the VGA connector. Notice that a digital-to-analog converter is placed in between. In total 29 Pins of the FPGA are dedicated to VGA.

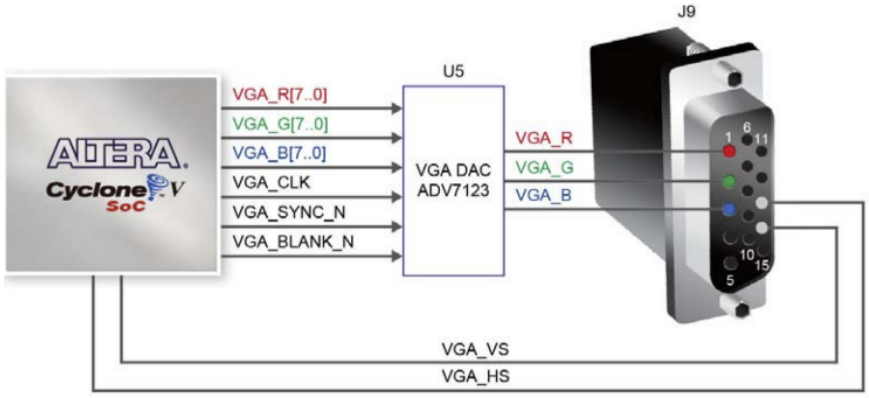


Figure 6. Connections between the DAC, VGA connector, and FPGA

3.4 SD Card

To store the images captured from the camera when motion is detected, an external SD card reader is used. To communicate with the SD card reader it uses the Serial Peripheral Interface (SPI) protocol, which will be discussed later in this report. The SD card has 6 pins for power (VCC), ground (GND), Master In Slave Out (MISO), Master Out Slave In (MOSI), Serial Clock (SCK), and Chip Select (CS) as shown below.

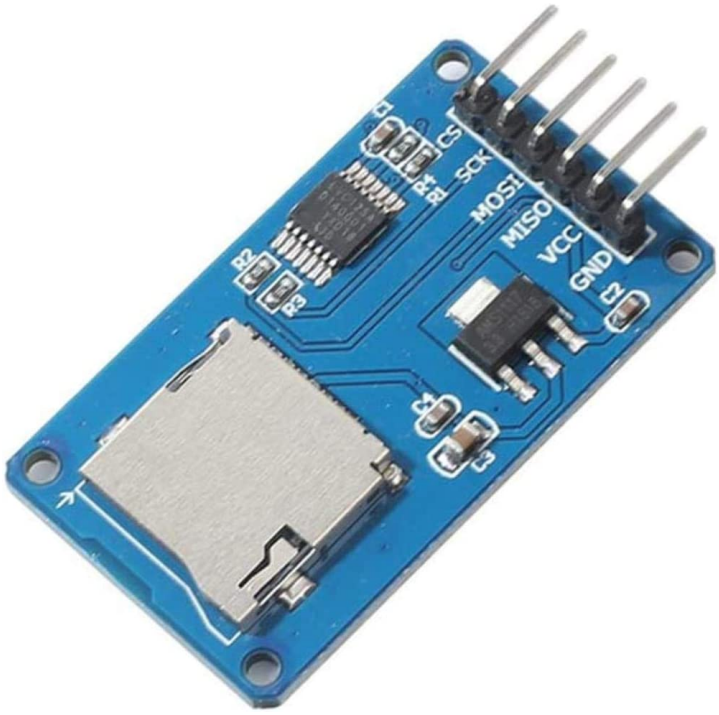


Figure 7. Image of the SD Card Reader

In the camera interface whenever the PIR sensor detects motion it writes one single frame to the asynchronous FIFO. It does this by connecting the empty signal of the FIFO to the write signal of the SD card interface. When the FIFO is not empty so it is full then write whatever is in the FIFO into the SD card. This should only be one frame at the time of motion detection. The SD card can be taken out of the reader and inserted into any computer to look at the various images captured.

3.5 HC-SR501 PIR sensor

The infrared sensor detects infrared light radiated from objects. It is a passive infrared sensor (PIR) that detects heat energy from objects. This type of sensor is widely used in alarm systems, often used as motion detectors. Due to the sensed data being analog, we need to convert it to digital. Also, the PIR sensor has a built-in noise immunity that helps to provide a smooth digital output pulse. It has an adjustable sensitivity where the range can be set from 3 to 7 meters. In fact, not only do the Fresnel lenses help to focus more light into the pyroelectric sensor but also help to increase the range. So the sensor detective can be more efficient. Similarly, the delay when the output goes high can be adjustable, which ranges from 1 second to 3 minutes. In addition, the sensor has two trigger modes where the first is a single-trigger mode and the second is a multiple-trigger mode. In the single

trigger mode, when motion is detected the output will go high and remain high depending on the delay setting. If motion continues within the delay, the sensor will not detect it (See figure [17]). In the multiple-trigger mode, the output will go high when motion is detected and will remain high depending on the delay setting. If motion is detected during the first or previous time delay, the output will be high for a new delay period (See figure [18]).

Since this sensor has many settings, it is suitable for our project. The idea is to configure one of the GPIO pins on the FPGA as input and connect the output of the sensor. Then, the power will be supplied through the VCC5 pin onboard.

Figure 8. HC-SR501 PIR sensor

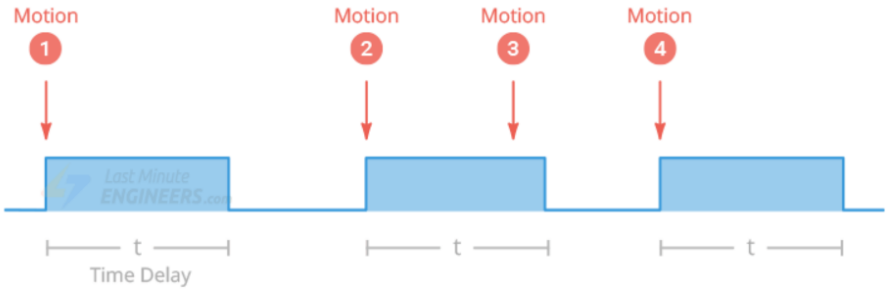


Figure 9. Single Trigger Mode Detection.

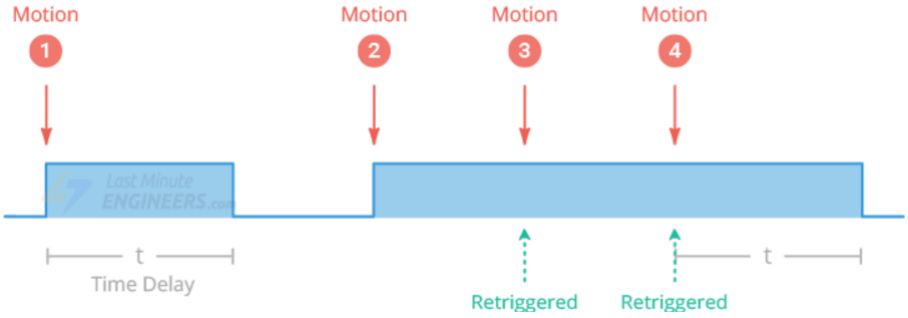


Figure 10. Multiple Trigger Mode Detection

4.1 SCCB Protocol

To communicate with the OV7670 camera module the Serial Camera Control Bus (SCCB) protocol is used, which is a subset of the I2C protocol. SCCB has two different styles: 3-wire and 2-wire variations. The 3-wire method is used to have multiple slaves controlled by one master and the 2-wire method is used for only one master and slave. This project will implement the 2-wire approach since there is only 1 camera being used.

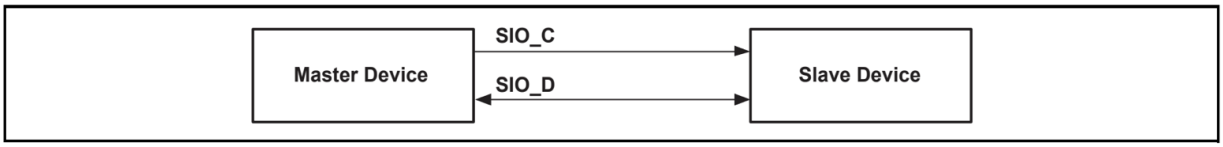


Figure 11. 2-Wire SCCB

The 2-wire SCCB protocol contains a clock signal SIO_C (Serial Input Output) and a data transmission signal SIO_D. Data on the SIO_D signal gets written based on the clock from the SIO_C signal.

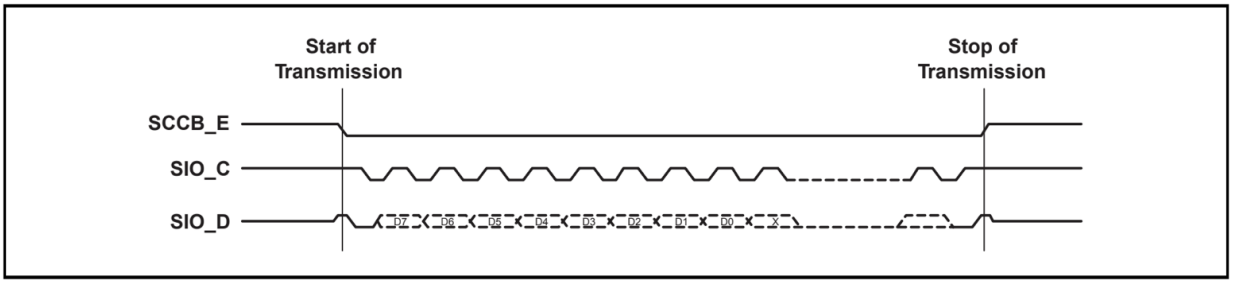


Figure 12. Waveforms for SCCB Protocol NOTE: This figure represents the 3-wire method.

Data is sent out in phases of 9 bits each, 8 for data and 1 Don't-Care bit depending on whether the transmission is a read or write. The purpose of the Don't-Care bit is to notify that the transmission is complete. The maximum number of phases a transmission can have is 3, one for ID Address, Sub-address, and Write Data.

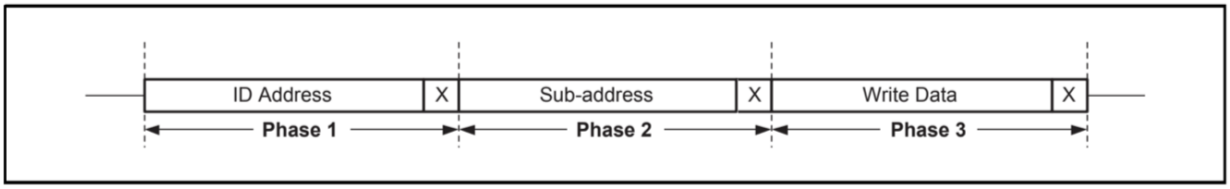


Figure 13. SCCB Data transmission

The ID Address identifies the slave to write and read data from, the Sub-addresses an address from the slave that contains the read data from the slave, and the Write Data is the data from the master to the slave.

4.2 SPI Protocol

The Serial Peripheral Interface (SPI) in this project is used mainly to communicate with the SD card. One of the reasons is that SPI uses less hardware and system resources compared to USB. The second reason is that SPI is supported by SD cards. Since SD cards have two modes of operation which are SD mode and SPI mode. Where the SD mode offers higher throughput compared to the SPI mode. The drawback of SD mode is that one has to sign a nondisclosure agreement and pay some royalties. Thus, we have to take some tradeoffs.

In Addition, the SPI protocol works in a master-slave fashion. Where the master is the controlling device (in this case the FPGA) and the slave takes instructions (in this case SD card). It is worth mentioning that the master can control different slaves. However, in this project, we only have to control one slave (Figure 14). The master output slave input (MOSI) line transfers data from the master to the slave. Usually, the data is sent from the master to the slave with the most significant bit(MSB) first. Inversely, the master input slave output (MISO) line transfers the data from the slave to the master. Typically, the data sent from the slave to the master starts with the least significant bit(LSB) first. The SCLK is the input clock signal for the slave. The CS is the chip select, this is in charge of selecting a slave. In the case of dealing with more than one slave, each slave will have a dedicated CS line(Figure 15). Then, the master will assert (active low) the correct slave device that it wants to communicate with. if the master is communicating with many slaves. When the communication is finished with a certain slave, the master will de-assert (logic high) the slave. In the case of communicating with a single slave, the CS can be active (logic 0) all the time(Figure 14).

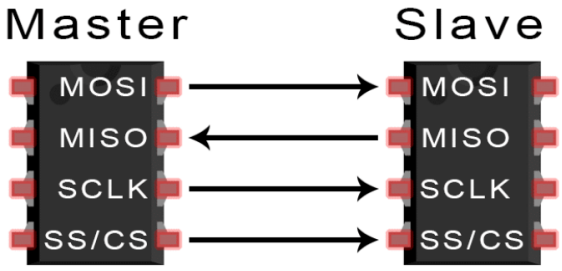


Figure 14. SPI protocol with a single slave

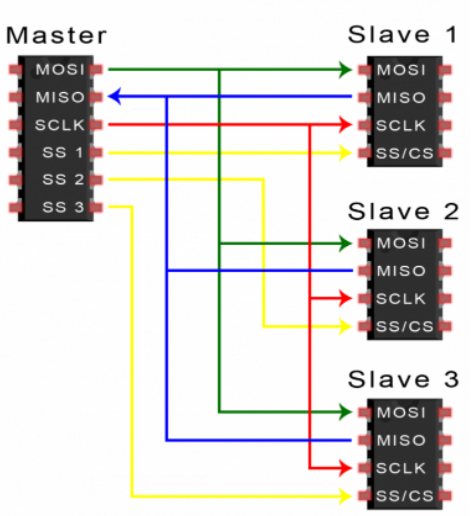


Figure 15. SPI protocol with multiple slaves

The SPI diver that we are implementing transfers and receives data at the positive edge of the clock. This is specified when the SPI mode is asserted to zero. In addition, this module has two frequency options. The reason for having two options is that the initialization of the SD card is performed at 400KHz. Once the initialization is done, writing data into the SD card is performed at 25MHz to maximize the throughput. An overview of the SPI module can be seen in Figure 16.

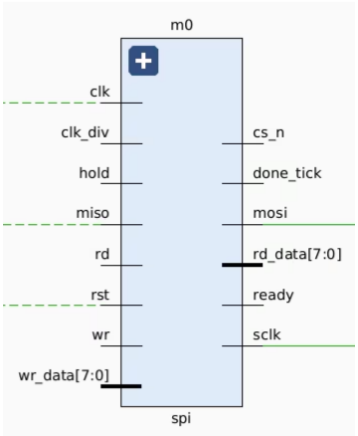


Figure 16. SPI module.

5.1 Insufficient on-Chip Memory

Initially, we wanted to use the on-chip memory to store the pixels but quickly realized that was a limitation due to the size of memory necessary to store a single frame in RGB565 format: 16 bits X 480 X 640 = 614.4KB. The on-chip RAM on the DE1-SoC is 256KB which is far less than what we needed. This forced us to add another peripheral and its driver; the On board 64MB SDRAM chip.

5.2 Several Clock Domains

We used three clock domains, one for the OV7670 video camera(24MHz), one for the VGA monitor(25MHz), and one more for the SDRAM(143MHz). We were stuck for a while because the VGA monitor was displaying a very distorted signal that looked like noise and this was because we were using the same clock(143MHz) for reading and writing to the SDRAM; A hold-setup timing violation was occurring. This issue was solved by creating a 180 degrees phase shift between the write clock and the read clock. For such a task, an Altera ALTDDIO IP was implemented.

6. Results and Improvements

The core component of the project was to first obtain live color video and then add peripherals to make the system smart. We successfully obtained live color video and motion detection. Unfortunately, time was a scarce resource and we were not able to successfully store frames into the SD card each time motion was detected. We created an sd card module and its driver(SPI) and connected it to the top module, but no binary file was written in the sd card. After troubleshooting, we think that we are probably not initializing the sd card correctly. Therefore, successfully saving frames into the sd card when motion is detected would be a further improvement.

7. References

https://community.element14.com/challenges-projects/design-challenges/summer-of-fpga/b/blog/posts/security-camera-1-project-proposal-629530496

https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/

http://embeddedprogrammer.blogspot.com/2012/07/hacking-ov7670-camera-module-sccb-cheat.html

http://web.mit.edu/6.111/www/f2016/tools/OV7670_2006.pdf

https://github.com/AngeloJacobo/FPGA_OV7670_Camera_Interface

DE1-SoC_User_manual.pdf

8. Appendix

```
`timescale 1ns / 1ps

  module top_module(
       input wire clk,rst_n,
       input wire[3:0] key, //key[1:0] for brightness control , key[3:2] for contrast control
       input wire pir,
       input wire pirTest,
       //input wire btn,
       output wire [9:3] LEDR,
       //camera pinouts
       input wire cmos_pclk,cmos_href,cmos_vsync,
       input wire[7:0] cmos_db,
       inout cmos_sda,cmos_scl,
       output wire cmos_rst_n, cmos_pwdn, cmos_xclk,
       //Debugging
       output wire led0_r, led0_g, led0_b,
       //controller to sdram
       output wire sdram_clk,
       output wire sdram_cke,
       output wire sdram_cs_n, sdram_ras_n, sdram_cas_n, sdram_we_n,
       output wire[12:0] sdram_addr,
       output wire[1:0] sdram_ba,
       output wire[1:0] sdram_dqm,
       inout[15:0] sdram_dq,
       //VGA output
       output wire clk_vga,
       output wire[7:0] vga_out_r,
       output wire[7:0] vga_out_g,
       output wire[7:0] vga_out_b,
       output wire vga_out_vs,vga_out_hs,
       //sd card

       input wire SD_MISO,
       output wire SD_MOSI,
       output wire SD_DCLK,SD_nCS,
       //UART for debugging
       output wire uart_rx,uart_tx
       //
       // output idle,
       //HOST interface
       //input write, //start writing to SD card
       //output reg rd_fifo, //read next data to be written
       //input[7:0] data //data to be written to SD catd

   );

        wire f2s_data_valid;
        wire[9:0] data_count_r;
        wire[15:0] dout, dout_SD;
        wire[15:0] din;
        wire clk_sdram;
        wire empty_fifo;
        //wire clk_vga;
        wire state;
```

```
      wire rd_en;
      wire empty, rd_en_cam;
      wire idle;

      //module instantiations
sdcard_interface m5
(
    .clk(clk_sdram),
    .rst(!rst_n),
    .led0_r(led0_r),
    .led0_g(led0_g),
    .led0_b(led0_b), //{red,green,blue} red if SDCARD initialization is stuck at CMD0, blue if stuck somewhere else, green if initialization complete
    .idle(idle),  //sdcard not busy
     //HOST interface
    .write({!empty}),
    .rd_fifo(rd_en_cam),
    .data(dout_SD[7:0]),
    //SPI pinouts
    .SD_MISO(SD_MISO),
    .SD_MOSI(SD_MOSI),
    .SD_DCLK(SD_DCLK),
    .SD_nCS(SD_nCS),
    //UART for debugging
    .uart_rx(uart_rx),
    .uart_tx(uart_tx)
);


    camera_interface m0 //control logic for retrieving data from camera, storing data to asyn_fifo, and  sending data to sdram
    (
            .clk(clk),
            .clk_100(clk_sdram),
            .rst_n(rst_n),
            .key(key),
            .empty(empty),
            .pir(pir),
            .start_camera(idle),
            //asyn_fifo IO
            .rd_en(f2s_data_valid),
            .rd_en_SD(rd_en_cam),
            .data_count_r(data_count_r),
            .dout(dout),
            .dout_SD(dout_SD),
            //camera pinouts
            .cmos_pclk(cmos_pclk),
            .cmos_href(cmos_href),
            .cmos_vsync(cmos_vsync),
            .cmos_db(cmos_db),
            .cmos_sda(cmos_sda),
            .cmos_scl(cmos_scl),
            .cmos_rst_n(cmos_rst_n),
            .cmos_pwdn(cmos_pwdn),
            .cmos_xclk(cmos_xclk),
            //Debugging
```

```
            .led(led)
);

    sdram_interface m1 //control logic for writing the pixel-data from camera to sdram and reading pixel-data from sdram to vga
    (
            .clk(clk_sdram),
            .rst_n(rst_n),
            //asyn_fifo IO
            .clk_vga(clk_vga),
            .rd_en(rd_en),
            .data_count_r(data_count_r),
            .f2s_data(dout),
            .f2s_data_valid(f2s_data_valid),
            .empty_fifo(empty_fifo),
            .dout(din),
            //controller to sdram
            .sdram_cke(sdram_cke),
            .sdram_cs_n(sdram_cs_n),
            .sdram_ras_n(sdram_ras_n),
            .sdram_cas_n(sdram_cas_n),
            .sdram_we_n(sdram_we_n),
            .sdram_addr(sdram_addr),
            .sdram_ba(sdram_ba),
            .sdram_dqm(sdram_dqm),
            .sdram_dq(sdram_dq)
);

    vga_interface m2 //control logic for retrieving data from sdram, storing data to asyn_fifo, and sending data to vga
    (
            .clk(clk),
            .rst_n(rst_n),
            //asyn_fifo IO
            .empty_fifo(empty_fifo),
            .din(din),
            .clk_vga(clk_vga),
            .rd_en(rd_en),
            //VGA output
            .vga_out_r(vga_out_r),
            .vga_out_g(vga_out_g),
            .vga_out_b(vga_out_b),
            .vga_out_vs(vga_out_vs),
            .vga_out_hs(vga_out_hs)
);

two_edge    two_edge_inst_a (
    .datain_h (1'b1),
    .datain_l (1'b0),
    .outclock (clk_sdram),
    .dataout (sdram_clk)
    );

 clock_143_0002 clock_143_inst (
            .refclk   (clk),   //  refclk.clk
            .rst      (rst),      //   reset.reset WHERE DOES RESET SIGNAL COME FROM?
```

```
                .outclk_0 (clk_sdram), // outclk0.clk
                .locked   ()           // (terminated)
        );

        pirSens m3 (
                .pirTest(pirTest),
                .ledPIR(LEDR[3])
        );

endmodule
```

```
`timescale 1ns / 1ps

 module camera_interface(
       input wire clk,clk_100,rst_n,
       input wire[3:0] key, //key[1:0] for brightness control , key[3:2] for contrast control
       input wire pir,
       input start_camera,
       //asyn_fifo IO
       input wire rd_en, rd_en_SD,
       output wire[9:0] data_count_r,
       output wire[15:0] dout,
       output wire[15:0] dout_SD,
       //camera pinouts
       input wire cmos_pclk,cmos_href,cmos_vsync,
       input wire[7:0] cmos_db,
       inout cmos_sda,cmos_scl, //i2c comm wires
       output wire cmos_rst_n, cmos_pwdn, cmos_xclk, empty,

       //Debugging
       output wire[3:0] led
   );
        //FSM state declarations
        localparam idle=0,
                                       start_sccb=1,
                                       write_address=2,
                                       write_data=3,
                                       digest_loop=4,
                                       delay=5,
                                       vsync_fedge=6,
                                       byte1=7,
                                       byte2=8,
                                       fifo_write=9,
                                       stopping=10;

        localparam wait_init=0,
                                       sccb_idle=1,
                                       sccb_address=2,
                                       sccb_data=3,
                                       sccb_stop=4;

       localparam                      rest = 0,
                                       vsync_fedge_SD = 1,
                                       byte1_SD = 2,
                                       byte2_SD = 3;


        localparam MSG_INDEX=77; //number of the last index to be digested by SCCB



        reg[3:0] state_q=0,state_d;
        reg[2:0] sccb_state_q=0,sccb_state_d;
```

```
        reg[7:0] addr_q,addr_d;
        reg[7:0] data_q,data_d;
        reg[7:0] brightness_q,brightness_d;
        reg[7:0] contrast_q,contrast_d;
        reg start,stop;
        reg[7:0] wr_data;
        wire rd_tick;
        wire[1:0] ack;
        wire[7:0] rd_data;
        wire[3:0] state;
        reg[3:0] led_q=0,led_d;
        reg[27:0] delay_q=0,delay_d;
        reg start_delay_q=0,start_delay_d;
        reg delay_finish;
        reg[15:0] message[250:0];
        reg[7:0] message_index_q=0,message_index_d;
        reg[15:0] pixel_q,pixel_d;
        reg wr_en, wr_en_SD;
        reg mod2_q=0,mod2_d;
        wire full;
        wire key0_tick,key1_tick,key2_tick,key3_tick;
        reg[2:0] lines_q,lines_d;
        reg[18:0] count_q=0,count_d;

        reg[3:0] state_q_SD=0, state_d_SD;

        //buffer for all inputs coming from the camera
        reg pclk_1,pclk_2,href_1,href_2,vsync_1,vsync_2;


        initial begin //collection of all adddresses and values to be written in the camera
                              //{address,data}
        message[0]=16'h12_80;  //reset all register to default values
        message[1]=16'h12_04;  //set output format to RGB
        message[2]=16'h15_20;  //pclk will not toggle during horizontal blank
        message[3]=16'h40_d0;  //RGB565

       // These are values scalped from https://github.com/jonlwowski012/OV7670_NEXYS4_Verilog/blob/master/ov7670_registers_verilog.v
    message[4]= 16'h12_04; // COM7,     set RGB color output
    message[5]= 16'h11_80; // CLKRC     internal PLL matches input clock
    message[6]= 16'h0C_00; // COM3,     default settings
    message[7]= 16'h3E_00; // COM14,    no scaling, normal pclock
    message[8]= 16'h04_00; // COM1,     disable CCIR656
    message[9]= 16'h40_d0; //COM15,     RGB565, full output range
    message[10]= 16'h3a_04; //TSLB       set correct output data sequence (magic)
         message[11]= 16'h14_18; //COM9       MAX AGC value x4 0001_1000
    message[12]= 16'h4F_B3; //MTX1       all of these are magical matrix coefficients
    message[13]= 16'h50_B3; //MTX2write
    message[14]= 16'h51_00; //MTX3
    message[15]= 16'h52_3d; //MTX4
    message[16]= 16'h53_A7; //MTX5
```

```
    message[17]= 16'h54_E4; //MTX6
    message[18]= 16'h58_9E; //MTXS
    message[19]= 16'h3D_C0; //COM13      sets gamma enable
    message[20]= 16'h17_14; //HSTART     start high 8 bits
    message[21]= 16'h18_02; //HSTOP      stop high 8 bits //these kill the odd colored line
    message[22]= 16'h32_80; //HREF       edge offset
    message[23]= 16'h19_03; //VSTART     start high 8 bits
    message[24]= 16'h1A_7B; //VSTOP      stop high 8 bits
    message[25]= 16'h03_0A; //VREF       vsync edge offset
    message[26]= 16'h0F_41; //COM6       reset timings
    message[27]= 16'h1E_00; //MVFP       disable mirror / flip //might have magic value of 03
    message[28]= 16'h33_0B; //CHLF       //magic value from the internet
    message[29]= 16'h3C_78; //COM12      no HREF when VSYNC low
    message[30]= 16'h69_00; //GFIX       fix gain control
    message[31]= 16'h74_00; //REG74      Digital gain control
    message[32]= 16'hB0_84; //RSVD       magic value from the internet *required* for good color
    message[33]= 16'hB1_0c; //ABLC1
    message[34]= 16'hB2_0e; //RSVD       more magic internet values
    message[35]= 16'hB3_80; //THL_ST
    //begin mystery scaling numbers
    message[36]= 16'h70_3a;
    message[37]= 16'h71_35;
    message[38]= 16'h72_11;
    message[39]= 16'h73_f0;
    message[40]= 16'ha2_02;
    //gamma curve values
    message[41]= 16'h7a_20;
    message[42]= 16'h7b_10;
    message[43]= 16'h7c_1e;
    message[44]= 16'h7d_35;
    message[45]= 16'h7e_5a;
    message[46]= 16'h7f_69;
    message[47]= 16'h80_76;
    message[48]= 16'h81_80;
    message[49]= 16'h82_88;
    message[50]= 16'h83_8f;
    message[51]= 16'h84_96;
    message[52]= 16'h85_a3;
    message[53]= 16'h86_af;
    message[54]= 16'h87_c4;
    message[55]= 16'h88_d7;
    message[56]= 16'h89_e8;
    //AGC and AEC
    message[57]= 16'h13_e0; //COM8, disable AGC / AEC
    message[58]= 16'h00_00; //set gain reg to 0 for AGC
    message[59]= 16'h10_00; //set ARCJ reg to 0
    message[60]= 16'h0d_40; //magic reserved bit for COM4
    message[61]= 16'h14_18; //COM9, 4x gain + magic bit
    message[62]= 16'ha5_05; // BD50MAX
    message[63]= 16'hab_07; //DB60MAX
    message[64]= 16'h24_95; //AGC upper limit
    message[65]= 16'h25_33; //AGC lower limit
```

```
    message[66]= 16'h26_e3; //AGC/AEC fast mode op region
    message[67]= 16'h9f_78; //HAECC1
    message[68]= 16'ha0_68; //HAECC2
    message[69]= 16'ha1_03; //magic
    message[70]= 16'ha6_d8; //HAECC3
    message[71]= 16'ha7_d8; //HAECC4
    message[72]= 16'ha8_f0; //HAECC5
    message[73]= 16'ha9_90; //HAECC6
    message[74]= 16'haa_94; //HAECC7
    message[75]= 16'h13_e5; //COM8, enable AGC / AEC
         message[76]= 16'h1E_23; //Mirror Image
         message[77]= 16'h69_06; //gain of RGB(manually adjusted)
 end

         //register operations
         always @(posedge clk_100,negedge rst_n) begin
                if(!rst_n) begin
                        state_q<=0;
                        state_q_SD<=0;
                        led_q<=0;
                        delay_q<=0;
                        start_delay_q<=0;
                        message_index_q<=0;
                        pixel_q<=0;

                        sccb_state_q<=0;
                        addr_q<=0;
                        data_q<=0;
                        brightness_q<=0;
                        contrast_q<=0;
                end
                else begin
                        state_q<=state_d;
                        state_q_SD<=state_d_SD;
                        delay_q<=delay_d;
                        start_delay_q<=start_delay_d;
                        message_index_q<=message_index_d;
                        pclk_1<=cmos_pclk;
                        pclk_2<=pclk_1;
                        href_1<=cmos_href;
                        href_2<=href_1;
                        vsync_1<=cmos_vsync;
                        vsync_2<=vsync_1;
                        pixel_q<=pixel_d;

                        sccb_state_q<=sccb_state_d;
                        addr_q<=addr_d;
                        data_q<=data_d;
                        brightness_q<=brightness_d;
                        contrast_q<=contrast_d;
                end
         end
```

```
        //FSM next-state logics
        always @* begin
                state_d=state_q;
                led_d=led_q;
                start=0;
                stop=0;
                wr_data=0;
                start_delay_d=start_delay_q;
                delay_d=delay_q;
                delay_finish=0;
                message_index_d=message_index_q;
                pixel_d=pixel_q;
                wr_en=0;

                sccb_state_d=sccb_state_q;
                addr_d=addr_q;
                data_d=data_q;
                brightness_d=brightness_q;
                contrast_d=contrast_q;

                //delay logic
                if(start_delay_q) delay_d=delay_q+1'b1;
                if(delay_q[16] && message_index_q!=(MSG_INDEX+1) && (state_q!=start_sccb))  begin  //delay between SCCB transmissions (0.66ms)
                        delay_finish=1;
                        start_delay_d=0;
                        delay_d=0;
                end
                else if((delay_q[26] && message_index_q==(MSG_INDEX+1)) || (delay_q[26] && state_q==start_sccb)) begin //delay BEFORE SCCB transmission, AFTER
BEFORE retrieving pixel data from camera (0.67s)
                        delay_finish=1;
                        start_delay_d=0;
                        delay_d=0;
                end
        if (!pir) begin
                case(state_q)

                                        ////////Begin: Setting register values of the camera via SCCB//////////

                        idle:  if(delay_finish) begin //idle for 0.6s to start-up the camera
                                                state_d=start_sccb;
                                                start_delay_d=0;
                                        end
                                        else start_delay_d=1;

                start_sccb:  begin   //start of SCCB transmission
                                                start=1;
                                                wr_data=8'h42; //slave address of OV7670 for write
                                                state_d=write_address;
                                        end
        write_address: if(ack==2'b11) begin
                                                wr_data=message[message_index_q][15:8]; //write address
                                                state_d=write_data;
                                        end
                write_data: if(ack==2'b11) begin
                                                wr_data=message[message_index_q][7:0]; //write data
                                                state_d=digest_loop;
                                        end
         digest_loop: if(ack==2'b11) begin //stop sccb transmission
                                                stop=1;
                                                start_delay_d=1;
                                                message_index_d=message_index_q+1'b1;
                                                state_d=delay;
                                        end
                        delay: begin
                                                if(message_index_q==(MSG_INDEX+1) && delay_finish) begin
                                                        state_d=vsync_fedge; //if all messages are already digested, proceed to retrieving camera pixel data
                                                        led_d=4'b0110;
                                                end
                                                else if(state==0 && delay_finish) state_d=start_sccb; //small delay before next SCCB transmission(if all messag
digested)
                                        end



                        //////////////Begin: Retrieving Pixel Data from Camera to be Stored to SDRAM/////////////////

                vsync_fedge: if(vsync_1==0 && vsync_2==1) state_d=byte1; //vsync falling edge means new frame is incoming
                                byte1: if(pclk_1==1 && pclk_2==0 && href_1==1 && href_2==1) begin //rising edge of pclk means new pixel data(first byte of 16-bit pixel
available at output
                                                        pixel_d[15:8]=cmos_db;
                                                        state_d=byte2;
                                                end
                                                else if(vsync_1==1 && vsync_2==1) begin
                                                        state_d=vsync_fedge;
                                                end
                                byte2: if(pclk_1==1 && pclk_2==0 && href_1==1 && href_2==1) begin //rising edge of pclk means new pixel data(second byte of 16-bit pixe
available at output
                                                        pixel_d[7:0]=cmos_db;
                                                        state_d=fifo_write;
                                                end
                                                else if(vsync_1==1 && vsync_2==1) begin
                                                        state_d=vsync_fedge;
                                                end
                fifo_write: begin //write the 16-bit data to asynchronous fifo to be retrieved later by SDRAM
                                                        wr_en=1;
                                                        state_d=byte1;
                                                        if(full) led_d=4'b1001; //debugging led
                                                end
                default: state_d=idle;
                endcase

                //Logic for increasing/decreasing brightness and contrast via the 4 keybuttons
                case(sccb_state_q)
```

```
                wait_init: if(state_q==byte1) begin //wait for initial SCCB transmission to finish
                                                sccb_state_d=sccb_idle;
                                                addr_d=0;
                                                data_d=0;
                                                brightness_d=8'h00;
                                                contrast_d=8'h40;
                                        end
                sccb_idle: if(state==0) begin //wait for any pushbutton
                                                if(key0_tick) begin//increase brightness
                                                        brightness_d=(brightness_q[7]==1)? brightness_q-1:brightness_q+1;
                                                        if(brightness_q==8'h80) brightness_d=0;
                                                        start=1;
                                                        wr_data=8'h42; //slave address of OV7670 for write
                                                        addr_d=8'h55; //brightness control address
                                                        data_d=brightness_d;
                                                        sccb_state_d=sccb_address;
                                                        led_d=0;
                                                end
                                                if(key1_tick) begin //decrease brightness
                                                        brightness_d=(brightness_q[7]==1)? brightness_q+1:brightness_q-1;
                                                        if(brightness_q==0) brightness_d=8'h80;
                                                        start=1;
                                                        wr_data=8'h42;
                                                        addr_d=8'h55;
                                                        data_d=brightness_d;
                                                        sccb_state_d=sccb_address;
                                                        led_d=0;
                                                end
                                                else if(key2_tick) begin //increase contrast
                                                        contrast_d=contrast_q+1;
                                                        start=1;
                                                        wr_data=8'h42; //slave address of OV7670 for write
                                                        addr_d=8'h56; //contrast control address
                                                        data_d=contrast_d;
                                                        sccb_state_d=sccb_address;
                                                        led_d=0;
                                                end
                                                else if(key3_tick) begin //change contrast
                                                        contrast_d=contrast_q-1;
                                                        start=1;
                                                        wr_data=8'h42;
                                                        addr_d=8'h56;
                                                        data_d=contrast_d;
                                                        sccb_state_d=sccb_address;
                                                        led_d=0;
                                                end
                                        end
        sccb_address: if(ack==2'b11) begin
                                                wr_data=addr_q; //write address
                                                sccb_state_d=sccb_data;
                                        end
          sccb_data: if(ack==2'b11) begin
```

```
                                        wr_data=data_q; //write databyte
                                        sccb_state_d=sccb_stop;
                                    end
    sccb_stop: if(ack==2'b11) begin //stop
                                        stop=1;
                                        sccb_state_d=sccb_idle;
                                        led_d=4'b1001;
                                    end
            default: sccb_state_d=wait_init;
endcase
end

else begin
case(state_q_SD)

                ///////////////Begin: Retrieving Pixel Data from Camera to be Stored to SDRAM/////////////////

rest: if(pir) begin
        lines_d=0;
        state_d_SD=vsync_fedge;
end

vsync_fedge_SD: begin
      if(vsync_1==0 && vsync_2==1 && empty && lines_q<5) begin
        lines_d=lines_q+1;
        state_d_SD=byte1; //vsync falling edge means new frame is incoming
        count_d=0;
        end
      else if(lines_q==5) begin
        state_d_SD=rest;
        led_d=4'b0110;
      end
end

byte1_SD: if(pclk_1==1 && pclk_2==0 && href_1==1 && href_2==1) begin //rising edge of pclk means new pixel data(first byte o
                case(lines_q)
                        1:wr_en_SD=count_q>=0 && count_q<=65535;
                        2:wr_en_SD=count_q>=65536 && count_q<=131071;
                        3:wr_en_SD=count_q>=131072 && count_q<=196607;
                        4:wr_en_SD=count_q>=196608 && count_q<=262143;
                        5:wr_en_SD=count_q>=262144 && count_q<=327679;
                endcase
            state_d_SD=byte2;
                                led_d=4'b1001;
        end
        else if(vsync_1==1 && vsync_2==1) begin
                state_d_SD=vsync_fedge;
        end

byte2_SD: if(pclk_1==1 && pclk_2==0 && href_1==1 && href_2==1) begin //rising edge of pclk means new pixel data(second byte
                case(lines_q)
                        1:wr_en_SD=count_q>=0 && count_q<=65535;
```

```
                                    2:wr_en_SD=count_q>=65536 && count_q<=131071;
                                    3:wr_en_SD=count_q>=131072 && count_q<=196607;
                                    4:wr_en_SD=count_q>=196608 && count_q<=262143;
                                    5:wr_en_SD=count_q>=262144 && count_q<=327679;
                                endcase
                                state_d_SD=byte1;
                                count_d=(count_q<307200)? (count_q+1'b1):count_q;
                        end
                        else if(vsync_1==1 && vsync_2==1) begin
                                                        state_d_SD=vsync_fedge;
                        end
                //default: state_d_SD=idle_SD;
                endcase
        end

end


        assign cmos_pwdn=0;
        assign cmos_rst_n=1;
        assign led=led_q;

        //module instantiations
        i2c_top #(.freq(100_000)) m0
        (
                .clk(clk_100),
                .rst_n(rst_n),
                .start(start),
                .stop(stop),
                .wr_data(wr_data),
                .rd_tick(rd_tick), //ticks when read data from servant is ready,data will be taken from rd_data
                .ack(ack), //ack[1] ticks at the ack bit[9th bit],ack[0] asserts when ack bit is ACK,else NACK
                .rd_data(rd_data),
                .scl(cmos_scl),
                .sda(cmos_sda),
                .state(state)
    );

    clock_24_0002 clock_24_inst (
                .refclk   (clk),   //  refclk.clk
                .rst      (rst),      //   reset.reset
                .outclk_0 (cmos_xclk), // outclk0.clk
                .locked   ()          // (terminated)
        );

        asyn_fifo #(.DATA_WIDTH(16),.FIFO_DEPTH_WIDTH(10)) m2 //1024x16 FIFO mem
        (
                .rst_n(rst_n),
                .clk_write(clk_100),
                .clk_read(clk_100), //clock input from both domains
                .write(wr_en),
                .read(rd_en),
```

```
                .data_write(pixel_q), //input FROM write clock domain
                .data_read(dout), //output TO read clock domain
                .full(full),
                .empty(), //full=sync to write domain clk , empty=sync to read domai
                .data_count_r(data_count_r) //asserted if fifo is equal or more thar
    );

        asyn_fifo #(.DATA_WIDTH(16),.FIFO_DEPTH_WIDTH(10)) m3 //2048x8 FIFO mem
        (
                .rst_n(rst_n),
                .clk_write(clk_100),
                .clk_read(clk_100), //clock input from both domains
                .write(wr_en_SD),
                .read(rd_en_SD),
                .data_write(cmos_db), //input FROM write clock domain
                .data_read(dout_SD), //output TO read clock domain
                .full(full),
                .empty(empty), //full=sync to write domain clk , empty=sync to read
    );

        debounce_explicit m4
        (
                .clk(clk_100),
                .rst_n(rst_n),
                .sw({!key[0]}),
                .db_level(),
                .db_tick(key0_tick)
    );

        debounce_explicit m5
        (
                .clk(clk_100),
                .rst_n(rst_n),
                .sw({!key[1]}),
                .db_level(),
                .db_tick(key1_tick)
    );

         debounce_explicit m6
        (
                .clk(clk_100),
                .rst_n(rst_n),
                .sw({!key[2]}),
                .db_level(),
                .db_tick(key2_tick)
    );

         debounce_explicit m7
        (
                .clk(clk_100),
                .rst_n(rst_n),
                .sw({!key[3]}),
```

```
                .sw({!key[3]}),
                .db_level(),
                .db_tick(key3_tick)
    );

endmodule
```

```
`timescale 1ns / 1ps

module vga_interface(
        input wire clk,rst_n,
        //asyn_fifo IO
        input wire empty_fifo,
        input wire[15:0] din,
        output wire clk_vga,
        output reg rd_en,
        //VGA output
        output reg[7:0] vga_out_r,
        output reg[7:0] vga_out_g,
        output reg[7:0] vga_out_b,
        output wire vga_out_vs,vga_out_hs
    );
         //FSM state declarations
         localparam delay=0,
                                          idle=1,
                                          display=2;

         reg[1:0] state_q,state_d;
         reg [7:0] r8Bit;
         reg [7:0] g8Bit;
         reg [7:0] b8Bit;
         wire[11:0] pixel_x,pixel_y;
         //register operations
         always @(posedge clk_out,negedge rst_n) begin
                if(!rst_n) begin
                        state_q<=delay;
                end
                else begin
                        state_q<=state_d;
                end
         end

         //FSM next-state logic
         always @* begin
         state_d=state_q;
         rd_en=0;
         r8Bit = din[15:11];
         g8Bit = din[10:5];
         b8Bit = din[4:0];
         r8Bit = (255/31) * r8Bit;
         g8Bit = (255/63) * g8Bit;
         b8Bit = (255/31) * b8Bit;
         //r8Bit = (din[15:11] * 10'b1000001111 + 23) >> 7;
         //g8Bit = (din[10:5] * 9'b100000011 + 33) >> 6;
         //b8Bit = (din[4:0] * 10'b1000001111 + 23) >>7;
         //r8Bit = (r8Bit/31) + 8'b0;
         //g8Bit = (g8Bit/63) + 8'b0;
         //b8Bit = (b8Bit/31) + 8'b0;
         vga_out_r=0;
```

```
        vga_out_g=0;
        vga_out_b=0;
                case(state_q)
                  delay: if(pixel_x==1 && pixel_y==1) state_d=idle; //delay of one frame(33ms) needed to start
                        idle:  if(pixel_x==1 && pixel_y==0 && !empty_fifo) begin //wait for pixel-data coming
                                                        //vga_out_r=din[15:11];
                                                        //vga_out_g=din[10:5];
                                                        //vga_out_b=din[4:0];
                                                        vga_out_r = r8Bit;
                                                        vga_out_g = g8Bit;
                                                        vga_out_b = b8Bit;
                                                        rd_en=1;
                                                        state_d=display;
                                        end
                display: if(pixel_x>=1 && pixel_x<=640 && pixel_y<480) begin //we will continue to read the as
screen(640x480)
                                                vga_out_r=r8Bit;
                                                vga_out_g=g8Bit;
                                                vga_out_b=b8Bit;

                                                rd_en=1;
                                        end
                        idle: state_d=delay;
                endcase
        end



       assign clk_vga=clk_out;

       //module instantiations
       vga_core m0
       (
               .clk(clk_out), //clock must be 25MHz for 640x480
               .rst_n(rst_n),
               .hsync(vga_out_hs),
               .vsync(vga_out_vs),
               .video_on(),
               .pixel_x(pixel_x),
               .pixel_y(pixel_y)
       );

        clock_25_0002 clock_25_inst (
               .refclk   (clk),   //  refclk.clk
               .rst      (rst),      //   reset.reset
               .outclk_0 (clk_out), // outclk0.clk
               .locked   ()          // (terminated)
       );
endmodule
```

```
`timescale 1ns / 1ps

module sdcard_interface(
    input wire clk,
    input wire rst,
    output wire led0_r,led0_g,led0_b, //{red,green,blue} red if SDCARD initialization is
    output idle, //sdcard not busy
    //HOST interface
    input write, //start writing to SD card
    output reg rd_fifo, //read next data to be written
    input[15:0] data, //data to be written to SD catd
    //SPI pinouts
        input wire SD_MISO,
        output wire SD_MOSI,
        output wire SD_DCLK,SD_nCS,
        //UART for debugging
        output wire uart_rx,uart_tx
    );
         //FSM states
    localparam POWER_ON=0,
               COMMANDS=1,
               SEND_COMMAND=2,
               RECEIVE_RESPONSE=3,
               END_CMD=4,
                           IDLE=5,
                           DELAY=6,
                           WRITE_1=7,
                           WRITE_2=8,
                           BUSY=9;


        reg[3:0] state_q=0,state_d;
        reg[9:0] counter_q=0,counter_d; //counter for the 74 clk cycles needed for power
        reg[3:0] cmd_counter_q=0,cmd_counter_d; //index for cmd_list
        reg[3:0] response_counter_q=0,response_counter_d; //number of bytes needed for a
        reg[55:0] wr_data_q=0,wr_data_d;
        reg[39:0] rd_data_q=0,rd_data_d;
        reg[2:0] led_q=0,led_d;
        reg stuck_q=0,stuck_d;
        reg[9:0] stuck_counter_q=0,stuck_counter_d;
        reg[15:0] addr_counter_q,addr_counter_d;

    //SPI pinouts
    reg rd,wr,hold;
    reg[7:0] wr_data;
    reg clk_div_q=0,clk_div_d;
    wire[7:0] rd_data;
    wire done_tick,ready;
    wire clk_div,cs_n_1;
```

```
//uart PINOUTS
      reg wr_uart,rd_uart;
      reg[7:0] wr_data_uart;
      wire[7:0] rd_data_uart;
      wire rx_empty;

      //list of commands for SDCARD initialization
      localparam INIT_LAST_INDEX=6;
      reg[55:0] cmd_list[10:0];
      initial begin
             cmd_list[0]=48'h40_00_00_00_00_95;  //CMD0: G
             cmd_list[1]=48'h48_00_00_01_AA_87;  //CMD8: S
             cmd_list[2]=48'h7B_00_00_00_00_83;  //CMD59 C
             cmd_list[3]=48'h77_00_00_00_00_00;  //CMD55:
             cmd_list[4]=48'h69_40_00_00_00_00;  //ACMD41:
             cmd_list[5]=48'h7A_00_00_00_00_00;  //CMD58:R
             cmd_list[6]=48'h50_00_00_02_00_00; //CMD16: s
             cmd_list[7]=48'h58_00_00_00_00_00;  //CMD24:
             cmd_list[8]=48'h4D_00_00_00_00_00;  //Status
      end


//register operations
always @(posedge clk,posedge rst) begin
    if(rst) begin
        state_q<=0;
        counter_q<=0;
        wr_data_q<=0;
        rd_data_q<=0;
        led_q<=0;
        cmd_counter_q<=0;
        response_counter_q<=0;
        stuck_q<=0;
        stuck_counter_q<=0;
        clk_div_q<=0;
        addr_counter_q<=0;
    end
    else begin
        state_q<=state_d;
        counter_q<=counter_d;
        wr_data_q<=wr_data_d;
        rd_data_q<=rd_data_d;
        led_q<=led_d;
        cmd_counter_q<=cmd_counter_d;
        response_counter_q<=response_counter_d;
        stuck_q<=stuck_d;
        stuck_counter_q<=stuck_counter_d;
        clk_div_q<=clk_div_d;
        addr_counter_q<=addr_counter_d;
    end
end
```

```
//FSM logic
always @* begin
    state_d=state_q;
    counter_d=counter_q;
    wr_data_d=wr_data_q;
    rd_data_d=rd_data_q;
    led_d=led_q;
    cmd_counter_d=cmd_counter_q;
    response_counter_d=response_counter_q;
    stuck_d=stuck_q;
    stuck_counter_d=stuck_counter_q;
    clk_div_d=clk_div_q;
    addr_counter_d=addr_counter_q;
    rd=0;
    wr=0;
    hold=0;
    wr_data=8'hff;
    rd_fifo=0;

    case(state_q)
  ////////////////////////////////////START SDCARD INITIALIZATION//////////////////////////
        POWER_ON: begin //send at least 74 clk cycles with cs_n and d_out line high
                    rd=1;
                    led_d=3'b100;
                                        cmd_counter_d=0;
                                        clk_div_d=0;
                                        addr_counter_d=0;
                    if(done_tick) begin
                        counter_d=counter_q+1'b1;
                        if(counter_q==15) begin//8*10=80 clk cycles had passed
                            rd=0;
                            state_d=COMMANDS;
                        end
                    end
                  end

        COMMANDS: if(ready) begin //commands to be sent to SDCARD
                    wr_data_d=cmd_list[cmd_counter_q];
                    if(cmd_counter_q==7) wr_data_d[39:8]=2049+addr_counter_q; //start addr
                    state_d=SEND_COMMAND;
                                            response_counter_d=0;
                                            counter_d=0;
                            stuck_counter_d=0;
                            stuck_d=0;
                  end

    SEND_COMMAND: if(ready || done_tick) begin
                    wr_data_d={wr_data_q[47:0],8'hff}; //shift by 1 byte
                    wr_data=wr_data_d[55:48];
                    wr=1;
                    counter_d=counter_q+1'b1;
```

```
                        if(counter_q==7) begin //6 bytes had been sent to SPI, another 1 byte for the 8 clk cycles nee
                                                wr=0;
                            rd=1; //response always starts at logic 0 so hold the clock until then
                            counter_d=0;
                            state_d=RECEIVE_RESPONSE;
                        end
                    end

    RECEIVE_RESPONSE: if(done_tick) begin
                        response_counter_d=response_counter_q+1; //counter for some responses that has multiple bytes
                        //rules for types of response for every command
                        case(cmd_counter_q)
                            0: begin
                                    cmd_counter_d=(rd_data==8'h01)? cmd_counter_q+1:cmd_counter_q; //CMD0: resets SDCA
                                    state_d=END_CMD;
                                end
                            1: begin
                                    rd_data_d={rd_data_q[31:0],rd_data};
                                    rd=1;
                                    led_d=3'b001;
                                    if(response_counter_d==5) begin//5 bytes had been received
                                        cmd_counter_d=(rd_data_d==40'h01_00_00_01_AA)? cmd_counter_q+1:cmd_counter_q;
later
                                        rd=0;
                                        state_d=END_CMD;
                                    end
                                end
                            2: begin
                                    cmd_counter_d=(rd_data==8'h01)? cmd_counter_q+1:cmd_counter_q; //CMD59: turns off
                                    state_d=END_CMD;
                                end
                            3: begin
                                    cmd_counter_d=(rd_data==8'h01)? cmd_counter_q+1:cmd_counter_q; //CMD55: Now ready
                                    state_d=END_CMD;
                                end
                            4: begin
                                    stuck_counter_d=stuck_counter_q+1;
                                    if(stuck_counter_q==1000) stuck_d=1; //if ACMD41 stuck 1000x , go back to power-on
                                    cmd_counter_d=(rd_data==8'h00)? cmd_counter_q+1:cmd_counter_q-1; //ACMD41: Initia
                                    state_d=END_CMD;
                                end
                            5: begin
                                    rd_data_d={rd_data_q[31:0],rd_data};
                                    rd=1;
                                    if(response_counter_d==5) begin//5 bytes had been received
                                            rd=0;
                                            cmd_counter_d=(rd_data_d==40'h00_C0_FF_80_00)? cmd_counter_q+1:cmd_counte
Capacity) or SDXC(eXtended Capacity)
                                            state_d=END_CMD;
                                    end
                                end
                            6: begin
```

```
                        6: begin
                               cmd_counter_d=(rd_data==8'h00)? cmd_counter_q+1:cmd_counter_q; //CMD55: Now ready for next comma
                               state_d=END_CMD;
                           end
                        7: begin //acknowledge for write
                               state_d=WRITE_1;
                           end
                        8: begin
                               rd_data_d={rd_data_q[31:0],rd_data};
                               rd=1;
                               if(response_counter_d==2) begin//5 bytes had been received
                                   cmd_counter_d=(rd_data_d==16'h0000)? cmd_counter_q+1:cmd_counter_q; //CMD8: Host Voltage Sup
                                   rd=1;
                                   state_d=IDLE;
                                   led_d=3'b010;
                               end
                           end
                    endcase
                   end

                       END_CMD:if(ready) begin //must provide 8 clks before shutting down sclk or starting new command
                       wr=1;
                       state_d=stuck_q? POWER_ON:COMMANDS;
                       if(cmd_counter_q==INIT_LAST_INDEX+1 || clk_div_q) state_d=DELAY;
                    end
            DELAY: if(ready) begin  //delay before switching to high frequency for writing in SDCARD
                     stuck_counter_d=stuck_counter_d+1;
                     clk_div_d=1;
                     if(stuck_counter_d==1000) begin
                     led_d=3'b010;
                     state_d=IDLE;
                     end
                   end
//////////////////////////////////////INITIALIZATION COMPLETE/////////////////////////////////////////////


//////////////////////////SDCARD READ/WRITE OPERATION/////////////////////////////////////////////////
                                   IDLE: if(write) begin
                                             cmd_counter_d=7; //WRITE
                                             state_d=COMMANDS;
                                             addr_counter_d=addr_counter_q+1;
                                             led_d=100;
                                         end
                                WRITE_1: if(ready || done_tick) begin
                     if(counter_q==0 || counter_q==513 || counter_q==514) wr_data=8'b1111_1110; //start_token
                     else begin
                         wr_data=data;////////////////////
                         rd_fifo=1;
                     end
                     wr=1;
                     counter_d=counter_q+1'b1;
```

```
                        if(counter_q==515) begin //515 bytes had been sent to SPI,
                            wr=0;
                            rd_fifo=0;
                            rd=1; //Data response immediately
                            counter_d=0;
                            state_d=WRITE_2;
                        end
                    end
            WRITE_2: if(done_tick) begin
                            if(rd_data[4:0] == 5'b0_010_1) begin //data accepted
                                state_d=BUSY;
                            end
                     end
                BUSY: begin
                        rd=1;
                        if(SD_MISO && done_tick) begin //sd card finishes the writing operation
                            cmd_counter_d=8;
                            state_d=COMMANDS;
                        end
                    end
          default: state_d=POWER_ON;
    endcase
end

assign SD_nCS=(state_q==POWER_ON || state_q==COMMANDS || (state_q==END_CMD && ready) || state_q==IDLE)? 1'b1:cs_n_1; /

    assign led0_r=led_q[2]? clk:1'b1,
           led0_g=led_q[1]? clk:1'b1,
           led0_b=led_q[0]? clk:1'b1; //PWM used is the clk itself
assign idle=state_q==IDLE;

//module instantiations
spi #(.HI_FREQ_DIV(20), .LO_FREQ_DIV(358), .SPI_MODE(0)) m0 //High freq: 100MHz/6=16.7MHz , Low freq: 100MHz/250=400KH
(
    .clk(clk),
    .rst(rst),
    //SPI control
    .clk_div(clk_div_q),
    .rd(rd),
    .wr(wr),
    .hold(hold),  //pins to start read or write operation, hold is for holding the clock for multibyte read/write
    .wr_data(wr_data), //data to be sent to the slave
    .rd_data(rd_data), //data received from the slave
    .done_tick(done_tick), //ticks if either write or read operation is finished
    .ready(ready), //can perform read/write operation only if ready is "1" (except for multibyte read/write where stat
    //SPI pinouts
    .miso(SD_MISO),
    .mosi(SD_MOSI),
    .sclk(SD_DCLK),
    .cs_n(cs_n_1)
    );
```

```
        uart #(.DBIT(8),.SB_TICK(16),.DVSR(326),.DVSR_WIDTH(9),.FIFO_W(10)) m1 //9600 Baud
       (
               .clk(clk),
               .rst_n({!rst}),
               .rd_uart(rd_uart),
               .wr_uart(wr_uart),
               .wr_data(wr_data_uart),
               .rx(uart_rx),
               .tx(uart_tx),
               .rd_data(rd_data_uart),
               .rx_empty(rx_empty),
               .tx_full()
    );

    //UART for debugging SDCARD responses
    always @* begin
    wr_uart=0;
    rd_uart=0;
        wr_data_uart=0;

        wr_uart= wr || (state_q==RECEIVE_RESPONSE && done_tick) || (state_q==WRITE_2 && done_tick);
        wr_data_uart=wr? wr_data:rd_data;
    end


endmodule
```

```
module pirSens(
        input wire pirTest,
        output reg ledPIR
        );

        always @* begin
                if (pirTest) begin
                        ledPIR = 1;
                end
                else begin
                        ledPIR = 0;
                end
        end
endmodule
```

```
`timescale 1ns / 1ps

module sdram_interface(
        input clk,rst_n,
        //asyn_fifo IO
        input wire clk_vga, rd_en,
        input wire[9:0] data_count_r,
        input wire[15:0] f2s_data,
        output wire f2s_data_valid,
        output wire empty_fifo,
        output wire[15:0] dout,
        //controller to sdram
        output wire sdram_clk,
        output wire sdram_cke,
        output wire sdram_cs_n, sdram_ras_n, sdram_cas_n, sdram_we_n,
        output wire[12:0] sdram_addr,
        output wire[1:0] sdram_ba,
        output wire[1:0] sdram_dqm,
        inout[15:0] sdram_dq
    );
         //FSM state declarations
         localparam idle=0,
                                          burst_op=1;

         reg state_q=0,state_d;
         reg[14:0] wr_addr_q=0,wr_addr_d;
         reg[14:0] rd_addr_q=0,rd_addr_d;
         reg rw,rw_en;
         reg[14:0] f_addr;
         wire[15:0] s2f_data;
         wire s2f_data_valid;
         wire ready;
         wire[9:0] data_count_w;



         //register operation
         always @(posedge clk,negedge rst_n) begin
                if(!rst_n) begin
                        state_q<=0;
                        wr_addr_q<=0;
                        rd_addr_q<=0;
                end
                else begin
                        state_q<=state_d;
                        wr_addr_q<=wr_addr_d;
                        rd_addr_q<=rd_addr_d;
                end
         end

         //FSM next-state declarations
         always @* begin
```

```
                state_d=state_q;
                wr_addr_d=wr_addr_q;
                rd_addr_d=rd_addr_q;
                f_addr=0;
                rw=0;
                rw_en=0;

                case(state_q)
                        idle: if(data_count_r>512 && ready) begin //wait for the first 512 pixel-data to fill the asyn_fifo then burst-write it
                                                        rw_en=1;
                                                        rw=0;
                                                        wr_addr_d=1;
                                                        f_addr=wr_addr_q;
                                                        state_d=burst_op;
                                                end
                              burst_op: if(ready) begin //choose whether to read the asyn_fifo of camera OR write to asyn_fifo of VGA
                                                                if(data_count_r>512) begin //asyn_fifo of camera is filled to 512 thus we can now b
to SDRAM

                                                                        rw_en=1;
                                                                        rw=0;
                                                                        wr_addr_d=(wr_addr_q==599)? 0:wr_addr_q+1'b1; //One frame(640x480) fills th
                                                                        f_addr=wr_addr_q;
                                                                end
                                                                else if(data_count_w<250) begin //asyn_fifo of VGA has only 250 pixel data left, we
data via burst reading the sdram

                                                                        rw_en=1;
                                                                        rw=1;
                                                                        rd_addr_d=(rd_addr_q==599)? 0:rd_addr_q+1'b1;
                                                                        f_addr=rd_addr_q;
                                                                end
                                                        end
                                                end
                               default: state_d=idle;
                        endcase
                end

                //module instantiations
                sdram_controller m0(
                        //fpga to controller
                        .clk(clk), //clk=165MHz
                        .rst_n(rst_n),
                        .rw(rw), // 1:read , 0:write
                        .rw_en(rw_en), //must be asserted before read/write
                        .f_addr(f_addr), //14:2=row(13)  , 1:0=bank(2) , no need for column address since full page mode will always start from zero and en
                        .f2s_data(f2s_data), //fpga-to-sdram data
                        .s2f_data(s2f_data), //sdram to fpga data
                        .s2f_data_valid(s2f_data_valid),  //asserts while  burst-reading(data is available at output UNTIL the next rising edge)
                        .f2s_data_valid(f2s_data_valid), //asserts while burst-writing(data must be available at input BEFORE the next rising edge)
                        .ready(ready), //"1" if sdram is available for nxt read/write operation
                        //controller to sdram
                        .s_clk(sdram_clk),
                        .s_cke(sdram_cke),
```

```
                .s_cs_n(sdram_cs_n),
                .s_ras_n(sdram_ras_n ),
                .s_cas_n(sdram_cas_n),
                .s_we_n(sdram_we_n),
                .s_addr(sdram_addr),
                .s_ba(sdram_ba),
                .LDQM(sdram_dqm[0]),
                .HDQM(sdram_dqm[1]),
                .s_dq(sdram_dq)
                );

        asyn_fifo #(.DATA_WIDTH(16),.FIFO_DEPTH_WIDTH(10)) m2 //1024x16 FIFO mem
        (
                .rst_n(rst_n),
                .clk_write(clk),
                .clk_read(clk_vga),
                .write(s2f_data_valid),
                .read(rd_en),
                .data_write(s2f_data), //input FROM write clock domain
                .data_read(dout), //output TO read clock domain
                .full(),
                .empty(empty_fifo), //full=sync to write domain clk , empty=sync to read domain clk
                .data_count_w(data_count_w)
    );


endmodule
```

9 Contributions

Noe Silva - SDRAM and Timing
Mir Naveen Alam - Camera Interface and VGA Interface
Eliot Flores Portillo - SCCB/I2C and VGA
Carlos Eduardo Cruz- SPI and VGA
Shifeng Zhang - PIR Sensor

Note: The contributions above represent what each member spent the most time on. However, none of the tasks were done completely individually. Every member was involved in each of the tasks.