Columbia University
CSEE 4840 Embedded Systems Design
Spring 2023

The Design Document

Eliot Samuel Flores Portillo (esf2150)
Mir Naveen Alam (ma4310)
Carlos Eduardo Cruz (cec2274)
Noe Silva (ns3567)
Shifeng Zhang(sz3104)

Table of Contents

Introduction

The main goal of this project is to implement a security camera that is capable of capturing images when motion is detected. To achieve our goal we are using Terasic's built DE1-SoC board which contains a Cyclone V FPGA system and an Arm Cortex A9 Dual Processor. We are programming the board with the software Quartus developed by Altera. The additional peripheral devices are:

- OV7670 Video Camera
- PIR Sensor
- SD Card
- VGA Monitor

System Block Diagram

Figure 1. Shows the top level block diagram of our security camera system. We are using four peripherals, a video camera for live video, a PIR sensor to detect motion, a SD card to save images and a VGA Monitor to display the images captured. As shown below, the video camera is connected to the General Purpose I/O pins of the FPGA, as well as the PIR(Passive Infrared) Sensor, The VGA monitor will display 640X480 pixel images stored in the SD Card. We will further explain how each peripheral communicates with the DE1-SoC board.

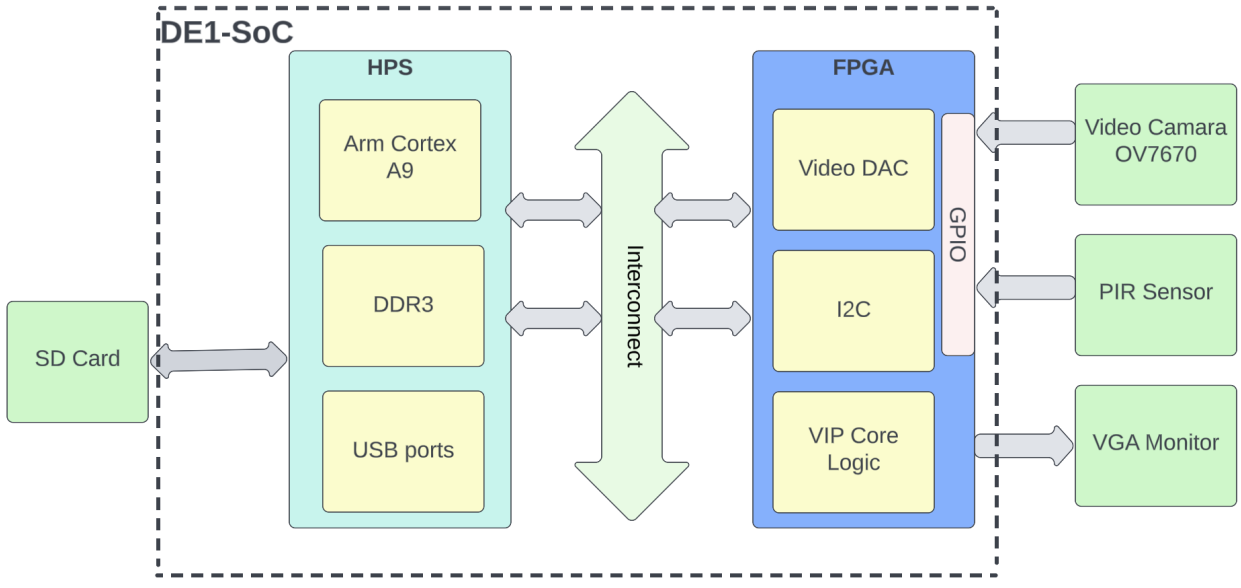

Figure 1. Block Diagram of the Security Camera System

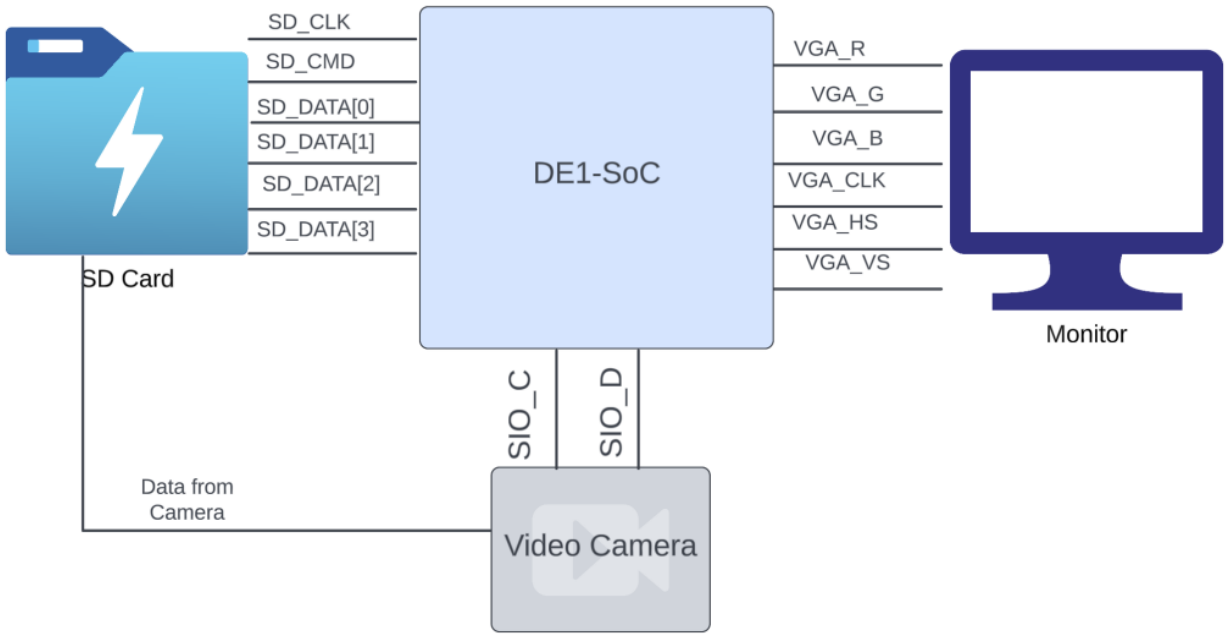

Figure 2. System Block Diagram specifying the signals from each module NOTE: Connection from SD Card Module to the Camera Module indicates the connection from the asynchronous fifo module inside the FPGA.

Algorithms

SCCB

For the communication to 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.

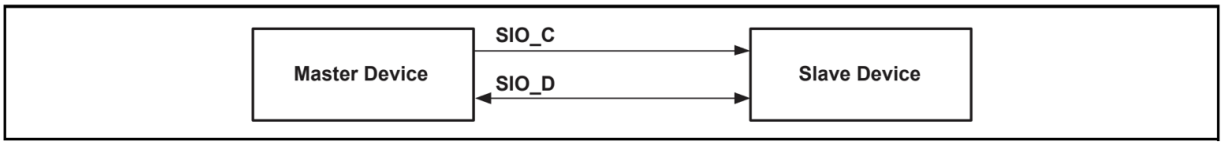

Figure 3. 2-Wire SCCB

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.

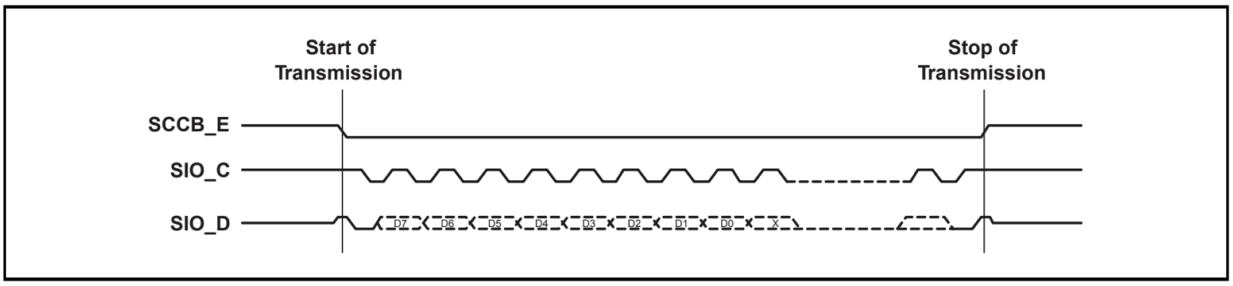

Figure 4. Waveforms for SCCB Protocol NOTE: This figure represents the 3-wire method.

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.

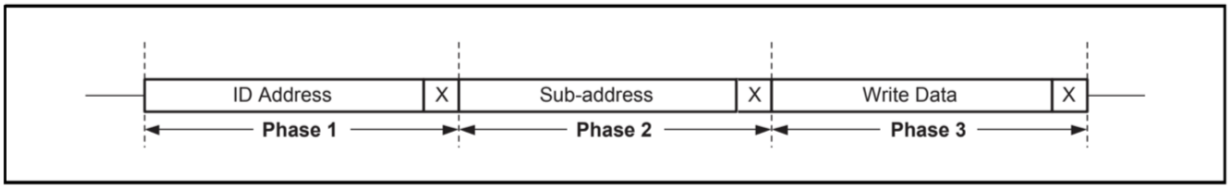

Figure 5. SCCB Data transmission

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.

SPI

Serial Peripheral Interface (SPI) will be implemented to communicate with the DE1-SOC's on board SD card reader. The SD card reader is needed to store the images captured from the camera module when motion is detected.

SPI comes in two different flavors, 3-wire and 4-wire. The 3-wire SPI has SCLK (Serial Clock), MISO (Master In Slave Out), and MOSI (Master In Slave Out). 4-wire SPI has CS (Chip Select), MOSI, MISO, and SCLK. This project will implement the 3-wire SPI since only one slave is needed. The 3-wire SPI option only uses SCLK, MISO, and MOSI.

Before storing data, initialization is needed. The HPS initiates communication by sending a clock signal to the slave device through the SCLK. Then, the HPS will send a command to the slave device though the MISO line. Once the slave receives the entire command, the slave will send a response through the MOSI line. The HPS and slave will continue doing the mentioned tasks

until there is no more data to be written into the SD card. Once the communication between the master and the slave is done, the SLCK will be deasserted.

The SD card interface consists of commands that write and read data from the SD card. The command frame consists of 6 bytes, where the first byte is the index command, the following four bytes are the arguments, and the last byte is the CRC. Once the data packet is sent, the clock will run for 8 more cycles and the SD card will send a response to the HPS through the MISO port (See figure [8]). In addition, it is known that there are 58 commands but in this project the most basic ones will be used.

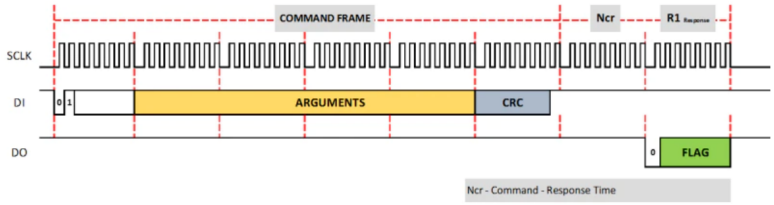

Figure 6. SPI command frame.

CMD Index	Name	Description
CMD0	GO_IDLE_STATE	Reset then start SPI mode
CMD8	SEND_IF_COND	Check voltage range
CMD59	CRC_ON_OFF	Turn on/off the CRC validation
ACMD41	SD_SEND_OP_COND	Start initialization
CMD58	READ_OCR	Check CCS bit
CMD24	WRITE_BLOCK	Write in a block(512 bytes fixed)
CMD13	SEND_STATUS	Check if write operation succeeded

Figure 7. Basic SPI commands.

Breaking each frame into five chunks

In order to fit a frame in our on chip memory, each frame has to be broken down into 5 chunks (see Resource Budgets below). To implement this in code, a counter is made that counts in all 307200 pixels (640*480) and then divides it by five. Each of the chunks is then saved to the asynchronous FIFO, which acts as a buffer for the pixel data coming from the camera. The code for this is shown below:

```
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
RGB565) is available at output
                ///////////////
                case(lines_q)
                1:wr_en=count_q>=0 && count_q<=65535;
                2:wr_en=count_q>=65536 && count_q<=131071;
                3:wr_en=count_q>=131072 && count_q<=196607;
                4:wr_en=count_q>=196608 && count_q<=262143;
                5:wr_en=count_q>=262144 && count_q<=327679;
                endcase
        state_d=byte2;
                ///////////////
                led_d=4'b1001;
        end
```

Figure 8. Frame Splitting

This shows the process for 1 of the 2 bytes coming in from the camera.

First-In-First-Out (FIFO)

As stated above, the FIFO algorithm is used to create a buffer for the pixel data coming from the camera. Asynchronous FIFO will be used to allow for more flexibility as this means that the receiving end of the data pixels does not need to be at the same clock frequency as the camera interface.

In FIFO, the first element is processed first and the newest element is processed at the end. For our project, the elements are the pixel data from the camera. The following diagram shows the basic principle of FIFO.

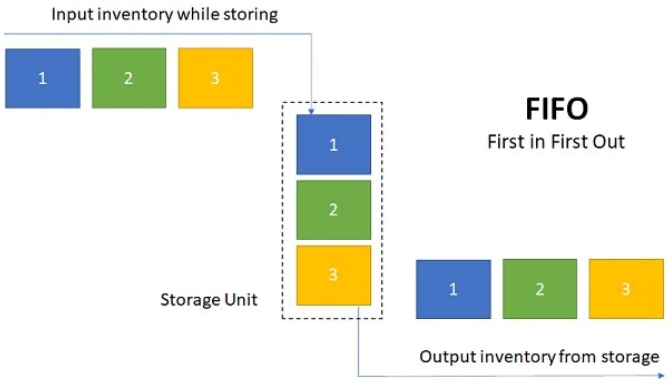

Figure 9. FIFO Block

A FIFO implementation in C is shown below where 5 elements are added to a queue and then the oldest element is removed from the queue, followed by the second oldest and so on in accordance with FIFO.

```cpp
// C++ program to demonstrate
// working of FIFO
// using Queue interface in C++

#include<bits/stdc++.h>
using namespace std;

// print the elements of queue
void print_queue(queue<int> q)
{
    while (!q.empty())
    {
        cout << q.front() << " ";
        q.pop();
    }
    cout << endl;
}

// Driver code
int main()
{
    queue<int> q ;

    // Adds elements {0, 1, 2, 3, 4} to queue
    for (int i = 0; i < 5; i++)
        q.push(i);

    // Display contents of the queue.
    cout << "Elements of queue-";

    print_queue(q);

    // To remove the head of queue.
    // In this the oldest element '0' will be removed
    int removedele = q.front();
    q.pop();
    cout << "removed element-" << removedele << endl;

    print_queue(q);

    // To view the head of queue
    int head = q.front();
    cout << "head of queue-" << head << endl;

    // Rest all methods of collection interface,
    // Like size and contains can be used with this
    // implementation.
    int size = q.size();
    cout << "Size of queue-" << size;

    return 0;
}
```

Figure 10. FIFO Code Using Integer Elements

VGA

A VGA module is implemented to display the image from the DE1-SoC's on board memory to a VGA monitor. The VGA module from Lab 3 (vga_counters) can be used as a starting point with some modifications if necessary.

```
module vga_counters(
 input logic     clk50, reset,
 output logic [10:0] hcount,  // hcount[10:1] is pixel column
 output logic [9:0]  vcount,  // vcount[9:0] is pixel row
 output logic     VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
 *
 * HCOUNT 1599 0             1279       1599 0
 *             _______________              ________
 * ____________|    Video      |____________|  Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *       _____________________     _____________
 * |____|       VGA_HS          |____|
 */
   // Parameters for hcount
   parameter HACTIVE      = 11'd 1280,
             HFRONT_PORCH = 11'd 32,
             HSYNC        = 11'd 192,
             HBACK_PORCH  = 11'd 96,
             HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC +
                            HBACK_PORCH; // 1600

   // Parameters for vcount
   parameter VACTIVE      = 10'd 480,
             VFRONT_PORCH = 10'd 10,
             VSYNC        = 10'd 2,
             VBACK_PORCH  = 10'd 33,
             VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC +
                            VBACK_PORCH; // 525

   logic endOfLine;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)          hcount <= 0;
     else if (endOfLine) hcount <= 0;
     else                hcount <= hcount + 11'd 1;

   assign endOfLine = hcount == HTOTAL - 1;

   logic endOfField;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)          vcount <= 0;
     else if (endOfLine)
if (endOfField)  vcount <= 0;
     else                vcount <= vcount + 10'd 1;

assign endOfField = vcount == VTOTAL - 1;

   // Horizontal sync: from 0x520 to 0x5DF (0x57F)
   // 101 0010 0000 to 101 1101 1111
```

```
66  assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
67                       !( vcount[9] | (vcount[8:5] == 4'b1111) );
68
69    /* VGA_CLK is 25 MHz
70     *             __    __    __
71     * clk50    __|  |__|  |__|
72     *
73     *            _____       __
74     * hcount[0]__|     |_____|
75     */
76    assign VGA_CLK = hcount[0]; // 25 MHz clock: rising edge sensitive
77
78  endmodule
```

Figure 11. Code for the VGA module

This module performs a rastor scanning algorithm that uses hcount and vcount parameters to display the pixels.

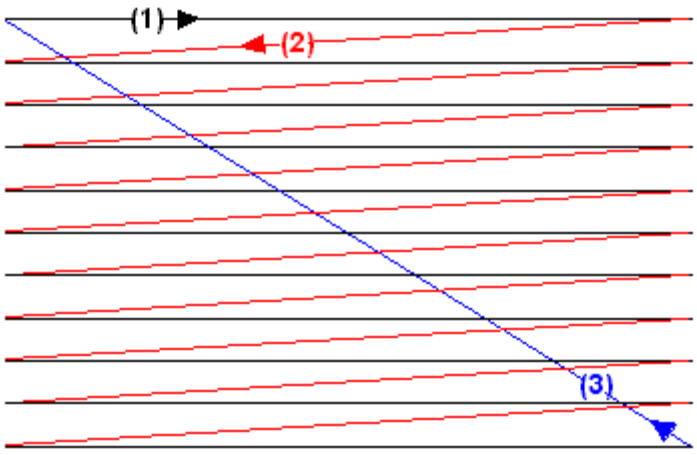

Figure 12. Rastor Scanning Diagram

Resource Budgets

Since the resolution from the camera is going to be 640x480 and each pixel has 16 bits, a single frame of the image will be:

$$16 \times 640 \times 480 = 4915200 \text{ bits/frame}$$

This means that each frame will be 4915 kb in size. The on chip memory of the FPGA is 256kB, or 2048kb. This means that the memory isn't enough to store one full frame.

To get around this, each frame can be split up into 5 horizontal chunks. The first chunk is saved on the asyn_fifo before being saved to the SD card. After the first chunk has been saved, the second chunk is then saved and so on. This way, instead of having to save 4915 kb to a 2048 kb memory, we are saving 983 kb. The code for this is shown in the algorithms section.

The Hardware/Software Interface

OV7670 Video Camera

In order to attain a live feed from the camera onto the VGA monitor, the camera is controlled using the SCCB(Serial Camera Control Bus) protocol which is a variation of the I2C communication protocol. A I2C driver in HDL is needed to establish communication between the camera and the FPGA. The advantage of SCCB over I2C in this application is the fact that no pull up resistors are required as the SCCB can produce all three states (high, low and high impedance), whereas I2C can only produce either low or high impedance.

In order to initialize the camera module, values have to be assigned to each of the control registers of the camera. Pins D0-D7 (8 bits) represent the pixel data from the camera in parallel. The pixel data is synchronized to the falling edge of the PCLK and, since each pixel is 16 bits (RGB565 -> 5+6+5=16), two cycles of PCLK are needed to get a single pixel. The PCLK needs a reference clock, which is where the input XCLK comes into play. To maintain the resolution of 640x480, the XCLK has to be 25MHz.

```
    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; //MTX2
    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, does not preserve reserved bits, [redacted]
    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
```

```
//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
```

Figure 13. Initializing Control Registers for the Camera

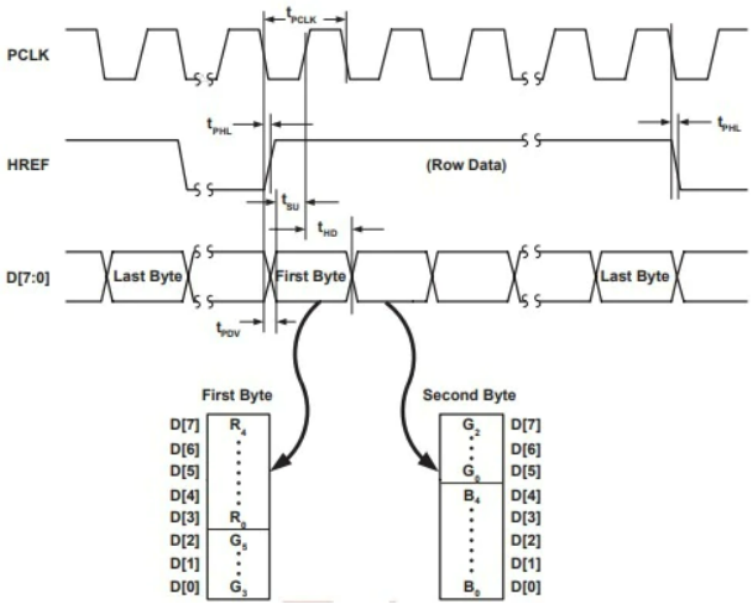

Figure 14. Pixel Data Timing Diagram

An asynchronous FIFO module would have to be implemented to act as a buffer for the pixel data coming from the camera.

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. Some of the main advantages of this sensor is that it can work with a voltage supply from 5V to 20V and 65 mA according to the data sheet. Another advantage is that it produces a digital output. High when motion is detected and low when it is idle. 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 the Fresnel lenses help to focus more light into the pyroelectric sensor but also helps to increase the range. So the sensor detectivity 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 one is a single trigger mode and the second one is 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 an input and connect the output of the sensor. Then, the power will be supplied through the VCC5 pin onboard.

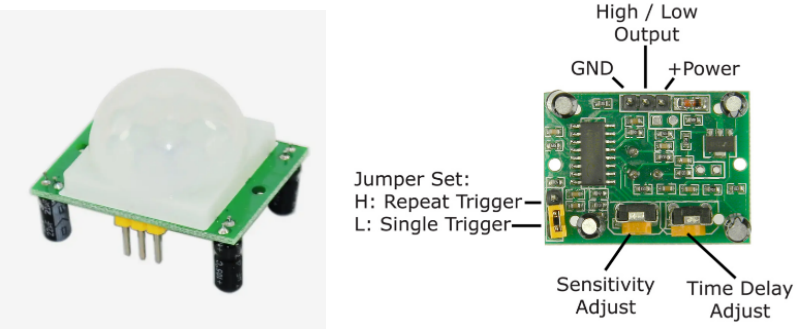

Figure 15. HC-SR501 PIR sensor

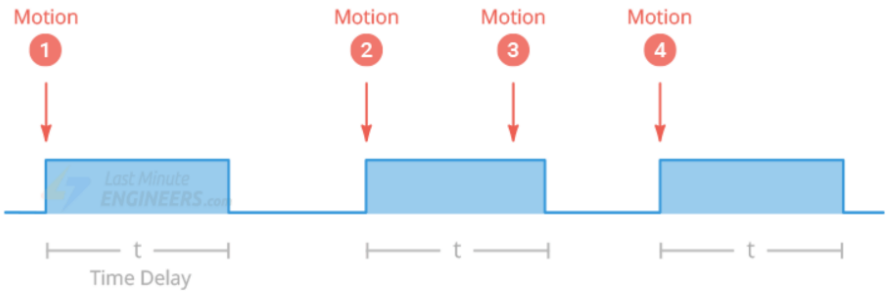

Figure 16. Single Trigger Mode Detection.

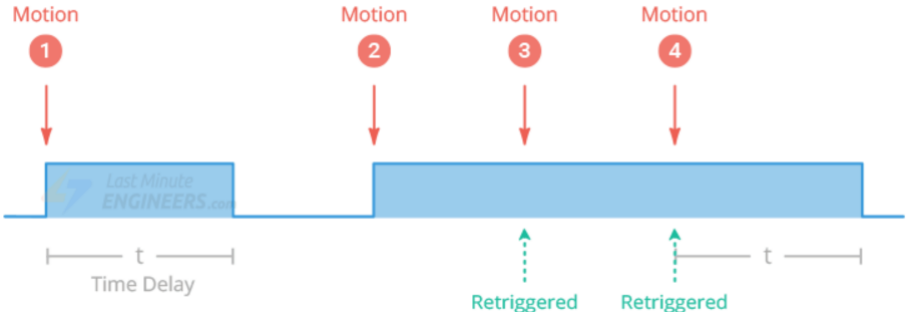

Figure 17. Multiple Trigger Mode Detection

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 16. Shows the connections of 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.

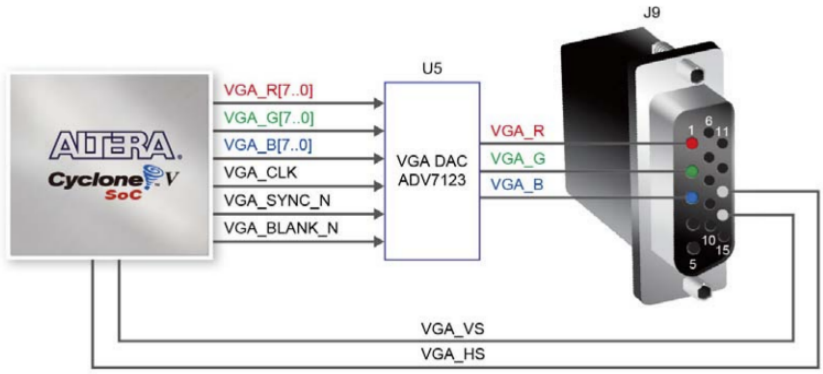

Figure 18. Connections between FPGA and VGA

SD Card

The board supports Micro SD card interface with x4 data lines. It serves not only as an external storage for the HPS, but also as an alternative boot option for the DE1-SoC board.

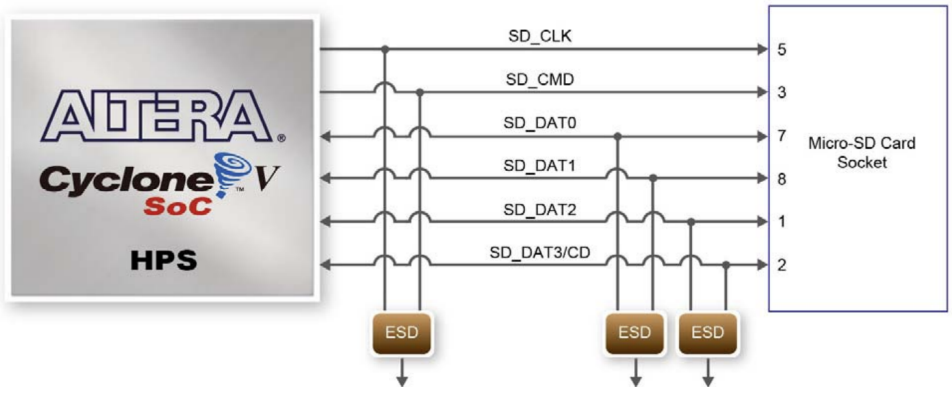

Figure 19. Connections between HPS and the Micro SD Card socket.

A total of 6 pins of the HPS are dedicated to interfaces the SD card socket, 4 bits for data, 1 bit for clock and 1 bit for command line.

Signal Name	*FPGA Pin No.*	*Description*	*I/O Standard*
HPS_SD_CLK	PIN_A16	HPS SD Clock	3.3V
HPS_SD_CMD	PIN_F18	HPS SD Command Line	3.3V
HPS_SD_DATA[0]	PIN_G18	HPS SD Data[0]	3.3V
HPS_SD_DATA[1]	PIN_C17	HPS SD Data[1]	3.3V
HPS_SD_DATA[2]	PIN_D17	HPS SD Data[2]	3.3V
HPS_SD_DATA[3]	PIN_B16	HPS SD Data[3]	3.3V

Figure 20. Pin assignment for SD card socket

References

1. https://lastminuteengineers.com/pir-sensor-arduino-tutorial/
2. https://www.mpja.com/download/31227sc.pdf
3. https://community.element14.com/challenges-projects/design-challenges/summer-of-fpga/b/blog/posts/security-camera-3-interfacing-with-ov7670-camera
4. https://www.ti.com/lit/ml/slva704/slva704.pdf?ts=1680108161191&ref_url=https%253A%252F%252Fwww.google.com%252F
5. https://www.waveshare.com/w/upload/1/14/OmniVision_Technologies_Seril_Camera_Control_Bus%28SCCB%29_Specification.pdf
6. DE1-SoC_User_manual_pdf