

Real-time detection of the position and orientation of a piece of paper using video feed

Ali Ghajarnia
Radu Raduta

TA: HyungBin Son
6.111 – Introductory Digital Systems Laboratory
December 9, 2004

Abstract

The aim was to develop a system that could determine the position and orientation of a piece of paper by using a video feed to compute the coordinates of the corners. Currently, the output of the system is in the form of a VGA display with an overlay at the detected corners. We would like to eventually improve on the accuracy and sophistication of the system and eventually add a motor controller that points a laser at the paper, creating a hybrid display consisting of a dynamic overlay superimposed on a static background.

Table of Contents

Overview.....	5
Description	5
Memory Timings.....	5
External components.....	6
Components implemented within FPGA.....	6
Memory Manager - Ali.....	7
Composite In - Ali, Radu.....	8
Static Image Generator - Ali.....	8
Camera Input - Ali, Radu.....	9
VGA out - Ali, Radu.....	11
Threshold select - Radu.....	12
Hex display - Radu.....	13
Edge follow – Radu.....	13
Motor control - Radu.....	13
Distribution of Work.....	14
Testing and Debugging.....	14
Conclusions	14
Appendix.....	16
Memmanager.v.....	16
Composite_in.v.....	17
Vga_out.v.....	20
Threshold_select.v.....	23
Hex_display.v.....	24
Edgefollow.v.....	25
Syncgen.v.....	30
Input Wiring Diagram.....	32

Index of Figures

Figure 1 - Detailed Block Diagram	7
Figure 2 - AD775 Timings	9
Figure 3 - Horizontal Sync Timing	10
Figure 4 - Input Wiring Diagram	32

Index of Tables

Table 1 - Memory access partitioning	6
Table 2 - Static Image Coordinates	8
Table 3 - VGA Horizontal Timing (640x480 60Hz)	12
Table 4 - VGA Vertical Timing (640x480 60Hz)	12

Overview

The main goal of the system was to determine the position and orientation of a piece of paper by applying a corner detection algorithm an incoming video feed. The incoming frames are stored in on-board RAM, where it can be accessed by the corner detection module. For developing, testing and debugging the corner detection algorithm, a video output module was required to display the image at different stages of processing.

The system was implemented on a Xilinx Spartan-3 Starter Kit development board, containing a 200,000 gate-equivalent Spartan-3 FPGA, 2 256x16 10ns memory chips, as well as a range of buttons, indicators and connectors. The FPGA was clocked using the on-board 50Mhz crystal oscillator. Development was done in Verilog using the bundled Xilinx ISE WebPack integrated development environment.

Description

Memory Timings

The system makes use of the on-board 10ns RAM found on the Spartan-3 Starter Board. The memory was configured as a single 256x16 memory space. Initially we envisioned the system containing a multitude of blocks requiring access to the memory, and thus we decided on a time-sharing method for partitioning access to the memory. Based on the timing of the input, memory access time was divided into 16 8-clock cycle partitions. Each module can access the memory during its allotted single or multiple partition.

A timing bus was distributed throughout the system allowing the different modules to determine if they can make use of the memory. Initially a 4-bit bus was distributed identifying the current partition ID, but it was later recognized that for modules involving complex state (such as the edge detection module), it was necessary to have available more detailed timing information to avoid starting a memory transaction just before access is transitioned to another module. Thus a full 7-bit 50Mhz counter was exposed to the system, with the top 4 bits indicating the current partition, and the lower 3 bits indicating how far along the 8-clock cycle partition has progressed. The timing is generated by the composite_in module, and all other modules must be capable of synchronizing their memory access accordingly.

A memory manager module rewires the memory interface based on the current partition, exposing a simple virtual memory interface to each of the modules, which includes an address bus, data bus, as well as the appropriate output enable or write enable signals.

The current slice assignments for memory access are show in Table 1.

The memory is used to store 4800 16-bit words, representing a 320x240 black-and-white image grouped as 16 pixels/word. A line will contain 20 such blocks, and there are a total of 240 lines. The image is stored in memory starting at address 0.

Table 1 - Memory access partitioning

<i>Partition ID</i>	<i>0x0</i>	<i>0x1</i>	<i>0x2</i>	<i>0x3</i>	<i>0x4 - 0xE</i>	<i>0xF</i>
active module	vga_out	unassigned	edge_follow	edge_follow	unassigned	composite_in

External components

Grayscale NTSC Camera

This camera delivers a 60 hertz grayscale interlaced NTSC composite signal. A 75 ohms termination resistor is applied and the signal is fed to the GS4981 and the AD775.

Gennum GS4981

This monolithic sync separator outputs vertical and horizontal syncs from a composite input.

AD775

This high speed 8 bit sampling analog to digital converter digitizes luminescence values from the camera that are output to the FPGA.

VGA Monitor

A standard VGA monitor connects to the FPGA to display contents of video as it is read from SRAM

Components implemented within FPGA

The diagram of the functional blocks implemented on the FPGA are shown in Figure 1. The functionality and implementation details for each functional block are discussed in the following sections.

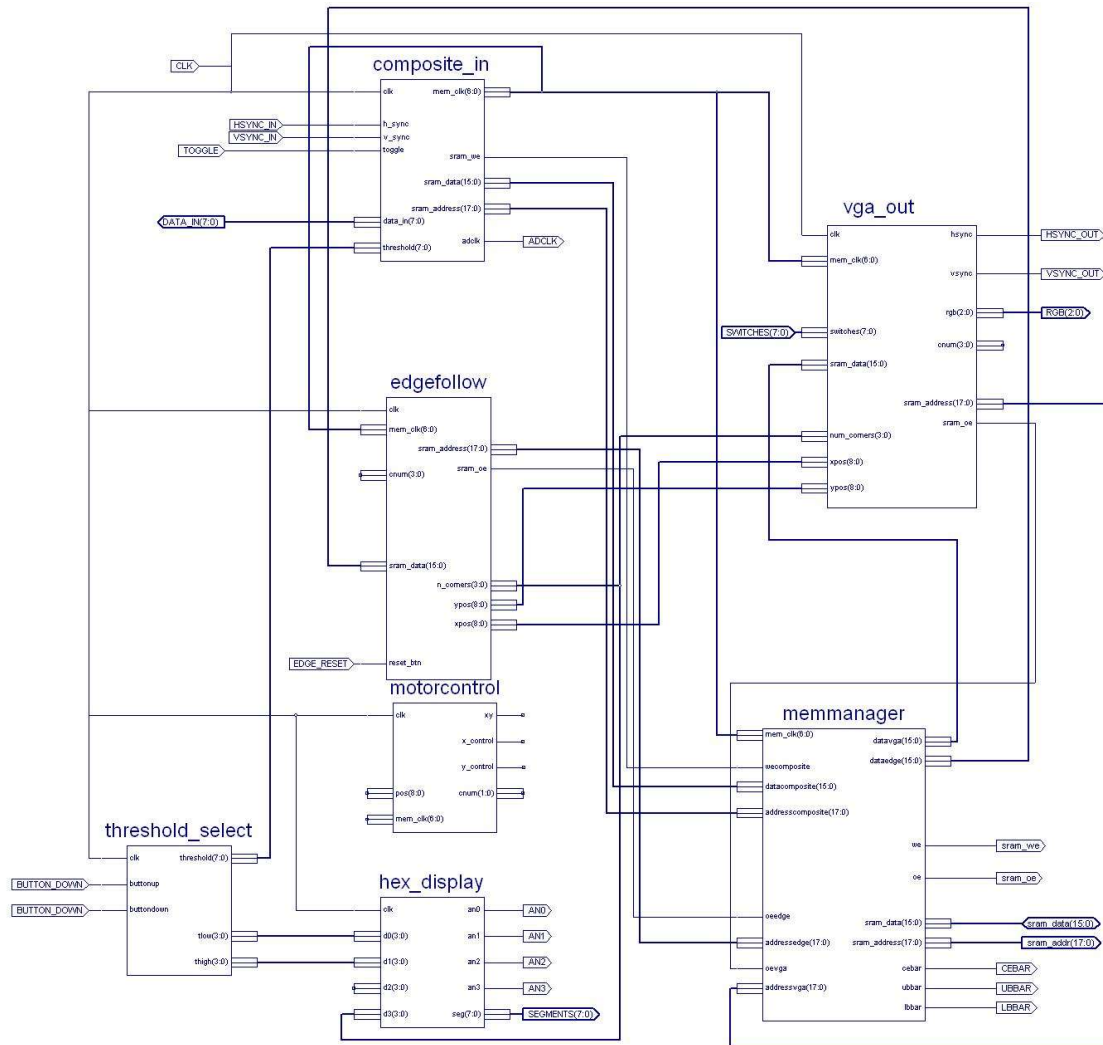


Figure 1 - Detailed Block Diagram

Memory Manager - Ali

This module interfaces with the SRAM directly, and all blocks that need memory access interface through this block. The video memory is stored 16 pixels at a time. This means that the data bus going to the SRAM is 16 bits wide. The address bus is 18 bits, but the five most significant bits of the address are tied to ground since we only need 13 bits to write to 4800 different SRAM address to result in a memory of 320x240 pixels. This represents one frame of video at the resolution with which we are working. The pixel clock used in the component_in block samples the input every 8 clock cycles, so it was decided that it would be appropriate to partition memory access into blocks of 8 clock cycles. Since data must be written once every 16 pixels and each pixel is 8 clock cycles, the memory manager divides memory access into 16 blocks of 8 cycles. Each module would then be allowed to access memory in a round-robin fashion during one of these 16 blocks which would be dedicated to that module alone. The advantage of this is

that there is a synchronized way to make sure no modules attempt memory access at the same time. Also, this means that without modifying our design, we could easily add up to 16 modules which need access to the same memory. Under the current implementation, the component_in block access memory when the pixel count reaches the last pixel (pixel count = 15) and the VGA_out block accesses memory on the pixel count of zero. The other 14 are reserved for the edge following module and any other modules that could be added in the future. The memory manager itself is composed entirely of combinational logic. Its inputs are the address buses, data buses, we, and oe of each of the modules that use it. It also has the global pixel counter input which partitions access time. Using this last variable, it switches pathways between the SRAM and each of the modules depending on which module's turn it is to access memory.

Composite In - Ali, Radu

This module is responsible for storing image data in the SRAM for analysis and display purposes. It also generates the global counter *mem_clk* which is used to control memory access by the different block. It is a seven bit variable whose value increases by one each clock cycle and rolls over to zero again after its largest value. The three least significant bits allow eight steps for reading or writing to SRAM and divides the clock to 6.125 MHz so that it can sample video at the appropriate rate. The four most significant bits create a cycle of sixteen stages. In each of these stages one module gets access to the SRAM. Also, in each of these stages, a new pixel from either the camera feed or the generated image is being stored in a buffer so that when it comes time for component_in to write to the SRAM, it will have a block of sixteen pixels available.

One of the inputs to this module is a toggle button. This button switches the image source between generating a predefined static image or grabbing video data from the NTSC camera.

Static Image Generator - Ali

When the toggle button is switched on, the image being written to SRAM takes the form of a tilted rectangle. The image is supposed to represent the video input of the piece of paper until more sophistication can be built into the system. The purpose for this image is to test the VGA_out module to make sure that information is being correctly written and retreated and displayed to the monitor. It's primary function however is to be a base case for the edgefollow module to operate on. The system first attempts to find the corners of an image with no noise or movement and then can be tested on a moving image from the camera. The image "border" is an the OR of the four line segments which make it up. The x and y coordinates of the four points are designated in the following table

Table 2 - Static Image Coordinates

X1	X2	X3	X4	Y1	Y2	Y3	Y4
18	300	290	18	4	25	221	200

These coordinates correspond to a 320x240 grid with the origin in the upper left corner. The four lines were represented in point-slope form as a function of the current x and y position of the pixel being written to a buffer called “recent” which is written to SRAM after the sixteenth pixel. The equations for the lines take the form of $((py-Y1)/2 == (ma*(px-X1)/8192))$ after running a check to see if px and py are in the correct range. In other words, if you have a line segment connecting two points, you don’t want the x-coordinate of any point to be greater than the x-coordinate of the rightmost point or less than the x-coordinate the leftmost point. The same reasoning applies to the vertical dimension. A division by two on both sides of the equation causes a shift right that chops off the least significant bit. This gives the border a width of two in the vertical dimension. The slope is precalculated to be $ma/4096$. The reason for the division by a large power of two is to avoid having to multiply or divide by a decimal value. Ma is the slope scaled by 4096, and the division causes a shift right which gives an approximation that is accurate to the nearest pixel. For the two lines that have very large slopes, the process is repeated by expressing the equation as a comparison between x and a function of y. The reason for this switch in convention is that we can use the division by two like before, but now it gives a width of two in the horizontal dimension. For each of the four line segments, the ordering of the terms changes around slightly to avoid negative numbers. Although the image does not change, the SRAM is still constantly updated as though it were a true video feed, which brings us to the next part of this module.

Camera Input - Ali, Radu

The composite_in module normally receives signals from the camera to store to SRAM for processing. These signals include eight data bits which carry luminescence information, as well as two bits for vertical syncs and horizontal syncs from the camera. The two syncs are extracted from the composite signal and fed to this module by the GS4981. The eight data bits are provided by the AD775 Analog to Digital converter, which is the modules clocked at 6.125 MHz. Data is read from this chip as soon as its clock goes low, this ensures sampling in the center of a data point.

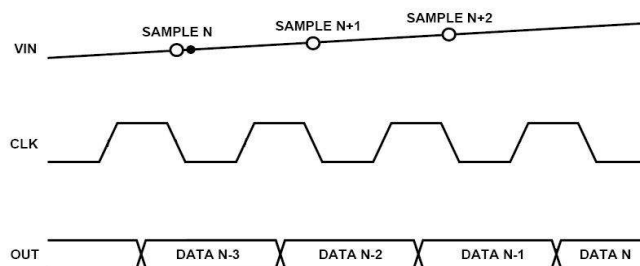


Figure 2 - AD775 Timings

This eight bit data is compared with an eight bit adjustable threshold which is the output of the threshold_select module. The result, a black and white pixel derived with an adjustable threshold, is stored to a buffer every eight clock cycles that the camera signal represents active video. Whether or not this is true is determined by the vertical and

horizontal syncs. For each, a register is dedicated to remembering the value of the sync signals in the last clock cycle. When this register is low and the current signal is low, this marks the first clock cycle after the positive edge of the sync signal. The positive edge of the vertical sync represents the start of a frame and the positive edge of the horizontal sync represents the start of a line. Now let's examine the horizontal sync more closely.

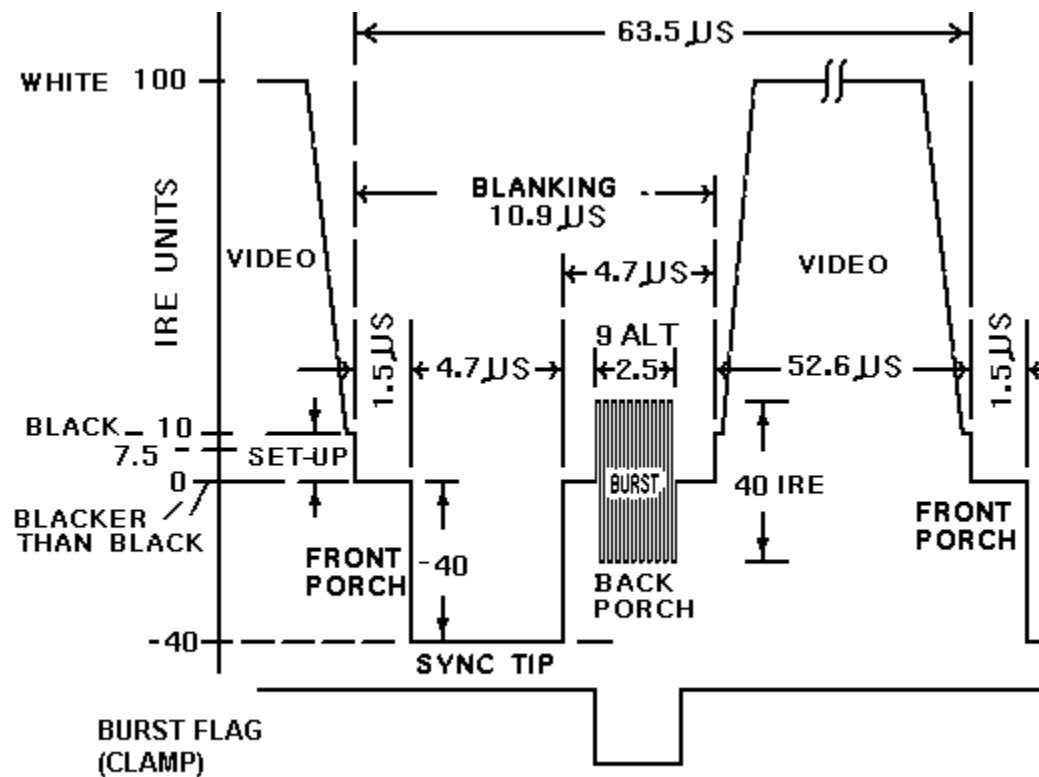


Figure 3 - Horizontal Sync Timing

According to the NTSC specifications, which were confirmed using an oscilloscope, the start of the active video region for each line is 4.7 microseconds after the rise of the hsync. At 6.125 MHz, which corresponds to 8 cycles of our 50 MHz clock, this corresponds to 28.8 cycles. This means that if we start incrementing a counter named hcount every eight clock cycles after the rise of the hsync, we have to wait at least until the counter reaches 29 before we can consider the data to be valid pixel information. The video time of 52.6 microseconds occurs during an interval of about 322 increments of hcount. This is perfect for our desired horizontal resolution of 320 pixels. We simply crop out the couple of pixels at the edges. In fact, this is the reason that we chose a clock divider of eight on a 50 MHz clock. Vcount is a counter which increments whenever hcount comes to the beginning of a new line. Since there are 262.5 lines per field and we only need 240 lines per frame, we consider the video region to be when vcount is between 15 and 254. Also, hcount must be between 31 and 350. In order to reference pixels

correctly, we do not consider them to be at coordinates of hcount and vcount. Instead we consider xcor and ycor which are basically (hcount – 31) and (vcount – 15) respectively. This way, the first pixel will be at the origin of xcor and ycor. These values are used to reset or increment the SRAM address.

As video data comes in, it gets stored in a 16 bit register named vid_buffer. The position within vid_buffer that each pixel gets written to is determined by the four least significant bits of xcor. When the four least significant bits of xcor are all ones, the buffer has filled up and it copies itself into another register named vid_bufferold. Also, the SRAM address will increase by one at this point, unless the xcor is 15 and the ycor is 0. If this is the case, the SRAM address is reset to zero. This system ensures that there will always be an SRAM address and sixteen consecutive values in vid_bufferold which correspond to that SRAM address. Also, the mapping of the SRAM address and the position on the camera image is constant. Addresses zero through 19 represent the first line of 320 pixels. Addresses 20 through 39 represent the 320 pixels on the second line and so forth.

Both even and odd fields were used from the camera. This causes at most a wobble of one pixel up and down but this was not a major concern. The benefit of using both even and odd fields is that it simplifies the code and doubles the update rate to 60 Hz.

VGA out - Ali, Radu

The VGA_out module displays the contents of the SRAM to VGA monitor with an overlay of red point coordinates from the edgefollow module. The output is a 3-bit per pixel, 60 frames per second image stream at a resolution of 640x480. This is four times the resolution of the data we've stored to SRAM meaning each pixel must be displayed twice per line and in turn each line must be shown twice.

The output is clocked according to the VGA timing specifications. A 60Hz 640x480 display requires a 25MHz clock, which is half of the FPGA clock frequency. The VGA output module uses a horizontal pixel counter and a vertical line counter to produce the appropriate signal timings.

Vertical timing is controlled through the value of the register vcount. The register is incremented each time a new video line is started. Out of the total 573 lines of a frame, lines 0-479 correspond the active video region, followed by the front porch (lines 480-492), the vertical sync area (lines 493-494) and the back porch (lines 495-572).

The horizontal timing is slightly more complex since it must be synchronized with the memory access division clock. The horizontal counter is normally incremented at 25Mhz (half of the system clock). Since producing a sync signal commits us to outputting a video line, the sync signal is delayed until the beginning of the line will happen just after we have read the first block of data pixels from memory. Thus each active line starts with the sync signal (pixels 0-96), followed by the back porch (pixels 97-142), and the active video region (pixels 143-783). After the active video region is displayed, the block enters the front porch area, waiting until the memory clock indicates a value of 97. From this point, we know that we would have just completed a memory read for the first 32 pixels of data in exactly 248 clock cycles (just as we enter the next active video region when hsync reaches a value of 143).

Table 2 :

Table 3 - VGA Horizontal Timing (640x480 60Hz)

Scanline time	31.77 μ s
Sync pulse length	3.77 μ s
Back porch	1.89 μ s
Active video time	25.17 μ s
Front porch	0.94 μ s

Table 4 - VGA Vertical Timing (640x480 60Hz)

Total frame time	16.68 ms
Sync length	0.06 ms
Back porch	1.02 ms
Active video time	15.25 ms
Front porch	0.35 ms

The memory access occurs when the on memory partition zero, as indicated by the 4 most significant bits of the input *mem_clk*. Memory is accessible only every 128 sytem clock cycles, and a 320x240 output on a 640x480 screen requires that a different pixel be output every 4 clock cycles. Thus during the eight clock cycle memory access period, two reads are performed resulting in a buffer containing the next 32 pixels to be displayed.

The module also queries the edgefollow module for a set of up to eight x/y coordinate pairs representing pixels identified as corners. The querying is done on line 480, which is just out of the active video region. The input *num_corners* indicates the number of corners identified that should be displayed. For each active pixel up to the number of dots to be displayed, a new x/y coordinate pair is queried from the edgefollow module by setting the corner number *cnum* and reading the inputs *xpos* and *ypos*. To speed up processing during the display phase, a mask is created indicating how many dots should be displayed. For each pixel displayed in the next frame, the x and y coordinates are compared with each of the coordinates of the polled corners. If there is a match, a red overlay is displayed at the appropriate point.

Threshold select - Radu

Since the threshold of distinguishing black from white may vary with lighting conditions, this module was developed as a convenient way to select the threshold level. Its input is from two buttons. This module runs with a large 22 bit clock divider. Once, on every cycle of the larger clock, it checks to see if either of the buttons is pressed. One button causes the threshold value to increase and the other causes it to decrease. When both are pressed at the same time, the threshold resets to 95. The threshold value is outputted to the *composite_in* module, which uses it to discriminate between black and white, and to the *hex_display* module, which outputs the value of the threshold on two seven-segment displays.

Hex display - Radu

The purpose of this module is to provide a means of easily viewing some detail about the status of the system. This is a useful module to have around for debugging purposes. It controls the four seven-segment LED displays on the Xilinx board. A four bit input to any of these displays is shown in hexadecimal. Currently, the first two displays report the value of the threshold, and another one reports the `n_corners` output of the `edgfollow` module.

Edge follow – Radu

The edge following module represents the core of the system, and at the same time the most uncertain component in terms of feasibility. A number of algorithm ideas were developed, yet none of the implementations proved satisfactory results on real-world data. The complexity of the algorithm could not be increased without producing very slow or very large logic.

In the current implementation, the module performs simple a simple algorithm to determine slope changes of black lines in the input. A simple case is assumed where the only object in the input is a white piece of paper with a black border. For each line of the input, the module computes the centers of all black regions. From line to line, the change in the x coordinate of the center of each black region is stored as a slope. Whenever a new edge is encountered, or when the slope changes by more than a certain threshold, the coordinates of the point are stored as a new corner.

Other implementations included recording extremes coordinates for the centers of the borders, and keeping track of positive and negative sloping lines separately and recording starting and ending coordinates. None of these proved to be more accurate using live video data.

The module reads data stored in RAM in 16-pixel blocks during its allotted memory access time. It also provides an interface for querying the location of the corners, through the input `c_num` which selects the corner queried, and the outputs `x_pos` and `y_pos` which are assigned to the appropriate x/y coordinate pair.

This module would have benefited from the implementation of a microprocessor core in order to allow the complex computations required for properly filtering and classifying the slope data. Whereas the edge detection needs to run at high speed and is suited for being implemented in digital logic, processing of line and frame information requires more complex algorithms than can be efficiently encoded directly in logic.

Motor control - Radu

This module produces a pair of PWM signals to be fed to a servo pair which are meant to control the direction of a laser pointer. The signals produced are standard RC servo controller signals, with a duty period of 1-2ms and a cycle period of 10ms.

Distribution of Work

Despite the modularity of this system, Ali and Radu worked together a lot. Ali focused primarily on `composite_in`, memory manager, `vga_out`, and external components, getting the timings and signals correct to access SRAM and display video. Radu spent some time on these parts as well, but also worked on `threshold_select`, `hex_display`, and `edgefollow`.

Testing and Debugging

We attempted to build our project in a way that would make testing easier. For instance, we used the threshold module to see the effect of modifying different variables. The hexadecimal display module gave us the ability to track the values of our counters and check for various events. When Ali was having trouble getting the `composite_in` to work with the video camera sync signals, he narrowed down the problem to corrupted sync signals from the external components by using the `syncgen.v` module which simulates proper `hsync` and `vsync` signals from the camera. He used these simulated syncs with a generated image and saw that the timings and writes to the SRAM were correct. As it turns out, a frequent problem was the AD775 burning out despite being wired according to specifications. A resistor was added to current limit the supply voltage. Radu meticulously rewired the circuit when some elements of noise were noticed. Also, the generated box image serves as a simplified situation in which the edge following could take place, but the overall scheme of debugging was considering assumptions we were making in each module and figuring out ways to test each assumption to see if it was in fact true. Once the `vga_out` module was able to properly read SRAM data, we were able to output conditionals such as `(h_count > 320)` or whatever else we needed. Nonetheless, the problems we encountered turned out to take a very long time to debug, and as a result we did not make as much progress as we would have liked. Also, a consequence of modifying each others code was that we would often spend a lot of time trying to fix a problem that the other had just fixed.

Conclusions

The project unfolded more like a scientific experiment than we would have expected. The nature of the incoming video was hard to predict, and thus the complexity of the edge detection algorithm needed was unknown. The system had to be over-design with respect to the number of modules which could access the memory, as we envisioned multiple modules being needed for pre-processing of the data, as well as performing different methods of corner/edge detection simultaneously. This complicated the memory interface for all of the parts, and in the end only three modules ended up being interfaced with the memory. For example, a blurring module was not needed since the black/white image obtained with the camera slightly out of focus resulted in an excellent noise-free image.

The initial assumption the required edge/corner detection algorithms could be simply implemented just as digital logic proved to be false. Even with the 200,000 gate-equivalents available on the Spartan-3 board, the logic resulting even for the simple algorithm presented was large (over 120,000 gates). A simple microprocessor core could make use of edge coordinates and widths calculated in logic to perform more complex calculation and determine corner coordinates much more accurately, allow for simpler and more flexible reprogramming, all with a smaller logic footprint.

Appendix

Memmanager.v

```
module memmanager(we, oe, sram_address, sram_data,
addressvga,datavg22a,dataacomposite,addresscomposite,mem_clk,
                addressedge,dataedge,oevga,wecomposite,oeedge,
cebar, ubbar, lbbar);

    output we;
    output oe;
    output [17:0] sram_address;
    inout [15:0] sram_data;

    input [17:0] addressvga;
    input [17:0] addresscomposite;    // for vga data connect to
sram_data
    input [17:0] addressedge;

    input [15:0] dataacomposite;
    output [15:0] datavg;
    output [15:0] dataedge;

    output lbbar, ubbar, cebar;

    assign cebar = 0;
    assign lbbar = 0;
    assign ubbar = 0;

    input [6:0] mem_clk;

    wire [3:0] pixelcount;
    assign pixelcount = mem_clk[6:3];

    // for edge data connect to sram_data

    input wecomposite;
    input oevga;
    input oeedge;

    //THIS BLOCK USES PIXELCOUNT TO PARTITION ACCESS TO SRAM BETWEEN
    //DIFFERENT BLOCKS.
    //THIS FILE IS CURRENTLY SET UP FOR TWO BLOCKS WHICH WILL READ FROM
    //SRAM AND ONE BLOCK WHICH WILL WRITE TO SRAM.

    wire vga_time, composite_time, edge_time;

    assign vga_time = (pixelcount == 0);
    assign composite_time = (pixelcount == 15);
    assign edge_time = (~vga_time && ~composite_time);

    assign we = ~(composite_time ? wecomposite : 0);
    assign oe = ~(vga_time ? oevga : edge_time ? oeedge : 0);
```



```

        assign sram_address = vga_time ? addressvga : composite_time ?
addresscomposite : addressedge;

        assign sram_data = composite_time ? datacomposite : 16'hz;

        assign datavga = vga_time ? sram_data : 0;
        assign dataedge = edge_time ? sram_data : 0;

endmodule

```

Composite_in.v

```

module composite_in(
    clk,
    mem_clk,
    sram_we, sram_data, sram_address, h_sync, v_sync, toggle, data_in,
    threshold, adclk);

    input clk;
    input h_sync;
    input v_sync;
    input toggle;
    input [7:0] data_in;
    input [7:0] threshold;
    output [6:0] mem_clk;
    output sram_we;
    output [15:0] sram_data;
    output [17:0] sram_address;
    output adclk;

    wire data;
    reg vs;
    reg hs;
    wire startframe;
    wire startline;
    reg video;
    reg line;
    reg [10:0] hcount;
    reg [10:0] vcount;
    wire [9:0] xcor;
    wire [9:0] ycor;
    wire en;
    reg [2:0] pixclk;
    reg oddeven;
    reg [15:0] vid_buffer;
    reg [15:0] vid_bufferold;
    reg [17:0] vid_sram_address;

    assign data = (data_in > threshold);

```

```

reg [6:0] mem_clk;

wire [3:0] pixel_count;
wire [2:0] div;
assign pixel_count = mem_clk[6:3];
assign div = mem_clk[2:0];

//*****
//This section generates a box image to store to sram

reg border,linea,lineb,linec,lined,diagonal;
wire [8:0] px;
wire [7:0] py;

wire [8:0] X1, X2, X3, X4;
wire [7:0] Y1, Y2, Y3, Y4;
reg [7:0] ycount;
reg [4:0] xcount;
wire [8:0] ma;
wire [16:0] mb;

assign py = ycount;
assign px = {xcount,pixel_count};

assign X1 = 18;
assign X2 = 300;
assign X3 = 290;
assign X4 = 8;
assign Y1 = 4;
assign Y2 = 25;
assign Y3 = 221;
assign Y4 = 200;
assign ma = 305;
assign mb = 209;

always @(posedge clk) begin
    linea <= ((Y1-2 <= py)&(py <= Y2-2)&(X1-2 <= px)&(px <= X2+2)) ?
    ((py-Y1)/2 == (ma*(px-X1)/8192)) : 0;
    lineb <= ((Y2-2 <= py)&(py <= Y3-2)&(X2+2 >= px)&(px >= X3-2)) ?
    ((X2-px)/2 == (mb*(py-Y2)/8192)) : 0;
    linec <= ((Y3+2 >= py)&(py >= Y4+2)&(X3+2 >= px)&(px >= X4-2)) ?
    ((Y3-py)/2 == (ma*(X3-px)/8192)) : 0;
    lined <= ((Y4+2 >= py)&(py >= Y1+2)&(X4-2 <= px)&(px <= X1+2)) ?
    ((px-X4)/2 == (mb*(Y4-py)/8192)) : 0;
    border <= (linea || lineb || linec || lined);

    diagonal <= ((px == py) || (px == 320 - py)); //This is another
possible test output.
end

//*****

reg sram_we;

```

```

reg [17:0] sram_address;
reg [15:0] sram_data;

reg [15:0] recent;

assign buffer_full = (pixel_count == 15);

always @(posedge clk) mem_clk <= mem_clk + 1;

always @(posedge clk) begin
    if (xcount == 20)
        ycount <= (ycount == 240) ? 0 : ycount + 1;

    if (div == 2)
        if (toggle) recent[pixel_count] <= border;
        else if (buffer_full) recent <= vid_bufferold;

    if (buffer_full)
        case (div)
            2: begin
                sram_address <= toggle ? (py*20 + px[8:4]) :
vid_sram_address;
                xcount <= (xcount == 20) ? 0 : xcount + 1;
            end
            4: sram_data <= recent;
            5: if (buffer_full)
                sram_we <= 1;
            7: if (buffer_full)
                sram_we <= 0;

        endcase

end

always @(posedge clk) begin
    vs <= v_sync;
    hs <= h_sync;

    if (video & line & en) begin
        if (xcor[3:0] == 'b1111) begin
            vid_bufferold[15:0] <= {data, vid_buffer[14:0]};
            if ((ycor[9:0] == 0) & (xcor[9:4] == 0))
                vid_sram_address <= 0;

            else
                vid_sram_address <= vid_sram_address + 1;

        end
        else
            vid_buffer[xcor[3:0]] <= data;
    end
end

```

```

    pixclk <= startline ? 0 : pixclk + 1;

    hcount <= startline ? 0 : (en ? hcount + 1 : hcount) ;
    vcount <= startframe ? 0 : (startline ? vcount + 1 : vcount);

    video <= (vcount > 14) & (vcount < 255);
    line <= (hcount > 30) & (hcount < 351);
end

assign en = (pixclk == 0);

assign startframe = ( v_sync & ~vs); // positive edge of vsync
assign startline = (h_sync & ~hs); // positive edge of hsync

assign xcor = line ? (hcount - 31) : 0;
assign ycor = video ? (vcount - 15) : 0;

assign adclk = pixclk[2];

endmodule

```

Vga_out.v

```

module vga_out(clk, mem_clk,
               switches,
               hsync,vsync,rgb,
               sram_oe,sram_data,sram_address,
               cnum, num_corners, xpos, ypos);

    input clk;           // 50Mhz

    input [7:0] switches;

    output hsync;
    output vsync;
    output [2:0] rgb;

    output [3:0] cnum;

    reg [3:0] cnum;

    input [3:0] num_corners;

    input [8:0] xpos;
    input [8:0] ypos;

    input [6:0] mem_clk;

    output sram_oe;
    input [15:0] sram_data;
    output [17:0] sram_address; //0-14

    reg [17:0] sram_address;
    reg sram_oe;

```

```

reg [31:0] sram_pixels;

//*****
//*****
//***
//*** Sync and Blanking Signals
//***
//*****
//*****

reg hsync, vsync, hblank, vblank;

reg [9:0] hcount; // pixel number on current line
reg [9:0] vcount; // line number

//*****
//*****
//***
//*** Pixel logic
//***
//*****
//*****

wire en;
wire [3:0] pixel_count;
wire read_cycle;

assign en = mem_clk[0];
assign pixel_count = mem_clk[6:3];
assign read_cycle = (pixel_count == 0);

wire hblankon, hblankoff, hsyncoff, hreset;

assign hblankon = en & (hcount == 783);
assign hblankoff = en & (hcount == 142);
assign hsyncoff = en & (hcount == 96);
assign hreset = (mem_clk == 97) & (hcount > 783 );

// vertical: 528 lines = 16.77us
// display 480 lines

wire vsyncon, vsyncoff, vreset, vblankon;

assign vblankon = hreset & (vcount == 479);
assign vsyncon = hreset & (vcount == 492);
assign vsyncoff = hreset & (vcount == 494);
assign vreset = hreset & (vcount == 527);

wire [2:0] div;
assign div = mem_clk[2:0];

wire [9:0] hpos;
assign hpos = hcount - 143;

```

```

reg nextpixel;

always @(posedge clk) begin
    hcount <= hreset ? 0 : (en? hcount + 1 : hcount);

    hblank <= hblankoff ? 0 : hblankon? 1 : hblank;
    hsync <= hreset ? 0 : hsyncoff? 1 : hsync;    // hsync is
active low

    vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
    vblank <= vreset ? 0 : vblankon ? 1 : vblank;
    vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;    // vsync is active
low

    if (read_cycle)
        case(div)
            1:begin
                sram_address <= vcount[9:1]*20 + hcount[9:5] - 4;
                sram_oe <= 1;
            end
            2:begin
                nextpixel <= sram_data[0];
                sram_pixels[15:0] <= sram_data[15:0];
                sram_address <= sram_address + 1;
            end
            3:begin
                sram_pixels[31:16] <= sram_data[15:0];
                sram_oe <= 0;
            end
        endcase

    if (en & ~hpos[0])
        nextpixel <= sram_pixels[hpos[5:1]];
end

reg dot;

reg [8:0] dotx[7:0];
reg [8:0] doty[7:0];
reg [15:0] dotmask;

wire [8:0] x;
wire [8:0] y;

assign x = hpos[9:1];
assign y = vcount[9:1];

always @(posedge clk) begin
    if (en & ~hpos[0] )
        dot <= ( dotmask[0] & ((x == dotx[0]) & (y == doty[0]))) |
        ( dotmask[1] & ((x == dotx[1]) & (y == doty[1]))) |
        ( dotmask[2] & ((x == dotx[2]) & (y == doty[2]))) |
        ( dotmask[3] & ((x == dotx[3]) & (y == doty[3]))) |
        ( dotmask[4] & ((x == dotx[4]) & (y == doty[4]))) |
        ( dotmask[5] & ((x == dotx[5]) & (y == doty[5]))) |

```

```

        ( dotmask[6] & ((x == dotx[6]) & (y == doty[6]))) |
        ( dotmask[7] & ((x == dotx[7]) & (y == doty[7]))) );

end

reg [2:0]  rgb;
reg [3:0]  num_dots;

always @(posedge clk) begin
    if (vcount == 480) begin
        if (hcount == 0) begin
            num_dots <= 0;
            dotmask <= 0;
        end
        else
            if ( (num_corners != 0 ) && (num_dots < {0,num_corners} + 1))
                begin
                    cnum          <= num_dots;
                    dotx[num_dots] <= xpos;
                    doty[num_dots] <= ypos;
                    dotmask <= dotmask * 2 + 1;
                end

            num_dots <= num_dots + 1;
        end
    end

always @(posedge clk) if (en) begin
    if (hblank || vblank )
        rgb <= 0;
    else if (switches[0])    // 1 pixel outline of visible area (white)
        rgb <= (hpos==0 | hpos==639 | vcount==0 | vcount==479) ? 7 : 0;
    else if (switches[1])    // color bars
        rgb <= hpos[8:6];
    else if (switches[2])    // checker board
        rgb <= (hpos[2] == vcount[2]) ? 7 : 0;
    else if (switches[3])    //from sram
        rgb <= { nextpixel, 1'b0, dot};
    else
        rgb <= 0;           // default: black
end

endmodule

```

Threshold_select.v

```

module threshold_select(threshold,clk,buttonup,buttondown,tlow,thigh);
    output [7:0] threshold;
    input clk;
    input buttonup;
    input buttondown;

```

```

        output [3:0] tlow;
        output [3:0] thigh;

    reg [21:0] big_clock;
    reg [7:0] threshold;

    assign tlow = threshold[3:0];
    assign thigh = threshold[7:4];

    always @(posedge clk) begin
        big_clock <= big_clock + 1;
        if (big_clock == 0) begin
            if (buttonup & buttondown) threshold <= 95;
            else begin
                if (buttonup) threshold <= threshold + 1;
                else if (buttondown) threshold <= threshold - 1;
            end
        end
    end

end

endmodule

```

Hex_display.v

```

module hex_display(clk,d0,d1,d2,d3,seg,an0,an1,an2,an3);
    input clk;
    input [3:0]d0;
    input [3:0]d1;
    input [3:0]d2;
    input [3:0]d3;

    output [7:0] seg;
    output an0,an1,an2,an3;

    reg [3:0]d;

    reg an0,an1,an2,an3;
    reg [7:0] seg;

    reg [1:0] div;

    reg [15:0] count;

    always @(posedge clk) begin
        count <= count+1;
        if (count == 0)

```



```

        div <= div+1;
end

always @(posedge clk) begin
    an0 <= (div != 'b00);
    an1 <= (div != 'b01);
    an2 <= (div != 'b10);
    an3 <= (div != 'b11);

    case(div)
        'b00: d <= d0;
        'b01: d <= d1;
        'b10: d <= d2;
        'b11: d <= d3;
    endcase

    case (d)
        0 : seg <= 'b00000011;
        1 : seg <= 'b10011111;
        2 : seg <= 'b00100101;
        3 : seg <= 'b00001101;
        4 : seg <= 'b10011001;
        5 : seg <= 'b01001001;
        6 : seg <= 'b01000001;
        7 : seg <= 'b00011111;
        8 : seg <= 'b00000001;
        9 : seg <= 'b00001001;
        10 : seg <= 'b00010001;
        11 : seg <= 'b11000001;
        12 : seg <= 'b01100011;
        13 : seg <= 'b10000101;
        14 : seg <= 'b01100001;
        15 : seg <= 'b01110001;
    endcase

end

endmodule

```

Edgefollow.v

```

module edgefollow(clk, mem_clk, cnum, ypos, xpos, sram_oe, sram_data,
sram_address, n_corners, reset_btn);
    input reset_btn;

    input clk;
    input [6:0] mem_clk;

    input [3:0] cnum;

    output [8:0] xpos;
    output [8:0] ypos;

```

```

output [3:0] n_corners;

reg      [3:0] n_corners;

output sram_oe;

output [17:0] sram_address;
input  [15:0] sram_data;

reg [17:0] sram_address;
reg      sram_oe;

reg [15:0] pixel_data;

reg [4:0] block_count;
reg [3:0] pixel;

wire [8:0] x;
assign    x = {block_count, pixel};

reg [3:0] div;

wire [2:0] mem_div;
assign mem_div = mem_clk[2:0];

wire [3:0] pixel_count;
assign pixel_count = mem_clk[6:3];

reg  [8:0] y;

reg  lastpixel;

wire enter_read_state;
assign enter_read_state = (pixel_count == 1 ) || (pixel_count == 2 );

reg [8:0] last_posedge;      // last positive edge found on current
line
reg [8:0] edge_center[7:0];  // coordinates edge centers  on current
line
reg [8:0] edge_center_prev[7:0]; // coordinates of edge centers on
previous line
reg [2:0] num_edges;          // number of edges found on current line
reg [2:0] num_edges_prev;    // previous number of edges

reg [5:0] slopes[7:0];       // the slopes of the edges from
previous to current line
reg [5:0] slopes_prev[7:0];  // the slopes of the edges onto
previous line

reg [8:0] corner_x[15:0];     // x corner coordinates
reg [8:0] corner_y[15:0];     // y corner coordinates

reg [8:0] corner_x_prev[15:0]; // values from previous frame
reg [8:0] corner_y_prev[15:0]; //

```

```

reg [3:0] num_corners;          // number of corners thus found

reg [5:0] state;
reg [5:0] proc_state;

parameter LINE_CHECK_EDGES  = 0;
parameter LINE_CALC_SLOPES = 1;
parameter LINE_CHECK_SLOPES = 2;
parameter LINE_END         = 3;

parameter WAIT              = 0;
parameter READ_BLOCK       = 1;
parameter PROCESS_BLOCK    = 2;
parameter PROCESS_LINE     = 3;
parameter PROCESS_FRAME    = 4;
parameter RESET            = 7;

reg [8:0] xpos;
reg [8:0] ypos;

always @(posedge clk) begin
    xpos <= corner_x_prev[cnum];
    ypos <= corner_y_prev[cnum];
end

always @(posedge clk) begin
    if (reset_btn)
        state <= RESET;
    else
        case (state)
            WAIT:  if (enter_read_state & (mem_div == 7 ))
                    state <= READ_BLOCK;

            READ_BLOCK: begin
                case (mem_div)
                    4: begin
                        sram_address <= y * 20 + block_count;
                        sram_oe      <= 1;
                    end
                    5: pixel_data    <= sram_data;
                    6: sram_oe        <= 0;
                    7: begin
                        state         <= PROCESS_BLOCK;
                        pixel          <= 0;
                    end
                endcase
            end

            PROCESS_BLOCK: begin
                pixel <= pixel + 1;

                case ( {lastpixel,pixel_data[pixel]} )
                    'b10: last_posedge <= x ;
                    'b01: begin

```

```

        edge_center[num_edges] <= (last_posedge + x)
/ 2 ;
        num_edges <= num_edges + 1;
    end
endcase

lastpixel <= pixel_data[pixel];

if (pixel > 14)
    if (block_count > 18) begin
        state <= PROCESS_LINE;
        proc_state <= LINE_CHECK_EDGES;
        block_count <= 0;
    end
    else begin
        state <= WAIT;
        block_count <= block_count + 1;
    end
end

end
PROCESS_LINE: begin
    case(proc_state)
        LINE_CHECK_EDGES: begin
            if (num_edges == 0) begin
                proc_state <= LINE_CHECK_SLOPES;
                div <= 0;
            end
            else begin
                div <= 0;
                proc_state <= LINE_CALC_SLOPES;
            end
        end
        LINE_CALC_SLOPES: begin
            if (num_edges_prev == 0 ) begin // new
edges <= record corners

                corner_x[num_corners] <= edge_center[div];
                corner_y[num_corners] <= y;
                num_corners <= num_corners + 1;

                if (div == (num_edges - 1))
                    proc_state <= LINE_END;
                else
                    div <= div + 1;
            end
            else begin // edges
found previously <= calculate slopes

                slopes[div] <= (edge_center[div] -
edge_center_prev[div]) ;

                if (div == num_edges - 1) begin
                    proc_state <= LINE_CHECK_SLOPES;
                    div <= 0;
                end
                else
                    div <= div + 1;
            end
        end
    end
end

```

```

        end
    end
    LINE_CHECK_SLOPES: begin
        if (num_edges != 0) begin
            if (slopes[div] > slopes_prev
[div])) // avoid negative numbers
                if ( (slopes[div] - slopes_prev[div]) > 4 )
                    begin
                        corner_x[num_corners] <= edge_center
[div];
                        corner_y[num_corners] <= y;
                        num_corners <= num_corners + 1;
                    end
                else
                    if ( (slopes_prev[div] - slopes[div]) > 4 )
                        begin
                            corner_x[num_corners] <= edge_center
[div];
                            corner_y[num_corners] <= y;
                            num_corners <= num_corners + 1;
                        end
                    end
                end
            end

            edge_center_prev[div] <= edge_center[div];
            slopes_prev[div] <= slopes[div];

            if (div == num_edges)
                proc_state <= LINE_END;
            else
                div <= div + 1;
            end
        end
    LINE_END: begin
        num_edges <= 0;
        num_edges_prev <= num_edges;

        lastpixel <= 1;
        div <= 0;
        if (y > 239) begin
            state <= PROCESS_FRAME;
            div <= 0;
        end
        else begin
            y <= y + 1;
            state <= WAIT;
        end
    end
    end
    default: proc_state <= LINE_END;
endcase

end
PROCESS_FRAME: begin
    div <= div + 1;
    num_edges <= 0;

    n_corners <= num_corners;

```

```

        corner_x_prev[div] <= corner_x[div];
        corner_y_prev[div] <= corner_y[div];

        if (div == num_corners) begin
            state <= WAIT;
            num_corners <= 0;
            lastpixel <= 1;
            y <= 3;
            block_count <= 1;
        end
    end

RESET: begin
    num_corners <= 0;
    lastpixel <= 1;
    n_corners <= 0;
    block_count <= 1;
    y <= 3;
    state <= WAIT;
end

default: state <= WAIT;

endcase
end

endmodule

```

Syncgen.v

```

module syncgen(clk,h,v);
    input clk;
    output h;
    output v;

    reg c1[11:0];
    reg c2[11:0];
    reg c3[19:0];
    reg c4[19:0];

    always @(posedge clk) begin
        if (c1 > 3175) begin
            h<=1;
            c1<=0;
            c2<=0;
        end
        else if (c2 > 2940) begin
            h <= 0;
            c2<=0;
        end
        else if (c3 > 980000) begin
            v <= 1;
            c3<=0;
        end
    end
endmodule

```

```
        c4<=0;
    end
    else if (c4 > 970000) begin
        v <=0;
        c4<=0;
    end
    else begin
        c1 <= c1 + 1;
        c2 <= c2 + 1;
        c3 <= c3 + 1;
        c4 <= c4 + 1;
    end
end

endmodule
```

Input Wiring Diagram

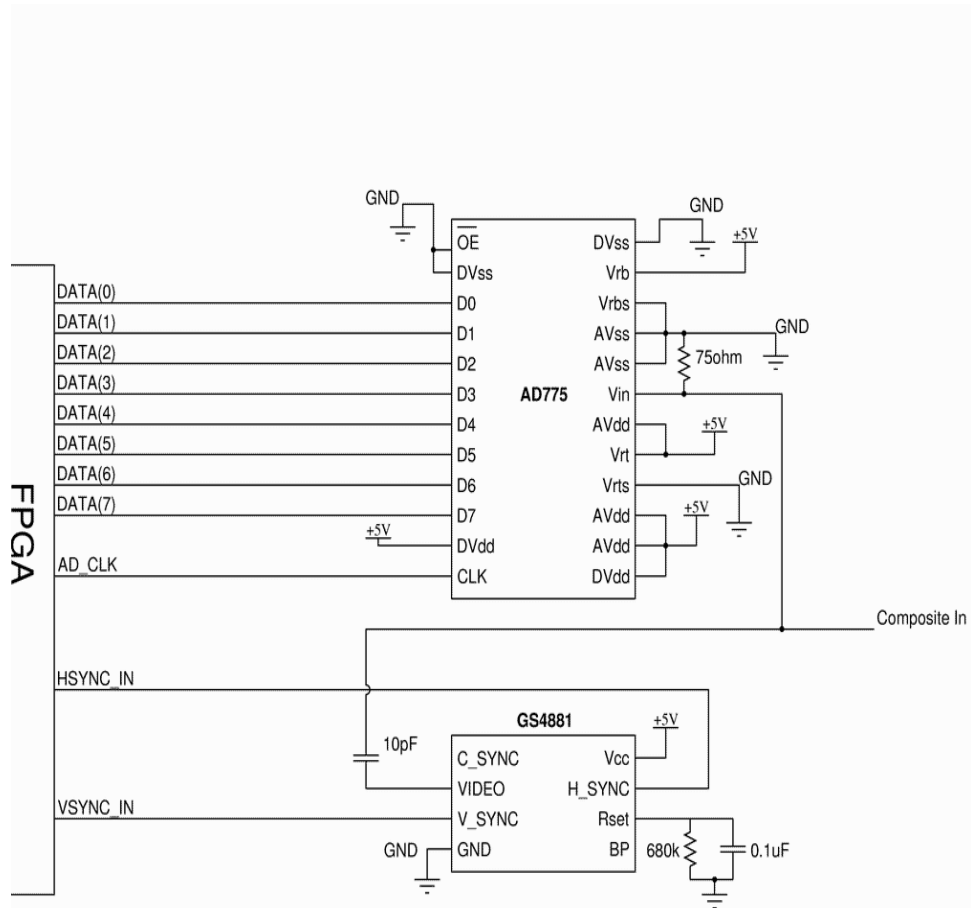


Figure 4 - Input Wiring Diagram