Hardware-Accelerated Pipelined Video Processor

Final Lab

Aditya Pola(pola3)

Jonathan Wang(jfwang3)

ECE 385: Digital Systems Laboratory

Lab Section AB1

TA: Peidong Yang

Fall 2025

Motivation

FPGAs excel at high-speed, low-latency computation, making them well-suited for real-time video processing. In this project, we leverage those strengths to build a live video pipeline that captures camera data, processes it on the fly, and outputs it to an HDMI display with overlay and control features.

The video processing pipeline begins with the camera interface, which captures pixel data in raster order. This data is temporarily stored in a BRAM-based frame buffer, enabling synchronization between the camera clock domain and the display clock domain. The frame buffer ensures safe and efficient crossing of clock domains while holding the full image data for downstream processing.

However, before the pixel data is written to the frame buffer, it is processed by a preprocessor. The preprocessor employs line buffers to store only the necessary neighborhood data needed to compute a new pixel value. This approach allows for real-time per-pixel processing while maintaining low latency. The preprocessor includes several image-processing modules, such as pixel filtering, grayscale conversion, multi-stage convolution, morphological/spatial filters, and shallow artificial intelligence identification models. Notably, the preprocessor also incorporates blob thresholding for simple object detection, enabling the system to identify and track objects within the video stream. The preprocesser stage ensures that processed pixels are immediately available for storage in the frame buffer without unnecessary delays.

After preprocessing, the postprocessor takes over and handles the pixel stream as it is forwarded to the HDMI output. This stage adds on-screen overlays, such as crosshairs, text, or bounding boxes, and interprets the processed pixel data to generate control signals. These control signals can be used to actuate hardware, such as motors, based on the detected objects' locations in the video frame. Additionally, it adds interactive states including augmented reality and pong.

The system is coordinated by a control unit, which facilitates demonstration mode. This mode cycles through various image-processing filters and operations, allowing users to compare the original and processed video streams. The control unit's goal is to show how different stages of image processing, such as convolution and thresholding, can be combined to perform sophisticated tasks like object detection, all while maintaining the FPGA's real-time, low-latency advantages.

Preparation

I. Architecture

To start, we made a plan for what architecture should look like for the project. In advance, we understood various subsystems were necessary, including a frame buffer to handle

clock domain crossing, a camera controller to initialize camera slave registers and stream pixel data, and some systems to actually process the data for output. The architecture outlined in Figure 1 shows a basic outline of our proposed video processor. Each of the blocks correspond to a real module in our top-level used in our video-processing pipeline. The Vivado generated block diagram for our actual implementation is showcased in Figure 2 and Figure 3.

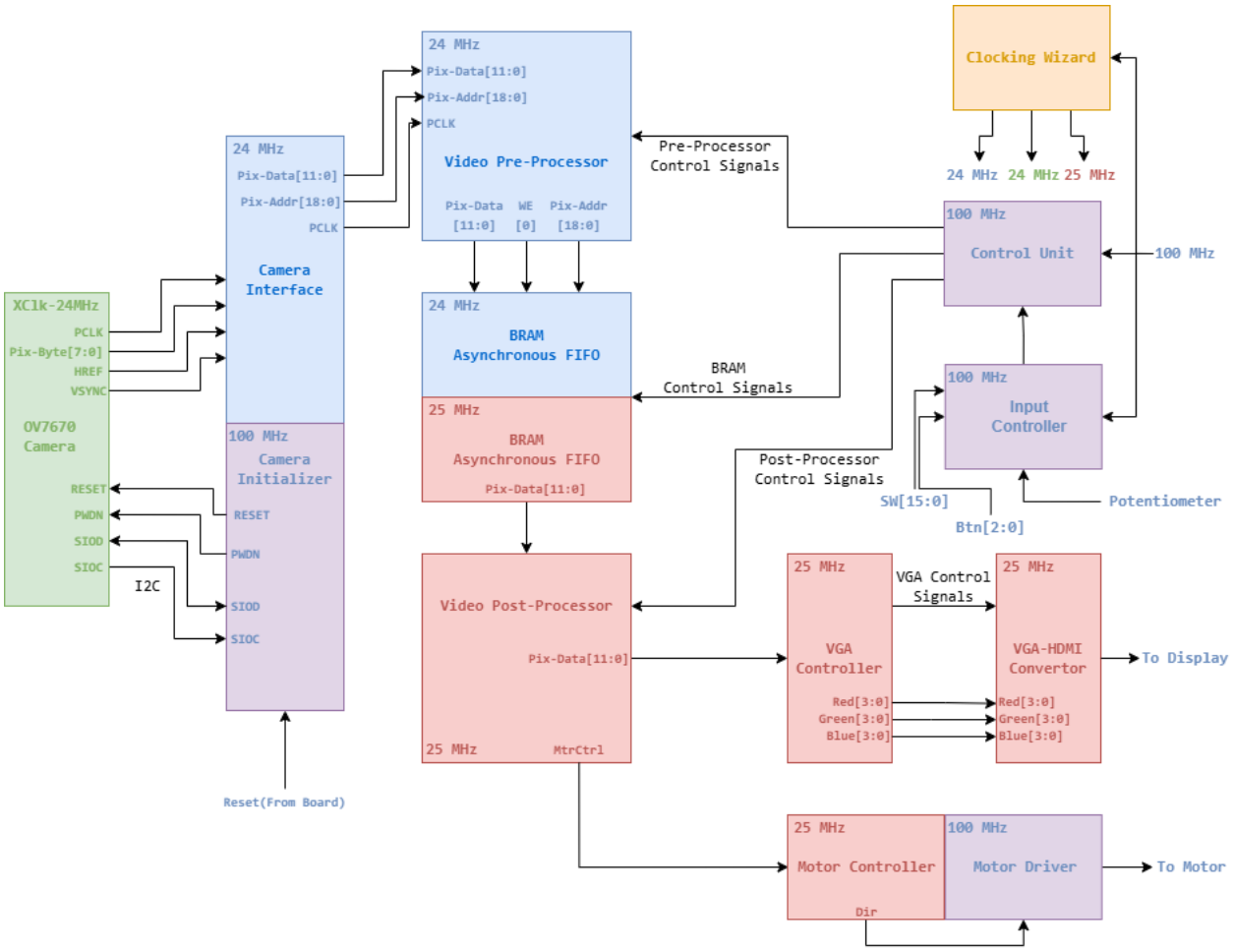

Figure 1: System Architecture for Video Processor

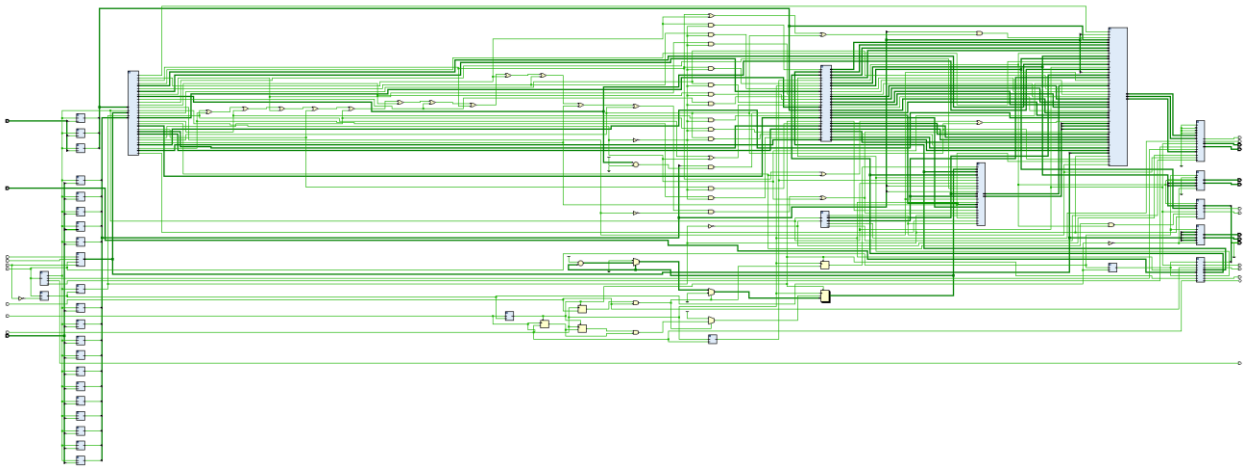

Figure 2: Complete RTL Block Diagram of Video Processor

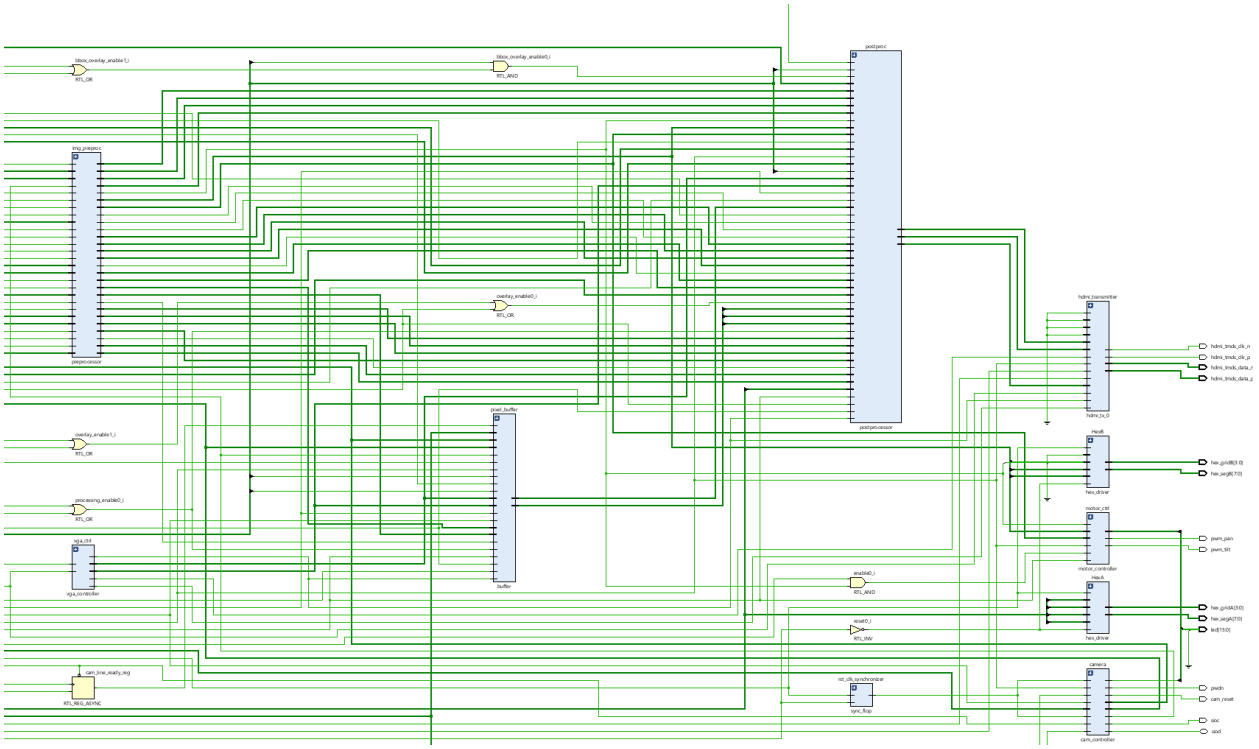

Figure 3: Focused RTL Block Diagram of Video Processor

II. Camera Controller

The camera we chose to use for our FPGA-based video processor is the OV7670. The OV7670 acts as the video source and provides an 8-bit parallel pixel bus along with synchronization signals. The pin layout of the OV7670 is shown in Table 1. Note that the OV7670 interacts with the FPGA as a slave device. The Urbana Board is able to interface with the OV7670 camera via the SCCB protocol to read and write to internal device registers. Once the internal registers are configured, the camera is able to output pixel data

in a selected format and in raster order based on these setup details. In this sense, the camera controller is naturally divided into two distinct phases: initialization and readback.

Pin Name	Purpose
3V3	Provides the 3.3-volt power supply required for the camera module
GND	Electrical ground reference for the camera's power and signals
SIOD	Data line used for SCCB register read/write communication with the FPGA
SIOC	**Cl**ock line driven by the FPGA for SCCB communication timing
VSYNC	Frame synchronization signal that indicates the start and end of each video frame
HREF	Line synchronization signal that stays high during valid pixel data for a given row
PCLK	Pixel clock generated by the camera that marks when each new pixel byte is valid
XCLK	External clock input from the FPGA that drives the camera's internal timing
D[7:0]	8-bit parallel data bus carrying pixel byte outputs from the camera
RST	Active-low reset pin used to force the camera into its default configuration
PWDN	Power-down control pin that disables the camera when driven high

Table 1: Pin Layout of OV7670 Camera

A. Camera Initializer

Upon startup, the OV7670 camera begins in a default configuration, which typically does not match the desired output format, frame size, clocking behavior, or exposure/gain settings. Thus, to retrieve any meaningful pixel data from the camera, the device must be configured through a sequence of register writes.

The SCCB interface functions very similarly to the I²C interface, using a 2-wire open-drain bus consisting of an SIOC signal and an SIOD signal. The SIOC signal is the clock line driven by the master device (the FPGA) to synchronize communication. The SIOD signal is a bidirectional data line used for transmitting read or write data between master and slave. Because this interface is half-duplex, only one direction of communication can occur at a time. In our setup, the FPGA is the master device and the OV7670 acts as the slave, with a fixed write address of 0x42 and a fixed read address of 0x43. Only writing is required for proper camera configuration.

To indicate the start of data transmission, the master drives the SIOD line from high to low while the SIOC line is high. Conversely, the master indicates the end of transmission by driving SIOD from low to high while SIOC is high. These transaction timings are shown in Figure 4 and Figure 5 below.

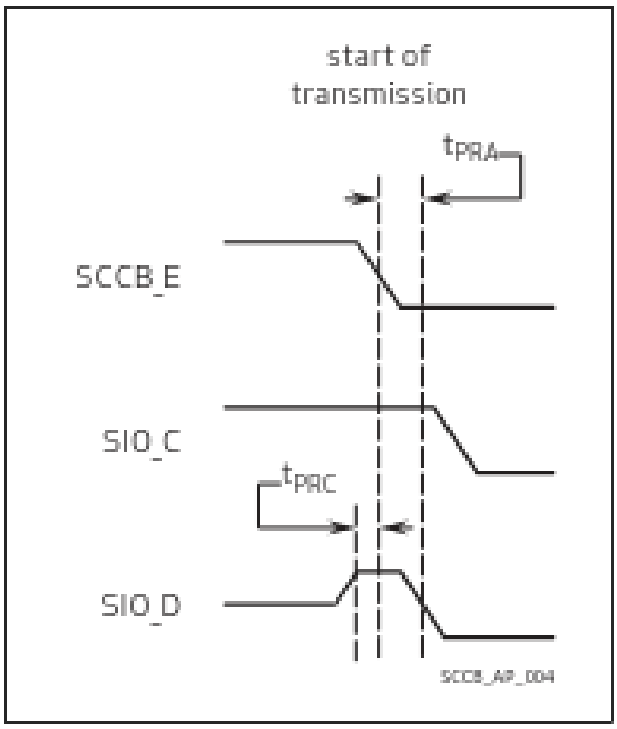

Figure 4: Transaction Timing for Start of Transmission

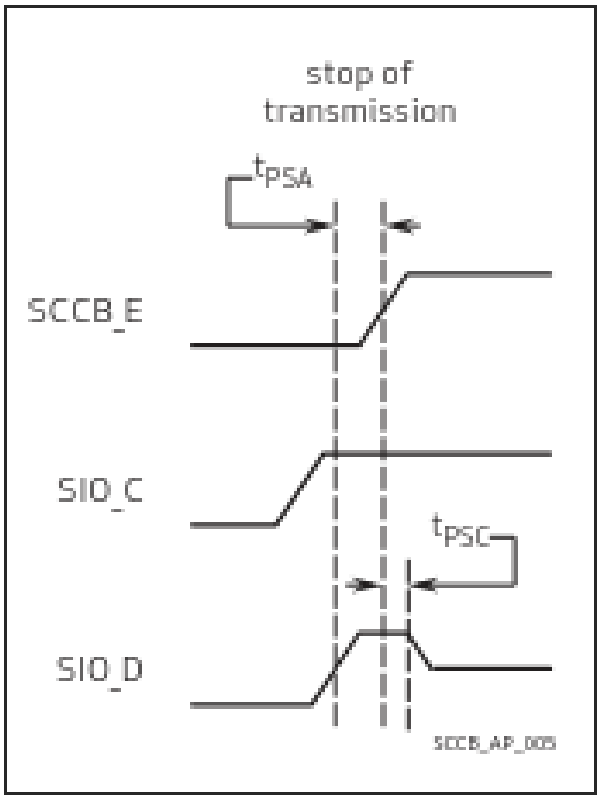

Figure 5: Transaction Timing for Stop of Transmission

The SCCB write interaction is a three-phase process shown in Figure 6. Each phase consists of one byte of data transmission followed by one don't-care bit, which functions similarly to the ACK/NACK bit in I²C. The first phase specifies the slave device address along with the transaction type (read/write). For writing, the second phase specifies which internal slave register we want to write to. Finally, the third phase transmits the data to be written into that selected register.

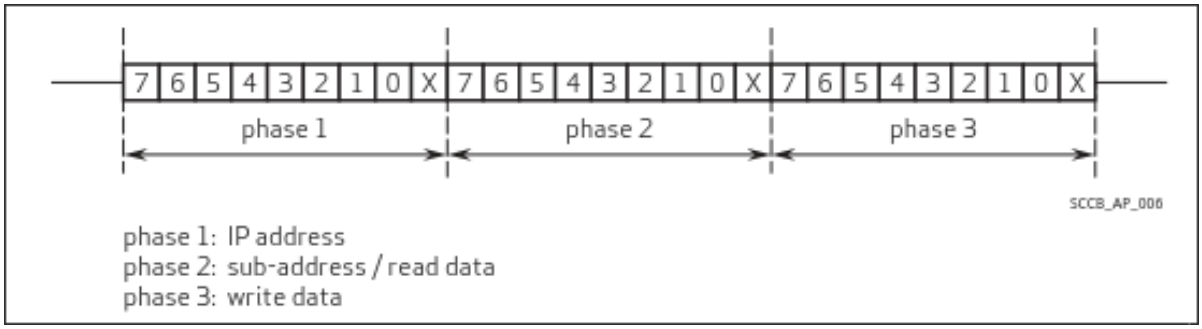

Figure 6: Byte Arrangement for SCCB 3-Phase Write

To implement this initialization sequence in hardware, we created a read-only ROM containing every register-value pair required for configuring the OV7670. This ROM is synthesized as a small block of constant memory indexed by an internal address counter. At system startup, the initializer FSM begins at ROM address zero and sequentially steps through the entire table, retrieving the device address, register address, and register data needed for each SCCB write transaction. Because the ROM is read-only and hardcoded during synthesis, it guarantees deterministic camera configuration and removes the need for any external microcontroller or software-driven setup.

During the first phase of writing, we transmit the slave device address as well as the R/W transaction indicator. The first seven bits represent the slave device identifier, while the eighth bit selects read or write mode. This explains why the write address is 0x42 and the read address is 0x43 (the LSB toggles the mode). The full SCCB transaction for phase 1 is shown in Figure 7.

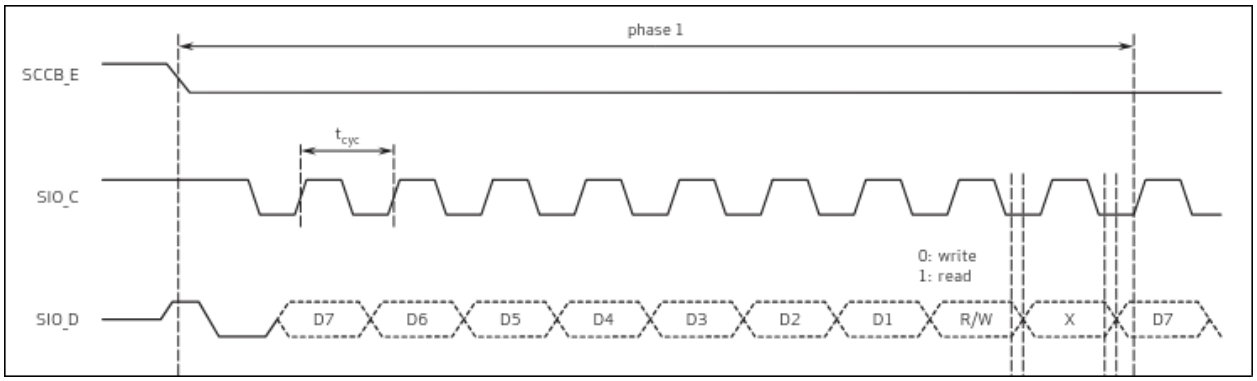

Figure 7: Timing Diagram for Phase 1 of SCCB Write Transaction

During the second phase of writing, we transmit the internal slave register address. The details of each register are not required for this lab document, but the device datasheet outlines color control settings, scaling options, timing configuration, and pixel formatting registers. In this phase, all eight bits correspond to the register address, followed by the Don't-Care bit, as shown in Figure 8.

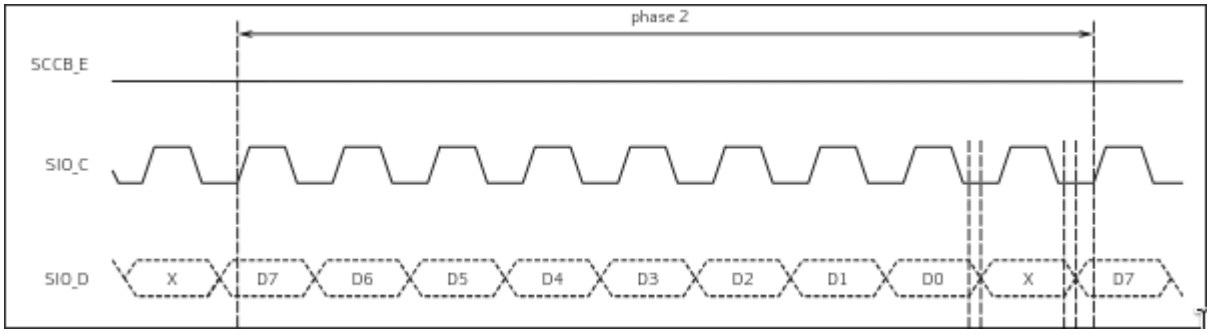

Figure 8: Timing Diagram for Phase 2 of SCCB Write Transaction

During the third and final phases of writing, we transmit the actual data to be written into the register selected in the previous phase. The data byte follows the same structure as the second phase, with the full byte containing the write value. Thus, the transaction for this phase appears identical to that of Figure 9.

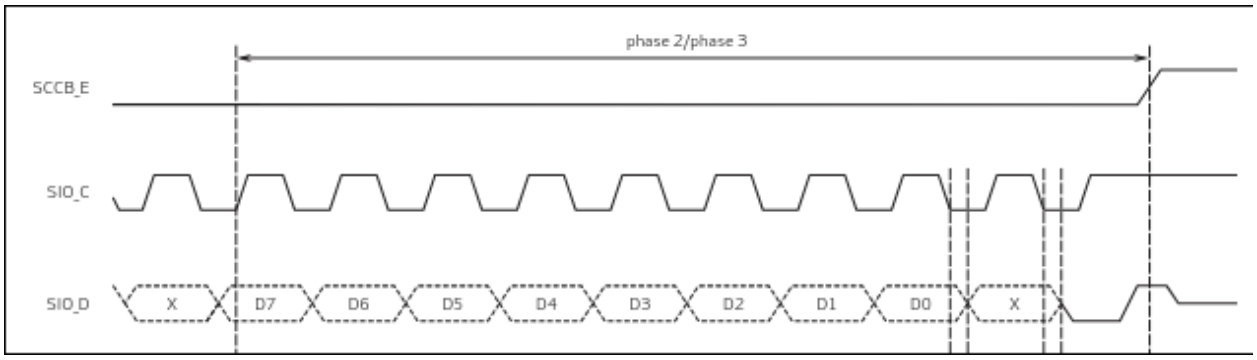

Figure 9: Timing Diagram for Phase 3 of SCCB Write Transaction

Note that each of these byte transactions ends with a don't-care bit analogous to the ACK/NACK bit of I²C. In this sense, standard I²C controllers can still be used to communicate with the OV7670.

To summarize, a typical SCCB write operation consists of sending one byte of data followed by a don't-care bit. To perform a write, the master begins by asserting the start condition (SIOD transitioning low while SIOC is high). Then, three SCCB write phases are performed: device address, register address, and register data. Finally, the master ends the transaction by asserting the stop condition (SIOD transitioning high while SIOC is high). Because the idle state of the bus is defined as SIOC high, this makes start and stop signaling straightforward. Additionally, data-bit changes occur only when SIOC is low, which prevents metastability by ensuring that the data has settled before being sampled. An internal counter was used to ensure transitions occur cleanly between SIOC pulses.

B. Camera Interface

Once the initialization phase has been completed and all SCCB register writes have been committed, the OV7670 begins producing continuous pixel data in the format selected during configuration. In our design, the camera is configured to output RGB444, meaning each pixel contains 4 bits for red, 4 bits for green, and 4 bits for blue, forming a 12-bit color representation. At this stage, the Camera Interface module becomes responsible for capturing this data, assembling the RGB444 pixels from the incoming bytes, and forwarding the processed pixels into the remainder of the video processor pipeline.

The OV7670 streams pixel data using its dedicated 8-bit data bus, D[7:0], accompanied by three essential synchronization signals: PCLK, HREF, and VSYNC. The pixel clock, PCLK, is generated by the camera and dictates when the FPGA must sample incoming data. Every rising edge of PCLK corresponds to a new byte of data being driven onto D[7:0]. The HREF signal remains high during active pixel transmission within a single row, and it drops low between rows to mark row boundaries. Meanwhile, the VSYNC signal asserts around frame boundaries, indicating when a new frame begins and when the

output of the previous frame has ended. A timing diagram illustrating the relationship between PCLK, HREF, VSYNC, and the data bus should be included at this point to clarify how valid pixel data is organized across each line and frame (Figure 9).

Because the camera is configured for RGB444 output, each pixel is transmitted as two successive bytes while HREF is high. The first byte contains the upper four bits of red concatenated with the upper four bits of green, and the second byte contains the upper four bits of blue followed by four unused bits. Thus, in RGB444 mode, the first byte includes {R[3:0], G[3:0]}, and the second byte includes {B[3:0], XXXX} where the lower nibble is not used. A figure showing how these bytes map to a single RGB444 pixel can be inserted here (Figure 10). Since pixel values are split across two clock cycles, the Camera Interface must maintain an internal state to track whether the incoming byte is the first or second half of the pixel. A simple byte-toggle mechanism within the interface logic ensures that the module correctly pairs the two bytes and reconstructs a complete 12-bit RGB444 pixel.

To capture valid pixel data, the interface samples D[7:0] on every rising edge of PCLK but only when HREF is asserted. The moment HREF goes high, the interface begins collecting bytes, alternating between “first byte” and “second byte” states. Once both bytes have been captured, the interface combines them into a single RGB444 pixel. This reconstructed pixel can then be zero-extended or padded to a convenient width (typically 16 bits) before being forwarded to the downstream modules that perform grayscale conversion, filtering, or display formatting. A block diagram demonstrating this byte-pairing and pixel-assembly logic belongs here (Figure 11).

Proper handling of row and frame boundaries is also crucial. Whenever VSYNC asserts, the interface resets its internal counters, signaling that a new frame has begun and that no valid pixel data should be captured until VSYNC falls. Similarly, when HREF deasserts at the end of a line, the byte-toggle mechanism resets to ensure that the next rising edge of HREF always begins with the first byte of a new pixel. These behaviors guarantee that the pixel stream is always aligned with the raster order expected by the subsequent stages of the video processor.

In summary, the Camera Interface acts as the real-time data acquisition stage of the camera controller. It synchronizes with PCLK, identifies valid pixel regions using HREF and VSYNC, assembles RGB444 pixel values from pairs of incoming bytes, and delivers a clean, correctly formatted pixel stream to the remainder of the system. With the initializer establishing the camera’s operating mode and the interface reliably capturing and formatting its output, the system provides a robust foundation for real-time video processing on the FPGA.

III. Pixel Buffer

The pixel buffer serves as a full-frame storage element that holds all processed pixel data prior to display. In order to store an entire 640×480 frame, the system requires memory for 307,200 pixels. Because OV7670 is configured to output RGB444 data, each pixel originally consisted of 12 bits. However, storing this full-resolution format exceeds the available BRAM capacity on the Urbana Board. To address this limitation, the design compresses each pixel to RGB333, reducing the per-pixel storage requirement to 9 bits. This compact representation preserves the essential color information while allowing the full frame to be stored within the FPGA's memory resources.

To support real-time data flow, the pixel buffer is implemented as a true dual-port BRAM, with Port A dedicated to camera writes and Port B dedicated to display reads. These ports operate in separate clock domains: the camera interface uses the pixel clock (PCLK) generated by the OV7670, while the VGA controller operates on the 25 MHz VGA pixel clock. Because each port has independent address, data, and enable signals, the camera can continuously stream pixel data into memory without interfering with simultaneous read operations performed by the display pipeline.

During frame capture, the camera interface computes a linear memory address from the current row and column coordinates, typically using the formula address = y * 640 + x. This address is driven into Port A of the BRAM along with the compressed RGB333 pixel. Since the frame is written strictly in raster order, the BRAM becomes a memory-mapped representation of the entire image that can be accessed randomly by downstream modules. This coordinate-based addressing scheme ensures that any stage of the processing pipeline, or the display engine, can request pixels from arbitrary screen positions without requiring line buffers or sequential reads.

When retrieving pixel data for display, the VGA controller supplies its own (x, y) coordinates to Port B of the BRAM, generating a corresponding linear address that selects the pixel to be shown on screen. Because FPGA block RAM is synchronous, each read operation incurs a one-cycle latency between presenting an address and receiving valid data. To compensate for this, the system uses a prefetch mechanism: the pixel address for the *next* VGA coordinate is computed for one clock cycle early and issued to the BRAM during the previous cycle. As a result, the data output aligns with the active display pixel at the correct time. Without this prefetching step, the VGA controller would display delayed or misaligned pixel values.

In addition to storing the raw RGB333 camera data, the pixel buffer is designed to remain flexible and accommodate the various pixel formats used in the preprocessing pipeline. Because the buffer is indexed by explicit (x, y) coordinates, it can store either unmodified camera pixels or the output of image-processing modules directly in the same memory

space. When frames are downscaled to a 3-bit grayscale representation for filtering or edge detection, the remaining six bits of the 9-bit BRAM entry become available for auxiliary data. Similarly, for 9-bit binarized images, the full entry is used but still does not exceed the storage available for color. This under-utilization relative to the original RGB333 format allows the system to incorporate temporal information, such as storing previous-frame grayscale values or frame-to-frame differences, in the same BRAM address space without requiring additional memory. By embedding both current and historical pixel information within the existing 9-bit structure, the pixel buffer supports more advanced temporal processing, such as motion detection or temporal smoothing, while remaining consistent with the dual-port architecture and the memory constraints of the Urbana Board. The encoding of the continuous 3-bit information is done by treating the 9-bit address words as a circular queue, overwriting the data from 3 frames prior as the current frame is being written.

IV. Preprocessor

The preprocessor contains various substates, which will each be explained here. The preprocessor is pipelined in that data is processed as soon as it becomes available rather than at the next clock cycle. In order to do this, we use line buffers to be space efficient due to the limited memory leftover. All processing typically occurs on grayscale data, so the first step in any sort of preprocessing is gray scaling the image, which also heavily compresses the image from 9 bits per pixel to 3 bits per pixel. With the pixel data each being 3 bits, we can create line buffers with LUTs to store 3 lines of data. The line buffers act like a stack, in which new lines are shuffled in from the top, and the remaining lines are shifted down. Using line buffers allows us to perform various filtering operations which require pixel data in surrounding neighborhoods to determine pixel mappings.

A. Grayscale

Grayscaling is performed as a per-pixel operation that converts the incoming RGB444 camera data into a 3-bit luminance value. The conversion uses a weighted average of the red, green, and blue channels, as shown in Equation 1, where each weight reflects the human eye's differing sensitivity to color, highest for green, and lowest for blue. Although this formula is typically expressed using floating-point arithmetic, such operations are computationally expensive in hardware. To ensure high performance, the equation is restructured into the integer-based approximation of Equation 2, which uses fixed multiplications and bit-shifts to perform efficient division. Grayscale conversion is foundational for nearly all subsequent preprocessing tasks: it dramatically reduces storage requirements and enables the entire pipeline to operate on a compact and uniform intensity scale.

$$Y = 0.299R + 0.587G + 0.114B$$

Equation 1: Exact Luminance Formula for Grayscale Conversion

$$Y \approx (77R + 150G + 29B)/256$$

Equation 2: Approximated Luminance Formula for Grayscale Conversion

B. Grayscale Threshold

Thresholding is a binarization technique used both to compress pixel data and to isolate regions of interest by intensity. The operation compares each pixel's grayscale value against a programmable threshold. Pixels exceeding the threshold are mapped to white, while those below are mapped to black, effectively reducing the image to a two-level representation. This process compresses the data by a factor of nine compared to the raw RGB input yet preserves essential structural information for downstream analysis. In practice, instead of thresholding directly on the grayscale value, the system often thresholds on the green channel, which provides the highest signal-to-noise ratio due to the human eye's natural sensitivity to green light. This produces cleaner segmentation and improves the reliability of filters and detection algorithms that follow.

C. Dithering

Dithering is performed as a per-pixel color compression step that reduces the incoming RGB444 camera stream of 12 bit/pixel to the RGB333 camera stream of 9-bits/pixel for the frame buffer and display pipeline. A naïve conversion used for raw output is to truncate the least significant bit of each 4-bit color channel. However, this method introduces artifacts called posterization where colors that are perceptively close to one another get rounded together to form the same color after truncation creating a splotchy image. Dithering is the processor of creating ordered noise within the image to reverse the perceived posterization. Ordered dithering computers the linear address of each pixel and adds a dither term of -1,0,+1 in the raster order of the pixels. This artificial noise prevents adjacent colors from being exactly the same creating texture in the image without destroying the general color scheme of any given region. This ordered dithering scheme creates noticeable diagonal streaks in the image due to the patterned nature of the induced noise. Bayer matrix dithering is a higher-quality form of ordered dithering that uses a small, fixed threshold matrix to distribute quantization error in a spatially uniform way across the image. Compared to the ordered method before, the Bayer matrix produces less directional raster texture and a more even, less repetitive grain so flat regions and smooth gradients exhibit less visible pattering while still reducing posterization.The Bayer scheme derives the pixel's location inside a repeating 4×4 tile using pix_x[1:0] and pix_y[1:0], looks up the corresponding Bayer matrix value (0–15), and then maps that value into a small dither term of -1, 0, or +1 using coarse thresholds (low values bias downward, mid values leave unchanged, high values bias upward). To avoid correlated color noise (where R/G/B shift together and create tinted artifacts), we phase-shift the Bayer indices per channel by offsetting the matrix value for G and B (G uses $(value + 5)\%16$, B uses $(value + 10)\%16$),

then add the resulting dither terms to the 4-bit channel values, clamp to $[0, 15]$, and finally quantize to RGB333 by taking bits $[3:1]$ of each channel. This method provides a cleaner output than the ordered dithering scheme before while still solving the posterization issue.

D. Convolution Filters

The convolution kernel module operates on a 3×3 neighborhood of 4-bit grayscale pixels and computes a filtered output pixel based on a selected kernel. For each clock cycle, the surrounding pixels p00–p22 are treated as a small window centered at p11, and the module performs a weighted sum according to the chosen kernel type. The result is then scaled and clamped back into a 4-bit range before being forwarded as pixel_out. When enable is deasserted, the module bypasses all convolution logic and simply passes the center pixel through unchanged, effectively behaving as an identity mapping. Algorithmically, this block encapsulates several classical image processing operations, smoothing, edge detection, sharpening, and embossing, using integer arithmetic and bit shifts to keep the implementation efficient and hardware-friendly.

The Gaussian blur mode implements a discrete approximation of a 2D Gaussian filter using the kernel depicted in Equation 3.

$$K_G = \frac{1}{64}\begin{bmatrix} 2 & 8 & 2 \\ 8 & 32 & 8 \\ 2 & 8 & 2 \end{bmatrix}$$

$$S_G = 2p_{00} + 8p_{01} + 2p_{02} + 8p_{10} + 32p_{11} + 8p_{12} + 2p_{20} + 8p_{21} + 2p_{22}$$

$$\text{pixel\_out} = \text{clamp}_{[0,15]}\left(\left\lfloor \frac{S_G + 32}{64} \right\rfloor\right)$$

Equation 3: Gaussian Filter Formulas

Each neighbor pixel is multiplied by its corresponding weight, with the larger central weights accentuating contributions from the center of the window while still averaging nearby pixels. In hardware, these multiplications are realized with left shifts (<< 2 for ×4 and << 4 for ×16), and the weighted sum is accumulated into gaussian_sum. Instead of performing an exact division by 36, which would require more expensive hardware, the design approximates it by a right shift of 6 bits (division by 64). This approximation is

sufficient for 4-bit grayscale data and greatly simplifies the implementation, producing a smooth, denoised output blurred_pixel that is then assigned to pixel_out.

The Sobel mode performs edge detection by computing the horizontal (gx) and vertical (gy) gradients using the standard sobel kernels depicted in Equation 4.

$$G_x = -p_{00} + p_{02} - 2p_{10} + 2p_{12} - p_{20} + p_{22}$$

$$G_y = -p_{00} - 2p_{01} - p_{02} + p_{20} + 2p_{21} + p_{22}$$

$$M = |G_x| + |G_y|$$

$$\text{pixel\_out} = \text{clamp}_{[0,15]}\left(\left\lfloor \frac{M}{4} \right\rfloor\right)$$

Equation 4: Sobel Filter Formulas

Each gradient is formed by taking signed combinations of the neighborhood pixels, with stronger weights on the mid-row and mid-column to emphasize central differences. The gradient magnitude is then approximated as |Gx |+|Gy|, which avoids the cost of computing a square root. This sum is stored in gradient, scaled down via bit selection (gradient[5:2]), and clamped into the 4-bit range. Values exceeding the maximum representable intensity are saturated to 0xF. This approximation is deliberately chosen to balance edge sensitivity with hardware simplicity, providing a strong edge response while remaining fully combinational and low-cost.

The sharpen mode enhances local contrast by applying the kernel show in Equation 5.

$$K_S = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

$$S_S = 5p_{11} - p_{01} - p_{10} - p_{12} - p_{21}$$

$$\text{pixel\_out} = \text{clamp}_{[0,15]}(S_S)$$

Equation 5: Sharpen Filter Formulas

The center pixel is multiplied by 5, and the values of its four direct neighbors (up, down, left, right) are subtracted. Intuitively, this operation boosts areas where the center differs from its surroundings, making edges and fine details more pronounced. In hardware, the multiplication by 5 is implemented as a simple 5 * p11, and all operations are performed with signed arithmetic in sharpen_result. After computation, the result is clamped so that negative values are forced to 0 (black) and values above 15 are saturated to 0xF (white). This ensures the sharpened output fits in the original 4-bit grayscale range while preserving the most prominent enhancements.

The emboss mode produces a 3D relief effect using the asymmetric kernel shown in Equation 6.

$$K_E = \begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

$$S_E = -2p_{00} - p_{01} - p_{10} + p_{11} + p_{12} + p_{21} + 2p_{22}$$

$$\text{pixel\_out} = \text{clamp}_{[0,15]}(S_E + 8)$$

Equation 6: Emboss Filter Formulas

This kernel emphasizes intensity differences along a diagonal direction, causing edges with a particular orientation to appear raised or recessed, mimicking directional lighting. The module computes a signed sum emboss_result from the weighted neighborhood, then adds a constant bias of 8 (mid-gray) to the output so that neutral regions sit around gray instead of black. Finally, the result is clamped to the 4-bit range: negative values are saturated to 0, and values above 15 are saturated to 0xF. The use of integer weights (−2, −1, 0, 1, 2) and a fixed bias allows embossing to be implemented with only additions, subtractions, and one small multiply, making it well-suited for real-time hardware processing.

E. Median Filters

The median filter module performs 3×3 median filtering over a neighborhood of 4-bit grayscale pixels to suppress impulsive noise while preserving edges. For each output pixel, the nine inputs p00–p22 are treated as a window centered at p11. Instead of

averaging the values, the filter selects the median (the 5th largest value), which is less sensitive to outliers than linear smoothing filters. This makes the median filter particularly effective against salt-and-pepper noise and other sparse, high-amplitude artifacts, while still maintaining sharp boundaries that are crucial for later edge detection and segmentation stages.

Algorithmically, the module uses a sorting network implemented via a series of compare–swap operations. The nine pixel values are first loaded into the sorted[0:8] array, and then passed through a fixed sequence of swap calls. Each swap(a, b) compares the two 4-bit values and exchanges them if a > b. The stages are carefully chosen so that, after all swaps complete, the element at sorted[4] is guaranteed to be the median, without requiring a full general-purpose sort of all nine elements. This fixed compare–swap sequence is fully combinational and has deterministic depth, which is ideal for hardware implementation.

From a hardware perspective, the median filter avoids multipliers, dividers, or large adders entirely. It relies only on simple **c**omparators and small multiplexers to implement the swap function. Because the sorting network is static and does not depend on data-dependent branching, it can be efficiently synthesized and pipelined at a known critical path. The decision rule for each output pixel is simply: Output the 5th element of the sorted 3×3 neighborhood, i.e., pixel_out = sorted[4]. This rule ensures that for each neighborhood, the output is representative of the "middle" intensity, suppressing extreme values (noise) while maintaining structures and edges.

F. Morphological Filters

The morphological filter operates on a binary mask encoded using 4-bit pixels, where 0xF represents a foreground ("skin") pixel and 0x0 represents a background (non-skin) pixel. For each 3x3 neighborhood (p00–p22), the module performs either erosion or dilation depending on which enable signal is active. Both operations are classical morphological filters used to clean up binary masks: erosion removes small foreground specks and shrinks blobs, while dilation expands and reconnects fragmented foreground regions. The output pixel_out is also 4-bit but effectively binary: either 0xF (foreground) or 0x0 (background), as defined in Equation 10.

$$c_{ij} = \begin{cases} 1, & p_{ij} \neq 0 \\ 0, & p_{ij} = 0 \end{cases} \qquad (i, j \in \{0, 1, 2\})$$

Equation 7: Single-Bit Flag Conversion

Internally, all nine 4-bit pixels (p00–p22) are first reduced to single-bit flags (c00–c22) using a reduction OR, i.e., any nonzero 4-bit value is treated as foreground. This binary conversion step is defined in Equation 7. The morphological decisions are then performed purely on these Boolean flags using AND and OR trees, which are extremely efficient structures in hardware.

$$e = c_{11} \cdot (c_{00}c_{01}c_{02}c_{10}c_{12}c_{20}c_{21}c_{22})$$

Equation 8: Erosion Formula

Erosion operates on a binary mask by shrinking foreground regions and eliminating small, isolated noise pixels. After the conversion in Equation 7, the erosion rule checks whether the center pixel and all eight neighbors in the 3x3 window are foreground. This decision rule is expressed in Equation 8. Only if every pixel in the window is foreground does the output remain white; otherwise, it becomes black. This effectively removes thin connections, isolated white specks, and noise present on a dark background.

$$d = c_{00} + c_{01} + c_{02} + c_{10} + c_{11} + c_{12} + c_{20} + c_{21} + c_{22}$$

Equation 9: Dilation Formula

Dilation performs the opposite transformation by expanding foreground regions and reconnecting nearby clusters of white pixels. After the conversion in Equation 7, the dilation rule checks whether any pixel within the 3x3 neighborhood is foreground. This decision rule is expressed in Equation 9. If at least one foreground pixel is present, the output becomes white; if not, the output remains black. This causes white regions to grow outward by one pixel in all directions, helping maintain continuity of legitimate blobs that may have been fragmented during erosion or thresholding.

$$\text{pixel\_out} = \begin{cases} 15, & \text{if erosion enabled and } e = 1 \\ 15, & \text{if dilation enabled and } d = 1 \\ 0, & \text{otherwise} \end{cases}$$

Equation 10: Output Pixel Decision Rule

G. Skin Filter

The skin threshold module performs real-time skin segmentation directly on the RGB444 camera stream, implementing a hardware-optimized version of the Python RGB333 logic. The incoming 4-bit RGB channels are first quantized to RGB333 by discarding the least significant bit of each channel, reducing the dynamic range to 0–7 (Equation 11). This matches the domain used by the original software thresholds while lowering hardware cost.

$$R_3 = R_4[3{:}1]$$
$$G_3 = G_4[3{:}1]$$
$$B_3 = B_4[3{:}1]$$

Equation 11: RGB444->RGB333 Conversion

From the RGB333 values, the module computes an approximate luma term and two chroma-like differences using only shifts, additions, and subtractions (Equation 12). The luma uses a divide-by-4 implemented as a right shift, and the chroma terms are signed channel differences that fit cleanly in small signed bit widths.
To allow on-board tuning without recompiling, the acceptable luma range is made adjustable via two signed 4-bit inputs, y_min_tune and y_max_tune. These signed nibbles offset the baseline Python luma bounds, then the results are clamped back into the valid 3-bit range [0,7] to prevent overflow/underflow (Equation 13). This expands or contracts the acceptable brightness band for skin pixels while leaving the chroma and hue-ordering constraints unchanged.

$$Y = \left\lfloor \frac{R_3 + 2G_3 + B_3}{4} \right\rfloor$$
$$Cr = R_3 - G_3$$
$$Cb = B_3 - G_3$$

Equation 12: YCrCb Colorspace Conversion

$$Y'_{\text{min}} = \text{clamp}_{[0,7]}(2 + y_{\text{min\_tune}})$$
$$Y'_{\text{max}} = \text{clamp}_{[0,7]}(6 + y_{\text{max\_tune}})$$

Equation 13: Tunable Skin-Detection Heuristics

A pixel is classified as skin only if it satisfies the tuned luma range, fixed chroma bounds, and simple RGB ordering constraints that enforce a plausible skin hue relationship in RGB333 space (Equation 14). All of these Boolean conditions are combined into a single is_skin flag.

$$\begin{aligned}\text{is\_skin} = (Y'_{\min} \le Y \le Y'_{\max}) \\ \wedge (1 \le Cr \le 4) \\ \wedge (-3 \le Cb \le 0) \\ \wedge (R_3 > G_3) \\ \wedge (R_3 - B_3 \ge 2) \\ \wedge (G_3 \ge B_3)\end{aligned}$$

Equation 14: Decision Rule for Skin Filtering

Finally, the decision is registered on the rising edge of clk. When enable is asserted, mask_pixel outputs 0xF for skin pixels and 0x0 otherwise; when enable is deasserted, the output is forced to 0x0 (Equation 15). This produces a clean binary mask suitable for downstream median, morphological, and spatial filtering stages.

$$\text{mask\_pixel} = \begin{cases} 15, & \text{if enable} = 1 \text{ and is\_skin} = 1 \\ 0, & \text{otherwise} \end{cases}$$

.

Equation 15: Decision Rule for Output Skin Mask

H. Spatial Filter

The spatial filter operates on the binary skin mask produced by the skin threshold module and enforces local spatial consistency in a 3x3 neighborhood. Its goal is to suppress isolated or weakly supported skin detections and retain only those pixels that are embedded within a sufficiently large foreground region. Each 4-bit pixel in the window is first reduced to a 1-bit flag by treating any nonzero nibble as "skin." The center pixel is stored as center_skin, while the eight neighbors (excluding the center) are mapped to flags c00, c01, c02, c10, c12, c20, c21, and c22. These neighbor bits are summed to form neighbors_skin_count, and the number of non-skin neighbors is computed as neighbors_non_skin = 8 - neighbors_skin_count.

The decision rule is simple: the center pixel is preserved as skin only if it is currently skin, and at least six of its eight neighbors are also skin. In implementation, this is expressed as keep_center = center_skin && (neighbors_non_skin < 3), which is equivalent to requiring neighbors_skin_count >= 6. If this condition holds, the output is set to 4'hF; otherwise, it is cleared to 4'h0. This threshold effectively removes small, spurious detections, thin appendages, and noisy edge pixels that are not well supported by their surroundings, while retaining coherent blobs that represent genuine skin regions.

From a hardware standpoint, the spatial filter is entirely combinational and uses only bitwise reductions, small adders, and a single comparison. The conversion from 4-bit nibbles to 1-bit flags is done with simple OR operations, and the neighbor count fits in a 4-bit signal since there are at most eight neighbors. Because it operates strictly on the already-binarized mask, the module avoids multipliers and large arithmetic units, making it compact and fast enough to sit inline in a high-throughput video pipeline. When enable is low, the module bypasses its logic and forces the output to 4'h0, allowing easy integration into a configurable preprocessing chain.

I. Centroid

The centroid detector module computes the center of mass of a connected foreground blob in a binary mask using streaming spatial moments, without storing a full frame. Each incoming pixel is accompanied by its coordinates (pixel_x, pixel_y) and a 4-bit mask_pixel that is either 0x0 (background) or nonzero (foreground). Whenever enable and pixel_valid are asserted and mask_pixel is nonzero, the module updates three running accumulators corresponding to the spatial moments defined in Equation 16: M00 is the foreground pixel count (blob area), M10 is the sum of x-coordinates over all foreground pixels, and M01 is the sum of y-coordinates over all foreground pixels. This accumulation is done in a single pass over the frame using only adders.

$$M_{00} = \sum_{(x,y)} f(x,y)$$

$$M_{10} = \sum_{(x,y)} x\, f(x,y)$$

$$M_{01} = \sum_{(x,y)} y\, f(x,y)$$

Equation 16: Accumulators for Centroid Detection

At the rising edge of frame_start, the module treats the accumulated moments as the measurements for the frame that just ended, computes the centroid using Equation 17, and registers centroid_x and centroid_y. Immediately after latching these results, the accumulators are reset to begin integrating the next frame. To guard against noise, the centroid is considered valid only if the blob area (M00) exceeds a minimum size threshold, as stated in Equation 19. If the blob is too small, centroid_valid is deasserted and the outputs are cleared.

$$C_x = \frac{M_{10}}{M_{00}}$$
$$C_y = \frac{M_{01}}{M_{00}}$$

Equation 17: Centroid X & Y Formulas

The implementation is dimensioned for a 640×480 frame in the worst case. The area accumulator (M00) is 20 bits wide (max 307,200), while the coordinate sums (M10 and M01) are 29 bits wide to hold the maximum possible sums. Division is performed once per frame when computing centroid_x = M10 / M00 and centroid_y = M01 / M00, and the results are truncated into 10-bit outputs that fit the frame dimensions.

$$x_{\min} = \min\{x \mid f(x,y) = 1\}, \quad x_{\max} = \max\{x \mid f(x,y) = 1\}$$
$$y_{\min} = \min\{y \mid f(x,y) = 1\}, \quad y_{\max} = \max\{y \mid f(x,y) = 1\}$$

Equation 18: Bounds Calculation Formulas

In parallel with moment accumulation, the module also tracks an axis-aligned bounding box for all foreground pixels. During the frame scan, it maintains running minima and maxima of pixel_x and pixel_y over foreground pixels; at the frame boundary, these are latched as bbox_min_x, bbox_min_y, bbox_max_x, and bbox_max_y when the blob passes the minimum-size test. Conceptually, the bounding box corresponds to the min/max definitions in Equation 18. All centroid, area, and bounding-box outputs are registered on the frame boundary, providing a stable set of measurements for an entire frame while the next frame's accumulators reset and begin integrating anew.

$$\text{centroid\_valid} = \begin{cases} 1, & M_{00} \geq \text{MIN\_BLOB\_SIZE} \\ 0, & \text{otherwise} \end{cases}$$

Equation 19: Centroid Validity Rule

J. Blob

The blob filter module uses the bounding box produced by the centroid detector to perform region-of-interest filtering on the binary mask. Its purpose is to keep only those pixels that lie within a padded bounding box around the primary detected blob, suppressing stray detections elsewhere in the frame while preserving the original mask values inside the region of interest. For each incoming pixel, the module compares

pixel_x and pixel_y against an expanded bounding box derived from bbox_min_x, bbox_min_y, bbox_max_x, and bbox_max_y. Padding amounts PADDING_X and PADDING_Y are added to all sides to give the blob some breathing room, ensuring that small motion or quantization jitter does not cause valid foreground regions to be clipped.

To maintain robustness at the edges of the frame, the module uses two small helper functions, saturating_sub and saturating_add, when expanding the bounding box. These functions ensure that padded coordinates do not underflow below zero or overflow beyond the frame maximum (FRAME_MAX_X = 639, FRAME_MAX_Y = 479). The padded bounds are computed as padded_min_x, padded_min_y, padded_max_x, and padded_max_y, and a simple set of comparisons determines whether the current pixel lies inside this expanded region. If enable is deasserted or bbox_valid is low, the module simply passes mask_pixel through unchanged, acting as a no-op. When both are asserted, any pixel inside the padded bounding box retains its original mask value, while pixels outside are forced to 0. This logic uses only adders, comparators, and a few multiplexers, making it extremely lightweight in hardware.

Algorithmically, the blob filter acts as a spatial gate keyed to the largest or most reliable blob identified by the centroid detector. It effectively discards noise and other skin-like regions that do not fall within the current frame's primary bounding box, making downstream processing, such as gesture recognition or object tracking, more stable and focused. Because it operates purely on the current pixel's coordinates and the previously computed bounding box, it integrates seamlessly into a streaming pipeline: no frame buffers or additional storage are required beyond the bounding-box registers already produced by the centroid stage.

K. Color Threshold

The color threshold module performs real-time color-based segmentation directly on the RGB444 camera stream using a Manhattan-distance (L1) similarity metric between the incoming pixel and a user-defined reference color. Each pixel arrives as three 4-bit channels (r_in, g_in, b_in), and the module compares this value against a 12-bit reference color ref_color = {ref_r, ref_g, ref_b}. In our design, ref_color is generated from the potentiometer input (with its bits interleaved across R/G/B to give a balanced 4-bit value per channel), allowing the user to "dial in" a target color at runtime. The module outputs a 4-bit binary mask: 4'hF for pixels considered close enough to the reference color, and 4'h0 otherwise. This provides a flexible way to isolate a colored object (e.g., a ball, marker, or glove) in hardware and feed it into the same binary-mask pipeline used elsewhere (spatial filtering, centroid, blob filtering, etc.).

Algorithmically, similarity is measured using Manhattan distance in RGB space: manhattan_dist = |r_in − ref_r| + |g_in − ref_g| + |b_in − ref_b|. Each per-channel absolute difference is in the range 0–15, so the total distance ranges from 0 (exact match) to 45 (maximally different across all channels). The computed manhattan_dist is then compared against a programmable 6-bit threshold provided by switches (0–45 recommended). If manhattan_dist <= threshold, the pixel is classified as a match and is_match is asserted.

From a hardware perspective, the design is lightweight and fully synthesizable without multipliers or dividers. The absolute-value operation is implemented using simple compare-and-subtract logic per channel, and the total Manhattan distance is formed by adding three zero-extended differences. The threshold comparison is a single less-than-or-equal operation. Finally, the mask output is registered on the rising edge of clk: when enable is asserted, mask_pixel is 4'hF for matching pixels and 4'h0 otherwise; when enable is deasserted, the output is forced to 4'h0

L. Gesture Recognition

Gesture recognition is implemented using an adaptive perceptron-based classifier that operates on blob statistics produced by the upstream skin/centroid pipeline. Once per frame, the module receives the current blob centroid (centroid_x, centroid_y), bounding box (bbox_min/max), and blob_area, along with a frame_start pulse that defines frame boundaries. Rather than performing expensive shape matching, the design extracts a small set of low-cost features that capture the main visual differences between gestures and feeds them into three parallel perceptrons representing the gesture classes: fist, open hand, and wave. Each perceptron computes an activation score as activation = bias + sum(weight[i] * feature[i]) and the module performs a winner-take-all decision, asserting the gesture whose activation is largest and exceeds a fixed threshold. This approach is fully streaming and hardware-friendly: it uses only adds, subtracts, small multiplies, and comparisons, and it produces stable, interpretable debug signals (activation values) that can be displayed on LEDs/hex for tuning.

Feature extraction is designed to be robust under real-time camera noise while remaining lightweight. Five features are computed and normalized to an 8-bit range (0–255) so that a single weight scale can be used. Compactness measures how filled the bounding box is and is based on the ratio blob_area / bbox_area (represented in Q8.8 style scaling), which distinguishes a tight fist from a spread hand. Velocity is computed from the per-frame change in centroid_x, using the absolute value to capture motion magnitude. Area is a coarse normalization of blob_area to represent hand size in the frame. Aspect ratio is derived from bbox_width / bbox_height and is approximated into representative values to indicate wide, tall, or square blobs. Finally, an oscillation feature detects wave-like

motion by counting direction changes of the centroid's horizontal movement across frames; this allows wave gestures to be separated from static poses even if the hand shape is similar.

The classifier supports online learning to adapt to different users and lighting conditions. In normal mode, the perceptrons use their current weights to classify gestures automatically. In learning mode (enabled by the state-specific switch), the buttons become teaching inputs: pressing btn[0], btn[1], or btn[2] indicates that the current frame should be labeled as FIST, OPEN, or WAVE, respectively. On a debounced button edge (and optionally for a short burst of consecutive frames to strengthen the update), the module compares the desired label to the current prediction and updates weights using a simple perceptron learning rule w = w + alpha * feature when the target class was missed, or w = w − alpha * feature when the class fired incorrectly (with a similar adjustment to the bias). The learning rate is implemented as a small fixed-point constant so updates can be performed with shifts/divides instead of floating-point arithmetic. To avoid flicker, the final gesture outputs apply a short hysteresis requirement: a gesture must be detected for multiple consecutive frames before the corresponding output is asserted.

V. Postprocessor

The postprocessor is responsible for modifying data as it comes out of the frame buffer and is sent to the hdmi output. Thus, given the pixel data, alongside other properties like center and bounding boxes from the centroid detector and blob filters, we can display overlays onto the screen. Performing these overlays after processing video helps decrease the total propagation delay prior to creating the frame buffer while still creating negligible overhead as the pixels are displayed.

A. Text Overlay

The text overlay is responsible for utilizing the IBM 437 Font Rom from Lab 7 and drawing text to the screen at a fixed location based on the control unit state. The control unit states are enumerated as a predefined array of text corresponding to each control unit state. The text overlay manager reads in the current input from the control unit and looks into the array to find which text should be drawn. Once it identifies which text to draw, it will overwrite pixel data with the text overlay on the output stream, thereby displaying the text data without damaging the raw camera frame.

B. Crosshair Overlay

The crosshair overlay simply draws a straight-line crosshair by utilizing the crosshair center. The centroid detector sends the centroid calculation to the crosshair overlay

generator. From there, the simple combination logic can be used to draw horizontal and vertical lines around the centroid.

C. Bounding Box Overlay

The bounding box overlay simply draws a bounding box around the filtered and binarized image. Once the bounding box has been generated, simple combinational logic can be used to draw a rectangle based on the coordinates of the bounding box.

D. Temporal Filtering

Note that we mentioned earlier that once images are converted to grayscale or binarized, the data can be heavily compressed, such that multiple frames can be stored in the frame buffer based on a particular output format. Using an FSM to treat the frame buffered pixels in a circular stack, we can perform one read to get 3 frames of data for grayscale images or 9 frames of data for binarized images. Temporal filtering is the process of utilizing past frames to process current data. In this sense, we can denoise 'static' areas of the image by taking the median of all the frames. This removes one-off outliers and staples any given pixel to its most common pixel, creating a flatter image frame.

E. Augmented Reality

The augmented reality portion of the design uses the outputs of the centroid detector and gesture recognition module to render interactive graphics directly onto the live video stream. Rather than performing additional image analysis, the AR logic treats the detected hand as an input device: the centroid provides a stable 2D position on the screen, while the recognized gestures (fist, open hand, wave) act as discrete control signals. This keeps the AR stage lightweight and deterministic, since all interaction is driven by already-computed metadata (centroid, bounding box, gesture flags) and simple state machines.

In the primary augmentation mode, the system draws simple geometric overlays (e.g., a square or circle) whose on-screen position is tied to the tracked centroid. A "grab/drag" interaction is implemented by gating motion updates with gesture state: when a fist is detected, the overlay is considered selected and its position is updated to follow the centroid; when the fist is not present, the overlay holds its last position. Additional gestures can be used as mode controls, such as changing the active shape or toggling interaction behavior, without requiring any new sensing beyond the existing gesture outputs.

A secondary augmentation mode implements a gesture-controlled whiteboard. In this mode, the video pipeline still displays the live camera image, but a persistent drawing layer is updated over time. The centroid acts like a brush tip whose position determines where to draw, and gestures act as simple commands such as enabling/disabling drawing,

clearing the canvas, or switching tools. Because the drawing layer is updated using straightforward pixel writes and small control logic, it integrates cleanly into the postprocessing pipeline and demonstrates an end-to-end interactive AR experience built entirely from centroid tracking and gesture classification

F. Pong

We implemented a video processing version of Pong where each player's hand acted as the paddle on screen. We accomplished this by setting up dual centroids by splitting the screen in half and independently calculating centroids for each half. Each centroid would be on their respective player's hand and would be depicted as a red dot on screen. This acts as visual feedback to the players so that they know where their hand is at and move accordingly. Where your hand is in the y-axis of the screen determines where the paddle position will be in the y-axis. We also have a gray dotted line on the middle of the screen to act as a reference line. This is to show if your hand/centroid is about to cross into your opponent's centroid calculation zone. Everything else was implemented as to how the game pong would be. The only other minor edit we made to our pong was that we added a powerup. The powerup gives you the ability to push the ball, increasing its speed. You first obtain the powerup after hitting the ball on your paddle 2 times in a row; you can tell if you got the powerup based on if your centroid is green. You activate it by pushing your hand towards the middle of the screen.

VI. Potentiometer Controller

The On-Board potentiometer is used as an additional control input for the control unit, which can be used to tune the threshold parameters. The potentiometer is an analog device, so it must be converted to a digital value to be utilized for calculations. The board has a built-in XADC IP to convert the analog value of the potentiometer to 16-bit value for board use. Note that the lower nibble has very little resolution with the rate at which the potentiometer turns, so the upper nibble is utilized to allow for more simplistic tuning.

VII. Motor Controller

We made our motors move by generating and using a PWM signal. The PWM period is 20 milliseconds. The motor turns fully clockwise when the duty cycle is at 5% and fully counterclockwise when the duty cycle is at 10%. We used the setup above as inspiration for how we would code our PWM signal. We have a counter that resets every PWM period (20 milliseconds). We then had an input data that we would compare with the counter value. When the counter value is less than the input data, the PWM signal is high. Once the counter value is greater than the input data, the PWM signal will be set to 0. We coded it so that we would just change the input data (adding to it or subtracting from it) when you want the motor to move. Adding to the input data would make the motor more

counterclockwise (which is left or up for our 2 motors). Subtracting from the input data would make the motors more clockwise (which is right or down for our 2 motors). We also coded a minimum and a maximum that the input data can be, as to not be out of the 5-10% duty cycle range.

Once we had PWM generation, we had to dictate which direction to send the motors based on centroid. To do this, we initially created a bang-bang controller which drilled the PWM left, right, up and down, by adjusting thresholds by 1 pixel based on which direction the camera would need to move to center the centroid. We noticed that this introduced a lot of jittering as the camera was never able to fully settle the centroid at the exact center. To resolve this issue, we added a deadzone in which motors would not be commanded to move until the centroid is a reasonable distance away from center. Finally, we noticed that the camera was a little slow, so we added a proportional controller to pwm generation allowing bigger steps to be taken if the target was further away. This allowed for faster convergence with the target. Finally, we noticed with this increased convergence rate, the camera might start from rest to high speed very quickly creating tearing in video and destroying data on the objects being used as target, causing the tracking process to fail altogether. To fix this, we added a slew rate which limits the max change we can make to pwm thresholds on any given step. Together, the slew rate allows for steady acceleration while the proportional controller created fast convergence with the target. The deadzone stabilized camera control to prevent camera jitter on stabilized objects.

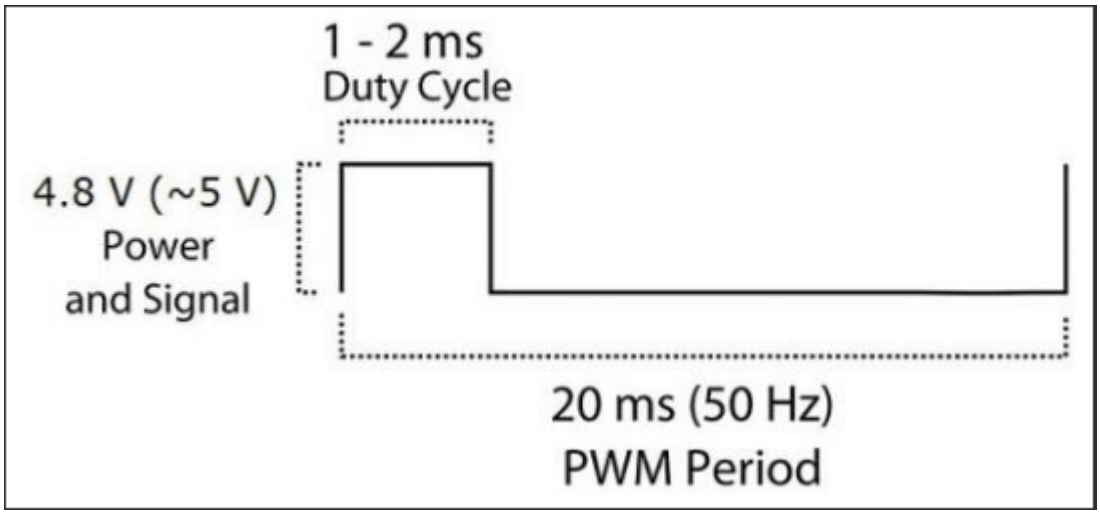

Figure 10: PWM Timing Diagram for MG996 Motor

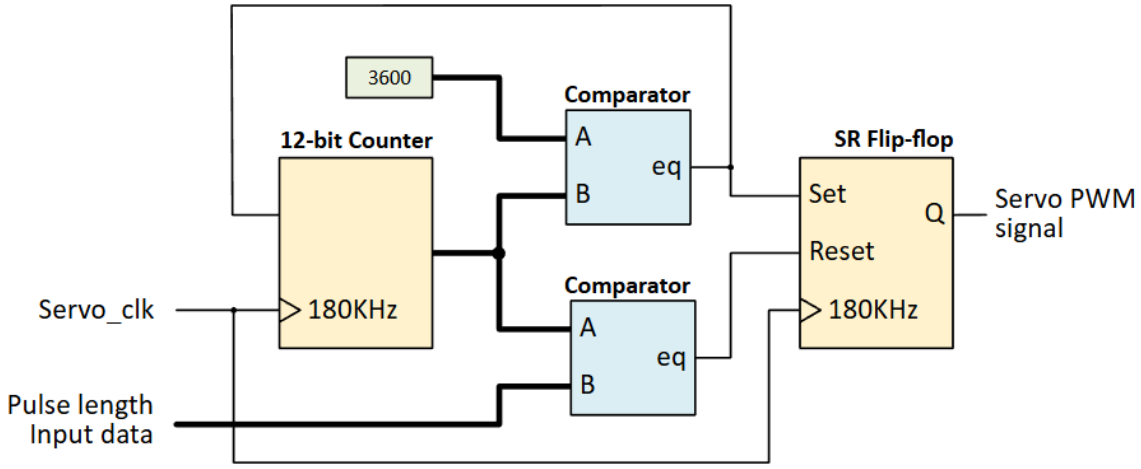

Figure 11: RealDigital PWM Signal Generation Architecture

VIII. Control Unit

The control unit is designed mainly for demoing the processor. We created a streamlined slideshow system which can be shifted to different states. Each different control unit states showcasing different filters being applied to the raw image. The raw or grayscale image can always be overlayed at any given state for comparison without the FSM going back to the raw or grayscale state again. The first button moves to the previous slide, and the second button moves to the next slide. The third button forces an overlay for comparison, with the overlay type (grayscale vs color) being determined by switch 15. Finally, the last button is the reset button for the system, which resets the vga controller and reinitializes the camera by rewriting the camera's control registers. The potentiometer is piped to relevant modules as control input, along with relevant switch values that can be used for state control. The order and enumeration of states in the control unit are summarized in Figure 11.

```
// ------------------------------------------------------------
// Demo State Enumeration
// ------------------------------------------------------------
typedef enum logic [4:0] {
    RAW          = 5'd0,   // Raw color output
    DITHER       = 5'd1,   // Dithered color compression (RGB444 -> RGB333 with dithering)
    CHANNEL_MODE = 5'd2,   // Individual channel display (controlled by sw[2:0])
    GRAYSCALE    = 5'd3,   // Grayscale converted output
    THRESHOLD    = 5'd4,   // Binary threshold on grayscale
    TEMPORAL     = 5'd5,   // Temporal filter: moving average across frames to reduce noise
    MEDIAN       = 5'd6,   // Median filter then threshold
    GAUSSIAN     = 5'd7,   // Gaussian blur (convolution, sw[15] for threshold)
    SOBEL        = 5'd8,   // Sobel edge detection with threshold
    SHARPEN      = 5'd9,   // Sharpen filter (sw[15] for threshold)
    EMBOSS       = 5'd10,  // Emboss 3D relief effect (sw[15] for threshold)
    SKIN         = 5'd11,  // Skin thresholding (binary mask)
    SPATIAL      = 5'd12,  // 3x3 spatial noise filter on skin mask
    ERODE        = 5'd13,  // Erosion: remove small white specs
    DILATE       = 5'd14,  // Dilation (opening visualization after erosion)
    CENTROID     = 5'd15,  // Centroid detection (after opening)
    BLOB         = 5'd16,  // Blob filtering around centroid
    SKIN_MOTION  = 5'd17,  // Skin motion tracking (motors enabled)
    COLOR_THRESH = 5'd18,  // Color thresholding (Hamming distance)
    COLOR_TRACK  = 5'd19,  // Color tracking + centroid + motion
    GESTURE      = 5'd20,  // Gesture detection based on blob/centroid pipeline
    AUGMENT      = 5'd23,  // Augment mode: square overlay with gesture-controlled dragging
    DUAL_CENT    = 5'd21,  // Dual centroid (left/right halves)
    PONG         = 5'd22   // Pong game mode
} demo_state_t;
```

Figure 12: Control Unit State Enumeration

IX. Code

The following table summarizes the code used to the video processor on the FPGA

System Verilog File Summaries
Module: processor.sv **Inputs:** clk, rst, [2:0] btn, [15:0] sw, VP, VN, pclk, [7:0] pix_byte, vsync, href, inout siod **Outputs:** [7:0] hex_segA, [7:0] hex_segB, [3:0] hex_gridA, [3:0] hex_gridB, [15:0] led, cam_reset, pwdn, xclk, sioc, pwm_pan, pwm_tilt, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n, [2:0] hdmi_tmds_data_p **Description:** Top-level processor coordinating camera capture, image processing, and HDMI display. Integrates all subsystems: camera controller, preprocessor, buffer, postprocessor, motors. **Purpose:** Main system integration and clock domain crossing management. Routes control signals and data between all major components. **Modifications:** None
Module: sync.sv **Inputs:** clk, d **Outputs:** q **Description:** Synchronizer w/ debouncer for asynchronous, noisy pushbutton inputs using a double-flop and stability counter.

Purpose: Provides stable, metastability-safe button signals for the rest of the design.
Modifications: None

Module: control_unit.sv
Inputs: clk, rst_n, [2:0] btn_sync, [15:0] sw_sync, [15:0] pot_in
Outputs: grayscale_enable, force_color, force_grayscale, channel_mode_enable, [2:0] channel_select, threshold_enable, [3:0] threshold_value, median_enable, convolution_enable, [1:0] kernel_select, skin_threshold_enable, spatial_filter_enable, erosion_enable, dilation_enable, centroid_enable, blob_filter_enable, motion_enable, [3:0] skin_y_min, [3:0] skin_y_max, color_threshold_enable, [11:0] ref_color, [5:0] color_threshold_value, temporal_filter_enable, gesture_enable, augment_enable, split_centroid_enable, dither_enable, [4:0] current_state_num, transition_trigger, [2:0] temporal_tune
Description: FSM that cycles through image processing demo modes. Controls all processing stages via enable signals.
Purpose: Central state machine for demo sequencing and mode selection. Provides user control over processing pipeline via buttons and switches.
Modifications: None

Module: hex_driver.sv
Inputs: clk, reset, [3:0] in[4]
Outputs: [7:0] hex_seg, [3:0] hex_grid
Description: Multiplexed 4-digit seven-segment display driver that converts four 4-bit nibbles into segment patterns and scans the active digit using a counter.
Purpose: Drive the on-board hex displays for debug/status output (e.g., potentiometer value, centroid position).
Modifications: None

Module: pot_controller.sv
Inputs: clk, reset, VP, VN
Outputs: [15:0] pot_out
Description: Potentiometer controller using XADC for analog input. Reads analog voltage from auxiliary channel 3.
Purpose: Provides analog input for tunable parameters (thresholds, color selection, etc.). Interfaces with XADC IP core for ADC conversion.
Modifications: None

Module: cam_controller.sv
Inputs: clk, rst_n_clk, rst_n_pclk, cam_start, pclk, [7:0] pix_byte, vsync, href, inout siod
Outputs: cam_done, cam_reset, cam_pwdn, sioc, [11:0] pix_data, [18:0] pix_addr, pix_write
Description: Top-level coordinator for OV7670 camera system. Manages initialization via SCCB and pixel capture to memory.
Purpose: Instantiate and connect camera initializer and interface modules. Provide unified control interface for camera system.
Modifications: None

Module: cam_initializer.sv
Inputs: clk, rst_n, cam_init_start, inout siod
Outputs: cam_init_done, sioc, data_sent_done, [7:0] sccb_dout

Description: Configures OV7670 camera registers via SCCB protocol. Sequences through ROM-stored register values with timing delays.
Purpose: Initialize camera with specific settings for RGB444 output mode. Handles SCCB write transactions and inter-command delays.
Modifications: None

Module: cam_rom.sv
Inputs: clk, reset, [7:0] addr
Outputs: [15:0] dout
Description: ROM containing OV7670 register address/value pairs and special delay/end markers used by the camera initializer.
Purpose: Provide a deterministic, synthesizable sequence of camera configuration writes to bring the OV7670 into RGB444 640x480 operation.
Modifications: None

Module: sccb_master.sv
Inputs: clk, rst, start, read, [6:0] dev_addr, [7:0] reg_addr, [7:0] wr_data, inout siod
Outputs: ready, done, [7:0] rd_data, sioc
Description: Serial Camera Control Bus (SCCB) master controller for OV7670 camera configuration. Implements write/read protocols with configurable clock generation.
Purpose: Generate SIOC from the FPGA clock and execute SCCB transactions via a start/ready/done handshake.
Modifications: None

Module: cam_interface.sv
Inputs: pclk, vsync, href, [7:0] pixel_data, cam_init_done
Outputs: [18:0] mem_addr, [11:0] mem_data, mem_write, [9:0] line_y, line_ready_pulse
Description: Captures pixel data from OV7670 camera and writes to BRAM. Assembles 8-bit byte stream into 12-bit RGB444 pixels.
Purpose: Bridge between camera's pixel clock domain and memory interface. Handle byte-to-pixel conversion and address generation.
Modifications: None

Module: preprocessor.sv
Inputs: clk, rst_n, [11:0] cam_pix_data, [18:0] cam_pix_addr, cam_pix_write, cam_vsync, grayscale_enable, threshold_enable, [3:0] threshold_value, median_enable, convolution_enable, [1:0] kernel_select, skin_threshold_enable, spatial_filter_enable, erosion_enable, dilation_enable, centroid_enable, blob_filter_enable, gesture_enable, split_centroid_enable, [3:0] skin_y_min, [3:0] skin_y_max, color_threshold_enable, [11:0] ref_color, [5:0] color_threshold_value, learning_mode, [2:0] learning_buttons
Outputs: [2:0] processed_pixel, [18:0] processed_addr, processed_valid, is_border_pixel, [9:0] centroid_x, [9:0] centroid_y, centroid_valid, [19:0] blob_area, [9:0] centroid_bbox_min_x, [9:0] centroid_bbox_min_y, [9:0] centroid_bbox_max_x, [9:0] centroid_bbox_max_y, [9:0] left_centroid_x, [9:0] left_centroid_y, left_centroid_valid, [9:0] left_bbox_min_x, [9:0] left_bbox_min_y, [9:0] left_bbox_max_x, [9:0] left_bbox_max_y, [9:0] right_centroid_x, [9:0] right_centroid_y, right_centroid_valid, [9:0] right_bbox_min_x, [9:0] right_bbox_min_y, [9:0] right_bbox_max_x, [9:0] right_bbox_max_y, gesture_fist, gesture_open, gesture_wave
Description: Complete image preprocessing pipeline before frame buffer. Camera RGB -> Grayscale -> Spatial Filters -> Morphology -> Centroid -> Gesture.

Purpose: All real-time image processing for camera feed. Outputs processed pixels and tracking metadata (centroid, gestures).
Modifications: None

Module: grayscale_converter.sv
Inputs: clk, enable, [11:0] rgb_in
Outputs: [11:0] data_out
Description: Converts 12-bit RGB444 format to grayscale using luminance formula. Replicates grayscale value across RGB channels for display.
Purpose: Transform color image to grayscale for processing pipeline. Preserves perceived brightness using weighted channel contributions.
Modifications: None

Module: skin_threshold.sv
Inputs: clk, enable, [3:0] r_in, [3:0] g_in, [3:0] b_in, [3:0] y_min_tune, [3:0] y_max_tune
Outputs: [3:0] mask_pixel
Description: Detects skin pixels using RGB-based color space thresholds. Implements Python-equivalent logic ported to FPGA hardware.
Purpose: Create binary skin mask for hand tracking and gesture detection. Filter non-skin regions from image for blob analysis.
Modifications: None

Module: color_threshold.sv
Inputs: clk, enable, [3:0] r_in, [3:0] g_in, [3:0] b_in, [11:0] ref_color, [5:0] threshold
Outputs: [3:0] mask_pixel
Description: Performs color-based segmentation using Manhattan distance (L1 norm). Compares input RGB pixels against a user-defined reference color.
Purpose: Track colored objects and create a binary mask based on color similarity.
Modifications: None

Module: line_buffer.sv
Inputs: clk, rst_n, [3:0] pixel_in, pixel_valid, frame_start
Outputs: [3:0] p00, [3:0] p01, [3:0] p02, [3:0] p10, [3:0] p11, [3:0] p12, [3:0] p20, [3:0] p21, [3:0] p22, neighborhood_valid, [9:0] out_x, [9:0] out_y
Description: Stores 3 rows of image data for spatial filtering operations. Provides 3x3 pixel neighborhood window for convolution/median filtering.
Purpose: Enable spatial operations by buffering multiple scan lines to support real-time filtering without a full frame buffer.
Modifications: None

Module: spatial_filter.sv
Inputs: clk, enable, [3:0] p00, [3:0] p01, [3:0] p02, [3:0] p10, [3:0] p11, [3:0] p12, [3:0] p20, [3:0] p21, [3:0] p22
Outputs: [3:0] pixel_out
Description: Noise reduction filter for binary masks using 3x3 neighborhood analysis to remove isolated detections.
Purpose: Improve blob quality before morphology/centroid detection by suppressing small noise specs.
Modifications: None

Module: morphological_filter.sv

Inputs: clk, erosion_enable, dilation_enable, [3:0] p00, [3:0] p01, [3:0] p02, [3:0] p10, [3:0] p11, [3:0] p12, [3:0] p20, [3:0] p21, [3:0] p22
Outputs: [3:0] pixel_out
Description: Implements erosion and dilation operations for binary masks to remove noise and restore blob size through morphological opening.
Purpose: Clean up binary masks before centroid detection by filtering small false positives while preserving the main blob.
Modifications: None

Module: blob_filter.sv
Inputs: clk, rst_n, enable, [3:0] mask_pixel, [9:0] pixel_x, [9:0] pixel_y, [9:0] bbox_min_x, [9:0] bbox_min_y, [9:0] bbox_max_x, [9:0] bbox_max_y, bbox_valid, frame_start
Outputs: [3:0] pixel_out
Description: Filters the blob mask to a padded region around a previously detected bounding box, with adaptive padding and bbox persistence.
Purpose: Reduce false positives by restricting processing to a region of interest and smoothing detection loss across frames.
Modifications: None

Module: centroid_detector.sv
Inputs: clk, rst_n, enable, [3:0] mask_pixel, [9:0] pixel_x, [9:0] pixel_y, pixel_valid, frame_start
Outputs: [9:0] centroid_x, [9:0] centroid_y, centroid_valid, [19:0] blob_area, [9:0] bbox_min_x, [9:0] bbox_min_y, [9:0] bbox_max_x, [9:0] bbox_max_y
Description: Computes center of mass of white pixels in a binary mask and produces centroid + bounding box using spatial moments.
Purpose: Track object position for overlays/motion control while rejecting small noise blobs via a minimum size threshold.
Modifications: None

Module: split_centroid_detector.sv
Inputs: clk, rst_n, enable, [3:0] mask_pixel, [9:0] pixel_x, [9:0] pixel_y, pixel_valid, frame_start
Outputs: [9:0] left_centroid_x, [9:0] left_centroid_y, left_centroid_valid, [19:0] left_blob_area, [9:0] left_bbox_min_x, [9:0] left_bbox_min_y, [9:0] left_bbox_max_x, [9:0] left_bbox_max_y, [9:0] right_centroid_x, [9:0] right_centroid_y, right_centroid_valid, [19:0] right_blob_area, [9:0] right_bbox_min_x, [9:0] right_bbox_min_y, [9:0] right_bbox_max_x, [9:0] right_bbox_max_y
Description: Computes two centroids (left and right halves) from a binary mask stream using separate moment accumulators and bounding boxes.
Purpose: Provide dual hand/object tracking (left/right) for two-player interaction modes such as Pong.
Modifications: None

Module: median_filter.sv
Inputs: clk, enable, [3:0] p00, [3:0] p01, [3:0] p02, [3:0] p10, [3:0] p11, [3:0] p12, [3:0] p20, [3:0] p21, [3:0] p22
Outputs: [3:0] pixel_out
Description: Performs 3x3 median filtering for noise reduction using a sorting network to find the median of 9 pixels.

Purpose: Remove salt-and-pepper noise while preserving edges before thresholding/edge detection.
Modifications: None

Module: convolution_kernel.sv
Inputs: clk, enable, [1:0] kernel_select, [3:0] p00, [3:0] p01, [3:0] p02, [3:0] p10, [3:0] p11, [3:0] p12, [3:0] p20, [3:0] p21, [3:0] p22
Outputs: [3:0] pixel_out
Description: Applies selectable 3x3 convolution kernels (Gaussian, Sobel, Sharpen, Emboss) for spatial filtering and edge detection.
Purpose: Provide configurable spatial processing effects as part of the grayscale pipeline.
Modifications: None

Module: threshold.sv
Inputs: enable, [3:0] threshold_value, [11:0] rgb_in
Outputs: [11:0] rgb_out
Description: Applies binary thresholding to grayscale images, producing a black/white mask based on a threshold value.
Purpose: Segment image regions by intensity to create binary masks for later blob processing.
Modifications: None

Module: gesture_perceptron.sv
Inputs: clk, rst_n, enable, frame_start, centroid_valid, [9:0] centroid_x, [9:0] centroid_y, [9:0] bbox_min_x, [9:0] bbox_min_y, [9:0] bbox_max_x, [9:0] bbox_max_y, [19:0] blob_area, learning_mode, [2:0] learning_buttons
Outputs: fist, open_hand, wave, [7:0] debug_activation_fist, [7:0] debug_activation_open, [7:0] debug_activation_wave
Description: Adaptive perceptron-based gesture classifier with online learning using blob features (compactness, velocity, area, aspect ratio, oscillation).
Purpose: Real-time gesture recognition (fist/open/wave) with optional user teaching via button inputs during learning mode.
Modifications: None

Module: buffer.sv
Inputs: clk_25m, rst_n, pclk, [9:0] draw_x, [9:0] draw_y, vde, [2:0] processed_pixel, [18:0] processed_addr, processed_valid, processing_enable, temporal_filter_enable, dither_enable, dither_type, force_color, [11:0] cam_pix_data, [18:0] cam_pix_addr, cam_pix_write, cam_vsync, [9:0] cam_line_y, cam_line_ready, compare_mode, transition_trigger
Outputs: [8:0] vga_pix_data, [1:0] frame_chunk_counter
Description: Frame buffer with dual-port BRAM for video storage. Port A writes from camera/preprocessor (pclk) and Port B reads for VGA (clk_25m).
Purpose: Bridge the camera and VGA clock domains and support temporal filtering, compare mode, dithering, and tear-guarded reads.
Modifications: None

Module: vga_controller.sv
Inputs: pixel_clk, reset
Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY
Description: Generates 640x480@60Hz VGA timing signals (hsync/vsync, blanking, and pixel coordinates).

Purpose: Provide raster timing and pixel coordinates used to read the frame buffer and to drive the HDMI/VGA output pipeline.
Modifications: None

Module: postprocessor.sv
Inputs: clk, rst_n, channel_mode_enable, [2:0] channel_select, overlay_enable, bbox_overlay_enable, processing_enable, temporal_filter_enable, [1:0] frame_chunk_counter, force_color, force_grayscale, [4:0] current_state, gesture_enable, augment_enable, split_centroid_enable, dither_type, augment_mode, [2:0] motion_threshold, [9:0] draw_x, [9:0] draw_y, vde, [9:0] centroid_x, [9:0] centroid_y, centroid_valid, [9:0] centroid_bbox_min_x, [9:0] centroid_bbox_min_y, [9:0] centroid_bbox_max_x, [9:0] centroid_bbox_max_y, [9:0] left_centroid_x, [9:0] left_centroid_y, left_centroid_valid, [9:0] left_bbox_min_x, [9:0] left_bbox_min_y, [9:0] left_bbox_max_x, [9:0] left_bbox_max_y, [9:0] right_centroid_x, [9:0] right_centroid_y, right_centroid_valid, [9:0] right_bbox_min_x, [9:0] right_bbox_min_y, [9:0] right_bbox_max_x, [9:0] right_bbox_max_y, gesture_fist, gesture_open, gesture_wave, [2:0] btn_sync, [3:0] rng, [2:0] pixel_red_in, [2:0] pixel_green_in, [2:0] pixel_blue_in
Outputs: [2:0] pixel_red_out, [2:0] pixel_green_out, [2:0] pixel_blue_out
Description: Post-processing pipeline between frame buffer and HDMI output. Handles channel selection, text overlay, and graphical overlays.
Purpose: Final stage before display: visual enhancements and debug information. Adds state labels, gesture text, crosshair, bounding boxes, augmentation, and pong rendering.
Modifications: None

Module: temporal_filter.sv
Inputs: [2:0] pixel_red_in, [2:0] pixel_green_in, [2:0] pixel_blue_in, [1:0] frame_chunk_counter, [2:0] motion_threshold
Outputs: [2:0] red_out, [2:0] blue_out, [2:0] green_out
Description: Computes a robust temporal visualization from three packed 3-bit frames and maps motion intensity to a heatmap-like RGB output.
Purpose: Visualize motion / temporal differences across frames (and provide a stable per-pixel temporal output for the display pipeline).
Modifications: None

Module: channel_selector.sv
Inputs: enable, [2:0] channel_select, [2:0] red_in, [2:0] green_in, [2:0] blue_in
Outputs: [2:0] red_out, [2:0] green_out, [2:0] blue_out
Description: Selectively displays RGB color channels based on control signals to enable independent viewing of R/G/B contributions.
Purpose: Debug and demonstrate channel contributions by masking chosen channels while passing others through.
Modifications: None

Module: text_overlay.sv
Inputs: clk, rst_n, [4:0] current_state, force_color, force_grayscale, [1:0] gesture_code, dither_type, augment_mode, channel_mode_enable, [2:0] channel_select, [3:0] pong_player_a_score, [3:0] pong_player_b_score, pong_player_a_wins, pong_player_b_wins, [9:0] draw_x, [9:0] draw_y, vde, [2:0] pixel_red_in, [2:0] pixel_green_in, [2:0] pixel_blue_in
Outputs: [2:0] pixel_red_out, [2:0] pixel_green_out, [2:0] pixel_blue_out

Description: Renders state/mode text (and gesture + pong status) using an 8x16 font ROM and overlays it on the bottom of the video output.
Purpose: Provide on-screen debug/UI feedback so the user can see the current processing mode and interaction status.
Modifications: None

Module: font_rom.sv
Inputs: [10:0] addr
Outputs: [7:0] data
Description: Read-only font bitmap ROM addressed by character/row index for text rendering.
Purpose: Provide glyph pixel patterns to `text_overlay` for drawing characters on-screen.
Modifications: None

Module: overlay_manager.sv
Inputs: clk, rst_n, enable, bbox_enable, use_dual_centroids, [9:0] centroid_x, [9:0] centroid_y, centroid_valid, [9:0] centroid2_x, [9:0] centroid2_y, centroid2_valid, [9:0] bbox_min_x, [9:0] bbox_min_y, [9:0] bbox_max_x, [9:0] bbox_max_y, [9:0] bbox2_min_x, [9:0] bbox2_min_y, [9:0] bbox2_max_x, [9:0] bbox2_max_y, [9:0] draw_x, [9:0] draw_y, vde, [2:0] pixel_red_in, [2:0] pixel_green_in, [2:0] pixel_blue_in
Outputs: [2:0] pixel_red_out, [2:0] pixel_green_out, [2:0] pixel_blue_out
Description: Draws graphical overlays (crosshair and bounding boxes) on the video stream based on centroid and bbox metadata.
Purpose: Provide visual feedback for tracking performance and aid debugging/verification of centroid + blob logic.
Modifications: None

Module: augment.sv
Inputs: clk, rst_n, enable, augment_mode, [9:0] centroid_x, [9:0] centroid_y, centroid_valid, gesture_fist, gesture_open, gesture_wave, [9:0] draw_x, [9:0] draw_y, vde, [2:0] pixel_red_in, [2:0] pixel_green_in, [2:0] pixel_blue_in
Outputs: [2:0] pixel_red_out, [2:0] pixel_green_out, [2:0] pixel_blue_out
Description: Dual-mode augmentation overlay: shape dragging (square/circle) and a modal whiteboard controlled by gestures.
Purpose: Demonstrate gesture-driven interaction by drawing interactive overlays on top of the live video stream.
Modifications: None

Module: pong.sv
Inputs: clk, rst_n, enable, [9:0] draw_x, [9:0] draw_y, vde, frame_start, rematch_pressed, [3:0] rng, [9:0] left_centroid_y, left_centroid_valid, [9:0] right_centroid_y, right_centroid_valid, [9:0] left_centroid_x, [9:0] right_centroid_x, [2:0] pixel_red_in, [2:0] pixel_green_in, [2:0] pixel_blue_in
Outputs: [2:0] pixel_red_out, [2:0] pixel_green_out, [2:0] pixel_blue_out, [3:0] player_a_score, [3:0] player_b_score, player_a_wins, player_b_wins
Description: Two-player Pong game rendered into the video stream; paddles are controlled by tracked hand centroids, and scores/win state are maintained in-game.
Purpose: Interactive demo showcasing dual centroid tracking and real-time control by mapping tracked hand motion to gameplay.
Modifications: None

Module: motor_controller.sv **Inputs:** clk, rst_n, enable, [9:0] centroid_x, [9:0] centroid_y, centroid_valid **Outputs:** pwm_pan, pwm_tilt, [3:0] motor_direction, [9:0] error_x, [9:0] error_y **Description:** High-level pan/tilt control that computes direction and step sizes from centroid error relative to screen center. **Purpose:** Enable servo-based object tracking by commanding the low-level PWM driver to recenter the tracked object. Modifications: None
Module: motor_driver.sv **Inputs:** clk, rst, enable, [3:0] direction, [18:0] pan_step, [18:0] tilt_step **Outputs:** pwm_pan, pwm_tilt, [18:0] pan_threshold, [18:0] tilt_threshold **Description:** Low-level servo PWM generator that updates pulse-width thresholds based on direction and step sizes. **Purpose:** Convert pan/tilt movement commands into standard 50Hz servo PWM signals and maintain position state. Modifications: None

Table 2: File Summaries

Results

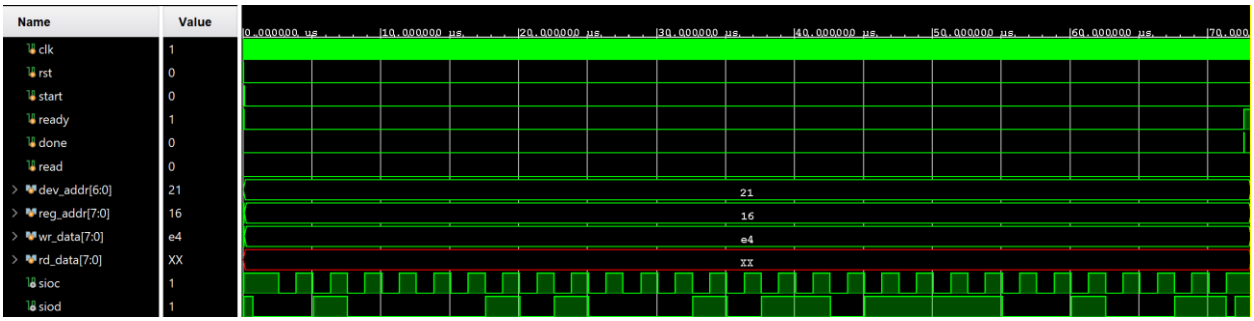

Figure 13: SCCB Transaction Testbench

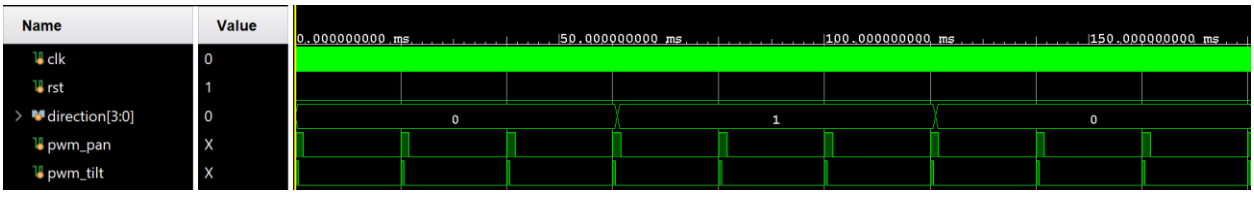

Figure 14: PWM Timing Waveform

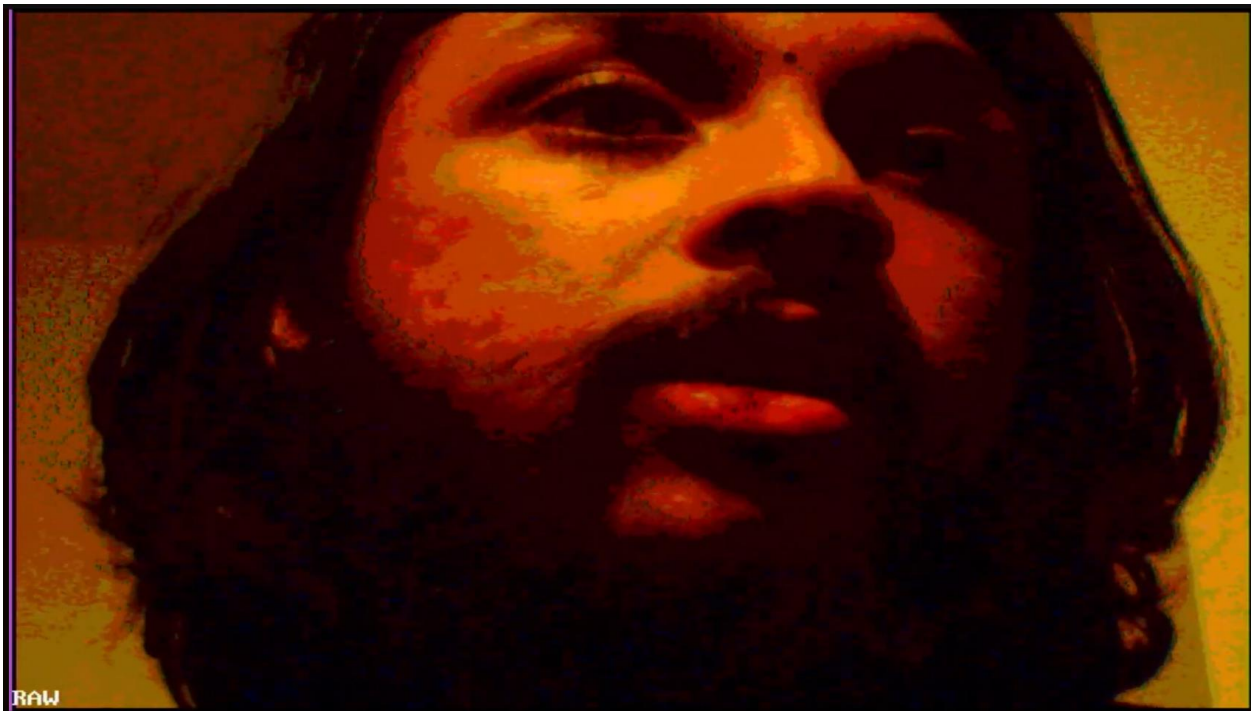

Figure 15:Raw Video Output from Video Processor

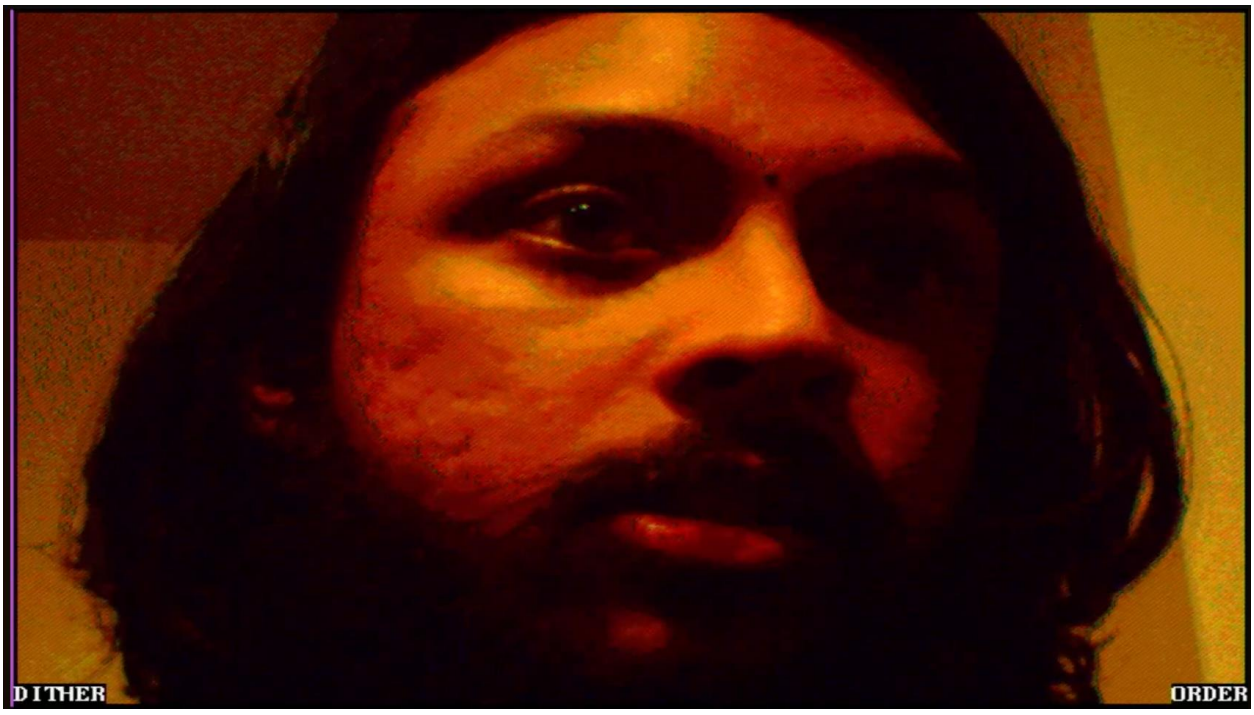

Figure 16:Ordered Dithering Video Output

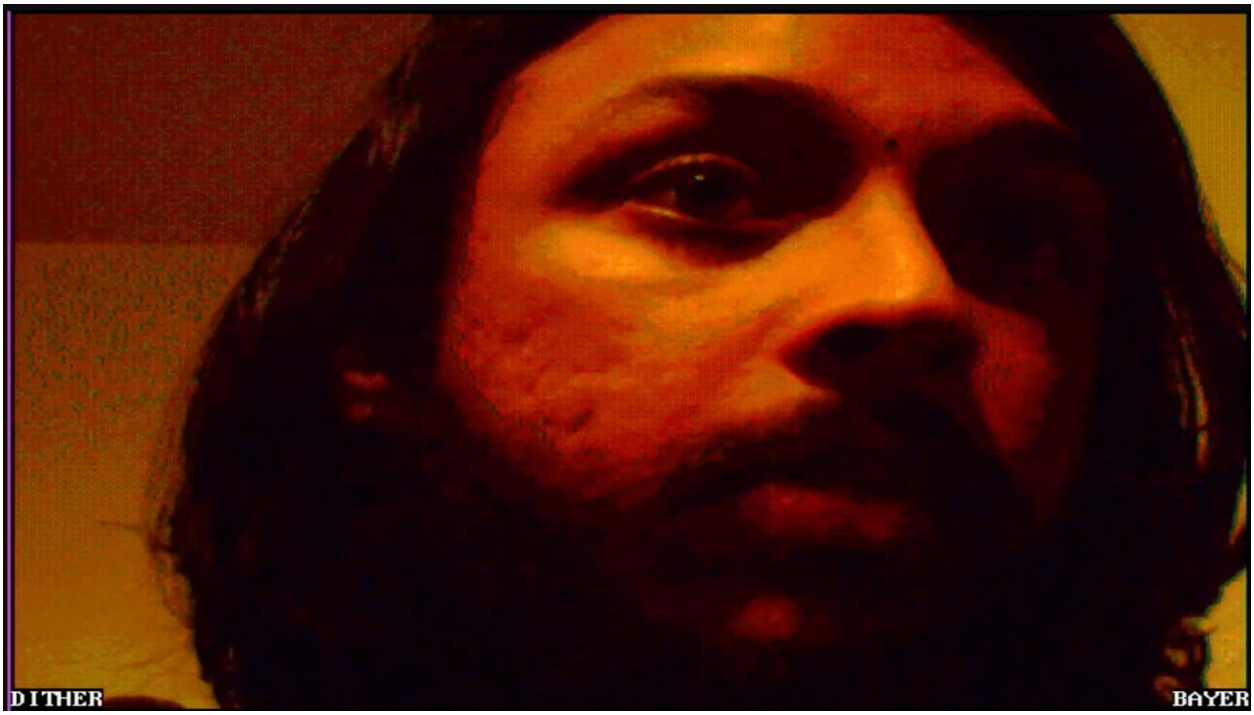

Figure 17:Bayer Dithering Video Output

Figure 18:Channel Selection Video Output

Figure 19:Grayscale Video Output

Figure 20:Thresholded Video Output

Figure 21:Thresholded Video Output with Split-Screen Grayscale Overlay

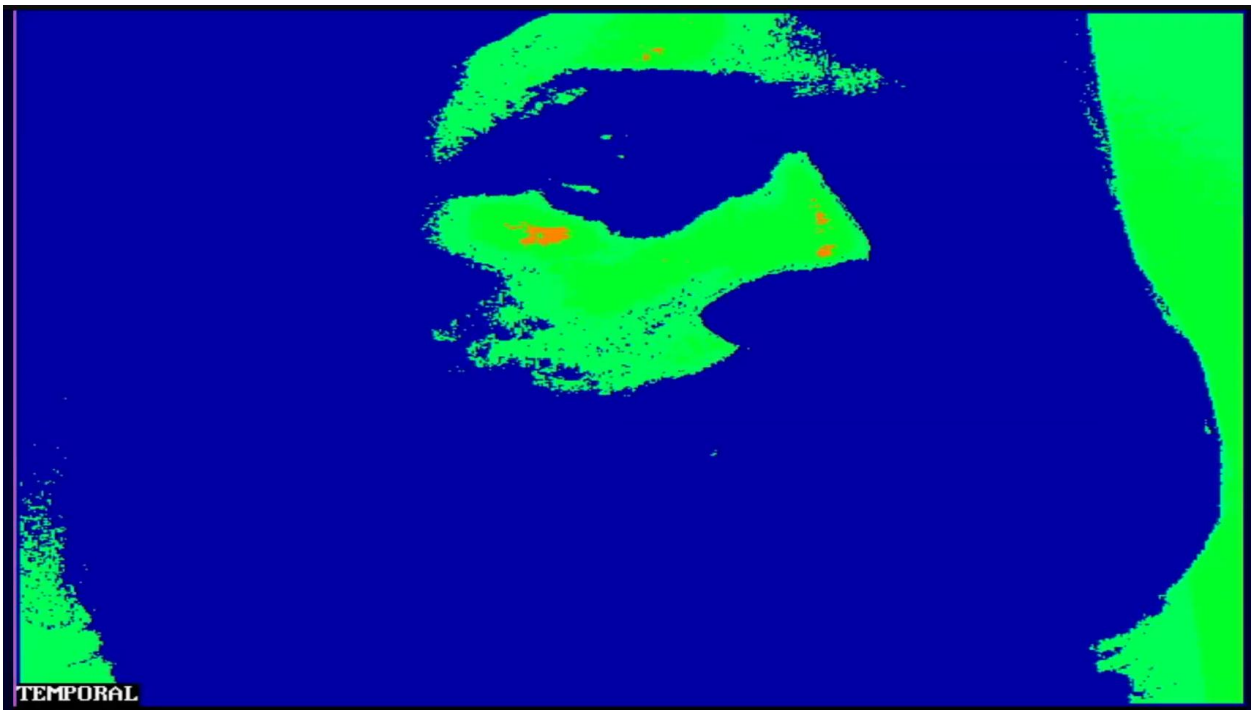

Figure 22:Temporal Motion Filter Video Output

Figure 23:Gaussian Filter Video Output

Figure 24:Sobel Filter Video Output

Figure 25:Sharpen Video Output with Split-Screen Grayscale Overlay

Figure 26: Emboss Video Output

Figure 27: Skin Thresholded Video Output

Figure 28:Spatial Filter Video Output

Figure 29:Erosion Filter Video Output

Figure 30:Dilation Filter Video Output

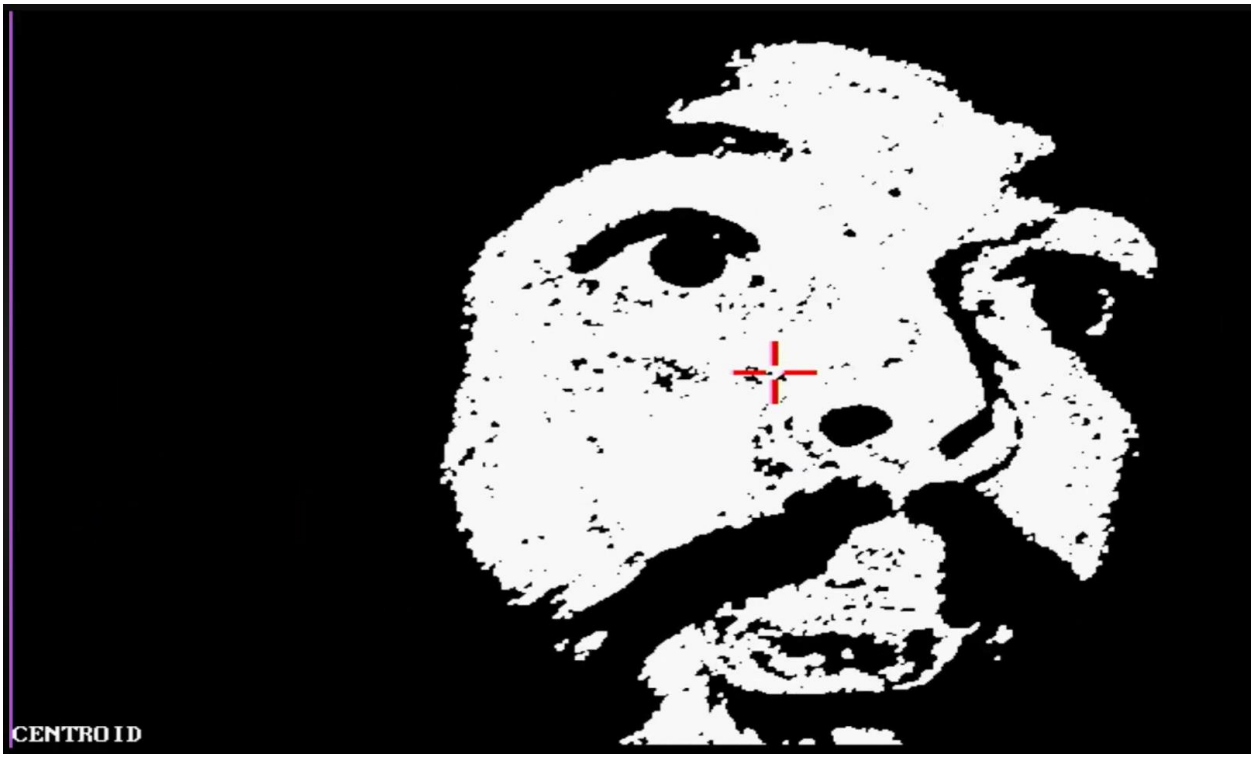

Figure 31: Centroid Detection Video Output

Figure 32:Centroid Detection Video Output with Raw Overlay

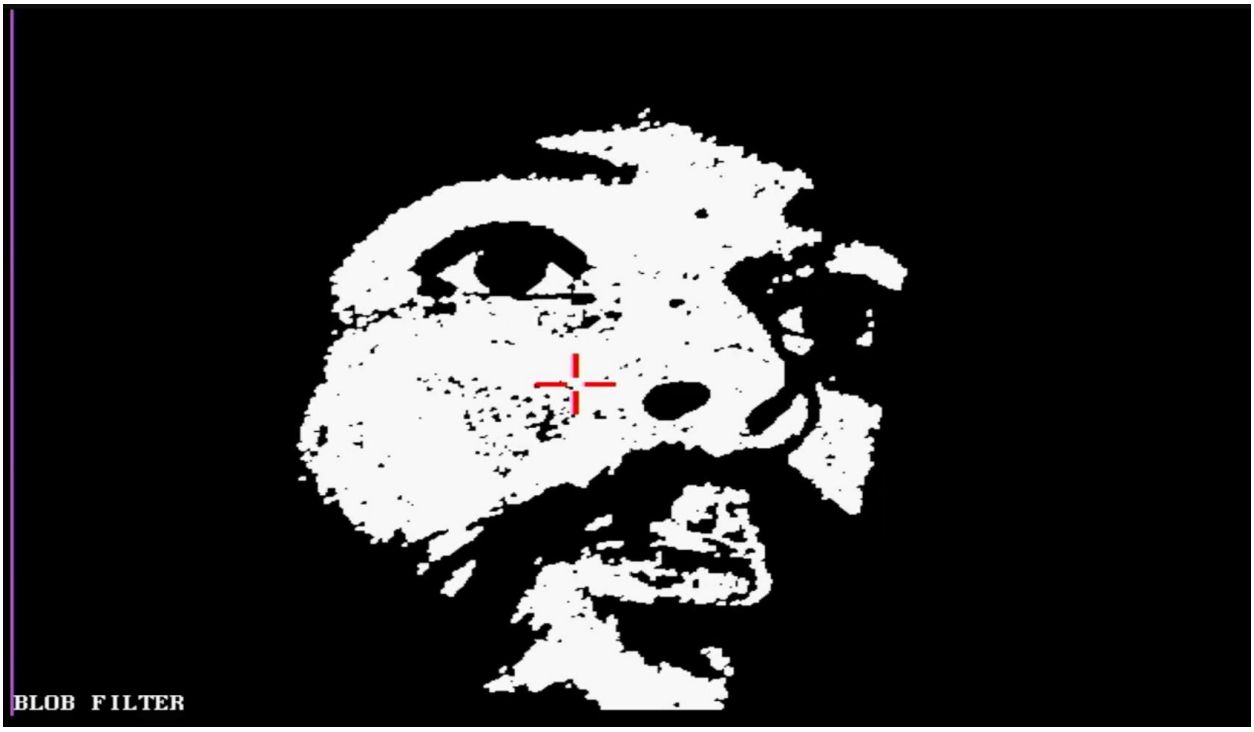

Figure 33: Blob Filter Video Output

Figure 34:Blob Filter Video Output with Bounding Box Overlay

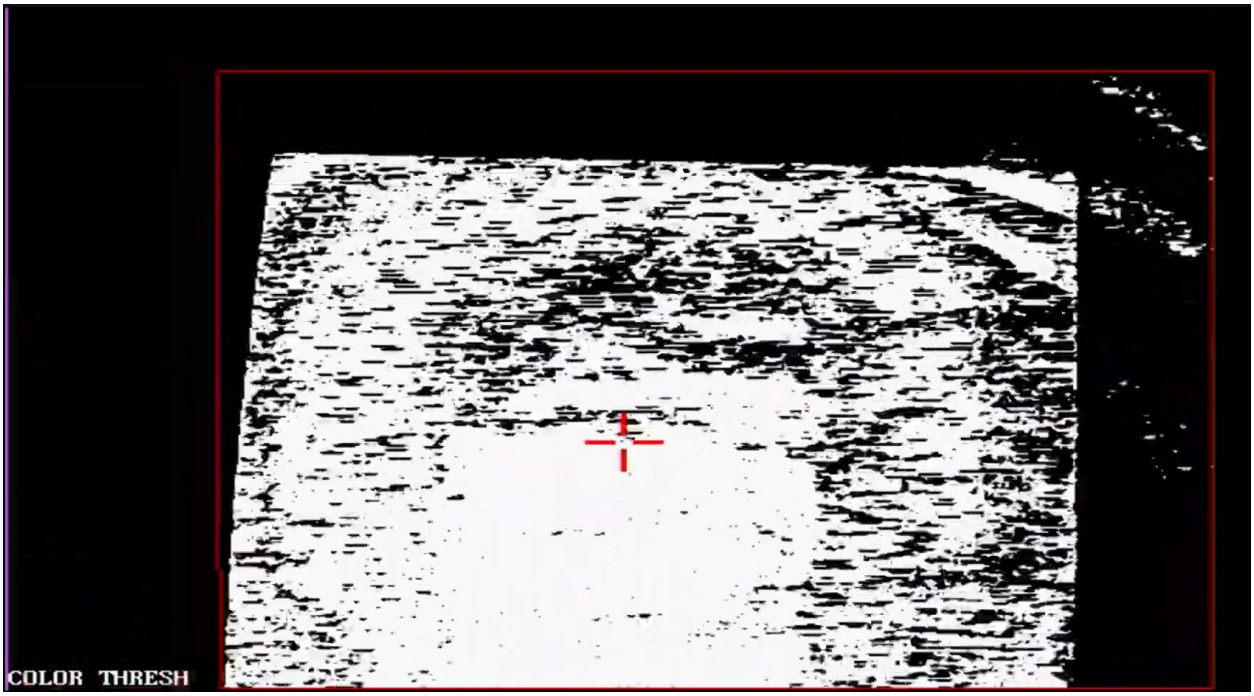

Figure 35: Color Threshold Video Output

Figure 36: Gesture Recognition Video Output for Fist

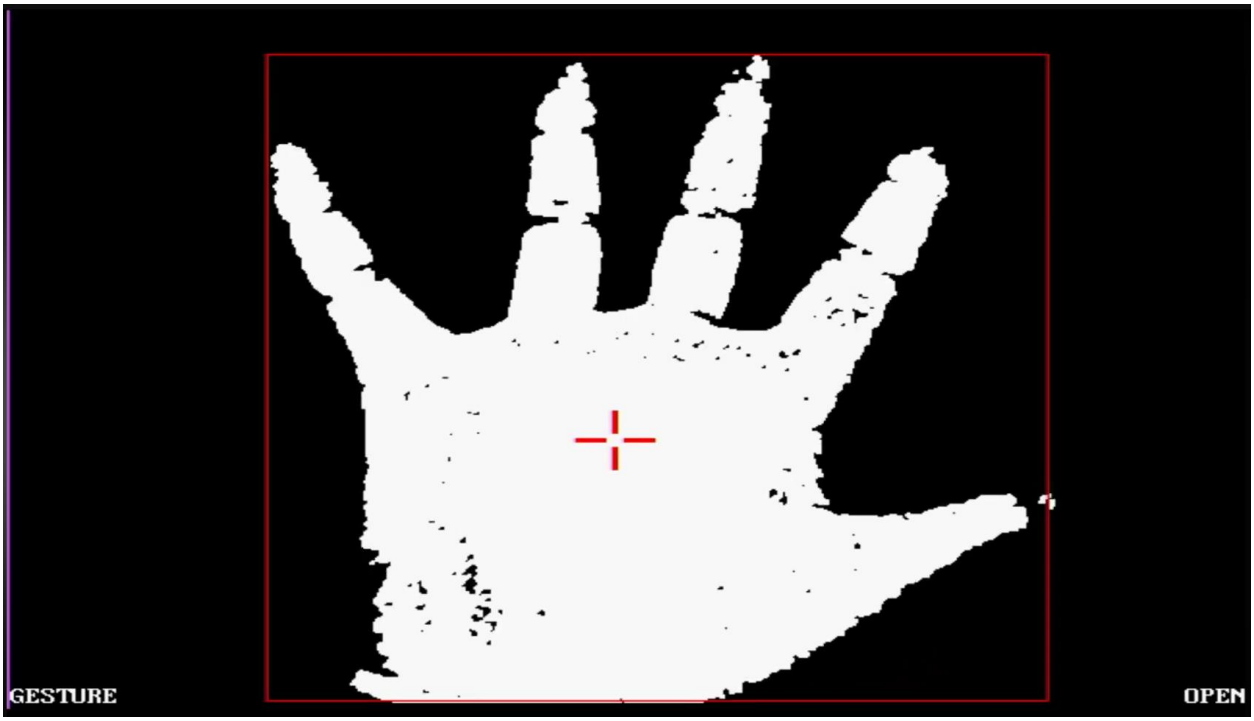

Figure 37:Gesture Recognition Video Output for Open Hand

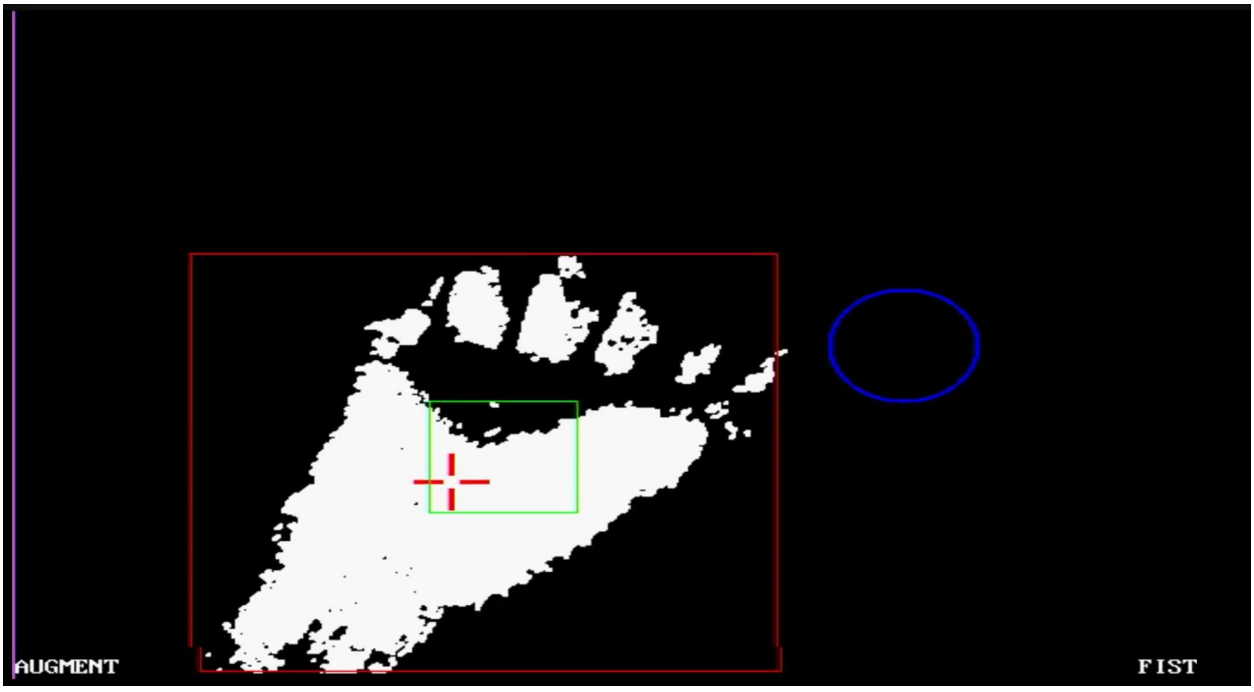

Figure 38: Augmented Reality Video Output

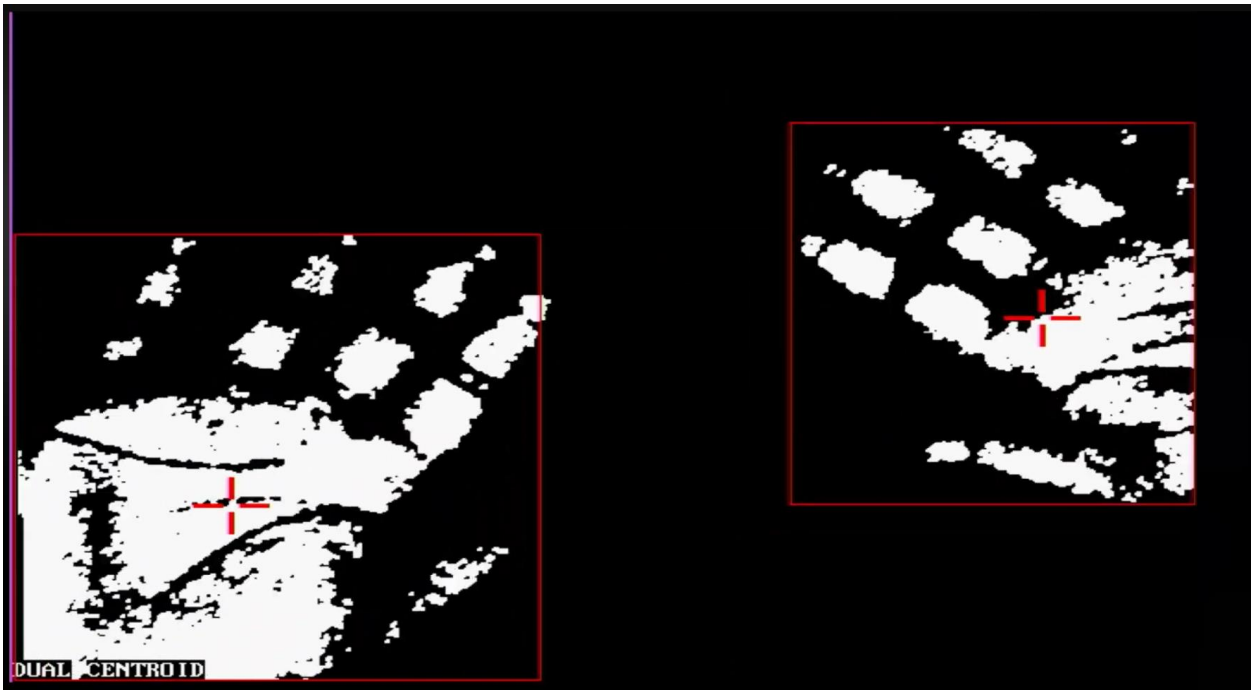

Figure 39:Dual Centroid Video Output

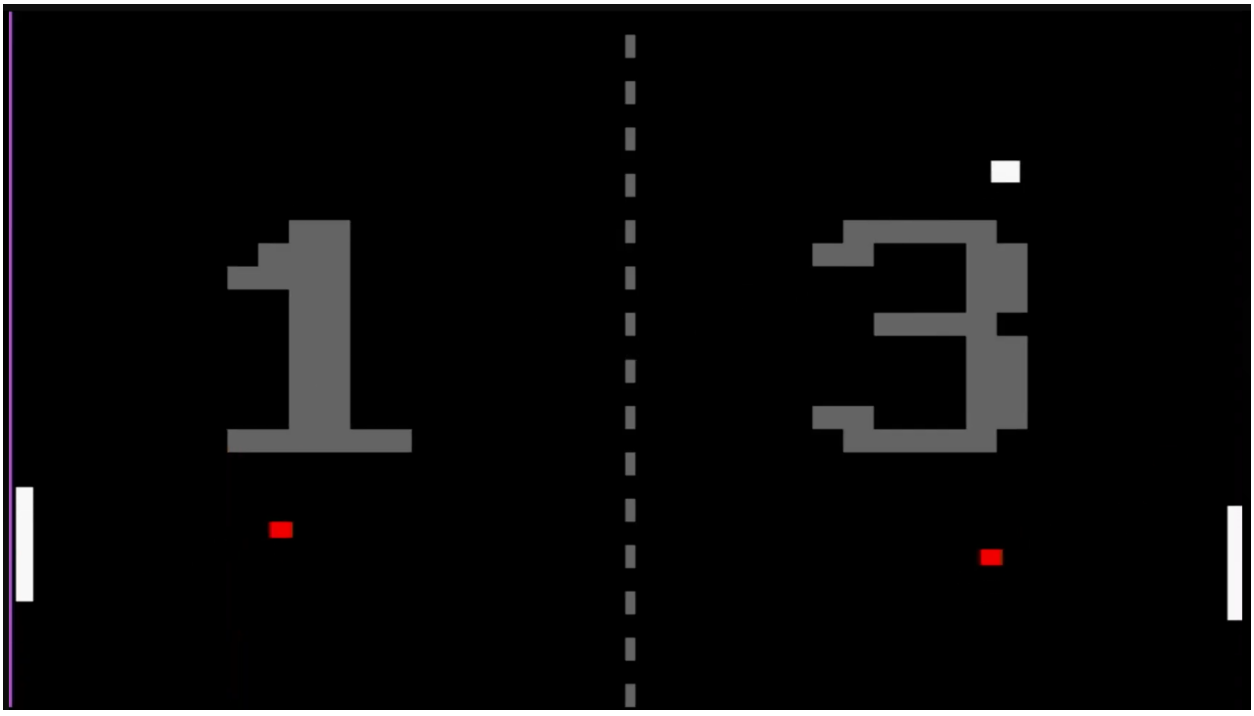

Figure 40:Motion Pong Video Output

Post-Lab Analysis

	Video Processor
LUT	28767
LUTRAM	13
DSP	15
Memory(BRAM)	75
Flip-Flop	18585
Latches	0
Frequency (MHz)	100.959
Static Power (W)	0.079
Dynamic Power (W)	0.521
Total Power (W)	0.6

Table 2: Design Statistics Summary

Conclusion

In conclusion, we took advantage of the FPGA's high capabilities at high-speed, low-latency computation by making a video processor. We first got the raw camera output. Then, we added various filters to the camera output, via convolution, skin thresholding, and morphological operations. After we got the filters, we created a perceptron-based gesture recognition system. We used this system to do some augmented reality and make our own version of pong, using camera input/output. We also implemented a motorized camera gimbal to control the camera on two axes.

Throughout this project, we encountered a few difficulties. Firstly, the camera, board, and VGA all operated in different clocking domains. In order to handle this, we had to be careful with synchronization of data so as to not introduce metastability into the system. Next, we had to figure out the XADC (Xilinx analog-digital converter) to read from the on-board potentiometer. The online documentation for interfacing with this sensor was confusing, but Cas from the past had some experience which helped us deal with this. Lastly, we had to deal with background noise which was introduced by default due to the cheap nature of our camera sensor. As a result, we had to employ various processing techniques to stabilize the image for smoother processing throughout the project.