CSEE4840 Final Report

Linus Lei, Daolin Li, Gurleen Kalra

Introduction **3**
Hardware Overview 3
Software Overview 4
Hardware **5**
Block Diagram 5
Interfacing OV7670 6
Processing (FPGA side): 8
Buffers 10
Frame Buffer 10
Meta Buffer 12
Text Buffer 12
FSM 13
VGA Display 14
Hardware Resource Budget 16
Hardware - Software Interface **17**
Control Registers 18
Software **20**
Software Pipeline 20
Auto-level 21
Otsu's Method 23
K-Means Clustering 24
Mask Generation 25
Verdict Logic 26
Results and Evaluation **28**
Hardware and Video Pipeline 28
Preprocessing Results 29
Mask Generation Results 31
Verdict Logic and System-Level Behavior 32
Verification 32
Summary 35
Reflection **36**
Who did what 36
Lessons Learned 36
Advice for future projects 37
File Listing **38**

Introduction

This project implements an embedded plant-monitoring system on the DE1-SoC platform using a combination of FPGA hardware and HPS software. An OV7670 camera is used to capture an image of a plant, and the resulting image is processed to estimate plant condition from its color and greenness characteristics. The system is designed to be lightweight, interpretable, and suitable for real-time embedded operation, with the FPGA handling timing-critical tasks and the HPS performing higher-level image analysis.

Hardware Overview

On the hardware side, the FPGA interfaces directly with the OV7670 camera, generates the required control clocks 25 MHz, receives the RGB565 pixel stream, and converts it into RGB888 format. In parallel, it computes an additional vegetation-related feature, Excess Green (ExG), for each pixel. The FPGA then packs the RGB and ExG values into a 32-bit word and stores the captured frame in on-chip memory. The camera can be configured via SCCB protocol. A SCCB controller is implemented to communicate with the camera. An VGA controller is also implemented to formulate the VGA display output. It reads from 3 buffers, creating a display interface with 3 panes and text console. Three M10K memory blocks are used as buffers. They are also initialized as Avalon MM Slave, allowing interfacing from the software. A group of control registers is used to formulate the handshake between software and hardware. A FSM is used to regulate the capture of camera inside the hardware.

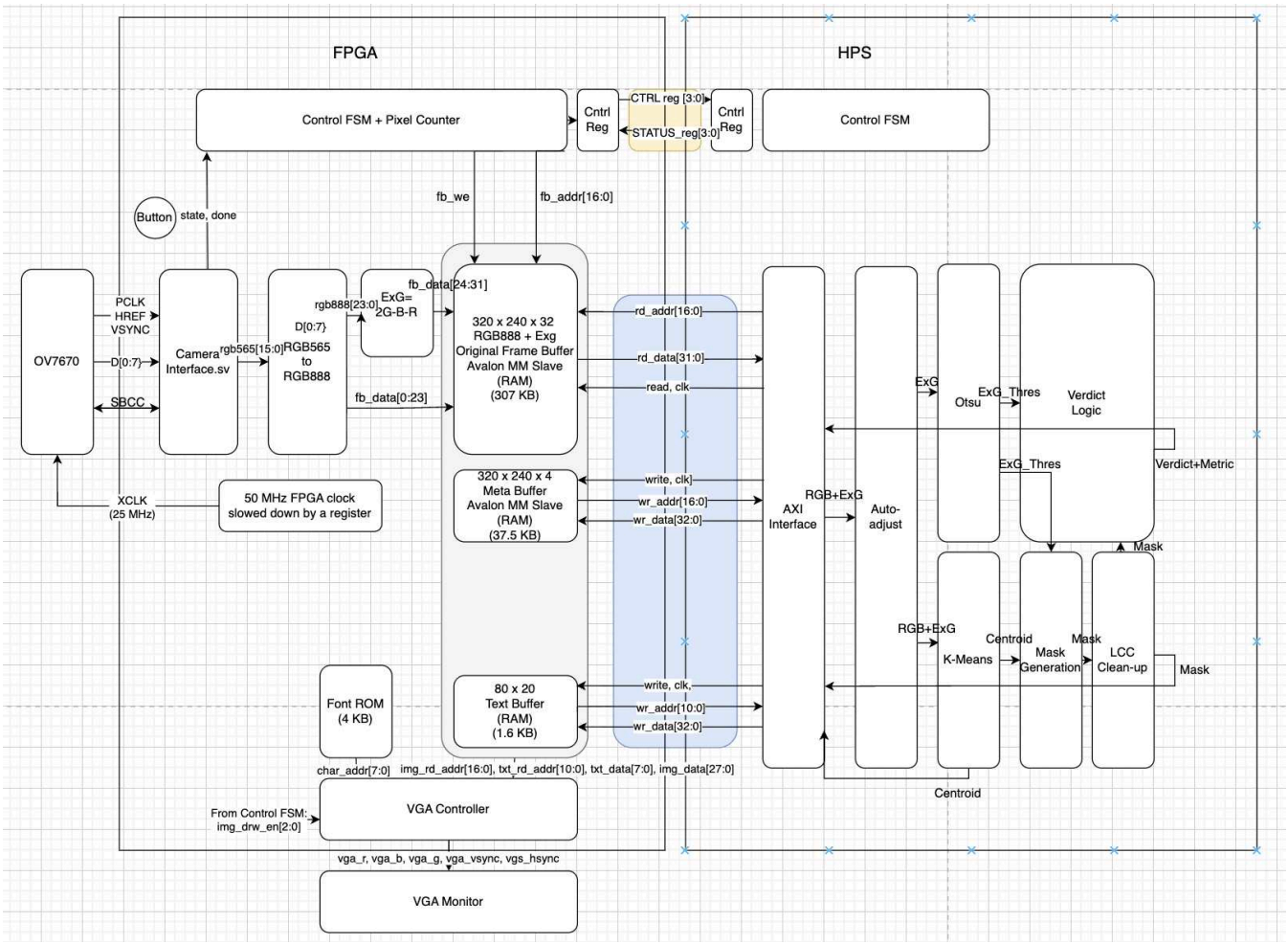
Software Overview

On the software side, the HPS reads the captured frame through the HPS-to-FPGA Fullweight bridge. Once it's been read, auto-level algorithm is applied to regulate the image color to true colors. Then K-Means clustering is applied to the image to separate different regions on the image. Otsu'method is also applied on ExG to find a suitable ExG threshold to help determine the roles of each cluster. The clustering result is used to separate plant regions from background and to distinguish healthier green regions from more stressed or yellow regions. From this result, the system generates a plant mask, computes simple health-related statistics, and produces a rule-based summary of plant condition. These processed outputs are then written back to FPGA-resident buffers so they can be displayed alongside the original image.

The final system provides both visual and textual feedback through a VGA monitor. The top portion of the screen displays the original image together with processed views, while the lower portion shows text-based results and status information. A user-interface is provided via the console. The user could use the console to perform different functions including live video viewing, white-balance calibration, and outputting image.

Hardware

Block Diagram

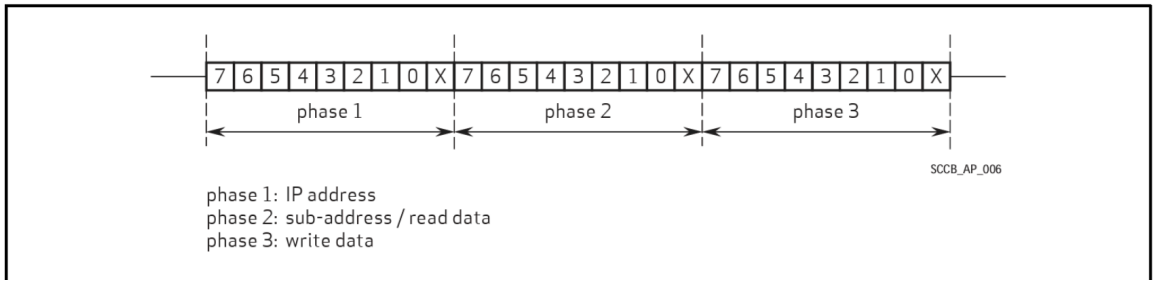


Interfacing OV7670

To capture the plant image, an OV7670 camera is used. It's a low-power compact image sensor which can be configured to output up to 640 x 480 VGA video stream at 30 frame per second. The OV7670 can be configured via the SCCB interface by Omnivision to adjust its output format, white balance and so on. The Pinout of OV7670 is included below, it utilize an 8-bit parallel output alongside HREF and VSYNC for locating the line and frame. It also needs a XCLK as input clock and regulates its 8-bit output via PCLK. 3.3V powering is needed.

The SCCB interface is compatible with I2C protocol. Primarily we would use a 3-phase write procedure to write to the configuration registers of the camera. The SCL would be provided and set to 100KHz by the FPGA per the datasheet. It would be generated using a counter to slow down the 50 MHz FPGA clock.

Figure 3-4 Transmission Phases



A bit-bang style SCCB controller is implemented to complete this 3-phase write process. A tri-state buffer is used to handle the "X" signal to Hi-Z state. For a normal register write, the SCCB controller performs a three-phase transaction: it sends a START condition,

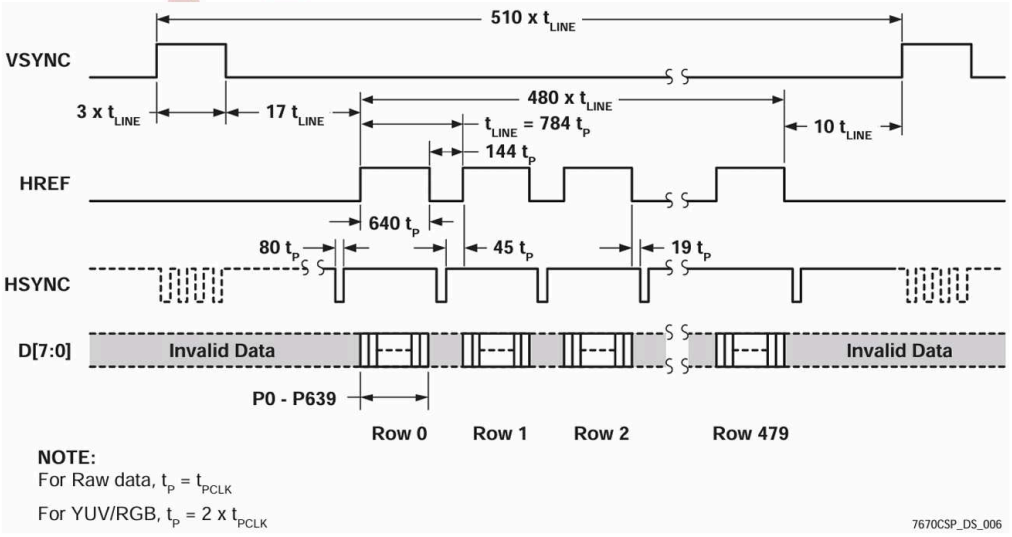
then the camera device address with the write bit, then the target register address, then the register value, and finally a STOP condition. After each byte, it releases SIOD so the camera can acknowledge by pulling the line low. The controller samples these ACK bits and reports an error if any ACK is missing. It does not pause inside the bit-level transaction waiting for ACK, but the higher-level setup FSM can retry a register command if the transaction reports an ACK error.

Physically, transmission starts when the controller's start input is asserted and the SCCB state machine begins stepping. On the bus, the actual START condition is SIOD falling while SIOC is high. The STOP condition is the reverse: SIOD rises while SIOC is high. Between those conditions, data bits are shifted out MSB-first, synchronized to the generated SCCB clock.

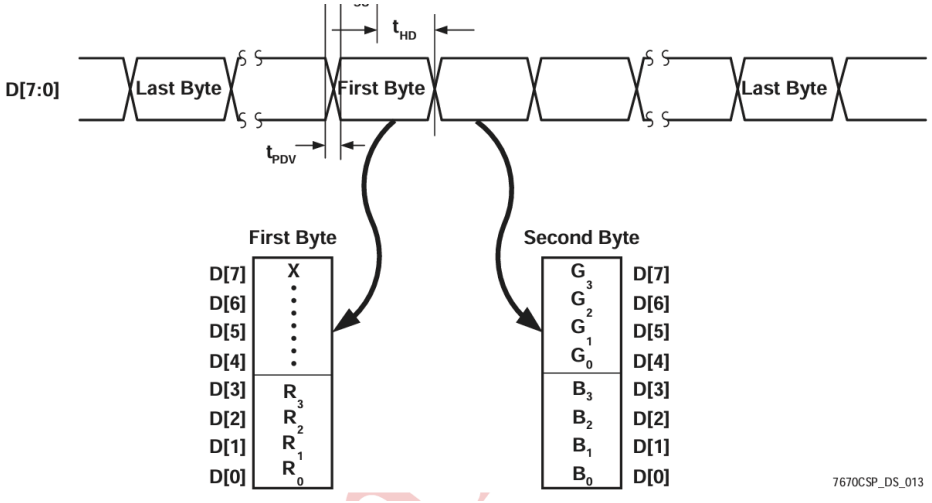
The XCLK would be provided from a register which halves the 50 MHz FPGA clock to 25 MHz, suitable for the camera per its datasheet.

By properly configuring OV7670, a QVGA output with resolution 320 x 240 will be outputted in the format of RGB565, with each pixel taking 2 clock, 16 bits to transmit.

Figure 6 VGA Frame Timing



The RGB receiver converts the OV7670's 8-bit streaming interface into 16-bit RGB565 pixels. It waits until a valid frame has started, then during each active video line (HREF=1) it samples the data bus on each rising edge of PCLK. Since RGB565 is transmitted as two bytes, the receiver toggles an internal odd/even flag: even cycles latch bits [15:8], odd cycles latch bits [7:0] and raise pixel_ready. When HREF goes low, the byte-pairing flag is reset so the next line starts aligned on the first byte of a pixel.



Our design will also utilize a shutter button that captures a single frame. This button will be debounced. Once triggered, it looks the next VSYNC to capture the next complete frame of the camera.

The SCCB controller and the RGB Receiver module is modified and referenced from the open source design of https://github.com/romovs/xula-lib-verilog.

Processing (FPGA side):

Once the RGB565 pixel stream is received from the camera, the FPGA performs two processing steps before writing the pixel into the main frame buffer: RGB565-to-RGB888 expansion and ExG feature calculation.

The first processing step separates these three color components from the 16-bit input pixel. Since the rest of the image-processing pipeline uses 8-bit color channels, each

component is expanded to RGB888 format. The 5-bit red and blue channels are expanded to 8 bits, and the 6-bit green channel is expanded to 8 bits. This is done using bit replication, which preserves the full 0-to-255 output range more accurately than simply appending zeros. After this step, each pixel is represented as three 8-bit color channels: `R`, `G`, and `B`.

The second processing step computes the ExG, or excess green, value for the pixel. ExG is used as a vegetation-oriented feature that measures how strongly green a pixel is relative to its red and blue components. This is more informative than using the green channel alone, because a bright white or gray pixel may have a high green value but is not actually vegetation. By subtracting the red and blue components, ExG emphasizes pixels where green is dominant.The formula used is: ExG = 2G - R - B

This calculation is simple enough to implement directly in FPGA combinational logic and is fast enough to compute for every pixel as it streams from the camera. Because the raw result can be negative or greater than 255, the value is clamped to the 8-bit range:
if ExG < 0, ExG = 0, if ExG > 255, ExG = 255

After RGB expansion and ExG computation, the FPGA packs the result into a single 32-bit word:
[31:24] ExG, [23:16] R, [15:8] G, [7:0] B

This 32-bit `{ExG, R, G, B}` word is then written into the Original Frame Buffer at the address corresponding to the pixel's `(x, y)` position. Storing ExG alongside the RGB888 data allows the HPS software to read a single word per pixel and immediately access both the color information and the precomputed vegetation feature for downstream processing such as thresholding, k-means clustering, mask generation, and health classification..

Buffers

The system uses several FPGA-resident memory blocks to separate camera acquisition, HPS software analysis, and VGA display. All of these buffers are implemented as on-chip M10K block RAMs, using a shared dual-port RAM primitive. This allows the design to keep high-bandwidth image and display data inside the FPGA fabric while exposing selected buffers to the HPS through Avalon-MM interfaces.

Frame Buffer

The main image memory is the Original Frame Buffer, which stores one full camera frame at a resolution of `320 x 240`. Each pixel is stored as a 32-bit word in the format:

[31:24] ExG

[23:16] R

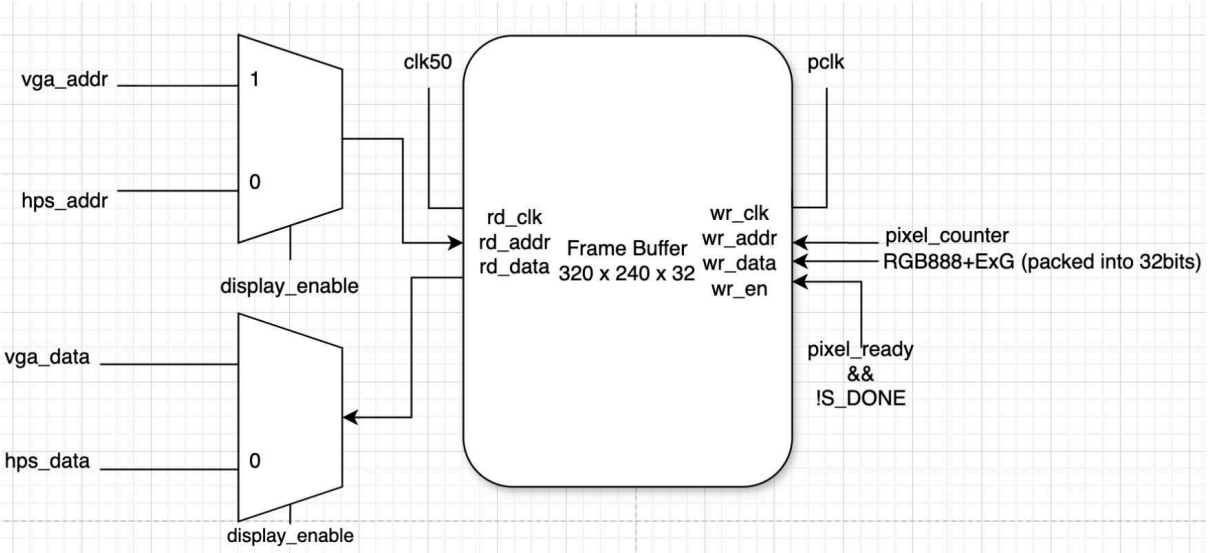
[15:8] G

[7:0]   B

The OV7670 camera outputs pixels in RGB565 format. In the FPGA camera pipeline, each 16-bit RGB565 pixel is expanded to RGB888 by bit replication: the 5-bit red and blue channels are expanded to 8 bits, and the 6-bit green channel is expanded to 8 bits. The FPGA also computes the ExG vegetation feature, `, before writing the final `{ExG, R, G, B}` word into memory. Since the frame resolution is `320 x 240`, the Original Frame Buffer contains `76,800` 32-bit words. This corresponds to `307,200` bytes, or approximately `300 KiB` of M10K block RAM. Pixels are stored in row-major order, so the address of pixel `(x, y)` is:address = y * 320 + x

The Original Frame Buffer is written only by the FPGA camera pipeline. The write port runs in the camera `PCLK` domain, and the write address is generated from the current pixel coordinates tracked by the FPGA logic. The write enable is asserted only when a

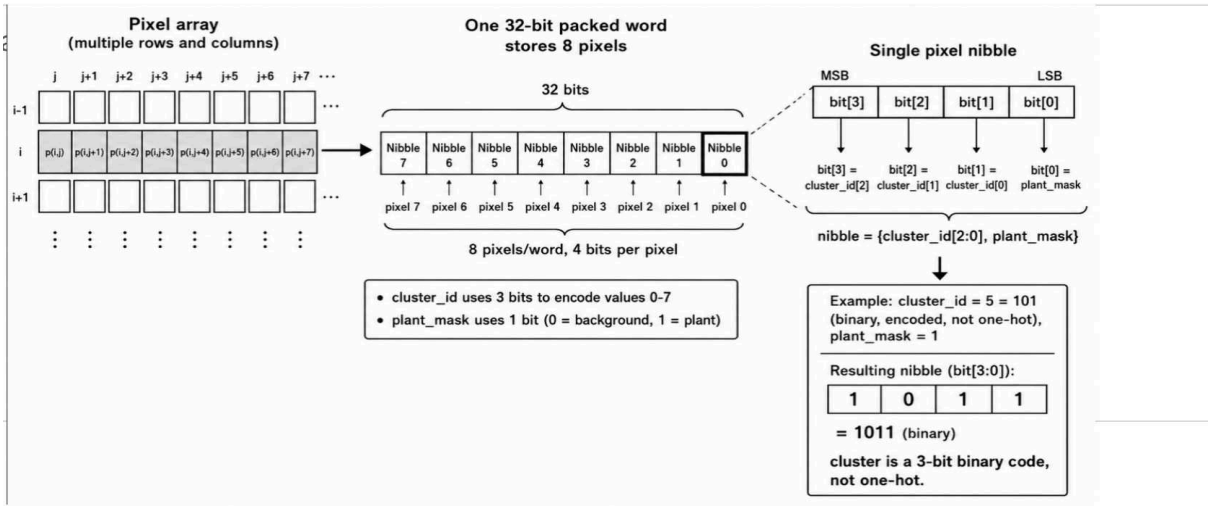
complete RGB565 pixel has been assembled, the pixel coordinate is inside the `320 x 240` image bounds, and the capture FSM is either actively capturing a frame or allowing live preview updates. This write-gating mechanism lets the same buffer support both live video and frozen snapshots: when writes are enabled continuously, VGA pane 0 appears as live video; when the FSM finishes a capture and disables writes, the stored image remains static for HPS processing.



The read side of the Original Frame Buffer is shared between the VGA pipeline and the HPS. The buffer is exposed to the HPS as a read-only Avalon-MM slave on the full HPS-to-FPGA bridge. A local read path is also connected to the VGA display logic. A `display_enable` control bit selects ownership of the read port: when `display_enable = 1`, the VGA pipeline supplies the read address and displays the original image; when `display_enable = 0`, the HPS supplies Avalon addresses and reads out the captured frame for software processing. This avoids copying the image into a separate FPGA display buffer, while still allowing the HPS to access the captured image.

Meta Buffer

In addition to the Original Frame Buffer, the design includes a Meta Buffer for processed image results. This buffer is also implemented in M10K RAM and is exposed as a read/write Avalon-MM slave. It stores per-pixel metadata for the full `320 x 240` image, but packs eight pixels into each 32-bit word. Each pixel uses a 4-bit nibble:



The `cluster_id` field is 3 bits wide, supporting up to eight k-means clusters. The `plant_mask` bit stores the cleaned plant segmentation result after largest-connected-component filtering. Because eight pixels are packed per word, the Cluster and Mask Buffer contains `76,800 / 8 = 9,600` 32-bit words, or `38,400` bytes, approximately `37.5 KiB` of M10K memory. This buffer is written by the HPS after software processing is complete and read continuously by the VGA display pipeline. The VGA controller uses the cluster ID to color the cluster pane and the mask bit to generate the binary plant-mask pane.

Text Buffer

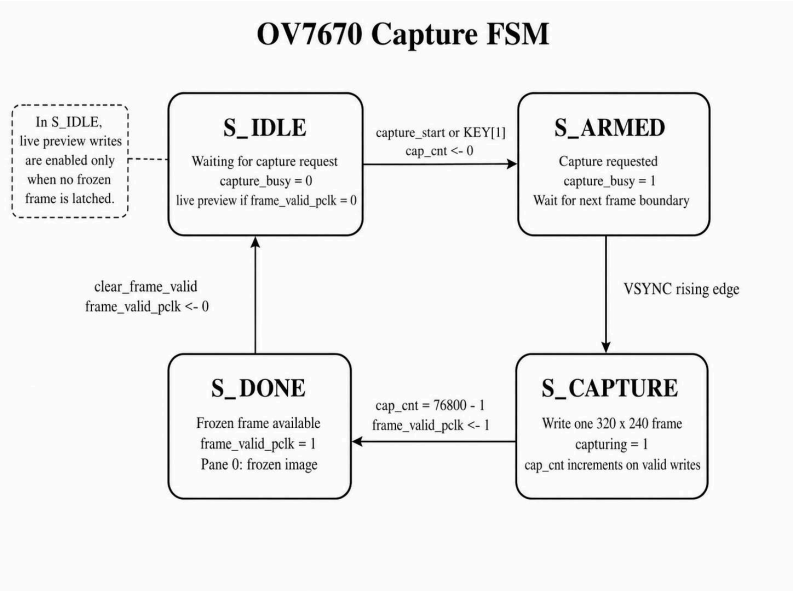
The design also includes a Text Buffer for the on-screen status display. This buffer stores an `80 x 20` character grid, with four 8-bit ASCII characters packed into each 32-bit word. It therefore contains `80 * 20 / 4 = 400` 32-bit words, or `1,600` bytes of M10K memory. The HPS writes status strings, timing information, classification results, and debug

information into this buffer through Avalon-MM. The VGA pipeline reads the character codes and passes them to a hardware font ROM, which converts each ASCII character into pixel rows for display. The text rendering itself is performed entirely in FPGA logic, so the HPS only needs to update character codes rather than draw individual pixels.

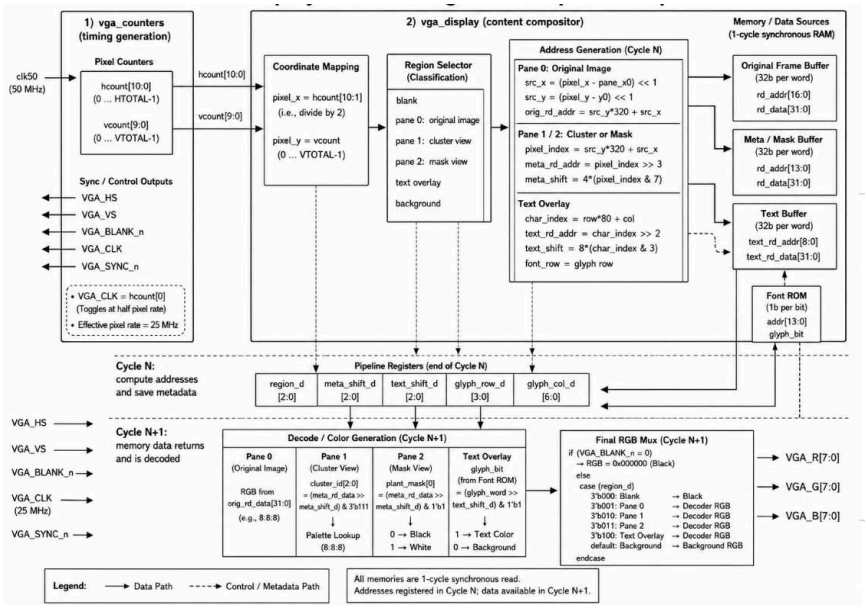
FSM

The main hardware control logic is the OV7670 capture FSM, implemented in ov7670_top.sv. This FSM runs in the camera pixel-clock domain because it must respond directly to camera timing signals such as VSYNC, HREF, and pixel_ready. Its purpose is to control when incoming camera pixels are written into the original frame buffer and when that buffer should be treated as a valid frozen frame for HPS processing. The FSM has four states: **S_IDLE, S_ARMED, S_CAPTURE, and S_DONE**.

In S_IDLE, no capture is in progress. If no valid frozen frame exists, the camera stream is allowed to continuously update the original frame buffer, enabling live preview on the VGA display. When the HPS writes *capture_start*, the FSM enters S_ARMED and waits for the next VSYNC rising edge so that capture begins cleanly at a new frame boundary. It then enters S_CAPTURE, where valid camera pixels are written into the frame buffer until exactly one 320 × 240 frame has been stored. After all 76,800 pixels are captured, the FSM enters S_DONE, asserts frame_valid, and disables further camera writes so the frame buffer remains frozen for HPS readout and processing. The FSM returns to S_IDLE only after the HPS issues clear_frame_valid, allowing live preview to resume.



VGA Display



The VGA display path is divided into two major components: timing generation and content composition. The vga_counters module generates the raster scan timing for the VGA output, including the horizontal and vertical counters, synchronization signals, blanking signal, and VGA clock. It operates from the 50 MHz system clock, while the effective VGA pixel rate is 25 MHz because VGA_CLK is derived from the lower bit of the horizontal

counter. The counter outputs, hcount and vcount, represent the current scan position in the VGA timing frame and are passed to the display compositor.

The vga_display module acts as the content compositor. Its role is to decide what color should be displayed at each active screen pixel. First, it converts the timing counters into visible pixel coordinates using pixel_x = hcount[10:1] and pixel_y = vcount. The compositor then classifies the current coordinate into one of several display regions: the original image pane, the cluster visualization pane, the binary mask pane, the text overlay region, background, or blanking. This region selection determines which memory should be read and how the returned data should be interpreted.

For the three image panes, the display area is 160 × 120, while the source frame stored in memory is 320 × 240. Therefore, the compositor maps each screen pixel to a source image coordinate using 2× nearest-neighbor scaling. For the original image pane, it computes an address into the original frame buffer and reads the packed {ExG, R, G, B} word. For the cluster and mask panes, it computes the corresponding pixel index in the 320 × 240 source image, then uses pixel_index >> 3 to address the packed meta buffer. Since each meta-buffer word stores eight 4-bit nibbles, the lower bits of the pixel index are saved as a shift amount to select the correct nibble after the memory data returns.

The text overlay region is rendered using the text buffer and font ROM. The screen is divided into an 80 × 20 character grid, where each character cell is 8 × 16 pixels. For each pixel in the text band, the compositor computes the character row and column, reads the corresponding ASCII value from the text buffer, and uses the font ROM to determine whether the current pixel belongs to the character foreground or background. This allows the HPS to write verdicts, statistics, and debugging information as ASCII text while the FPGA renders the characters directly on the VGA display.

Because the FPGA memories are synchronous, the VGA compositor uses a one-cycle pipeline. During cycle N, it computes the memory addresses and stores metadata such as the selected region, meta-buffer shift amount, text-buffer shift amount, glyph row, and glyph column. During cycle N+1, the memory data is available and the compositor decodes it according to the delayed region information. The final RGB mux then selects the output color: original RGB for pane 0, palette color for the cluster pane, black or white for the mask pane, text color for the text overlay, or background color otherwise. If the blanking signal is inactive, the output is forced to black.

Hardware Resource Budget

FPGA buffer: 320 x 240 x 4 (bytes per pixel) = 307.2 KB (RGB888: 24bits, ExG: 8 bits)

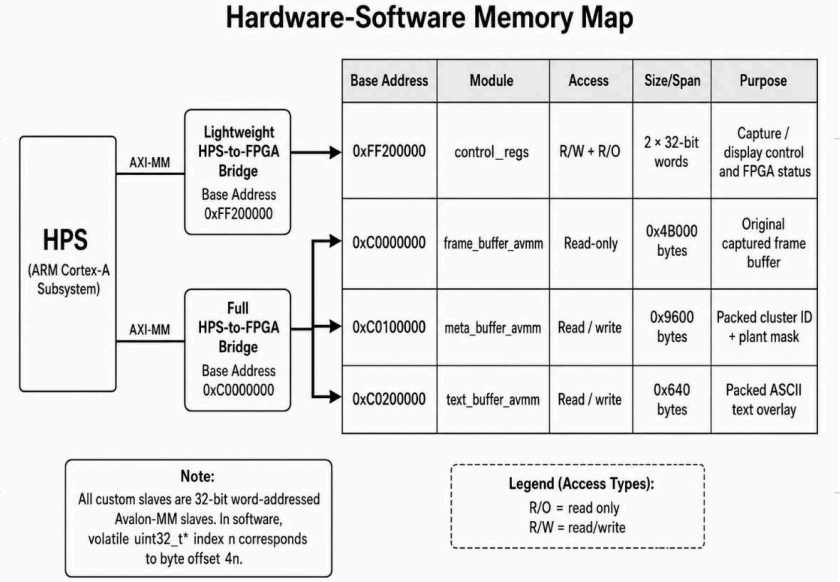
Cluster +Masked Buffer: 320 x 240 x 4bit = 38.4 KB

Font Memory: 8 x 16 x 256 (characters) = 4 KB

Text Input Memory = 20 lines x 80 columns x 8 bit per character = 1.6 KB

Summary: 307.2 + 38.4 + 4 + 1.6 = 351.2 KB

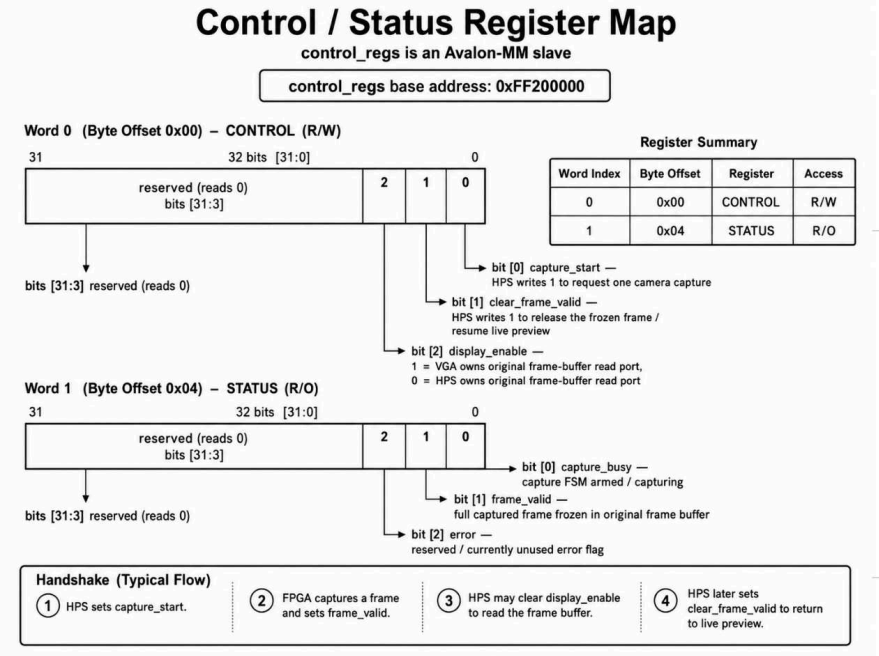
Hardware - Software Interface



The HPS communicates with the FPGA through memory-mapped Avalon-MM slave peripherals exposed over the Cyclone V HPS-to-FPGA bridges. In this design, the interface is divided by access pattern: the lightweight HPS-to-FPGA bridge is used for the small control_regs block, while the full HPS-to-FPGA bridge is used for the larger data buffers. This allows low-bandwidth control and status signals to remain separate from bulk image and display data transfers.

As shown in the memory map, the HPS accesses four main FPGA-side regions. The control_regs block at 0xFF200000 provides capture and display control as well as FPGA status feedback. The original frame buffer at 0xC0000000 stores the captured 320 × 240 frame and is read-only from the HPS side in the current hardware. The meta buffer at 0xC0100000 stores the packed cluster ID and plant mask information generated by the HPS processing pipeline. Finally, the text buffer at 0xC0200000 stores ASCII characters used by the VGA text overlay. All custom slaves are accessed as 32-bit word-addressed regions in software, so a volatile uint32_t * pointer index corresponds to a four-byte offset in the mapped address space.

Control Registers



The control_regs block provides the main synchronization interface between the HPS software and the FPGA capture logic. It is mapped as a small Avalon-MM slave at base address 0xFF200000 and contains two 32-bit registers: CONTROL at byte offset 0x00 and STATUS at byte offset 0x04. The CONTROL register is written by the HPS to issue commands to the FPGA, while the STATUS register is read by the HPS to observe the current capture state. Only the lowest three bits are currently used in each register; the remaining bits are reserved and read as zero.

The CONTROL register contains three command bits. Bit 0, capture_start, requests that the FPGA capture one camera frame. Bit 1, clear_frame_valid, releases the frozen frame and allows the system to return to live preview. Bit 2, display_enable, controls ownership of the original frame-buffer read port: when display_enable = 1, the VGA compositor reads the frame buffer for display; when display_enable = 0, the HPS takes ownership and reads the captured frame for software processing. The STATUS register reports the capture FSM state. Bit 0, capture_busy, indicates that the FPGA is armed or actively capturing a frame. Bit 1,

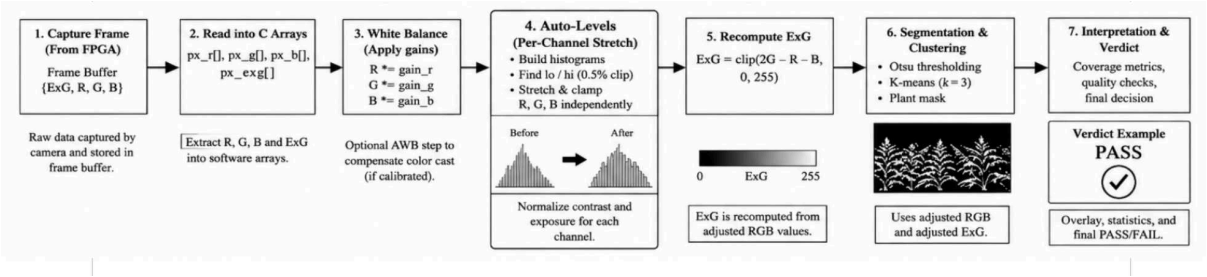
frame_valid, indicates that a complete frame has been captured and is frozen in the original frame buffer. Bit 2 is reserved as an unused error flag.

The typical handshake begins when the HPS writes capture_start = 1. The FPGA capture FSM then waits for the next frame boundary, captures one full 320 × 240 frame, and asserts frame_valid once the frame is complete. The HPS polls the STATUS register until frame_valid becomes high, then clears display_enable so it can safely read the original frame buffer through Avalon-MM. After software processing is complete, the HPS writes the processed metadata and text results to the corresponding buffers and re-enables display. When the system is ready to resume live preview, the HPS writes clear_frame_valid = 1, allowing the FPGA to release the frozen frame and return to continuous camera preview.

Software

The HPS-side application, hps_processing, runs in Linux user space and accesses the FPGA peripherals through /dev/mem by memory-mapping the physical HPS-to-FPGA bridge address ranges. It maps the lightweight bridge at 0xFF200000 for the control/status registers and the full HPS-to-FPGA bridge at 0xC0000000 for the frame, meta, and text buffers. The mapped regions are cast to volatile uint32_t * pointers so the software can issue direct memory-mapped I/O reads and writes to the Avalon-MM slaves implemented in the FPGA. This approach bypasses a kernel driver, so it requires root privileges and relies on the application to correctly manage register offsets, buffer ownership, and capture/display handshakes.

Software Pipeline



The HPS software pipeline starts after the FPGA capture logic has stored a valid frame in the original frame buffer. The software first reads the captured 320 × 240 frame from the FPGA into local C arrays, separating each pixel into px_r[], px_g[], px_b[], and px_exg[]. This converts the memory-mapped hardware frame into a software-friendly representation that can be processed sequentially on the ARM processor.

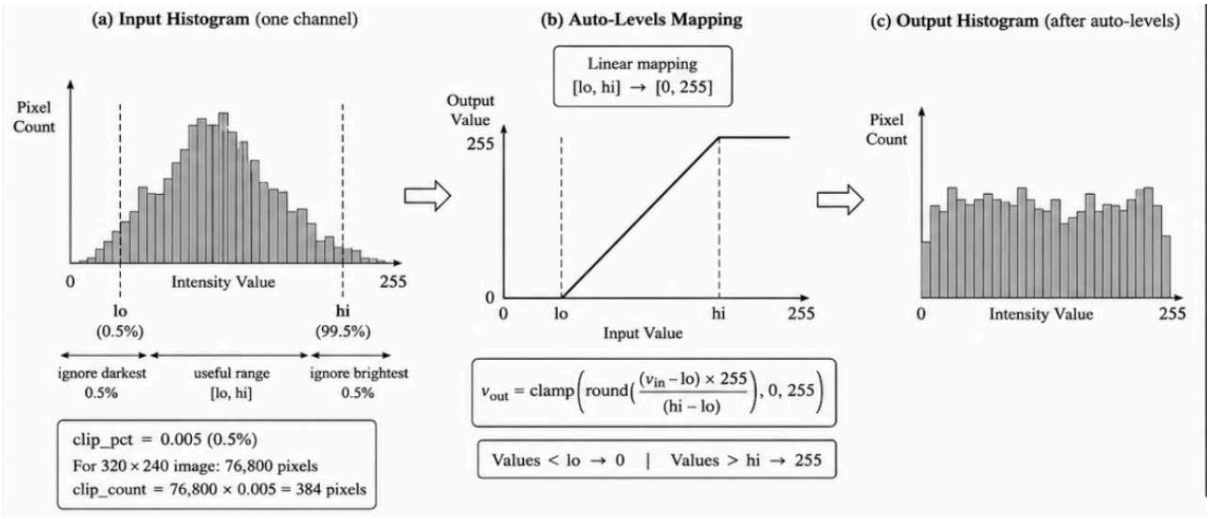
The first processing stage is optional white balance. If a white-balance reference has been calibrated, the software applies gain factors to the red, green, and blue channels to compensate for camera color cast. After this, the auto-levels stage performs a per-channel

contrast stretch: for each RGB channel, it builds a histogram, clips the darkest and brightest 0.5% of pixels, and maps the remaining intensity range to [0, 255]. This normalizes the image before segmentation so that lighting and exposure differences do not dominate the result.

Because both white balance and auto-levels modify the RGB values, the software then recomputes ExG using the adjusted channels.

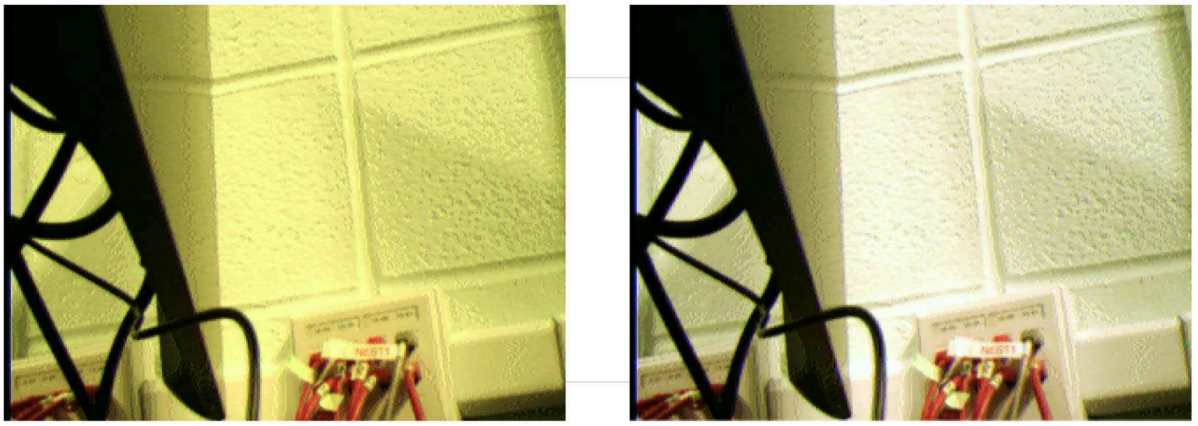
The updated RGB and ExG features are then used for segmentation and clustering. Otsu's method computes an adaptive ExG threshold for the current frame, while k-means groups pixels in the combined RGB/ExG feature space. The cluster interpretation logic labels clusters as green plant, stressed plant, or background, and the plant mask is generated from the union of green and stressed clusters. Finally, the software extracts health-related features, such as plant area and stress ratio, and produces the final verdict. The resulting cluster IDs, mask bits, and text summary are written back to the FPGA meta and text buffers for VGA display.

Auto-level



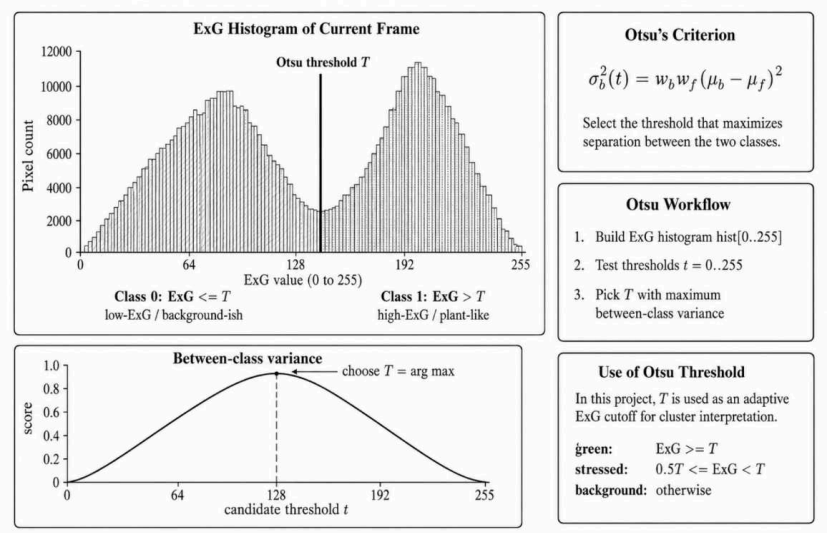
Auto-levels is used as a preprocessing step to reduce lighting and exposure variation before segmentation. After the captured frame is read into software arrays, the algorithm

processes the red, green, and blue channels independently. For each channel, it builds a 256-bin intensity histogram, discards the darkest and brightest 0.5% of pixels, and uses the remaining low and high cutoff values as the useful intensity range. This range is then linearly stretched to cover the full [0, 255] output range, while values below the low cutoff are clamped to 0 and values above the high cutoff are clamped to 255.



The two example images illustrate the effect of this operation. The original image has a strong yellow-green color cast and uneven contrast, which would make fixed color or ExG thresholds less reliable. After auto-levels, the image uses a wider intensity range and the color channels are better normalized, making the scene less dominated by the camera's lighting bias. Since auto-levels changes the RGB values, the software recomputes ExG afterward using the adjusted channels, ExG = clip(2G - R - B, 0, 255). This ensures that Otsu thresholding, k-means clustering, and later mask generation operate on the corrected image features rather than the raw camera output.

Otsu's Method



In this project, Otsu's method is used to choose an adaptive threshold on the ExG image. After white balance and auto-levels, the software recomputes ExG for every pixel . Instead of using a fixed threshold for all scenes, the HPS builds a 256-bin histogram of the current frame's ExG values and searches over all possible thresholds from 0 to 255. For each candidate threshold, the pixels are divided into two groups: a low-ExG group, which is more likely to be background, and a high-ExG group, which is more likely to contain plant pixels.

Otsu's method selects the threshold T that maximizes the separation between these two groups. More formally, for each candidate threshold t, the algorithm computes the size and mean ExG value of the low-ExG and high-ExG classes, then evaluates the between-class variance. The threshold with the highest between-class variance is chosen as the best split point. This makes the threshold frame-specific, which is useful because the OV7670 image can change significantly with lighting, exposure, and background conditions.

In our pipeline, the Otsu threshold is not used as the final mask by itself. Instead, it provides an adaptive ExG reference for interpreting the k-means clusters. Clusters with centroid ExG above T and sufficient green dominance are labeled as green plant, while

clusters with moderate ExG, such as between 0.5T and T, can be labeled as stressed plant if they also pass brightness and green-dominance checks. The final plant mask is then formed from the union of green and stressed clusters. This makes Otsu a supporting threshold-selection step rather than the entire classifier.

K-Means Clustering

K-means clustering groups pixels with similar color and greenness properties. Each pixel is represented as a four-dimensional feature vector:

$$x_i = [R_i, G_i, B_i, ExG_i]$$

where R_i, G_i, and B_i are the RGB888 components and ExG_i is the Excess Green value computed in hardware.

The algorithm maintains K centroids, where centroid j is:

$$\mu_j = [\mu_{j,R}, \mu_{j,G}, \mu_{j,B}, \mu_{j,ExG}]$$

For each pixel, the HPS computes the squared Euclidean distance to every centroid:

$$d_{ij} = (R_i - \mu_{j,R})^2 + (G_i - \mu_{j,G})^2 + (B_i - \mu_{j,B})^2 + (ExG_i - mu_{j,ExG})^2$$

The pixel is assigned to the cluster with minimum distance:

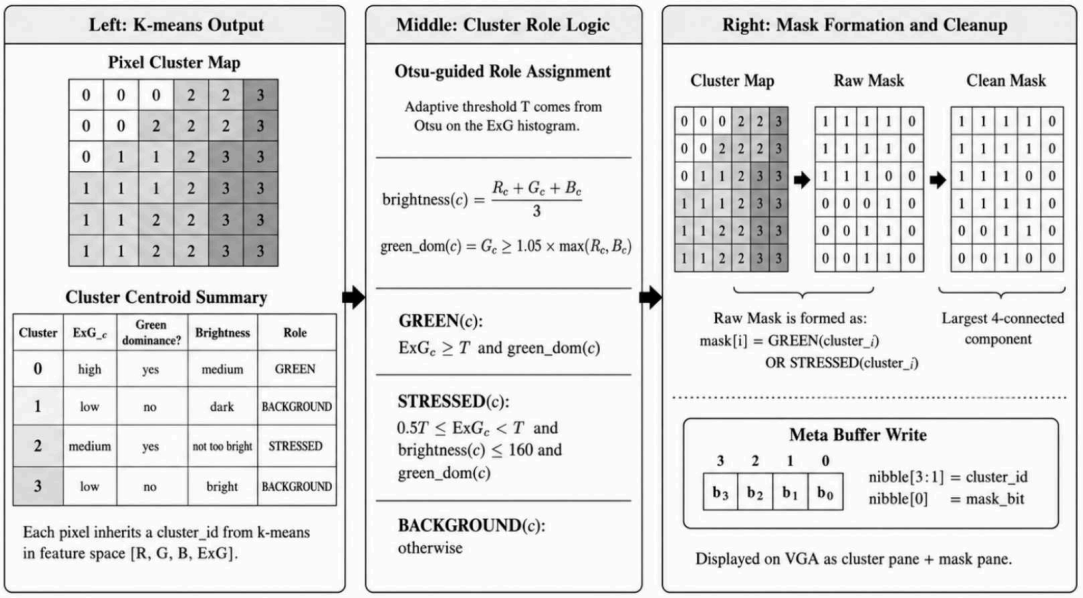
$$c_i = argmin_j d_{ij}$$

After all pixels have been assigned, each centroid is updated by averaging the feature vectors of all pixels assigned to that cluster. If S_j is the set of pixels currently belonging to cluster j, then the new centroid is:

$$\mu_j = (1/|S_j|) \sum_{iinS_j} x_i$$

These assignment and update steps are repeated until convergence or until a fixed maximum number of iterations is reached.

Mask Generation



Cluster	ExG_c	Green dominance?	Brightness	Role
0	high	yes	medium	GREEN
1	low	no	dark	BACKGROUND
2	medium	yes	not too bright	STRESSED
3	low	no	bright	BACKGROUND

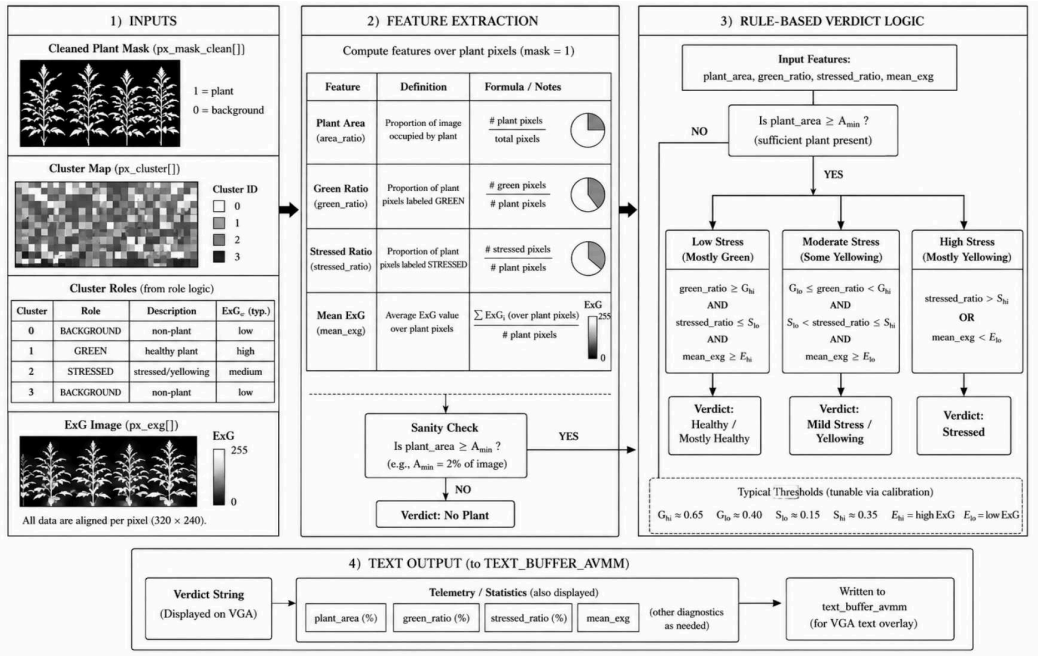
For mask generation, the key design choice is that the plant mask is produced from cluster roles, not from a direct per-pixel ExG threshold. After preprocessing, k-means assigns every pixel to a cluster in the four-dimensional feature space [R, G, B, ExG]. Each cluster is summarized by its centroid, which represents the average color and ExG value of the pixels assigned to that group. The software then interprets each centroid using the adaptive Otsu threshold, green-dominance check, and brightness constraint.

A cluster is labeled as green plant if its centroid ExG is above the Otsu threshold and its green channel dominates red and blue. A cluster is labeled as stressed plant if its ExG is moderately high, between 0.5T and T, while still passing the green-dominance test and not being too bright. This brightness gate helps prevent bright neutral backgrounds from being classified as stressed vegetation. Clusters that do not satisfy either condition are treated as background.

Once the roles are assigned, each pixel inherits the role of its k-means cluster. The raw plant mask is formed by selecting all pixels whose cluster is labeled either green or stressed:mask[i] = GREEN(cluster_i) OR STRESSED(cluster_i)

The raw mask may still contain small disconnected false positives, so the software performs a largest connected component cleanup using 4-connected neighborhoods. This keeps the dominant plant blob and removes isolated noise. Finally, the cleaned mask bit is packed together with the pixel's cluster ID into the meta buffer, where nibble[3:1] stores the encoded cluster_id and nibble[0] stores the cleaned mask bit. This packed metadata is then used by the VGA display to render both the cluster visualization and the binary mask pane.

Verdict Logic



The verdict logic is the final software stage that converts the cleaned segmentation result into a human-readable plant health decision. Instead of classifying directly from the raw image, the software first uses the cleaned plant mask to restrict analysis to pixels that are likely part of the plant. This avoids allowing background regions, shadows, or bright

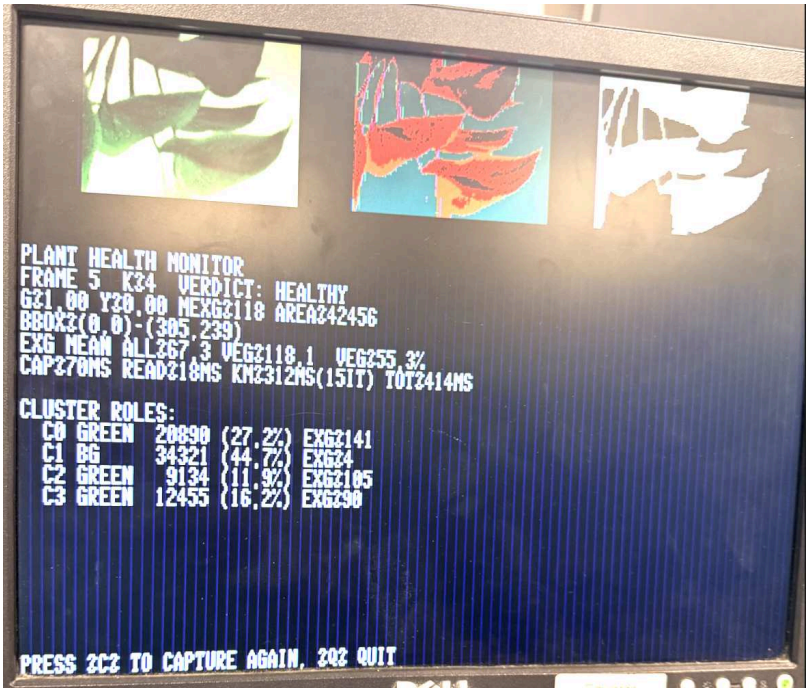
non-plant objects to affect the final decision. The inputs to this stage are the cleaned mask, the k-means cluster labels, the interpreted cluster roles, and the recomputed ExG image.

From these inputs, the HPS extracts several plant-level features. The plant area measures how much of the image is occupied by the cleaned plant mask. The green ratio measures the fraction of detected plant pixels assigned to healthy green clusters, while the stressed ratio measures the fraction assigned to stressed or yellowing clusters. The mean ExG value is also computed over the plant region to summarize the overall greenness of the detected plant. These features provide a compact description of the plant's visual condition after segmentation.

The rule-based classifier then uses these features to assign the final verdict. First, it performs a sanity check on plant area; if the detected plant region is too small, the result is labeled as "No Plant" because the system does not have enough reliable plant pixels to analyze. Otherwise, the software evaluates the balance between green and stressed regions. A high green ratio with low stressed ratio indicates a healthy or mostly healthy plant, while increasing stressed coverage or lower mean ExG leads to labels such as mild stress, yellowing, or stressed. The final verdict, along with the key statistics, is written into the text buffer and displayed on the VGA overlay.

Results and Evaluation

Hardware and Video Pipeline

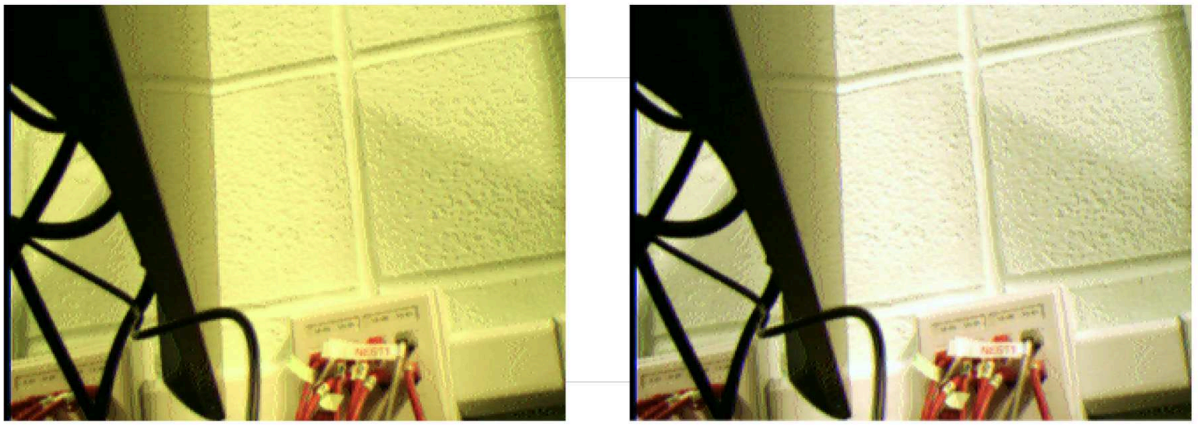


The completed prototype successfully demonstrates the full hardware-software image-processing loop on the DE1-SoC platform. The OV7670 camera was configured through the SCCB interface and produced a stable QVGA RGB565 video stream. On the FPGA side, the camera pixel receiver correctly captured the `PCLK`, `HREF`, `VSYNC`, and `D[7:0]` signals, reconstructed RGB565 pixels, expanded them to RGB888, computed the ExG value, and wrote the packed `{ExG, R, G, B}` words into the original frame buffer. The successful live preview confirms that the camera configuration, pixel capture logic, frame-buffer write path, and VGA read path are all functioning correctly.

The VGA output also validates the memory-mapped display architecture. The top row of the display shows the original camera image, the k-means cluster visualization, and the binary mask output. The lower text region displays telemetry generated by the HPS, including frame number, verdict, ExG statistics, area, cluster roles, and timing measurements.

This confirms that the HPS can read captured image data from the FPGA, process it in software, write the processed metadata and text results back to FPGA buffers, and display the updated results through the VGA compositor.

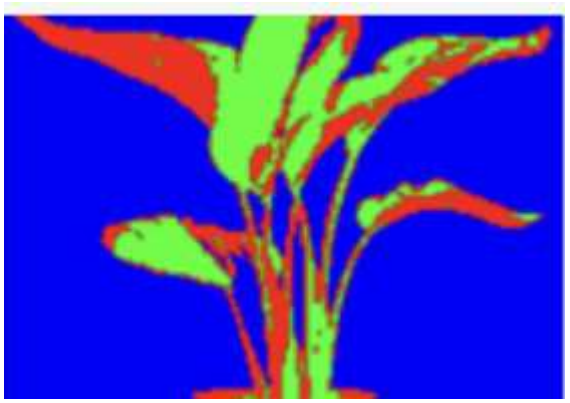
Preprocessing Results



The auto-levels stage produced one of the clearest improvements in the software pipeline. Raw images from the OV7670 often showed strong color cast and uneven contrast, especially under indoor lighting. To reduce this effect, the HPS applies a per-channel histogram stretch to the red, green, and blue channels. For each channel, the software builds a 256-bin histogram, discards the darkest and brightest 0.5% of pixels, and maps the remaining useful range to `[0, 255]`.

The before-and-after comparison shows that auto-levels improves the visual quality of the image by reducing the yellow-green camera bias and increasing contrast. This is important because later stages depend on RGB and ExG features. After auto-levels modifies the RGB channels, ExG is recomputed using the corrected values. As a result, Otsu thresholding and k-means clustering operate on a more normalized feature representation rather than directly using the raw camera output.

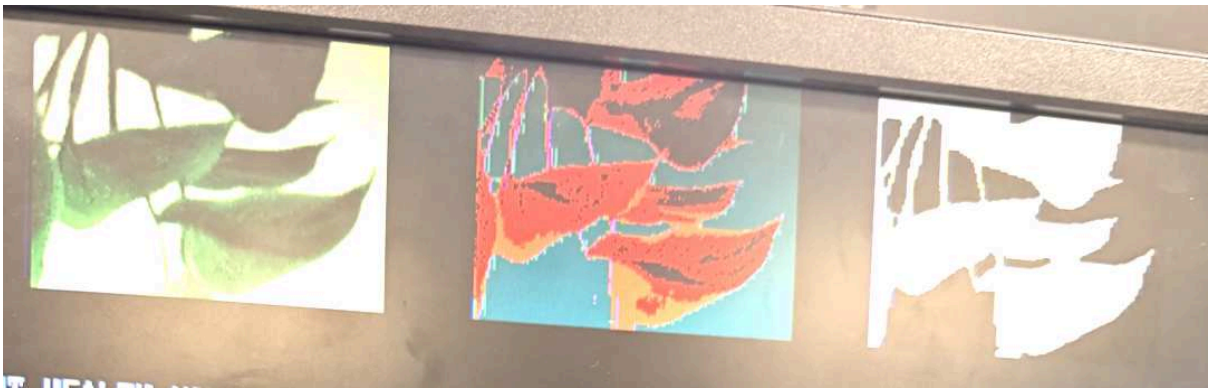
K-Means and Cluster Visualization



The k-means stage also worked as expected. To validate the HPS C implementation, the same input image was tested against the earlier Python software prototype. The resulting cluster visualization was qualitatively consistent with the Python demo, showing that the C implementation preserves the intended behavior of the reference algorithm. This provides confidence that the HPS software correctly performs feature extraction, k-means assignment, centroid updates, and cluster output generation.

On the hardware display, the middle VGA pane shows the cluster visualization generated from the packed meta buffer. This verifies not only the clustering algorithm but also the HPS-to-FPGA writeback path. The HPS successfully packs the cluster ID for each pixel into the meta buffer, and the VGA compositor correctly unpacks the stored cluster ID and maps it to the display palette.

Mask Generation Results



The plant mask was generally successful in the tested examples. In many frames, the binary mask captured the main plant region well and rejected most of the background. Qualitatively, the mask was often close to the expected plant shape, especially when the plant was visually distinct from the background. The largest connected component cleanup helped remove small isolated false positives and made the displayed mask more stable.

However, the mask was not perfect. Some errors came from the sensitivity of ExG to lighting, shadows, camera color balance, and yellow or bright regions in the scene. Since the mask is cluster-based, not a direct pixel threshold, its quality also depends on whether k-means separates the plant and background into meaningful groups. When background colors overlap with plant colors in RGB/ExG space, or when the ExG histogram is not clearly separated, the cluster role logic can include some non-plant pixels or miss parts of the plant.

Overall, the mask generation demonstrates a working and useful segmentation approach, but it remains sensitive to lighting and scene composition.

Verdict Logic and System-Level Behavior

The final verdict logic is functional but less reliable than the intermediate visual outputs. The HPS successfully computes plant area, green ratio, stressed ratio, mean ExG, cluster statistics, and timing information, and these values are correctly written to the text buffer and displayed on the VGA overlay. This confirms that the feature extraction and text output path work properly.

However, the final categorical verdict can sometimes be unstable or unintuitive. This is mainly because the verdict depends on the upstream segmentation and cluster interpretation. If k-means produces clusters that do not correspond cleanly to healthy plant, stressed plant, and background, then the computed green and stressed ratios may become misleading. Similarly, if ExG is distorted by lighting or camera color response, the Otsu threshold and centroid role logic may classify clusters incorrectly. Therefore, the final verdict should be interpreted as a heuristic estimate rather than a fully reliable plant diagnosis.

Verification

A structured verification plan was developed to validate the functionality, timing behavior, memory consistency, and end-to-end integration of the FPGA-based plant monitoring system. The verification strategy combined SystemVerilog and C++ functional testbenches to ensure that each RTL module operated correctly both independently and as part of the complete pipeline. The plan focused on validating reset behavior, synchronous

memory operation, Avalon interface communication, VGA timing generation, frame-buffer consistency, and full system integration between the OV7670 camera interface, frame buffer, metadata buffer, and VGA display subsystem.

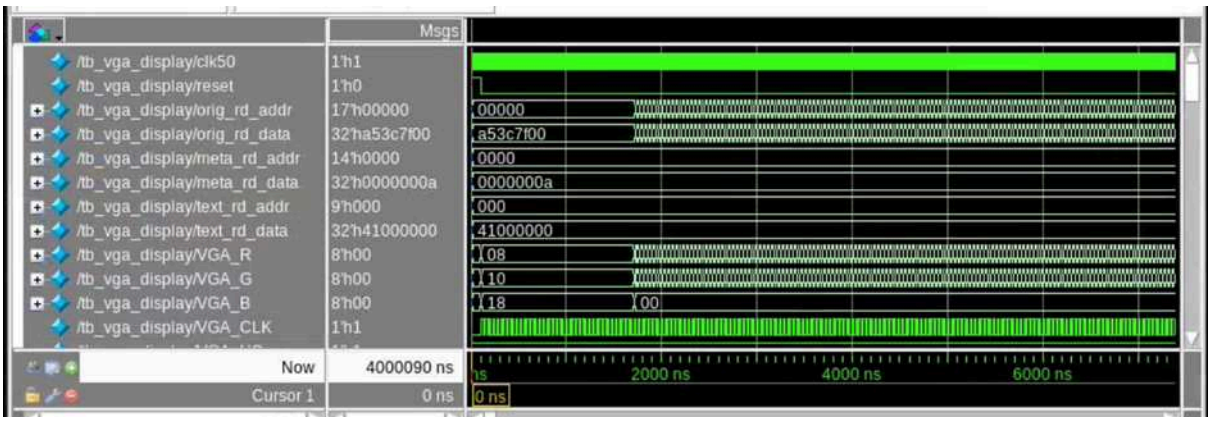
Test ID	Testcase Name	Where Implemented in TB	Objective / Feature Verified	Stimulus Applied	Expected Result / Checks
TC0	Power-On Reset / Basic Boot	All TBs	Verify reset initialization across modules	Reset asserted then deasserted	Registers reset, stable outputs, no X states
TC1	Buffer Write/Read Integrity	tb_buffer.sv	Verify synchronous RAM behavior	Sequential writes & reads	Correct data stored and retrieved
TC2	Control Register Latch + Ack Flow	tb_control_regs.sv	Verify Avalon write + ack behavior	avs_write + ack pulses	capture_start sets and clears correctly
TC3	Debounce Stability Filtering	tb_debounce.sv	Verify glitch rejection	Noisy input + stable press	Single clean pulse generated
TC4	Font ROM Lookup Correctness	tb_font.sv	Verify ROM character mapping	char_code + row scan	Correct 8×16 bitmap output
TC5	Frame Buffer Dual-Port Read/Write	tb_frame_buffer.sv	Verify camera + VGA/HPS access	Concurrent write/read	No corruption, correct latency
TC6	Meta Buffer Memory Consistency	tb_meta_buffer.sv	Verify AVS + VGA consistency	AVS write + VGA read	Data matches across interfaces
TC7	VGA Display Pixel Pipeline Test	tb_vga_display.sv	Verify full display pipeline	Synthetic memory inputs	HSYNC/VSYNC + pixel updates
TC8	Address Mapping Correctness	tb_vga_display.sv	Verify framebuffer addressing	Display scan region	Correct orig/meta/text mapping
TC9	VGA Timing Generation	tb_vga_display.sv	Verify sync generation	Continuous clock	Proper HSYNC/VSYNC timing
TC10	End-to-End System Integration	tb_vga_display.sv (extended)	Verify full system pipeline	Combined stimuli	Stable video output, no glitches

The verification process was divided into modular test cases (TC0–TC10) covering all critical hardware blocks. Initial tests verified proper global reset initialization and ensured that no undefined (“X”) states propagated through the design after reset deassertion. Dedicated memory verification testbenches validated single-port and dual-port RAM functionality, including independent read/write clock domains, latency handling, and data integrity during concurrent accesses. Control register verification confirmed Avalon-MM write/read behavior, acknowledgement logic, and capture-start synchronization. Debounce logic testing ensured stable push-button operation by filtering noisy transitions and

generating only a single clean pulse per press.

Test ID	Testcase Name	Where Implemented in TB	Objective / Feature Verified	Stimulus Applied	Expected Result / Checks
TC0	Global Reset / Basic Boot	All TBs (system-wide)	Verify correct reset initialization across all RTL blocks	Reset asserted then deasserted	All modules initialized correctly, no X propagation
TC1	Buffer Memory Integrity (Single + Dual Clock + Latency)	tb_buffer.cpp	Verify synchronous RAM behavior, clock-domain separation, and read stability	Sequential writes + independent wr_clk/rd_clk toggling	Data integrity maintained, no race conditions, correct readback
TC2	Control Register Interface	tb_control_regs.cpp	Verify Avalon register write/read + control handshake logic	AVS write transactions + capture start trigger	Correct register updates and ack behavior
TC3	Debounce Logic Validation	tb_debounce.cpp	Verify switch debounce and glitch filtering	Noisy input followed by stable press	Single clean pulse generated per press
TC4	Font ROM Functionality	tb_font.cpp	Verify ROM lookup correctness for character generation	Sweep ASCII (A/0/Z) across rows	Stable, correct bitmap output with no X values
TC5	Frame Buffer System (AVS + VGA + Camera + Frame Handling)	tb_frame_buffer.cpp	Verify dual-port memory, latency, frame sync, and HPS access	Write/read via AVS + VGA read + frame boundary updates	Correct data consistency, proper frame reset, no tearing
TC6	Meta Buffer Shared Memory Access	tb_meta_buffer.cpp	Verify shared memory correctness between AVS and VGA	AVS write + VGA read access	Consistent data across both interfaces
TC7	VGA Display Pipeline & Timing	tb_vga_display.cpp	Verify full VGA pipeline (timing, address gen, RGB output, long-run stability)	Continuous clk50 run (~200k cycles)	Stable HSYNC/VSYNC, valid RGB transitions, no deadlock
TC8	End-to-End System Integration	system TB (camera → frame → VGA)	Verify full system pipeline integration	Simulated OV7670 + frame flow	Continuous correct VGA output updates

Additional verification focused on graphics and display functionality. Font ROM testbenches validated correct ASCII character lookup and bitmap generation across row scans. VGA pipeline tests verified address generation, RGB pixel output, HSYNC/VSYNC timing, and long-run display stability under continuous clock operation. Frame-buffer and metadata-buffer tests ensured consistency between AVS, VGA, camera, and HPS accesses without corruption or tearing. Address mapping verification confirmed that original image data, metadata overlays, and text regions were correctly aligned on the VGA display.



Finally, a complete end-to-end integration test simulated the entire image acquisition and display pipeline, including camera frame capture, frame-buffer storage, metadata generation, and VGA rendering. Combined stimuli were applied to emulate realistic operation of the OV7670 camera and VGA controller simultaneously. Expected results included stable video output, valid synchronization signals, correct RGB transitions, and glitch-free display updates. This verification methodology ensured both functional correctness and reliable system-level behavior before FPGA deployment.

Summary

Overall, the project successfully demonstrates a complete FPGA/HPS plant health monitoring prototype. The hardware pipeline works reliably: the camera is configured correctly, live preview is stable, frames can be captured and frozen, and the VGA display shows the original image, cluster visualization, mask, and text telemetry. The software pipeline also works end-to-end, including auto-levels preprocessing, ExG recomputation, k-means clustering, mask generation, feature extraction, and text output.

The strongest results are the hardware integration, live camera display, auto-levels preprocessing, and cluster/mask visualization. The main limitation is the final health verdict, which is sensitive to the quality of k-means clustering and ExG-based role assignment. Future

improvements could include a more robust segmentation model, better color calibration, additional features beyond ExG, or a learned classifier trained on labeled plant images.

Reflection

Who did what

Linus Lei: Hardware Design, Software Design, Documentation

Daolin Li: Software Design, Hardware-software interface, Documentation

Gurleen Kalra: Design Verification, Testbench generation, Documentation

Lessons Learned

One important lesson from this project is that computer vision is much harder and more complex than it first appears. Even a task that seems simple, such as detecting a plant from an image, becomes difficult when lighting, camera exposure, shadows, background colors, and color cast change from frame to frame. The intermediate outputs, such as auto-levels, k-means clusters, and masks, were useful for understanding where the pipeline was working and where it was failing. This showed that a reliable vision system needs not only an algorithm, but also careful preprocessing, visualization, parameter tuning, and failure-case analysis.

Another lesson is that hardware debugging is much more effective when real signals can be observed directly. For the OV7670 camera interface, signals such as PCLK, HREF,

VSYNC, SCCB lines, and pixel data timing are difficult to debug from code alone. A scope or logic analyzer is extremely helpful for confirming whether the camera is actually outputting the expected timing, whether SCCB configuration is successful, and whether pixel data is aligned correctly. Simulation is useful for module-level logic, but for camera bring-up and board-level integration, observing the physical signals is often the fastest way to identify the problem.

A third lesson is that memory budget is precious on an FPGA. A full image buffer can consume a large amount of on-chip memory, especially when storing 32 bits per pixel. This motivated several design choices, such as packing the cluster ID and mask bit into a 4-bit nibble, storing eight pixels per 32-bit word in the meta buffer, and using a compact text buffer for the VGA overlay. The project made it clear that hardware designs must balance clarity, bandwidth, memory usage, and display requirements. Efficient data packing and careful buffer organization are essential when building image-processing systems on limited FPGA resources.

Advice for future projects

Advice for future projects: When working with physical hardware components such as a camera, verify the signals early using an oscilloscope or logic analyzer. Software messages alone may not reveal whether the device is actually configured correctly or producing valid timing. It is also important to build debugging paths on both sides of the system: hardware-level visibility for signals such as clocks, sync, and data lines, and software-level logging for register values, frame status, and processing results. This makes it much easier to determine whether a problem comes from the physical interface, FPGA logic, or HPS software.

File Listing

buffer.sv

control_regs_hw.tcl

control_regs.sv

de1-soc-project.tcl

debounce.sv

font8x16_rom.sv

frame_buffer_avmm_hw.tcl

frame_buffer_avmm.sv

hex7seg.sv

meta_buffer_avmm_hw.tcl

meta_buffer_avmm.sv

ov7670_CommandSet.sv

ov7670_config.sv

ov7670_RGBRcrv.sv

ov7670_SCCB.sv

ov7670_top_video.sv

ov7670_top.qpf

ov7670_top.qsf

ov7670_top.sdc

ov7670_top.sv

sim_cpp/

sim_cpp/Makefile.txt

sim_cpp/buffer.cpp

sim_cpp/control_regs.cpp

sim_cpp/debounce.cpp

sim_cpp/font8x16_rom.cpp

sim_cpp/frame_buffer_avmm.cpp

sim_cpp/meta_buffer_avmm.cpp

sim_cpp/vga_display.cpp

sim_sv/

sim_sv/Makefile.txt

sim_sv/runsim.do

sim_sv/tb_buffer.sv

sim_sv/tb_control_regs.sv

sim_sv/tb_debounce.sv

sim_sv/tb_font.sv

sim_sv/tb_frame_buffer.sv

sim_sv/tb_meta_buffer.sv

sim_sv/tb_vga_display.sv

soc_system.qsys

supplemental_material/tb_config.sv

sw/Makefile

sw/fpga_test.c

sw/hps_processing.c

text_buffer_avmm_hw.tcl

text_buffer_avmm.sv

vga_ball.sv

vga_display.sv