

FPGA-Based Live Interpreter for Sign Language Final Report

1st Brenna Kennedy-Moore
EECS

Massachusetts Institute of Technology
Cambridge, MA, USA
brennakm@mit.edu

2nd Fedaa A Alsoufi
EECS

Massachusetts Institute of Technology
Cambridge, MA, USA
fedaa@mit.edu

3rd Mariya Al Hinai
EECS

Massachusetts Institute of Technology
Cambridge, MA, USA
mariyaah@mit.edu

Abstract—For our final project, we present Sign Language Live Interpreter using an FPGA with an attached camera. It takes in a hand gesture as an input from the camera and matches it to its corresponding alphabet letter. The live camera data undergoes a preprocessing pipeline that applies Canny Edge Detection to extract salient features. The resulting edge image is evaluated against a reference library of 26 letters using the Chamfer Matching Algorithm. The identified letter is then displayed to the user as well as the matching scores for each letter.

Index Terms—Digital systems, Field Programmable Gate Array, ASL Interpreter, Chamfer Matching

I. INTRODUCTION - FEDAA

Creating a sign language interpreter on an FPGA can be challenging. Most existing approaches rely on machine learning models that require large training datasets and significant computational resources. Deploying such models directly on an FPGA is difficult, so for this project we focused on interpreting the 26 letters of the ASL alphabet. Our system captures a live image, performs image processing, and then uses precomputed distance maps to identify the closest match using the Chamfer Matching Algorithm. An overlay module cycles through 26 precomputed binary edge maps and stops when a match is detected, indicating the translated letter. This approach requires considerable storage, and due to the limited memory resources of the FPGA, alternative strategies, such as data packing and down-sampling, were implemented and are presented in the following sections.

II. IMAGE PROCESSING - MARIYA

The image-processing pipeline is based on the Canny Edge Detection algorithm, with each stage implemented as a dedicated SystemVerilog module. The main modules are:

- Line Buffering and 3×3 Neighborhood Generation
- Gaussian Blur
- Gradient Sobel Computation
- Non-maximum Suppression
- Double Thresholding

A. Line Buffer

In the pipeline, each stage (Sobel, non-maximum suppression, double threshold) feeds its pixel stream and coordinates into a line buffer (two for the non-maximum suppression since we have the magnitude and direction line buffers), then an

`always_ff` block turns the three row outputs into a packed 72-bit 3×3 window as seen in the top level. To further optimize our BRAM usage, we have used down-sampled versions of the line buffers with parameters 320×180 .

B. Gaussian Blur



(1) Gaussian Blur Artifacts

The Gaussian blur stage smooths the grayscale image to reduce noise before edge detection. In the design, the `filter` module is instantiated with `K_SELECT = 1`, corresponding to a 3×3 Gaussian kernel operating on the streaming grayscale pixels. After downsampling, we have seen some very noisy artifacts that happen because of the Gaussian Blur filter. I have tried different methods, such as applying the Gaussian filter after every down-sampling stage, starting from 1280×720 and then applying it again after down-sampling to 640×360 , however, the artifacts continued to appear as shown in Fig. 1. In the end, I determined that not applying the Gaussian Blur stage produces the best possible image for edge detection while keeping the down-sampling logic for the sake of meeting BRAM requirements.

C. Finding Gradients

In an image, edges can point in many directions, so the Canny pipeline groups them into four main orientations. The Sobel module first computes the discrete derivatives in the horizontal and vertical directions, denoted G_x and G_y . From these, the code forms the ratio G_y/G_x and uses this value to infer the edge direction. Instead of explicitly computing an angle $\theta = \arctan(G_y/G_x)$, the design compares the ratio

with fixed thresholds that approximate the tangents of the boundary angles [1]. Each range of G_y/G_x is then assigned to one of four discrete direction codes (corresponding to 0° , 45° , 90° , or 135°), so that all edges falling in, for example, a certain slope interval around 45° are assigned the same quantized direction [1]. This quantized direction is passed forward to the non-maximum suppression stage to decide which neighboring pixels should be considered when thinning edges. I first implemented this module `sobel_sv` using an expensive `sqrt` function, this led to failing timing where WNS was -32. This negative slack introduced flickering which led to very unstable edge detection. Removing this `sqrt` function and replacing it with a cheap computation of just adding the absolute values of G_x and G_y . This led to a positive slack of 0.429 with a considerable decrease in flickering. In this way, optimizing the final image processing pipeline has produced satisfactory results despite these compromises.

$$|G| \approx |G_x| + |G_y|,$$

$$r \approx \frac{G_y}{G_x},$$

$$\theta_q(x, y) = \begin{cases} 0^\circ, & |r| < \tau_1, \\ 45^\circ, & r \geq 0 \text{ and } \tau_1 \leq |r| < \tau_2, \\ 135^\circ, & r < 0 \text{ and } \tau_1 \leq |r| < \tau_2, \\ 90^\circ, & |r| \geq \tau_2, \end{cases}$$

where θ_q is the quantized edge direction and τ_1, τ_2 are fixed thresholds chosen to approximate $\tan(22.5^\circ)$ and $\tan(67.5^\circ)$, respectively.

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad K_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

D. Non-maximum Suppression

Non-maximum Suppression (NMS) is implemented as an edge-thinning stage operating on a 3×3 window of gradient magnitudes and their quantized directions. For each center pixel in this window, the module uses its discrete direction code (one of four bins) to select the two neighboring pixels that lie along the corresponding edge orientation [1]. The center magnitude is then compared to these two neighbors: if it is not strictly larger than both, the output is set to zero; if it is the largest, the center magnitude is passed through [1]. In this way, only the strongest responses along each potential edge are kept, while weaker, nearby responses in the 3×3 neighborhood are suppressed.

E. Double Thresholding

After non-maximum suppression, the remaining edge pixels more closely match the true edges, but weak responses from noise can still appear. In this design, a dedicated `double_threshold` module takes the NMS output magnitude and applies two fixed thresholds, `UPPER_THRESHOLD` and `LOWER_THRESHOLD` (here, 220 and 85). Pixels with

gradient above the upper threshold are labeled as strong edges, pixels with gradient below the lower threshold are treated as non-edges, and pixels with gradient between the two thresholds are marked as weak edges [1]. The pixels are then fed into a frame buffer and outputted to the HDMI pipeline after up-scaling to 1280×720 and then captured via capture card to produce our reference library seen below.

III. REFERENCE LIBRARY - FEDAA & BRENNA

We first tried sending 16 bit camera pixels over UART to a laptop in order to build a reference library. Because the UART handled only 8 bit transfers, we created a small trigger module that sent the upper and lower halves separately and then rebuilt the pixels in Python. This worked in simulation, but failed once connected to the live camera, and debugging took too much time. Since this step was not central to the main design, we switched to a USB capture card to gather the images, processed them on the laptop, and stored the 26 reference images in memory files.

Using the capture card, we recorded American Sign Language (ASL) hand gestures performed by Brenna and resized the images to a resolution of 160×90 . We developed two Python scripts to process this data: one to generate distance maps and another for binary edge maps.

To optimize memory usage and prevent BRAM exhaustion, the scripts employ a pixel-packing strategy. For the edge maps, we pack 32 binary pixels into a single memory address. Similarly, for the distance maps, we pack four 8-bit pixel values into one 32-bit word. To summarize:

- **Edge Maps** (`asl_overlay_edge_maps_packed.mem`)
Packing: 32 pixels per line (1 bit per pixel).
Lines per map: $14400 \text{ pixels} / 32 = 450 \text{ lines}$.
Total lines: $450 \text{ lines} \times 26 \text{ maps} = \mathbf{11700 \text{ lines}}$.
- **Distance Maps** (`asl_distance_maps.mem`)
Packing: 4 pixels per line (8 bits per pixel).
Lines per map: $14400 \text{ pixels} / 4 = 3600 \text{ lines}$.
Total lines: $3600 \text{ lines} \times 26 \text{ maps} = \mathbf{93600 \text{ lines}}$.

The packed layouts are summarized in Table I and Table II, and a sample of edge maps is illustrated in Fig. 2.

TABLE I: Memory layout of packed edge maps

Line	Map	Content	Pixels
0	A	[px31 ... px0]	0 to 31
1	A	[px63 ... px32]	32 to 63
...			
449	A	[px14399 ... px14368]	14368 to 14399
450	B	[px31 ... px0]	0 to 31
...			

TABLE II: Memory layout of packed distance maps

Line	Map	Content	Pixels
0	A	$0 \times [\text{Px}3] [\text{Px}2] [\text{Px}1] [\text{Px}0]$	0 to 3
1	A	$0 \times [\text{Px}7] [\text{Px}6] [\text{Px}5] [\text{Px}4]$	4 to 7
...			
3599	A	$0 \times [\text{Px}14399] \dots [\text{Px}14396]$	14396 to 14399
3600	B	$0 \times [\text{Px}3] [\text{Px}2] [\text{Px}1] [\text{Px}0]$	0 to 3
...			



(1) A



(2) H



(3) R

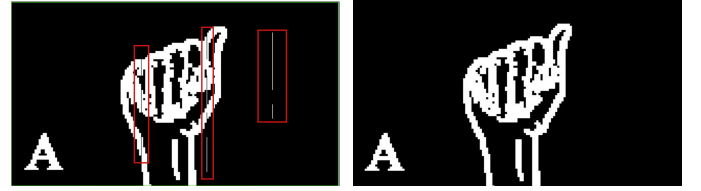
Fig. 2: Sample letters from reference library

IV. OVERLAY EDGE MAPS - FEDAA

The Overlay module provides visual feedback by superimposing reference edge maps onto the HDMI video stream. The system stores twenty six ASL gesture maps in Block-RAM (BRAM) as described above. The module implements an FSM with two stages: *SEARCH* and *LOCKED*. By default, the module operates in the *SEARCH* state, cycling through the map library at two second intervals. When the Chamfer Algorithm identifies a gesture, the *match_valid* signal asserts and the display locks to the detected character and then transitions to the *LOCKED* stage where it waits for the button push for starting another matching round so that it can go back to the *SEARCH* state.

A. Memory and Addressing

The edge maps are loaded in packed format using a *bram_32bit* module. The loading logic runs at the pixel clock. For each pixel, the module computes a linear memory address from the current horizontal (*h_count*) and vertical



(1) Before (artifact present)

(2) After (artifact removed)

Fig. 3: Overlay output before and after pipelining

(*v_count*) positions relative to the Region of Interest (ROI). The index is

$$\begin{aligned} \text{pixel_idx} = & (\text{curr_map} \times \text{MAP_SIZE}) + \\ & ((\text{hdmi_hcount} - \text{ROI_X_ACT}) \gg \text{SHIFT}) + \\ & (((\text{hdmi_vcount} - \text{ROI_Y_ACT}) \gg \text{SHIFT}) \\ & \times \text{MEM_W}) \end{aligned} \quad (1)$$

where *ROI_X_ACT* and *ROI_Y_ACT* are the top left ROI coordinates. The memory width (*MEM_W*) is 160 pixels and the height (*MEM_H*) is 90 pixels. *SHIFT* is computed as $\text{clog2}(\text{SCALE})$ where *SCALE* is a constant of 4.

B. Pipeline Latency and Visual Artifacts

During hardware testing, we observed vertical shearing and line artifacts in the overlay output (see Fig. 3). We traced this to a latency mismatch between address computation and BRAM read timing. The returned data arrived one clock cycle after the associated pixel. We added pipeline registers for the *bit_select* signal to align bit extraction from the packed data, which removed the artifacts.

C. Output Logic

The module generates RGB output based on pixel location. A green bounding box marks the ROI boundaries. Inside the ROI, pixels corresponding to a logic one in the edge map are rendered white, otherwise the camera feed passes through.

D. Testing

We tested the module using a Cocotb testbench that simulated HDMI timing signals. The tests checked ROI boundary behavior and confirmed that overlay pixels appear only inside the 160 by 90 region. We also verified correct map switching and display lock when *match_valid* is high. Pixel level tests confirmed that pipelined memory reads matched the expected coordinates.

V. REGION OF INTEREST (ROI) CAPTURE AND OVERLAY SYSTEM - FEDAA

Our system relies on two modules to record a user's hand gesture and project it back onto the screen for visual feedback: the *roi capture* module and the *roi_overlay_display* module. These modules work together, utilizing a shared BRAM to decouple the high-speed camera write operations from the display read operations.

A. ROI Capture Module

The `roi` module detects a valid hand gesture within a defined on-screen Green Box, down-sampling the high-resolution video stream, and storing the binary edge map into memory.

1) *Architecture and State Machine*: To ensure frame coherency and prevent image tearing (capturing half of one frame and half of another), the module utilizes a Finite State Machine (FSM).

- **CLEAR**: Upon button press, the system enters a clear state, sequentially zeroing out the BRAM to remove previous data.
- **WAIT_FRAME_START**: This synchronization state was introduced to wait until the video beam returns to coordinates (0,0). This prevents "hot-switching" into a capture mid-frame, which results in vertically shifted or partial images.
- **CAPTURING**: The system records active edge pixels only when the beam is inside the ROI.

2) *Signal Processing and Pipelining*: To address timing violations and "dotted line" artifacts caused by single-cycle address calculations, a two-stage input pipeline was implemented:

- 1) **Input Registration**: Incoming coordinates (`h_count`, `v_count`) and 8-bit edge data are pipelined to isolate combinational logic delay.
- 2) **Edge Thickening**: Due to the $4\times$ downsampling factor, thin 1-pixel edges were occasionally missed by the sampling grid (aliasing). A logical "OR" operation was added between the current edge pixel and the previous registered pixel:

$$Edge_{thick} = (Edge_{curr} > 0) \vee Edge_{prev} \quad (2)$$

This effectively widens vertical lines, ensuring they are captured reliably by the lower-resolution BRAM.

- 3) **Address Calculation**: The memory address is calculated in a dedicated pipeline stage to meet timing:

$$Addr = \left(\frac{y_{rel}}{Scale} \right) \times W_{roi} + \left(\frac{x_{rel}}{Scale} \right) \quad (3)$$

3) *True Dual-Port Memory Inference*: Early iterations of the design suffered from read/write collisions, manifested as missing pixels (dotted lines) when the HDMI controller read from memory at the exact cycle the camera attempted to write. This was resolved by separating the logic into two distinct `always_ff` blocks, forcing the synthesis tool to infer a True Dual-Port RAM. This allows the capture logic to write to Port A simultaneously while the overlay logic reads from Port B.

B. Overlay Display Module

This module retrieves the stored binary map and overlays it onto the live video feed using `sw[1]`. The primary challenge in this module was **Pipeline Latency Mismatch**.

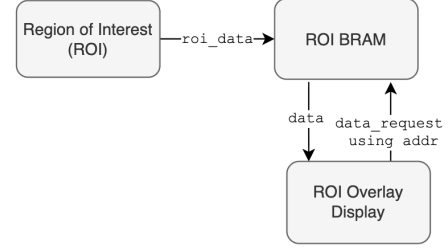


Fig. 4: Block Diagram of the ROI Capture and Overlay System architecture. The design utilizes a Dual-Port BRAM to decouple the high-speed capture logic (Port A) from the display overlay logic (Port B).

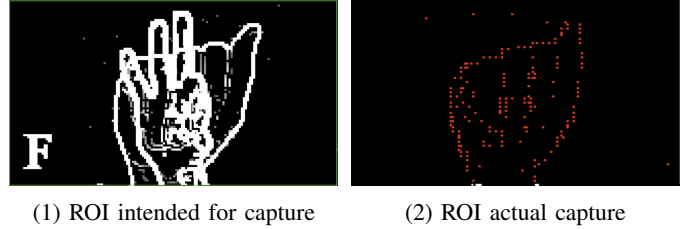


Fig. 5: ROI Capture Sample

1) *Latency*: The BRAM read operation introduces a latency of 2 clock cycles. If the display logic uses the current (x, y) coordinates to decide color output, the overlay appears shifted to the right, creating vertical black bars or "cuts" in the image.

To correct this, the control signals—specifically `inside_roi`, `pixel_valid`, and `show_overlay`—are passed through a shift register pipeline matching the BRAM latency (configurable via parameter `BRAM_LATENCY`). This ensures that the control signals arrive at the output multiplexer on the exact same clock cycle as the valid data from memory.

C. Testing

Testing was performed using the **Cocotb** in Python.

1) *Testbench Strategy*: The testbench (`test_roi.py`) simulates a complete video frame cycle by driving values for `h_count`, `v_count`, and pixel data,

- **Button Debounce Checking**: The testbench asserts the capture button for a duration exceeding the `DEBOUNCE_LIMIT` to verify the trigger logic.
- **Pattern Generation**: A mathematical XOR pattern $(x \oplus y)$ is generated in Python and driven into the edge input.
- **Automated Scoreboarding**: After the capture phase, the testbench reads back the BRAM contents via the read port. It mathematically recreates the expected downsampled image in Python and compares it bit-by-bit against the hardware memory, asserting failure on any mismatch.

This rigorous testing approach confirmed the fix for the "missing row" bug by verifying that the `WAIT_FRAME_START` state correctly aligned the capture window with the video stream.

VI. MATCHING MECHANISM - BRENNIA

The final stage of the interpreter pipeline is the matching module, which compares the live edge map produced by the image processing pipeline against a library of precomputed reference distance maps. Its goal is to identify the alphabet letter whose structural features most closely resemble the shape of the user's hand. We implement this stage using the Chamfer Matching Algorithm due to its simplicity, hardware efficiency, and robustness to small geometric variations.

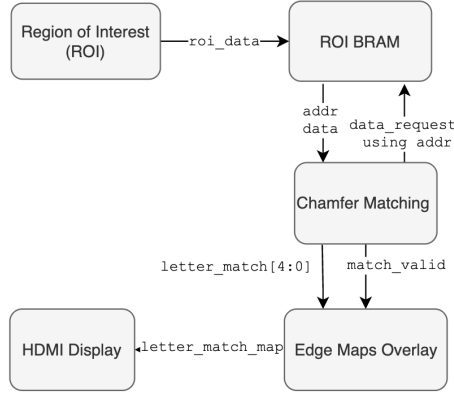


Fig. 6: Chamfer, ROI and Edge Overlay Interaction

A. Chamfer Distance Formulation

Chamfer matching evaluates similarity by summing the distance values at all edge pixels in the live image. Each reference letter is represented by an 8-bit distance transform map, where smaller values correspond to proximity to the edge of the template.

Given a binary edge map

$$E_{\text{live}}(x, y) \in \{0, 1\},$$

and a reference distance map

$$D_k(x, y),$$

the Chamfer score for template k is defined as

$$S_k = \sum_{x,y} E_{\text{live}}(x, y) \cdot D_k(x, y).$$

Lower scores indicate better matches, with an exact duplicate achieving a score of 0.

B. Parallel Memory Architecture

Unlike standard software implementations that iterate through templates sequentially, our hardware architecture optimizes for throughput by evaluating all 26 letters in parallel. The reference maps are stored in independent BRAM (BRAM) instances instantiated via a generate loop.

- **Parallel Access:** The system instantiates LEN_ALPHABET (26) unique ROMs.

- **Addressing:** All ROMs share a common read address pointer, `pixel_index`, which corresponds to the current pixel being requested from the ROI module.
- **Data Flow:** In a single clock cycle, the system retrieves the distance values $D_0(x, y) \dots D_{25}(x, y)$ simultaneously.

This parallel memory structure allows the system to compute the scores for the entire alphabet in a single pass of the image, significantly reducing latency compared to an iterative approach.

C. FSM-Driven Pipeline

The Chamfer Matching is controlled by a Finite State Machine (FSM) that manages two key functional states: Parallel Accumulation and Serial Reduction.

1) *State 1: Parallel Accumulation:* The ACCUM state performs a single raster scan of the $\text{WIDTH} \times \text{HEIGHT}$ of live edge map.

- The FSM increments `pixel_index` from 0 to $N - 1$ where N is the total number of pixels.
- If the incoming `roi_data` indicates an edge pixel, the system simultaneously updates 26 separate accumulators:

$$\text{score_array}[k] \leftarrow \text{score_array}[k] + \text{map_pixel}[k]$$
- A distinct `edge\_count\_array[k]` is also incremented to track the total number of edges pixels in the live image for normalization.

2) *State 2: Serial Reduction and Normalization:* Once the image scan is complete, the FSM transitions to the REDUCE state. To save FPGA area, we utilize a single shared hardware divider rather than instantiating 26 parallel dividers.

- The FSM iterates through the accumulated scores ($k = 0 \dots 25$).
- For each letter, it performs the normalization:

$$S'_k = \frac{S_k}{N_{\text{edges}}}$$

- This process is time-multiplexed; the FSM waits for the `divider_done` signal before proceeding to the next letter.
- As normalized scores are computed, the logic tracks the global minimum to determine the `best_letter`.

D. Output Behavior

Upon completion of the reduction phase, the module transitions to the DONE state and outputs:

- `letter_match`: The index (0-25) of the letter with the lowest normalized score which is used to stop the cycling through the edge maps and display the match.
- `final_scores`: The complete array of normalized scores for debugging and visualization on the `seven_segment_display`
- `match_valid`: A signal indicating that the match is valid and ready for the display driver.

In Fig. 5, we illustrate a high-level overview of how the ROI module interacts with the Chamfer matcher and how the Chamfer matcher interacts with the edge maps overlay module.

VII. SYSTEM EVALUATION - FEDAA

Our design met timing with a positive Worst Negative Slack (WNS) of 0.506. Most of the system has fixed latency, aside from the division module used in Chamfer matching, which has a data-dependent latency of $Q + 2$ cycles, where Q is the quotient. The main cost of maintaining high throughput is register usage. To keep the critical paths fast enough, especially in the Chamfer matching logic, we introduced pipelining that increases flip-flop usage.

Post-route logs show that the design uses 16 RAMB36 tiles and 3 DSP slices. Smaller line buffers rely on Distributed RAM built from LUTs. Further optimization is possible through additional memory packing, where several distance and edge maps values are stored in one memory word and unpacked in logic. This would reduce BRAM pressure at the expense of added logic and extra pipeline stages for bit selection when indexing into packed words.

We committed to demonstrating image processing in hardware, which was achieved as described in Mariya's section. We also implemented the Chamfer Matching Algorithm in hardware, shown in Brenna's section. The overlay and ROI modules were completed as well, as outlined in Fedaa's section. One of our primary goals was to build a working interpreter for all 26 English letters, which is shown in the video submission. The system performs accurate, real-time recognition of ASL letters, and we reached our stretch goal of running all Chamfer calculations in parallel.

While we did not complete a full interpreter for the Arabic alphabet, the design is flexible enough to support all 28 letters by substituting the appropriate edge and distance maps. This same approach could extend to other alphabets and shapes as well.

VIII. FUTURE DEVELOPMENT - FEDAA

The current system forms a solid base for real-time gesture recognition, and several additions could expand its usefulness and accessibility.

A. Functional Extensions

- **Sentence Construction Interface:** To move beyond single-character recognition, a text buffer and a small state machine could be added to assemble words and sentences. A dedicated hand gesture, such as a flat palm, could serve as a trigger for inserting a space character.
- **Peripheral Integration:** Adding a PS/2 or USB interface would allow the system to stream the constructed text to an external terminal or text editor, letting it operate as a full input device.
- **Audio Feedback:** For users who benefit from non-visual output, a lightweight audio path could be introduced to provide text-to-speech for recognized letters or completed words in real time.

B. Technical Insights

Building the system highlighted several important aspects of hardware design, especially timing. One of the main challenges was handling cross clock domain boundaries between

the video capture pipeline, the processing stages and the matching logic. Addressing this required careful pipelining to balance latencies across parallel modules, which reinforced how critical timing analysis is when working with larger FPGA designs. In addition, using the display to observe intermediate values during testing proved extremely helpful, since it made it easier to pinpoint where behavior diverged from expectation. Simulation alone was not always reliable, particularly when modules were tested in isolation rather than within the full integrated system.

IX. CONTRIBUTIONS

- Brenna: Created the reference library, the chamfer matching system, the precomputed distance maps and the python script that generated them, and the distance maps module that loads data into the BRAMs for chamfer matching to use. Debugged all modules written. Worked on top level wiring and instantiations.
- Fedaa: tried generating the reference library using UART, implemented the overlay display for the binary edge maps and the green ROI box, a loading module for the binary edge maps, and developed the module for capturing live ROI data. Worked the packing strategy and wrote the scripts used to create the packed reference library. Produced Figures 3 through 6 and Tables I and II.
- Mariya: Drew the first Block Diagram and designed the initial plan of the system, calculated how to optimize BRAM usage and accounted for worst case scenario, debugging the image processing pipeline by looking at what the HDMI output looks like to make sure it outputted the best edge maps with the given constraints.

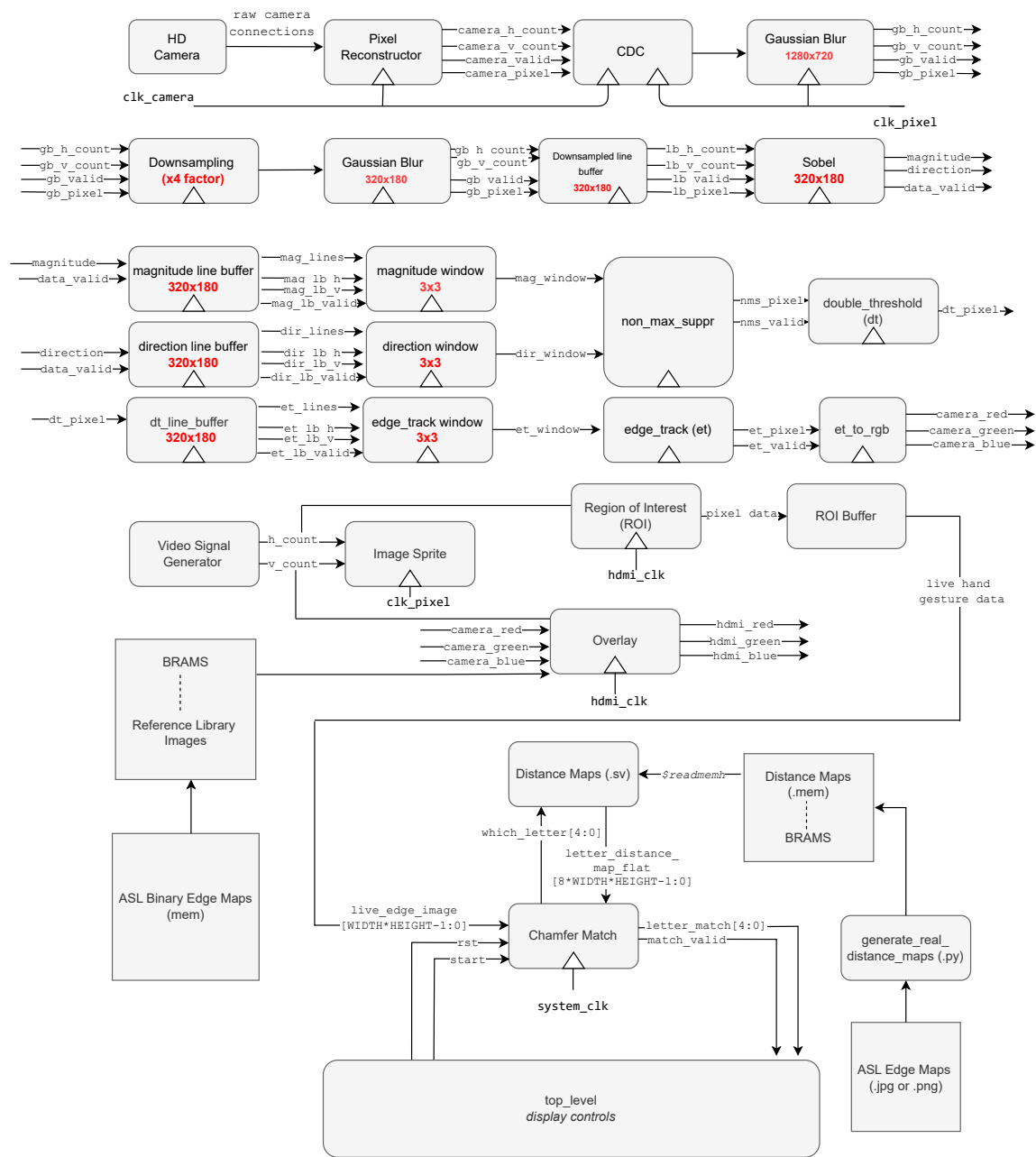


Fig. 7: Full System Block Diagram

ACKNOWLEDGMENT

Thank you to the teaching staff of 6.205 for your support throughout the project!

REFERENCES

- [1] JeffreySamuel. "canny_edge_detection_in_FPGA." *GitHub repository README*. Accessed November 26, 2025. https://github.com/JeffreySamuel/canny_edge_detection_in_FPGA/blob/main/README.md
- [2] D. M. Gavrilă. "The Chamfer System." *Online resource*. Accessed November 26, 2025. http://www.gavrila.net/Research/Chamfer_System/chamfer_system.html