Fitness Program of Guided Activities Final Report

1st Ayyub Abdulrezak
EECS Department
Massachusetts Institute of Technology
Cambridge, MA
yubzak@mit.edu

2nd Esha Ranade
EECS Department
Massachusetts Institute of Technology
Cambridge, MA
eranade@mit.edu

***Abstract*—Our project lays the groundwork for developing a wellness module that can assist people in performing various exercises through positional feedback. Using the camera sensors and our FPGAs, we detect a user's positioning based on placement of specified body parts. Using chroma red colored labels on the user, our system identifies blobs corresponding to various joints (wrist, elbows, shoulders, etc.) on a user's body and performs image processing to assess their pose and draw a stick-figure-like rendering on a screen. Future work may include matching this to a desired pose and providing feedback to the user on how to correct their poses to match, which could be used to assist users in correcting their stretching form to avoid injuries, help direct guided physical therapy routines, or learn new fitness techniques.**

***Index Terms*—FPGA, camera, image processing, blob detection**

I. Background Info (Esha)

The system has 3 primary subproblems: reading and processing camera output, identifying distinct labels and categorizing them as body parts, and rendering a drawing of the user's pose.

The first part of the system builds off of the camera lab from class, taking in camera information and processing it to produce a thresholded pixel of a selected mask. The camera sensors are particularly successful in detecting the chroma red colorspace so we chose to use chroma red labels to mark body parts on a user in the forms of sweatbands – or in our case, on a mannequin doll with duct tape. This is a relatively inexpensive way to simulate the kind of joint tracking that is done in movies with green-screen technology. We perform a mask on the red chroma channel and threshold with some high value such that fewer than 1000 pixels remain in order to reduce noise. The row and column of these thresholded pixels are loaded into a BRAM using the mask_loader module so that we can have access to all of them at once in a buffer, ready to be input into the second part of the system.

Once the thresholded pixels have been identified, the next steps include distinguishing the distinct concentrations of thresholded pixels and mapping them to the specific body part that they are likely to represent. The blob_detection module reads in the buffer input containing the masked and thresholded image and performs k-means clustering with K=8 to classify the distinct blobs into eight clusters representing a user's two wrists, two elbows, two shoulders, neck, and forehead. Due to the nature of random initialization in the k-means clustering algorithm, there may be cases in which multiple centroids represent the same cluster. However, because the algorithm runs repeatedly until a set MAX_ITERS iterations or the centroids converging, they will likely be drawn to distinct clusters by the end of the run. Ultimately, the centers of mass of each of the clusters is returned. The body_part_identification module takes in these centers of mass to identify the body parts that each blob corresponds to using geometric considerations and heuristics.

The third part of the system is responsible for rendering the user's pose. It takes in the labeled locations of each body part and creates a drawing of a "stick figure" that resembles the user's pose by connecting the correct pairs of points, and transmitting it using the HDMI connection to a screen.

The majority of our time was spent on implementing the blob detection algorithm in Verilog as that was the most intensive and novel part of the system.

II. Design Information (Esha)

The logic for our design was split in several different modules. The overall structure is visualized in Fig. 4.

The pixels are first read and processed using the camera, recover, rgb_to_ycrcb, and threshold modules. One major change from the lab 05 code (upon which they were based) is that our rgb_to_ycrcb module only outputs the chroma red color channel, preventing the need for a channel_select module in between. Once processed, the thresholded pixels are loaded into a BRAM. These pixels then go through the full image pipeline, described in more detail below. Ultimately, they are adapted to be rendered through an HDMI connection with the video_sig_gen, scale, rotate, and point_on_line modules. The point_on_line module determines whether a specific pixel is either a centroid or falls on one of the lines joining them, and assigns it an RGB value accordingly such that the final display is entirely black, aside from these specific pixels.

A. Full Image Pipeline Module

The full image pipeline module controls the flow of full frames of image data through each of the following modules: mask_loader, blob_detection, body_part_detection, and

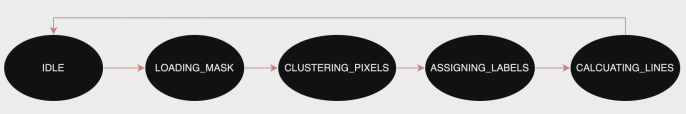

Fig. 1: State machine used within the full image pipeline module

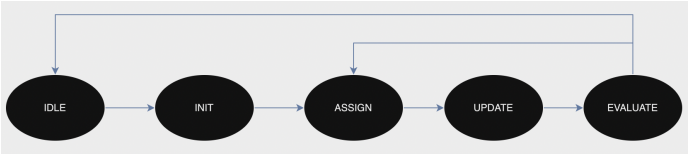

Fig. 2: State machine used within the blob detection module

line_calculator. This allows feedback to be sent back so that a new frame from the camera is only accepted when processing of the previous frame is complete and manages the timing of each module. An internal state machine with the states shown in Fig.2 manages the timing, with start and stop signals from each of the module indicating whether or not to move on. The goal of this module is to capture a full frame whenever it is available, send it along through the various full-frame transformations, and output the necessary rendering information so that it may be used by the display modules.

B. Blob Detection Module

The majority of the logic is done through the blob_detection module which runs a K-Means Clustering algorithm, provided the thresholded image's masked pixel (row, col) "coordinates" as inputs. Internally, the logic is done using a state machine with the following states, as shown in Fig.2:

- IDLE: The state machine will remain in this state while waiting for the coordinates of the masked pixels from a full frame to be processed and loaded into a BRAM, ready to be used by the blob_detection module. Upon receiving a mask_done signal, it will transition to the INIT state.
- INIT: The INIT state sequentially sets the initial centroid locations. We tested several different options of initializing them. Originally, we went with pseudo-randomly selected row and column values anywhere in the image bounds. This was done by combinationally generating a random number with a 16-bit LFSR module, selecting a set bit range from it, and scaling the value such that it would cover the majority of the image's pixels. However, after realizing that intelligent initialization may result in significantly better results, we chose to change our strategy. We considered that the centroid would ideally fall on a masked pixel, and switched the function of our LFSR module to generate a random value, init_index, that could be used to index into the BRAM containing our masked pixel coordinates. This resulted in additional clock cycle counting logic within the INIT module so that we could wait for the specific row and col coordinates of each of the centroids to be read from the BRAM. We scaled the init_index to be in a range within our expected value of num_bram_elems, but ultimately concluded that it would be unlikely for all 8 centroids to be initialized to distinct clusters. Ultimately, we decided to initialize the 8 centroids to be equidistant within the mask_buffer BRAM, with the assumption that each cluster would produce approximately equal numbers of masked pixels. Once all 8 centroids are initialized, we transition to the ASSIGN state.
- ASSIGN: The ASSIGN state sequentially retrieves each masked pixel's coordinates from the BRAM and calculates its squared Euclidean distance to each of the centroids to determine which centroid the pixel is closest to. Because the squared Euclidean distance calculation consisted of several operations – namely, calculating the difference in row and column values between the pixel and centroid, squaring each value, and taking their sum – and was repeated for each of the 8 centroids, we split them up into being calculated over 4 clock cycles each: 1 for taking the two differences, 1 for squaring the row_diff, 1 for squaring the col_diff, and 1 for finding the sum of row_diff_squared and col_diff_squared. The row and column values of a given pixel are then added to a running sum of the rows and columns of each pixel for the centroid that it is closest to using the running_coordinate_sum submodule. This submodule keeps track of all eight centroids' nearest pixels using three array of size 8: one for the row_sum, one for the col_sum, and one for a count of how many pixels were assigned to it. Once all pixels have been retrieved from the BRAM and properly processed, indicated by the number of pixels seen being equal to num_bram_elems, we will transition to the UPDATE state.
- UPDATE: The centroid coordinates will be recalculated sequentially by calculating an average of the row and column values for each cluster. This is done using two divider modules, one for the row value and one for column value, which are fed the running_coordinate_sum values for the specific cluster. If a certain cluster has fewer than 5 pixels assigned to it, instead of dividing, the centroid will randomly be reassigned by generating a new BRAM index using the lfsr_16 module. Once each cluster's centroids have been updated, we move into the EVALUATE state.
- EVALUATE: This stage determines whether to transition back into IDLE or ASSIGN, depending on whether or not the algorithm has terminated. If the maximum number of iterations has been achieved or the centroids have not updated since the previous iteration, the state machine will return to IDLE, ready to accept a new frame of masked pixels. Otherwise it will return to ASSIGN and continue iterating to improve the centroid location.

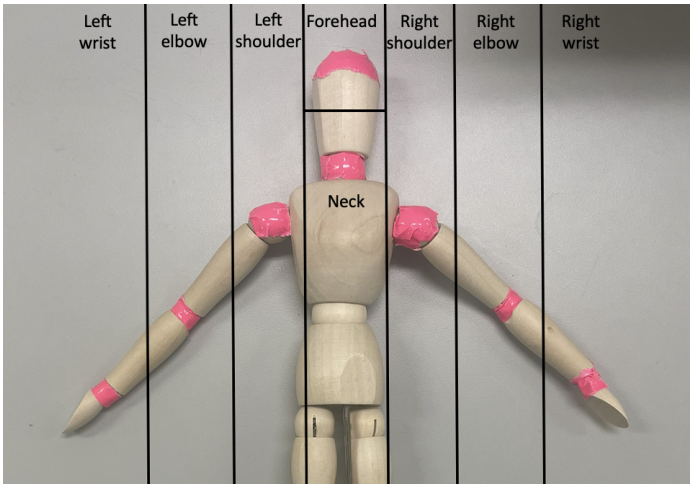

Fig. 3: All detectable poses must satisfy the constraints of the labeled body in the image above.

C. Body Part Detection Module

The body part detection module assigns each of the eight centroids found to its corresponding body part. We decided to narrow the scope of poses that can be detected by the system to only include those that satisfy the following assumptions: in order, from left to right, we should have left wrist, left elbow, left shoulder, right shoulder, right elbow, and right wrist. The forehead and neck must fall between the two shoulders, such that the forehead is above the neck. One such pose is visualized in Fig. 3.

With these assumptions in mind, we first sorted the centroids by their column values using a bitonic sorter in 6 cycles. Afterwards, the 2 centroids deemed to fall in the center are sorted by their row values. Once this full sorting process is complete, the sorted centroids are assigned to body parts in the following order: left wrist, left elbow, left shoulder, forehead, neck, right shoulder, right elbow, right wrist.

D. Line Calculator Module

The line calculator module takes in pairs of points to be connected (left wrist to left elbow, left elbow to left shoulder, left shoulder to right shoulder, right shoulder to right elbow, right elbow to right wrist, and forehead to neck) and computes necessary values to represent the line segments in between them. We use the standard form for linear equations to represent our lines: $A * x + B * y = C$. This prevents us from having to conduct unnecessary division operations, which would be required for slope-intercept form. This module outputs the coefficients and constants (A, B, and C) of each line, along with the row and column bounds of each line in order to properly represent them as segments rather than having them continue across the screen. These values will later be used to compute whether a specific pixel falls on a line when rendering to the screen.

III. Testbenching (Ayyub)

In order to visualize the Verilog hardware module and benchmark its speed, we created System Verilog testbenches to send all of the required signals and analyzed the waveform using the gtkwave software. Simulating with the testbenches provided us with a means of viewing both the Verilog module's outputs and time markers indicating how long it took.

We took a fairly thorough testing approach, which separated the process int two parts: unit (or module) testing, and integration testing. The unit tests tested specific modules, and were designed with edge cases for those specific modules in mind. Some modules that were unit tested in simulation thoroughly include the blob detection and body part identification modules. However, there are often issues integrating modules together, e.g. in top-level modules that go in hardware. We solved this problem both by making wrapper submodules that encompassed more of our module logic and made testing easier (full_image_pipeline) and by writing integration tests that combined the modules together and ensured that they behaved. Figs 12 and 13 depict GTKWave visualizations of our testbenches.

IV. Python Reference (Ayyub)

In order to evaluate our design, we began by implementing the full pipeline on a still frame in software to have a comparison metric for the hardware components.

Our reference image processing is done in Python. We feed in a still, RGB image (as seen in FIG. 2) with dimensions 320x240, in order to be consistent with the future output from the camera sensor. We then convert the image from a RGB color scheme to YCrCb, to be able to work with the chroma red channel. We then mask it with a threshold to get a binary image; we chose a high mask threshold of 238 (out of a maximum of 255) for our initial testing. This resulted in 55 pixels above the threshold, to be ultimately classified into eight clusters. These masked pixels are visualized in FIG. 4 and FIG. 6. The pairs of (row, col) "coordinates" for each of the masked pixels are then passed into scikit-learn's sklearn.cluster.KMeans algorithm, with a K-value of 8, since in our image there are 8 joints to be detected. From this, their center-of-masses are extracted and visualized.

V. Evaluation Results (Ayyub)

Creating the Python reference provided us with a well-tested reference, in terms of both correctness and speed (scikit-learn is a battle-tested and efficient library).

Scikit-learn's sklearn.cluster.KMeans algorithm took approximately 0.04381 seconds to run with 20 iterations and random initialization of centroids. Under the same constraints, our working implementation of K-Means Clustering in Verilog resulted in the following runtimes: 679 clock cycles = 0.00000679 seconds to get through one iteration of the K-Means Clustering algorithm, which includes the processing of the ASSIGN, UPDATE, and EVALUATE states. 21385 clock cycles = 0.00021385 seconds to get through the full K-Means algorithm (20 iterations).

With max_iter = 20, random initialization of centroids, & 55 masked pixels, the following cluster centroids were generated by each of the algorithms. These pixels are also visualized in FIG. 4, 5, 6, 7 below.

Python K-Means Centroids:

[[55 197]
[50 87]
[56 127]
[46 163]
[48 274]
[11 171]
[44 47]
[9 150]]

Verilog K-Means Centroids with random initialization of centroids:

[[48 172]
[190 98]
[54 196]
[10 162]
[48 274]
[44 47]
[55 124]
[50 87]]

In the general case, we saw roughly between 400 and 1000 masked pixels when a subject was in frame. When the fully integrated pipeline ran on this, it resulted in a centroid-updating frame rate of approximately 15-20 fps, which was more than enough for our use case. We observed substantial improvements by initializing to pixels within the range, as well as streamlining many of the modules to be combinational.

In the end, our TNS was 0.0 and our WNS was about 0.6. This meant that not only were we meeting the timing constraints given, but also that we used the combinational logic we could reasonably fit in a cycle. This helped us reduce the cycle count substantially. We also did empirical measurements for parameter tuning, e.g. the threshold for masking and the maximum allowed iterations of the K-means clustering algorithm. For the iterations, we found 10-15 iterations to be more than enough for convergence in most of our cases. Tuning these things lead to substantial speed-ups as well.

VI. Future Work (Ayyub)

On visual inspection of our produced clusterings, we see some interesting results. Our algorithm has some shared clusters with those of the reference, but many are distinct, to varying degrees. The algorithm seems to struggle with assigning all 8 clusters to distinct groupings of pixels, which is a weakness of K-means itself, especially when you have clusters of varying depth and get unlucky with your random initialization. There are two different ways that we plan to tackle this moving forward.

The first is to increase the number of masked pixels to give more resolution to each cluster and reduce the effect of outlier pixels. This needs to be balanced with speed, as each masked pixel in the input increases the number of cycles by a large factor.

The second way is to simply do more trials. One of the recommended ways to improve K-means, even with unlucky initializations, is to simply do it multiple times, score each result, and take the best looking one. This is something we will explore further, and have tested this to work quite well at improving performance in Python with T=5 trials.

Another component we would like to explore more is the visualization of the information. We tried two different approaches, rendering lines between the relevant body parts, and drawing body blocks over the resulting image. The body blocks were simpler and had less overhead. The lines would be cleaner, but are trickier to get to display well. We would like to explore different line rendering algorithms, such as Bresenham's, further in future work.

VII. Appendix

Link to the Github repository for the project: https://github.com/esharanade/fpga-yoga

VIII. Diagrams

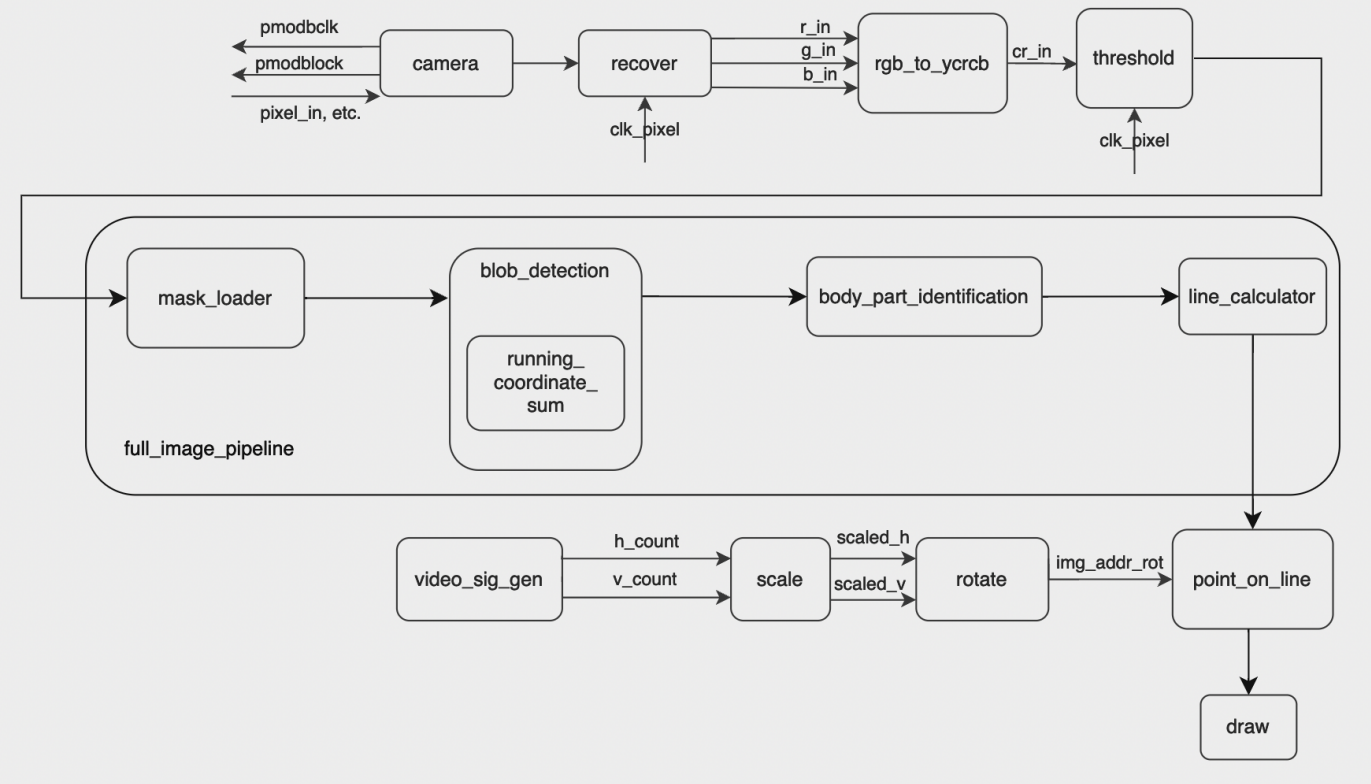

Fig. 4: block_diagram representing the flow of data through the system.

Fig. 5: Lunge pose used for testing the blob detection algorithm.

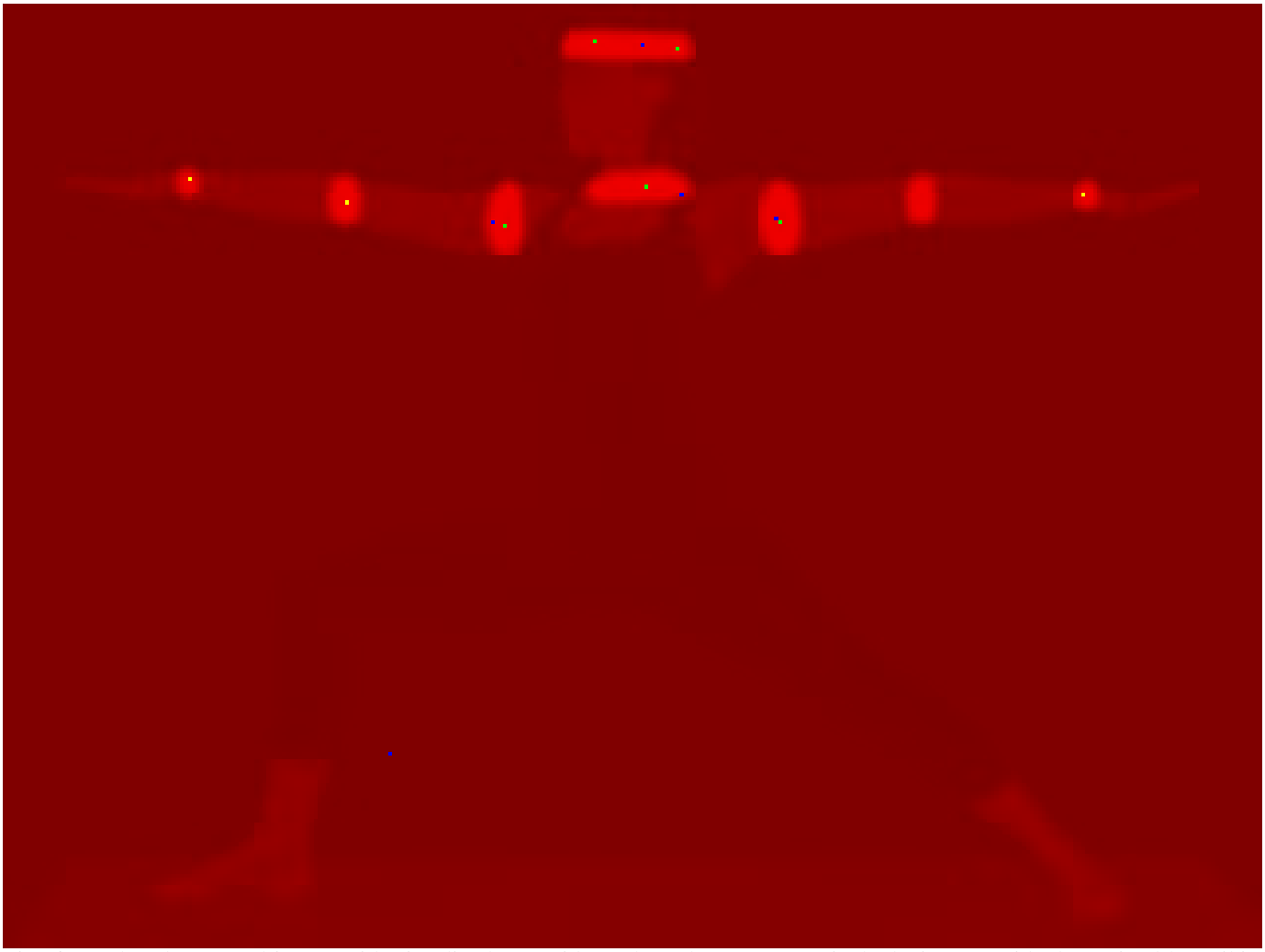

Fig. 6: Seed 1 chroma red image with visualized centroids. Color scheme: green pixels depict centroids determined by scikit-learn's machine learning package for k-means clustering, blue pixels depict centroids determined by our Verilog algorithm, yellow pixels depict centroids identified by both.

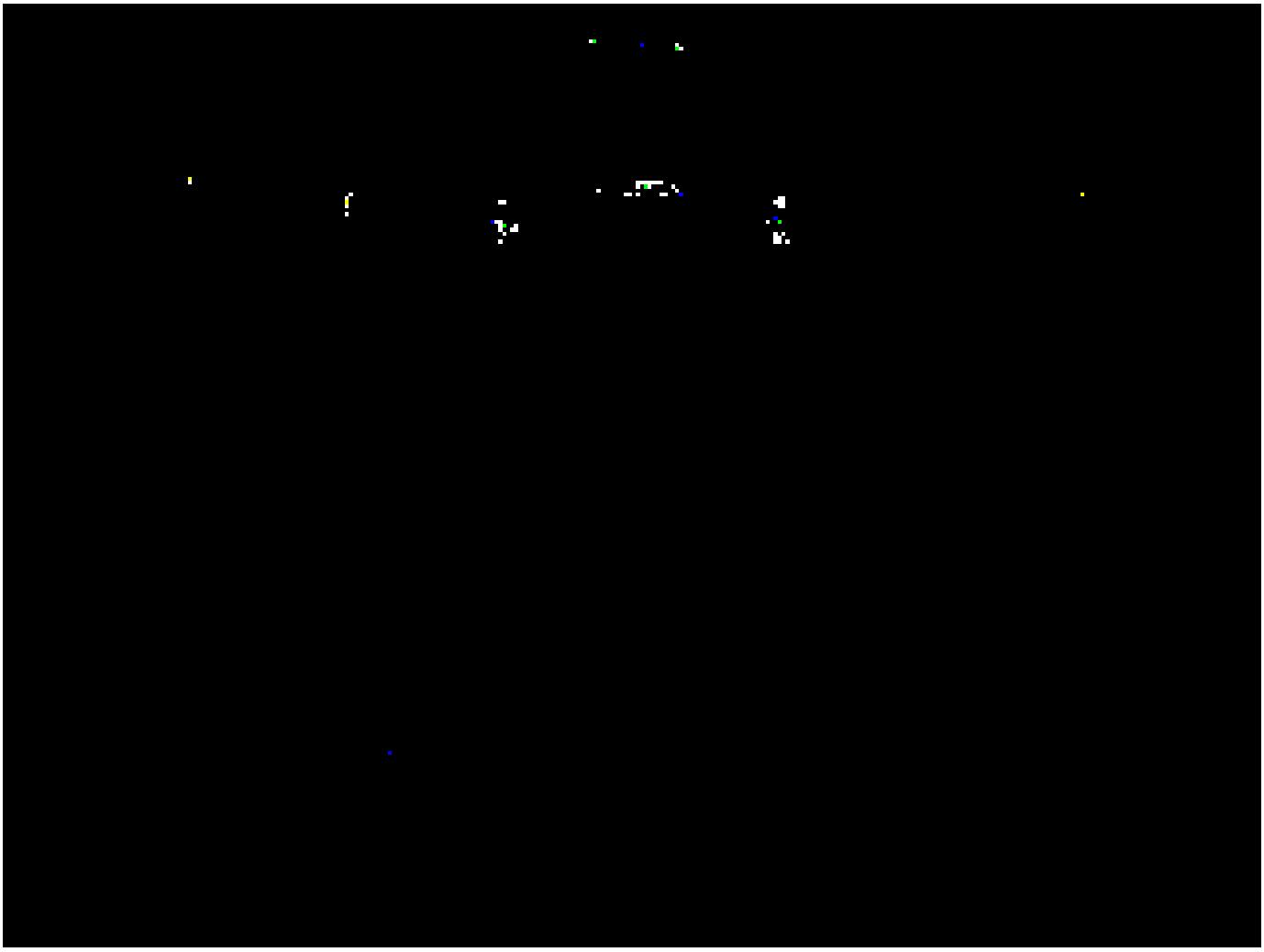

Fig. 7: Seed 1 thresholded image with visualized centroids.

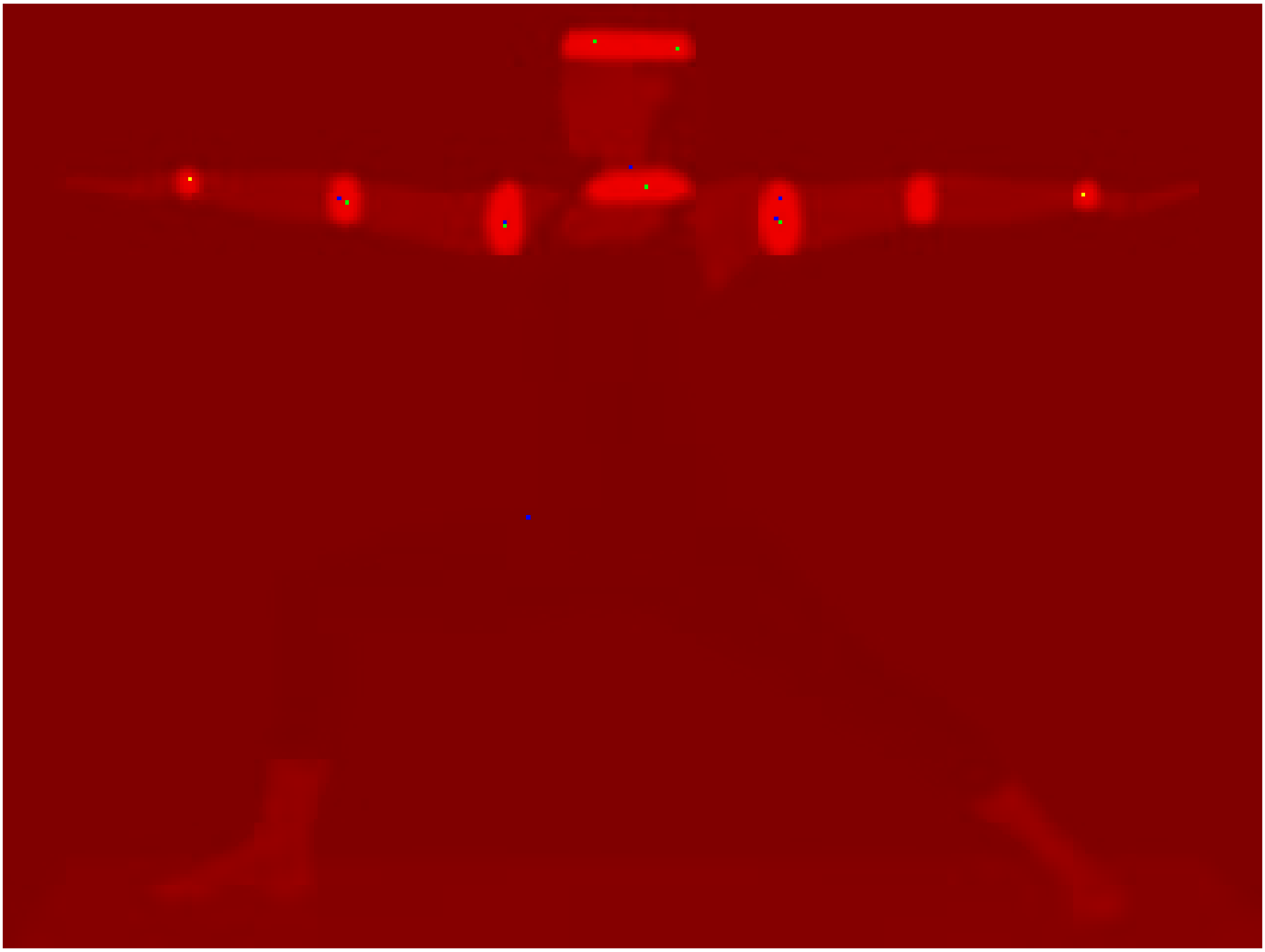

Fig. 8: Seed 2 chroma red image with visualized centroids.

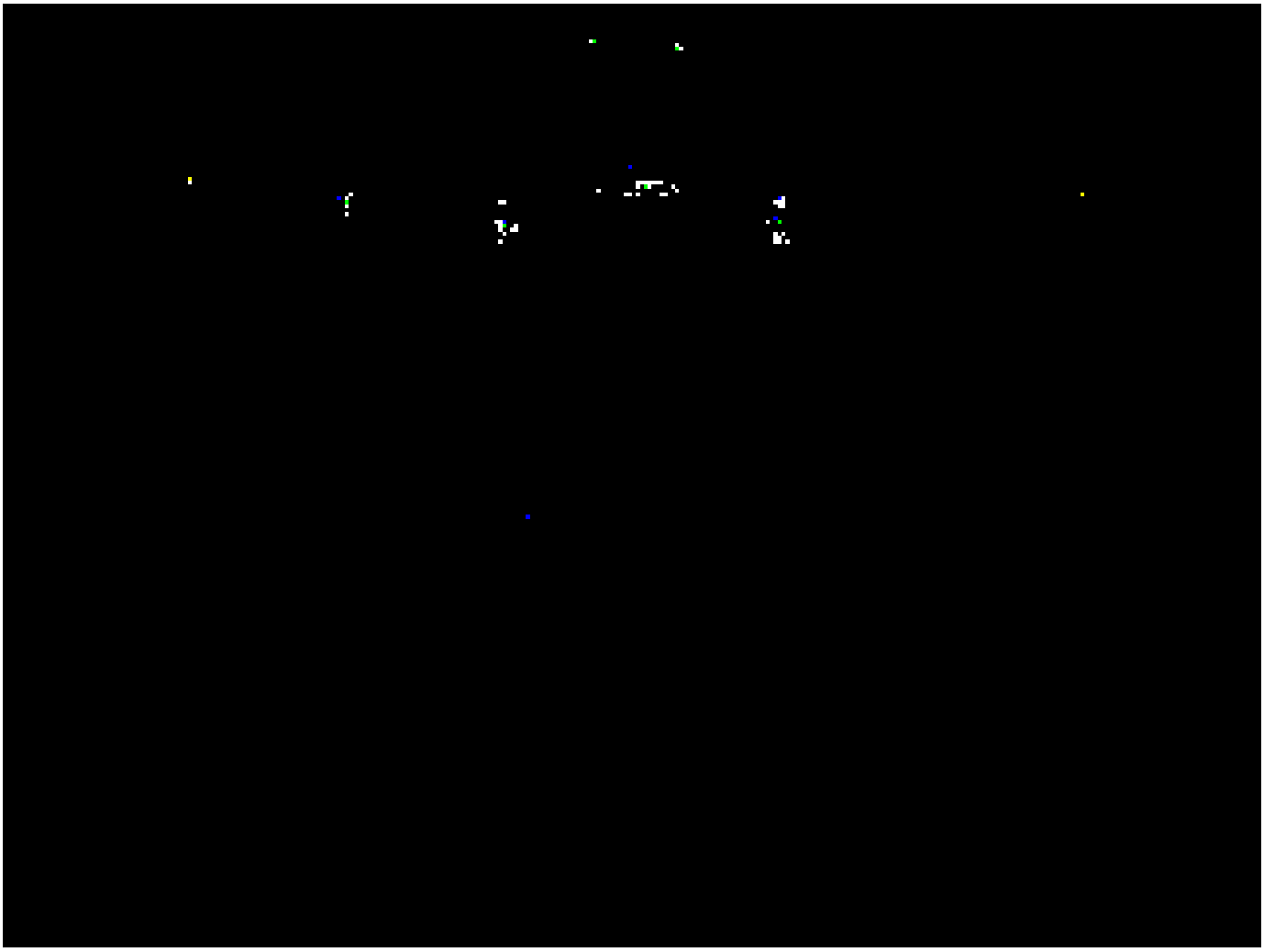

Fig. 9: Seed 2 thresholded image with visualized centroids.

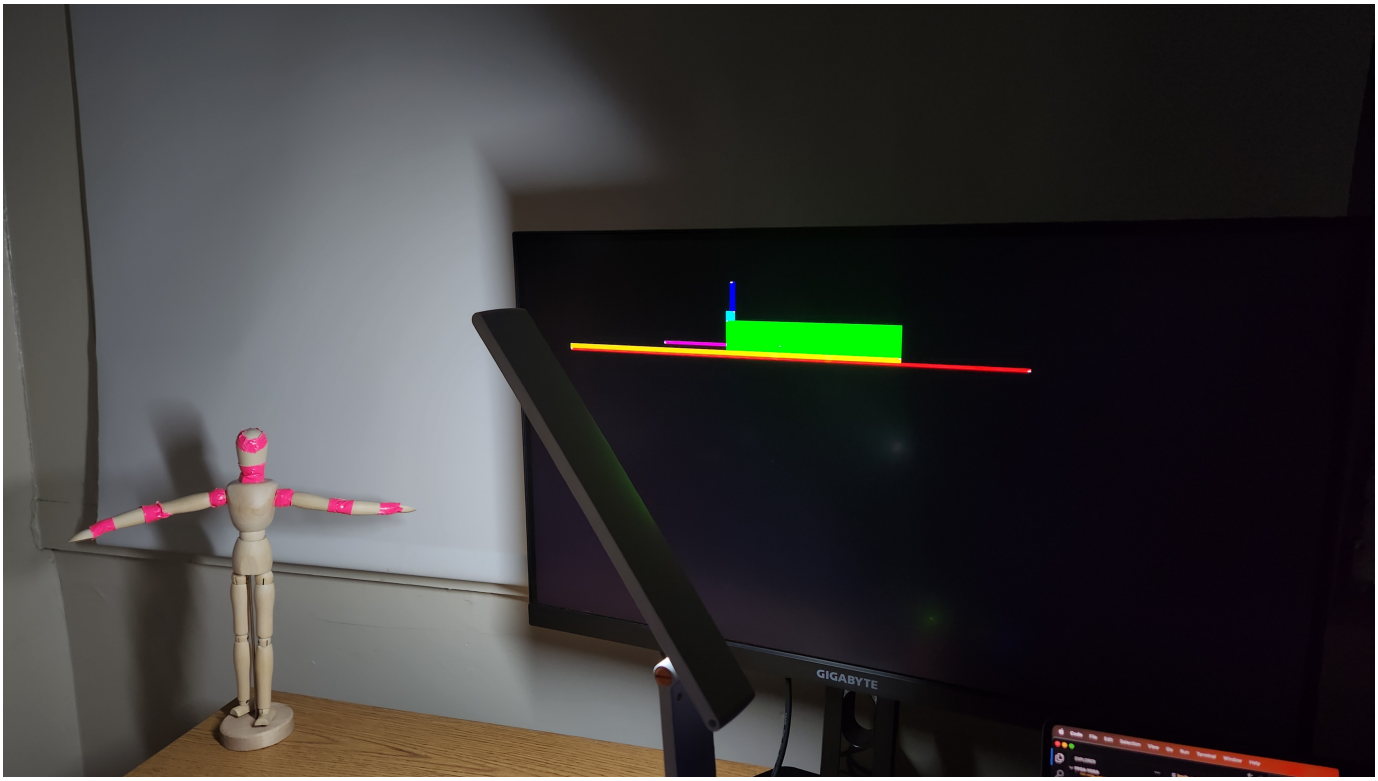

Fig. 10: Visualization of pose with body blocks superimposed

Fig. 11: Centroids displayed, roughly representing the doll's pose

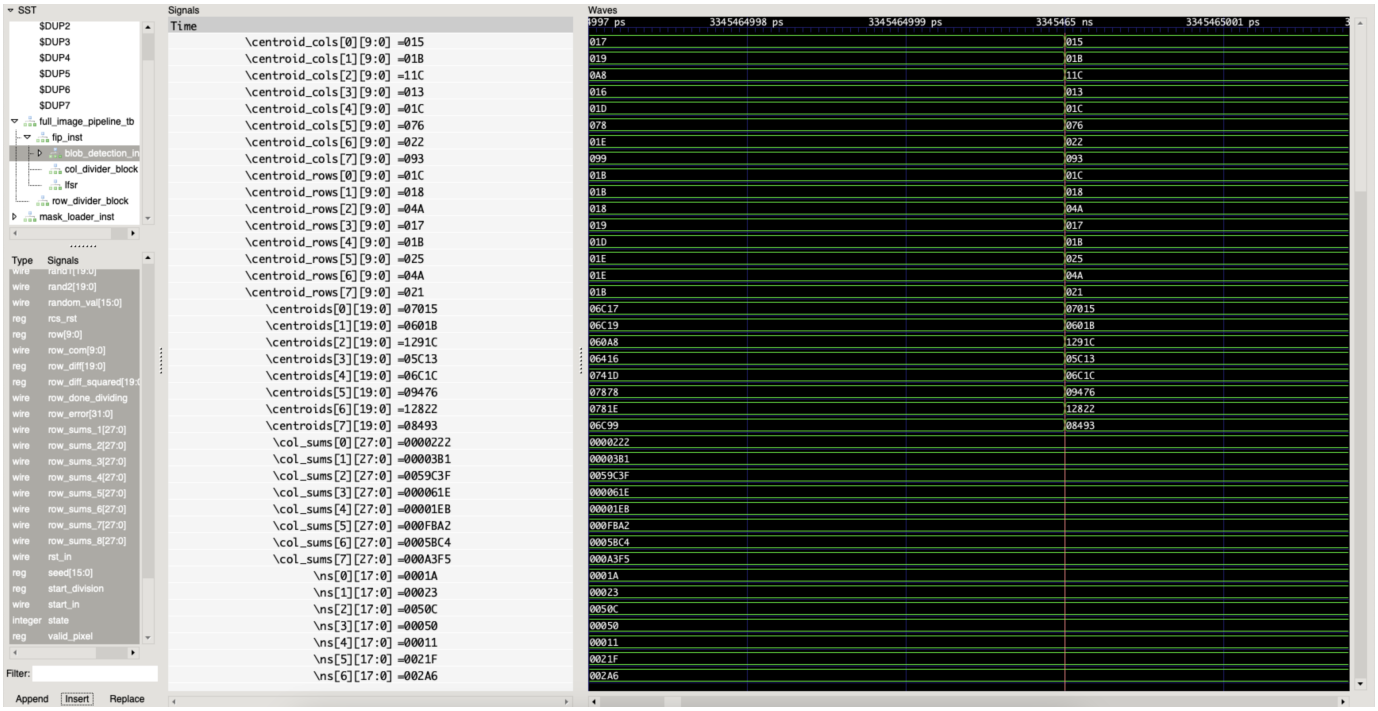

Fig. 12: Unit testbench

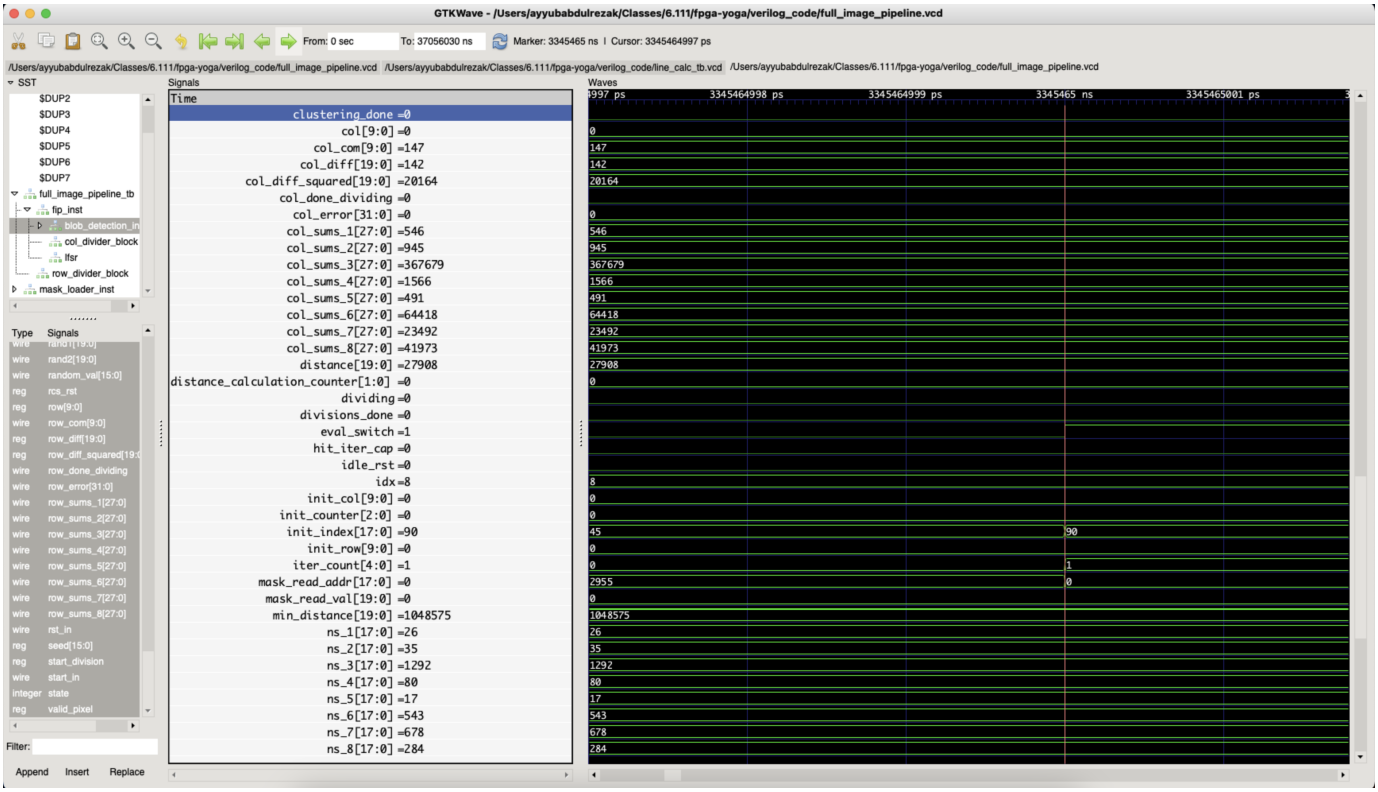

Fig. 13: Integration testbench