Columbia University
Department of Electrical Engineering

Uncalibrated Stereo Infrared Beacon Tracking on the DE1-SoC

Dual OV7670 Moment Pipelines, Runtime Calibration, and 3D Reconstruction

Final Project Report

CSEE W4840 Embedded Systems

Leen Alshorafa and Kuan Zhang

May 2026

Contents

1 Introduction

This project implements a real-time stereo infrared beacon tracker. Two OV7670 cameras stream synchronized grayscale pixel data into the FPGA, where each camera has a dedicated moment-accumulation pipeline that thresholds the image and compresses every frame into foreground area, horizontal beacon moment, and vertical beacon moment. The HPS reads only those compact moments, computes centroids to find the beacon coordinates, performs runtime stereo calibration from user-logged correspondences, triangulates the beacon position, and applies world-frame alignment through origin, floor-plane, and scale calibration. Precise motion tracking is achieved without the measurement of camera positions and with very low-cost camera hardware.

2 Implemented System Overview

The final system has four major subsystems:

1. **Dual OV7670 acquisition.** Modified with IR-block filters removed and replaced with IR-pass filters to isolate only 940 nm light from the beacons, capturing in 640 x 480 VGA. I2C camera control register communication was implemented through bit-banged software PIO.

2. **FPGA moment extraction.** Each camera has an independent coordinate decoder, threshold stage, and accumulator bank for U, V, and area.

3. **Debug frame capture.** A selectable frame-buffer path stores a full grayscale frame for inspection, threshold tuning, and intrinsics calibration.

4. **HPS runtime software.** The C terminal configures the cameras, reads moment registers, logs calibration correspondences, solves for projection matrices, and triangulates live points. The runtime mode lets the user set an origin, align the floor plane to the world vertical axis, and set metric scale from a measured distance.

3 Hardware Block Diagram

3.1 Datapath and Control

The design implements a dual-camera image-processing pipeline withseparate 25 MHz pixel-clock domains for Camera A and Camera B, plus a 50 MHz system/control clock domain. Each camera stream provides pclk, href, vsync, and 8-bit pixel data into a coordinate decoder, which generates pixel-valid timing, frame-start/frame-done events, and (u, v) pixel coordinates. The pixel stream is thresholded using an 8-bit active threshold latched at frame start from the CONTROL register, producing a foreground mask that feeds an accumulator bank.

For each camera, the accumulator computes three 32-bit image moments per frame: foreground area, summed u-coordinate, and summed v-coordinate. On frame completion, results cross into the 50 MHz clock domain through toggle-based clock-domain crossing and synchronizers, then latch into software-readable registers A_RESULT, U_RESULT, and V_RESULT. The DONE bits are hardware-set and software-cleared; result registers only update after the corresponding DONE bit is cleared, preventing software from reading partially overwritten results. Frame capture for the realtime pipelines continue regardless of DONE status, to prevent the FPGA from missing a frame start VGA edge due to software not having cleared DONE fast enough.

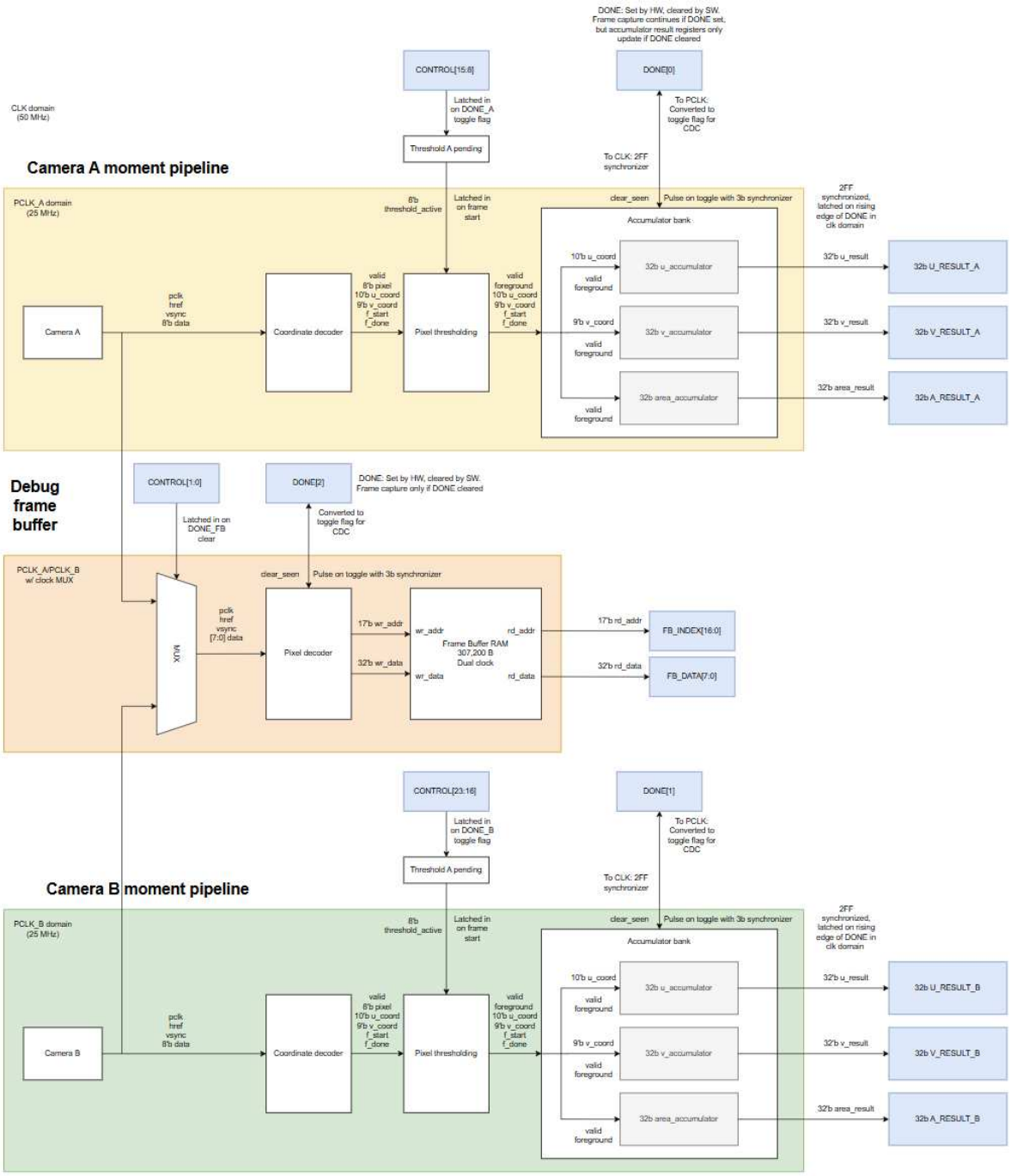

Figure 1: Hardware block diagram

The design also includes a selectable debug frame-buffer path. CONTROL[1 : 0] selects which camera stream is written into a dual-clock 307,200-byte frame buffer in FPGA BRAM, with writes occurring in the selected camera pixel-clock domain and reads exposed to software through FB_INDEX and FB_DATA. Frame-buffer capture uses its own DONE[2] handshake, also synchronized across clock domains, so software can capture and inspect raw image data independently of the moment-accumulation pipelines. This module runs only for single-shot tests, so does not impact realtime processing; therefore, unlike the moment pipelines, it halts frame capture if DONE is not cleared.

3.2 Physical modules

The diagram in Figure 1 is abstracted to illustrate dataflow, but not perfectly representative of divisions between modules. In the physical design, the entire hardware component is wrapped into the top-level `imgproc.sv` block, which decodes results from the memory-mapped control/status registers, generates the camera `xclk`, selects the debug frame-buffer source, and exposes results to software.

Each camera is handled by `camera_moment_pipeline.sv`, which contains the coordinate decoder, thresholding logic, moment accumulator, and clock-domain crossing logic. The `pixel_coordinate_decoder.sv` module applies thresholding internally, as the operation is extremely simple and follows naturally after luminance decoding. It tells `moment_accumulators.sv` whether the incoming pixel is a valid luma pixel that made it past thresholding, which then computes the foreground area and coordinate sums for each frame. In parallel, `frame_buffer.sv` provides a debug path that captures one full selected camera frame into M10K RAM, packing four Y pixels into each 32-bit word for software readout through `FB_INDEX` and `FB_DATA`.

3.3 Pixel decoding

Most of the complicated image-processing logic comes from decoding the raw VGA camera stream and extracting luminance data from the YUV format. `pixel_coordinate_decoder.sv` runs entirely in the camera `pclk` domain and converts the raw `href`, `vsync`, and 8-bit data stream into valid Y pixels with corresponding (u, v) image coordinates.

The decoder assumes a YUV/YUYV-style byte stream, where only every other byte is luminance. A small byte-phase counter tracks the position within the YUV sequence, allowing the module to ignore chroma bytes and assert `pixel_valid` only when the current byte is a Y pixel. During active capture, it increments the horizontal coordinate across each row and wraps to the next vertical coordinate at the end of the line.

Frame capture is synchronized using `vsync`. A falling edge of `vsync` starts a new frame, after which the decoder counts valid Y pixels through the full 640×480 image. When the final pixel is reached, it asserts `frame_done` for one `pclk` cycle and returns to the waiting state. If `vsync` occurs unexpectedly during capture, the frame is aborted and the decoder resets for the next frame.

3.4 GPIO and Physical Setup

Camera A uses GPIO0 for the parallel data and timing signals:

`GPIO_0[7:0]`	DATA
`GPIO_0[8]`	HREF
`GPIO_0[9]`	VSYNC
`GPIO_0[10]`	PCLK
`GPIO_0[26]`	XCLK

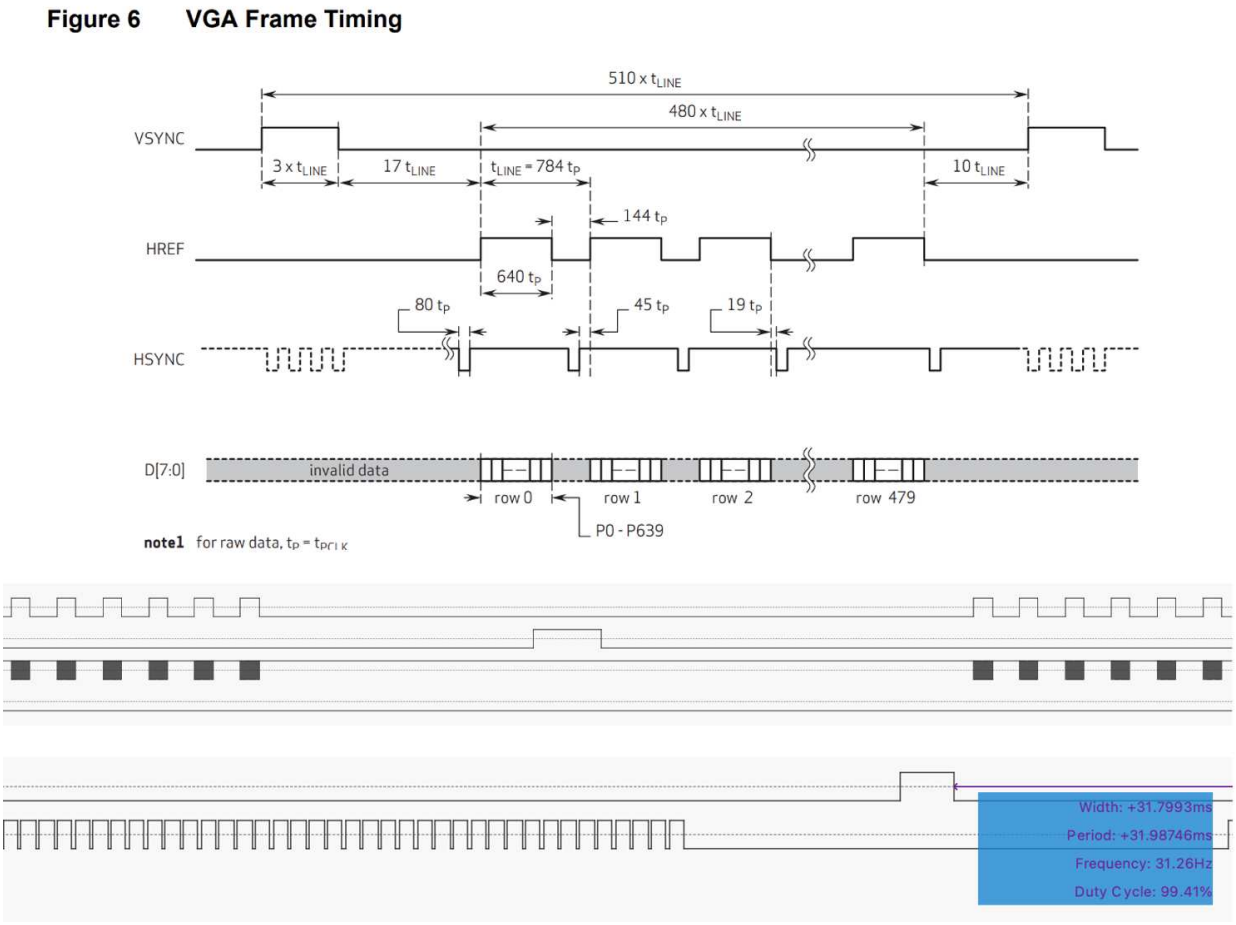

Figure 2: VGA timing from the camera datasheet above, and two traces we scoped to verify our hardware timing below. The first shows HREF and DATA, with a VSYNC pulse between frames. The bottom shows a measured VSYNC frequency of 31.26 FPS, which was exactly what we measured in software, confirming we weren't skipping frames.

Camera B mirrors the parallel camera interface on GPIO1. Additional GPIO0 pins are reserved for software-controlled I/O and camera configuration:

`GPIO_0[35:27]`	Software PIO
`GPIO_0[22]`	Software I2C SCL
`GPIO_0[23]`	Software I2C SDA
`GPIO_0[25:22]`	Software I2C / reserved control pins

The implementation bit-bangs SCL and SDA through memory-mapped GPIO rather than relying on a separate I2C controller, which simplified prototyping. We used external 4.7K pullups on both lines. To write an OV7670 register, we send a START condition, transmit the camera's 7-bit address 0x21 with the write bit, send the register address byte, then send the data byte, checking for ACKs after each byte before sending STOP. To read a register, we first do a short write transaction to select the register: START, device address with write bit, register address, STOP. Then we start a second transaction: START, device address with read bit, read one byte from the camera, send NACK because we only want one byte, then STOP. The configuration code resets the sensor, programs VGA/YUV-related registers, fixes chroma fields for grayscale behavior, sets full output range, and verifies register readback before capture. The default runtime settings in the source are:

$$\texttt{gain} = \texttt{0x0A0}, \qquad \texttt{exposure} = \texttt{0x0100}, \qquad \texttt{AGC} = \texttt{0}, \qquad \texttt{AEC} = \texttt{1}.$$

4 Hardware/Software Interface

Table 1: Hardware/software partition.

Subsystem	FPGA responsibility	HPS/software responsibility
Camera timing	Decode PCLK/HREF/VSYNC into pixel coordinates and frame boundaries	Configure OV7670 registers and verify camera readiness
Foreground extraction	Threshold each pixel against camera-specific threshold fields	Choose thresholds and minimum valid area at runtime
Centroid measurement	Accumulate A, U, and V per frame	Compute $c_u = U/A$, $c_v = V/A$, reject invalid area
Calibration	Provide fresh, synchronized camera observations	Log pairs, estimate geometry, save projection matrices
Tracking	Produce frame-consistent moments at camera rate	Triangulate, transform into world coordinates, print/plot results
Debugging	Capture one selected full frame into RAM	Export PGM frames and inspect labeled calibration images

The implemented C code maps the lightweight HPS bridge at base address $0xFF200000$ with a $0x1000$-byte span. The camera moment and debug registers are exposed as word offsets. Table 2 describes the interface used by the current software.

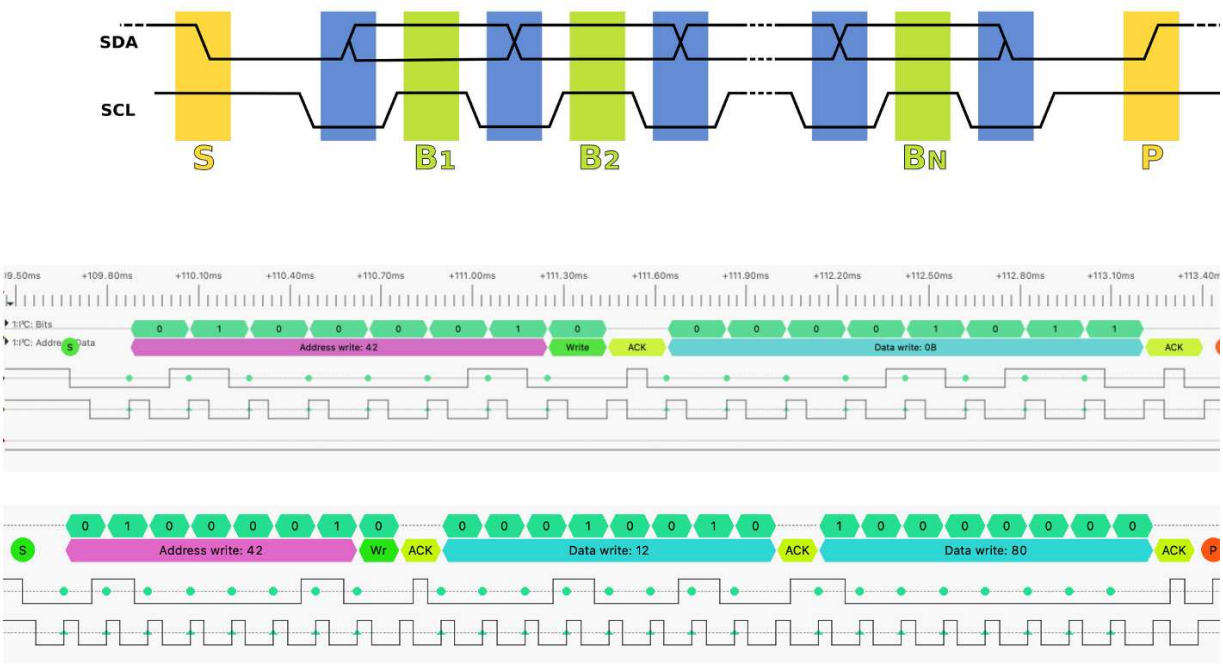

Figure 3: Camera-control verification. The OV7670 sensors were configured through software bit-banged SCCB/I2C transactions using HPS-controlled PIO. Above is ideal I2C bus timing. Below are two actual I2C packets scoped with a logic analyzer. The first shows a write setting register address to 0x08; the second writes 0x80 to register 0x12 (common control 7).

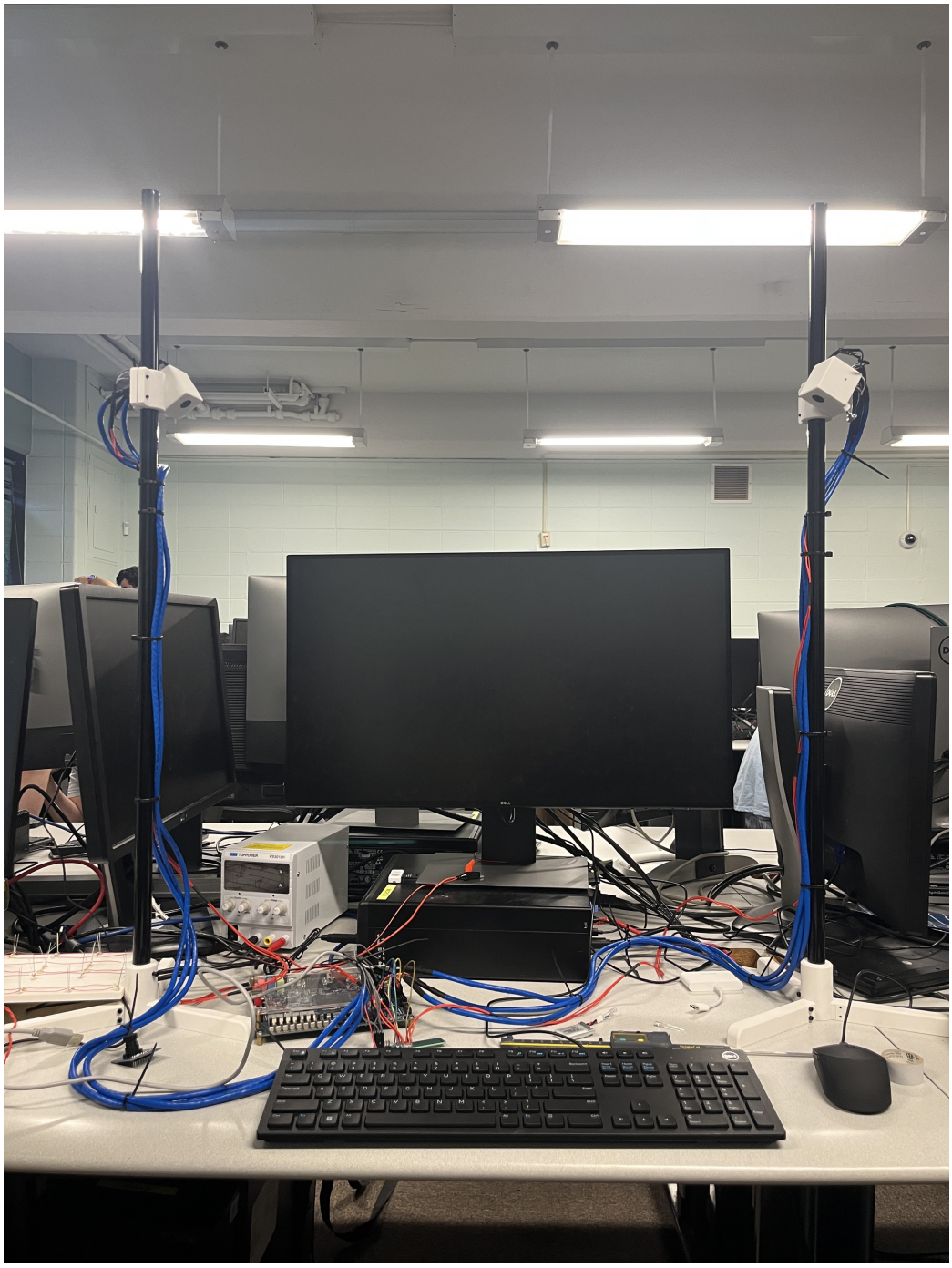

Figure 4: Full setup. Custom camera mounts were designed and printed; Cat 6 cables were spliced and repurposed for fast PCLK-domain signals, with each twisted pair carrying a signal and return ground.

Table 2: Implemented memory-mapped register interface.

Offset	Register	Meaning
0x00	AREA_A	Foreground pixel count for Camera A
0x04	U_A	Horizontal moment $\sum u$ for Camera A
0x08	V_A	Vertical moment $\sum v$ for Camera A
0x0c	AREA_B	Foreground pixel count for Camera B
0x10	U_B	Horizontal moment $\sum u$ for Camera B
0x14	V_B	Vertical moment $\sum v$ for Camera B
0x18	DONE	DONE bitfield: bit 0 for Camera A moments, bit 1 for Camera B moments, bit 2 for debug frame buffer
0x1c	CONTROL	Bits [1:0] select debug frame storage mode; bits [15:8] hold Camera A threshold; bits [23:16] hold Camera B threshold
0x20	FB_INDEX	Debug frame-buffer word index
0x24	FB_DATA	Packed debug frame-buffer data, read as four 8-bit pixels per 32-bit word

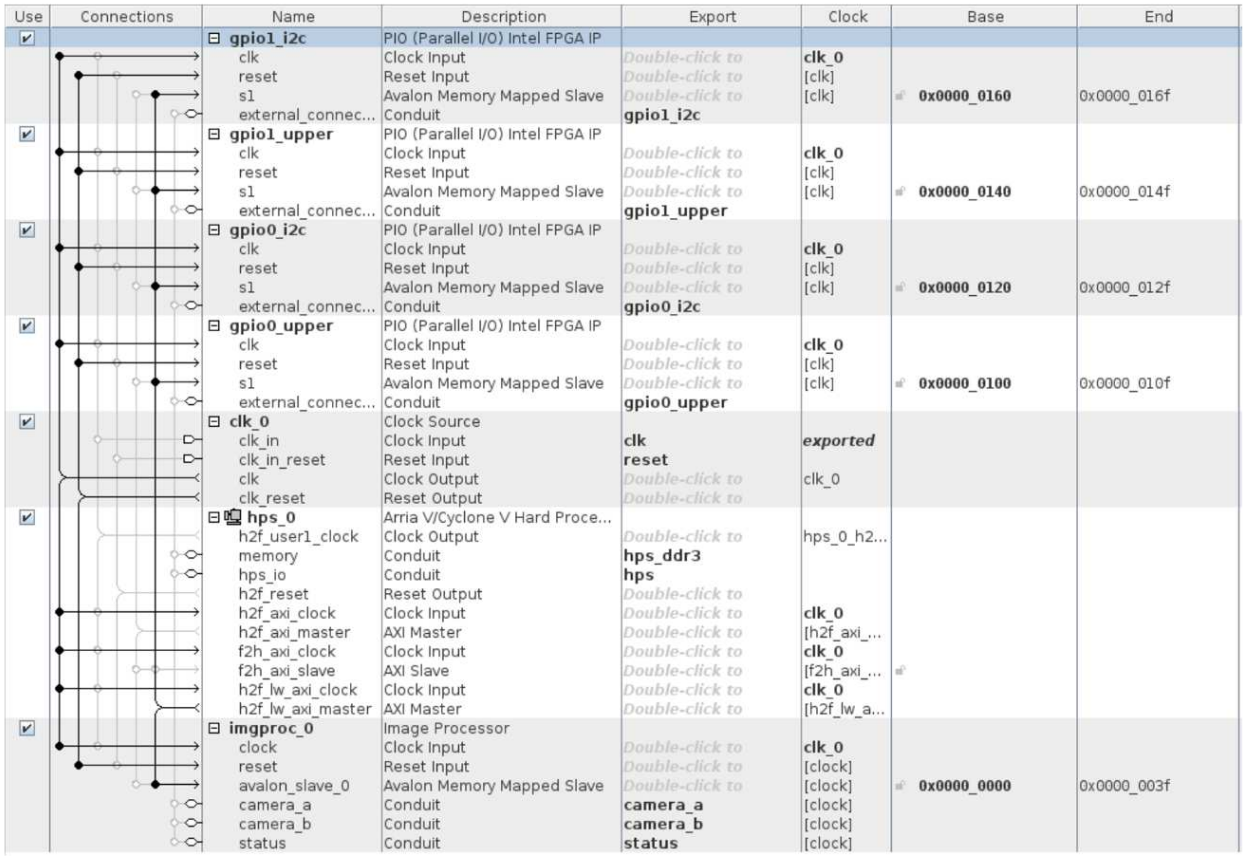

Figure 5: Platform Designer/Qsys view of the HPS-to-FPGA interface.

The software pattern is:

1. Wait until at least one camera DONE bit is visible.

2. For each visible DONE bit, read that camera's area and moments.
3. Clear only the DONE bit for the camera that was just read.
4. Wait until the hardware observes that clear.
5. Keep each camera's latest fresh result until both cameras have produced one new result.
6. Only then accept the pair as fresh.

5 Runtime Software Architecture

5.1 Startup Flow

The main program initializes application state, attempts to load saved projection matrices from `stereo_projection.txt`, configures both cameras, and then chooses between runtime tracking and recalibration. If no valid calibration file is present, calibration is mandatory. If a file is present, the user can enter runtime mode immediately or request a new calibration.

The software state includes:

K_A, K_B	known camera intrinsics
P_A, P_B	saved projection matrices
T_A, T_B	runtime thresholds
min_area	minimum valid foreground area
o	world origin
R_f	floor-alignment rotation
s	metric scale factor

The intrinsic matrices we found are:

$$K_A = \begin{bmatrix} 879.693451 & 0 & 365.717241 \\ 0 & 879.693451 & 241.590062 \\ 0 & 0 & 1 \end{bmatrix}, \quad K_B = \begin{bmatrix} 900.939218 & 0 & 307.065820 \\ 0 & 900.939218 & 218.219419 \\ 0 & 0 & 1 \end{bmatrix}.$$

5.2 Calibration Mode

Calibration mode is interactive by design. The user places the beacon at visible positions, presses a key to log a valid pair, and repeats until enough correspondences exist. The minimum for the fundamental matrix is eight pairs, but the system supports up to 128 correspondences. Once the user requests a solve, the software:

1. estimates the fundamental matrix from the logged pixel correspondences;
2. normalizes the same pixels with K_A and K_B;
3. constructs the essential matrix;
4. recovers relative rotation and translation direction;
5. chooses the physically valid pose by chirality;
6. constructs P_A and P_B;
7. saves them to `stereo_projection.txt`.

The terminal mode is not glamorous, but it is appropriate for the application at this stage. Calibration needs human placement and judgment, while runtime tracking needs a stable loop.

5.3 Runtime Mode

In runtime mode the software repeatedly reads a fresh centroid pair, triangulates the raw 3D point, applies the configured world transform, and prints:

$$x,\ y,\ z.$$

Three additional user actions refine the coordinate frame:

- **Origin.** Pressing `O` stores the current raw point as the origin o, and future points are shifted by $X - o$.
- **Floor.** Pressing `F` at three floor positions logs three points. The software computes the floor normal and builds a rotation that aligns the floor normal with the world $+Z$ direction.
- **Scale.** Pressing `S` at two positions logs a pair of transformed points, asks the user for the real distance, and sets $s = d_{\text{real}}/d_{\text{triangulated}}$.

The final world-coordinate transform is therefore:

$$X_{\text{world}} = sR_f(X_{\text{raw}} - o).$$

6 Stereo Reconstruction Mathematics

This section follows the design document's mathematical model and relates it to the final implementation.

6.1 Epipolar Constraint

Let x_1 and x_2 be homogeneous image coordinates of the same 3D scene point in the two camera images. The fundamental matrix F satisfies:

$$x_2^T F x_1 = 0.$$

This equation is the projective statement that a point in one view must lie on the corresponding epipolar line in the other view.

Because the camera intrinsics are known, the image measurements can be normalized:

$$\tilde{x}_1 = K_1^{-1} x_1, \qquad \tilde{x}_2 = K_2^{-1} x_2.$$

The essential matrix is then:

$$E = K_2^T F K_1,$$

and normalized correspondences satisfy:

$$\tilde{x}_2^T E \tilde{x}_1 = 0.$$

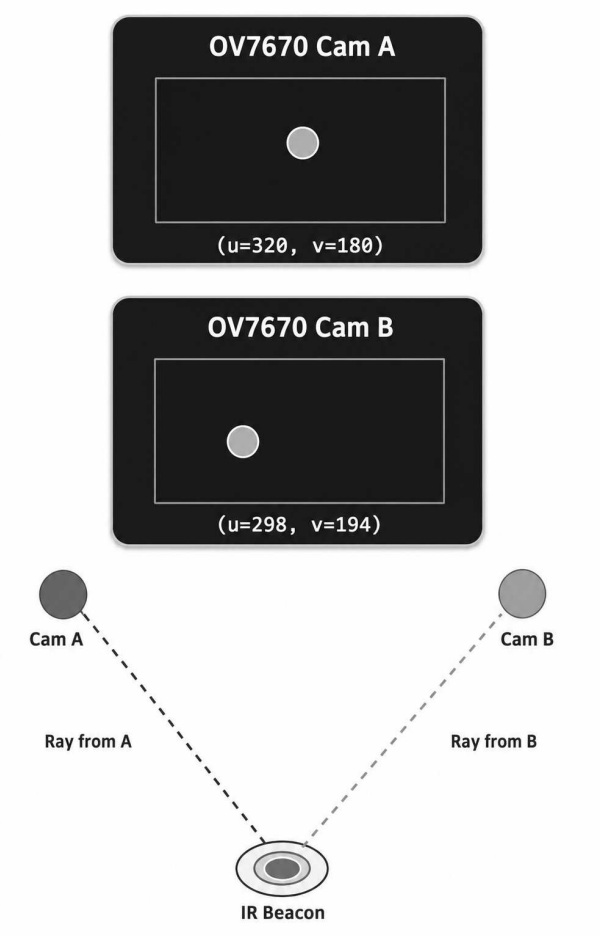

Figure 6: Conceptual view of the runtime stereo measurement. Each OV7670 image produces a beacon centroid in pixel coordinates; after calibration, those two image observations define rays whose intersection gives the beacon's 3D position.

6.2 Eight-Point Least Squares

For a correspondence

$$\tilde{x}_1 = (x_1, y_1, 1)^T, \qquad \tilde{x}_2 = (x_2, y_2, 1)^T,$$

the epipolar constraint expands to one linear equation:

$$\begin{bmatrix} x_2x_1 & x_2y_1 & x_2 & y_2x_1 & y_2y_1 & y_2 & x_1 & y_1 & 1 \end{bmatrix} e = 0,$$

where e is the vectorized form of the 3×3 matrix.

Stacking $N \geq 8$ correspondences produces:

$$Ae = 0.$$

The implementation accumulates A^TA and uses a Jacobi-style eigensolver to recover the eigenvector associated with the smallest eigenvalue. This matches the design document's emphasis on a small symmetric least-squares problem rather than a large external numerical dependency.

6.3 Pose Recovery

The essential matrix encodes camera pose through:

$$E = [t]_\times R,$$

where

$$[t]_\times = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}.$$

The design document decomposes E with an SVD:

$$E = U\Sigma V^T,$$

and uses

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

to produce the two candidate rotations:

$$R_1 = UWV^T, \qquad R_2 = UW^TV^T,$$

with translation direction $t = \pm U(:, 3)$. This yields four candidate poses:

$$(R_1, +t), \quad (R_1, -t), \quad (R_2, +t), \quad (R_2, -t).$$

The implementation evaluates these candidates using a chirality score. A candidate is physically correct when triangulated calibration points lie in front of both cameras. This is a necessary step because the algebra alone cannot distinguish the four poses.

6.4 Projection Matrices and Triangulation

Once the correct R and t are selected, the projection matrices become:

$$P_1 = K_1[I|0], \qquad P_2 = K_2[R|t].$$

For each live centroid pair, triangulation solves:

$$AX = 0,$$

where A is built from the rows of P_1, P_2, and the measured image coordinates. The final homogeneous solution is the right singular vector associated with the smallest singular value, followed by dehomogenization. In the C implementation this appears as a 4×4 normal matrix and another Jacobi smallest-eigenvector solve.

The translation direction recovered from the essential matrix is scale-ambiguous. That is why runtime scale calibration is not optional if metric coordinates are desired. The two-point scale mode resolves this final degree of freedom through a user-supplied real distance.

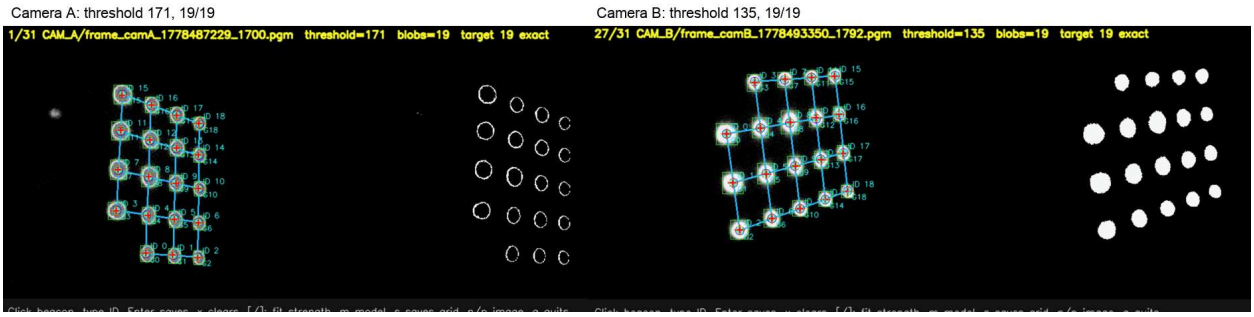

Figure 7: Camera calibration data.

7 Intrinsic Calibration

The debug capture path produced raw calibration images from both cameras. We used a 3D-printed grid with 19 IR beacons (with one missing corner to provide information on orientation) to generate a calibration reference of known geometry for OpenCV intrinsic matrix calibration. We constrained the detected grid point centroids to a homography grid, and fixed both focal length axes to be equal since we knew the lens was circular. We then manually assigned each point the same index across all frames so that OpenCV had information about exact 3D point positions.

8 Experimental Results

The final integrated system achieved the major functional goals of the project:

- both OV7670 cameras can be configured from HPS software through bit-banged SCCB;
- the FPGA receives camera timing and pixel data directly;
- each camera has an independent moment pipeline for area, horizontal moment, and vertical moment;
- the HPS can read fresh paired moments through DONE handshakes;
- full debug frames can be captured and exported for inspection;
- calibration images can be labeled with exact 19/19 beacon assignment in representative frames;
- runtime calibration estimates the stereo geometry and writes reusable projection matrices;
- runtime mode triangulates live centroid pairs and applies origin, floor, and scale transforms.

9 Failure Modes and Engineering Responses

9.1 Threshold Sensitivity

The calibration screenshots show threshold values ranging from 133 to 171. That range is expected: beacon brightness changes with distance, angle, exposure, and ambient IR. The implemented response is to expose thresholds in the CONTROL register and parse `threshold`, `thresholdA`, and `thresholdB` commands in software. A future automatic exposure/threshold loop would improve usability, but the current design at least makes threshold choice explicit and debuggable.

9.2 Invalid Foreground Area

Centroid division is only meaningful if the foreground area is nonzero and large enough to reject noise. The software therefore rejects a pair unless both camera areas exceed `min_area`. This also prevents a bad frame from entering the calibration correspondence set.

9.3 Clock-Domain Hazards

Camera A PCLK, Camera B PCLK, and the HPS-visible register clock are different timing domains. The design uses DONE flags and synchronizer logic so frame completion becomes a stable software-observable event. The C code reinforces the protocol by clearing DONE bits and waiting for the clear before accepting the next frame.

9.4 Calibration Conditioning

The eight-point method becomes unstable if correspondences are nearly degenerate, mislabeled, or too clustered in the image. The manual grid-labeling workflow addresses this by using many spatially distributed points and visually confirming exact assignments. A future implementation could add RANSAC or residual rejection, but the current labeled grid already reduces the most dangerous failure mode: wrong correspondences.

9.5 Scale Ambiguity

Essential-matrix recovery produces translation direction, not metric translation length. The project handles this honestly through scale calibration. The user logs two world points and enters their measured distance, producing a scale factor. Without this step, the coordinates are geometrically consistent but not metric.

9.6 Persistence Gaps

Projection matrices are saved to `stereo_projection.txt`. The origin, floor alignment, and scale are maintained in memory during runtime, but the current source does not persist that full world transform. Persisting all world-frame calibration values would make repeated demos less manual.

10 Future Work

The current system is a complete beacon tracker, but several upgrades would make it more robust and easier to apply at scale:

- automatic threshold selection based on area feedback and saturation margin;
- persistence of origin, floor rotation, and scale alongside projection matrices;
- residual reporting after calibration so the user can see reprojection error;
- outlier rejection for mislabeled or weak calibration correspondences;
- hardware-side multi-blob extraction for tracking more than one beacon;
- rectification or epipolar search visualization for easier calibration debugging;
- a small GUI for calibration state, live centroids, and 3D trajectory display.

The most valuable next step is not adding more math or so we both choose to believe. It's exposing the calibration quality to the user. A report of per-point reprojection error would turn calibration from a black box into an engineering measurement.

11 Conclusion

This project works because it treats stereo tracking as an embedded-systems problem, not only as a computer-vision problem. The FPGA handles the work that is naturally tied to the pixel stream: timing, thresholding, accumulation, frame completion, and debug capture. The HPS handles the work that needs state, user input, and floating-point math: configuration, calibration, pose recovery, triangulation, coordinate-frame alignment, and terminal interaction. The final system can observe two OV7670 streams, reduce each frame to the moments needed for centroid recovery, calibrate the stereo pair from logged correspondences, recover projection matrices, triangulate live beacon positions, and map them into a usable world frame. It also leaves behind the debugging hooks that made the project possible: raw frame capture, labeled calibration evidence, status registers, threshold controls, and source-level numerical routines.

Acknowledgments

We thank Professor Stephen Edwards and the CSEE W4840 teaching staff for guidance on embedded hardware/software integration, FPGA timing, and system debugging.

References

[1] K. Zhang and L. Alshorafa, *CSEE4840 Design Document: Uncalibrated Stereo Motion Tracking*, Columbia University, April 2026.

[2] OmniVision Technologies, *OV7670/OV7171 CMOS VGA CameraChip Sensor Preliminary Datasheet*, Version 1.4, August 21, 2006.

[3] Project source repository, *Embedded-Systems-W4840/project*, May 2026.

[4] R. I. Hartley, "In Defence of the 8-Point Algorithm," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1997.

[5] OpenCV, "Camera Calibration and 3D Reconstruction," OpenCV Documentation. Available: https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html

[6] OpenCV, "Triangulation," OpenCV SFM Documentation. Available: https://docs.opencv.org/3.4/d0/dbd/group__triangulation.html

[7] Stack Overflow, "OpenCV: Essential Matrix Decomposition." Available: https://stackoverflow.com/questions/21232734/opencv-essential-matrix-decomposition

[8] Stack Overflow, "OpenCV recoverPose from essential matrix E." Available: https://stackoverflow.com/questions/65771642/opencv-recoverpose-from-essential-matrix-e

[9] Stack Overflow, "Homogeneous 4D vector normalization after OpenCVs triangulatePoints()." Available: https://stackoverflow.com/questions/31344866/homogeneous-4d-vector-normalization-after-opencvs-triangulatepoints

A Software Source Appendix

A.1 main.c

```c
#define _DEFAULT_SOURCE

#include "camera_capture.h"
#include "camera_config.h"
#include "essential.h"
#include "fundamental.h"
#include "projection.h"
#include "stereo.h"

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>
#include <termios.h>
#include <unistd.h>

#define CALIBRATION_FILE "stereo_projection.txt"
#define MAX_CORRESPONDENCES 128
#define MIN_FUNDAMENTAL_POINTS 8
#define LINE_LEN 256
#define MAX_ARGS 24
#define DEFAULT_MIN_AREA 10u
#define DEFAULT_THRESHOLD 0x0au

static const double K_A[9] = {
    879.693451, 0.0,        365.717241,
    0.0,        879.693451, 241.590062,
    0.0,        0.0,        1.0,
};

static const double K_B[9] = {
    900.939218, 0.0,        307.065820,
    0.0,        900.939218, 218.219419,
    0.0,        0.0,        1.0,
};

typedef struct {
    double p_a[12];
    double p_b[12];
    int valid;
} calibration_t;

typedef struct {
    unsigned threshold_a;
    unsigned threshold_b;
    unsigned min_area;
    calibration_t calibration;
    double origin[3];
    int origin_set;
    double floor_rot[9];
    int floor_set;
    double floor_points[3][3];
    unsigned floor_count;
    double scale_points[2][3];
    unsigned scale_count;
    double scale;
} app_state_t;

static void app_state_init(app_state_t *state)
{
    memset(state, 0, sizeof(*state));
    state->threshold_a = DEFAULT_THRESHOLD;
```

```
    state->threshold_b = DEFAULT_THRESHOLD;
    state->min_area = DEFAULT_MIN_AREA;
    projection_identity3(state->floor_rot);
    state->scale = 1.0;
}

static int save_calibration(const calibration_t *calibration)
{
    FILE *out = fopen(CALIBRATION_FILE, "w");

    if (!out) {
        perror("fopen calibration");
        return -1;
    }

    fprintf(out, "STEREO_CALIBRATION_V1\n");
    fprintf(out, "P_A");
    for (int i = 0; i < 12; i++) {
        fprintf(out, " %.12g", calibration->p_a[i]);
    }
    fprintf(out, "\nP_B");
    for (int i = 0; i < 12; i++) {
        fprintf(out, " %.12g", calibration->p_b[i]);
    }
    fprintf(out, "\n");
    fclose(out);
    return 0;
}

static int load_calibration(calibration_t *calibration)
{
    FILE *in = fopen(CALIBRATION_FILE, "r");
    char tag[64];
    char name[16];

    memset(calibration, 0, sizeof(*calibration));
    if (!in) {
        return -1;
    }
    if (fscanf(in, "%63s", tag) != 1 ||
        strcmp(tag, "STEREO_CALIBRATION_V1") != 0) {
        fclose(in);
        return -1;
    }
    if (fscanf(in, "%15s", name) != 1 || strcmp(name, "P_A") != 0) {
        fclose(in);
        return -1;
    }
    for (int i = 0; i < 12; i++) {
        if (fscanf(in, "%lf", &calibration->p_a[i]) != 1) {
            fclose(in);
            return -1;
        }
    }
    if (fscanf(in, "%15s", name) != 1 || strcmp(name, "P_B") != 0) {
        fclose(in);
        return -1;
    }
    for (int i = 0; i < 12; i++) {
        if (fscanf(in, "%lf", &calibration->p_b[i]) != 1) {
            fclose(in);
            return -1;
        }
    }
    fclose(in);
    calibration->valid = 1;
    return 0;
}
```

```
132
133 static int read_valid_pair(const app_state_t *state, point2_t *a, point2_t *b)
134 {
135     camera_moments_t moments_a;
136     camera_moments_t moments_b;
137     int valid = 0;
138
139     if (camera_read_moment_pair(&moments_a, &moments_b, state->min_area,
140                                 &valid) < 0) {
141         return -1;
142     }
143     if (!valid) {
144         fprintf(stderr,
145                 "invalid␣beacon␣pair:␣areaA=%u␣areaB=%u␣min=%u\n",
146                 moments_a.area, moments_b.area, state->min_area);
147         return 1;
148     }
149
150     a->x = moments_a.cx;
151     a->y = moments_a.cy;
152     b->x = moments_b.cx;
153     b->y = moments_b.cy;
154     return 0;
155 }
156
157 static int triangulate_raw(const calibration_t *calibration,
158                            const point2_t *a, const point2_t *b,
159                            double x[3])
160 {
161     if (!calibration->valid) {
162         return -1;
163     }
164     return projection_triangulate(calibration->p_a, calibration->p_b,
165                                   a, b, x);
166 }
167
168 static int current_raw_point(const app_state_t *state, double raw[3])
169 {
170     point2_t a;
171     point2_t b;
172     int ret = read_valid_pair(state, &a, &b);
173
174     if (ret != 0) {
175         return ret;
176     }
177     if (triangulate_raw(&state->calibration, &a, &b, raw) < 0) {
178         fprintf(stderr, "triangulation␣failed\n");
179         return -1;
180     }
181     return 0;
182 }
183
184 static int configure_defaults(app_state_t *state, camera_select_t cameras)
185 {
186     camera_settings_t settings_a;
187     camera_settings_t settings_b;
188     int failed = 0;
189
190     camera_settings_default_for(CAMERA_A, &settings_a);
191     camera_settings_default_for(CAMERA_B, &settings_b);
192
193     if ((cameras & CAMERA_A) &&
194         camera_configure(CAMERA_A, &settings_a) < 0) {
195         failed = 1;
196     }
197     if ((cameras & CAMERA_B) &&
198         camera_configure(CAMERA_B, &settings_b) < 0) {
199         failed = 1;
```

```
    }
    if (camera_set_thresholds(state->threshold_a, state->threshold_b) < 0) {
        failed = 1;
    }

    return failed ? -1 : 0;
}

static int solve_calibration(app_state_t *state, const point2_t *points_a,
                             const point2_t *points_b, size_t count)
{
    point2_t norm_a[MAX_CORRESPONDENCES];
    point2_t norm_b[MAX_CORRESPONDENCES];
    camera_pose_t pose;
    double f[9];
    double e[9];

    if (count < MIN_FUNDAMENTAL_POINTS) {
        fprintf(stderr, "need at least %u correspondences\n",
                MIN_FUNDAMENTAL_POINTS);
        return -1;
    }

    if (fundamental_estimate(points_a, points_b, count, f) < 0) {
        fprintf(stderr, "fundamental matrix estimation failed\n");
        return -1;
    }

    for (size_t i = 0; i < count; i++) {
        norm_a[i] = projection_normalize_pixel(K_A, points_a[i]);
        norm_b[i] = projection_normalize_pixel(K_B, points_b[i]);
    }

    essential_from_fundamental(K_A, K_B, f, e);
    if (essential_recover_pose(e, norm_a, norm_b, count, &pose) < 0) {
        fprintf(stderr, "essential pose recovery failed\n");
        return -1;
    }

    projection_make_left_camera(K_A, state->calibration.p_a);
    essential_projection_matrix(K_B, &pose, state->calibration.p_b);
    state->calibration.valid = 1;

    if (save_calibration(&state->calibration) < 0) {
        return -1;
    }

    printf("wrote projection matrices to %s\n", CALIBRATION_FILE);
    return 0;
}

static int read_key_timeout_ms(int timeout_ms)
{
    fd_set fds;
    struct timeval tv;
    unsigned char c;
    int ret;

    FD_ZERO(&fds);
    FD_SET(STDIN_FILENO, &fds);
    tv.tv_sec = timeout_ms / 1000;
    tv.tv_usec = (timeout_ms % 1000) * 1000;

    ret = select(STDIN_FILENO + 1, &fds, NULL, NULL, &tv);
    if (ret <= 0) {
        return -1;
    }
    if (read(STDIN_FILENO, &c, 1) != 1) {
```

```
268         return -1;
269     }
270     return c;
271 }
272 
273 static int set_raw_mode(const struct termios *old_term);
274 
275 static int enable_raw(struct termios *old_term)
276 {
277     if (tcgetattr(STDIN_FILENO, old_term) < 0) {
278         perror("tcgetattr");
279         return -1;
280     }
281     setvbuf(stdin, NULL, _IONBF, 0);
282     return set_raw_mode(old_term);
283 }
284 
285 static int set_raw_mode(const struct termios *old_term)
286 {
287     struct termios raw = *old_term;
288 
289     raw.c_lflag &= (tcflag_t)~(ICANON | ECHO);
290     raw.c_cc[VMIN] = 0;
291     raw.c_cc[VTIME] = 0;
292     if (tcsetattr(STDIN_FILENO, TCSANOW, &raw) < 0) {
293         perror("tcsetattr");
294         return -1;
295     }
296     return 0;
297 }
298 
299 static void restore_terminal(const struct termios *old_term)
300 {
301     tcsetattr(STDIN_FILENO, TCSANOW, old_term);
302 }
303 
304 static int prompt_double(const char *prompt, double *value)
305 {
306     char line[LINE_LEN];
307     char *end = NULL;
308 
309     printf("%s", prompt);
310     fflush(stdout);
311     if (!fgets(line, sizeof(line), stdin)) {
312         return -1;
313     }
314     *value = strtod(line, &end);
315     while (end && isspace((unsigned char)*end)) {
316         end++;
317     }
318     return end && *end == '\0' ? 0 : -1;
319 }
320 
321 static int runtime_mode(app_state_t *state);
322 
323 static int calibration_mode(app_state_t *state)
324 {
325     point2_t points_a[MAX_CORRESPONDENCES];
326     point2_t points_b[MAX_CORRESPONDENCES];
327     size_t count = 0;
328     struct termios old_term;
329 
330     printf("\nCalibration mode\n");
331     printf("Q log valid beacon pair, C solve after %u pairs, G config defaults, X exit\n",
332            MIN_FUNDAMENTAL_POINTS);
333 
334     if (enable_raw(&old_term) < 0) {
335         return -1;
```

```
336     }
337
338     for (;;) {
339         int key = read_key_timeout_ms(100);
340
341         if (key < 0) {
342             continue;
343         }
344         key = tolower(key);
345
346         if (key == 'x') {
347             restore_terminal(&old_term);
348             printf("\nleaving␣calibration␣mode\n");
349             return 0;
350         }
351
352         if (key == 'g') {
353             restore_terminal(&old_term);
354             configure_defaults(state, CAMERA_BOTH);
355             if (enable_raw(&old_term) < 0) {
356                 return -1;
357             }
358             continue;
359         }
360
361         if (key == 'q') {
362             point2_t a;
363             point2_t b;
364             int ret = read_valid_pair(state, &a, &b);
365
366             if (ret == 0 && count < MAX_CORRESPONDENCES) {
367                 points_a[count] = a;
368                 points_b[count] = b;
369                 count++;
370                 printf("\n%zu:␣A=(%.2f,␣%.2f)␣B=(%.2f,␣%.2f)\n",
371                        count, a.x, a.y, b.x, b.y);
372                 if (count >= MIN_FUNDAMENTAL_POINTS) {
373                     printf("press␣C␣to␣solve␣projection␣matrices\n");
374                 }
375             }
376             continue;
377         }
378
379         if (key == 'c') {
380             int ret;
381
382             restore_terminal(&old_term);
383             ret = solve_calibration(state, points_a, points_b, count);
384             if (ret == 0) {
385                 runtime_mode(state);
386                 return 0;
387             }
388             if (enable_raw(&old_term) < 0) {
389                 return -1;
390             }
391         }
392     }
393 }
394
395 static void compute_floor_rotation(app_state_t *state)
396 {
397     if (projection_floor_rotation_from_points(state->floor_points[0],
398                                               state->floor_points[1],
399                                               state->floor_points[2],
400                                               state->floor_rot) < 0) {
401         fprintf(stderr, "floor␣points␣are␣degenerate\n");
402         return;
403     }
```

```
    state->floor_set = 1;
    printf("\nfloor␣normal␣aligned␣to␣+Z\n");
}

static void log_floor_point(app_state_t *state, const double raw[3])
{
    double shifted[3];

    if (state->floor_count >= 3) {
        printf("\nfloor␣already␣calibrated;␣restarting␣floor␣collection\n");
        state->floor_count = 0;
        state->floor_set = 0;
        projection_identity3(state->floor_rot);
    }

    shifted[0] = raw[0] - (state->origin_set ? state->origin[0] : 0.0);
    shifted[1] = raw[1] - (state->origin_set ? state->origin[1] : 0.0);
    shifted[2] = raw[2] - (state->origin_set ? state->origin[2] : 0.0);
    memcpy(state->floor_points[state->floor_count], shifted, sizeof(shifted));
    state->floor_count++;
    printf("\nfloor␣point␣%u/3␣logged\n", state->floor_count);

    if (state->floor_count == 3) {
        compute_floor_rotation(state);
    }
}

static int log_scale_point(app_state_t *state, const double raw[3],
                           const struct termios *old_term)
{
    double p[3];

    projection_apply_transform_unscaled(raw, state->origin, state->origin_set,
                                        state->floor_rot, p);
    if (state->scale_count >= 2) {
        printf("\nscale␣already␣calibrated;␣restarting␣scale␣collection\n");
        state->scale_count = 0;
        state->scale = 1.0;
    }

    memcpy(state->scale_points[state->scale_count], p, sizeof(p));
    state->scale_count++;
    printf("\nscale␣point␣%u/2␣logged\n", state->scale_count);

    if (state->scale_count == 2) {
        double dist;
        double measured;

        dist = projection_distance3(state->scale_points[0],
                                    state->scale_points[1]);
        if (dist < 1.0e-12) {
            fprintf(stderr, "scale␣points␣are␣identical\n");
            state->scale_count = 0;
            return 0;
        }

        restore_terminal(old_term);
        if (prompt_double("real␣distance␣between␣scale␣points:␣", &measured) < 0 ||
            measured <= 0.0) {
            fprintf(stderr, "invalid␣scale␣distance\n");
            if (set_raw_mode(old_term) < 0) {
                return -1;
            }
            return 0;
        }
        state->scale = measured / dist;
        printf("scale␣set␣to␣%.6f\n", state->scale);
        if (set_raw_mode(old_term) < 0) {
```

```
472             return -1;
473         }
474     }
475
476     return 0;
477 }
478
479 static int runtime_mode(app_state_t *state)
480 {
481     struct termios old_term;
482     double raw[3] = {0.0, 0.0, 0.0};
483     double xyz[3] = {0.0, 0.0, 0.0};
484     int have_point = 0;
485
486     if (!state->calibration.valid) {
487         fprintf(stderr, "projection calibration is not valid\n");
488         return -1;
489     }
490
491     printf("\nRuntime triangulation mode\n");
492     printf("O set origin, F log floor point, S log scale point, X exit\n");
493
494     if (enable_raw(&old_term) < 0) {
495         return -1;
496     }
497
498     for (;;) {
499         int valid_read = current_raw_point(state, raw);
500         int key;
501
502         if (valid_read == 0) {
503             projection_apply_transform(raw, state->origin, state->origin_set,
504                                        state->floor_rot, state->scale, xyz);
505             have_point = 1;
506             printf("\rxyz = %.4f, %.4f, %.4f        ",
507                    xyz[0], xyz[1], xyz[2]);
508             fflush(stdout);
509         }
510
511         key = read_key_timeout_ms(1);
512         if (key < 0) {
513             continue;
514         }
515         key = tolower(key);
516
517         if (key == 'x') {
518             restore_terminal(&old_term);
519             printf("\nleaving runtime mode\n");
520             return 0;
521         }
522
523         if (!have_point) {
524             printf("\rno valid point available for command\n");
525             continue;
526         }
527
528         if (key == 'o') {
529             memcpy(state->origin, raw, sizeof(state->origin));
530             state->origin_set = 1;
531             printf("\rorigin set\n");
532         } else if (key == 'f') {
533             log_floor_point(state, raw);
534         } else if (key == 's') {
535             if (log_scale_point(state, raw, &old_term) < 0) {
536                 restore_terminal(&old_term);
537                 return -1;
538             }
539         }
```

```
    }
}

static int split_line(char *line, char **argv)
{
    int argc = 0;
    char *tok = strtok(line, "␣\t\r\n");

    while (tok && argc < MAX_ARGS) {
        argv[argc++] = tok;
        tok = strtok(NULL, "␣\t\r\n");
    }
    return argc;
}

static int parse_config_key(app_state_t *state, const char *arg,
                            camera_settings_t *settings_a,
                            camera_settings_t *settings_b,
                            camera_select_t cameras)
{
    const char *value = strchr(arg, '=');

    if (!value) {
        if ((cameras & CAMERA_A) &&
            camera_config_parse_setting(arg, settings_a) < 0) {
            return -1;
        }
        if ((cameras & CAMERA_B) &&
            camera_config_parse_setting(arg, settings_b) < 0) {
            return -1;
        }
        return 0;
    }

    if (strncmp(arg, "threshold=", 10) == 0 ||
        strncmp(arg, "th=", 3) == 0) {
        unsigned th;
        const char *v = value + 1;
        if (parse_unsigned_arg(v, CAMERA_REG8_MAX, &th) < 0) {
            return -1;
        }
        state->threshold_a = th;
        state->threshold_b = th;
        return 0;
    }
    if (strncmp(arg, "thresholdA=", 11) == 0 ||
        strncmp(arg, "thA=", 4) == 0) {
        if (parse_unsigned_arg(value + 1, CAMERA_REG8_MAX,
                               &state->threshold_a) < 0) {
            return -1;
        }
        return 0;
    }
    if (strncmp(arg, "thresholdB=", 11) == 0 ||
        strncmp(arg, "thB=", 4) == 0) {
        if (parse_unsigned_arg(value + 1, CAMERA_REG8_MAX,
                               &state->threshold_b) < 0) {
            return -1;
        }
        return 0;
    }
    if (strncmp(arg, "minarea=", 8) == 0 ||
        strncmp(arg, "min_area=", 9) == 0) {
        if (parse_unsigned_arg(value + 1, FRAME_PIXELS, &state->min_area) < 0) {
            return -1;
        }
        return 0;
    }
```

```
608
609     if ((cameras & CAMERA_A) &&
610         camera_config_parse_setting(arg, settings_a) < 0) {
611         return -1;
612     }
613     if ((cameras & CAMERA_B) &&
614         camera_config_parse_setting(arg, settings_b) < 0) {
615         return -1;
616     }
617     return 0;
618 }
619
620 static int command_config(app_state_t *state, int argc, char **argv)
621 {
622     camera_select_t cameras = CAMERA_BOTH;
623     camera_settings_t settings_a;
624     camera_settings_t settings_b;
625     int argi = 0;
626     int failed = 0;
627
628     camera_settings_default_for(CAMERA_A, &settings_a);
629     camera_settings_default_for(CAMERA_B, &settings_b);
630
631     if (argc > 0 && camera_parse_select(argv[0], &cameras) == 0) {
632         argi = 1;
633     }
634
635     while (argi < argc) {
636         if (parse_config_key(state, argv[argi], &settings_a, &settings_b,
637                              cameras) < 0) {
638             fprintf(stderr, "invalid␣config␣option:␣%s\n", argv[argi]);
639             return -1;
640         }
641         argi++;
642     }
643
644     if ((cameras & CAMERA_A) && camera_configure(CAMERA_A, &settings_a) < 0) {
645         failed = 1;
646     }
647     if ((cameras & CAMERA_B) && camera_configure(CAMERA_B, &settings_b) < 0) {
648         failed = 1;
649     }
650     if (camera_set_thresholds(state->threshold_a, state->threshold_b) < 0) {
651         failed = 1;
652     }
653
654     printf("min␣valid␣area=%u\n", state->min_area);
655     return failed ? -1 : 0;
656 }
657
658 static int command_capture(int argc, char **argv)
659 {
660     camera_select_t camera = CAMERA_A;
661     int argi = 0;
662
663     if (argi < argc && camera_parse_single(argv[argi], &camera) == 0) {
664         argi++;
665     }
666     if (argi != argc) {
667         fprintf(stderr, "usage:␣capture␣[A|B]\n");
668         return -1;
669     }
670     return camera_capture_serial(camera, stdout);
671 }
672
673 static void print_help(void)
674 {
675     printf("commands:\n");
```

```
676     printf("␣␣config␣[A|B|both]␣[gain=...]␣[agc=...]␣[aec=...]␣[exposure=...]\n");
677     printf("␣␣␣␣␣␣␣␣␣[threshold=...]␣[thresholdA=...]␣[thresholdB=...]␣[minarea=...]\n");
678     printf("␣␣capture␣[A|B]␣␣␣␣␣␣␣emit␣one␣serial␣frame␣block\n");
679     printf("␣␣debug␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣stream␣area/u/v/centroid␣debug␣output\n");
680     printf("␣␣calibrate␣␣␣␣␣␣␣␣␣␣enter␣correspondence␣logging␣mode\n");
681     printf("␣␣runtime␣␣␣␣␣␣␣␣␣␣␣␣enter␣triangulation␣mode\n");
682     printf("␣␣status␣␣␣␣␣␣␣␣␣␣␣␣␣show␣hardware␣status\n");
683     printf("␣␣quit\n");
684 }
685
686 static int prompt_initial_mode(app_state_t *state)
687 {
688     char line[LINE_LEN];
689
690     if (!state->calibration.valid) {
691         printf("No␣valid␣%s;␣calibration␣is␣required.\n", CALIBRATION_FILE);
692         return calibration_mode(state);
693     }
694
695     printf("Loaded␣%s.\n", CALIBRATION_FILE);
696     printf("Enter␣runtime␣triangulation␣or␣recalibrate?␣[r/c]:␣");
697     fflush(stdout);
698     if (!fgets(line, sizeof(line), stdin)) {
699         return -1;
700     }
701     if (tolower((unsigned char)line[0]) == 'c') {
702         return calibration_mode(state);
703     }
704     return runtime_mode(state);
705 }
706
707 int main(void)
708 {
709     app_state_t state;
710     char line[LINE_LEN];
711
712     app_state_init(&state);
713     if (load_calibration(&state.calibration) == 0) {
714         state.calibration.valid = 1;
715     }
716
717     printf("stereo␣terminal\n");
718     printf("intrinsics␣loaded␣for␣camera␣A␣and␣B\n");
719     printf("applying␣startup␣camera␣config\n");
720     if (configure_defaults(&state, CAMERA_BOTH) < 0) {
721         printf("startup␣config␣reported␣an␣error;␣continuing␣anyway\n");
722     }
723     if (prompt_initial_mode(&state) < 0) {
724         printf("initial␣mode␣ended␣with␣an␣error;␣entering␣command␣prompt\n");
725     }
726
727     print_help();
728     while (1) {
729         char *argv[MAX_ARGS];
730         int argc;
731
732         printf("stereo>␣");
733         fflush(stdout);
734         if (!fgets(line, sizeof(line), stdin)) {
735             break;
736         }
737
738         argc = split_line(line, argv);
739         if (argc == 0) {
740             continue;
741         }
742
743         if (strcmp(argv[0], "quit") == 0 || strcmp(argv[0], "exit") == 0) {
```

```
            break;
        } else if (strcmp(argv[0], "help") == 0) {
            print_help();
        } else if (strcmp(argv[0], "config") == 0 ||
                   strcmp(argv[0], "configure") == 0) {
            command_config(&state, argc - 1, argv + 1);
        } else if (strcmp(argv[0], "capture") == 0) {
            command_capture(argc - 1, argv + 1);
        } else if (strcmp(argv[0], "debug") == 0) {
            camera_debug_stream(30);
        } else if (strcmp(argv[0], "calibrate") == 0) {
            calibration_mode(&state);
        } else if (strcmp(argv[0], "runtime") == 0) {
            runtime_mode(&state);
        } else if (strcmp(argv[0], "status") == 0) {
            camera_print_status();
        } else {
            fprintf(stderr, "unknown command: %s\n", argv[0]);
            print_help();
        }
    }

    return 0;
}
```

A.2 camera_capture.c

```
#define _DEFAULT_SOURCE

#include "camera_capture.h"

#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <time.h>
#include <unistd.h>

#define REG32(byte_offset) ((byte_offset) / 4u)

#define IMG_AREA_A REG32(0x00u)
#define IMG_U_A REG32(0x04u)
#define IMG_V_A REG32(0x08u)
#define IMG_AREA_B REG32(0x0cu)
#define IMG_U_B REG32(0x10u)
#define IMG_V_B REG32(0x14u)
#define IMG_DONE REG32(0x18u)
#define IMG_CONTROL REG32(0x1cu)
#define IMG_FB_INDEX REG32(0x20u)
#define IMG_FB_DATA REG32(0x24u)

#define DONE_A (1u << 0)
#define DONE_B (1u << 1)
#define DONE_FB (1u << 2)
#define DONE_MOMENTS (DONE_A | DONE_B)

#define STORE_MASK 0x3u
#define STORE_NONE 0u
#define STORE_CAMERA_A 1u
#define STORE_CAMERA_B 2u

#define THRESH_A_SHIFT 8
#define THRESH_B_SHIFT 16
#define THRESH_MASK 0xffu
```

```
40
41  #define DONE_TIMEOUT_MS 5000
42  #define CLEAR_TIMEOUT_MS 100
43  #define CLEAR_TIMEOUT_POLLS 10000000u
44  #define DONE_TIMEOUT_POLLS 100000000u
45  #define NS_PER_SEC 1000000000.0
46  #define NAME_LEN 256
47  #define PGM_HEADER_LEN 32
48
49  static const char b64[] =
50      "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
51
52  static int map_regs(volatile uint32_t **regs_out, int *fd_out)
53  {
54      int fd = open("/dev/mem", O_RDWR | O_SYNC);
55      void *map;
56
57      if (fd < 0) {
58          perror("open␣/dev/mem");
59          return -1;
60      }
61
62      map = mmap(NULL, DE1_LW_BRIDGE_SPAN, PROT_READ | PROT_WRITE, MAP_SHARED,
63                 fd, DE1_LW_BRIDGE_BASE);
64      if (map == MAP_FAILED) {
65          perror("mmap");
66          close(fd);
67          return -1;
68      }
69
70      *regs_out = (volatile uint32_t *)map;
71      *fd_out = fd;
72      return 0;
73  }
74
75  static void unmap_regs(volatile uint32_t *regs, int fd)
76  {
77      munmap((void *)regs, DE1_LW_BRIDGE_SPAN);
78      close(fd);
79  }
80
81  static unsigned store_select(camera_select_t camera)
82  {
83      return camera == CAMERA_B ? STORE_CAMERA_B : STORE_CAMERA_A;
84  }
85
86  static void clear_done(volatile uint32_t *regs, uint32_t bits)
87  {
88      regs[IMG_DONE] = ~bits;
89  }
90
91  static int wait_clear(volatile uint32_t *regs, uint32_t bits, const char *name)
92  {
93      for (unsigned polls = 0; (regs[IMG_DONE] & bits) != 0; polls++) {
94          if (polls >= CLEAR_TIMEOUT_POLLS) {
95              fprintf(stderr, "%s:␣timeout␣clearing␣DONE;␣DONE=0x%08x\n",
96                      name, regs[IMG_DONE]);
97              return -1;
98          }
99      }
100     return 0;
101 }
102
103 static double elapsed_seconds(const struct timespec *start,
104                               const struct timespec *end)
105 {
106     return (double)(end->tv_sec - start->tv_sec) +
107            (double)(end->tv_nsec - start->tv_nsec) / NS_PER_SEC;
```

```
}

static double centroid_coord(uint32_t moment, uint32_t area)
{
    if (area == 0) {
        return -1.0;
    }
    return (double)moment / (double)area;
}

static void finish_moments(camera_moments_t *moments)
{
    moments->cx = centroid_coord(moments->u, moments->area);
    moments->cy = centroid_coord(moments->v, moments->area);
}

static void read_moments_a(volatile uint32_t *regs, camera_moments_t *moments)
{
    moments->area = regs[IMG_AREA_A];
    moments->u = regs[IMG_U_A];
    moments->v = regs[IMG_V_A];
    finish_moments(moments);
}

static void read_moments_b(volatile uint32_t *regs, camera_moments_t *moments)
{
    moments->area = regs[IMG_AREA_B];
    moments->u = regs[IMG_U_B];
    moments->v = regs[IMG_V_B];
    finish_moments(moments);
}

static int discard_first_results(volatile uint32_t *regs)
{
    int discarded_a = 0;
    int discarded_b = 0;
    camera_moments_t unused;

    for (unsigned polls = 0; !discarded_a || !discarded_b; polls++) {
        uint32_t done = regs[IMG_DONE];

        if (!discarded_a && (done & DONE_A)) {
            read_moments_a(regs, &unused);
            clear_done(regs, DONE_A);
            if (wait_clear(regs, DONE_A, "initial camera A moments") < 0) {
                return -1;
            }
            discarded_a = 1;
        }

        if (!discarded_b && (done & DONE_B)) {
            read_moments_b(regs, &unused);
            clear_done(regs, DONE_B);
            if (wait_clear(regs, DONE_B, "initial camera B moments") < 0) {
                return -1;
            }
            discarded_b = 1;
        }

        if (polls >= DONE_TIMEOUT_POLLS) {
            fprintf(stderr,
                    "initial moments: timeout waiting for first results; DONE=0x%08x\n",
                    regs[IMG_DONE]);
            return -1;
        }
    }

    return 0;
```

```
}

static void set_threshold_regs(volatile uint32_t *regs, unsigned threshold_a,
                               unsigned threshold_b)
{
    regs[IMG_CONTROL] =
        (regs[IMG_CONTROL] &
         ~(STORE_MASK | (THRESH_MASK << THRESH_A_SHIFT) |
           (THRESH_MASK << THRESH_B_SHIFT))) |
        STORE_NONE | (threshold_a << THRESH_A_SHIFT) |
        (threshold_b << THRESH_B_SHIFT);
}

int camera_set_thresholds(unsigned threshold_a, unsigned threshold_b)
{
    volatile uint32_t *regs;
    int fd;

    if (threshold_a > THRESH_MASK || threshold_b > THRESH_MASK) {
        fprintf(stderr, "thresholds must be 0..255\n");
        return -1;
    }
    if (map_regs(&regs, &fd) < 0) {
        return -1;
    }

    set_threshold_regs(regs, threshold_a, threshold_b);
    printf("frame store off, thresholdA=%u thresholdB=%u CONTROL=0x%08x\n",
           threshold_a, threshold_b, regs[IMG_CONTROL]);

    clear_done(regs, DONE_MOMENTS);
    if (wait_clear(regs, DONE_MOMENTS, "initial moments") < 0 ||
        discard_first_results(regs) < 0) {
        unmap_regs(regs, fd);
        return -1;
    }

    unmap_regs(regs, fd);
    return 0;
}

static int read_moment_pair_regs(volatile uint32_t *regs,
                                 camera_moments_t *moments_a,
                                 camera_moments_t *moments_b,
                                 unsigned min_area, int *valid)
{
    int fresh_a = 0;
    int fresh_b = 0;

    for (unsigned polls = 0; !fresh_a || !fresh_b; polls++) {
        uint32_t done = regs[IMG_DONE];

        if (done & DONE_A) {
            read_moments_a(regs, moments_a);
            clear_done(regs, DONE_A);
            if (wait_clear(regs, DONE_A, "camera A moments") < 0) {
                return -1;
            }
            fresh_a = 1;
        }

        if (done & DONE_B) {
            read_moments_b(regs, moments_b);
            clear_done(regs, DONE_B);
            if (wait_clear(regs, DONE_B, "camera B moments") < 0) {
                return -1;
            }
            fresh_b = 1;
```

```
        }

        if (polls >= DONE_TIMEOUT_POLLS) {
            fprintf(stderr,
                    "moments: timeout waiting for pair; DONE=0x%08x\n",
                    regs[IMG_DONE]);
            return -1;
        }
    }

    *valid = moments_a->area >= min_area && moments_b->area >= min_area;
    return 0;
}

int camera_read_moment_pair(camera_moments_t *moments_a,
                            camera_moments_t *moments_b,
                            unsigned min_area, int *valid)
{
    volatile uint32_t *regs;
    int fd;
    int ret;

    if (map_regs(&regs, &fd) < 0) {
        return -1;
    }
    ret = read_moment_pair_regs(regs, moments_a, moments_b, min_area, valid);
    unmap_regs(regs, fd);
    return ret;
}

static int capture_frame_regs(volatile uint32_t *regs, camera_select_t camera,
                              unsigned char *pixels)
{
    uint32_t control = regs[IMG_CONTROL];
    unsigned store = store_select(camera);

    if ((control & STORE_MASK) != STORE_NONE && (regs[IMG_DONE] & DONE_FB) == 0) {
        printf("waiting for previous framebuffer capture to finish\n");
        for (unsigned ms = 0; (regs[IMG_DONE] & DONE_FB) == 0; ms++) {
            if (ms >= DONE_TIMEOUT_MS) {
                fprintf(stderr, "timeout waiting for prior frame DONE; DONE=0x%08x\n",
                        regs[IMG_DONE]);
                return -1;
            }
            usleep(1000);
        }
    }

    regs[IMG_CONTROL] = (regs[IMG_CONTROL] & ~STORE_MASK) | store;

    printf("camera %c selected; CONTROL=0x%08x DONE=0x%08x\n",
           camera_letter(camera), regs[IMG_CONTROL], regs[IMG_DONE]);
    clear_done(regs, DONE_FB);

    for (unsigned ms = 0; (regs[IMG_DONE] & DONE_FB) != 0; ms++) {
        if (ms >= CLEAR_TIMEOUT_MS) {
            fprintf(stderr, "timeout clearing frame DONE; DONE=0x%08x\n",
                    regs[IMG_DONE]);
            return -1;
        }
        usleep(1000);
    }

    printf("capture armed; waiting for DONE\n");

    for (unsigned ms = 0; (regs[IMG_DONE] & DONE_FB) == 0; ms++) {
        if (ms >= DONE_TIMEOUT_MS) {
            fprintf(stderr, "timeout waiting for frame DONE; DONE=0x%08x\n",
```

```
                    regs[IMG_DONE]);
            return -1;
        }
        usleep(1000);
    }

    regs[IMG_CONTROL] &= ~STORE_MASK;

    for (uint32_t i = 0; i < FRAME_WORDS; i++) {
        uint32_t word;

        regs[IMG_FB_INDEX] = i;
        word = regs[IMG_FB_DATA];

        pixels[4 * i + 0] = (unsigned char)((word >> 0) & 0xffu);
        pixels[4 * i + 1] = (unsigned char)((word >> 8) & 0xffu);
        pixels[4 * i + 2] = (unsigned char)((word >> 16) & 0xffu);
        pixels[4 * i + 3] = (unsigned char)((word >> 24) & 0xffu);
    }

    return 0;
}

int camera_capture_frame(camera_select_t camera, unsigned char *pixels)
{
    volatile uint32_t *regs;
    int fd;
    int ret;

    if (map_regs(&regs, &fd) < 0) {
        return -1;
    }
    ret = capture_frame_regs(regs, camera, pixels);
    unmap_regs(regs, fd);
    return ret;
}

static void base64_write(FILE *out, const unsigned char *data, size_t len)
{
    unsigned line = 0;

    for (size_t i = 0; i < len; i += 3) {
        unsigned v = (unsigned)data[i] << 16;
        int have1 = i + 1 < len;
        int have2 = i + 2 < len;

        if (have1) {
            v |= (unsigned)data[i + 1] << 8;
        }
        if (have2) {
            v |= data[i + 2];
        }

        fputc(b64[(v >> 18) & 0x3f], out);
        fputc(b64[(v >> 12) & 0x3f], out);
        fputc(have1 ? b64[(v >> 6) & 0x3f] : '=', out);
        fputc(have2 ? b64[v & 0x3f] : '=', out);

        line += 4;
        if (line >= 76) {
            fputc('\n', out);
            line = 0;
        }
    }

    if (line) {
        fputc('\n', out);
    }
```

```
}

int camera_capture_serial(camera_select_t camera, FILE *out)
{
    long run_time = (long)time(NULL);
    long pid = (long)getpid();
    char name[NAME_LEN];
    char header[PGM_HEADER_LEN];
    int name_len;
    int header_len;
    unsigned char *pixels = malloc(FRAME_PIXELS);
    unsigned char *pgm;

    if (!pixels) {
        perror("malloc pixels");
        return -1;
    }
    if (camera_capture_frame(camera, pixels) < 0) {
        free(pixels);
        return -1;
    }

    name_len = snprintf(name, sizeof(name), "frame_cam%c_%ld_%ld.pgm",
                        camera_letter(camera), run_time, pid);
    header_len = snprintf(header, sizeof(header), "P5\n%u %u\n255\n",
                          FRAME_WIDTH, FRAME_HEIGHT);
    if (name_len < 0 || (size_t)name_len >= sizeof(name) ||
        header_len < 0 || (size_t)header_len >= sizeof(header)) {
        fprintf(stderr, "serial filename/header too long\n");
        free(pixels);
        return -1;
    }

    pgm = malloc((size_t)header_len + FRAME_PIXELS);
    if (!pgm) {
        perror("malloc pgm");
        free(pixels);
        return -1;
    }

    memcpy(pgm, header, (size_t)header_len);
    memcpy(pgm + header_len, pixels, FRAME_PIXELS);

    fprintf(out, "BEGIN_FRAME 0 %s\n", name);
    base64_write(out, pgm, (size_t)header_len + FRAME_PIXELS);
    fprintf(out, "END_FRAME 0\n");
    fflush(out);

    free(pgm);
    free(pixels);
    return 0;
}

int camera_debug_stream(unsigned print_every)
{
    volatile uint32_t *regs;
    int fd;
    struct timespec last_print;
    int have_last_print = 0;
    int fresh_a = 0;
    int fresh_b = 0;
    unsigned sample = 0;
    camera_moments_t moments_a = {0, 0, 0, 0.0, 0.0};
    camera_moments_t moments_b = {0, 0, 0, 0.0, 0.0};
    uint32_t paired_done_snapshot = 0;
    unsigned idle_polls = 0;

    if (print_every == 0) {
```

```
        print_every = 1;
    }
    if (map_regs(&regs, &fd) < 0) {
        return -1;
    }

    printf("areaA,uA,vA,cxA,cyA,areaB,uB,vB,cxB,cyB,done,hz\n");

    while (1) {
        uint32_t done = regs[IMG_DONE];

        if ((done & DONE_MOMENTS) == 0) {
            idle_polls++;
            if (idle_polls >= DONE_TIMEOUT_POLLS) {
                fprintf(stderr,
                        "moments: timeout waiting for DONE; DONE=0x%08x\n",
                        regs[IMG_DONE]);
                break;
            }
            continue;
        }
        idle_polls = 0;

        if (done & DONE_A) {
            read_moments_a(regs, &moments_a);
            clear_done(regs, DONE_A);
            if (wait_clear(regs, DONE_A, "camera A moments") < 0) {
                break;
            }
            fresh_a = 1;
            paired_done_snapshot |= DONE_A;
        }

        if (done & DONE_B) {
            read_moments_b(regs, &moments_b);
            clear_done(regs, DONE_B);
            if (wait_clear(regs, DONE_B, "camera B moments") < 0) {
                break;
            }
            fresh_b = 1;
            paired_done_snapshot |= DONE_B;
        }

        if (!fresh_a || !fresh_b) {
            continue;
        }

        sample++;

        if (sample % print_every == 0) {
            struct timespec now;
            double hz = 0.0;

            clock_gettime(CLOCK_MONOTONIC, &now);
            if (have_last_print) {
                double dt = elapsed_seconds(&last_print, &now);
                if (dt > 0.0) {
                    hz = (double)print_every / dt;
                }
            }
            last_print = now;
            have_last_print = 1;

            printf("%u,%u,%u,%.2f,%.2f,%u,%u,%u,%.2f,%.2f,0x%08x,%.2f\n",
                   moments_a.area, moments_a.u, moments_a.v,
                   moments_a.cx, moments_a.cy,
                   moments_b.area, moments_b.u, moments_b.v,
                   moments_b.cx, moments_b.cy,
```

```
                    paired_done_snapshot, hz);
            fflush(stdout);
        }

        fresh_a = 0;
        fresh_b = 0;
        paired_done_snapshot = 0;
    }

    unmap_regs(regs, fd);
    return -1;
}

int camera_print_status(void)
{
    volatile uint32_t *regs;
    int fd;
    uint32_t done;
    uint32_t control;

    if (map_regs(&regs, &fd) < 0) {
        return -1;
    }

    done = regs[IMG_DONE];
    control = regs[IMG_CONTROL];
    printf("DONE=0x%08x␣A=%u␣B=%u␣FB=%u␣CONTROL=0x%08x␣store=%u␣thA=%u␣thB=%u\n",
           done, (done & DONE_A) != 0, (done & DONE_B) != 0,
           (done & DONE_FB) != 0, control, control & STORE_MASK,
           (control >> THRESH_A_SHIFT) & THRESH_MASK,
           (control >> THRESH_B_SHIFT) & THRESH_MASK);

    unmap_regs(regs, fd);
    return 0;
}
```

A.3 camera_config.c

```
#define _DEFAULT_SOURCE

#include "camera_config.h"

#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>

#define GPIO0_I2C_OFFSET 0x120u
#define GPIO1_I2C_OFFSET 0x160u
#define PIO_DATA 0x00u
#define PIO_DIR 0x04u

#define SCL_BIT (1u << 0)
#define SDA_BIT (1u << 1)
#define I2C_BITS (SCL_BIT | SDA_BIT)

#define OV7670_ADDR 0x21u
#define REG_GAIN 0x00u
#define REG_VREF 0x03u
#define REG_COM1 0x04u
#define REG_AECHH 0x07u
#define REG_PID 0x0au
#define REG_VER 0x0bu
#define REG_COM3 0x0cu
```

```
#define REG_AECH 0x10u
#define REG_CLKRC 0x11u
#define REG_COM7 0x12u
#define REG_COM8 0x13u
#define REG_COM9 0x14u
#define REG_COM10 0x15u
#define REG_HSTART 0x17u
#define REG_HSTOP 0x18u
#define REG_VSTART 0x19u
#define REG_VSTOP 0x1au
#define REG_HREF 0x32u
#define REG_TSLB 0x3au
#define REG_COM13 0x3du
#define REG_COM14 0x3eu
#define REG_COM15 0x40u
#define REG_MANU 0x67u
#define REG_MANV 0x68u
#define REG_RGB444 0x8cu

#define COM7_RESET 0x80u
#define COM7_YUV 0x00u
#define COM8_FASTAEC 0x80u
#define COM8_AECSTEP 0x40u
#define COM8_AGC 0x04u
#define COM8_AEC 0x01u
#define COM15_FULL_RANGE 0xc0u
#define TSLB_FIXED_UV 0x10u

#define I2C_DELAY_US 5
#define SCCB_STOP_US 1000
#define VGA_VREF_LOW_BITS 0x0au

struct camera_reg {
    const char *name;
    uint8_t reg;
    uint8_t val;
    uint8_t mask;
};

static const struct camera_reg grayscale_vga_regs[] = {
    {"CLKRC", REG_CLKRC, 0x40, 0x7f},
    {"COM7", REG_COM7, COM7_YUV, 0xff},
    {"COM3", REG_COM3, 0x00, 0xff},
    {"COM14", REG_COM14, 0x00, 0xff},
    {"HSTART", REG_HSTART, 0x13, 0xff},
    {"HSTOP", REG_HSTOP, 0x01, 0xff},
    {"HREF", REG_HREF, 0xb6, 0xff},
    {"VSTART", REG_VSTART, 0x02, 0xff},
    {"VSTOP", REG_VSTOP, 0x7a, 0xff},
    {"TSLB", REG_TSLB, TSLB_FIXED_UV, 0xff},
    {"MANU", REG_MANU, 0x80, 0xff},
    {"MANV", REG_MANV, 0x80, 0xff},
    {"COM13", REG_COM13, 0x00, 0xff},
    {"RGB444", REG_RGB444, 0x00, 0xff},
    {"COM15", REG_COM15, COM15_FULL_RANGE, 0xff},
    {"COM10", REG_COM10, 0x00, 0xff},
};

static volatile uint32_t *pio_data;
static volatile uint32_t *pio_dir;

static void delay_i2c(void)
{
    usleep(I2C_DELAY_US);
}

static void drive_low(uint32_t bit)
{
```

```
    *pio_data &= ~bit;
    *pio_dir |= bit;
    delay_i2c();
}

static void release_line(uint32_t bit)
{
    *pio_data &= ~bit;
    *pio_dir &= ~bit;
    delay_i2c();
}

static int read_line(uint32_t bit)
{
    return (*pio_data & bit) != 0;
}

static int wait_high(uint32_t bit)
{
    for (unsigned i = 0; i < 1000; i++) {
        if (read_line(bit)) {
            return 0;
        }
        usleep(1);
    }
    return -1;
}

static int scl_high(void)
{
    release_line(SCL_BIT);
    return wait_high(SCL_BIT);
}

static void scl_low(void)
{
    drive_low(SCL_BIT);
}

static void sda_high(void)
{
    release_line(SDA_BIT);
}

static void sda_low(void)
{
    drive_low(SDA_BIT);
}

static int i2c_start(void)
{
    sda_high();
    if (scl_high() < 0) {
        return -1;
    }
    sda_low();
    scl_low();
    return 0;
}

static int i2c_stop(void)
{
    sda_low();
    if (scl_high() < 0) {
        return -1;
    }
    sda_high();
    return 0;
```

```
}

static int i2c_write_byte(uint8_t value)
{
    for (int bit = 7; bit >= 0; bit--) {
        if (value & (1u << bit)) {
            sda_high();
        } else {
            sda_low();
        }

        if (scl_high() < 0) {
            return -1;
        }
        scl_low();
    }

    sda_high();
    if (scl_high() < 0) {
        return -1;
    }

    int ack = !read_line(SDA_BIT);
    scl_low();
    return ack ? 0 : -2;
}

static int i2c_read_byte(uint8_t *value, int nack)
{
    uint8_t byte = 0;

    sda_high();
    for (int bit = 7; bit >= 0; bit--) {
        if (scl_high() < 0) {
            return -1;
        }
        if (read_line(SDA_BIT)) {
            byte |= (uint8_t)(1u << bit);
        }
        scl_low();
    }

    if (nack) {
        sda_high();
    } else {
        sda_low();
    }

    if (scl_high() < 0) {
        return -1;
    }
    scl_low();
    sda_high();

    *value = byte;
    return 0;
}

static int ov7670_read_reg(uint8_t reg, uint8_t *value)
{
    int ret = i2c_start();

    if (ret < 0) {
        return ret;
    }
    ret = i2c_write_byte((uint8_t)(OV7670_ADDR << 1));
    if (ret < 0) {
        i2c_stop();
```

```
        return ret;
    }
    ret = i2c_write_byte(reg);
    if (ret < 0) {
        i2c_stop();
        return ret;
    }
    i2c_stop();
    usleep(SCCB_STOP_US);

    ret = i2c_start();
    if (ret < 0) {
        return ret;
    }
    ret = i2c_write_byte((uint8_t)((OV7670_ADDR << 1) | 1u));
    if (ret < 0) {
        i2c_stop();
        return ret;
    }
    ret = i2c_read_byte(value, 1);
    i2c_stop();
    usleep(SCCB_STOP_US);
    return ret;
}

static int ov7670_write_reg(uint8_t reg, uint8_t value)
{
    int ret = i2c_start();

    if (ret < 0) {
        return ret;
    }
    ret = i2c_write_byte((uint8_t)(OV7670_ADDR << 1));
    if (ret < 0) {
        i2c_stop();
        return ret;
    }
    ret = i2c_write_byte(reg);
    if (ret < 0) {
        i2c_stop();
        return ret;
    }
    ret = i2c_write_byte(value);
    i2c_stop();
    usleep(SCCB_STOP_US);
    return ret;
}

static int ov7670_write_table(const struct camera_reg *regs, size_t count)
{
    for (size_t i = 0; i < count; i++) {
        int ret = ov7670_write_reg(regs[i].reg, regs[i].val);
        if (ret < 0) {
            fprintf(stderr, "write␣%-18s␣reg␣0x%02x␣failed:␣ret=%d\n",
                    regs[i].name, regs[i].reg, ret);
            return ret;
        }
    }
    return 0;
}

static int ov7670_set_gain(unsigned gain)
{
    uint8_t low = (uint8_t)(gain & 0xffu);
    uint8_t high = (uint8_t)(((gain >> 8) & 0x03u) << 6);
    int ret = ov7670_write_reg(REG_GAIN, low);

    if (ret < 0) {
```

```
        return ret;
    }
    return ov7670_write_reg(REG_VREF, (uint8_t)(VGA_VREF_LOW_BITS | high));
}

static uint8_t ov7670_com8_value(const camera_settings_t *settings)
{
    uint8_t value = COM8_FASTAEC | COM8_AECSTEP;

    if (settings->agc) {
        value |= COM8_AGC;
    }
    if (settings->aec) {
        value |= COM8_AEC;
    }

    return value;
}

static int ov7670_set_exposure(unsigned exposure)
{
    uint8_t com1 = 0;
    uint8_t aechh = 0;
    int ret = ov7670_read_reg(REG_COM1, &com1);

    if (ret < 0) {
        return ret;
    }
    ret = ov7670_read_reg(REG_AECHH, &aechh);
    if (ret < 0) {
        return ret;
    }

    ret = ov7670_write_reg(REG_AECHH,
                           (uint8_t)((aechh & 0xc0u) |
                                     ((exposure >> 10) & 0x3fu)));
    if (ret < 0) {
        return ret;
    }
    ret = ov7670_write_reg(REG_AECH, (uint8_t)((exposure >> 2) & 0xffu));
    if (ret < 0) {
        return ret;
    }
    return ov7670_write_reg(REG_COM1,
                            (uint8_t)((com1 & 0xfcu) |
                                      (exposure & 0x03u)));
}

static int ov7670_configure_grayscale_vga(const camera_settings_t *settings)
{
    int ret = ov7670_write_reg(REG_COM7, COM7_RESET);

    if (ret < 0) {
        return ret;
    }
    usleep(10000);

    ret = ov7670_write_table(grayscale_vga_regs,
                             sizeof(grayscale_vga_regs) /
                                 sizeof(grayscale_vga_regs[0]));
    if (ret < 0) {
        return ret;
    }
    ret = ov7670_write_reg(REG_COM8, ov7670_com8_value(settings));
    if (ret < 0) {
        return ret;
    }
    ret = ov7670_set_gain(settings->gain);
```

```
    if (ret < 0) {
        return ret;
    }
    if (settings->exposure_set) {
        ret = ov7670_set_exposure(settings->exposure);
        if (ret < 0) {
            return ret;
        }
    }
    if (settings->gain_ceiling_set) {
        ret = ov7670_write_reg(REG_COM9, (uint8_t)settings->gain_ceiling);
        if (ret < 0) {
            return ret;
        }
    }
    return ov7670_write_reg(REG_COM8, ov7670_com8_value(settings));
}

static int ov7670_verify_reg(const char *name, uint8_t reg, uint8_t expected,
                             uint8_t mask)
{
    uint8_t actual = 0;
    int ret = ov7670_read_reg(reg, &actual);
    int pass;

    if (ret < 0) {
        printf("%-18s reg=0x%02x read failed ret=%d FAIL\n", name, reg, ret);
        return -1;
    }

    pass = (actual & mask) == (expected & mask);
    printf("%-18s reg=0x%02x read=0x%02x expect=0x%02x mask=0x%02x %s\n",
           name, reg, actual, expected, mask, pass ? "OK" : "FAIL");
    return pass ? 0 : -1;
}

static int ov7670_verify_exposure(unsigned expected)
{
    uint8_t com1 = 0;
    uint8_t aech = 0;
    uint8_t aechh = 0;
    int ret = ov7670_read_reg(REG_COM1, &com1);
    unsigned actual;
    int pass;

    if (ret < 0) {
        printf("%-18s read failed ret=%d FAIL\n", "EXPOSURE", ret);
        return -1;
    }
    ret = ov7670_read_reg(REG_AECH, &aech);
    if (ret < 0) {
        printf("%-18s read failed ret=%d FAIL\n", "EXPOSURE", ret);
        return -1;
    }
    ret = ov7670_read_reg(REG_AECHH, &aechh);
    if (ret < 0) {
        printf("%-18s read failed ret=%d FAIL\n", "EXPOSURE", ret);
        return -1;
    }

    actual = (unsigned)(com1 & 0x03u) |
             ((unsigned)aech << 2) |
             ((unsigned)(aechh & 0x3fu) << 10);
    pass = actual == expected;

    printf("%-18s read=0x%04x expect=0x%04x %s\n",
           "EXPOSURE", actual, expected, pass ? "OK" : "FAIL");
    return pass ? 0 : -1;
```

```
}

static int ov7670_verify_grayscale_vga(const camera_settings_t *settings)
{
    unsigned failures = 0;
    uint8_t gain_low = (uint8_t)(settings->gain & 0xffu);
    uint8_t gain_high = (uint8_t)(((settings->gain >> 8) & 0x03u) << 6);
    uint8_t vref = (uint8_t)(VGA_VREF_LOW_BITS | gain_high);
    uint8_t com8 = ov7670_com8_value(settings);

    printf("Verifying␣OV7670␣register␣readback␣before␣capture:\n");

    for (size_t i = 0; i < sizeof(grayscale_vga_regs) /
                           sizeof(grayscale_vga_regs[0]);
         i++) {
        const struct camera_reg *v = &grayscale_vga_regs[i];
        if (ov7670_verify_reg(v->name, v->reg, v->val, v->mask) < 0) {
            failures++;
        }
    }

    if (ov7670_verify_reg("COM8", REG_COM8, com8, 0xff) < 0) {
        failures++;
    }

    if (settings->gain_ceiling_set &&
        ov7670_verify_reg("COM9", REG_COM9, (uint8_t)settings->gain_ceiling,
                          0xff) < 0) {
        failures++;
    }

    if (settings->agc) {
        printf("%-18s␣auto␣gain␣enabled;␣exact␣manual␣gain␣readback␣skipped\n",
               "GAIN");
        if (ov7670_verify_reg("VREF", REG_VREF, VGA_VREF_LOW_BITS, 0x0f) < 0) {
            failures++;
        }
    } else {
        if (ov7670_verify_reg("GAIN", REG_GAIN, gain_low, 0xff) < 0) {
            failures++;
        }
        if (ov7670_verify_reg("VREF", REG_VREF, vref, 0xff) < 0) {
            failures++;
        }
    }

    if (settings->exposure_set) {
        if (settings->aec) {
            printf("%-18s␣auto␣exposure␣enabled;␣exact␣readback␣skipped\n",
                   "EXPOSURE");
        } else if (ov7670_verify_exposure(settings->exposure) < 0) {
            failures++;
        }
    }

    if (failures) {
        fprintf(stderr, "%u␣register␣verification␣failure(s);␣capture␣not␣ready\n",
                failures);
        return -1;
    }

    printf("Register␣verification␣passed;␣capture␣may␣start.\n");
    return 0;
}

int camera_config_parse_setting(const char *text, camera_settings_t *settings)
{
    const char *value = strchr(text, '=');
```

```
    size_t name_len;

    if (!value) {
        return parse_unsigned_arg(text, CAMERA_GAIN_MAX, &settings->gain);
    }

    name_len = (size_t)(value - text);
    value++;

    if (name_len == 4 && strncmp(text, "gain", name_len) == 0) {
        return parse_unsigned_arg(value, CAMERA_GAIN_MAX, &settings->gain);
    }
    if (name_len == 3 && strncmp(text, "agc", name_len) == 0) {
        return parse_bool_arg(value, &settings->agc);
    }
    if (name_len == 3 && strncmp(text, "aec", name_len) == 0) {
        return parse_bool_arg(value, &settings->aec);
    }
    if (name_len == 8 && strncmp(text, "exposure", name_len) == 0) {
        if (parse_unsigned_arg(value, CAMERA_EXPOSURE_MAX,
                               &settings->exposure) < 0) {
            return -1;
        }
        settings->exposure_set = 1;
        return 0;
    }
    if (name_len == 7 && strncmp(text, "ceiling", name_len) == 0) {
        if (parse_unsigned_arg(value, CAMERA_REG8_MAX,
                               &settings->gain_ceiling) < 0) {
            return -1;
        }
        settings->gain_ceiling_set = 1;
        return 0;
    }
    if (name_len == 4 && strncmp(text, "com9", name_len) == 0) {
        if (parse_unsigned_arg(value, CAMERA_REG8_MAX,
                               &settings->gain_ceiling) < 0) {
            return -1;
        }
        settings->gain_ceiling_set = 1;
        return 0;
    }

    return -1;
}

static int configure_one_camera(void *map, uint32_t i2c_offset,
                                const char *label,
                                const camera_settings_t *settings)
{
    uint32_t old_data;
    uint32_t old_dir;
    uint8_t pid = 0;
    uint8_t ver = 0;
    int ret_pid;
    int ret_ver;
    int ret_cfg = 0;
    int ret_verify = 0;
    int failed = 0;

    printf("\nConfiguring %s\n", label);

    pio_data = (volatile uint32_t *)((uint8_t *)map + i2c_offset + PIO_DATA);
    pio_dir = (volatile uint32_t *)((uint8_t *)map + i2c_offset + PIO_DIR);

    old_data = *pio_data;
    old_dir = *pio_dir;

```

```
    *pio_data = old_data & ~I2C_BITS;
    *pio_dir = old_dir & ~I2C_BITS;
    usleep(SCCB_STOP_US);

    if (!read_line(SCL_BIT) || !read_line(SDA_BIT)) {
        fprintf(stderr, "%s: SCL/SDA did not release high; check wiring\n",
                label);
        failed = 1;
        goto out;
    }

    ret_pid = ov7670_read_reg(REG_PID, &pid);
    ret_ver = ov7670_read_reg(REG_VER, &ver);

    if (ret_pid == 0 && ret_ver == 0) {
        ret_cfg = ov7670_configure_grayscale_vga(settings);
        if (ret_cfg == 0) {
            ret_verify = ov7670_verify_grayscale_vga(settings);
        }
    }

    if (ret_pid < 0 || ret_ver < 0) {
        fprintf(stderr, "%s: OV7670 read failed: PID ret=%d, VER ret=%d\n",
                label, ret_pid, ret_ver);
        fprintf(stderr, "ret=-1 means SCL stuck low, ret=-2 means no ACK\n");
        failed = 1;
        goto out;
    }

    if (ret_cfg < 0) {
        fprintf(stderr, "%s: grayscale VGA configuration failed: ret=%d\n",
                label, ret_cfg);
        fprintf(stderr, "ret=-1 means SCL stuck low, ret=-2 means no ACK\n");
        failed = 1;
        goto out;
    }

    if (ret_verify < 0) {
        failed = 1;
        goto out;
    }

    printf("%s: OV7670 PID=0x%02x VER=0x%02x configured grayscale VGA\n",
           label, pid, ver);
    printf("%s: gain=0x%03x agc=%d aec=%d",
           label, settings->gain, settings->agc, settings->aec);
    if (settings->exposure_set) {
        printf(" exposure=0x%04x", settings->exposure);
    }
    if (settings->gain_ceiling_set) {
        printf(" ceiling=0x%02x", settings->gain_ceiling);
    }
    printf("\n");

out:
    i2c_stop();
    *pio_data = old_data;
    *pio_dir = old_dir;
    return failed ? -1 : 0;
}

int camera_configure(camera_select_t cameras, const camera_settings_t *settings)
{
    int fd = open("/dev/mem", O_RDWR | O_SYNC);
    int failed = 0;
    void *map;

    if (fd < 0) {
```

```
641         perror("open␣/dev/mem");
642         return -1;
643     }
644 
645     map = mmap(NULL, DE1_LW_BRIDGE_SPAN, PROT_READ | PROT_WRITE, MAP_SHARED,
646                fd, DE1_LW_BRIDGE_BASE);
647     if (map == MAP_FAILED) {
648         perror("mmap");
649         close(fd);
650         return -1;
651     }
652 
653     if ((cameras & CAMERA_A) &&
654         configure_one_camera(map, GPIO0_I2C_OFFSET, "camera␣A␣(GPIO0)",
655                              settings) < 0) {
656         failed = 1;
657     }
658     if ((cameras & CAMERA_B) &&
659         configure_one_camera(map, GPIO1_I2C_OFFSET, "camera␣B␣(GPIO1)",
660                              settings) < 0) {
661         failed = 1;
662     }
663 
664     munmap(map, DE1_LW_BRIDGE_SPAN);
665     close(fd);
666     return failed ? -1 : 0;
667 }
```

A.4 fundamental.c

```
1  #include "fundamental.h"
2  
3  #include <math.h>
4  #include <stddef.h>
5  #include <stdlib.h>
6  #include <string.h>
7  
8  #define F_N 9
9  #define JACOBI_ITERS 128
10 #define JACOBI_EPS 1.0e-12
11 
12 static void jacobi_smallest_eigenvector_9(const double a_in[F_N][F_N],
13                                            double v_out[F_N])
14 {
15     double a[F_N][F_N];
16     double v[F_N][F_N] = {{0.0}};
17 
18     memcpy(a, a_in, sizeof(a));
19     for (int i = 0; i < F_N; i++) {
20         v[i][i] = 1.0;
21     }
22 
23     for (int iter = 0; iter < JACOBI_ITERS; iter++) {
24         int p = 0;
25         int q = 1;
26         double max_off = fabs(a[p][q]);
27 
28         for (int r = 0; r < F_N; r++) {
29             for (int c = r + 1; c < F_N; c++) {
30                 double off = fabs(a[r][c]);
31                 if (off > max_off) {
32                     max_off = off;
33                     p = r;
34                     q = c;
35                 }
36             }
```

```
        }

        if (max_off < JACOBI_EPS) {
            break;
        }

        double app = a[p][p];
        double aqq = a[q][q];
        double apq = a[p][q];
        double tau = (aqq - app) / (2.0 * apq);
        double t = (tau >= 0.0 ? 1.0 : -1.0) /
                   (fabs(tau) + sqrt(1.0 + tau * tau));
        double c = 1.0 / sqrt(1.0 + t * t);
        double s = t * c;

        a[p][p] = app - t * apq;
        a[q][q] = aqq + t * apq;
        a[p][q] = 0.0;
        a[q][p] = 0.0;

        for (int k = 0; k < F_N; k++) {
            if (k != p && k != q) {
                double akp = a[k][p];
                double akq = a[k][q];
                a[k][p] = c * akp - s * akq;
                a[p][k] = a[k][p];
                a[k][q] = s * akp + c * akq;
                a[q][k] = a[k][q];
            }
        }

        for (int k = 0; k < F_N; k++) {
            double vkp = v[k][p];
            double vkq = v[k][q];
            v[k][p] = c * vkp - s * vkq;
            v[k][q] = s * vkp + c * vkq;
        }
    }

    int smallest = 0;
    for (int i = 1; i < F_N; i++) {
        if (a[i][i] < a[smallest][smallest]) {
            smallest = i;
        }
    }

    for (int i = 0; i < F_N; i++) {
        v_out[i] = v[i][smallest];
    }
}

static void normalize_points(const point2_t *in, point2_t *out, size_t count,
                             double t[9])
{
    double mx = 0.0;
    double my = 0.0;
    double avg_dist = 0.0;

    for (size_t i = 0; i < count; i++) {
        mx += in[i].x;
        my += in[i].y;
    }
    mx /= (double)count;
    my /= (double)count;

    for (size_t i = 0; i < count; i++) {
        double dx = in[i].x - mx;
        double dy = in[i].y - my;
```

```
        avg_dist += sqrt(dx * dx + dy * dy);
    }
    avg_dist /= (double)count;

    double s = avg_dist > 0.0 ? sqrt(2.0) / avg_dist : 1.0;

    t[0] = s;
    t[1] = 0.0;
    t[2] = -s * mx;
    t[3] = 0.0;
    t[4] = s;
    t[5] = -s * my;
    t[6] = 0.0;
    t[7] = 0.0;
    t[8] = 1.0;

    for (size_t i = 0; i < count; i++) {
        out[i].x = s * (in[i].x - mx);
        out[i].y = s * (in[i].y - my);
    }
}

static void mat3_mul(const double a[9], const double b[9], double out[9])
{
    double tmp[9];

    for (int r = 0; r < 3; r++) {
        for (int c = 0; c < 3; c++) {
            tmp[3 * r + c] = 0.0;
            for (int k = 0; k < 3; k++) {
                tmp[3 * r + c] += a[3 * r + k] * b[3 * k + c];
            }
        }
    }
    memcpy(out, tmp, sizeof(tmp));
}

static void mat3_transpose(const double in[9], double out[9])
{
    for (int r = 0; r < 3; r++) {
        for (int c = 0; c < 3; c++) {
            out[3 * r + c] = in[3 * c + r];
        }
    }
}

int fundamental_estimate(const point2_t *left, const point2_t *right,
                         size_t count, double f_out[9])
{
    double ata[F_N][F_N] = {{0.0}};
    double f_norm[9];
    double t_left[9];
    double t_right[9];
    double t_right_t[9];
    double tmp[9];
    point2_t *left_norm;
    point2_t *right_norm;

    if (!left || !right || !f_out || count < 8) {
        return -1;
    }

    left_norm = malloc(count * sizeof(*left_norm));
    right_norm = malloc(count * sizeof(*right_norm));
    if (!left_norm || !right_norm) {
        free(left_norm);
        free(right_norm);
        return -1;
```

```
    }

    normalize_points(left, left_norm, count, t_left);
    normalize_points(right, right_norm, count, t_right);

    for (size_t i = 0; i < count; i++) {
        double x1 = left_norm[i].x;
        double y1 = left_norm[i].y;
        double x2 = right_norm[i].x;
        double y2 = right_norm[i].y;
        double row[F_N] = {
            x2 * x1, x2 * y1, x2,
            y2 * x1, y2 * y1, y2,
            x1,      y1,      1.0,
        };

        for (int r = 0; r < F_N; r++) {
            for (int c = 0; c < F_N; c++) {
                ata[r][c] += row[r] * row[c];
            }
        }
    }

    jacobi_smallest_eigenvector_9(ata, f_norm);

    mat3_transpose(t_right, t_right_t);
    mat3_mul(t_right_t, f_norm, tmp);
    mat3_mul(tmp, t_left, f_out);

    double norm = 0.0;
    for (int i = 0; i < 9; i++) {
        norm += f_out[i] * f_out[i];
    }
    norm = sqrt(norm);
    if (norm > 0.0) {
        for (int i = 0; i < 9; i++) {
            f_out[i] /= norm;
        }
    }

    free(left_norm);
    free(right_norm);
    return 0;
}
```

A.5 essential.c

```
#include "essential.h"

#include "projection.h"

#include <math.h>
#include <string.h>

#define E_N 3
#define E_JACOBI_ITERS 64
#define E_JACOBI_EPS 1.0e-12

static double dot3(const double a[3], const double b[3])
{
    return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}

static void cross3(const double a[3], const double b[3], double out[3])
{
    out[0] = a[1] * b[2] - a[2] * b[1];
```

```
    out[1] = a[2] * b[0] - a[0] * b[2];
    out[2] = a[0] * b[1] - a[1] * b[0];
}

static void normalize3(double v[3])
{
    double n = sqrt(dot3(v, v));

    if (n > 0.0) {
        v[0] /= n;
        v[1] /= n;
        v[2] /= n;
    }
}

static double det3(const double m[9])
{
    return m[0] * (m[4] * m[8] - m[5] * m[7]) -
           m[1] * (m[3] * m[8] - m[5] * m[6]) +
           m[2] * (m[3] * m[7] - m[4] * m[6]);
}

static void mat3_transpose(const double in[9], double out[9])
{
    for (int r = 0; r < 3; r++) {
        for (int c = 0; c < 3; c++) {
            out[3 * r + c] = in[3 * c + r];
        }
    }
}

static void mat3_mul(const double a[9], const double b[9], double out[9])
{
    double tmp[9];

    for (int r = 0; r < 3; r++) {
        for (int c = 0; c < 3; c++) {
            tmp[3 * r + c] = 0.0;
            for (int k = 0; k < 3; k++) {
                tmp[3 * r + c] += a[3 * r + k] * b[3 * k + c];
            }
        }
    }
    memcpy(out, tmp, sizeof(tmp));
}

static void jacobi_eigen_symmetric3(const double a_in[9], double eval[3],
                                    double evec[9])
{
    double a[9];

    memcpy(a, a_in, sizeof(a));
    memset(evec, 0, 9 * sizeof(double));
    evec[0] = 1.0;
    evec[4] = 1.0;
    evec[8] = 1.0;

    for (int iter = 0; iter < E_JACOBI_ITERS; iter++) {
        int p = 0;
        int q = 1;
        double max_off = fabs(a[1]);

        if (fabs(a[2]) > max_off) {
            max_off = fabs(a[2]);
            p = 0;
            q = 2;
        }
        if (fabs(a[5]) > max_off) {
```

```
            max_off = fabs(a[5]);
            p = 1;
            q = 2;
        }
        if (max_off < E_JACOBI_EPS) {
            break;
        }

        double app = a[3 * p + p];
        double aqq = a[3 * q + q];
        double apq = a[3 * p + q];
        double tau = (aqq - app) / (2.0 * apq);
        double t = (tau >= 0.0 ? 1.0 : -1.0) /
                   (fabs(tau) + sqrt(1.0 + tau * tau));
        double c = 1.0 / sqrt(1.0 + t * t);
        double s = t * c;

        a[3 * p + p] = app - t * apq;
        a[3 * q + q] = aqq + t * apq;
        a[3 * p + q] = 0.0;
        a[3 * q + p] = 0.0;

        for (int k = 0; k < 3; k++) {
            if (k != p && k != q) {
                double akp = a[3 * k + p];
                double akq = a[3 * k + q];
                a[3 * k + p] = c * akp - s * akq;
                a[3 * p + k] = a[3 * k + p];
                a[3 * k + q] = s * akp + c * akq;
                a[3 * q + k] = a[3 * k + q];
            }
        }

        for (int k = 0; k < 3; k++) {
            double vkp = evec[3 * k + p];
            double vkq = evec[3 * k + q];
            evec[3 * k + p] = c * vkp - s * vkq;
            evec[3 * k + q] = s * vkp + c * vkq;
        }
    }

    eval[0] = a[0];
    eval[1] = a[4];
    eval[2] = a[8];

    for (int i = 0; i < 2; i++) {
        for (int j = i + 1; j < 3; j++) {
            if (eval[j] > eval[i]) {
                double ev = eval[i];
                eval[i] = eval[j];
                eval[j] = ev;
                for (int r = 0; r < 3; r++) {
                    double vv = evec[3 * r + i];
                    evec[3 * r + i] = evec[3 * r + j];
                    evec[3 * r + j] = vv;
                }
            }
        }
    }
}

static void svd3(const double e[9], double u[9], double v[9])
{
    double et[9];
    double ete[9];
    double eval[3];
    double col0[3];
    double col1[3];
```

```
    double col2[3];

    mat3_transpose(e, et);
    mat3_mul(et, e, ete);
    jacobi_eigen_symmetric3(ete, eval, v);

    memset(u, 0, 9 * sizeof(double));
    for (int col = 0; col < 3; col++) {
        double sigma = eval[col] > 0.0 ? sqrt(eval[col]) : 0.0;

        if (sigma > 1.0e-12) {
            for (int row = 0; row < 3; row++) {
                for (int k = 0; k < 3; k++) {
                    u[3 * row + col] += e[3 * row + k] * v[3 * k + col];
                }
                u[3 * row + col] /= sigma;
            }
        }
    }

    for (int i = 0; i < 3; i++) {
        col0[i] = u[3 * i + 0];
        col1[i] = u[3 * i + 1];
    }
    normalize3(col0);
    double dot01 = dot3(col0, col1);
    for (int i = 0; i < 3; i++) {
        col1[i] -= dot01 * col0[i];
    }
    normalize3(col1);
    cross3(col0, col1, col2);
    normalize3(col2);

    for (int i = 0; i < 3; i++) {
        u[3 * i + 0] = col0[i];
        u[3 * i + 1] = col1[i];
        u[3 * i + 2] = col2[i];
    }

    if (det3(u) < 0.0) {
        for (int i = 0; i < 3; i++) {
            u[3 * i + 2] = -u[3 * i + 2];
        }
    }
    if (det3(v) < 0.0) {
        for (int i = 0; i < 3; i++) {
            v[3 * i + 2] = -v[3 * i + 2];
        }
    }
}

void essential_from_fundamental(const double k_left[9],
                                const double k_right[9],
                                const double f[9],
                                double e_out[9])
{
    double k_right_t[9];
    double tmp[9];

    mat3_transpose(k_right, k_right_t);
    mat3_mul(k_right_t, f, tmp);
    mat3_mul(tmp, k_left, e_out);
}

static void make_candidate(const double u[9], const double v[9],
                           const double w[9], const double t[3],
                           camera_pose_t *pose)
{
```

```
    double vt[9];
    double tmp[9];

    mat3_transpose(v, vt);
    mat3_mul(u, w, tmp);
    mat3_mul(tmp, vt, pose->r);

    if (det3(pose->r) < 0.0) {
        for (int i = 0; i < 9; i++) {
            pose->r[i] = -pose->r[i];
        }
    }

    pose->t[0] = t[0];
    pose->t[1] = t[1];
    pose->t[2] = t[2];
}

static int chirality_score(const camera_pose_t *pose, const point2_t *left,
                           const point2_t *right, size_t count)
{
    int score = 0;

    for (size_t i = 0; i < count; i++) {
        double x[3];
        double z_right;

        if (projection_triangulate_normalized(&left[i], &right[i],
                                              pose->r, pose->t, x) < 0) {
            continue;
        }

        z_right = pose->r[6] * x[0] + pose->r[7] * x[1] +
                  pose->r[8] * x[2] + pose->t[2];
        if (x[2] > 0.0 && z_right > 0.0) {
            score++;
        }
    }

    return score;
}

int essential_recover_pose(const double e[9], const point2_t *left_norm,
                           const point2_t *right_norm, size_t count,
                           camera_pose_t *pose_out)
{
    static const double w[9] = {
        0.0, -1.0, 0.0,
        1.0,  0.0, 0.0,
        0.0,  0.0, 1.0,
    };
    static const double wt[9] = {
        0.0, 1.0, 0.0,
       -1.0, 0.0, 0.0,
        0.0, 0.0, 1.0,
    };
    double u[9];
    double v[9];
    double t[3];
    camera_pose_t candidates[4];
    int best = -1;
    int best_score = -1;

    if (!e || !left_norm || !right_norm || !pose_out || count == 0) {
        return -1;
    }

    svd3(e, u, v);
```

```
    t[0] = u[2];
    t[1] = u[5];
    t[2] = u[8];
    normalize3(t);

    make_candidate(u, v, w, t, &candidates[0]);
    t[0] = -t[0];
    t[1] = -t[1];
    t[2] = -t[2];
    make_candidate(u, v, w, t, &candidates[1]);
    t[0] = -t[0];
    t[1] = -t[1];
    t[2] = -t[2];
    make_candidate(u, v, wt, t, &candidates[2]);
    t[0] = -t[0];
    t[1] = -t[1];
    t[2] = -t[2];
    make_candidate(u, v, wt, t, &candidates[3]);

    for (int i = 0; i < 4; i++) {
        int score = chirality_score(&candidates[i], left_norm, right_norm,
                                    count);
        if (score > best_score) {
            best_score = score;
            best = i;
        }
    }

    if (best < 0) {
        return -1;
    }

    *pose_out = candidates[best];
    return best_score > 0 ? 0 : -1;
}

void essential_projection_matrix(const double k[9],
                                 const camera_pose_t *pose,
                                 double p_out[12])
{
    projection_from_pose(k, pose->r, pose->t, p_out);
}
```

A.6 projection.c

```
#include "projection.h"

#include <math.h>
#include <string.h>

#define J4_N 4
#define J4_ITERS 64
#define J4_EPS 1.0e-12

static double dot3(const double a[3], const double b[3])
{
    return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}

static void cross3(const double a[3], const double b[3], double out[3])
{
    out[0] = a[1] * b[2] - a[2] * b[1];
    out[1] = a[2] * b[0] - a[0] * b[2];
    out[2] = a[0] * b[1] - a[1] * b[0];
}

```

```
22 static double norm3(const double v[3])
23 {
24     return sqrt(dot3(v, v));
25 }
26
27 static int normalize3(double v[3])
28 {
29     double n = norm3(v);
30
31     if (n < 1.0e-12) {
32         return -1;
33     }
34     v[0] /= n;
35     v[1] /= n;
36     v[2] /= n;
37     return 0;
38 }
39
40 static void mat3_vec(const double m[9], const double v[3], double out[3])
41 {
42     for (int r = 0; r < 3; r++) {
43         out[r] = m[3 * r + 0] * v[0] +
44                  m[3 * r + 1] * v[1] +
45                  m[3 * r + 2] * v[2];
46     }
47 }
48
49 static void rotation_from_to(const double from_in[3], const double to_in[3],
50                              double r[9])
51 {
52     double from[3] = {from_in[0], from_in[1], from_in[2]};
53     double to[3] = {to_in[0], to_in[1], to_in[2]};
54     double v[3];
55     double c;
56     double s;
57     double vx[9];
58
59     projection_identity3(r);
60     if (normalize3(from) < 0 || normalize3(to) < 0) {
61         return;
62     }
63
64     c = dot3(from, to);
65     if (c > 0.999999) {
66         return;
67     }
68
69     cross3(from, to, v);
70     s = norm3(v);
71     if (s < 1.0e-12) {
72         r[0] = -1.0;
73         r[4] = 1.0;
74         r[8] = -1.0;
75         return;
76     }
77
78     vx[0] = 0.0;
79     vx[1] = -v[2];
80     vx[2] = v[1];
81     vx[3] = v[2];
82     vx[4] = 0.0;
83     vx[5] = -v[0];
84     vx[6] = -v[1];
85     vx[7] = v[0];
86     vx[8] = 0.0;
87
88     for (int row = 0; row < 3; row++) {
89         for (int col = 0; col < 3; col++) {
```

```
            double vx2 = 0.0;
            for (int k = 0; k < 3; k++) {
                vx2 += vx[3 * row + k] * vx[3 * k + col];
            }
            r[3 * row + col] += vx[3 * row + col] +
                                vx2 * ((1.0 - c) / (s * s));
        }
    }
}

static void jacobi_smallest_eigenvector_4(const double a_in[J4_N][J4_N],
                                          double v_out[J4_N])
{
    double a[J4_N][J4_N];
    double v[J4_N][J4_N] = {{0.0}};

    memcpy(a, a_in, sizeof(a));
    for (int i = 0; i < J4_N; i++) {
        v[i][i] = 1.0;
    }

    for (int iter = 0; iter < J4_ITERS; iter++) {
        int p = 0;
        int q = 1;
        double max_off = fabs(a[p][q]);

        for (int r = 0; r < J4_N; r++) {
            for (int c = r + 1; c < J4_N; c++) {
                double off = fabs(a[r][c]);
                if (off > max_off) {
                    max_off = off;
                    p = r;
                    q = c;
                }
            }
        }

        if (max_off < J4_EPS) {
            break;
        }

        double app = a[p][p];
        double aqq = a[q][q];
        double apq = a[p][q];
        double tau = (aqq - app) / (2.0 * apq);
        double t = (tau >= 0.0 ? 1.0 : -1.0) /
                   (fabs(tau) + sqrt(1.0 + tau * tau));
        double c = 1.0 / sqrt(1.0 + t * t);
        double s = t * c;

        a[p][p] = app - t * apq;
        a[q][q] = aqq + t * apq;
        a[p][q] = 0.0;
        a[q][p] = 0.0;

        for (int k = 0; k < J4_N; k++) {
            if (k != p && k != q) {
                double akp = a[k][p];
                double akq = a[k][q];
                a[k][p] = c * akp - s * akq;
                a[p][k] = a[k][p];
                a[k][q] = s * akp + c * akq;
                a[q][k] = a[k][q];
            }
        }

        for (int k = 0; k < J4_N; k++) {
            double vkp = v[k][p];
```

```
            double vkq = v[k][q];
            v[k][p] = c * vkp - s * vkq;
            v[k][q] = s * vkp + c * vkq;
        }
    }

    int smallest = 0;
    for (int i = 1; i < J4_N; i++) {
        if (a[i][i] < a[smallest][smallest]) {
            smallest = i;
        }
    }
    for (int i = 0; i < J4_N; i++) {
        v_out[i] = v[i][smallest];
    }
}

void projection_identity3(double r[9])
{
    memset(r, 0, 9 * sizeof(double));
    r[0] = 1.0;
    r[4] = 1.0;
    r[8] = 1.0;
}

double projection_distance3(const double a[3], const double b[3])
{
    double d[3] = {
        b[0] - a[0],
        b[1] - a[1],
        b[2] - a[2],
    };

    return norm3(d);
}

point2_t projection_normalize_pixel(const double k[9], point2_t pixel)
{
    point2_t out;

    out.x = (pixel.x - k[2]) / k[0];
    out.y = (pixel.y - k[5]) / k[4];
    return out;
}

void projection_make_left_camera(const double k[9], double p[12])
{
    static const double eye[9] = {
        1.0, 0.0, 0.0,
        0.0, 1.0, 0.0,
        0.0, 0.0, 1.0,
    };
    static const double zero[3] = {0.0, 0.0, 0.0};

    projection_from_pose(k, eye, zero, p);
}

void projection_from_pose(const double k[9], const double r[9],
                          const double t[3], double p[12])
{
    double rt[12];

    for (int row = 0; row < 3; row++) {
        for (int col = 0; col < 3; col++) {
            rt[4 * row + col] = r[3 * row + col];
        }
        rt[4 * row + 3] = t[row];
    }
```

```
226
227     for (int row = 0; row < 3; row++) {
228         for (int col = 0; col < 4; col++) {
229             p[4 * row + col] = 0.0;
230             for (int kk = 0; kk < 3; kk++) {
231                 p[4 * row + col] += k[3 * row + kk] * rt[4 * kk + col];
232             }
233         }
234     }
235 }
236
237 int projection_project(const double p[12], const double x[3],
238                        point2_t *image_point)
239 {
240     double h[4] = {x[0], x[1], x[2], 1.0};
241     double u = 0.0;
242     double v = 0.0;
243     double w = 0.0;
244
245     for (int i = 0; i < 4; i++) {
246         u += p[i] * h[i];
247         v += p[4 + i] * h[i];
248         w += p[8 + i] * h[i];
249     }
250
251     if (fabs(w) < 1.0e-12) {
252         return -1;
253     }
254
255     image_point->x = u / w;
256     image_point->y = v / w;
257     return 0;
258 }
259
260 void projection_apply_transform(const double raw[3], const double origin[3],
261                                 int origin_set, const double rotation[9],
262                                 double scale, double out[3])
263 {
264     double rotated[3];
265
266     projection_apply_transform_unscaled(raw, origin, origin_set, rotation,
267                                         rotated);
268     out[0] = scale * rotated[0];
269     out[1] = scale * rotated[1];
270     out[2] = scale * rotated[2];
271 }
272
273 void projection_apply_transform_unscaled(const double raw[3],
274                                          const double origin[3],
275                                          int origin_set,
276                                          const double rotation[9],
277                                          double out[3])
278 {
279     double shifted[3];
280
281     shifted[0] = raw[0] - (origin_set ? origin[0] : 0.0);
282     shifted[1] = raw[1] - (origin_set ? origin[1] : 0.0);
283     shifted[2] = raw[2] - (origin_set ? origin[2] : 0.0);
284     mat3_vec(rotation, shifted, out);
285 }
286
287 int projection_floor_rotation_from_points(const double p0[3],
288                                           const double p1[3],
289                                           const double p2[3],
290                                           double rotation_out[9])
291 {
292     double a[3];
293     double b[3];
```

```
    double normal[3];
    const double up[3] = {0.0, 0.0, 1.0};

    for (int i = 0; i < 3; i++) {
        a[i] = p1[i] - p0[i];
        b[i] = p2[i] - p0[i];
    }
    cross3(a, b, normal);
    if (normalize3(normal) < 0) {
        return -1;
    }

    rotation_from_to(normal, up, rotation_out);
    return 0;
}

int projection_triangulate_normalized(const point2_t *left,
                                      const point2_t *right,
                                      const double r[9],
                                      const double t[3],
                                      double x_out[3])
{
    double rows[4][4] = {
        {-1.0, 0.0, left->x, 0.0},
        {0.0, -1.0, left->y, 0.0},
        {
            right->x * r[6] - r[0],
            right->x * r[7] - r[1],
            right->x * r[8] - r[2],
            right->x * t[2] - t[0],
        },
        {
            right->y * r[6] - r[3],
            right->y * r[7] - r[4],
            right->y * r[8] - r[5],
            right->y * t[2] - t[1],
        },
    };
    double ata[4][4] = {{0.0}};
    double h[4];

    for (int row = 0; row < 4; row++) {
        for (int i = 0; i < 4; i++) {
            for (int j = 0; j < 4; j++) {
                ata[i][j] += rows[row][i] * rows[row][j];
            }
        }
    }

    jacobi_smallest_eigenvector_4(ata, h);
    if (fabs(h[3]) < 1.0e-12) {
        return -1;
    }

    x_out[0] = h[0] / h[3];
    x_out[1] = h[1] / h[3];
    x_out[2] = h[2] / h[3];
    return 0;
}

int projection_triangulate(const double p_left[12], const double p_right[12],
                           const point2_t *left, const point2_t *right,
                           double x_out[3])
{
    double rows[4][4];
    double ata[4][4] = {{0.0}};
    double h[4];

```

```
    for (int c = 0; c < 4; c++) {
        rows[0][c] = left->x * p_left[8 + c] - p_left[c];
        rows[1][c] = left->y * p_left[8 + c] - p_left[4 + c];
        rows[2][c] = right->x * p_right[8 + c] - p_right[c];
        rows[3][c] = right->y * p_right[8 + c] - p_right[4 + c];
    }

    for (int row = 0; row < 4; row++) {
        for (int i = 0; i < 4; i++) {
            for (int j = 0; j < 4; j++) {
                ata[i][j] += rows[row][i] * rows[row][j];
            }
        }
    }

    jacobi_smallest_eigenvector_4(ata, h);
    if (fabs(h[3]) < 1.0e-12) {
        return -1;
    }

    x_out[0] = h[0] / h[3];
    x_out[1] = h[1] / h[3];
    x_out[2] = h[2] / h[3];
    return 0;
}
```

A.7 stereo.c

```
#include "stereo.h"

#include <errno.h>
#include <stdlib.h>
#include <string.h>

#define DEFAULT_AGC 0
#define DEFAULT_AEC 1
#define DEFAULT_EXPOSURE 0x0100u
#define DEFAULT_GAIN 0xa0u

void camera_settings_default(camera_settings_t *settings)
{
    camera_settings_default_for(CAMERA_A, settings);
}

void camera_settings_default_for(camera_select_t camera,
                                 camera_settings_t *settings)
{
    (void)camera;
    settings->gain = DEFAULT_GAIN;
    settings->exposure = DEFAULT_EXPOSURE;
    settings->gain_ceiling = 0;
    settings->agc = DEFAULT_AGC;
    settings->aec = DEFAULT_AEC;
    settings->exposure_set = 1;
    settings->gain_ceiling_set = 0;
}

int parse_unsigned_arg(const char *text, unsigned max, unsigned *value_out)
{
    char *end = NULL;
    unsigned long value;

    errno = 0;
    value = strtoul(text, &end, 0);
    if (errno || *end || value > max) {
        return -1;
```

```
    }

    *value_out = (unsigned)value;
    return 0;
}

int parse_bool_arg(const char *text, int *value_out)
{
    if (strcmp(text, "1") == 0 || strcmp(text, "on") == 0 ||
        strcmp(text, "ON") == 0 || strcmp(text, "true") == 0 ||
        strcmp(text, "TRUE") == 0 || strcmp(text, "yes") == 0 ||
        strcmp(text, "YES") == 0) {
        *value_out = 1;
        return 0;
    }

    if (strcmp(text, "0") == 0 || strcmp(text, "off") == 0 ||
        strcmp(text, "OFF") == 0 || strcmp(text, "false") == 0 ||
        strcmp(text, "FALSE") == 0 || strcmp(text, "no") == 0 ||
        strcmp(text, "NO") == 0) {
        *value_out = 0;
        return 0;
    }

    return -1;
}

int camera_parse_select(const char *text, camera_select_t *camera)
{
    if (strcmp(text, "A") == 0 || strcmp(text, "a") == 0 ||
        strcmp(text, "0") == 0) {
        *camera = CAMERA_A;
        return 0;
    }
    if (strcmp(text, "B") == 0 || strcmp(text, "b") == 0 ||
        strcmp(text, "1") == 0) {
        *camera = CAMERA_B;
        return 0;
    }
    if (strcmp(text, "both") == 0 || strcmp(text, "BOTH") == 0 ||
        strcmp(text, "all") == 0 || strcmp(text, "ALL") == 0) {
        *camera = CAMERA_BOTH;
        return 0;
    }

    return -1;
}

int camera_parse_single(const char *text, camera_select_t *camera)
{
    if (camera_parse_select(text, camera) < 0) {
        return -1;
    }
    return *camera == CAMERA_A || *camera == CAMERA_B ? 0 : -1;
}

char camera_letter(camera_select_t camera)
{
    return camera == CAMERA_B ? 'B' : 'A';
}

const char *camera_label(camera_select_t camera)
{
    return camera == CAMERA_B ? "camera B" : "camera A";
}
```

A.8 serial_frame_decode.c

```c
#define _DEFAULT_SOURCE

#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>

#define LINE_LEN 256
#define NAME_LEN 256
#define PATH_LEN 512

static int b64_value(int c)
{
    if (c >= 'A' && c <= 'Z') {
        return c - 'A';
    }
    if (c >= 'a' && c <= 'z') {
        return c - 'a' + 26;
    }
    if (c >= '0' && c <= '9') {
        return c - '0' + 52;
    }
    if (c == '+') {
        return 62;
    }
    if (c == '/') {
        return 63;
    }
    if (c == '=') {
        return -2;
    }
    if (c == '\r' || c == '\n') {
        return -3;
    }
    return -1;
}

static int decode_base64_line(const char *line, FILE *out)
{
    int val[4];
    int pad[4];
    unsigned q = 0;

    for (const char *p = line; *p; p++) {
        int v = b64_value((unsigned char)*p);

        if (v == -3) {
            continue;
        }
        if (v < -2) {
            fprintf(stderr, "invalid␣base64␣character␣0x%02x\n",
                    (unsigned char)*p);
            return -1;
        }

        val[q] = v < 0 ? 0 : v;
        pad[q] = v == -2;
        q++;

        if (q == 4) {
            unsigned triple = ((unsigned)val[0] << 18) |
                              ((unsigned)val[1] << 12) |
                              ((unsigned)val[2] << 6) |
                              (unsigned)val[3];
```

```
66
67              fputc((triple >> 16) & 0xff, out);
68              if (!pad[2]) {
69                  fputc((triple >> 8) & 0xff, out);
70              }
71              if (!pad[3]) {
72                  fputc(triple & 0xff, out);
73              }
74              q = 0;
75          }
76      }
77
78      return q == 0 ? 0 : -1;
79  }
80
81  static const char *base_name(const char *path)
82  {
83      const char *slash = strrchr(path, '/');
84      return slash ? slash + 1 : path;
85  }
86
87  int main(int argc, char **argv)
88  {
89      const char *log_path;
90      const char *out_dir = ".";
91      FILE *log;
92      FILE *out = NULL;
93      char line[LINE_LEN];
94      unsigned frames = 0;
95
96      if (argc < 2 || argc > 3) {
97          fprintf(stderr, "usage: %s screenlog.0 [output_dir]\n", argv[0]);
98          return 2;
99      }
100
101     log_path = argv[1];
102     if (argc == 3) {
103         out_dir = argv[2];
104         if (mkdir(out_dir, 0777) < 0) {
105             struct stat st;
106
107             if (stat(out_dir, &st) < 0 || !S_ISDIR(st.st_mode)) {
108                 perror("mkdir output_dir");
109                 return 1;
110             }
111         }
112     }
113
114     log = fopen(log_path, "rb");
115     if (!log) {
116         perror("fopen log");
117         return 1;
118     }
119
120     while (fgets(line, sizeof(line), log)) {
121         if (!out) {
122             unsigned index;
123             char name[NAME_LEN];
124
125             if (sscanf(line, "BEGIN_FRAME %u %255s", &index, name) == 2) {
126                 char path[PATH_LEN];
127                 int n = snprintf(path, sizeof(path), "%s/%s", out_dir,
128                                  base_name(name));
129
130                 if (n < 0 || (size_t)n >= sizeof(path)) {
131                     fprintf(stderr, "output path too long\n");
132                     fclose(log);
133                     return 1;
```

```
                }

                out = fopen(path, "wb");
                if (!out) {
                    perror("fopen output");
                    fclose(log);
                    return 1;
                }
                printf("decoding frame %u -> %s\n", index, path);
            }
            continue;
        }

        if (strncmp(line, "END_FRAME", 9) == 0) {
            fclose(out);
            out = NULL;
            frames++;
            continue;
        }

        if (decode_base64_line(line, out) < 0) {
            fprintf(stderr, "base64 decode failed\n");
            fclose(out);
            fclose(log);
            return 1;
        }
    }

    if (out) {
        fprintf(stderr, "log ended inside a frame block\n");
        fclose(out);
        fclose(log);
        return 1;
    }

    fclose(log);
    printf("decoded %u frame(s)\n", frames);

    if (frames > 0 && unlink(log_path) < 0) {
        perror("unlink log");
        return 1;
    }

    return 0;
}
```

B Hardware Source Appendix

The following listings are the SystemVerilog hardware files included with the hardware submission. They show the camera moment pipeline, frame-buffer debug path, accumulator logic, pixel-coordinate decoder, FPGA-side image processor module, and top-level DE1-SoC integration wrapper.

B.1 camera_moment_pipeline.sv

```
module camera_moment_pipeline #(
   parameter int FRAME_WIDTH = 640,
   parameter int FRAME_HEIGHT = 480
) (
   input logic clk,
   input logic reset,
   input logic clear_req_toggle,
   input logic [7:0] threshold_cfg,

   input logic pclk,
   input logic href,
   input logic vsync,
   input logic [7:0] data,

   output logic [31:0] area_result,
   output logic [31:0] u_result,
   output logic [31:0] v_result,
   output logic done,
   output logic active
);

   logic [2:0] clear_req_sync; //3-bit synchronizer
   logic clear_seen; //Edge detection on clear toggle from 3-bit synchronizer (pclk)
   logic clear_req_toggle_d; //Delayed clear_req_toggle for edge detection (clk)
   logic [7:0] threshold_pending; //Threshold value pending for next frame (CDC)
   logic threshold_update_sync;
   logic [7:0] threshold_active; //Active threshold, latched at frame start (pclk)

   logic pixel_valid;
   logic [7:0] pixel_data;
   logic [9:0] u_coord;
   logic [8:0] v_coord;
   logic frame_start_pulse;
   logic frame_done_pulse;
   logic foreground;

   logic [31:0] area_result_pclk;
   logic [31:0] u_result_pclk;
   logic [31:0] v_result_pclk;
   logic done_pclk;
   logic done_sync;
   logic done_sync_d;

   assign clear_seen = clear_req_sync[2] ^ clear_req_sync[1]; //One-cycle pulse on edge

   always_ff @(posedge clk or posedge reset) begin
      if (reset) begin
         clear_req_toggle_d <= 1'b0;
         threshold_pending <= 8'd0;
      end else begin
         clear_req_toggle_d <= clear_req_toggle;
         if (clear_req_toggle_d != clear_req_toggle) begin
            threshold_pending <= threshold_cfg; //Update pending threshold in clk domain on done clear toggle
         end
      end
   end
```

```
57
58    always_ff @(posedge pclk or posedge reset) begin
59       if (reset) begin
60          clear_req_sync <= 3'b000;
61          threshold_update_sync <= 1'b0;
62          threshold_active <= 8'd0;
63       end else begin
64          clear_req_sync <= { clear_req_sync[1:0], clear_req_toggle }; //Update 3-bit sync
65          if (clear_seen) begin //Update active pclk threshold register on frame start if done cleared
66             threshold_update_sync <= 1'b1; //Arm
67          end
68          if (frame_start_pulse) begin
69             if (threshold_update_sync || clear_seen) begin //Set active threshold if armed or incoming
     clear_seen (latter is scary)
70                threshold_active <= threshold_pending;
71                threshold_update_sync <= 1'b0;
72             end
73          end
74       end
75    end
76
77    pixel_coordinate_decoder #(
78       .FRAME_WIDTH ( FRAME_WIDTH ),
79       .FRAME_HEIGHT ( FRAME_HEIGHT )
80    ) coordinate_decoder (
81       .pclk ( pclk ),
82       .reset ( reset ),
83       .href ( href ),
84       .vsync ( vsync ),
85       .data ( data ),
86       .pixel_valid ( pixel_valid ),
87       .pixel_data ( pixel_data ),
88       .u_coord ( u_coord ),
89       .v_coord ( v_coord ),
90       .frame_start ( frame_start_pulse ),
91       .frame_done ( frame_done_pulse ),
92       .active ( active )
93    );
94
95    assign foreground = pixel_valid && pixel_data >= threshold_active;
96
97    moment_accumulators accumulator_bank (
98       .pclk ( pclk ),
99       .reset ( reset ),
100      .clear ( clear_seen ),
101      .pixel_valid ( pixel_valid ),
102      .foreground ( foreground ),
103      .u_coord ( u_coord ),
104      .v_coord ( v_coord ),
105      .frame_start ( frame_start_pulse ),
106      .frame_done ( frame_done_pulse ),
107      .area_result_pclk ( area_result_pclk ),
108      .u_result_pclk ( u_result_pclk ),
109      .v_result_pclk ( v_result_pclk ),
110      .done_pclk ( done_pclk )
111   );
112
113   always_ff @(posedge clk or posedge reset) begin
114      if (reset) begin
115         done <= 1'b0;
116         done_sync <= 1'b0;
117         done_sync_d <= 1'b0;
118         area_result <= 32'd0;
119         u_result <= 32'd0;
120         v_result <= 32'd0;
121      end else begin
122         done_sync <= done_pclk;
123         done <= done_sync;
```

```
124         done_sync_d <= done_sync;
125
126         if (done_sync && !done_sync_d) begin
127            area_result <= area_result_pclk;
128            u_result <= u_result_pclk;
129            v_result <= v_result_pclk;
130         end
131      end
132   end
133
134endmodule
```

B.2 frame_buffer.sv

```
1  module frame_buffer #(
2     parameter int FRAME_WIDTH = 640,
3     parameter int FRAME_HEIGHT = 480
4  ) (
5     input logic clk,
6     input logic reset,
7     input logic clear_req_toggle,
8     input logic enable,
9
10    input logic pclk,
11    input logic href,
12    input logic vsync,
13    input logic [7:0] data,
14
15    input logic index_write,
16    input logic [31:0] index_writedata,
17    input logic [31:0] frame_index,
18    output logic [31:0] frame_data,
19    output logic done,
20    output logic active
21 );
22
23    localparam int FRAME_PIXELS = FRAME_WIDTH * FRAME_HEIGHT;
24    localparam int FRAME_WORDS = FRAME_PIXELS / 4;
25    localparam int FRAME_WORD_BITS = $clog2(FRAME_WORDS);
26    localparam int FRAME_PIXEL_BITS = $clog2(FRAME_PIXELS);
27    localparam logic [FRAME_PIXEL_BITS-1:0] LAST_FRAME_PIXEL =
28       FRAME_PIXEL_BITS'(FRAME_PIXELS - 1);
29
30    typedef enum logic [1:0] {
31       STATE_WAIT_FRAME,
32       STATE_CAPTURE
33    } capture_state_t;
34
35    logic frame_index_valid;
36    logic new_index_valid;
37    logic [FRAME_WORD_BITS-1:0] frame_rd_addr;
38    logic [FRAME_WORD_BITS-1:0] new_index_addr;
39    logic [FRAME_WORD_BITS-1:0] frame_ram_rd_addr;
40    logic [31:0] frame_ram_wr_data;
41    logic frame_ram_wren;
42
43    (* ramstyle = "M10K" *) logic [31:0] frame_ram [0:FRAME_WORDS-1]; //Dual clock two-port RAM
44
45    logic [2:0] clear_req_sync; //3-bit synchronizer
46    logic clear_seen;
47    logic done_pclk;
48    logic done_sync;
49    logic enable_sync;
50    logic capture_enable_current;
51    logic capture_open_current;
52    capture_state_t capture_state;
```

```
53    logic vsync_d;
54    logic [1:0] yuv_byte_phase;
55    logic [1:0] pixel_pack_phase;
56    logic [23:0] pixel_pack_word; //3B packing + 1B new data to fit 4B RAM line length
57    logic [FRAME_WORD_BITS-1:0] frame_wr_addr;
58    logic [FRAME_PIXEL_BITS-1:0] captured_pixels;
59
60    assign clear_seen = clear_req_sync[2] ^ clear_req_sync[1]; //One-cycle pulse on edge
61    assign frame_index_valid = frame_index < FRAME_WORDS; //Read index within RAM range?
62    assign frame_rd_addr = frame_index_valid ? frame_index[FRAME_WORD_BITS-1:0] : '0; //RAM read address
63    assign new_index_valid = index_writedata < FRAME_WORDS; //Incoming SW-set read index within RAM range?
64    assign new_index_addr = new_index_valid ? index_writedata[FRAME_WORD_BITS-1:0] : '0; //New RAM read address
65    assign frame_ram_rd_addr = index_write ? new_index_addr : frame_rd_addr; //Use new SW-set address if index
        update flag set
66    assign frame_ram_wr_data = { data, pixel_pack_word }; //Packed pixel data
67    assign frame_ram_wren =
68       enable_sync && capture_state == STATE_CAPTURE && !vsync && href &&
69       !yuv_byte_phase[0] && pixel_pack_phase == 2'd3; //Write enable: Write in active rows every fourth Y
        pixel for correct packing
70    assign capture_enable_current = clear_seen ? enable : enable_sync; //Use SW-set enable directly on DONE
        clear for immediate re-arm
71    assign capture_open_current = !done_pclk || clear_seen; //Block new capture if framebuffer occupied
72
73    always_ff @(posedge pclk or posedge reset) begin
74       if (reset) begin
75          clear_req_sync <= 3'b000;
76          done_pclk <= 1'b0;
77          enable_sync <= 1'b0;
78          capture_state <= STATE_WAIT_FRAME;
79          vsync_d <= 1'b0;
80          yuv_byte_phase <= 2'd0;
81          pixel_pack_phase <= 2'd0;
82          pixel_pack_word <= 24'd0;
83          frame_wr_addr <= '0;
84          captured_pixels <= '0;
85       end else begin
86          clear_req_sync <= { clear_req_sync[1:0], clear_req_toggle };
87          vsync_d <= vsync;
88
89          if (clear_seen) begin
90             done_pclk <= 1'b0;
91             if (capture_state != STATE_CAPTURE) begin
92                enable_sync <= enable;
93             end
94          end
95
96          case (capture_state)
97             STATE_WAIT_FRAME: begin
98                yuv_byte_phase <= 2'd0;
99
100               if (capture_enable_current && capture_open_current && vsync_d && !vsync) begin //Falling edge
        vsync + capture armed and not blocked
101                  capture_state <= STATE_CAPTURE;
102                  pixel_pack_phase <= 2'd0;
103                  pixel_pack_word <= 24'd0;
104                  frame_wr_addr <= '0;
105                  captured_pixels <= '0;
106               end
107            end
108
109            STATE_CAPTURE: begin
110               if (vsync) begin //Early vsync: go back to wait
111                  capture_state <= STATE_WAIT_FRAME;
112                  yuv_byte_phase <= 2'd0;
113               end else if (href) begin
114                  yuv_byte_phase <= yuv_byte_phase + 2'd1;
115
116                  if (!yuv_byte_phase[0]) begin //Byte is Y
```

```
                    case (pixel_pack_phase)
                       2'd0: pixel_pack_word[7:0] <= data;
                       2'd1: pixel_pack_word[15:8] <= data;
                       2'd2: pixel_pack_word[23:16] <= data;
                       default: begin
                          frame_wr_addr <= frame_wr_addr + 1'b1; //2'd3 case. Current byte is concatenated directly to RAM writedata
                       end
                    endcase

                    pixel_pack_phase <= pixel_pack_phase + 2'd1;

                    if (captured_pixels == LAST_FRAME_PIXEL) begin
                       done_pclk <= 1'b1;
                       capture_state <= STATE_WAIT_FRAME;
                    end else begin
                       captured_pixels <= captured_pixels + 1'b1;
                    end
                 end
              end else begin
                 yuv_byte_phase <= 2'd0;
              end
           end

           default: begin
              capture_state <= STATE_WAIT_FRAME;
           end
        endcase
     end
  end

  always_ff @(posedge clk or posedge reset) begin
     if (reset) begin
        done <= 1'b0;
        done_sync <= 1'b0;
     end else begin
        done_sync <= done_pclk;
        done <= done_sync;
     end
  end

  always_ff @(posedge clk) begin
     frame_data <= frame_ram[frame_ram_rd_addr];
  end

  always_ff @(posedge pclk) begin
     if (frame_ram_wren) begin
        frame_ram[frame_wr_addr] <= frame_ram_wr_data;
     end
  end

  assign active = capture_state == STATE_CAPTURE;

endmodule
```

B.3 imgproc.sv

```
/*
 * Top-level image processing peripheral.
 *
 * Register map, word addressed:
 *
 * Address  Register  Access  Meaning
 *   0      AREA_A    R       Camera A foreground area
 *   1      U_A       R       Camera A foreground u/x moment sum
 *   2      V_A       R       Camera A foreground v/y moment sum
```

```
 *   3      AREA_B    R       Camera B foreground area
 *   4      U_B       R       Camera B foreground u/x moment sum
 *   5      V_B       R       Camera B foreground v/y moment sum
 *   6      DONE      R/W     bit 0: A, bit 1: B, bit 2: frame buffer
 *   7      CONTROL   R/W     [1:0] frame source: 0 none, 1 A, 2 B
 *                            [15:8] threshold A, [23:16] threshold B
 *   8      FB_INDEX  R/W     Frame buffer word index, 0 .. 76799
 *   9      FB_DATA   R       Four pixels packed as {p3, p2, p1, p0}
 */

module imgproc(
   input logic clk,
   input logic reset,

   input logic [31:0] writedata,
   input logic write,
   input logic chipselect,
   input logic [3:0] address,

   output logic [31:0] readdata,

   input logic a_pclk,
   input logic a_href,
   input logic a_vsync,
   input logic [7:0] a_data,
   output logic a_xclk,

   input logic b_pclk,
   input logic b_href,
   input logic b_vsync,
   input logic [7:0] b_data,
   output logic b_xclk,

   output logic [9:0] leds
);

   localparam int FRAME_WIDTH = 640;
   localparam int FRAME_HEIGHT = 480;

   typedef enum logic [3:0] {
      REG_AREA_A = 4'd0,
      REG_U_A = 4'd1,
      REG_V_A = 4'd2,
      REG_AREA_B = 4'd3,
      REG_U_B = 4'd4,
      REG_V_B = 4'd5,
      REG_DONE = 4'd6,
      REG_CONTROL = 4'd7,
      REG_FB_INDEX = 4'd8,
      REG_FB_DATA = 4'd9
   } register_address_t;

   typedef enum logic [1:0] {
      STORE_NONE = 2'd0,
      STORE_A = 2'd1,
      STORE_B = 2'd2
   } frame_store_select_t;

   logic clk25;
   logic [31:0] control;
   logic [31:0] frame_index;
   frame_store_select_t frame_store_select;

   logic a_clear_toggle;
   logic b_clear_toggle;
   logic frame_clear_toggle;

   logic [31:0] a_area;
```

```
78    logic [31:0] a_u_sum;
79    logic [31:0] a_v_sum;
80    logic a_done;
81    logic a_active;
82
83    logic [31:0] b_area;
84    logic [31:0] b_u_sum;
85    logic [31:0] b_v_sum;
86    logic b_done;
87    logic b_active;
88
89    logic frame_index_write;
90    logic [31:0] frame_data;
91    logic frame_done;
92    logic frame_active;
93
94    logic selected_pclk;
95    logic selected_href;
96    logic selected_vsync;
97    logic [7:0] selected_data;
98    logic frame_store_enable;
99    register_address_t register_address;
100
101   assign register_address = register_address_t'(address);
102   assign frame_index_write = chipselect && write && (register_address == REG_FB_INDEX); //Combinational RAM
        read index update flag to avoid one-cycle delay
103   assign frame_store_enable = frame_store_select != STORE_NONE;
104
105   function automatic frame_store_select_t decode_frame_store(input logic [1:0] value);
106      case (value)
107         STORE_A: decode_frame_store = STORE_A;
108         STORE_B: decode_frame_store = STORE_B;
109         default: decode_frame_store = STORE_NONE;
110      endcase
111   endfunction
112
113   always_comb begin
114      case (frame_store_select)
115         STORE_A: begin
116            selected_pclk = a_pclk;
117            selected_href = a_href;
118            selected_vsync = a_vsync;
119            selected_data = a_data;
120         end
121
122         STORE_B: begin
123            selected_pclk = b_pclk;
124            selected_href = b_href;
125            selected_vsync = b_vsync;
126            selected_data = b_data;
127         end
128
129         default: begin
130            selected_pclk = a_pclk;
131            selected_href = a_href;
132            selected_vsync = a_vsync;
133            selected_data = a_data;
134         end
135      endcase
136   end
137
138   always_ff @(posedge clk or posedge reset) begin
139      if (reset) begin
140         clk25 <= 1'b0;
141         control <= 32'd0;
142         frame_index <= 32'd0;
143         frame_store_select <= STORE_NONE;
144         a_clear_toggle <= 1'b0;
```

```
145         b_clear_toggle <= 1'b0;
146         frame_clear_toggle <= 1'b0;
147      end else begin
148         clk25 <= ~clk25;
149
150         if (chipselect && write) begin
151            case (register_address)
152               REG_CONTROL: begin
153                  control <= writedata;
154               end
155
156               REG_DONE: begin
157                  if (!writedata[0]) begin
158                     a_clear_toggle <= ~a_clear_toggle; //Generate toggle flags for done clear
159                  end
160                  if (!writedata[1]) begin
161                     b_clear_toggle <= ~b_clear_toggle;
162                  end
163                  if (!writedata[2]) begin
164                     frame_store_select <= decode_frame_store(control[1:0]);
165                     frame_clear_toggle <= ~frame_clear_toggle;
166                  end
167               end
168
169               REG_FB_INDEX: begin
170                  frame_index <= writedata;
171               end
172
173               default: begin
174               end
175            endcase
176         end
177      end
178   end
179
180   camera_moment_pipeline #(
181      .FRAME_WIDTH ( FRAME_WIDTH ),
182      .FRAME_HEIGHT ( FRAME_HEIGHT )
183   ) camera_a_moments (
184      .clk ( clk ),
185      .reset ( reset ),
186      .clear_req_toggle ( a_clear_toggle ),
187      .threshold_cfg ( control[15:8] ),
188      .pclk ( a_pclk ),
189      .href ( a_href ),
190      .vsync ( a_vsync ),
191      .data ( a_data ),
192      .area_result ( a_area ),
193      .u_result ( a_u_sum ),
194      .v_result ( a_v_sum ),
195      .done ( a_done ),
196      .active ( a_active )
197   );
198
199   camera_moment_pipeline #(
200      .FRAME_WIDTH ( FRAME_WIDTH ),
201      .FRAME_HEIGHT ( FRAME_HEIGHT )
202   ) camera_b_moments (
203      .clk ( clk ),
204      .reset ( reset ),
205      .clear_req_toggle ( b_clear_toggle ),
206      .threshold_cfg ( control[23:16] ),
207      .pclk ( b_pclk ),
208      .href ( b_href ),
209      .vsync ( b_vsync ),
210      .data ( b_data ),
211      .area_result ( b_area ),
212      .u_result ( b_u_sum ),
```

```
      .v_result ( b_v_sum ),
      .done ( b_done ),
      .active ( b_active )
   );

   frame_buffer #(
      .FRAME_WIDTH ( FRAME_WIDTH ),
      .FRAME_HEIGHT ( FRAME_HEIGHT )
   ) debug_frame_buffer (
      .clk ( clk ),
      .reset ( reset ),
      .clear_req_toggle ( frame_clear_toggle ),
      .enable ( frame_store_enable ),
      .pclk ( selected_pclk ),
      .href ( selected_href ),
      .vsync ( selected_vsync ),
      .data ( selected_data ),
      .index_write ( frame_index_write ),
      .index_writedata ( writedata ),
      .frame_index ( frame_index ),
      .frame_data ( frame_data ),
      .done ( frame_done ),
      .active ( frame_active )
   );

   always_comb begin
      case (register_address)
         REG_AREA_A: readdata = a_area;
         REG_U_A: readdata = a_u_sum;
         REG_V_A: readdata = a_v_sum;
         REG_AREA_B: readdata = b_area;
         REG_U_B: readdata = b_u_sum;
         REG_V_B: readdata = b_v_sum;
         REG_DONE: readdata = { 29'd0, frame_done, b_done, a_done };
         REG_CONTROL: readdata = control;
         REG_FB_INDEX: readdata = frame_index;
         REG_FB_DATA: readdata = frame_data;
         default: readdata = 32'd0;
      endcase
   end

   assign a_xclk = clk25;
   assign b_xclk = clk25;
   assign leds = {
      1'b0,
      !reset,
      frame_store_select,
      frame_active,
      b_active,
      a_active,
      frame_done,
      b_done,
      a_done
   };

endmodule
```

B.4 moment_accumulators.sv

```
module moment_accumulators(
   input logic pclk,
   input logic reset,
   input logic clear,
   input logic pixel_valid,
   input logic foreground,
   input logic [9:0] u_coord,
```

```
   input logic [8:0] v_coord,
   input logic frame_start,
   input logic frame_done,

   output logic [31:0] area_result_pclk,
   output logic [31:0] u_result_pclk,
   output logic [31:0] v_result_pclk,
   output logic done_pclk
);

   logic [31:0] area_accum;
   logic [31:0] u_accum;
   logic [31:0] v_accum;
   logic [31:0] area_next;
   logic [31:0] u_next;
   logic [31:0] v_next;
   logic publish_this_frame;

   assign area_next = area_accum + ((pixel_valid && foreground) ? 32'd1 : 32'd0);
   assign u_next = u_accum + ((pixel_valid && foreground) ? { 22'd0, u_coord } : 32'd0);
   assign v_next = v_accum + ((pixel_valid && foreground) ? { 23'd0, v_coord } : 32'd0);

   always_ff @(posedge pclk or posedge reset) begin
      if (reset) begin
         area_accum <= 32'd0;
         u_accum <= 32'd0;
         v_accum <= 32'd0;
         area_result_pclk <= 32'd0;
         u_result_pclk <= 32'd0;
         v_result_pclk <= 32'd0;
         done_pclk <= 1'b0;
         publish_this_frame <= 1'b0;
      end else begin
         if (clear) begin
            done_pclk <= 1'b0;
         end

         if (frame_start) begin
            area_accum <= 32'd0;
            u_accum <= 32'd0;
            v_accum <= 32'd0;
            publish_this_frame <= !done_pclk || clear;
         end else begin
            area_accum <= area_next;
            u_accum <= u_next;
            v_accum <= v_next;

            if (frame_done) begin
               if (publish_this_frame) begin
                  area_result_pclk <= area_next;
                  u_result_pclk <= u_next;
                  v_result_pclk <= v_next;
                  done_pclk <= 1'b1;
               end
            end
         end
      end
   end

endmodule
```

B.5 pixel_coordinate_decoder.sv

```
module pixel_coordinate_decoder #(
   parameter int FRAME_WIDTH = 640,
   parameter int FRAME_HEIGHT = 480
```

```
) (
   input logic pclk,
   input logic reset,
   input logic href,
   input logic vsync,
   input logic [7:0] data,

   output logic pixel_valid, //Pixel is Y
   output logic [7:0] pixel_data,
   output logic [9:0] u_coord,
   output logic [8:0] v_coord,
   output logic frame_start, //One-cycle start pulse
   output logic frame_done, //One-cycle done pulse
   output logic active
);

   localparam int FRAME_PIXELS = FRAME_WIDTH * FRAME_HEIGHT;
   localparam int FRAME_PIXEL_BITS = $clog2(FRAME_PIXELS);
   localparam logic [FRAME_PIXEL_BITS-1:0] LAST_FRAME_PIXEL =
      FRAME_PIXEL_BITS'(FRAME_PIXELS - 1);
   localparam logic [9:0] LAST_U_COORD = 10'(FRAME_WIDTH - 1);
   localparam logic [8:0] LAST_V_COORD = 9'(FRAME_HEIGHT - 1);

   typedef enum logic [1:0] {
      STATE_WAIT_FRAME,
      STATE_CAPTURE
   } capture_state_t;

   capture_state_t capture_state;
   logic vsync_d; //Delayed vsync for falling edge detection
   logic [1:0] yuv_byte_phase;
   logic [9:0] u_count;
   logic [8:0] v_count;
   logic [FRAME_PIXEL_BITS-1:0] pixel_count;
   logic current_is_y;

   assign current_is_y = (capture_state == STATE_CAPTURE) && href && !yuv_byte_phase[0]; //YUYV unpacking

   always_ff @(posedge pclk or posedge reset) begin
      if (reset) begin
         capture_state <= STATE_WAIT_FRAME;
         vsync_d <= 1'b0;
         yuv_byte_phase <= 2'd0;
         u_count <= 10'd0;
         v_count <= 9'd0;
         pixel_count <= '0;
         pixel_valid <= 1'b0;
         pixel_data <= 8'd0;
         u_coord <= 10'd0;
         v_coord <= 9'd0;
         frame_start <= 1'b0;
         frame_done <= 1'b0;
      end else begin
         vsync_d <= vsync;
         pixel_valid <= current_is_y;
         pixel_data <= data;
         u_coord <= u_count;
         v_coord <= v_count;
         frame_start <= 1'b0;
         frame_done <= 1'b0;

         case (capture_state)
            STATE_WAIT_FRAME: begin
               yuv_byte_phase <= 2'd0;
               u_count <= 10'd0;
               v_count <= 9'd0;
               pixel_count <= '0;
               pixel_valid <= 1'b0;
```

```
72
73              if (vsync_d && !vsync) begin //Start capture on falling edge
74                 capture_state <= STATE_CAPTURE;
75                 frame_start <= 1'b1;
76              end
77           end
78
79           STATE_CAPTURE: begin
80              if (vsync) begin //Early vsync: go back to wait
81                 capture_state <= STATE_WAIT_FRAME;
82                 yuv_byte_phase <= 2'd0;
83                 u_count <= 10'd0;
84                 v_count <= 9'd0;
85                 pixel_count <= '0;
86                 pixel_valid <= 1'b0;
87              end else if (href) begin
88                 yuv_byte_phase <= yuv_byte_phase + 2'd1;
89
90                 if (current_is_y) begin
91                    if (pixel_count == LAST_FRAME_PIXEL) begin
92                       frame_done <= 1'b1;
93                       capture_state <= STATE_WAIT_FRAME;
94                    end else begin
95                       pixel_count <= pixel_count + 1'b1;
96                    end
97
98                    if (u_count == LAST_U_COORD) begin
99                       u_count <= 10'd0;
100                      if (v_count != LAST_V_COORD) begin
101                         v_count <= v_count + 1'b1;
102                      end
103                   end else begin
104                      u_count <= u_count + 1'b1;
105                   end
106                end
107             end else begin
108                yuv_byte_phase <= 2'd0;
109             end
110          end
111
112          default: begin
113             capture_state <= STATE_WAIT_FRAME;
114          end
115       endcase
116    end
117 end
118
119 assign active = capture_state == STATE_CAPTURE;
120
121 endmodule
```

B.6 soc_system_top.sv

```
1  // ====================================================================
2  // Copyright (c) 2013 by Terasic Technologies Inc.
3  // ====================================================================
4  //
5  // Modified 2019 by Stephen A. Edwards
6  //
7  // Permission:
8  //
9  //   Terasic grants permission to use and modify this code for use
10 //   in synthesis for all Terasic Development Boards and Altera
11 //   Development Kits made by Terasic.  Other use of this code,
12 //   including the selling ,duplication, or modification of any
13 //   portion is strictly prohibited.
```

```
//
// Disclaimer:
//
//   This VHDL/Verilog or C/C++ source code is intended as a design
//   reference which illustrates how these types of functions can be
//   implemented.  It is the user's responsibility to verify their
//   design for consistency and functionality through the use of
//   formal verification methods.  Terasic provides no warranty
//   regarding the use or functionality of this code.
//
// ============================================================
//
//  Terasic Technologies Inc

//  9F., No.176, Sec.2, Gongdao 5th Rd, East Dist, Hsinchu City, 30070. Taiwan
//
//
//                     web: http://www.terasic.com/
//                     email: support@terasic.com
module soc_system_top(

 ///////// ADC /////////
 inout         ADC_CS_N,
 output        ADC_DIN,
 input         ADC_DOUT,
 output        ADC_SCLK,

 ///////// AUD /////////
 input         AUD_ADCDAT,
 inout         AUD_ADCLRCK,
 inout         AUD_BCLK,
 output        AUD_DACDAT,
 inout         AUD_DACLRCK,
 output        AUD_XCK,

 ///////// CLOCK2 /////////
 input         CLOCK2_50,

 ///////// CLOCK3 /////////
 input         CLOCK3_50,

 ///////// CLOCK4 /////////
 input         CLOCK4_50,

 ///////// CLOCK /////////
 input         CLOCK_50,

 ///////// DRAM /////////
 output [12:0] DRAM_ADDR,
 output [1:0]  DRAM_BA,
 output        DRAM_CAS_N,
 output        DRAM_CKE,
 output        DRAM_CLK,
 output        DRAM_CS_N,
 inout [15:0]  DRAM_DQ,
 output        DRAM_LDQM,
 output        DRAM_RAS_N,
 output        DRAM_UDQM,
 output        DRAM_WE_N,

 ///////// FAN /////////
 output        FAN_CTRL,

 ///////// FPGA /////////
 output        FPGA_I2C_SCLK,
 inout         FPGA_I2C_SDAT,

 ///////// GPIO /////////
```

```
inout [35:0]  GPIO_0,
inout [35:0]  GPIO_1,

///////// HEX0 /////////
output [6:0]  HEX0,

///////// HEX1 /////////
output [6:0]  HEX1,

///////// HEX2 /////////
output [6:0]  HEX2,

///////// HEX3 /////////
output [6:0]  HEX3,

///////// HEX4 /////////
output [6:0]  HEX4,

///////// HEX5 /////////
output [6:0]  HEX5,

///////// HPS /////////
inout         HPS_CONV_USB_N,
output [14:0] HPS_DDR3_ADDR,
output [2:0]  HPS_DDR3_BA,
output        HPS_DDR3_CAS_N,
output        HPS_DDR3_CKE,
output        HPS_DDR3_CK_N,
output        HPS_DDR3_CK_P,
output        HPS_DDR3_CS_N,
output [3:0]  HPS_DDR3_DM,
inout [31:0]  HPS_DDR3_DQ,
inout [3:0]   HPS_DDR3_DQS_N,
inout [3:0]   HPS_DDR3_DQS_P,
output        HPS_DDR3_ODT,
output        HPS_DDR3_RAS_N,
output        HPS_DDR3_RESET_N,
input         HPS_DDR3_RZQ,
output        HPS_DDR3_WE_N,
output        HPS_ENET_GTX_CLK,
inout         HPS_ENET_INT_N,
output        HPS_ENET_MDC,
inout         HPS_ENET_MDIO,
input         HPS_ENET_RX_CLK,
input [3:0]   HPS_ENET_RX_DATA,
input         HPS_ENET_RX_DV,
output [3:0]  HPS_ENET_TX_DATA,
output        HPS_ENET_TX_EN,
inout         HPS_GSENSOR_INT,
inout         HPS_I2C1_SCLK,
inout         HPS_I2C1_SDAT,
inout         HPS_I2C2_SCLK,
inout         HPS_I2C2_SDAT,
inout         HPS_I2C_CONTROL,
inout         HPS_KEY,
inout         HPS_LED,
inout         HPS_LTC_GPIO,
output        HPS_SD_CLK,
inout         HPS_SD_CMD,
inout [3:0]   HPS_SD_DATA,
output        HPS_SPIM_CLK,
input         HPS_SPIM_MISO,
output        HPS_SPIM_MOSI,
inout         HPS_SPIM_SS,
input         HPS_UART_RX,
output        HPS_UART_TX,
input         HPS_USB_CLKOUT,
inout [7:0]   HPS_USB_DATA,
```

```
 input         HPS_USB_DIR,
 input         HPS_USB_NXT,
 output        HPS_USB_STP,

 ///////// IRDA /////////
 input         IRDA_RXD,
 output        IRDA_TXD,

 ///////// KEY /////////
 input [3:0]   KEY,

 ///////// LEDR /////////
 output [9:0]  LEDR,

 ///////// PS2 /////////
 inout         PS2_CLK,
 inout         PS2_CLK2,
 inout         PS2_DAT,
 inout         PS2_DAT2,

 ///////// SW /////////
 input [9:0]   SW,

 ///////// TD /////////
 input         TD_CLK27,
 input [7:0]   TD_DATA,
 input         TD_HS,
 output        TD_RESET_N,
 input         TD_VS
);

   soc_system soc_system0(
     .clk_clk                       ( CLOCK_50 ),
     .reset_reset_n                 ( KEY[0] ),

     .gpio1_i2c_export              ( GPIO_1[25:22] ),
     .gpio1_upper_export            ( { GPIO_1[27], GPIO_1[35:28] } ),
     .gpio0_i2c_export              ( GPIO_0[25:22] ),
     .gpio0_upper_export            ( { GPIO_0[27], GPIO_0[35:28] } ),

     .hps_ddr3_mem_a                ( HPS_DDR3_ADDR ),
     .hps_ddr3_mem_ba               ( HPS_DDR3_BA ),
     .hps_ddr3_mem_ck               ( HPS_DDR3_CK_P ),
     .hps_ddr3_mem_ck_n             ( HPS_DDR3_CK_N ),
     .hps_ddr3_mem_cke              ( HPS_DDR3_CKE ),
     .hps_ddr3_mem_cs_n             ( HPS_DDR3_CS_N ),
     .hps_ddr3_mem_ras_n            ( HPS_DDR3_RAS_N ),
     .hps_ddr3_mem_cas_n            ( HPS_DDR3_CAS_N ),
     .hps_ddr3_mem_we_n             ( HPS_DDR3_WE_N ),
     .hps_ddr3_mem_reset_n          ( HPS_DDR3_RESET_N ),
     .hps_ddr3_mem_dq               ( HPS_DDR3_DQ ),
     .hps_ddr3_mem_dqs              ( HPS_DDR3_DQS_P ),
     .hps_ddr3_mem_dqs_n            ( HPS_DDR3_DQS_N ),
     .hps_ddr3_mem_odt              ( HPS_DDR3_ODT ),
     .hps_ddr3_mem_dm               ( HPS_DDR3_DM ),
     .hps_ddr3_oct_rzqin            ( HPS_DDR3_RZQ ),

     .hps_hps_io_emac1_inst_TX_CLK ( HPS_ENET_GTX_CLK ),
     .hps_hps_io_emac1_inst_TXD0   ( HPS_ENET_TX_DATA[0] ),
     .hps_hps_io_emac1_inst_TXD1   ( HPS_ENET_TX_DATA[1] ),
     .hps_hps_io_emac1_inst_TXD2   ( HPS_ENET_TX_DATA[2] ),
     .hps_hps_io_emac1_inst_TXD3   ( HPS_ENET_TX_DATA[3] ),
     .hps_hps_io_emac1_inst_RXD0   ( HPS_ENET_RX_DATA[0] ),
     .hps_hps_io_emac1_inst_MDIO   ( HPS_ENET_MDIO  ),
     .hps_hps_io_emac1_inst_MDC    ( HPS_ENET_MDC   ),
     .hps_hps_io_emac1_inst_RX_CTL ( HPS_ENET_RX_DV ),
     .hps_hps_io_emac1_inst_TX_CTL ( HPS_ENET_TX_EN ),
     .hps_hps_io_emac1_inst_RX_CLK ( HPS_ENET_RX_CLK ),
```

```
218      .hps_hps_io_emac1_inst_RXD1   ( HPS_ENET_RX_DATA[1]  ),
219      .hps_hps_io_emac1_inst_RXD2   ( HPS_ENET_RX_DATA[2]  ),
220      .hps_hps_io_emac1_inst_RXD3   ( HPS_ENET_RX_DATA[3]  ),
221
222      .hps_hps_io_sdio_inst_CMD     ( HPS_SD_CMD          ),
223      .hps_hps_io_sdio_inst_D0      ( HPS_SD_DATA[0]      ),
224      .hps_hps_io_sdio_inst_D1      ( HPS_SD_DATA[1]      ),
225      .hps_hps_io_sdio_inst_CLK     ( HPS_SD_CLK          ),
226      .hps_hps_io_sdio_inst_D2      ( HPS_SD_DATA[2]      ),
227      .hps_hps_io_sdio_inst_D3      ( HPS_SD_DATA[3]      ),
228
229      .hps_hps_io_usb1_inst_D0      ( HPS_USB_DATA[0]     ),
230      .hps_hps_io_usb1_inst_D1      ( HPS_USB_DATA[1]     ),
231      .hps_hps_io_usb1_inst_D2      ( HPS_USB_DATA[2]     ),
232      .hps_hps_io_usb1_inst_D3      ( HPS_USB_DATA[3]     ),
233      .hps_hps_io_usb1_inst_D4      ( HPS_USB_DATA[4]     ),
234      .hps_hps_io_usb1_inst_D5      ( HPS_USB_DATA[5]     ),
235      .hps_hps_io_usb1_inst_D6      ( HPS_USB_DATA[6]     ),
236      .hps_hps_io_usb1_inst_D7      ( HPS_USB_DATA[7]     ),
237      .hps_hps_io_usb1_inst_CLK     ( HPS_USB_CLKOUT      ),
238      .hps_hps_io_usb1_inst_STP     ( HPS_USB_STP         ),
239      .hps_hps_io_usb1_inst_DIR     ( HPS_USB_DIR         ),
240      .hps_hps_io_usb1_inst_NXT     ( HPS_USB_NXT         ),
241
242      .hps_hps_io_spim1_inst_CLK    ( HPS_SPIM_CLK  ),
243      .hps_hps_io_spim1_inst_MOSI   ( HPS_SPIM_MOSI ),
244      .hps_hps_io_spim1_inst_MISO   ( HPS_SPIM_MISO ),
245      .hps_hps_io_spim1_inst_SS0    ( HPS_SPIM_SS   ),
246
247      .hps_hps_io_uart0_inst_RX     ( HPS_UART_RX     ),
248      .hps_hps_io_uart0_inst_TX     ( HPS_UART_TX     ),
249
250      .hps_hps_io_i2c0_inst_SDA     ( HPS_I2C1_SDAT     ),
251      .hps_hps_io_i2c0_inst_SCL     ( HPS_I2C1_SCLK     ),
252
253      .hps_hps_io_i2c1_inst_SDA     ( HPS_I2C2_SDAT     ),
254      .hps_hps_io_i2c1_inst_SCL     ( HPS_I2C2_SCLK     ),
255
256      .hps_hps_io_gpio_inst_GPIO09  ( HPS_CONV_USB_N ),
257      .hps_hps_io_gpio_inst_GPIO35  ( HPS_ENET_INT_N ),
258      .hps_hps_io_gpio_inst_GPIO40  ( HPS_LTC_GPIO ),
259
260      .hps_hps_io_gpio_inst_GPIO48  ( HPS_I2C_CONTROL ),
261      .hps_hps_io_gpio_inst_GPIO53  ( HPS_LED ),
262      .hps_hps_io_gpio_inst_GPIO54  ( HPS_KEY ),
263      .hps_hps_io_gpio_inst_GPIO61  ( HPS_GSENSOR_INT ),
264
265     //Name conduits in platform designer: camera_a, camera_b, status
266      .camera_a_data (GPIO_0[7:0]),
267      .camera_a_href (GPIO_0[8]),
268      .camera_a_vsync (GPIO_0[9]),
269      .camera_a_pclk (GPIO_0[10]),
270      .camera_a_xclk (GPIO_0[26]),
271
272      .camera_b_data (GPIO_1[7:0]),
273      .camera_b_href (GPIO_1[8]),
274      .camera_b_vsync (GPIO_1[9]),
275      .camera_b_pclk (GPIO_1[10]),
276      .camera_b_xclk (GPIO_1[26]),
277
278      .status_leds (LEDR)
279
280  );
281
282   // The following quiet the "no driver" warnings for output
283   // pins and should be removed if you use any of these peripherals
284
285   assign ADC_CS_N = SW[1] ? SW[0] : 1'bZ;
```

```
286     assign ADC_DIN = SW[0];
287     assign ADC_SCLK = SW[0];
288
289     assign AUD_ADCLRCK = SW[1] ? SW[0] : 1'bZ;
290     assign AUD_BCLK = SW[1] ? SW[0] : 1'bZ;
291     assign AUD_DACDAT = SW[0];
292     assign AUD_DACLRCK = SW[1] ? SW[0] : 1'bZ;
293     assign AUD_XCK = SW[0];
294
295     assign DRAM_ADDR = { 13{ SW[0] } };
296     assign DRAM_BA = { 2{ SW[0] } };
297     assign DRAM_DQ = SW[1] ? { 16{ SW[0] } } : { 16{ 1'bZ } };
298     assign {DRAM_CAS_N, DRAM_CKE, DRAM_CLK, DRAM_CS_N,
299             DRAM_LDQM, DRAM_RAS_N, DRAM_UDQM, DRAM_WE_N} = { 8{SW[0]} };
300
301     assign FAN_CTRL = SW[0];
302
303     assign FPGA_I2C_SCLK = SW[0];
304     assign FPGA_I2C_SDAT = SW[1] ? SW[0] : 1'bZ;
305
306     assign GPIO_0[21:0] = {22{1'bZ}};
307     assign GPIO_1[21:0] = {22{1'bZ}};
308
309     assign HEX0 = { 7{ SW[1] } };
310     assign HEX1 = { 7{ SW[2] } };
311     assign HEX2 = { 7{ SW[3] } };
312     assign HEX3 = { 7{ SW[4] } };
313     assign HEX4 = { 7{ SW[5] } };
314     assign HEX5 = { 7{ SW[6] } };
315
316     assign IRDA_TXD = SW[0];
317
318     assign PS2_CLK = SW[1] ? SW[0] : 1'bZ;
319     assign PS2_CLK2 = SW[1] ? SW[0] : 1'bZ;
320     assign PS2_DAT = SW[1] ? SW[0] : 1'bZ;
321     assign PS2_DAT2 = SW[1] ? SW[0] : 1'bZ;
322
323     assign TD_RESET_N = SW[0];
324
325
326 endmodule
```