CSEE 4840 Design Document

Real-Time CT Image Processing

Kristine Vergara (kev2128), Saha Dev Shanmugam (ss7654), Nitali Arora (na3202)

Contents

1. Introduction

Medical imaging systems such as CT scanners, X-Ray machines, and MRI systems generate images that are commonly stored in DICOM (Digital Imaging and Communications in Medicine) format. DICOM is a standardized protocol used for managing, storing, transmitting, and displaying medical imaging data.

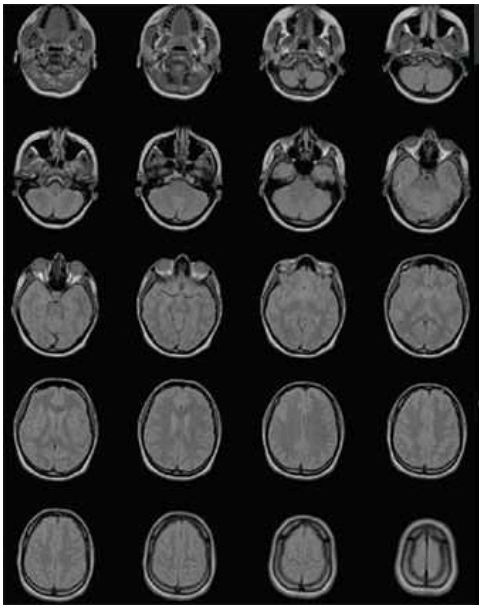


Fig.1: Brain MR images stored in DICOM format

Radiologists often need to apply image processing techniques such as edge detection, contrast enhancement, and sharpening to better visualize anatomical structures and pathological features. With the rise of smaller and more portable imaging devices, the demand for fast image processing systems capable of providing immediate visual feedback has grown significantly. Convolution-based edge detection requires a large number of arithmetic operations to be performed for every pixel in the image. Although these operations can be implemented on a CPU, they are inherently parallel and spatially structured, making them well-suited for FPGA implementation. The objective of this project is to develop an FPGA-accelerated image processing pipeline that is capable of applying multiple filters to CT images and displaying the processed result in real time on a VGA monitor.

2. System Block Diagram

2.1 Software Pipeline

The final end-to-end data path begins with a DICOM file, which is parsed by the HPS (ARM CPU). The ARM processor extracts the 512×512 pixel array stored as 16-bit signed Hounsfield Unit (HU) values, compresses them to 12-bit by masking (value & 0xFFF), and normalizes the data to 8 bit range. Finally, due to memory constraints, the image is downsampled from 512×512 to 128×128 while being written to the Avalon bus, through the bridge, to the M10K blocks based memory storage within each of the image processor algorithms.

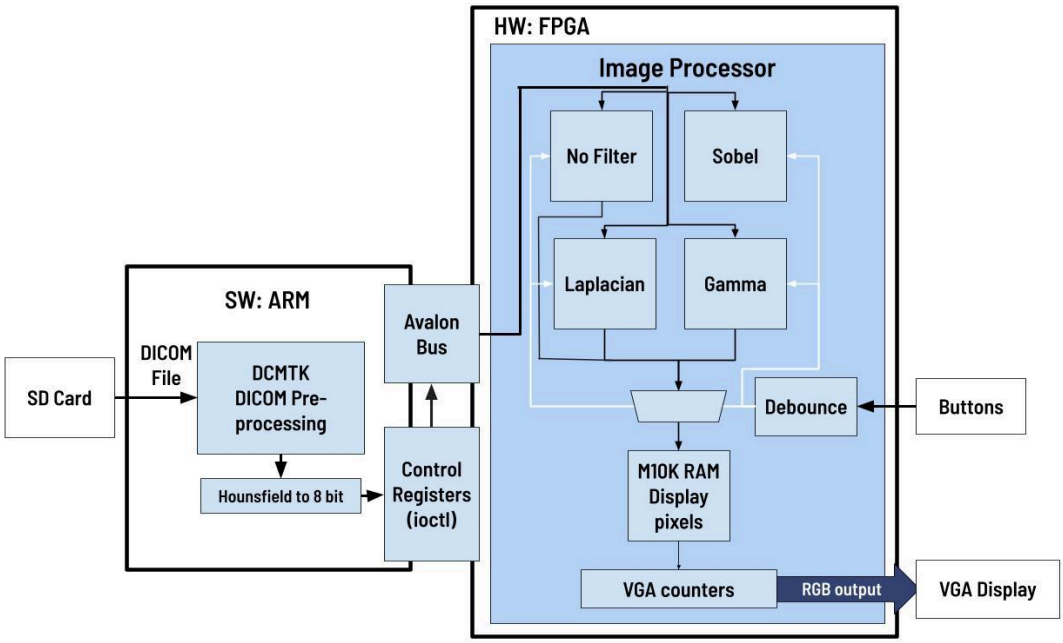


Fig.2: Block Diagram

2.2 FPGA Processing Pipeline

A raster scanner reads the image data sequentially loads four rows at a time into the M10K-based rolling line buffers.
These four streaming frame pipelines are instantiated inside hw/image_processor.sv, which multiplexes their outputs into the shared VGA display RAM. The user selects the active algorithm via the board's keys:

- KEY[0]: Passthrough (no filtering) — passthrough_frame_pipeline
- KEY[1]: Sobel edge detection + threshold — sobel_frame_pipeline
- KEY[2]: Laplacian sharpening — laplacian_frame_pipeline
- KEY[3]: Gamma (power-law) correction via LUT — gamma_frame_pipeline

Pixel data from the M10K line buffers is passed into sliding-window registers (window3x3_regs) that assemble the 3×3 neighborhood required for convolution-based algorithms. Each pipeline streams its output into the shared VGA display RAM (8-bit pixels) for real-time rendering.

2.3 VGA Output Interface

- Timing: 640×480 @ 50 Hz
- The processed 128×128 edge map is placed in the top-left of the VGA frame.
- Blanking is applied to all pixels outside the valid 128×128 region.

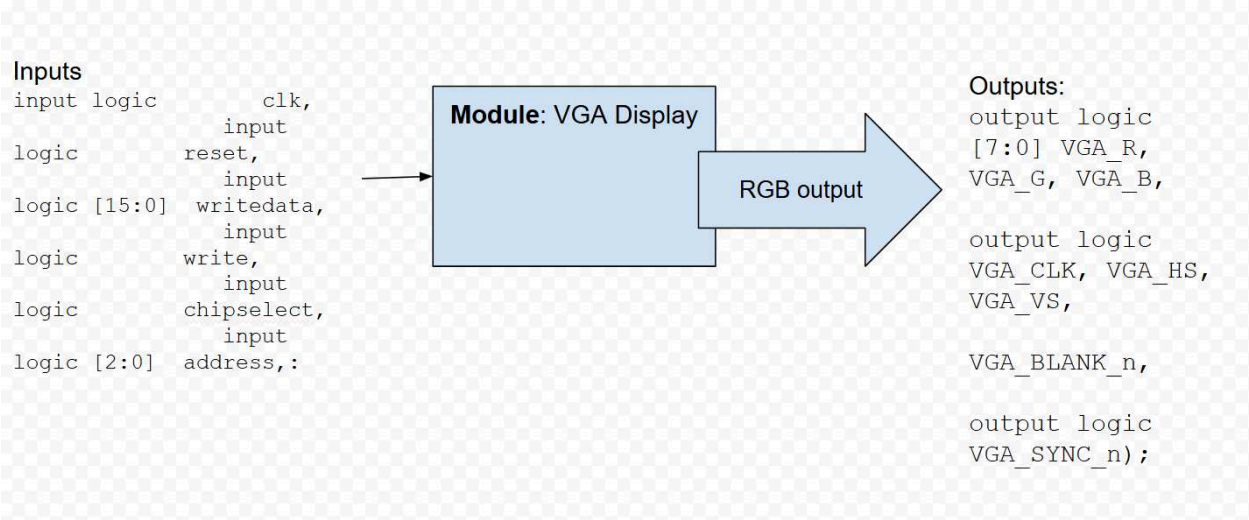


Fig. 3: VGA Timing and Mapping Diagram

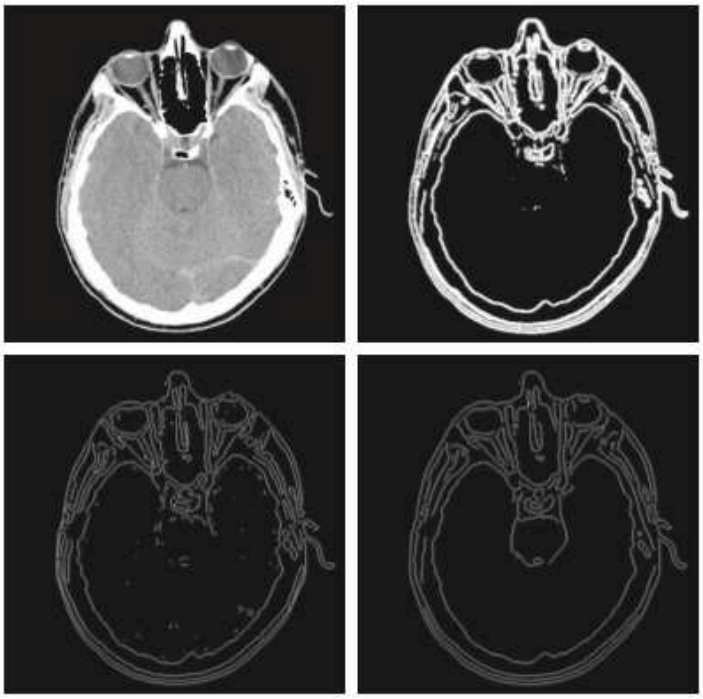


Fig.4: Reference image of brain MRI edge detection

Figure 4. shows an example of the expected output from applying the edge detection algorithm to a brain MRI scan. The skull, gray matter, white matter, and ventricles are highlighted as bright areas against a dark background. The edges are produced by computing the spatial gradient of the image intensity, as implemented in the Sobel filter. The strong edges along the outer boundaries of the brain and internal tissue interfaces (as seen in the lower right) demonstrate how gradient-based filters emphasize regions of rapid intensity change while suppressing uniform regions.

3. Hardware-Software Interface

3.1 Memory Layout (Image Storage within Each Filter)

The compressed image data occupies this region as follows:

- Pixel format: 8-bit per pixel
- Total bits: 128 × 128 × 8 = 131,072 bits
- Total words: 128 × 128 = 16,384 words
- Total bytes: 1 128 × 128 = 16,384 bytes (16 KB)
- Image region: 0x0000 → 0x3FFF

3.2 Register Map (Avalon Slave via Lightweight Bridge)

The FPGA exposes the starting address in memory to write the original image to.

Offset	Name	Description
0x0000	pixel[0]	Top left pixel
0x0001	pixel[1]	Second pixel in row 0 of image array
…	…	…
0x3FFF	pixel[16383]	Bottom right pixel

Fig. 5: HPS→FPGA Register Map Diagram

4. Algorithms

We selected three algorithms that serve distinct purposes in processing medical images: Sobel edge detection, Laplacian sharpening, and Power Law (gamma) transformation. All three operate on the same streaming pipeline architecture, consuming pixels from the M10K line buffers and writing 8-bit results into the shared VGA display RAM.

4.1 Sobel Filter

Fig. 6: Sobel Filter Kernels

The Sobel filter detects edges by computing the discrete spatial gradient of image intensity. For each pixel, the 3×3 neighborhood is read from the window3x3_regs module (fed by line4_m10k_buffer) and convolved with two orthogonal kernels:

- Gx measures the horizontal gradient (strength of vertical edges):
- Gx = -w00 + w02 - 2·w10 + 2·w12 - w20 + w22
- Gy measures the vertical gradient (strength of horizontal edges):
- Gy = -w00 - 2·w01 - w02 + w20 + 2·w21 + w22

These are the exact expressions implemented in `sobel3x3_threshold.sv`. The gradient magnitude is then approximated as:

$$G = |Gx| + |Gy|$$

This L1-norm approximation avoids the cost of squaring and square-root operations in hardware while remaining a good edge-strength estimate. Using absolute values ensures that transitions from bright to dark and dark to bright are treated equally.

After computing the magnitude, a fixed binary threshold is applied. In the current FPGA build, this threshold is set to 100 (an 8-bit parameter `SOBEL_THRESHOLD = 100` in `hw/image_processor.sv`):

```
output_pixel = (G > 100) ? 8'hFF : 8'h00
```

The pipeline output is aligned to in_valid, with out_valid delayed one cycle to match the convolution latency. The resulting binary edge map is written to the shared VGA display RAM in raster order.

Bit-width analysis: After weighted convolution, Gx and Gy each span the range [-1020, +1020], requiring 16-bit signed representation. The combined magnitude reaches up to 2040. The threshold step compresses this back to an 8-bit binary output before the stream is forwarded to the VGA controller.

Pseudocode:

```
None
const Kx = [[-1,  0, +1],
            [-2,  0, +2],
            [-1,  0, +1]]   // horizontal gradient

const Ky = [[-1, -2, -1],
            [ 0,  0,  0],
            [+1, +2, +1]]   // vertical gradient

Function Sobel(image):
    for y = 1 to H-2:
        for x = 1 to W-2:
            window = get_3x3_window(image, x, y)   // from line buffers /
window3x3_regs

            Gx = 0
```

```
            for i = 0 to 2:
                for j = 0 to 2:
                    Gx += window[i][j] * Kx[i][j]

            Gy = 0
            for i = 0 to 2:
                for j = 0 to 2:
                    Gy += window[i][j] * Ky[i][j]

            G = abs(Gx) + abs(Gy)       // L1 magnitude approximation

            if G > 100:                  // SOBEL_THRESHOLD = 100 (fixed
parameter)
                output[y][x] = 255
            else:
                output[y][x] = 0
    return output
```

4.2 Power Law (Gamma)

The power-law (gamma) transformation brightens dark regions of an image by applying a non-linear remapping to pixel intensities. Because this is a point operation, each output pixel depends only on the value of its corresponding input pixel, and it is well-suited to a streaming hardware architecture.

The transformation maps an 8-bit input pixel $x \in [0, 255]$ to an 8-bit output y according to:

$$y = 255 \cdot (x / 255)^{\gamma}$$

The current implementation uses a fixed $\gamma = 0.5$ (square root), which expands lower intensity values (shadows) while compressing higher intensities (highlights), effectively brightening the image.

RTL Implementation

The design is partitioned into two components implemented in `hw/buttons/gamma_lut.sv` and `hw/buttons/gamma_frame_pipeline.sv`.

Precomputed 256-entry LUT (**gamma_lut.sv**): To avoid computing exponents and divisions at runtime, the transformation is precomputed into a 256-entry lookup table. The LUT is synthesized as dedicated M10K RAM blocks (using the `(* ramstyle = "M10K" *)` attribute), with each of the 256 entries storing an 8-bit result. The input pixel intensity serves directly as the memory address, producing a single-cycle lookup:

$$y = lut[idx]$$

Streaming pipeline (`gamma_frame_pipeline.sv`): Incoming grayscale data is buffered in an internal `frame_m10k_store`. Processing begins only after the host asserts `host_frame_done_pulse`, indicating a complete frame is in memory. The frame_read_scheduler then traverses the framebuffer in raster order, generating `sched_pix_valid` and `sched_pix_data`. Each pixel is fed into the LUT, and the result is registered as `out_pixel`. Output addresses are computed as:

$$stream\_lin = sched\_row \times IMG\_WIDTH + sched\_col$$

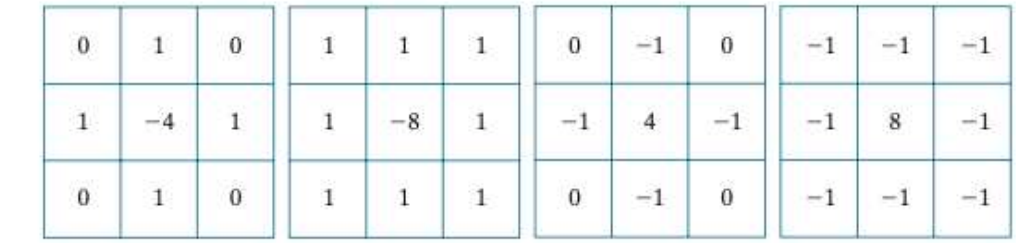
This ensures the LUT output overwrites the display buffer in raster order. Because each pixel requires exactly one clock cycle for the LUT lookup, the pipeline achieves deterministic single-cycle latency per pixel with no stalls, and uses M10K blocks for the LUT to preserve logic elements for other tasks.

Pseudocode:

```
None
Function BuildGammaLUT(gamma, c = 1):
    for i = 0 to 255:
        LUT[i] = clamp( c * (i / 255)^gamma * 255, 0, 255 )
    return LUT

Function GammaCorrection(image, LUT):
    for y = 0 to H-1:
        for x = 0 to W-1:
            pixel_out = LUT[image[y][x]]    // single-cycle M10K lookup
            output[y][x] = pixel_out
    return output
```

4.3 Laplacian Filter



0	1	0	1	1	1	0	−1	0	−1	−1	−1
1	−4	1	1	−8	1	−1	4	−1	−1	8	−1
0	1	0	1	1	1	0	−1	0	−1	−1	−1

a b c d

FIGURE 3.45 (a) Laplacian kernel used to implement Eq. (3-53). (b) Kernel used to implement an extension of this equation that includes the diagonal terms. (c) and (d) Two other Laplacian kernels.

Fig. 7: Laplacian Filter Kernel

The Laplacian filter sharpens the image by computing the second spatial derivative and combining it with the original pixel values. This pipeline is implemented in `hw/buttons/laplacian_frame_pipeline.sv` and `hw/buttons/laplacian3x3_sharpen.sv`, and selected via KEY[2].

The discrete Laplacian is computed from the 3×3 neighborhood using the standard 4-connected kernel:

$$L(x,y) = \nabla^2 I(x,y)$$

The sharpened output is then obtained by subtracting the Laplacian from the original pixel:

$$I\_sharp = I\_original - L(x,y)$$

The result is saturated to the 8-bit range [0, 255]: values below 0 are clipped to black, values above 255 are clipped to white, and mid-range values pass through unchanged. This maps the Laplacian output range of [-1020, +1020] to a valid 8-bit display value.

In hardware, both the original pixel (8-bit) and the Laplacian result (16-bit signed) are held in registers during the sharpening step:

```
None
reg [7:0]        original_pixel;
reg signed [15:0] laplacian_value;
reg signed [15:0] sharp_value;

sharp_value = original_pixel - laplacian_value;
```

```
if      (sharp_value < 0)   output_pixel = 0;
else if (sharp_value > 255) output_pixel = 255;
else                        output_pixel = sharp_value;
```

Bit-width analysis: The Laplacian output spans [-1020, +1020], requiring 16-bit signed representation. After the sharpening, subtraction, and saturation clipping, the final output is 8-bit, matching the VGA display format.

Pseudocode:

```
None
const K4 = [[ 0, +1,  0],
            [+1, -4, +1],
            [ 0, +1,  0]]

const K8 = [[+1, +1, +1],
            [+1, -8, +1],
            [+1, +1, +1]]

Function Laplacian(image, kernel):
    for y = 1 to H-2:
        for x = 1 to W-2:
            window = get_3x3_window(image, x, y)   // from M10K line
buffers + shift registers

            L = 0
            for i = 0 to 2:
                for j = 0 to 2:
                    L += window[i][j] * kernel[i][j]

            sharp = image[y][x] - L

            if      sharp < 0:   output[y][x] = 0
            else if sharp > 255: output[y][x] = 255
            else:                output[y][x] = sharp
    return output
```

5. Resource Budgets (Memory Constraints)

5.1 ARM DICOM Preprocessing

DICOM data is stored as Hounsfield Units (HU), ranging from -1024 to 3071 HU, giving 4096 possible values. Since $2^{12} = 4096$, pixel data can be represented in 12 bits. The DICOM image is 512×512 but is downsampled to 128×128 on the HPS using nearest neighbor interpolation before being sent to the FPGA. The resulting transfer size is:

```
None
128 × 128 × 8 bits/pixel ÷ 8 bits/byte = 16,384 bytes = 16 KB
```

The cost of downsampling is loss of spatial resolution; however since the VGA display window is also 128×128 this does not affect displayed output quality.

5.3 M10K Frame Stores

Once the ARM processor completes preprocessing, pixels are transferred one byte at a time over the Avalon bus via ioctl calls from the Linux device driver. All four pipeline frame stores receive the same broadcast write simultaneously, so each stores an identical copy of the full 128×128 frame:

$$128 \times 128 \times 8 \text{ bits} = 131{,}072 \text{ bits} = 16{,}384 \text{ bytes} = 16 \text{ KB per frame store}$$

$$16 \text{ KB} \times 4 \text{ pipelines} = 64 \text{ KB total frame store usage}$$

Each frame store requires:

$$16{,}384 \text{ words} \div 1{,}024 \text{ words per M10K} = 16 \text{ M10K blocks per frame store}$$

$$16 \times 4 = 64 \text{ M10K blocks total for frame stores}$$

M10K blocks also implement the rolling line buffers that support 3×3 convolution operations in the Sobel and Laplacian pipelines. The line buffer holds 4 rows of pixel data at any time:

$$4 \text{ rows} \times 128 \text{ pixels} \times 8 \text{ bits} = 4{,}096 \text{ bits} = 512 \text{ bytes per pipeline}$$

Each pipeline that uses convolution (Sobel and Laplacian) has its own line buffer instance.

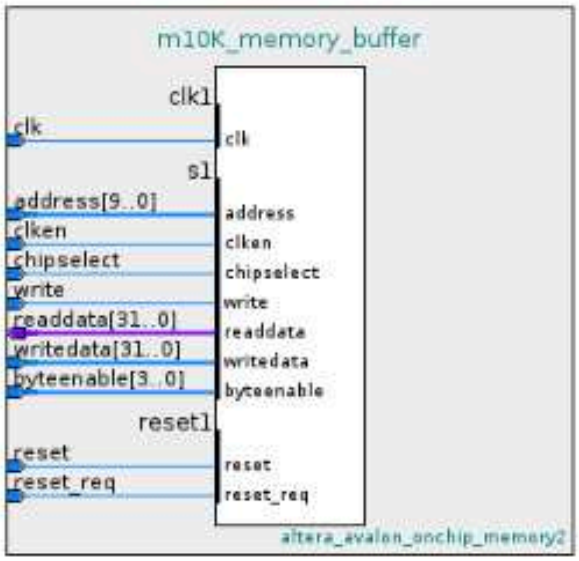


Fig.8: M10k Buffer Inputs (Blue lines) and Outputs (Purple Line)

5.4 Display Buffer

Processed output pixels are written into a dedicated true dual-port display buffer (`u_disp`). Port A receives pipeline output writes and CPU readback. Port B is read continuously by the VGA controller:

$$128 \times 128 \times 8 \text{ bits} = 131{,}072 \text{ bits} = 16{,}384 \text{ bytes} = 16 \text{ KB per frame store}$$

$$16{,}384 \text{ words} \div 1{,}024 \text{ words per M10K} = 16 \text{ M10K blocks per frame store}$$

5.4.1 Total M10K Usage

4 × frame stores	*= 64 M10K blocks*
1 × display buffer	*= 16 M10K blocks*
4 × line buffers	*= ~2 M10K block each (512 bytes each)*
1 × gamma LUT	*= 1 M10K block (256 bytes)*
Total	*≈ 84 M10K blocks*

5.4.2 Neighborhood Pixel Buffer Register Map

Sobel and Laplacian filtering require access to a 3×3 neighborhood window for each output pixel. This window is dynamically constructed in hardware from the M10K line buffers and shift registers, holding 9 pixels at 8-bit precision:

$$3 \times 3 \times 8 \text{ bits} = 72 \text{ bits}$$

Three shift registers hold the pixel rows being processed at any time, forming the sliding 3×3 window supplied to the algorithm modules.

5.4.3 Header File Interface

```C/C++
#define IMG_WIDTH                128

#define IMG_HEIGHT               128

#define PIXEL_DEPTH              8

#define IMG_SIZE                 (IMG_WIDTH * IMG_HEIGHT)

#define FRAME_BUFFER_SIZE_BYTES (IMG_SIZE * (PIXEL_DEPTH / 8))  //
16,384 bytes
```

`IMG_SIZE` is the total pixel count. `FRAME_BUFFER_SIZE_BYTES` is the total memory footprint of one frame in bytes. All image data lives in M10K blocks within the FPGA fabric.

5.5 Algorithms (Memory)

5.5.1 Sobel Filter

- Input: 128×128 8-bit normalized image from M10K frame store (16 KB)
- Line buffer: 4 rows × 128 pixels × 8 bits = 512 bytes (M10K)
- Gradient range: Gx, Gy ∈ [-1020, +1020] → 16-bit signed representation required
- Magnitude range: up to 2040 → compared against fixed threshold of 100
- Output: 8-bit binary image (0 or 255) written to display buffer in raster order

The sum of absolute Sobel kernel weights is 8 per gradient direction. Since only half the weights contribute positively or negatively, the maximum per-gradient output is:

```None
Max output = 255 × 4 = 1020
```

After thresholding, the 16-bit intermediate values are compressed back to 8-bit binary output before display.

5.5.2 Power-Law (Gamma) Transformation

- Input: 128×128 8-bit normalized image from M10K frame store (16 KB)
- LUT storage: 256 entries × 8 bits = 256 bytes (M10K ROM)
- Latency: 1 clock cycle per pixel (deterministic)
- Output: 8-bit remapped pixel written to display buffer in raster order

The LUT precomputes y = 255 · (x/255)^0.5 for all 256 input values, with results rounded and clamped to [0, 255]. This eliminates all runtime floating-point computation and ensures maximum throughput. Unlike Sobel and Laplacian, gamma requires no line buffer since it operates on one pixel at a time with no neighborhood context.

5.5.3 Laplacian Filter

- Input: 128×128 8-bit normalized image from M10K frame store (16 KB)
- Line buffer: 4 rows × 128 pixels × 8 bits = 512 bytes (M10K)
- Laplacian range: L(x,y) ∈ [-1020, +1020] → 16-bit signed representation required
- Sharpening: I_sharp = I_original - L(x,y), saturated to [0, 255]
- Output: 8-bit saturated pixel written to display buffer in raster order

Unlike Sobel, the Laplacian result is not used directly as the output. It is combined with the original 8-bit pixel value in a signed 16-bit subtraction before saturation clipping maps the result back to the 8-bit display range.

6. Team Contributions

- System Prototyping and FPGA Implementation: Saha led the development of the initial system prototype and was primarily responsible for the successful implementation of the core algorithm on the FPGA hardware.
- Software Infrastructure and Environment: Kristine spearheaded the foundational code architecture and managed the comprehensive software environment setup, ensuring a stable platform for integration.
- Theoretical Framework and Data Handling: Nitali established the theoretical underpinnings of the project and managed the critical data pipeline, specifically the process of formatting and transferring DICOM image data onto the SD card.

The Design Documentation and final presentations were treated as shared milestones, with components partitioned equally to ensure a cohesive representation of the team's work. Crucial hardware milestones were tackled sequentially due to the limitation of only one person being able to work on implementing hardware on the FPGA. This included the successful debugging of the M10K memory buffers and the subsequent verification of the VGA display interface for standalone image rendering.

7. Lessons You Learned

SD Card Initialization and Mounting

To process DICOM image data on the ARM processor, files must be accessible via the FPGA board's external SD card. Since the board operates in a Linux-based environment, a specific mounting procedure is required before data can be transferred.

After inserting the SD card, the device is identified with lsblk, typically appearing as /dev/mmcblk0, with the primary data partition at/dev/mmcblk0p1. A dedicated mount point is then created and the partition linked to it:

```
None
mkdir /mnt/sdcard
mount /dev/mmcblk0p1 /mnt/sdcard
```

Data Transfer Methods:

Two distinct hardware interfaces were used to move DICOM datasets from the host desktop to the FPGA's file system.

Method A: USB Mass Storage

Files were copied from the desktop to a USB thumb drive, which was then inserted into the FPGA board and mounted (typically at /dev/sda1). The cp command moved files from the USB mount point to the SD card mount point.

Method B: Ethernet (Network Transfer)

The FPGA and desktop were connected via Ethernet and configured on the same subnet. Files were transferred directly to the FPGA using SCP or FTP, then moved to the SD card:

```
None

cp /home/root/images.dcm /mnt/sdcard/
```

Action	Command
List block devices	`lsblk`
Create directory	`mkdir /mnt/sdcard`

Mount partition	`mount /dev/mmcblk0p1 /mnt/sdcard`
Copy data	`cp [source] /mnt/sdcard/`
Verify transfer	`ls -l /mnt/sdcard/`

Using Platform Designer

Going through and adding our own connections via platform designer with ports we had designed with our own sizing allowed us to visualize how the Avalon Bus works to connect components. Connecting our image_processor component required exposing the correct Avalon MM slave port signals (chipselect, write, address, writedata, readdata) and specifying the address span to match our register map size of 16,384 bytes. Seeing the physical address assigned by Platform Designer directly correspond to the SDRAM_BASE_OFFSET in our header file made the connection between software and hardware concrete — a write() call in C at a given offset maps directly to the address signal arriving at our RTL.

Designing Hardware for Custom Purposes

During our design review process, the hardware design for convolutions was explained, and that method of problem solving bounded by memory and time limitations was very insightful. Choices in algorithm implementation came with additional questions, and now the implementation of convolutions is clear. A key insight was understanding why a line buffer architecture is necessary — unlike software where random access to any pixel is free, hardware pipelines must process pixels in strict raster order, meaning the 3×3 neighborhood window cannot be constructed by simply indexing an array. Instead, M10K blocks buffer entire rows so that three vertically adjacent pixels are simultaneously available as the window slides across each column

Hardware and Software Interfaces - Drivers

Although our HW SW interface implementation is very similar to Lab 3, it still provided insight as to how memory management from high levels of computing matches with lower levels. The importance of drivers was very helpful in ensuring property memory management, and during our design and set up of the drivers, we made mistakes which helped solidify our understanding on the basics of device driver design.

Value of Buffers

One challenge we faced was completing a pipeline project while each phase had a different input and output speed. The value of buffers was key. We had two registers involved following reading from M10K buffers due to their read delay, which then solved our slanted image display problem. Without the framebuffer at the end, the timing was completely off on our VGA display and nothing would’ve showed.

Instead, the pipeline never stalls because the scheduler, line buffer, window registers, and convolution compute stages are all operating concurrently on different pixels at different stages — by the time the convolution module is processing pixel N, the scheduler has already read ahead to fill the window for pixel N+1. Understanding the latency introduced at each stage — the memory read delay, the line buffer fill time, and the window accumulation — and how coordinate tracking must be carefully pipelined alongside pixel data to avoid spatial artifacts, gave us a much deeper appreciation for the constraints that separate hardware algorithm design from its software equivalent.

Document all your attempts

It becomes really easy for your teammates to help if they can understand where you went wrong. Since we worked in a sequential method, with a teammate working on this during the day, and another teammate continuing on this in the afternoon and evening, our time would have been more effective with improved communication.

8. Advice for Future Projects

1. Start early! We ran into issues because we did not understand how tools and software and hardware worked. Although designing to begin with is nice, to design the entire project requires some understanding of how to implement some of the basics. Making the decision of what memory type to use becomes easier once we ourselves implement in practice. Although it was discussed in class, having hands on learning always improves our understanding. Take the time to do mini lessons weekly on what's been discussed, so when planning a project that will put these together, you understand how they will piece together and can design accordingly.
2. This goes into the second topic, which is to use what you learn in class. Unfortunately with the power of the internet and AI, it's very tempting to prompt online information to improve your design, but more often than not these methods fail. This project is not meant to be extravagant. It is meant to give us hands-on beginner experience, and extending what you see in class is perfect. No need to go big with the project, but rather take the time to struggle with the class material. We used M10K blocks, but we didn't really connect it to what we had learned in class, until Professor Edwards pointed out the slides in which he had mentioned our exact issue. If we had referenced the class materials on this issue, we would have avoided 4 days worth of time.
3. Understand memory at its core. A big question that was brought up is how much resource utilization will this project cost, but that question extends far beyond the number of bits of data we were using, but also the capacity of the technology. Understand how to use M10K blocks, and what their sizing signifies. Why use certain sizes and what advantages do these choices provide? Understanding that *before* designing your project is key for maximum efficiency.
4. We made the mistake of trying to implement the entire project without step by step testing. When first creating a project, it's important to go slow, despite a sense of urgency. For this reason, do it the most obvious and simplest way first. Don't worry about memory or time optimization, even if it means the design is now slower or less impressive. Smaller mistakes are easier to fix.

5. Practice using the tools. Before this project, Quartus and Platform Designer were very intimidating. We had seen how it could get complicated really quickly in class, but after some time playing around, it became easier to understand.
6. Utilize testbenches for better separation of work. Testbenches ensure proper design of interfaces and allow for a divide and conquer approach to implementing the FPGA portion of the project.

9. Listing of Files

/hw
debounce_keys.sv

None

```
// Debounce active-low FPGA keys (e.g. KEY[3:0]); outputs active-high when pressed
& stable.
module debounce_keys #(
    parameter int           NUM_KEYS        = 4,
    parameter int unsigned DEBOUNCE_CYCLES = 20'd1_000_000  // ~20 ms @ 50 MHz
) (
    input  logic                    clk,
    input  logic                    reset,
    input  logic [NUM_KEYS-1:0]     KEY_n,
    output logic [NUM_KEYS-1:0]     key_pressed
);

  logic [NUM_KEYS-1:0] sync1;
  logic [NUM_KEYS-1:0] sync2;

  always_ff @(posedge clk or posedge reset) begin
    if (reset)
      sync1 <= {NUM_KEYS{1'b1}};
    else
      sync1 <= KEY_n;
  end

  always_ff @(posedge clk or posedge reset) begin
    if (reset)
      sync2 <= {NUM_KEYS{1'b1}};
    else
      sync2 <= sync1;
  end

  genvar k;
  generate
    for (k = 0; k < NUM_KEYS; k++) begin : g_debounce
```

```
      logic [19:0] cnt;
      logic raw_pressed;
      assign raw_pressed = ~sync2[k];

      always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
          cnt              <= '0;
          key_pressed[k]   <= 1'b0;
        end else begin
          if (raw_pressed == key_pressed[k]) begin
            cnt <= '0;
          end else begin
            if (cnt == DEBOUNCE_CYCLES[19:0]) begin
              key_pressed[k] <= raw_pressed;
              cnt            <= '0;
            end else begin
              cnt <= cnt + 1'b1;
            end
          end
        end
      end
    end
  endgenerate

endmodule
```

frame_m10k_store.sv

```
None
module frame_m10k_store #(
  parameter int IMG_WIDTH  = img_pkg::IMG_WIDTH,
  parameter int IMG_HEIGHT = img_pkg::IMG_HEIGHT,
  parameter int PIXEL_W    = img_pkg::PIXEL_W
) (
  input  logic                                  clk,
  input  logic                                  rst_n,
  input  logic                                  wr_en,
  input  logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] wr_addr,
  input  logic [PIXEL_W-1:0]                    wr_data,
  input  logic                                  rd_en,
  input  logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] rd_addr,
  output logic [PIXEL_W-1:0]                    rd_data
```

```
);
  localparam int FRAME_PIXELS = IMG_WIDTH * IMG_HEIGHT;
  localparam int ADDR_W       = $clog2(FRAME_PIXELS);

  (* ramstyle = "M10K" *) logic [PIXEL_W-1:0] frame_mem [0:FRAME_PIXELS-1];

  // Registered read matching rd_addr asserted this cycle (M10K 1-cycle read).
  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n)
      rd_data <= '0;
    else begin
      if (wr_en)
        frame_mem[wr_addr] <= wr_data;
      if (rd_en)
        rd_data <= frame_mem[rd_addr];
    end
  end
endmodule
```

frame_read_scheduler.sv

```
None
module frame_read_scheduler #(
  parameter int IMG_WIDTH  = img_pkg::IMG_WIDTH,
  parameter int IMG_HEIGHT = img_pkg::IMG_HEIGHT,
  parameter int PIXEL_W    = img_pkg::PIXEL_W
) (
  input  logic clk,
  input  logic rst_n,
  input  logic start,
  output logic running,
  output logic done,
  output logic rd_en,
  output logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] rd_addr,
  input  logic [PIXEL_W-1:0] rd_data,
  output logic pix_valid,
  output logic [PIXEL_W-1:0] pix_data,
  output logic [$clog2(IMG_WIDTH)-1:0] pix_col,
  output logic [$clog2(IMG_HEIGHT)-1:0] pix_row
);
  localparam int FRAME_PIXELS = IMG_WIDTH * IMG_HEIGHT;
  localparam int ADDR_W       = $clog2(FRAME_PIXELS);
```

```
logic [ADDR_W-1:0] addr_ctr;
logic              rd_en_q;
logic [$clog2(IMG_WIDTH)-1:0]  col_q;
logic [$clog2(IMG_HEIGHT)-1:0] row_q;
logic [$clog2(IMG_WIDTH)-1:0]  pix_col_r;
logic [$clog2(IMG_HEIGHT)-1:0] pix_row_r;

always_ff @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    running   <= 1'b0;
    done      <= 1'b0;
    rd_en     <= 1'b0;
    rd_en_q   <= 1'b0;
    rd_addr   <= '0;
    addr_ctr  <= '0;
    pix_valid <= 1'b0;
    pix_data  <= '0;
    pix_col   <= '0;
    pix_row   <= '0;
    pix_col_r <= '0;
    pix_row_r <= '0;
    col_q     <= '0;
    row_q     <= '0;
  end else begin
    done      <= 1'b0;
    pix_valid <= rd_en_q;
    pix_data  <= rd_data;
    pix_col   <= pix_col_r;
    pix_row   <= pix_row_r;
    pix_col_r <= col_q;
    pix_row_r <= row_q;
    rd_en_q   <= rd_en;

    if (start && !running) begin
      running  <= 1'b1;
      rd_en    <= 1'b1;
      rd_addr  <= '0;
      // Issue addr 0 on this cycle. First running cycle must issue addr 1
      // (avoid duplicate read of 0), and {row_q,col_q} must track addr_ctr.
      addr_ctr <= ADDR_W'(1);
      col_q    <= ($clog2(IMG_WIDTH))'(1);
      row_q    <= '0;
    end else if (running) begin
      rd_addr <= addr_ctr;
```

```
        if (addr_ctr == FRAME_PIXELS-1) begin
          running <= 1'b0;
          rd_en   <= 1'b0;
          done    <= 1'b1;
        end else begin
          addr_ctr <= addr_ctr + 1'b1;
          rd_en    <= 1'b1;
          if (col_q == IMG_WIDTH-1) begin
            col_q <= '0;
            row_q <= row_q + 1'b1;
          end else begin
            col_q <= col_q + 1'b1;
          end
        end
      end else begin
        rd_en <= 1'b0;
      end
    end
  end
endmodule
```

gamma_frame_pipeline.sv

None

```
// Power-law gamma (LUT) streaming pipeline — Avalon-free; host writes frame_m10k
externally.
// Port list matches svtest sobel_frame_pipeline for drop-in with
gammatest/image_processor.sv.
// Uses unchanged svtest/frame_m10k_store + frame_read_scheduler (compiled from
../svtest).
module gamma_frame_pipeline #(
    parameter int IMG_WIDTH  = img_pkg::IMG_WIDTH,
    parameter int IMG_HEIGHT = img_pkg::IMG_HEIGHT,
    parameter int ADDR_W     = img_pkg::FRAME_ADDR_W,
    parameter int PIXEL_W    = img_pkg::PIXEL_W
) (
    input logic clk,
    input logic rst_n,

    input logic                                  frame_wr_en,
    input logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] frame_wr_addr,
    input logic [7:0]                             frame_wr_data,
```

```
    input logic host_frame_done_pulse,

    output logic                                    out_valid,
    output logic [7:0]                              out_pixel,
    output logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] out_ram_addr,
    output logic                                    sched_done
);

  logic h1, h2;
  logic process_start;

  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      h1 <= 1'b0;
      h2 <= 1'b0;
    end else begin
      h2 <= h1;
      h1 <= host_frame_done_pulse;
    end
  end

  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n)
      process_start <= 1'b0;
    else
      process_start <= h1 & ~h2;
  end

  logic rd_en;
  logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] rd_addr;
  logic [7:0] rd_data;

  logic sched_pix_valid;
  logic [7:0] sched_pix_data;
  logic [$clog2(IMG_WIDTH)-1:0]  sched_col;
  logic [$clog2(IMG_HEIGHT)-1:0] sched_row;

  logic sched_running_unused;
  logic [7:0] lut_y;

  wire [ADDR_W-1:0] stream_lin =
      ADDR_W'(sched_row) * ADDR_W'(IMG_WIDTH) + ADDR_W'(sched_col);

  gamma_lut u_gamma_lut (
```

```
    .idx(sched_pix_data),
    .y  (lut_y)
);

frame_m10k_store #(
    .IMG_WIDTH (IMG_WIDTH),
    .IMG_HEIGHT(IMG_HEIGHT),
    .PIXEL_W  (8)
) u_frame_m10k_store (
    .clk    (clk),
    .rst_n  (rst_n),
    .wr_en  (frame_wr_en),
    .wr_addr(frame_wr_addr),
    .wr_data(frame_wr_data),
    .rd_en  (rd_en),
    .rd_addr(rd_addr),
    .rd_data(rd_data)
);

frame_read_scheduler #(
    .IMG_WIDTH (IMG_WIDTH),
    .IMG_HEIGHT(IMG_HEIGHT),
    .PIXEL_W  (8)
) u_frame_read_scheduler (
    .clk      (clk),
    .rst_n    (rst_n),
    .start    (process_start),
    .running  (sched_running_unused),
    .done     (sched_done),
    .rd_en    (rd_en),
    .rd_addr  (rd_addr),
    .rd_data  (rd_data),
    .pix_valid(sched_pix_valid),
    .pix_data (sched_pix_data),
    .pix_col  (sched_col),
    .pix_row  (sched_row)
);

//------------------------------------------------------------------
// Register path: VGA/display writes gate on one valid per pixel streamed.
//------------------------------------------------------------------
always_ff @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    out_valid    <= 1'b0;
    out_pixel    <= '0;
```

```
      out_ram_addr <= '0;
    end else begin
      out_valid <= sched_pix_valid;
      if (sched_pix_valid) begin
        out_pixel    <= lut_y;
        out_ram_addr <= stream_lin;
      end
    end
  end

endmodule
```

gamma_lut.sv

```
None
// 256×8 power-law LUT (implements s = round(255 * (x/255)^γ), γ = 1/2 here).
// Per design doc: gamma via lookup stored in FPGA memory — infers ROM / M10K.
module gamma_lut (
    input logic [7:0] idx,
    output logic [7:0] y
);
  (* ramstyle = "M10K" *) logic [7:0] lut[256];

  initial begin
      lut[0] = 8'd0;
      lut[1] = 8'd16;
      lut[2] = 8'd23;
      lut[3] = 8'd28;
      lut[4] = 8'd32;
      lut[5] = 8'd36;
      lut[6] = 8'd39;
      lut[7] = 8'd42;
      lut[8] = 8'd45;
      lut[9] = 8'd48;
      lut[10] = 8'd50;
      lut[11] = 8'd53;
      lut[12] = 8'd55;
      lut[13] = 8'd58;
      lut[14] = 8'd60;
      lut[15] = 8'd62;
      lut[16] = 8'd64;
      lut[17] = 8'd66;
```

```
lut[18] = 8'd68;
lut[19] = 8'd70;
lut[20] = 8'd71;
lut[21] = 8'd73;
lut[22] = 8'd75;
lut[23] = 8'd77;
lut[24] = 8'd78;
lut[25] = 8'd80;
lut[26] = 8'd81;
lut[27] = 8'd83;
lut[28] = 8'd84;
lut[29] = 8'd86;
lut[30] = 8'd87;
lut[31] = 8'd89;
lut[32] = 8'd90;
lut[33] = 8'd92;
lut[34] = 8'd93;
lut[35] = 8'd94;
lut[36] = 8'd96;
lut[37] = 8'd97;
lut[38] = 8'd98;
lut[39] = 8'd100;
lut[40] = 8'd101;
lut[41] = 8'd102;
lut[42] = 8'd103;
lut[43] = 8'd105;
lut[44] = 8'd106;
lut[45] = 8'd107;
lut[46] = 8'd108;
lut[47] = 8'd109;
lut[48] = 8'd111;
lut[49] = 8'd112;
lut[50] = 8'd113;
lut[51] = 8'd114;
lut[52] = 8'd115;
lut[53] = 8'd116;
lut[54] = 8'd117;
lut[55] = 8'd118;
lut[56] = 8'd119;
lut[57] = 8'd121;
lut[58] = 8'd122;
lut[59] = 8'd123;
lut[60] = 8'd124;
lut[61] = 8'd125;
lut[62] = 8'd126;
```

```
lut[63] = 8'd127;
lut[64] = 8'd128;
lut[65] = 8'd129;
lut[66] = 8'd130;
lut[67] = 8'd131;
lut[68] = 8'd132;
lut[69] = 8'd133;
lut[70] = 8'd134;
lut[71] = 8'd135;
lut[72] = 8'd135;
lut[73] = 8'd136;
lut[74] = 8'd137;
lut[75] = 8'd138;
lut[76] = 8'd139;
lut[77] = 8'd140;
lut[78] = 8'd141;
lut[79] = 8'd142;
lut[80] = 8'd143;
lut[81] = 8'd144;
lut[82] = 8'd145;
lut[83] = 8'd145;
lut[84] = 8'd146;
lut[85] = 8'd147;
lut[86] = 8'd148;
lut[87] = 8'd149;
lut[88] = 8'd150;
lut[89] = 8'd151;
lut[90] = 8'd151;
lut[91] = 8'd152;
lut[92] = 8'd153;
lut[93] = 8'd154;
lut[94] = 8'd155;
lut[95] = 8'd156;
lut[96] = 8'd156;
lut[97] = 8'd157;
lut[98] = 8'd158;
lut[99] = 8'd159;
lut[100] = 8'd160;
lut[101] = 8'd160;
lut[102] = 8'd161;
lut[103] = 8'd162;
lut[104] = 8'd163;
lut[105] = 8'd164;
lut[106] = 8'd164;
lut[107] = 8'd165;
```

```
lut[108] = 8'd166;
lut[109] = 8'd167;
lut[110] = 8'd167;
lut[111] = 8'd168;
lut[112] = 8'd169;
lut[113] = 8'd170;
lut[114] = 8'd170;
lut[115] = 8'd171;
lut[116] = 8'd172;
lut[117] = 8'd173;
lut[118] = 8'd173;
lut[119] = 8'd174;
lut[120] = 8'd175;
lut[121] = 8'd176;
lut[122] = 8'd176;
lut[123] = 8'd177;
lut[124] = 8'd178;
lut[125] = 8'd179;
lut[126] = 8'd179;
lut[127] = 8'd180;
lut[128] = 8'd181;
lut[129] = 8'd181;
lut[130] = 8'd182;
lut[131] = 8'd183;
lut[132] = 8'd183;
lut[133] = 8'd184;
lut[134] = 8'd185;
lut[135] = 8'd186;
lut[136] = 8'd186;
lut[137] = 8'd187;
lut[138] = 8'd188;
lut[139] = 8'd188;
lut[140] = 8'd189;
lut[141] = 8'd190;
lut[142] = 8'd190;
lut[143] = 8'd191;
lut[144] = 8'd192;
lut[145] = 8'd192;
lut[146] = 8'd193;
lut[147] = 8'd194;
lut[148] = 8'd194;
lut[149] = 8'd195;
lut[150] = 8'd196;
lut[151] = 8'd196;
lut[152] = 8'd197;
```

```
lut[153] = 8'd198;
lut[154] = 8'd198;
lut[155] = 8'd199;
lut[156] = 8'd199;
lut[157] = 8'd200;
lut[158] = 8'd201;
lut[159] = 8'd201;
lut[160] = 8'd202;
lut[161] = 8'd203;
lut[162] = 8'd203;
lut[163] = 8'd204;
lut[164] = 8'd204;
lut[165] = 8'd205;
lut[166] = 8'd206;
lut[167] = 8'd206;
lut[168] = 8'd207;
lut[169] = 8'd208;
lut[170] = 8'd208;
lut[171] = 8'd209;
lut[172] = 8'd209;
lut[173] = 8'd210;
lut[174] = 8'd211;
lut[175] = 8'd211;
lut[176] = 8'd212;
lut[177] = 8'd212;
lut[178] = 8'd213;
lut[179] = 8'd214;
lut[180] = 8'd214;
lut[181] = 8'd215;
lut[182] = 8'd215;
lut[183] = 8'd216;
lut[184] = 8'd217;
lut[185] = 8'd217;
lut[186] = 8'd218;
lut[187] = 8'd218;
lut[188] = 8'd219;
lut[189] = 8'd220;
lut[190] = 8'd220;
lut[191] = 8'd221;
lut[192] = 8'd221;
lut[193] = 8'd222;
lut[194] = 8'd222;
lut[195] = 8'd223;
lut[196] = 8'd224;
lut[197] = 8'd224;
```

```
lut[198] = 8'd225;
lut[199] = 8'd225;
lut[200] = 8'd226;
lut[201] = 8'd226;
lut[202] = 8'd227;
lut[203] = 8'd228;
lut[204] = 8'd228;
lut[205] = 8'd229;
lut[206] = 8'd229;
lut[207] = 8'd230;
lut[208] = 8'd230;
lut[209] = 8'd231;
lut[210] = 8'd231;
lut[211] = 8'd232;
lut[212] = 8'd233;
lut[213] = 8'd233;
lut[214] = 8'd234;
lut[215] = 8'd234;
lut[216] = 8'd235;
lut[217] = 8'd235;
lut[218] = 8'd236;
lut[219] = 8'd236;
lut[220] = 8'd237;
lut[221] = 8'd237;
lut[222] = 8'd238;
lut[223] = 8'd238;
lut[224] = 8'd239;
lut[225] = 8'd240;
lut[226] = 8'd240;
lut[227] = 8'd241;
lut[228] = 8'd241;
lut[229] = 8'd242;
lut[230] = 8'd242;
lut[231] = 8'd243;
lut[232] = 8'd243;
lut[233] = 8'd244;
lut[234] = 8'd244;
lut[235] = 8'd245;
lut[236] = 8'd245;
lut[237] = 8'd246;
lut[238] = 8'd246;
lut[239] = 8'd247;
lut[240] = 8'd247;
lut[241] = 8'd248;
lut[242] = 8'd248;
```

```
      lut[243] = 8'd249;
      lut[244] = 8'd249;
      lut[245] = 8'd250;
      lut[246] = 8'd250;
      lut[247] = 8'd251;
      lut[248] = 8'd251;
      lut[249] = 8'd252;
      lut[250] = 8'd252;
      lut[251] = 8'd253;
      lut[252] = 8'd253;
      lut[253] = 8'd254;
      lut[254] = 8'd254;
      lut[255] = 8'd255;
  end

  always_comb y = lut[idx];
endmodule
```

img_pkg.sv

None

```
// Shared package for Sobel pipeline constants (match m10test 128×128
framebuffer).
package img_pkg;
  parameter int IMG_WIDTH    = 128;
  parameter int IMG_HEIGHT   = 128;
  parameter int PIXEL_W      = 8;
  parameter int FRAME_PIXELS = IMG_WIDTH * IMG_HEIGHT;
  parameter int FRAME_ADDR_W = $clog2(FRAME_PIXELS);
  parameter int ACC_W        = 16;

  typedef enum logic [1:0] {
    ALGO_SOBEL = 2'd0,
    ALGO_LAPL  = 2'd1,
    ALGO_GAMMA = 2'd2
  } algo_sel_t;

  function automatic logic [ACC_W-1:0] abs_s(input logic signed [ACC_W-1:0]
value);
    if (value < 0) begin
      abs_s = -value;
    end else begin
      abs_s = value;
```

```
    end
  endfunction
endpackage
```

laplacian3x3_sharpen.sv

```
None
// Laplacian sharpen on 3×3 window (4-neighbor stencil on center):
// lap = (N + W + E + S) - 4·C,  out = saturate(C - lap)  → high-frequency
emphasis / sharpening.
// Port shape matches svtest/sobel3x3_threshold for drop-in with
laplacian_frame_pipeline.sv.
module laplacian3x3_sharpen #(
    parameter int PIXEL_W = img_pkg::PIXEL_W,
    parameter int ACC_W   = img_pkg::ACC_W,
    parameter int ADDR_W  = img_pkg::FRAME_ADDR_W
) (
    input  logic clk,
    input  logic rst_n,
    input  logic in_valid,
    input  logic [ADDR_W-1:0] in_dst_addr,
    input  logic [PIXEL_W-1:0] w00,
    input  logic [PIXEL_W-1:0] w01,
    input  logic [PIXEL_W-1:0] w02,
    input  logic [PIXEL_W-1:0] w10,
    input  logic [PIXEL_W-1:0] w11,
    input  logic [PIXEL_W-1:0] w12,
    input  logic [PIXEL_W-1:0] w20,
    input  logic [PIXEL_W-1:0] w21,
    input  logic [PIXEL_W-1:0] w22,
    output logic out_valid,
    output logic [7:0] out_pixel,
    output logic [ADDR_W-1:0] out_ram_addr
);
  logic signed [ACC_W-1:0] lap_c;
  logic signed [ACC_W-1:0] sharp_c;

  always_comb begin
    lap_c = $signed({1'b0, w01}) + $signed({1'b0, w10}) + $signed({1'b0, w12}) +
            $signed({1'b0, w21}) - ($signed({1'b0, w11}) <<< 2);
    sharp_c = $signed({1'b0, w11}) - lap_c;
  end
```

```
  function automatic logic [7:0] clamp_u8(input logic signed [ACC_W-1:0] v);
    if (v < 0)
      clamp_u8 = 8'h00;
    else if (v > 16'sd255)
      clamp_u8 = 8'hFF;
    else
      clamp_u8 = v[7:0];
  endfunction

  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      out_valid    <= 1'b0;
      out_pixel    <= '0;
      out_ram_addr <= '0;
    end else begin
      out_valid <= in_valid;
      out_pixel <= clamp_u8(sharp_c);
      out_ram_addr <= in_dst_addr;
    end
  end
endmodule
```

laplacian_frame_pipeline.sv

None

```
// Streaming Laplacian-sharpen pipeline (same shell as svtest Sobel path).
module laplacian_frame_pipeline #(
    parameter int IMG_WIDTH  = img_pkg::IMG_WIDTH,
    parameter int IMG_HEIGHT = img_pkg::IMG_HEIGHT,
    parameter int ADDR_W     = img_pkg::FRAME_ADDR_W,
    parameter int CTR_OFF    = img_pkg::IMG_WIDTH + 1,
    parameter int PIXEL_W    = img_pkg::PIXEL_W
) (
    input logic clk,
    input logic rst_n,

    input logic                                  frame_wr_en,
    input logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] frame_wr_addr,
    input logic [7:0]                             frame_wr_data,

    input logic host_frame_done_pulse,
```

```
    output logic                                    out_valid,
    output logic [7:0]                              out_pixel,
    output logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] out_ram_addr,
    output logic                                    sched_done
);

  logic h1, h2;
  logic process_start;

  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      h1 <= 1'b0;
      h2 <= 1'b0;
    end else begin
      h2 <= h1;
      h1 <= host_frame_done_pulse;
    end
  end

  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n)
      process_start <= 1'b0;
    else
      process_start <= h1 & ~h2;
  end

  logic rd_en;
  logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] rd_addr;
  logic [7:0] rd_data;

  logic sched_pix_valid;
  logic [7:0] sched_pix_data;
  logic [$clog2(IMG_WIDTH)-1:0]  sched_col;
  logic [$clog2(IMG_HEIGHT)-1:0] sched_row;

  logic line_taps_valid;
  logic [7:0] row_m2, row_m1, row_0, row_p1_preload_unused;

  logic                          line_taps_valid_q;
  logic                          line_taps_valid_q2;
  logic [PIXEL_W-1:0]            lb_row_m2, lb_row_m1, lb_row_0;
  logic [PIXEL_W-1:0]            lb_row_m2_q2, lb_row_m1_q2, lb_row_0_q2;
  logic [$clog2(IMG_WIDTH)-1:0]  lb_col;
  logic [$clog2(IMG_WIDTH)-1:0]  lb_col_q2;
  logic [$clog2(IMG_HEIGHT)-1:0] lb_row;
```

```
logic [$clog2(IMG_HEIGHT)-1:0]   lb_row_q2;

logic win_valid;
logic [7:0] w00, w01, w02, w10, w11, w12, w20, w21, w22;
logic lap_valid;
logic [7:0] lap_pixel;
logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] lap_ram_addr;
logic sched_running_unused;

wire [ADDR_W-1:0] stream_lin =
    ADDR_W'(lb_row_q2) * ADDR_W'(IMG_WIDTH) + ADDR_W'(lb_col_q2);
wire [ADDR_W-1:0] ctr_lin = stream_lin - ADDR_W'(CTR_OFF);

frame_m10k_store #(
    .IMG_WIDTH (IMG_WIDTH),
    .IMG_HEIGHT(IMG_HEIGHT),
    .PIXEL_W  (8)
) u_frame_m10k_store (
    .clk    (clk),
    .rst_n  (rst_n),
    .wr_en  (frame_wr_en),
    .wr_addr(frame_wr_addr),
    .wr_data(frame_wr_data),
    .rd_en  (rd_en),
    .rd_addr(rd_addr),
    .rd_data(rd_data)
);

frame_read_scheduler #(
    .IMG_WIDTH (IMG_WIDTH),
    .IMG_HEIGHT(IMG_HEIGHT),
    .PIXEL_W  (8)
) u_frame_read_scheduler (
    .clk       (clk),
    .rst_n     (rst_n),
    .start     (process_start),
    .running   (sched_running_unused),
    .done      (sched_done),
    .rd_en     (rd_en),
    .rd_addr   (rd_addr),
    .rd_data   (rd_data),
    .pix_valid (sched_pix_valid),
    .pix_data  (sched_pix_data),
    .pix_col   (sched_col),
    .pix_row   (sched_row)
```

```
);

line4_m10k_buffer #(
    .IMG_WIDTH (IMG_WIDTH),
    .IMG_HEIGHT(IMG_HEIGHT),
    .PIXEL_W  (8)
) u_line4_m10k_buffer (
    .clk(clk),
    .rst_n(rst_n),
    .in_valid(sched_pix_valid),
    .in_pixel(sched_pix_data),
    .in_col(sched_col),
    .in_row(sched_row),
    .taps_valid(line_taps_valid),
    .row_m2(row_m2),
    .row_m1(row_m1),
    .row_0(row_0),
    .row_p1_preload(row_p1_preload_unused)
);

always_ff @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    line_taps_valid_q  <= 1'b0;
    line_taps_valid_q2 <= 1'b0;

    lb_row_m2    <= '0;
    lb_row_m1    <= '0;
    lb_row_0     <= '0;
    lb_col       <= '0;
    lb_row       <= '0;

    lb_row_m2_q2 <= '0;
    lb_row_m1_q2 <= '0;
    lb_row_0_q2  <= '0;
    lb_col_q2    <= '0;
    lb_row_q2    <= '0;
  end else begin
    line_taps_valid_q  <= line_taps_valid;
    lb_row_m2          <= row_m2;
    lb_row_m1          <= row_m1;
    lb_row_0           <= row_0;
    lb_col             <= sched_col;
    lb_row             <= sched_row;

    line_taps_valid_q2 <= line_taps_valid_q;
```

```
      lb_row_m2_q2       <= lb_row_m2;
      lb_row_m1_q2       <= lb_row_m1;
      lb_row_0_q2        <= lb_row_0;
      lb_col_q2          <= lb_col;
      lb_row_q2          <= lb_row;
    end
  end

  window3x3_regs #(
      .IMG_WIDTH (IMG_WIDTH),
      .PIXEL_W  (8)
  ) u_window3x3_regs (
      .clk(clk),
      .rst_n(rst_n),
      .in_valid(line_taps_valid_q2),
      .row_m2(lb_row_m2_q2),
      .row_m1(lb_row_m1_q2),
      .row_0(lb_row_0_q2),
      .in_col(lb_col_q2),
      .win_valid(win_valid),
      .w00(w00), .w01(w01), .w02(w02),
      .w10(w10), .w11(w11), .w12(w12),
      .w20(w20), .w21(w21), .w22(w22)
  );

  laplacian3x3_sharpen u_laplacian3x3_sharpen (
      .clk         (clk),
      .rst_n       (rst_n),
      .in_valid    (win_valid),
      .in_dst_addr (ctr_lin),
      .w00(w00), .w01(w01), .w02(w02),
      .w10(w10), .w11(w11), .w12(w12),
      .w20(w20), .w21(w21), .w22(w22),
      .out_valid   (lap_valid),
      .out_pixel   (lap_pixel),
      .out_ram_addr(lap_ram_addr)
  );

  assign out_valid    = lap_valid;
  assign out_pixel    = lap_pixel;
  assign out_ram_addr = lap_ram_addr;
endmodule
```

line4_m10k_buffer.sv

None

```
module line4_m10k_buffer #(
  parameter int IMG_WIDTH  = img_pkg::IMG_WIDTH,
  parameter int IMG_HEIGHT = img_pkg::IMG_HEIGHT,
  parameter int PIXEL_W    = img_pkg::PIXEL_W
) (
  input  logic clk,
  input  logic rst_n,
  input  logic in_valid,
  input  logic [PIXEL_W-1:0] in_pixel,
  input  logic [$clog2(IMG_WIDTH)-1:0] in_col,
  input  logic [$clog2(IMG_HEIGHT)-1:0] in_row,
  output logic taps_valid,
  output logic [PIXEL_W-1:0] row_m2,
  output logic [PIXEL_W-1:0] row_m1,
  output logic [PIXEL_W-1:0] row_0,
  output logic [PIXEL_W-1:0] row_p1_preload
);
  (* ramstyle = "M10K" *) logic [PIXEL_W-1:0] line0 [0:IMG_WIDTH-1];
  (* ramstyle = "M10K" *) logic [PIXEL_W-1:0] line1 [0:IMG_WIDTH-1];
  (* ramstyle = "M10K" *) logic [PIXEL_W-1:0] line2 [0:IMG_WIDTH-1];
  (* ramstyle = "M10K" *) logic [PIXEL_W-1:0] line3 [0:IMG_WIDTH-1];

  logic [1:0] bank_cur;
  logic [1:0] bank_m1;
  logic [1:0] bank_m2;
  logic [1:0] bank_p1;

  assign bank_cur = in_row[1:0];
  assign bank_m1  = bank_cur - 2'd1;
  assign bank_m2  = bank_cur - 2'd2;
  assign bank_p1  = bank_cur + 2'd1;

  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      taps_valid     <= 1'b0;
      row_m2         <= '0;
      row_m1         <= '0;
      row_0          <= '0;
      row_p1_preload <= '0;
    end else begin
      taps_valid <= 1'b0;
      if (in_valid) begin
        case (bank_cur)
          2'd0: line0[in_col] <= in_pixel;
```

```
          2'd1: line1[in_col] <= in_pixel;
          2'd2: line2[in_col] <= in_pixel;
          default: line3[in_col] <= in_pixel;
        endcase

        if (in_row >= 2) begin
          taps_valid <= 1'b1;
          case (bank_m2)
            2'd0: row_m2 <= line0[in_col];
            2'd1: row_m2 <= line1[in_col];
            2'd2: row_m2 <= line2[in_col];
            default: row_m2 <= line3[in_col];
          endcase
          case (bank_m1)
            2'd0: row_m1 <= line0[in_col];
            2'd1: row_m1 <= line1[in_col];
            2'd2: row_m1 <= line2[in_col];
            default: row_m1 <= line3[in_col];
          endcase
          row_0 <= in_pixel;
          case (bank_p1)
            2'd0: row_p1_preload <= line0[in_col];
            2'd1: row_p1_preload <= line1[in_col];
            2'd2: row_p1_preload <= line2[in_col];
            default: row_p1_preload <= line3[in_col];
          endcase
        end
      end
    end
  end
endmodule
```

passthrough_frame_pipeline.sv

```
None
// Passthrough: stream uploaded grayscale to display RAM (identity on pixel
value).
// Same ports as gamma_frame_pipeline for muxed image_processor.
module passthrough_frame_pipeline #(
    parameter int IMG_WIDTH  = img_pkg::IMG_WIDTH,
    parameter int IMG_HEIGHT = img_pkg::IMG_HEIGHT,
    parameter int ADDR_W     = img_pkg::FRAME_ADDR_W,
    parameter int PIXEL_W    = img_pkg::PIXEL_W
```

```
) (
    input logic clk,
    input logic rst_n,

    input logic                                     frame_wr_en,
    input logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] frame_wr_addr,
    input logic [7:0]                               frame_wr_data,

    input logic host_frame_done_pulse,

    output logic                                    out_valid,
    output logic [7:0]                              out_pixel,
    output logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] out_ram_addr,
    output logic                                    sched_done
);

  logic h1, h2;
  logic process_start;

  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      h1 <= 1'b0;
      h2 <= 1'b0;
    end else begin
      h2 <= h1;
      h1 <= host_frame_done_pulse;
    end
  end

  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n)
      process_start <= 1'b0;
    else
      process_start <= h1 & ~h2;
  end

  logic rd_en;
  logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] rd_addr;
  logic [7:0] rd_data;

  logic sched_pix_valid;
  logic [7:0] sched_pix_data;
  logic [$clog2(IMG_WIDTH)-1:0]  sched_col;
  logic [$clog2(IMG_HEIGHT)-1:0] sched_row;
```

```
logic sched_running_unused;

wire [ADDR_W-1:0] stream_lin =
    ADDR_W'(sched_row) * ADDR_W'(IMG_WIDTH) + ADDR_W'(sched_col);

frame_m10k_store #(
    .IMG_WIDTH (IMG_WIDTH),
    .IMG_HEIGHT(IMG_HEIGHT),
    .PIXEL_W  (8)
) u_frame_m10k_store (
    .clk    (clk),
    .rst_n  (rst_n),
    .wr_en  (frame_wr_en),
    .wr_addr(frame_wr_addr),
    .wr_data(frame_wr_data),
    .rd_en  (rd_en),
    .rd_addr(rd_addr),
    .rd_data(rd_data)
);

frame_read_scheduler #(
    .IMG_WIDTH (IMG_WIDTH),
    .IMG_HEIGHT(IMG_HEIGHT),
    .PIXEL_W  (8)
) u_frame_read_scheduler (
    .clk      (clk),
    .rst_n    (rst_n),
    .start    (process_start),
    .running  (sched_running_unused),
    .done     (sched_done),
    .rd_en    (rd_en),
    .rd_addr  (rd_addr),
    .rd_data  (rd_data),
    .pix_valid(sched_pix_valid),
    .pix_data (sched_pix_data),
    .pix_col  (sched_col),
    .pix_row  (sched_row)
);

always_ff @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    out_valid    <= 1'b0;
    out_pixel    <= '0;
    out_ram_addr <= '0;
  end else begin
```

```
      out_valid <= sched_pix_valid;
      if (sched_pix_valid) begin
        out_pixel    <= sched_pix_data;
        out_ram_addr <= stream_lin;
      end
    end
  end

endmodule
```

sobel3x3_threshold.sv

```
None
module sobel3x3_threshold #(
  parameter int IMG_WIDTH  = img_pkg::IMG_WIDTH,
  parameter int IMG_HEIGHT = img_pkg::IMG_HEIGHT,
  parameter int PIXEL_W    = img_pkg::PIXEL_W,
  parameter int ACC_W      = img_pkg::ACC_W,
  parameter int ADDR_W     = img_pkg::FRAME_ADDR_W
) (
  input  logic clk,
  input  logic rst_n,
  input  logic in_valid,
  /** Sobel-center linear index (= stream index - (W+1)); valid when `in_valid`.
*/
  input  logic [ADDR_W-1:0] in_dst_addr,
  input  logic [PIXEL_W-1:0] w00,
  input  logic [PIXEL_W-1:0] w01,
  input  logic [PIXEL_W-1:0] w02,
  input  logic [PIXEL_W-1:0] w10,
  input  logic [PIXEL_W-1:0] w11,
  input  logic [PIXEL_W-1:0] w12,
  input  logic [PIXEL_W-1:0] w20,
  input  logic [PIXEL_W-1:0] w21,
  input  logic [PIXEL_W-1:0] w22,
  input  logic [7:0] threshold,
  output logic out_valid,
  output logic [7:0] out_pixel,
  output logic [ADDR_W-1:0] out_ram_addr
);
  logic signed [ACC_W-1:0] gx_c;
  logic signed [ACC_W-1:0] gy_c;
  logic [ACC_W-1:0] mag_c;
```

```
  always_comb begin
    gx_c = '0;
    gy_c = '0;
    gx_c += -$signed({1'b0, w00});
    gx_c +=  $signed({1'b0, w02});
    gx_c += -($signed({1'b0, w10}) <<< 1);
    gx_c +=  ($signed({1'b0, w12}) <<< 1);
    gx_c += -$signed({1'b0, w20});
    gx_c +=  $signed({1'b0, w22});

    gy_c += -$signed({1'b0, w00});
    gy_c += -($signed({1'b0, w01}) <<< 1);
    gy_c += -$signed({1'b0, w02});
    gy_c +=  $signed({1'b0, w20});
    gy_c +=  ($signed({1'b0, w21}) <<< 1);
    gy_c +=  $signed({1'b0, w22});

    mag_c = img_pkg::abs_s(gx_c) + img_pkg::abs_s(gy_c);
  end

  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      out_valid <= 1'b0;
      out_pixel <= '0;
      out_ram_addr <= '0;
    end else begin
      out_valid <= in_valid;
      out_pixel <= (mag_c > threshold) ? 8'hFF : 8'h00;
      out_ram_addr <= in_dst_addr;
    end
  end
endmodule
```

sobel_frame_pipeline.sv

```
None
// Streaming Sobel pipeline (no Avalon). External logic writes the input
frame_ram;
// one-cycle host_frame_done_pulse starts the scheduler (same as
image_pipeline_top).
module sobel_frame_pipeline #(
  parameter int IMG_WIDTH       = img_pkg::IMG_WIDTH,
```

```
  parameter int IMG_HEIGHT      = img_pkg::IMG_HEIGHT,
  parameter int SOBEL_THRESHOLD = 100,
  parameter int ADDR_W          = img_pkg::FRAME_ADDR_W,
  parameter int CTR_OFF         = img_pkg::IMG_WIDTH + 1,
  parameter int PIXEL_W         = img_pkg::PIXEL_W
) (
  input logic clk,
  input logic rst_n,

  input logic                                     frame_wr_en,
  input logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] frame_wr_addr,
  input logic [7:0]                               frame_wr_data,

  input logic host_frame_done_pulse,

  output logic                                     out_valid,
  output logic [7:0]                               out_pixel,
  output logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] out_ram_addr,
  output logic                                     sched_done
);
  logic h1, h2;
  logic process_start;

  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      h1 <= 1'b0;
      h2 <= 1'b0;
    end else begin
      h2 <= h1;
      h1 <= host_frame_done_pulse;
    end
  end

  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n)
      process_start <= 1'b0;
    else
      process_start <= h1 & ~h2;
  end

  logic rd_en;
  logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] rd_addr;
  logic [7:0] rd_data;

  logic sched_pix_valid;
```

```
  logic [7:0] sched_pix_data;
  logic [$clog2(IMG_WIDTH)-1:0]  sched_col;
  logic [$clog2(IMG_HEIGHT)-1:0] sched_row;

  logic line_taps_valid;
  logic [7:0] row_m2, row_m1, row_0, row_p1_preload_unused;

  //------------------------------------------------------------------
  // Two-cycle skid after line_buffer (covers BRAM read / tool ordering).
  // CRITICAL: ctr_lin MUST use the same pipeline stage as row_* and in_col
  // into window3x3_regs. Using lb_row/lb_col while feeding _q2 taps puts
  // addresses one row/column phase off → diagonal shear and ghost copies.
  //------------------------------------------------------------------
  logic                           line_taps_valid_q;
  logic                           line_taps_valid_q2;
  logic [PIXEL_W-1:0]             lb_row_m2, lb_row_m1, lb_row_0;
  logic [PIXEL_W-1:0]             lb_row_m2_q2, lb_row_m1_q2, lb_row_0_q2;
  logic [$clog2(IMG_WIDTH)-1:0]   lb_col;
  logic [$clog2(IMG_WIDTH)-1:0]   lb_col_q2;
  logic [$clog2(IMG_HEIGHT)-1:0]  lb_row;
  logic [$clog2(IMG_HEIGHT)-1:0]  lb_row_q2;

  logic win_valid;
  logic [7:0] w00, w01, w02, w10, w11, w12, w20, w21, w22;
  logic sobel_valid;
  logic [7:0] sobel_pixel;
  logic [$clog2(IMG_WIDTH*IMG_HEIGHT)-1:0] sobel_ram_addr;
  logic sched_running_unused;

  /** Sobel-center row-major index (must match window input stage _q2). */
  wire [ADDR_W-1:0] stream_lin = ADDR_W'(lb_row_q2) * ADDR_W'(IMG_WIDTH) +
ADDR_W'(lb_col_q2);
  wire [ADDR_W-1:0] ctr_lin    = stream_lin - ADDR_W'(CTR_OFF);

  frame_m10k_store #(
    .IMG_WIDTH (IMG_WIDTH),
    .IMG_HEIGHT(IMG_HEIGHT),
    .PIXEL_W  (8)
  ) u_frame_m10k_store (
    .clk    (clk),
    .rst_n  (rst_n),
    .wr_en  (frame_wr_en),
    .wr_addr(frame_wr_addr),
    .wr_data(frame_wr_data),
    .rd_en  (rd_en),
```

```
  .rd_addr(rd_addr),
  .rd_data(rd_data)
);

frame_read_scheduler #(
  .IMG_WIDTH (IMG_WIDTH),
  .IMG_HEIGHT(IMG_HEIGHT),
  .PIXEL_W  (8)
) u_frame_read_scheduler (
  .clk       (clk),
  .rst_n     (rst_n),
  .start     (process_start),
  .running   (sched_running_unused),
  .done      (sched_done),
  .rd_en     (rd_en),
  .rd_addr   (rd_addr),
  .rd_data   (rd_data),
  .pix_valid (sched_pix_valid),
  .pix_data  (sched_pix_data),
  .pix_col   (sched_col),
  .pix_row   (sched_row)
);

line4_m10k_buffer #(
  .IMG_WIDTH (IMG_WIDTH),
  .IMG_HEIGHT(IMG_HEIGHT),
  .PIXEL_W  (8)
) u_line4_m10k_buffer (
  .clk(clk),
  .rst_n(rst_n),
  .in_valid(sched_pix_valid),
  .in_pixel(sched_pix_data),
  .in_col(sched_col),
  .in_row(sched_row),
  .taps_valid(line_taps_valid),
  .row_m2(row_m2),
  .row_m1(row_m1),
  .row_0(row_0),
  .row_p1_preload(row_p1_preload_unused)
);

always_ff @(posedge clk or negedge rst_n) begin
  if (!rst_n) begin
    line_taps_valid_q  <= 1'b0;
    line_taps_valid_q2 <= 1'b0;
```

```
    lb_row_m2   <= '0;
    lb_row_m1   <= '0;
    lb_row_0    <= '0;
    lb_col      <= '0;
    lb_row      <= '0;

    lb_row_m2_q2 <= '0;
    lb_row_m1_q2 <= '0;
    lb_row_0_q2  <= '0;
    lb_col_q2    <= '0;
    lb_row_q2    <= '0;
  end else begin
    line_taps_valid_q  <= line_taps_valid;
    lb_row_m2          <= row_m2;
    lb_row_m1          <= row_m1;
    lb_row_0           <= row_0;
    lb_col             <= sched_col;
    lb_row             <= sched_row;

    line_taps_valid_q2 <= line_taps_valid_q;
    lb_row_m2_q2       <= lb_row_m2;
    lb_row_m1_q2       <= lb_row_m1;
    lb_row_0_q2        <= lb_row_0;
    lb_col_q2          <= lb_col;
    lb_row_q2          <= lb_row;
  end
end

window3x3_regs #(
  .IMG_WIDTH (IMG_WIDTH),
  .PIXEL_W  (8)
) u_window3x3_regs (
  .clk(clk),
  .rst_n(rst_n),
  .in_valid(line_taps_valid_q2),
  .row_m2(lb_row_m2_q2),
  .row_m1(lb_row_m1_q2),
  .row_0(lb_row_0_q2),
  .in_col(lb_col_q2),
  .win_valid(win_valid),
  .w00(w00), .w01(w01), .w02(w02),
  .w10(w10), .w11(w11), .w12(w12),
  .w20(w20), .w21(w21), .w22(w22)
);
```

```
  logic [7:0] thresh8;
  assign thresh8 = SOBEL_THRESHOLD[7:0];

  sobel3x3_threshold u_sobel3x3_threshold (
    .clk(clk),
    .rst_n(rst_n),
    .in_valid(win_valid),
    .in_dst_addr(ctr_lin),
    .w00(w00), .w01(w01), .w02(w02),
    .w10(w10), .w11(w11), .w12(w12),
    .w20(w20), .w21(w21), .w22(w22),
    .threshold(thresh8),
    .out_valid(sobel_valid),
    .out_pixel(sobel_pixel),
    .out_ram_addr(sobel_ram_addr)
  );

  assign out_valid    = sobel_valid;
  assign out_pixel    = sobel_pixel;
  assign out_ram_addr = sobel_ram_addr;
endmodule
```

window3x3_regs.sv

```
None
module window3x3_regs #(
  parameter int IMG_WIDTH = img_pkg::IMG_WIDTH,
  parameter int PIXEL_W   = img_pkg::PIXEL_W
) (
  input  logic clk,
  input  logic rst_n,
  input  logic in_valid,
  input  logic [PIXEL_W-1:0] row_m2,
  input  logic [PIXEL_W-1:0] row_m1,
  input  logic [PIXEL_W-1:0] row_0,
  input  logic [$clog2(IMG_WIDTH)-1:0] in_col,
  output logic win_valid,
  output logic [PIXEL_W-1:0] w00,
  output logic [PIXEL_W-1:0] w01,
  output logic [PIXEL_W-1:0] w02,
  output logic [PIXEL_W-1:0] w10,
  output logic [PIXEL_W-1:0] w11,
```

```
  output logic [PIXEL_W-1:0] w12,
  output logic [PIXEL_W-1:0] w20,
  output logic [PIXEL_W-1:0] w21,
  output logic [PIXEL_W-1:0] w22
);
  logic [PIXEL_W-1:0] r0_s0, r0_s1, r0_s2;
  logic [PIXEL_W-1:0] r1_s0, r1_s1, r1_s2;
  logic [PIXEL_W-1:0] r2_s0, r2_s1, r2_s2;

  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      r0_s0 <= '0; r0_s1 <= '0; r0_s2 <= '0;
      r1_s0 <= '0; r1_s1 <= '0; r1_s2 <= '0;
      r2_s0 <= '0; r2_s1 <= '0; r2_s2 <= '0;
      win_valid <= 1'b0;
    end else begin
      win_valid <= 1'b0;
      if (in_valid) begin
        r0_s0 <= r0_s1; r0_s1 <= r0_s2; r0_s2 <= row_m2;
        r1_s0 <= r1_s1; r1_s1 <= r1_s2; r1_s2 <= row_m1;
        r2_s0 <= r2_s1; r2_s1 <= r2_s2; r2_s2 <= row_0;
        if (in_col >= 2) begin
          win_valid <= 1'b1;
        end
      end
    end
  end

  assign w00 = r0_s0; assign w01 = r0_s1; assign w02 = r0_s2;
  assign w10 = r1_s0; assign w11 = r1_s1; assign w12 = r1_s2;
  assign w20 = r2_s0; assign w21 = r2_s1; assign w22 = r2_s2;
endmodule
```

image_processor.sv

```
None
/*
 * Avalon MM + selectable filter VGA (buttons KEY[3:0], debounced).
 * After host finishes uploading the frame (last pixel write), passthrough runs
once
 * automatically so VGA shows unfiltered grayscale immediately.
 * KEY0 — passthrough again; KEY1 — Sobel; KEY2 — Laplacian; KEY3 — Gamma.
 * KEY inputs are active-low (pressed = 0). Tie unused bits high.
```

```
 */

module true_dual_port_ram_single_clock #(
    parameter int DATA_WIDTH  = 8,
    parameter int ADDR_WIDTH = 14
) (
    input  logic clk,
    input  logic [DATA_WIDTH-1:0] data_a,
    input  logic [DATA_WIDTH-1:0] data_b,
    input  logic [ADDR_WIDTH-1:0] addr_a,
    input  logic [ADDR_WIDTH-1:0] addr_b,
    input  logic we_a,
    input  logic we_b,
    output logic [DATA_WIDTH-1:0] q_a,
    output logic [DATA_WIDTH-1:0] q_b
);
  (* ramstyle = "M10K, no_rw_check" *) logic [DATA_WIDTH-1:0]
      ram[0:(1 << ADDR_WIDTH) - 1];

  always_ff @(posedge clk) begin
    if (we_a) begin
      ram[addr_a] <= data_a;
      q_a <= data_a;
    end else begin
      q_a <= ram[addr_a];
    end
  end

  always_ff @(posedge clk) begin
    if (we_b) begin
      ram[addr_b] <= data_b;
      q_b <= data_b;
    end else begin
      q_b <= ram[addr_b];
    end
  end
endmodule


module image_processor #(
    parameter int IP_ADDR_W = img_pkg::FRAME_ADDR_W
) (
    input  logic clk,
    input  logic reset,
```

```
    input  logic chipselect,
    input  logic write,
    input  logic [IP_ADDR_W-1:0] address,
    input  logic [7:0]           writedata,
    output logic [7:0]           readdata,

    input  logic [3:0] KEY_n,

    output logic [7:0] VGA_R,
    output logic [7:0] VGA_G,
    output logic [7:0] VGA_B,
    output logic VGA_CLK,
    output logic VGA_HS,
    output logic VGA_VS,
    output logic VGA_BLANK_n,
    output logic VGA_SYNC_n
);

  localparam int ADDR_W = IP_ADDR_W;
  localparam logic [ADDR_W-1:0] LAST_IDX =
      ADDR_W'(img_pkg::FRAME_PIXELS) - ADDR_W'(1);

  logic rst_n;
  assign rst_n = ~reset;

  logic [3:0] key_stable;
  debounce_keys #(
      .NUM_KEYS(4),
      .DEBOUNCE_CYCLES(20'd1_000_000)
  ) u_debounce (
      .clk        (clk),
      .reset      (reset),
      .KEY_n      (KEY_n),
      .key_pressed(key_stable)
  );

  logic [3:0] key_prev;
  always_ff @(posedge clk or posedge reset) begin
    if (reset)
      key_prev <= '0;
    else
      key_prev <= key_stable;
  end

  wire [3:0] key_press_edge = key_stable & ~key_prev;
```

```
logic [1:0] mode_enc;
always_comb begin
  mode_enc = 2'd0;
  if (key_press_edge[0])
    mode_enc = 2'd0;
  else if (key_press_edge[1])
    mode_enc = 2'd1;
  else if (key_press_edge[2])
    mode_enc = 2'd2;
  else if (key_press_edge[3])
    mode_enc = 2'd3;
end

logic proc_busy;
logic frame_loaded;

wire addr_ok = (address <= LAST_IDX);
wire host_wr_px = chipselect && write && addr_ok && !proc_busy;
wire last_px_w = chipselect && write && (address == LAST_IDX);

logic last_px_w_d;
always_ff @(posedge clk or posedge reset) begin
  if (reset)
    last_px_w_d <= 1'b0;
  else
    last_px_w_d <= last_px_w;
end
wire last_px_w_rise = last_px_w && !last_px_w_d;

// Auto-run passthrough when upload completes (no key press required).
wire auto_pt_start = last_px_w_rise && !proc_busy;
wire key_start       = |key_press_edge && frame_loaded && !proc_busy;
wire start_pulse     = auto_pt_start | key_start;

wire [1:0] mode_at_start = auto_pt_start ? 2'd0 : mode_enc;

wire pulse_pt  = auto_pt_start || (key_start && mode_enc == 2'd0);
wire pulse_sob = key_start && mode_enc == 2'd1;
wire pulse_lap = key_start && mode_enc == 2'd2;
wire pulse_gam = key_start && mode_enc == 2'd3;

logic [1:0] active_alg;

always_ff @(posedge clk or posedge reset) begin
```

```
  if (reset)
    active_alg <= 2'd0;
  else if (start_pulse)
    active_alg <= mode_at_start;
end

wire [1:0] sel_alg = start_pulse ? mode_at_start : active_alg;

// Four parallel frame stores (same upload broadcast); one processes / frame.
logic pt_valid, sob_valid, lap_valid, gam_valid;
logic [7:0] pt_px, sob_px, lap_px, gam_px;
logic [ADDR_W-1:0] pt_addr, sob_addr, lap_addr, gam_addr;
logic pt_done, sob_done, lap_done, gam_done;

passthrough_frame_pipeline u_pt (
    .clk                   (clk),
    .rst_n                 (rst_n),
    .frame_wr_en           (host_wr_px),
    .frame_wr_addr         (address[ADDR_W-1:0]),
    .frame_wr_data         (writedata),
    .host_frame_done_pulse (pulse_pt),
    .out_valid             (pt_valid),
    .out_pixel             (pt_px),
    .out_ram_addr          (pt_addr),
    .sched_done            (pt_done)
);

sobel_frame_pipeline #(
    .SOBEL_THRESHOLD(100)
) u_sob (
    .clk                   (clk),
    .rst_n                 (rst_n),
    .frame_wr_en           (host_wr_px),
    .frame_wr_addr         (address[ADDR_W-1:0]),
    .frame_wr_data         (writedata),
    .host_frame_done_pulse (pulse_sob),
    .out_valid             (sob_valid),
    .out_pixel             (sob_px),
    .out_ram_addr          (sob_addr),
    .sched_done            (sob_done)
);

laplacian_frame_pipeline u_lap (
    .clk                   (clk),
    .rst_n                 (rst_n),
```

```
    .frame_wr_en           (host_wr_px),
    .frame_wr_addr         (address[ADDR_W-1:0]),
    .frame_wr_data         (writedata),
    .host_frame_done_pulse (pulse_lap),
    .out_valid             (lap_valid),
    .out_pixel             (lap_px),
    .out_ram_addr          (lap_addr),
    .sched_done            (lap_done)
);

gamma_frame_pipeline u_gam (
    .clk                   (clk),
    .rst_n                 (rst_n),
    .frame_wr_en           (host_wr_px),
    .frame_wr_addr         (address[ADDR_W-1:0]),
    .frame_wr_data         (writedata),
    .host_frame_done_pulse (pulse_gam),
    .out_valid             (gam_valid),
    .out_pixel             (gam_px),
    .out_ram_addr          (gam_addr),
    .sched_done            (gam_done)
);

logic mux_valid;
logic [7:0] mux_px;
logic [ADDR_W-1:0] mux_addr;
logic mux_done;

always_comb begin
  mux_valid = 1'b0;
  mux_px    = 8'h00;
  mux_addr  = '0;
  mux_done  = 1'b0;
  case (sel_alg)
    2'd0: begin
      mux_valid = pt_valid;
      mux_px    = pt_px;
      mux_addr  = pt_addr;
      mux_done  = pt_done;
    end
    2'd1: begin
      mux_valid = sob_valid;
      mux_px    = sob_px;
      mux_addr  = sob_addr;
      mux_done  = sob_done;
```

```
    end
    2'd2: begin
      mux_valid = lap_valid;
      mux_px    = lap_px;
      mux_addr  = lap_addr;
      mux_done  = lap_done;
    end
    2'd3: begin
      mux_valid = gam_valid;
      mux_px    = gam_px;
      mux_addr  = gam_addr;
      mux_done  = gam_done;
    end
    default: ;
  endcase
end

logic           result_ready;
logic [ADDR_W-1:0] addr_mux_a;

assign addr_mux_a = proc_busy ? mux_addr : address[ADDR_W-1:0];

logic disp_we_a;
logic [7:0] disp_din_a;

assign disp_we_a = proc_busy & mux_valid;
assign disp_din_a = mux_px;

logic [7:0] disp_q_a;
logic [7:0] vga_q_raw;

logic [ADDR_W-1:0] vga_rd_addr;

true_dual_port_ram_single_clock #(
    .DATA_WIDTH(8),
    .ADDR_WIDTH(ADDR_W)
) u_disp (
    .clk   (clk),
    .data_a(disp_din_a),
    .data_b(8'd0),
    .addr_a(addr_mux_a),
    .addr_b(vga_rd_addr),
    .we_a  (disp_we_a),
    .we_b  (1'b0),
    .q_a   (disp_q_a),
```

```
    .q_b   (vga_q_raw)
);

localparam int unsigned POST_FLUSH_CY = 1024;
localparam int unsigned FLUSH_CNT_W   = $clog2(POST_FLUSH_CY + 1);

logic [FLUSH_CNT_W-1:0] flush_rem;

always_ff @(posedge clk or posedge reset) begin
  if (reset) begin
    proc_busy      <= 1'b0;
    result_ready   <= 1'b0;
    flush_rem      <= '0;
    frame_loaded   <= 1'b0;
  end else begin
    if (host_wr_px && address == '0)
      result_ready <= 1'b0;

    if (host_wr_px && address == '0)
      frame_loaded <= 1'b0;

    if (last_px_w)
      frame_loaded <= 1'b1;

    if (start_pulse) begin
      proc_busy    <= 1'b1;
      flush_rem    <= '0;
    end

    if (proc_busy && mux_done)
      flush_rem <= POST_FLUSH_CY[FLUSH_CNT_W-1:0];
    else if (flush_rem != 0) begin
      flush_rem <= flush_rem - 1'b1;
      if (flush_rem == 1) begin
        proc_busy    <= 1'b0;
        result_ready <= 1'b1;
      end
    end
  end
end

wire host_rd =
    chipselect && !write && addr_ok && result_ready && !proc_busy;

always_ff @(posedge clk or posedge reset) begin
```

```
  if (reset)
    readdata <= 8'h00;
  else if (host_rd)
    readdata <= disp_q_a;
end

logic [10:0] hcount;
logic [9:0] vcount;

vga_counters u_counters (
    .clk50(clk),
    .reset(reset),
    .hcount(hcount),
    .vcount(vcount),
    .VGA_CLK(VGA_CLK),
    .VGA_HS(VGA_HS),
    .VGA_VS(VGA_VS),
    .VGA_BLANK_n(VGA_BLANK_n),
    .VGA_SYNC_n(VGA_SYNC_n)
);

wire       in_win = (hcount[10:1] < 11'd256) && (vcount < 10'd256);
wire [6:0] src_x  = hcount[8:2];
wire [6:0] src_y  = vcount[7:1];
assign vga_rd_addr = ({7'b0, src_y} << 7) + {7'b0, src_x};

logic [7:0] pixel;

always_ff @(posedge clk) begin
  if (in_win)
    pixel <= vga_q_raw;
  else
    pixel <= 8'h00;
end

always_comb begin
  VGA_R = 8'h00;
  VGA_G = 8'h00;
  VGA_B = 8'h00;
  if (VGA_BLANK_n) begin
    VGA_R = pixel;
    VGA_G = pixel;
    VGA_B = pixel;
  end
end
```

```
endmodule


module vga_counters (
    input  logic clk50,
    input  logic reset,
    output logic [10:0] hcount,
    output logic [9:0] vcount,
    output logic VGA_CLK,
    output logic VGA_HS,
    output logic VGA_VS,
    output logic VGA_BLANK_n,
    output logic VGA_SYNC_n);
  parameter HACTIVE      = 11'd1280,
            HFRONT_PORCH = 11'd32,
            HSYNC        = 11'd192,
            HBACK_PORCH  = 11'd96,
            HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH;

  parameter VACTIVE      = 10'd480,
            VFRONT_PORCH = 10'd10,
            VSYNC        = 10'd2,
            VBACK_PORCH  = 10'd33,
            VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH;

  logic endOfLine;
  logic endOfField;

  always_ff @(posedge clk50 or posedge reset)
    if (reset)
      hcount <= 0;
    else if (endOfLine)
      hcount <= 0;
    else
      hcount <= hcount + 11'd1;

  assign endOfLine = hcount == HTOTAL - 1;

  always_ff @(posedge clk50 or posedge reset)
    if (reset)
      vcount <= 0;
    else if (endOfLine)
      if (endOfField)
        vcount <= 0;
      else
```

```
        vcount <= vcount + 10'd1;

  assign endOfField = vcount == VTOTAL - 1;

  assign VGA_HS = !((hcount[10:8] == 3'b101) & !(hcount[7:5] == 3'b111));
  assign VGA_VS = !(vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);
  assign VGA_SYNC_n = 1'b0;
  assign VGA_BLANK_n = !(hcount[10] & (hcount[9] | hcount[8])) &
      !(vcount[9] | (vcount[8:5] == 4'b1111));
  assign VGA_CLK = hcount[0];
endmodule
```

/sw
hello.c

None

```
/*
 * DICOM → 128×128 grayscale → FPGA frame buffer → hardware Sobel → VGA edge map.
 * Protocol matches m10test ioctl writes; FPGA starts Sobel on the write to pixel
 * (IMAGE_PIXELS - 1). Allow time for the pipeline before the next capture.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include "image_processor.h"

#define HOUNSFIELD_OFFSET 1024
#define DICOM_WIDTH  512
#define DICOM_HEIGHT 512

#ifndef DICOM_PATH
#define DICOM_PATH "/mnt/22970"
#endif

typedef struct {
```

```
	int16_t *data;
	int	width, height;
	int16_t  min_val, max_val;
} Image;

static inline uint16_t compress_hounsfield(int16_t val)
{
	int32_t shifted = (int32_t)val + HOUNSFIELD_OFFSET;

	if (shifted < 0) {
		shifted = 0;
		printf("Warning: HU value %d underflowed to 0\n", val);
	} else if (shifted > 0xFFF) {
		shifted = 0xFFF;
		printf("Warning: HU value %d overflowed to 4095\n", val);
	}
	return (uint16_t)(shifted & 0xFFF);
}

static inline uint8_t normalize_to_8bit(uint16_t val, uint16_t min_val, uint16_t
max_val)
{
	int32_t range = max_val - min_val;
	if (range == 0)
		return 0;
	int32_t normalized = ((int32_t)(val - min_val) * 255) / range;
	if (normalized < 0)
		normalized = 0;
	if (normalized > 255)
		normalized = 255;
	return (uint8_t)normalized;
}

static long find_pixel_data_offset(FILE *fp)
{
	fseek(fp, 0, SEEK_END);
	long file_size = ftell(fp);
	fseek(fp, 0, SEEK_SET);

	uint8_t *buf = malloc(file_size);
	if (!buf) {
		fprintf(stderr, "FATAL: could not allocate %ld bytes\n", file_size);
		return -1;
	}
	fread(buf, 1, file_size, fp);
```

```
	long pixel_start = -1;

	for (long i = 0; i < file_size - 12; i++) {
		if (buf[i] == 0xE0 && buf[i + 1] == 0x7F &&
		    buf[i + 2] == 0x10 && buf[i + 3] == 0x00) {

			fprintf(stderr, "Pixel data tag found at offset %ld\n", i);

			if (buf[i + 4] == 'O' && (buf[i + 5] == 'W' || buf[i + 5] ==
'B')) {
				uint32_t len;
				memcpy(&len, &buf[i + 8], 4);
				fprintf(stderr, "Explicit VR, length=%u bytes\n", len);
				pixel_start = i + 12;
			} else {
				uint32_t len;
				memcpy(&len, &buf[i + 4], 4);
				fprintf(stderr, "Implicit VR, length=%u bytes\n", len);
				pixel_start = i + 8;
			}

			fprintf(stderr, "Pixel data starts at offset %ld\n",
pixel_start);
			break;
		}
	}

	free(buf);
	return pixel_start;
}

static Image *dicom_load(const char *filename)
{
	FILE *fp = fopen(filename, "rb");
	if (!fp) {
		perror("fopen");
		return NULL;
	}

	uint8_t preamble[132];
	if (fread(preamble, 1, 132, fp) != 132) {
		fprintf(stderr, "File too small to be DICOM\n");
		fclose(fp);
		return NULL;
```

```
}
if (memcmp(preamble + 128, "DICM", 4) != 0) {
       fprintf(stderr, "Warning: No DICM magic — trying raw read\n");
       rewind(fp);
} else {
       fprintf(stderr, "DICM magic confirmed\n");
}

long pixel_offset = find_pixel_data_offset(fp);
if (pixel_offset < 0) {
       fprintf(stderr, "FATAL: Could not find pixel data tag\n");
       fclose(fp);
       return NULL;
}
fseek(fp, pixel_offset, SEEK_SET);

Image *img = malloc(sizeof(Image));
if (!img) {
       fclose(fp);
       return NULL;
}
img->width  = DICOM_WIDTH;
img->height = DICOM_HEIGHT;
img->data   = malloc(DICOM_WIDTH * DICOM_HEIGHT * sizeof(int16_t));
if (!img->data) {
       free(img);
       fclose(fp);
       return NULL;
}

size_t got = fread(img->data, sizeof(int16_t),
                DICOM_WIDTH * DICOM_HEIGHT, fp);
if (got != (size_t)(DICOM_WIDTH * DICOM_HEIGHT))
       fprintf(stderr, "Warning: short read — got %zu pixels\n", got);

for (int i = 0; i < DICOM_WIDTH * DICOM_HEIGHT; i++)
       img->data[i] -= HOUNSFIELD_OFFSET;

img->min_val = img->data[0];
img->max_val = img->data[0];
for (int i = 1; i < DICOM_WIDTH * DICOM_HEIGHT; i++) {
       if (img->data[i] < img->min_val)
              img->min_val = img->data[i];
       if (img->data[i] > img->max_val)
              img->max_val = img->data[i];
```

```
	}

	printf("DICOM loaded: %dx%d, min=%d HU, max=%d HU\n",
	       img->width, img->height, img->min_val, img->max_val);

	fclose(fp);
	return img;
}

static void image_free(Image *img)
{
	if (img) {
		free(img->data);
		free(img);
	}
}

int main(int argc, char *argv[])
{
	const char *dicom_file = DICOM_PATH;
	if (argc >= 2)
		dicom_file = argv[1];

	int fd;
	image_processor_arg_t ipa;

	printf("svtest: DICOM -> %dx%d -> FPGA Sobel -> VGA (threshold in RTL)\n",
	       IMAGE_WIDTH, IMAGE_HEIGHT);

	Image *img = dicom_load(dicom_file);
	if (!img) {
		fprintf(stderr, "Failed to load DICOM: %s\n", dicom_file);
		return -1;
	}

	if ((fd = open("/dev/image_processor", O_RDWR)) == -1) {
		perror("open /dev/image_processor");
		image_free(img);
		return -1;
	}

	uint16_t cmin = 0xFFFF, cmax = 0;
	for (int i = 0; i < DICOM_WIDTH * DICOM_HEIGHT; i++) {
		uint16_t c = compress_hounsfield(img->data[i]);
		if (c < cmin)
```

```
                cmin = c;
            if (c > cmax)
                cmax = c;
        }

        printf("Sending %dx%d frame (ioctl per pixel); last write starts Sobel…\n",
               IMAGE_WIDTH, IMAGE_HEIGHT);

        for (int y = 0; y < IMAGE_HEIGHT; y++) {
            for (int x = 0; x < IMAGE_WIDTH; x++) {
                int src_x = (x * DICOM_WIDTH) / IMAGE_WIDTH;
                int src_y = (y * DICOM_HEIGHT) / IMAGE_HEIGHT;
                int dicom_idx = src_y * DICOM_WIDTH + src_x;
                int hw_idx = y * IMAGE_WIDTH + x;

                uint16_t compressed =
compress_hounsfield(img->data[dicom_idx]);
                ipa.offset = (uint16_t)hw_idx;
                ipa.value = normalize_to_8bit(compressed, cmin, cmax);
                if (ioctl(fd, IMAGE_PROCESSOR_WRITE_PIXEL, &ipa) < 0) {
                    perror("ioctl IMAGE_PROCESSOR_WRITE_PIXEL");
                    image_free(img);
                    close(fd);
                    return -1;
                }
            }
        }

        /*
         * Allow the FPGA to finish streaming Sobel output into display RAM (~ms).
         * Tight polling is possible via READ_PIXEL once result_ready asserts in HW.
         */
        usleep(200 * 1000);

        printf("Done (Sobel output should appear in the VGA window).\n");
        close(fd);
        image_free(img);
        return 0;
}
```

image_processor.c

None

```
/* Linux driver — byte-MMIO framebuffer + Sobel result readout (svtest FPGA). */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/mm.h>

#include "image_processor.h"

#define DRIVER_NAME "image_processor"

struct image_processor_dev {
	struct resource res;
	void __iomem *virtbase;
} dev;

static inline void ip_writeb(u8 value, u16 reg_off)
{
	iowrite8(value, dev.virtbase + reg_off);
}

static inline u8 ip_readb(u16 reg_off)
{
	return ioread8(dev.virtbase + reg_off);
}

static long image_processor_ioctl(struct file *f, unsigned int cmd, unsigned long
arg)
{
	image_processor_arg_t ipa;

	switch (cmd) {
	case IMAGE_PROCESSOR_WRITE_PIXEL:
		if (copy_from_user(&ipa, (image_processor_arg_t __user *)arg,
sizeof(ipa)))
			return -EACCES;
```

```
              if (ipa.offset >= IMAGE_PIXELS)
                     return -EINVAL;
              ip_writeb(ipa.value, ipa.offset);
              break;

       case IMAGE_PROCESSOR_READ_PIXEL:
              if (copy_from_user(&ipa, (image_processor_arg_t __user *)arg,
sizeof(ipa)))
                     return -EACCES;
              if (ipa.offset >= IMAGE_PIXELS)
                     return -EINVAL;
              ipa.value = ip_readb(ipa.offset);
              if (copy_to_user((image_processor_arg_t __user *)arg, &ipa,
sizeof(ipa)))
                     return -EACCES;
              break;

       default:
              return -EINVAL;
       }

       return 0;
}

static int image_processor_mmap(struct file *f, struct vm_area_struct *vma)
{
       unsigned long size = vma->vm_end - vma->vm_start;

       if (size > PAGE_SIZE)
              return -EINVAL;

       vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);

       if (remap_pfn_range(vma,
                           vma->vm_start,
                           dev.res.start >> PAGE_SHIFT,
                           size,
                           vma->vm_page_prot))
              return -EAGAIN;

       return 0;
}

static const struct file_operations image_processor_fops = {
       .owner          = THIS_MODULE,
```

```
	.unlocked_ioctl = image_processor_ioctl,
	.mmap           = image_processor_mmap,
};

static struct miscdevice image_processor_misc_device = {
	.minor = MISC_DYNAMIC_MINOR,
	.name  = DRIVER_NAME,
	.fops  = &image_processor_fops,
};

static int __init image_processor_probe(struct platform_device *pdev)
{
	int ret;

	ret = misc_register(&image_processor_misc_device);
	if (ret) {
		pr_err(DRIVER_NAME ": misc_register failed\n");
		return ret;
	}

	ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
	if (ret) {
		ret = -ENOENT;
		goto out_deregister;
	}

	if (request_mem_region(dev.res.start, resource_size(&dev.res), DRIVER_NAME)
== NULL) {
		ret = -EBUSY;
		goto out_deregister;
	}

	dev.virtbase = of_iomap(pdev->dev.of_node, 0);
	if (dev.virtbase == NULL) {
		ret = -ENOMEM;
		goto out_release_mem_region;
	}

	pr_info(DRIVER_NAME ": svtest Sobel path, MMIO phys=0x%08lx len=%llu\n",
		(unsigned long)dev.res.start,
		(unsigned long long)resource_size(&dev.res));
	return 0;

out_release_mem_region:
	release_mem_region(dev.res.start, resource_size(&dev.res));
```

```
out_deregister:
	misc_deregister(&image_processor_misc_device);
	return ret;
}

static int image_processor_remove(struct platform_device *pdev)
{
	iounmap(dev.virtbase);
	release_mem_region(dev.res.start, resource_size(&dev.res));
	misc_deregister(&image_processor_misc_device);
	return 0;
}

#ifdef CONFIG_OF
static const struct of_device_id image_processor_of_match[] = {
	{ .compatible = "csee4840,image_processor-1.0" },
	{},
};
MODULE_DEVICE_TABLE(of, image_processor_of_match);
#endif

static struct platform_driver image_processor_driver = {
	.driver = {
		.name		= DRIVER_NAME,
		.owner		= THIS_MODULE,
		.of_match_table = of_match_ptr(image_processor_of_match),
	},
	.remove = __exit_p(image_processor_remove),
};

static int __init image_processor_init(void)
{
	pr_info(DRIVER_NAME ": init (svtest Sobel+VGA)\n");
	return platform_driver_probe(&image_processor_driver,
image_processor_probe);
}

static void __exit image_processor_exit(void)
{
	platform_driver_unregister(&image_processor_driver);
	pr_info(DRIVER_NAME ": exit\n");
}

module_init(image_processor_init);
module_exit(image_processor_exit);
```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("CSEE4840");
MODULE_DESCRIPTION("128x128 Sobel display via frame load + FPGA pipeline
(svtest)");
```

image_processor.h

```
None
#ifndef _IMAGE_PROCESSOR_H
#define _IMAGE_PROCESSOR_H

#include <linux/ioctl.h>
#include <linux/types.h>

/* Must match svtest image_processor.sv / img_pkg.sv (128x128). */
#define IMAGE_WIDTH   128
#define IMAGE_HEIGHT  128
#define IMAGE_PIXELS  (IMAGE_WIDTH * IMAGE_HEIGHT)

typedef struct {
      __u16 offset; /* 0 .. IMAGE_PIXELS-1 */
      __u8  value;
      __u8  _pad;
} image_processor_arg_t;

#define IMAGE_PROCESSOR_IOC_MAGIC 'i'
#define IMAGE_PROCESSOR_WRITE_PIXEL _IOW(IMAGE_PROCESSOR_IOC_MAGIC, 0,
image_processor_arg_t)
#define IMAGE_PROCESSOR_READ_PIXEL  _IOWR(IMAGE_PROCESSOR_IOC_MAGIC, 1,
image_processor_arg_t)

#endif
```

Makefile

```
None
KDIR ?= /usr/src/linux-headers-4.19.0
PWD := $(shell pwd)

obj-m := image_processor.o
```

```
all:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
	cc hello.c -o hello

clean:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean
	rm -f hello
```