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.

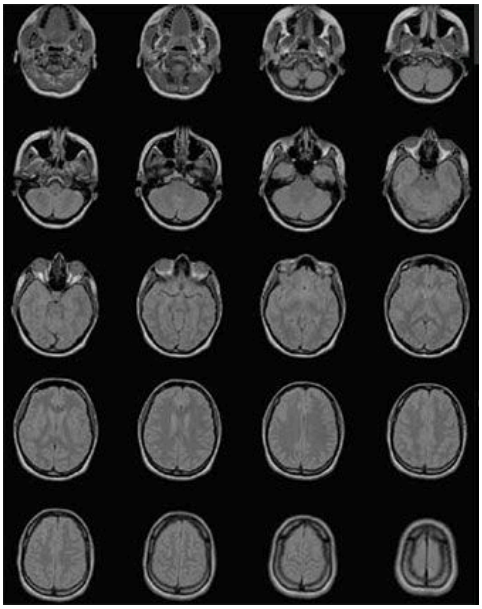

Fig.1: Brain MR images stored in DICOM format

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

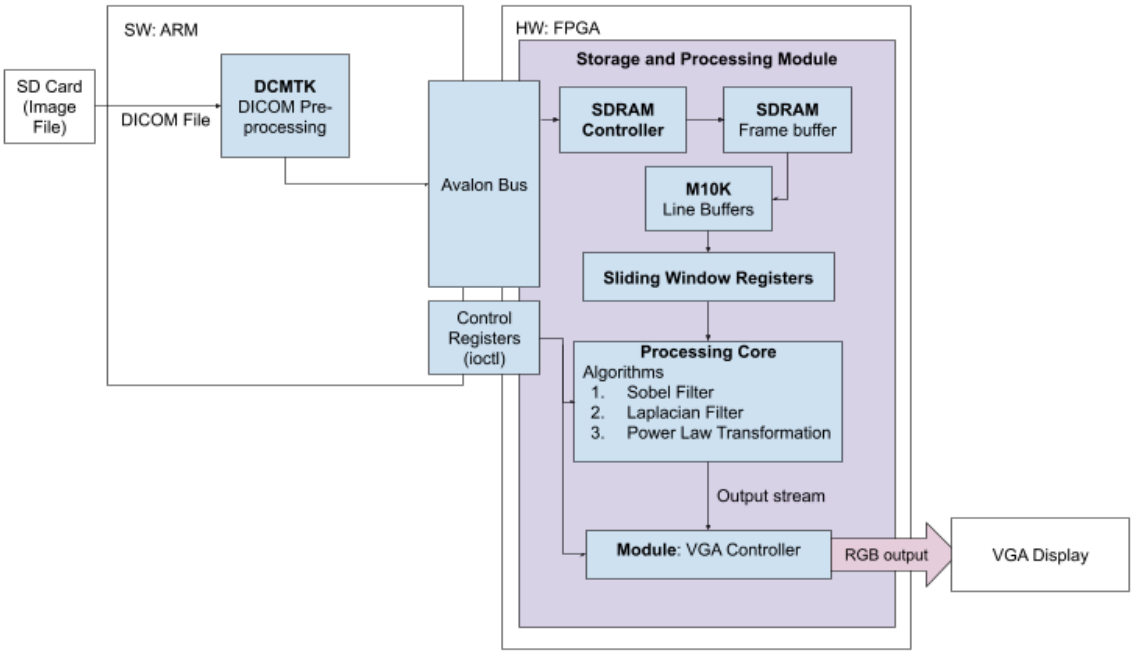

Fig.2: Block diagram Block Diagram

SDRAM Controller

An **SDRAM Controller** will send the image data to the SDRAM. The SDRAM controller, generated using the QSys system integration tool, will interface between the Avalon bus and the SDRAM chip. When the FPGA issues memory read or write requests, the controller translates these requests into the appropriate SDRAM control and data signals.

The diagrams below (from Altera Corporation's site) illustrate the role of the SDRAM controller within the system architecture and the specific signals used to send data to the SDRAM chip.

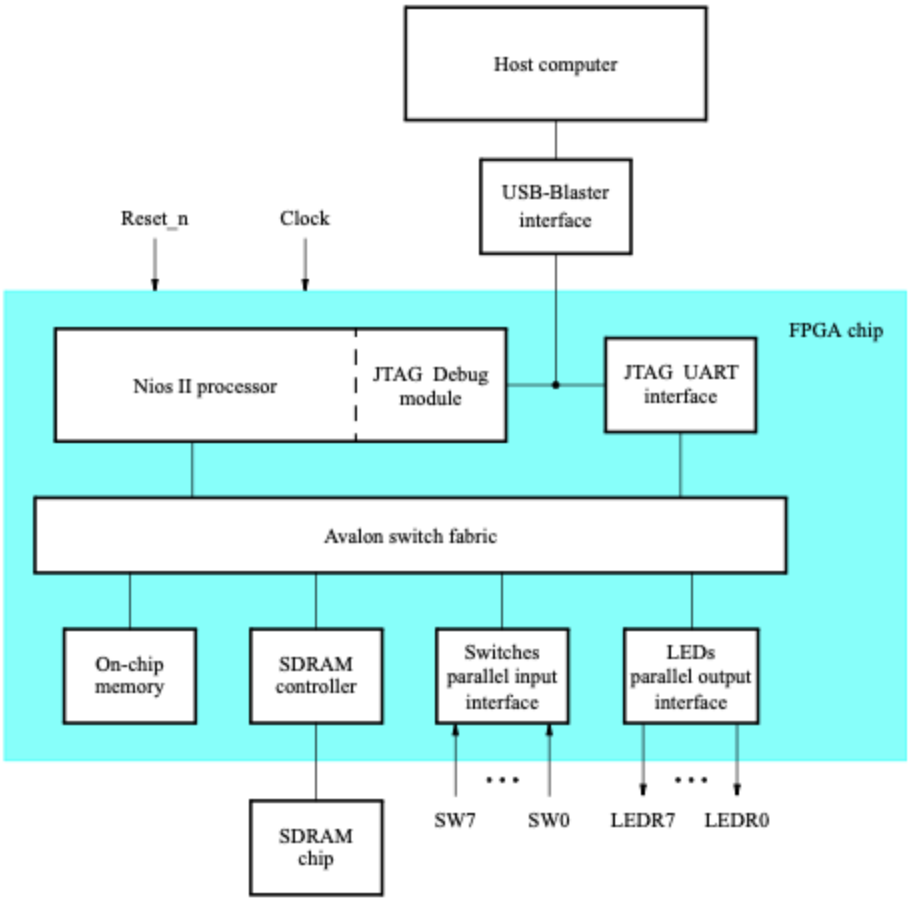

Fig.3: SDRAM controller architecture

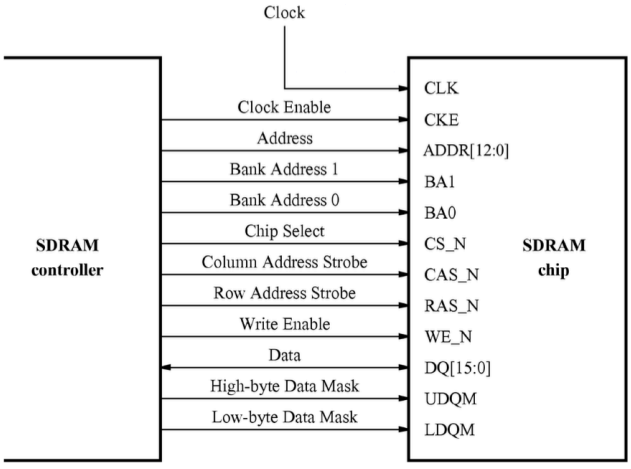

Fig.4: SDRAM signals

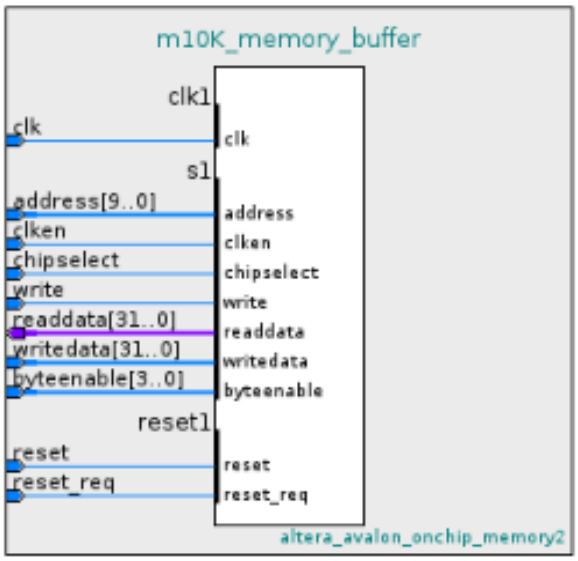

Fig.5: M10k Buffer Inputs (Blue lines) and Outputs (Purple Line)

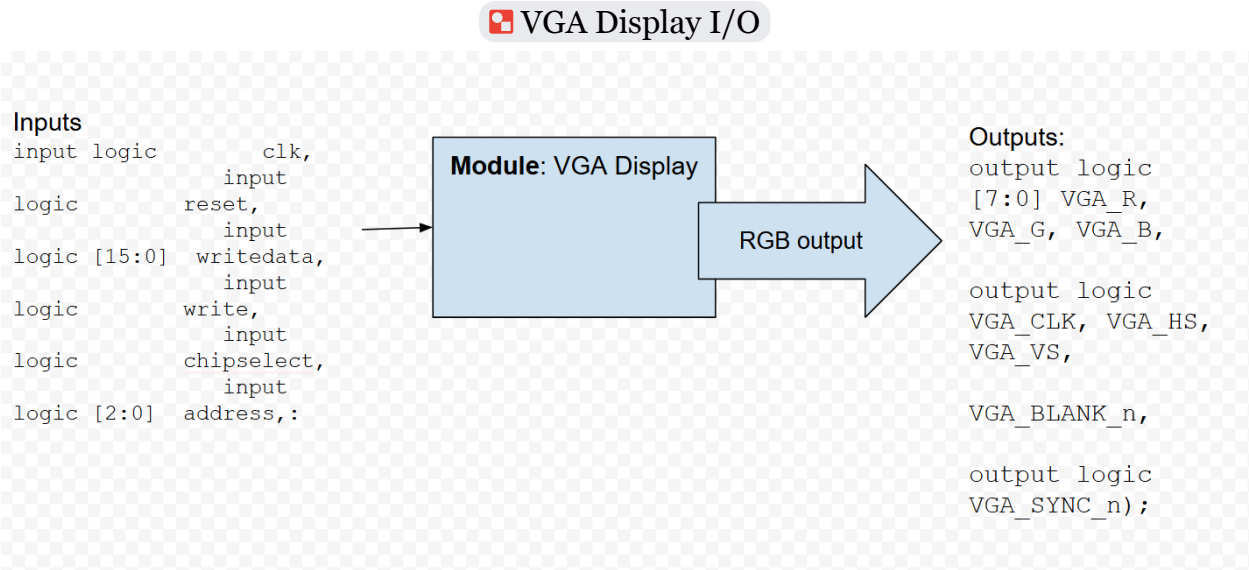

Fig.6: Inputs and outputs of the VGA display

3. Hardware-Software Interface

1. The DICOM image data will be loaded onto the ARM processor via an external SD card.
2. The DICOM files are unpacked using DCMTK (DICOM Toolkit - https://support.dcmtk.org/docs/), which extracts Hounsfield Unit (HU) values.
3. After parsing and compression, the dataset is transferred from the HPS to the FPGA through the Avalon Bus.
4. Control signals (e.g. filter selection, threshold values, zoom parameters) are also communicated through the control interface via ioctls.
5. On the FPGA, image data is written through the SDRAM controller into an SDRAM frame buffer.
6. The FPGA reads image data sequentially from the SDRAM line-by-line and loads 4 rows at a time into the M10K line buffers, where 3 rows are used to construct the active convolution window and 1 additional row is used to preload incoming data.
7. Pixel data from the M10K line buffers is passed into sliding window registers, which assemble the 3x3 window required for convolution.
8. FPGA applies one of the algorithms (Sobel filter, Laplacian filter, Power Law Transformation) on the 3x3 sliding window. The Power Law Transformation uses a Lookup Table (LUT) stored in the M10K buffer. Filter selection can be controlled by FPGA switches.
9. After processing, the processed image is rendered in real time on the VGA display monitor.

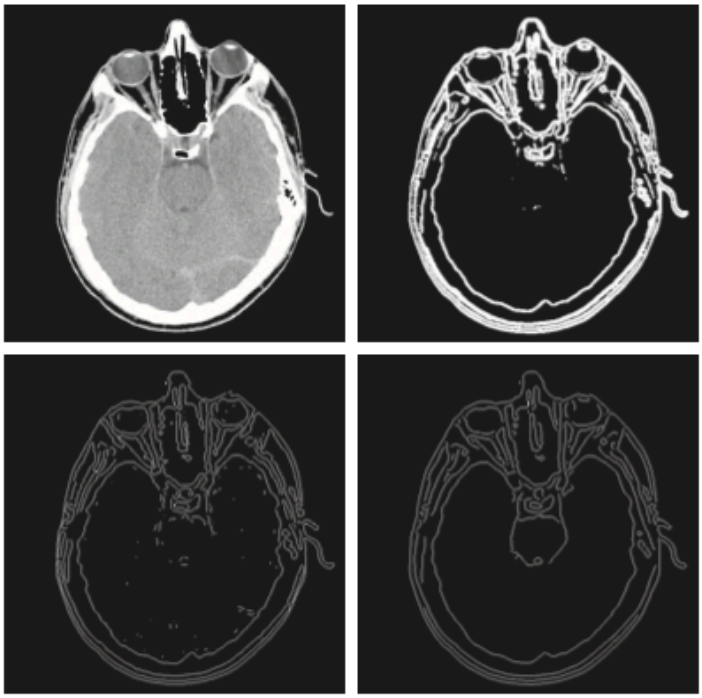

Fig.7: Reference image of brain MRI edge detection

Figure 7 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.

4. Algorithms

We selected three algorithms that serve three purposes in processing medical images. The following steps are used to detect and visualize edges in the image: first, the image intensity array pixels are convolved with the Sobel filter, and then the threshold filter is used to identify the edge.

4.1 Sobel Filter

−1	−2	−1	−1	0	1
0	0	0	−2	0	2
1	2	1	−1	0	1

Sobel

Fig.1: Sobel filter

For each pixel:

$$G_x = I * S_x, G_y = I * S_y$$

We use a 3x3 window from the line buffers in the M10K around that pixel and multiply each value by the corresponding Sobel kernel value. Gx represents how much the intensity changes from left to right (strength of vertical edge) and Gy represents how much the intensity changes from top to bottom (strength of horizontal edge).

$$G = |G_x| + |G_y|$$

We then combine the horizontal and vertical changes into a single edge strength value. Absolute values are used so that changes from bright to dark and dark to bright are treated the same.

Pseudocode

```
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:

         // 3×3 neighborhood (from line buffers / registers)
         Window:  get_3x3_window(image, x, y)

         // horizontal gradient
         Gx = 0
         for i = 0 to 2:
```

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

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

        // gradient magnitude (hardware-friendly approximation)
        G = abs(Gx) + abs(Gy)

        // thresholding
        if G > T:
            output[y][x] = 255
        else:
            output[y][x] = 0

    return output
```

4.2 Power Law

To brighten the dark areas of an image, the power-law (gamma) transformation

$$s = cr^{y}$$

is used to stretch out the dark intensity values.

Pseudocode

```
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):

    (H, W): dimensions(image)
    output: zeros(H, W)
```

```
for y = 0 to H-1:
   for x = 0 to W-1:

      pixel_in  = image[y][x]     // 8-bit
      pixel_out = LUT[pixel_in]   // direct mapping

      output[y][x] = pixel_out

return output
```

4.3 Laplacian Filter

To sharpen the image, the Laplacian filter is convolved with the image intensity array.

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.2: Laplacian filter

The Laplacian filter enhances edges and fine details.

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

We compute the second spatial derivative by combining the center pixel with its 4 immediate neighbors using a discrete convolution kernel.

$$I_{sharp} = I_{original} - L(x, y)$$

We subtract this value from the original image values to obtain the sharpened image.

Pseudocode

```
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):
```

```
(H, W) : dimensions(image)
output : zeros(H, W)

for y = 1 to H-2:
   for x = 1 to W-2:

      // 3×3 window from line buffers + shift registers
      window : get_3x3_window(image, x, y)

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

      output[y][x] = L

return output
```

5. Resource Budgets (Memory constraints)

5.1 ARM DICOM Preprocessing

DICOM data is stored as "Hounsfield Units" which range from -1024 to 3071 HU, which contains 4096 possible values. Since $2^{12} = 4096$, the pixel bits can be truncated from 16 bits to 12 bits by removing the top 4 most significant bits. The resulting image storage size would be approximately 393 KB.

$$(512\ pixel\ rows \times 512\ pixels\ columns \times 8\ \ bits\ /\ pixel) \div\ 8\ bits/byte\ =\ 262,144\ bytes\ /image$$

5.2 SDRAM – DDR3

The SDRAM DDR3 storage has 1 GB of DRAM available to store the 263 KB image.

5.3 M10K buffer

Once the ARM processor completes preprocessing and transfers the image over the Avalon Bus, the complete DICOM image is stored in SDRAM. M10K blocks are used to implement the line buffers that support 3x3 convolution operations. The buffers will store a small subset of the image at any time:

4 rows x 512 pixels x 8 bits = 16,384 bits = 2048 bytes (2KB)

The buffer is then read sequentially, supplying pixel data to the register module so that the Algorithm module can process each pixel in turn. Critically, the buffer is enabled to update, meaning that as the algorithm finishes with a pixel, the buffer advances to prepare the data window for the next pixel, keeping the pipeline moving without stalling.

5.4 Neighborhood Pixel Buffer Register Map

The design currently targets 12-bit depth to preserve the full Hounsfield dynamic range, but the header file defines HOUNSFIELD_12BIT_MAX and M10K_BUF_SIZE_BYTES so that this value is changed in one place if the bit depth is revised.

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

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

This 72-bit window is what the Sobel and Laplacian filters operate on directly, since both are convolution-based algorithms that require surrounding pixel context rather than just the current pixel value.

5.4.1 Header File Interface

The header file would define all constants and types that govern how the ARM processor interacts with the M10K buffer. The relevant definitions are:

```C/C++
#define IMG_WIDTH 512
#define IMG_HEIGHT 512

#define PIXEL_DEPTH 8
#define IMG_SIZE (IMG_WIDTH * IMG_HEIGHT)

#define SDRAM_BASE_OFFSET 0x00000000
#define FRAME_BUFFER_SIZE_BYTES (IMG_SIZE * (PIXEL_DEPTH/8))
```

SDRAM_BASE_OFFSET defines the starting address of the image frame in the SDRAM as seen through the Avalon memory-mapped interface. IMG_SIZE represents the total number of pixels in the frame buffer. FRAME_BUFFER_SIZE_BYTES represents the total memory footprint of the image in SDRAM in bytes.

5.4.2 Register Map Interface

Once the M10K buffer supplies a pixel to the pipeline, the register module captures all data needed for the algorithm to compute that pixel's output. The register map exposed over the Avalon Bus is:

Register	Offset	Description
REG_IMG_MIN	0x08	Minimum Hounsfield value in the image
REG_IMG_MAX	0x0C	Maximum Hounsfield value in the image
REG_PIXEL_VAL	0x18	8-bit value of current input pixel
REG_ALGO_SELECT	0x28	Algorithm to apply (Sobel / Laplacian / Power Law)
REG_THRESHOLD	0x2C	Edge detection threshold (0–255)
REG_GAMMA_Q8	0x30	Gamma value for Power Law, Q8 fixed-point

The ARM processor writes REG_IMG_MIN, REG_IMG_MAX, REG_ALGO_SELECT, REG_THRESHOLD, and REG_GAMMA_Q8 once at startup via the FpgaConfig struct defined in the header. Pixel data streams from the M10K line buffers at 1 pixel per clock cycle, while REG_PIXEL_VAL serves as a monitoring register and the convolution is performed directly on the 3x3 sliding window, avoiding any readback or stalling.

5.5 Algorithms (Memory)

5.5.1 Sobel Filter

Input: Normalized image from SDRAM (256 KB)

Access pattern: Sequential burst reads
(3x3 window is obtained from M10K buffers (4 rows: 4 x 512 x 8 = 2KB) - 1 row for prefetching)
Absolute sum of Sobel kernel weights =

$$|-1| + |0| + |1| + |-2| + |0| + |2| + |-1| + |0| + |1| = 8$$

Max output = (Max pixel value) x (sum of absolute kernel weights)
Since only half the weights contribute positively/negatively,
Max output bits = $255 \times 4 = 1020$
Min output bits = $255 \times (-4) = -1020$

$$G_x , G_y \in [-1020, +1020]$$

After the 8-bit pixels undergo weighted convolution, G_x and G_y would require 16-bit signed representation due to bit growth. These gradients are combined into a magnitude value (up to 2040). The thresholding step then compresses the result back to an 8-bit binary image, before sending the final output stream to the VGA controller for real-time display.

5.5.2 Laplacian Filter

Input: Normalized image from SDRAM (256 KB)
Access pattern: Sequential burst reads
(3x3 window is obtained from M10K buffers (4 rows: 4 x 512 x 8 = 2KB) - 1 row for prefetching)

Max positive case:
Center pixel = 0
Neighbors = 255
$L = 4(255) - 0 = 1020$

Max negative case:
Center pixel = 255
Neighbors = 0
$L = 4(0) - 255 = -1020$

$$L(x, y) \in [-1020, +1020]$$

Unlike Sobel, the Laplacian output is not directly used. It is first combined with the original 8-bit pixel value to perform image sharpening. This means that the hardware must hold:
Original pixel: 8-bit register
Laplacian result: 16-bit register

$$I_{sharp} = I_{original} - L(x, y)$$

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

reg [7:0] output_pixel

Negative values are clipped to black, large positive values are clipped to white, and mid-range values are passed through. This maps the range from [-1020, +1020] to [0, 255], producing 8-bit output pixels that can be displayed through the VGA.

5.5.3 Power-Law (Gamma) Transformation

The pixel data is streamed directly from the SDRAM since the output pixels depend solely on the corresponding input value and not on the neighboring pixels. The transformation is implemented using a lookup table (LUT), where each 8-bit pixel serves as an index into a precomputed table of 256 entries, stored in the M10K buffer. We would store the output as uint8 (since the result is clamped back to 0–255), so the LUT is 256 bytes. The mapping is performed in a single clock cycle with minimal arithmetic overhead and fixed 8-bit output width. This allows the pixel values to remain as 8-bit throughout the transformation.