EE671 Project 2

Team 39

Jay Mehta (22B1281)

Vatsal Melwani (22B0396)

Jainam Ravani (22B1242)

Amol Pagare (22B3971)

November 2024

Contents

Chapter 1

The Project: Laplacian Filter for Edge Detection

1.1 Introduction

The Laplacian filter is a second derivative operator used in image processing to compute the Laplacian of the image. The Laplacian of an image highlights regions of rapid intensity change and is therefore often used for edge detection. The Laplacian filter is a simple and effective way to detect edges in an image. In this project, we will implement the Laplacian filter in VLSI using Verilog and test it on a set of 16x16 pixel images.

The Laplacian filter is defined as:

$$\nabla^2 f(x,y) = \frac{\partial^2 f(x,y)}{\partial x^2} + \frac{\partial^2 f(x,y)}{\partial y^2} \tag{1.1}$$

The Laplacian filter can be implemented using a 3x3 convolution kernel:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 \\ 0 & -2 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{1.2}$$

The first term in the sum corresponds to $\frac{\partial^2 f(x,y)}{\partial x^2}$ and the second term corresponds to $\frac{\partial^2 f(x,y)}{\partial y^2}$.

The Laplacian filter is a high-pass filter that enhances high-frequency components in the image. It is particularly effective at detecting edges in an image, as edges correspond to regions of rapid intensity change. The Laplacian filter is also sensitive to noise, so it is often used in combination with other filters to improve edge detection performance. But in this project, we will only implement the Laplacian filter as floating point arithmetic is difficult to implement in hardware.

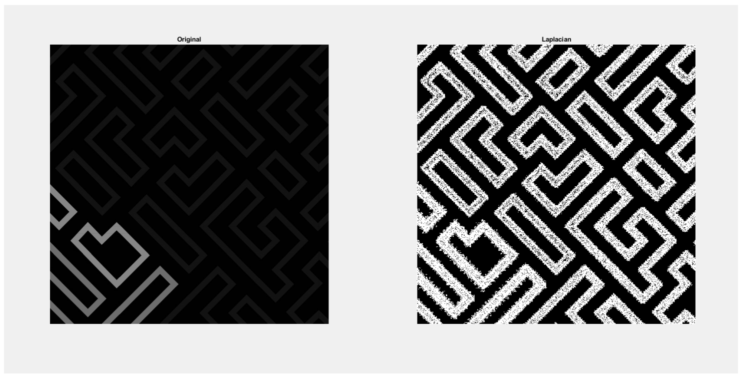

Figure 1.1: Sample Image

The results of the Laplacian Filter on an image can be seen with the given set of images. This image has dimensions of 1280x1280. For hardware implementation, this is infeasible. Hence, we have done hardware implementation for 16x16 images only.

1.2 Implementation

The Laplacian filter is implemented as a 3x3 convolution kernel. The input image is convolved with the kernel to compute the Laplacian of the image. The Laplacian filter is implemented in Verilog using a 16x16 pixel image as input. The Verilog code reads the pixels of the input image one at a time, convolves them with the Laplacian kernel, and writes the result to the output image. The output image pixel is computed as the sum of the products of the input image pixels and the kernel coefficients. The output image pixel is also output one at a time.

The input pixels are read from a text file in the testbench module. The input image stored as flattened version of the 16x16 image with each pixel value on a new line. The input file has 16 trailing zeros, this is to perform zero padding on the input image. The input image is read one pixel at a time and passed to the Laplacian filter module. The output image is written to a text file in the testbench module. The output image is also stored as a flattened version of the 16x16 image with each pixel value on a new line.

The output file also has valid_out bit with each pixel value to indicate if the pixel is valid or not. Typically, there will be 16 invalid pixels at the start of an image. This is because we need to store the first row completely before we can start processing. Due to this reason, the first 16 pixels are invalid, or rather we have 16 cycles of latency.

In the testbench, the output file generated has the form on each line:

$$< \text{pixel value} > \quad < \text{valid\_out} >$$

The output file is then read by a Python script to generate the output image. The output image is displayed using the matplotlib library. The output image is displayed as a 16x16 pixel image with the pixel values represented as grayscale intensities. The edges in the output image correspond to regions of rapid intensity change in the input image.

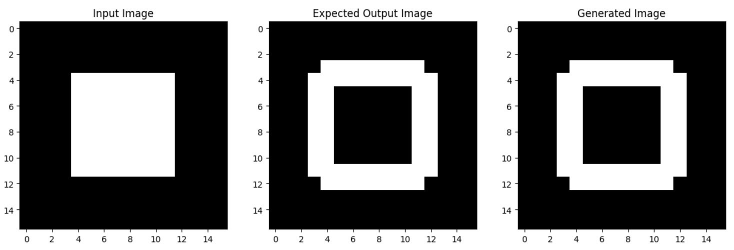

Figure 1.2: Input, Benchmark and Output Images

The Benchmark Image is determined by running the same input image through a Laplacian filter in MATLAB. The output image is then compared with the benchmark image to verify the correctness of the implementation.

1.3 Details of the Design

The design of the edge detector filter was broken down into various blocks and each was coded as a module in verilog. At the end, in the top module, all the blocks were utilized to implement the whole algorithm of the filter. Following is the description of each module created, starting from elementary ones to the top level:-

1.3.1 PG_gen

This block takes input the propagate and generate of [i:j] and [j-1:k] and uses it to calculate the propagate and generate signals for [i : k] using the following logic:

$$P[i:k] = P[i:j] \cdot P[j-1:k] \tag{1.3}$$

$$G[i:k] = G[i:j] + P[i:j].G[j-1:k] \tag{1.4}$$

1.3.2 Kogge Stone Adder (11 bits)

A kogge stone adder was coded in verilog for 11 bits input. The output of this block was only the sum generated. The carry was not required in the whole of the filter, hence it was not taken as an output from the adder. It consisted to first calculating the propagate and generate signals, i : 0, for all i using the kogge stone tree structure and then calculate the carry and sum bits for all stages using the propagate and generate signals with the following logic:

$$S[i] = P[i] \oplus C[i-1] \tag{1.5}$$

$$C[i] = G[i:0] + P[i:0] \cdot Cin \tag{1.6}$$

1.3.3 mux_21_8 and mux_21_11

The above are 2 input muxes, one with 8 bits input, other with 11 bits input. According to the select bit, it generates the output.

1.3.4 Mat_operation

This module consists of the main convolution operation to be calculated. It takes 5 - 8 bit inputs, one of which is the main pixel on which the convolution is being applied and the rest 4 pixels are it's neighbouring pixels of positions where the matrix entry was non-zero. Following is it's flow graph:

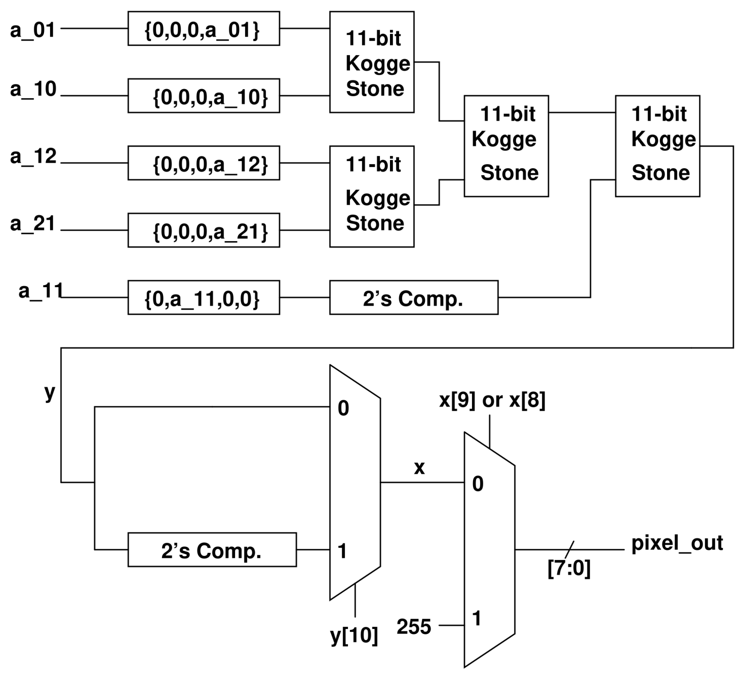

Figure 1.3: Matrix Operation

As can be seen, a_{11} is the main pixel and a_{01}, a_{10}, a_{12} and a_{21} are neighbouring pixels. Firstly, each pixel is converted to 11 bits from 8 bits, This is due to the range of the output and the 2's complement notation which will be used ahead for calculating the difference. Hence, a_{01}, a_{10}, a_{12} and a_{21} were directly appended with 3 zeros in most significant positions, while a_{11} was appended with one zero in the most significant position and 2 zeros in the least significant position. This is because the matrix operation requires a_{11} to be multiplied by 4, hence the two left shifts. After this, 2's complement of a_{11} is calculated as the final operation to be done now is:

$$a_{01} + a_{10} + a_{12} + a_{21} - 4 * a_{11} \tag{1.7}$$

Parallely, all 4 a_i's were added using three kogge stone adders. Then, the 2s complement of a_{11} was also added in the partial sum to get the final matrix operation sum. After this, there are 2 rules to be followed for the filter:

1. If the output is negative, take its modulus
2. If the output is greater than 255, clip it to the max value, i.e., 255.

To implement these rules, first the sign of the output had to be checked. Hence, the first mux acts as an absolute operator. If the MSB of the output is 1, it is negative, else the output is positive. If the MSB is 1, the mux gives the output as 2s complement of the input, which is essentially making the output positive. After this mux, the 11th bit is useless as the notation is back to unsigned. Hence, to check for the value of the output, 8th and the 9th bits are used. If any one of them is high, it indicates that the absolute value is greater than 255 and so the output of the mux is 255. Else, the output remains the same as the input, but only 8 bits will be given as the output. Hence, the second mux acts as a clipping operator. This output is now the final pixel output of the laplacian filter for the input pixel a_{11}, and is given as the output of the module.

1.3.5 Top Level Module: laplacian

This module is responsible for taking the input pixels and storing them efficiently so as to use minimum storage elements and also continuously start generating the output pixels. This was done by maintaining a storage of 3 18-byte registers, i.e., each row has 18 units and each unit is of 8 bits, capable of storing a pixel value in grayscale. Following are the steps that take place in this module:

- All the rows are initialized to 0. The image is 16 pixels * 16 pixels, the row size is kept 18 because the first and last pixels in the row are permanently kept as 0 to induce zero padding in the image to get the convolution output.
- The input is given pixel by pixel. Hence, the new pixel always gets stored in the first row. So, the first pixel goes to first_row[1], 2nd pixel to first_row[2], . . . , till 16th pixel to first_row[16].
- Now, as soon as the first row is filled, all the data from the first row gets transferred to the second row and the new pixel input again starts to get stored in first_row[1].

- After the first row has again filled, the second row data gets fully transferred to the third row and the first row data fully transferred to the second row and now the first is again ready to store the new pixel inputs.
- There are only 3 row storages because in a convolution operation, when calculating for an input pixel, only the pixels immediately right and left to it and immediate top and bottom to it are useful. So, only three rows of an image are necessary to get stored.
- The matrix_operation is only applied in the second row, which is also obvious as it's the middle storage row.
- Since the second row is empty for the first 16 cycles, the pixel out starts to get useful from the 17th cycle. So the design has a **latency of 16 cycles**.
- Also, there is a corner case. For the last row calculation, zero row is required below it. Hence, the user must ensure to give an extra row of 0s so that the last row also gets a zero padding.

There are two extra important signals in this module, valid_in and valid_out. Valid_in denotes that the input to the design is a valid pixel of the image. valid_out denotes the output is a valid pixel of the output image. Hence, for the first 16 outputs, the valid_out signal will be low indicating the latency. Also, from the 17th cycle onwards, the valid_out starts to be high and again goes to zero as soon as 256 valid input pixels have been filtered.

There are two more signals to the design, clock and reset signal. Reset signal resets all the storage rows to 0 and hence a new test image can start to get processed. Clock signal is derived from the testbench.

The input file corresponding to image in Fig. 1.2, 'image.txt' is given in the 'Extra_Files' folder. The output of the MATLAB script for the same is given in 'output.txt'. The output file, 'output_file.txt' generated by the testbench is also given in the same folder. The Jupyter notebook 'reconstruct.ipynb', present in the same folder, takes all the three files and generates the same figure as in Fig. 1.2.

Chapter 2

Layout Details

2.1 GDS Layout

The GDS layout of the design is as follows:

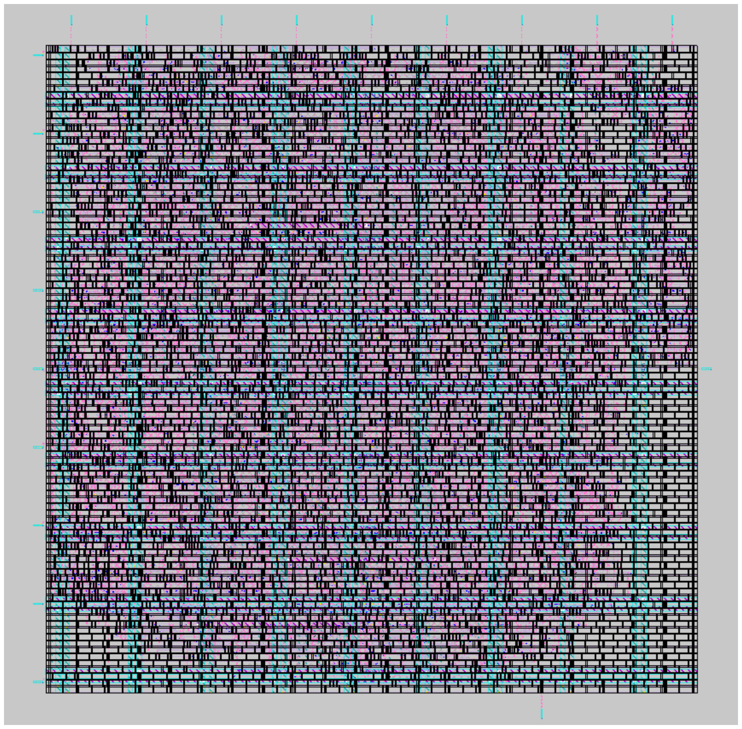

Figure 2.1: GDS Layout

2.2 Summary Table of Wires and Cells

Parameter	**Count**
Number of wires	3146
Number of wire bits	3160
Number of public wires	539
Number of public wire bits	553
Number of memories	0
Number of memory bits	0
Number of processes	0
Number of cells	3149

Taken from 1-synthesis.log file.

2.3 Cell Count Details

Cell Name	**Count**
sky130_fd_sc_hd__a2111o_2	30
sky130_fd_sc_hd__a211o_2	19
sky130_fd_sc_hd__a211oi_2	1
sky130_fd_sc_hd__a21bo_2	1
sky130_fd_sc_hd__a21boi_2	1
sky130_fd_sc_hd__a21o_2	14
sky130_fd_sc_hd__a21oi_2	24
sky130_fd_sc_hd__a221o_2	139
sky130_fd_sc_hd__a22o_2	157
sky130_fd_sc_hd__a22oi_2	1
sky130_fd_sc_hd__a2bb2o_2	1
sky130_fd_sc_hd__a311o_2	1
sky130_fd_sc_hd__a311oi_2	1
sky130_fd_sc_hd__a31o_2	19
sky130_fd_sc_hd__a32o_2	4
sky130_fd_sc_hd__and2_2	53
sky130_fd_sc_hd__and2b_2	33
sky130_fd_sc_hd__and3_2	70
sky130_fd_sc_hd__and3b_2	6
sky130_fd_sc_hd__and4_2	12
sky130_fd_sc_hd__buf_1	646
sky130_fd_sc_hd__dfrtp_2	530
sky130_fd_sc_hd__dfstp_2	3
sky130_fd_sc_hd__dfxtp_2	1

sky130_fd_sc_hd__inv_2	541
sky130_fd_sc_hd__mux2_2	434
sky130_fd_sc_hd__nand2_2	30
sky130_fd_sc_hd__nand2b_2	2
sky130_fd_sc_hd__nand3_2	3
sky130_fd_sc_hd__nor2_2	103
sky130_fd_sc_hd__nor3_2	14
sky130_fd_sc_hd__nor3b_2	3
sky130_fd_sc_hd__nor4_2	5
sky130_fd_sc_hd__o2111a_2	1
sky130_fd_sc_hd__o211a_2	1
sky130_fd_sc_hd__o211ai_2	1
sky130_fd_sc_hd__o21a_2	7
sky130_fd_sc_hd__o21ai_2	7
sky130_fd_sc_hd__o21ba_2	1
sky130_fd_sc_hd__o21bai_2	1
sky130_fd_sc_hd__o221a_2	2
sky130_fd_sc_hd__o22a_2	13
sky130_fd_sc_hd__o22ai_2	8
sky130_fd_sc_hd__o31a_2	2
sky130_fd_sc_hd__o32a_2	8
sky130_fd_sc_hd__o41a_2	7
sky130_fd_sc_hd__or2_2	19
sky130_fd_sc_hd__or2b_2	8
sky130_fd_sc_hd__or3_2	20
sky130_fd_sc_hd__or3b_2	1
sky130_fd_sc_hd__or4_2	42
sky130_fd_sc_hd__or4b_2	3
sky130_fd_sc_hd__xnor2_2	37
sky130_fd_sc_hd__xor2_2	58

Taken from 1-synthesis.log file.

Chapter 3

Waveforms

3.1 Pre-Synthesis Waveform

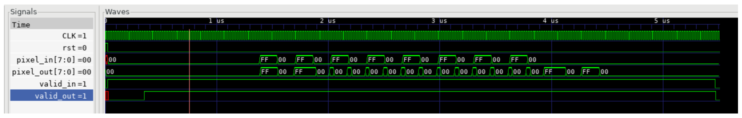

Figure 3.1: Pre-Synthesis Waveform

3.2 Post-Synthesis Waveform

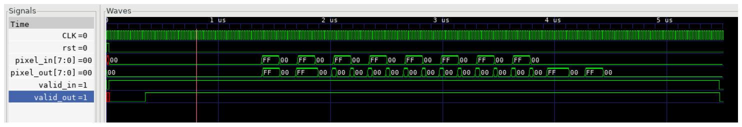

Figure 3.2: Post-Synthesis Waveform

3.3 Post-Routing Waveform

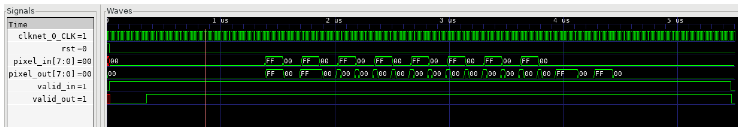

Figure 3.3: Post-Routing Waveform

Chapter 4

Tables for Power, Performance and Area

4.1 Power

Group	**Internal**	**Switching**	**Leakage**	**Total**	**Percent**
Sequential	8.44*e*-4	6.11*e*-5	6.93*e*-9	9.05*e*-4	34.2%
Combinational	6.85*e*-4	5.01*e*-4	1.06*e*-8	1.19*e*-3	44.9%
Clock	3.34*e*-4	2.18*e*-4	1.52*e*-9	5.52*e*-4	20.9%
Macro	0.00	0.00	0.00	0.00	0.00
Pad	0.00	0.00	0.00	0.00	0.00
Total	1.86*e*-3	7.8*e*-4	1.91*e*-8	2.64*e*-3	100%
Percent	70.5%	29.5%	0%	-	-

Table 4.1: Power Table - from 20-grt_sta.log Typical Corner (routing)

Group	**Internal**	**Switching**	**Leakage**	**Total**	**Percent**
Sequential	8.44*e*-4	1.51*e*-4	5.28*e*-7	9.95*e*-4	29.6%
Combinational	6.94*e*-4	9.95*e*-4	1.06*e*-8	1.69*e*-3	50.2%
Clock	3.36*e*-4	3.44*e*-4	6.4*e*-9	6.8*e*-4	20.2%
Macro	0.00	0.00	0.00	0.00	0.00
Pad	0.00	0.00	0.00	0.00	0.00
Total	1.87*e*-3	1.49*e*-3	5.45*e*-7	3.36*e*-3	100%
Percent	55.7%	44.3%	0%	-	-

Table 4.2: Power Table - from 29-rcx_mcsta.nom.log Typical Corner (signoff)

4.2 Area

Area of the design - 35253 μm^2

4.3 Performance

Clock Frequency - 33.33 MHz
Time Period - 30 ns

Chapter 5

Uses of the Laplacian Filter and Future Prospects

The Laplacian filter is a powerful tool for edge detection in image processing. It is widely used in computer vision applications to detect edges in images. The Laplacian filter can also be used in feature extraction applications, such as object recognition and image segmentation. The Laplacian filter can be implemented in hardware using FPGA or ASIC technology to accelerate edge detection in real-time applications. The Laplacian filter can also be implemented in software using OpenCV and MATLAB to perform edge detection on images and videos. The Laplacian filter is a versatile tool for edge detection and feature extraction in image processing applications.

The Laplacian Filter is not used directly generally owing to noise in the images and is replaced by Laplacian of Gaussian (LoG) which performs a smoothening operation before doing the edge detection resulting in better edge detection. This can be a future prospect for the project. This wasn't implemented in the project due to the complexity of the filter and the hardware constraints and also floating point arithmetic which is difficult to implement in hardware.

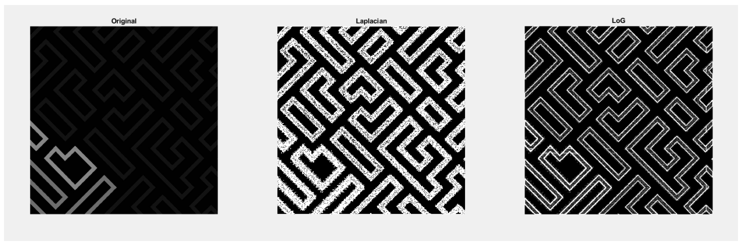

Figure 5.1: Laplacian of Gaussian

Chapter 6

Work Distribution

- Vatsal Melwani - Datapath for the filter on paper, Verilog codes for Kogge Stone Adder, Laplacian module, 2's complement block and muxes, Debugging of the output, Design Details for modules in report
- Jay Mehta - Verilog Code for Laplacian and Mat_operation module, MATLAB and Python scripts for test image generation and reconstruction, Debugging of the output, Power, Performance and Area tables, Uses of the Laplacian Filter and Future Prospects
- Jainam Ravani - Verilog Code for PG_gen, muxes, Debugging of the output, GDS Layout
- Amol Pagare - Initial Design of Laplacian, Testing of the design on Quartus (locally) and Server, Waveforms, Report Compilation, Debugging of the output