Image processing Using Verilog

Yellow and Blue Colour detection

By: Amrish S

Table of Content:

Introduction:

The field of image processing has an recent advancements and have opened up a wide range of applications, from medical imaging to autonomous vehicles. One such application is color detection, which has applications in robotics, traffic signal recognition, and color-based sorting systems.

This report describes the design and implementation of a yellow and blue light detector using Verilog, a hardware description language commonly used for designing electronic systems. Verilog allows for high-level design of complex algorithms and systems.

The report focuses on developing an algorithm to detect yellow and blue lights in images. This algorithm uses image enhancement techniques, color space transformations, and filtering methods. The challenges of implementing these techniques in hardware using Verilog are also discussed.

The report provides a comprehensive overview of the design process, implementation details, and performance analysis of a yellow and blue light detector using image processing in Verilog.

1.1 Research Motivation

Image processing is a rapidly growing field with a wide range of applications in various fields, including medical imaging, security, and robotics. Verilog, a hardware description language (HDL), is commonly used for designing digital systems. Combining image processing techniques with Verilog can lead to the development of efficient and high-performance hardware solutions for image processing tasks.

Object or color detection in images is a crucial application of image processing. It requires the ability to identify and extract relevant information from image data. For instance, in medical imaging, image processing can be employed to detect tumors or other abnormalities in X-ray or MRI scans. In the field of security, image processing can be used to identify potential threats in surveillance footage.

Color thresholding is one of the most common color segmentation techniques. It involves selecting a threshold value for each color channel (red, green, and blue) and then classifying each pixel in the image as belonging to a particular color based on its intensity values. However, this technique is sensitive to noise and illumination variations, potentially leading to inaccurate results.

Edge detection is another color segmentation technique. It involves identifying the edges of objects in the image and then using the edge information to segment the image into different regions. However, this technique can be computationally expensive and may not be suitable for real-time applications.

1.2 Proposed Research

The proposed research aims to develop a novel hardware architecture for image processing in Verilog that can efficiently and accurately detect yellow and blue light in an image. The proposed architecture will utilize a combination of color thresholding and edge detection techniques. The

architecture will be designed to be scalable and efficient, making it suitable for real-time applications.

1.3 Expected Outcomes

A novel hardware architecture for image processing in Verilog that can efficiently and accurately detect yellow and blue light in an image.

A prototype implementation of the proposed architecture on a Field Programmable Gate Array (FPGA) or Application-Specific Integrated Circuit (ASIC).

An evaluation of the performance of the proposed architecture on real-world images.

Main objective is to detect the Yellow and Blue light in the Image.

1.4 Significance of Research

The proposed hardware architecture will provide a new and efficient method for detecting yellow and blue light in an image.

The proposed architecture will be scalable and efficient, making it suitable for real-time applications.

The results of this research will have applications in a variety of fields, including medical imaging, security, and robotics.

1.5 Scope of Project

Objective:

To create a novel hardware architecture for image processing in Verilog, focusing on efficiency and leveraging the Verilog programming language.

Scalability and Real-Time Performance:

The proposed architecture should be scalable to accommodate images of varying sizes and complexities, maintaining real-time performance.

Prototyping and Evaluation:

A prototype of the proposed architecture will be implemented on either an FPGA (Field Programmable Gate Array) or ASIC (Application-Specific Integrated Circuit) to assess its feasibility and performance. Additionally, the performance of the proposed architecture will be evaluated using a benchmark set of real-world images to gauge its accuracy and efficiency.

Comparison and Applications:

The proposed architecture's performance will be compared to existing methods for detecting yellow and blue light in images, demonstrating its superiority. Furthermore, the potential applications of the proposed architecture will be explored in various fields, including medical imaging, security, and robotics.

Literature Review:

Introduction:

Image processing is a rapidly growing field with a wide range of applications, including medical imaging, security, and robotics. Verilog is a hardware description language (HDL) that is commonly used for designing digital systems. By combining image processing techniques with Verilog, it is possible to develop efficient and high-performance hardware solutions for image processing tasks.

Existing Hardware Architectures for Image Processing in Verilog

Numerous researchers have explored the use of Verilog to implement hardware architectures for image processing tasks. Some of the notable examples include:

- **FPGA-based Image Processing Architectures:**

FPGAs (Field Programmable Gate Arrays) offer a balance between flexibility and performance, making them suitable for real-time image processing applications. Several studies have proposed FPGA-based architectures for image processing tasks such as edge detection, filtering, and feature extraction.

- **ASIC-based Image Processing Architectures:**

ASICs (Application-Specific Integrated Circuits) provide superior performance and energy efficiency compared to FPGAs. However, they require a more complex design process. Several studies have investigated ASIC-based architectures for image processing, particularly for low-power applications.

- **Hybrid FPGA-ASIC Architectures:**

Combining FPGAs and ASICs can leverage the strengths of both platforms to achieve a balance between performance, flexibility, and cost. Hybrid architectures have been proposed for image processing tasks such as video encoding and decoding.

Techniques for Detecting Yellow and Blue Light

Detecting yellow and blue light in images is a fundamental task in various applications, including color image analysis, object detection, and medical imaging. Several techniques have been employed for this purpose, including:

- **Color Thresholding:**

This method involves setting threshold values for each color channel (red, green, and blue) and classifying pixels based on their intensity values. While simple and efficient, color thresholding can be sensitive to noise and illumination variations.

- **Edge Detection:**

This technique identifies the edges of objects in the image, which can aid in distinguishing between different colors. Popular edge detection algorithms include Canny edge detection and Sobel edge detection.

- **Color Space Conversion:**

Converting the color space of the image, such as from RGB to HSV (Hue, Saturation, Value), can enhance the separation of yellow and blue hues, making detection more accurate.

Comparison of Existing Methods

Existing methods for detecting yellow and blue light in images offer varying trade-offs in terms of accuracy, efficiency, and complexity. Color thresholding is simple and fast but can be sensitive to noise. Edge detection provides more detailed information about object boundaries but can be computationally expensive. Color space conversion can improve accuracy but introduces additional processing steps.

Future Directions

The field of image processing in Verilog continues to evolve, with researchers exploring new techniques and architectures to achieve higher performance, efficiency, and flexibility. Potential future directions include:

- **Investigation of Deep Learning-based Approaches:**

Deep learning algorithms have demonstrated remarkable performance in image processing tasks, and their integration with Verilog-based architectures could lead to significant advancements.

- **Development of Adaptive Architectures:**

Adaptive architectures that can adjust their configuration based on the specific image processing task could improve resource utilization and efficiency.

- **Exploration of Emerging Hardware Platforms:**

Emerging hardware platforms such as neuromorphic computing and quantum computing could offer new possibilities for image processing in Verilog.

Why do we Process an Image?

Image processing is a vital element in numerous technological applications. Here are some reasons why we process images:

Quality Enhancement:

One of the main purposes of image processing is to improve the image's quality. This could include enhancing the image's contrast, brightness, or sharpness. It could also involve eliminating noise or other distortions from the image.

Information Extraction:

Image processing can be utilized to extract specific information from an image. This could involve identifying specific objects in the image, recognizing patterns, or quantifying certain attributes of the image.

Image Recognition:

Image processing is essential in image recognition. This involves identifying specific objects or features in an image. Image recognition applications include facial recognition, object detection, and optical character recognition.

Preparation for Further Processing:

In some instances, an image may need to be processed before it can be used for further processing. For example, an image might need to be resized, cropped, or transformed in some way before it can be used in a machine learning algorithm.

Improving Interpretability:

Image processing can also make images more interpretable for humans. For example, medical images often need to be processed so that doctors can better interpret them.

Impact factor of doing image processing in Verilog:

Image processing in Verilog has a significant impact on various fields, including digital signal processing, computer vision, and embedded systems.

Advantages of doing image processing using Verilog:

- **Efficiency and Speed:**

Verilog, when combined with hardware like Field Programmable Gate Arrays (FPGAs), can process images at high speeds, making it ideal for real-time image processing applications where fast response times are critical.

- **Flexibility and Reusability:**

Verilog is a hardware description language that enables the design of complex digital systems in a manageable way. It provides designers with the flexibility to simulate and evaluate digital circuit performance, taking timing considerations into account. Additionally, developed modules can be reused in other projects, promoting code reusability.

- **Real-time Processing:**

Verilog can simulate an image sensor or camera, allowing for real-time image processing on FPGAs. This is essential for applications like autonomous driving, where real-time processing of visual data is paramount.

Image Enhancement Techniques:

Verilog supports a variety of image enhancement techniques, including inversion, brightness control, and thresholding operations. More sophisticated techniques like clamp filters and sharpening spatial filters can also be employed to improve the quality of scaled images.

Color Detection:

Verilog can be used to develop algorithms for detecting specific colors in images, which has applications in areas like robotics and traffic signal recognition.

Disadvantages of doing image processing in Verilog:

- **Not optimized for image processing:**

Verilog is primarily designed for digital circuit design and simulation, and it lacks built-in support for common image processing data structures and operations.

This necessitates the creation of custom modules to handle complex mathematical operations and image formats, increasing development time and complexity.

- **Slow and inefficient:**

The performance of image processing in Verilog is constrained by the resources and speed of the FPGA device. Finite memory, logic gates, and clock cycles limit the quality and efficiency of image processing tasks. Additionally, data conversion and communication between the FPGA and host computer add latency and overhead. Debugging and testing image processing algorithms in Verilog is also challenging due to the need for specialized tools and hardware.

- **Not portable or scalable:**

Image processing in Verilog is tied to the specific FPGA device and its configuration. Each FPGA architecture has unique features, constraints, and limitations that impact the implementation and optimization of image processing algorithms. Verilog code optimized for one FPGA device may not function on another or may require significant modifications. Moreover, adapting image processing algorithms in Verilog to different image sizes, formats, or algorithms is cumbersome, requiring code changes and FPGA recompilation.

Methodology:

The process for creating a system that detects yellow and blue colors using image processing techniques in Verilog can be summarized in the following steps:

Image Collection:

The initial step in any image processing task is to collect the image. In this case, we need to gather an image that includes the colors we aim to detect - yellow and blue. This image could be sourced from various places, such as a digital camera or an existing image file.

Pre-processing:

After the image has been collected, it might need to undergo pre-processing before the color detection can occur. Pre-processing might include resizing the image, modifying its contrast or brightness, or converting it to a different color space to simplify the color detection.

Color Identification:

The subsequent step is to actually identify the colors in the image. This involves scanning the image pixel by pixel and verifying if the color of each pixel corresponds to the colors we are searching for. In this case, we would verify if each pixel is either yellow or blue.

Post-processing:

Once the color identification has been completed, some post-processing might be necessary. For instance, we might want to emphasize the areas of the image where the colors were identified, or we might want to tally the number of pixels that were identified as each color.

Display or Preservation:

Lastly, the results of the color identification process need to be displayed or preserved. This could involve displaying the image on a screen, with the identified colors emphasized, or it could involve preserving the results in a file or database for future analysis.

Quartus Prime:

The Software that I have used for this project is Quartus prime lite edition. The Quartus Prime Lite Edition by Intel is a complimentary version of the Quartus Prime design software, specifically designed for cost-effective FPGAs. It is utilized for the design and modeling of digital systems, which includes FPGA, CPLD, and SoC designs.

This software offers a swift route to transform your ideas into reality and is compatible with numerous third-party tools for synthesis, static timing analysis, board-level simulation, signal integrity analysis, and formal verification

Neural Network:

Neural networks are a branch of machine learning and the foundation of deep learning algorithms. Inspired by the structure and function of the human brain, they mimic the way biological neurons communicate with each other.

Structure of Neural Networks:

A neural network receives input data, processes it through hidden layers, and generates an output prediction. Each hidden layer contains neurons with adjustable weights that are modified during training.

Input Layer:

The input layer receives data from the dataset. It's sometimes called the visible layer because it directly interacts with the data.

Hidden Layer:

Hidden layers, located between the input and output layers, can be stacked in multiple levels.

Output Layer:

The final layer, the output layer, produces the prediction based on the input data.

Training Neural Networks:

Training a neural network involves adjusting the weights of neurons based on the error in the output prediction. This process is called backpropagation.

Backpropagation:

Backpropagation calculates the error as the difference between the expected and predicted output. This error is propagated backward through the network, starting from the output layer and moving towards the input layer, adjusting weights along the way.

Gradient Descent:

An optimization algorithm, gradient descent minimizes a function by iteratively moving in the direction of steepest descent, determined by the negative of the gradient.

Applications of Neural Networks:

Neural networks have found widespread applications in image and speech recognition, self-driving cars, financial forecasting, and weather prediction.

Convolutional Neural Networks (CNNs):

It is the Powerhouses of Image Processing

CNNs, a specialized type of neural network, are particularly adept at handling structured grid data, such as images. Their ability to autonomously extract features from input images has revolutionized the field of image processing.

Image Classification:

CNNs can classify images by assigning them labels or categories. For instance, a CNN trained on animal images can determine whether a new image depicts a cat, dog, bird, or other animal.

Object Detection:

CNNs can identify and pinpoint objects within an image. For example, a CNN could detect and localize cars and pedestrians in an image captured by a self-driving car's camera.

Image Segmentation:

CNNs can divide images into distinct regions corresponding to objects or their components. For instance, a CNN could segment a medical image to identify different tissues or organs.

Image Generation:

CNNs can create new images based on specific criteria or characteristics. Generative Adversarial Networks (GANs), a type of CNN, can generate novel images that resemble a set of training images.

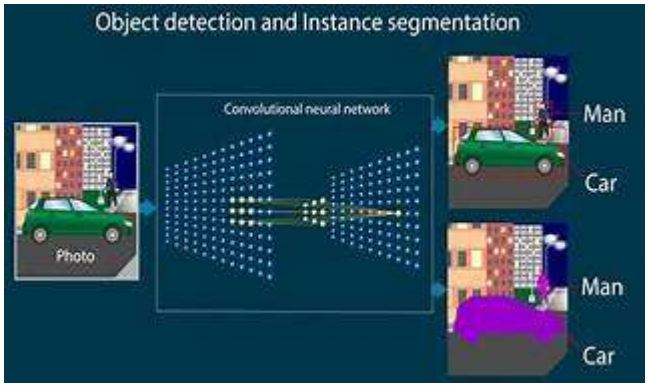
There are 3 Building Blocks of CNNs:

1) Convolutional
2) Pooling
3) Fully Connected Layers

Convolutional Layers:

These layers are the heart of CNNs and are responsible for feature extraction. They convolve the input image with learnable filters, each designed to detect a specific type of feature.

Convolutional layers serve as the fundamental components in convolutional neural networks (CNNs). They function by applying a filter to an input, which leads to an activation. When the same filter is repeatedly applied to an input, it results in an activation map, also known as a feature map. This map indicates the locations and intensity of a detected feature in an input, such as an image.



Operational Mechanism:

Convolution is a linear operation that involves multiplying a set of weights with the input. As this technique was designed for two-dimensional input, the multiplication is carried out between an array of input data and a two-dimensional array of weights, referred to as a filter or a kernel.

The novelty of convolutional neural networks lies in their ability to automatically learn a significant number of filters in parallel, tailored to a training dataset under the constraints of a specific predictive modeling problem, such as image classification. The outcome is highly specific features that can be detected anywhere on input images.

Pooling Layers:

The Cornerstones of Convolutional Neural Networks

Pooling layers are an essential component of convolutional neural networks (CNNs), widely used in image processing tasks. Their primary purpose is to progressively reduce the spatial dimensions of the feature maps, thereby minimizing the number of parameters and computational requirements of the network.

Functional Overview of Pooling Layer:

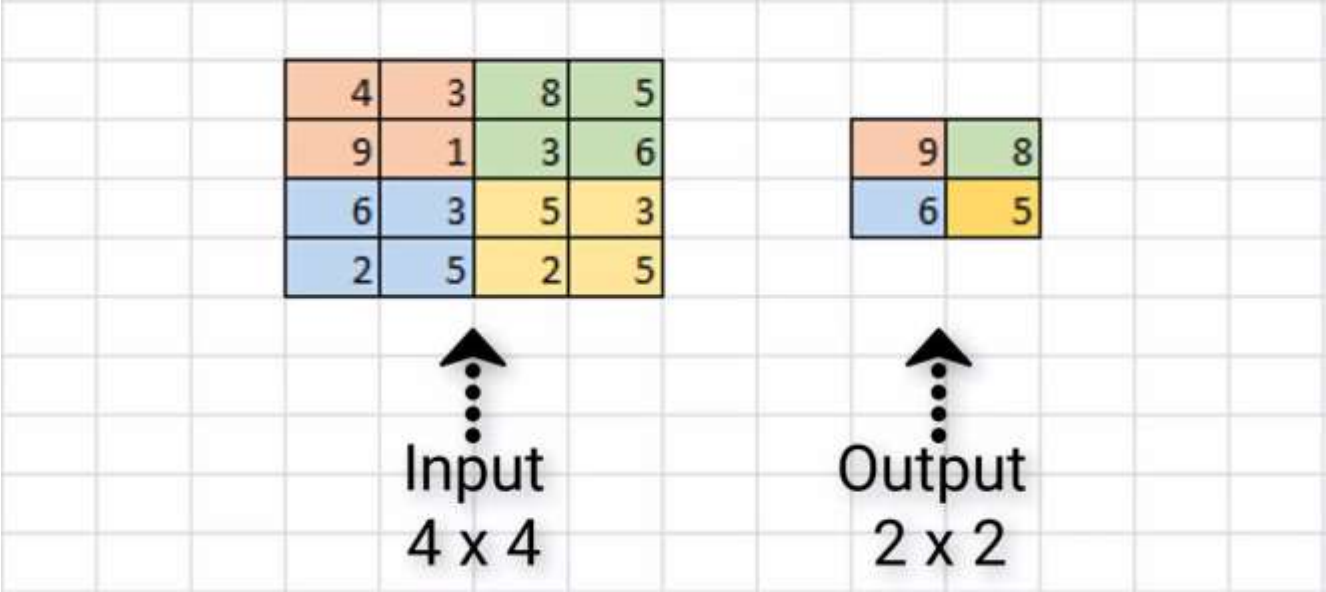
Pooling layers serve the crucial function of aggregating features extracted from feature maps generated by applying a filter over an image. They achieve this by applying a pooling operation, such as max or average pooling, to each region of the feature map. This operation involves sliding a two-dimensional filter across each channel of the feature map and summarizing the features within the filter's coverage area.

Types of Pooling Operations:

1) Max Pooling
2) Average Pooling
3) Global Pooling

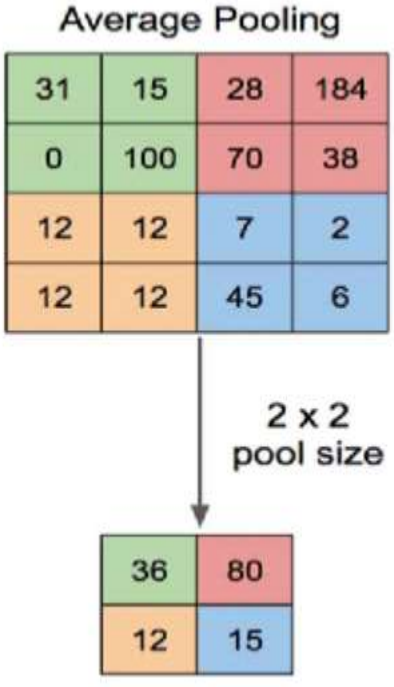
1. **Max Pooling:**

This operation selects the maximum element from the region of the feature map covered by the filter. Consequently, the output of a max-pooling layer is a feature map containing the most prominent features of the preceding feature map.



2. Average Pooling:

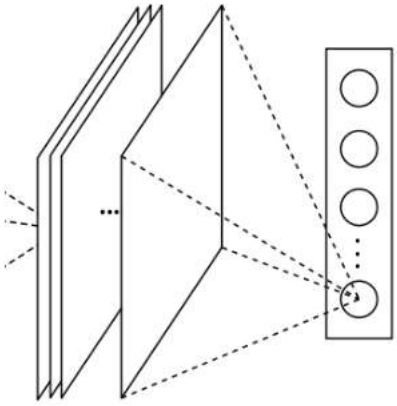
This operation computes the average of the elements present within the region of the feature map covered by the filter. While max pooling identifies the most prominent feature in a particular patch of the feature map, average pooling provides the average of features present in that patch.



3. **Global Pooling:**

Global Average Pooling is an operation that aims to take the place of fully connected layers in traditional CNNs. The concept involves creating one feature map for each category in the classification task in the final mlpconv layer. Rather than incorporating fully connected layers above the feature maps, we calculate the average of each feature map, and the resulting vector is directly input into the softmax layer.

One of the benefits of global average pooling over fully connected layers is that it maintains the convolution structure by enforcing correspondences between feature maps and categories. As a result, the feature maps can be easily seen as category confidence maps. Another advantage is that there are no parameters to optimize in the global average pooling, thus avoiding overfitting at this layer. Moreover, global average pooling eliminates spatial information, making it more robust to spatial translations of the input.



Advantages of Pooling Layer:

- **Dimensionality Reduction:**

Pooling layers reduce the dimensions of feature maps, thereby minimizing the number of parameters to be learned and the amount of computation performed in the network.

- **Feature Summarization:**

Pooling layers summarize the features present in a region of the feature map generated by a convolutional layer.

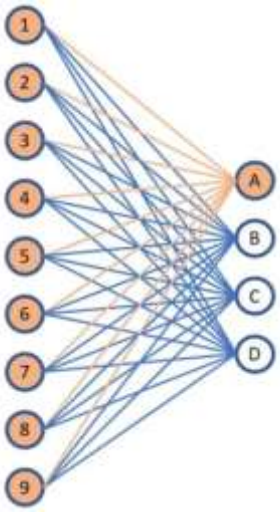
- **Translational Invariance:**

Pooling layers make the model more robust to variations in the position of features within the input image.

Pooling layers are an integral component of CNNs, particularly in image processing applications. They contribute to reducing computational complexity, providing translational invariance, and enhancing the robustness of feature detection within feature maps. The incorporation of pooling layers in CNNs has significantly improved their performance and efficiency, making them a cornerstone of modern image recognition and analysis systems.

Fully Connected Layers:

It is the Backbone of Image Processing.



In the realm of neural networks, a fully connected layer, also known as a dense or inner product layer, stands out as a distinct type of layer where every neuron in one layer is intricately connected to every neuron in the subsequent layer. This intricate network of interconnected neurons facilitates complex information processing and feature extraction, making it an indispensable tool in image processing tasks.

Within the domain of image processing, fully connected layers often play a crucial role in the final stages of a convolutional neural network (CNN), empowering the network to perform tasks such as image classification, object detection, and image segmentation. Their ability to effectively combine information extracted from all feature maps generated by convolutional layers makes them particularly well-suited for tasks that demand global reasoning and decision-making.

Roles of fully connected layers in image processing:

Image Classification:

Fully connected layers excel at classifying images into distinct categories, such as animals, plants, or human faces. This remarkable ability stems from their capability to learn patterns of features, extracted from convolutional layers, and associate them with specific categories.

Object Detection:

With their prowess in identifying regions of an image containing objects, fully connected layers play a pivotal role in object detection. They achieve this by classifying these regions into specific object categories.

Image Segmentation:

Fully connected layers are instrumental in image segmentation, where the goal is to divide an image into distinct regions. They accomplish this by learning to associate patterns of features, extracted from convolutional layers, with specific regions of the image.

Algorithm for Neural Network:

Initializing the Network:

The CNN algorithm begins by initializing the network parameters, which define the architecture and behavior of the network. These parameters include the number and size of filters, stride and padding values, pooling size and type, and activation functions. These parameters determine how the network processes and analyzes the input image.

Processing Input Images:

For each input image, the CNN algorithm follows a series of steps:

- **Convolutional Operations:**

Filters, the heart of CNNs, are applied to the input image, generating a set of feature maps. Each filter corresponds to a feature map, and each element in the feature map represents the outcome of a convolution operation between the filter and a specific region of the input image.

- **Non-Linear Activation:**

To introduce non-linearity and enable learning of complex patterns, a non-linear activation function, such as ReLU, is applied to each element in the feature map.

Pooling:

To reduce the spatial dimensions of the feature map, making the network more robust to variations and reducing computational complexity, a pooling operation, such as max or average pooling, is applied to each region of the feature map.

Feature Extraction and Prediction:

The processed feature maps are then flattened into a one-dimensional vector, capturing the learned features from the input image. This vector serves as the input for fully connected layers.

Fully connected layers, unlike convolutional layers, connect every neuron in one layer to every neuron in the next layer. They process the flattened feature vector and make predictions based on the learned features and weights. The output layer, with a size equal to the number of classes, utilizes a softmax activation function to assign probabilities to each class label. The class with the highest probability is predicted as the image's label.

Error Calculation and Parameter Update:

The algorithm calculates the error or loss between the predicted output and the actual output using a loss function, such as cross-entropy. This loss function measures the network's performance on the given input image and the corresponding label.

To minimize the loss function and improve performance, the network parameters, such as filter weights and biases, are updated using an optimization algorithm, such as gradient descent. This algorithm adjusts the network parameters in the direction of the negative gradient of the loss function with respect to the parameters.

Training and Convergence:

The algorithm iterates through the entire training dataset, processing each input image and updating the network parameters accordingly. This process continues until the network converges to a satisfactory level of accuracy or a maximum number of iterations is reached.

Formulas:
Convolution layer (CONV): This layer performs convolution operations on the input image using filters of size FtimesFtimesC, where C is the number of channels in the input image. The output feature map has a size of OtimesOtimesK, where K is the number of filters applied. The formula for computing the output size O is:

O=fracI−F+2PS+1
where I is the input size, P is the padding size, and S is the stride size1.

Pooling layer (POOL): This layer reduces the spatial dimensions of the input feature map by applying a pooling operation (such as max or average) over a window of size FtimesF. The output feature map has a

size of OtimesOtimesC, where C is the number of channels in the input feature map. The formula for computing the output size O is the same as the convolution layer, except that there is no padding (P=0)1.

Fully connected layer (FC): This layer connects every neuron in the input feature map to every neuron in the output layer. The output layer has a size of N, where N is the number of classes to predict. The formula for computing the number of weights in this layer is:

W=ItimesOtimesCtimesN
where I, O, and C are the input size, output size, and number of channels of the input feature map, respectively

Cyclone IV E:

The Cyclone IV E family of FPGAs from Intel excels in low-power consumption, making them ideal for battery-powered or portable devices. They are crafted with an optimized 60-nm low-power process, boasting up to 115K logic elements (LEs), 4 Mbits of embedded memory, and 266 embedded 18 x 18 multipliers. Additionally, Cyclone IV E FPGAs can be equipped with an integrated transceiver, capable of delivering data rates of up to 3.125 Gbps.

These FPGAs stand out due to their key features:

Reduced Power Consumption:

Cyclone IV E FPGAs rank among the most power-efficient FPGAs available, making them an excellent choice for extending battery life and minimizing heat dissipation.

Enhanced Performance:

These FPGAs deliver exceptional performance and logic density, making them well-suited for a diverse range of applications.

Cost-Effectiveness:

Cyclone IV E FPGAs are among the most affordable FPGAs, making them a practical choice for price-sensitive applications.

The versatility of Cyclone IV E FPGAs extends to a wide array of applications, including:

- **Consumer Electronics:**

  Cyclone IV E FPGAs power a variety of consumer electronics devices, including televisions, set-top boxes, and mobile phones.

- **Medical Devices:**

  These FPGAs play a crucial role in medical devices such as hearing aids, pacemakers, and defibrillators.

- **Industrial Automation:**

  Cyclone IV E FPGAs are employed in various industrial automation equipment, including programmable logic controllers (PLCs) and motor controllers.

The utilization of Cyclone IV E FPGAs offers compelling benefits:

- **Energy Efficiency:**

Cyclone IV E FPGAs can significantly reduce power consumption compared to other FPGAs, extending battery life and minimizing heat generation.

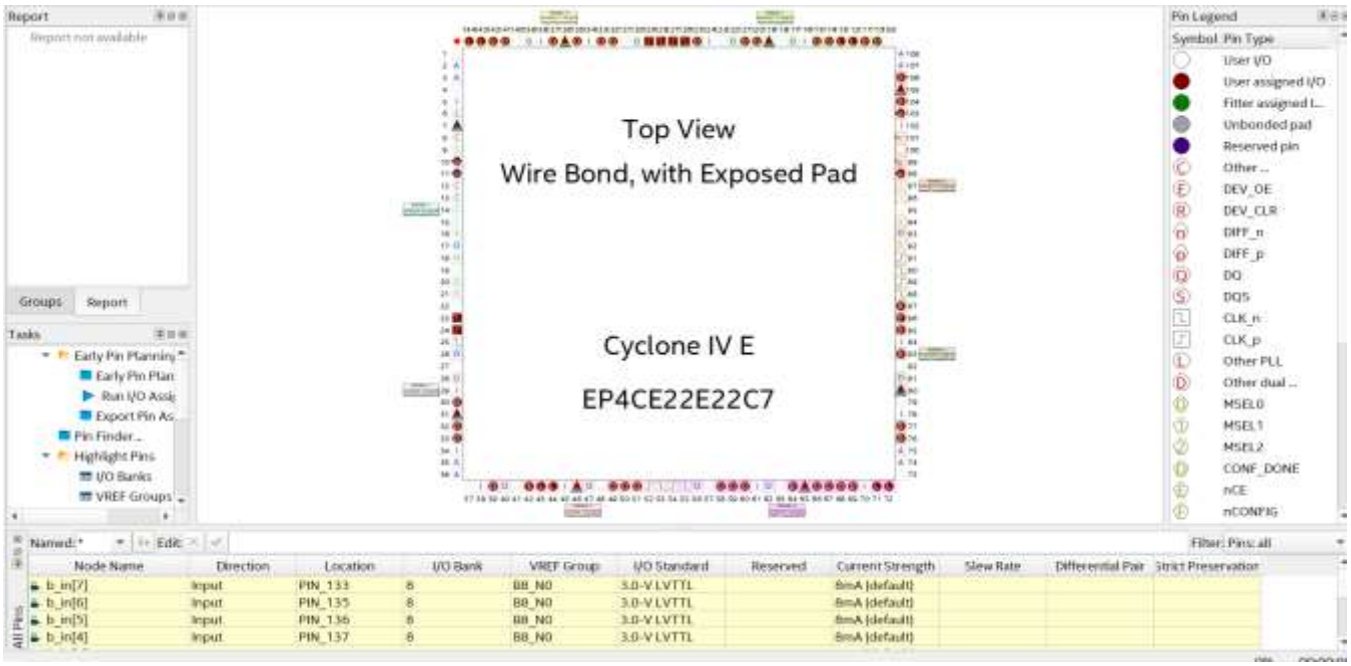
- **Boosted Performance:**

These FPGAs deliver superior performance compared to other FPGAs, enhancing the speed and responsiveness of applications.

- **Cost Optimization:**

Cyclone IV E FPGAs are among the most cost-effective FPGAs available, reducing the overall cost of a product.

Pin Planner Cyclone IV E:



Color detection Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.CONFIG.ALL;


entity nn_rgb is
  port (clk      : in  std_logic;                    -- input clock 74.25 MHz, video 720p
        reset_n  : in  std_logic;                    -- reset (invoked during configuration)
        enable_in : in  std_logic_vector(2 downto 0);   -- three slide switches
        -- video in
        vs_in    : in  std_logic;                    -- vertical sync
        hs_in    : in  std_logic;                    -- horizontal sync
        de_in    : in  std_logic;                    -- data enable is '1' for valid pixel
```

```
        r_in      : in  std_logic_vector(7 downto 0);   -- red component of pixel
        g_in      : in  std_logic_vector(7 downto 0);   -- green component of pixel
        b_in      : in  std_logic_vector(7 downto 0);   -- blue component of pixel
        -- video out
        vs_out    : out std_logic;                      -- corresponding to video-in
        hs_out    : out std_logic;
        de_out    : out std_logic;
        r_out     : out std_logic_vector(7 downto 0);
        g_out     : out std_logic_vector(7 downto 0);
        b_out     : out std_logic_vector(7 downto 0);
        --
        clk_o     : out std_logic;                      -- output clock (do not modify)
        led       : out std_logic_vector(2 downto 0));  -- not supported by remote lab
end nn_rgb;

architecture behave of nn_rgb is

    -- input FFs
    signal reset                : std_logic;
    signal enable               : std_logic_vector(2 downto 0);
    signal vs_0, hs_0, de_0     : std_logic;


    -- output of signal processing
    signal vs_1, hs_1, de_1         : std_logic;
    signal result_r, result_g, result_b : std_logic_vector(7 downto 0);

    type   y_array is array (0 to 10) of std_logic_vector(7 downto 0);
    signal y : y_array;


begin


--generate the neural network with the parameters from config.vhd
--the outer loops creates the layers and the inner loop the neurons within the layer
--input Layer is assgined later
gen : FOR i IN 1 TO networkStructure'length - 1 GENERATE --layers
    gen2: FOR j IN 0 TO networkStructure(i) - 1 GENERATE --neurons within the Layers
     begin
```

```
        knot: entity work.neuron
            generic map ( weightsIn => weights(positions(j+1,i)-1 downto positions(j,i)))
            port map (  clk      => clk,
                        inputsIn => (connection(connnectionRange(i)-1 downto connnectionRange(i-1))),
                        output   => connection(connnectionRange(i)+j));
    END GENERATE;
END GENERATE;

--delay the control signals for the time of the processing
control: entity work.control
    generic map (delay => 9)
    port map (  clk      => clk,
                reset    => reset,
                vs_in    => vs_0,
                hs_in    => hs_0,
                de_in    => de_0,
                vs_out   => vs_1,
                hs_out   => hs_1,
                de_out   => de_1);

process
begin
   wait until rising_edge(clk);

   -- input FFs for control
   reset <= not reset_n;
   enable <= enable_in;
    -- input FFs for video signal
   vs_0  <= vs_in;
   hs_0  <= hs_in;
   de_0  <= de_in;

   --assign values of the input layer
   connection(0) <= to_integer(unsigned(r_in));
   connection(1) <= to_integer(unsigned(g_in));
   connection(2) <= to_integer(unsigned(b_in));

   -- convert RGB to luminance: Y (5*R + 9*G + 2*B)
   y(0) <= std_logic_vector(to_unsigned(
         (5*connection(0) + 9*connection(1) + 2*connection(2))/16,8));
```

```
   for i in 1 to 10 loop
      y(i) <= y(i-1);
   end loop;


end process;

process
variable luminance : std_logic_vector(7 downto 0);
variable r_yellow, r_blue, r_gray : std_logic_vector(7 downto 0);
variable g_yellow, g_blue, g_gray : std_logic_vector(7 downto 0);
variable b_yellow, b_blue, b_gray : std_logic_vector(7 downto 0);

begin

  wait until rising_edge(clk);
-- output processing
-- assign the pixel a value depending on the output of the neural network

luminance := y(8);

-- yellow: amplify red and green
r_yellow := '1' & luminance(7 downto 1);
g_yellow := '1' & luminance(7 downto 1);
b_yellow := '0' & luminance(7 downto 1);

-- blue: amplify blue
r_blue   := '0' & luminance(7 downto 1);
g_blue   := '0' & luminance(7 downto 1);
b_blue   := '1' & luminance(7 downto 1);

-- gray: use luminance
r_gray   := luminance;
g_gray   := luminance;
b_gray   := luminance;

     if(connection(11) > 127) then

          if(connection(11) > connection(10)) then
             -- yellow
```

```
            result_r <= r_yellow;
            result_g <= g_yellow;
            result_b <= b_yellow;
        else
            -- blue
            result_r <= r_blue;
            result_g <= g_blue;
            result_b <= b_blue;
        end if;
    elsif (connection(10)>127) then
        -- blue
        result_r <= r_blue;
        result_g <= g_blue;
        result_b <= b_blue;

    else
        -- gray
        result_r <= r_gray;
        result_g <= g_gray;
        result_b <= b_gray;
    end if;

  -- output FFs
  vs_out  <= vs_1;
  hs_out  <= hs_1;
  de_out  <= de_1;
  r_out   <= result_r;
  g_out   <= result_g;
  b_out   <= result_b;

end process;

clk_o <= clk;
led   <= "000";

end behave;
```

Explanation for the Code:

Entity Declaration

The entity declaration defines the ports of the circuit. The ports are:

clk: The input clock signal.
reset_n: An active-low reset signal.
enable_in: A three-bit input vector that controls the processing of the video signal.
vs_in: The vertical sync signal of the input video.
hs_in: The horizontal sync signal of the input video.
de_in: The data enable signal of the input video.
r_in: The red component of the input pixel.
g_in: The green component of the input pixel.
b_in: The blue component of the input pixel.
vs_out: The vertical sync signal of the output video.
hs_out: The horizontal sync signal of the output video.
de_out: The data enable signal of the output video.
r_out: The red component of the output pixel.
g_out: The green component of the output pixel.
b_out: The blue component of the output pixel.
clk_o: The output clock signal.
led: A three-bit output vector that is not used in this code.
Architecture

The architecture defines the behavior of the circuit. The architecture consists of three main parts:

Signal Processing:

This part of the code generates the output control signals (vs_1, hs_1, and de_1) from the input control signals (vs_in, hs_in, and de_in).

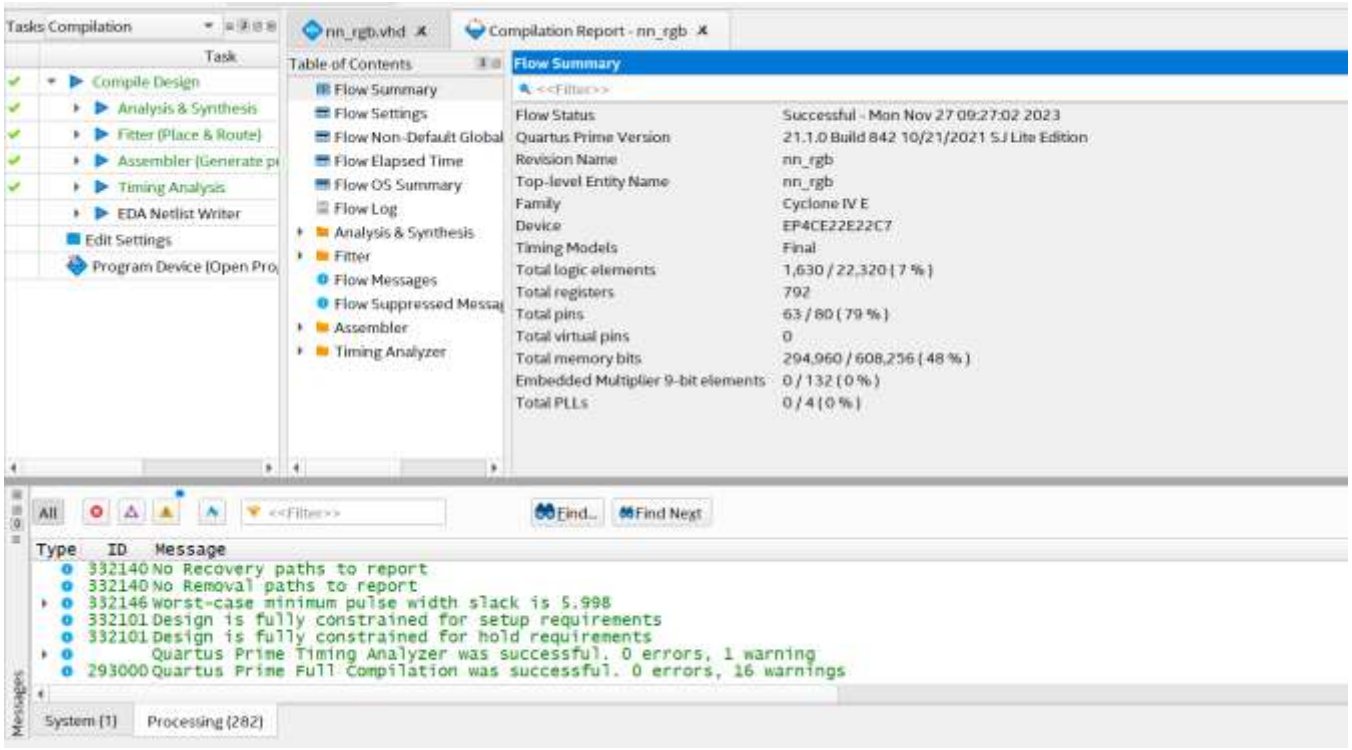
Neural Network:

This part of the code generates the output pixel values (result_r, result_g, and result_b) from the input pixel values (connection(0), connection(1), and connection(2)) using a neural network. The neural network is implemented using a series of interconnected neurons. Each neuron has a set of weights and an output function. The output of a neuron is computed by applying the output function to the weighted sum of its inputs.

Output Processing:

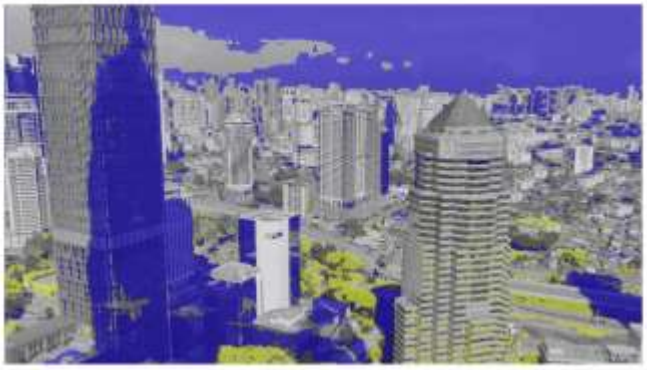
This part of the code assigns the output pixel values (result_r, result_g, and result_b) to the output ports (r_out, g_out, and b_out) based on the classification of the input pixel. The classification is determined by the value of connection(11). If connection(11) is greater than 127, then the pixel is classified as yellow. If connection(11) is greater than 127, then the pixel is classified as blue. Otherwise, the pixel is classified as gray.

Output:



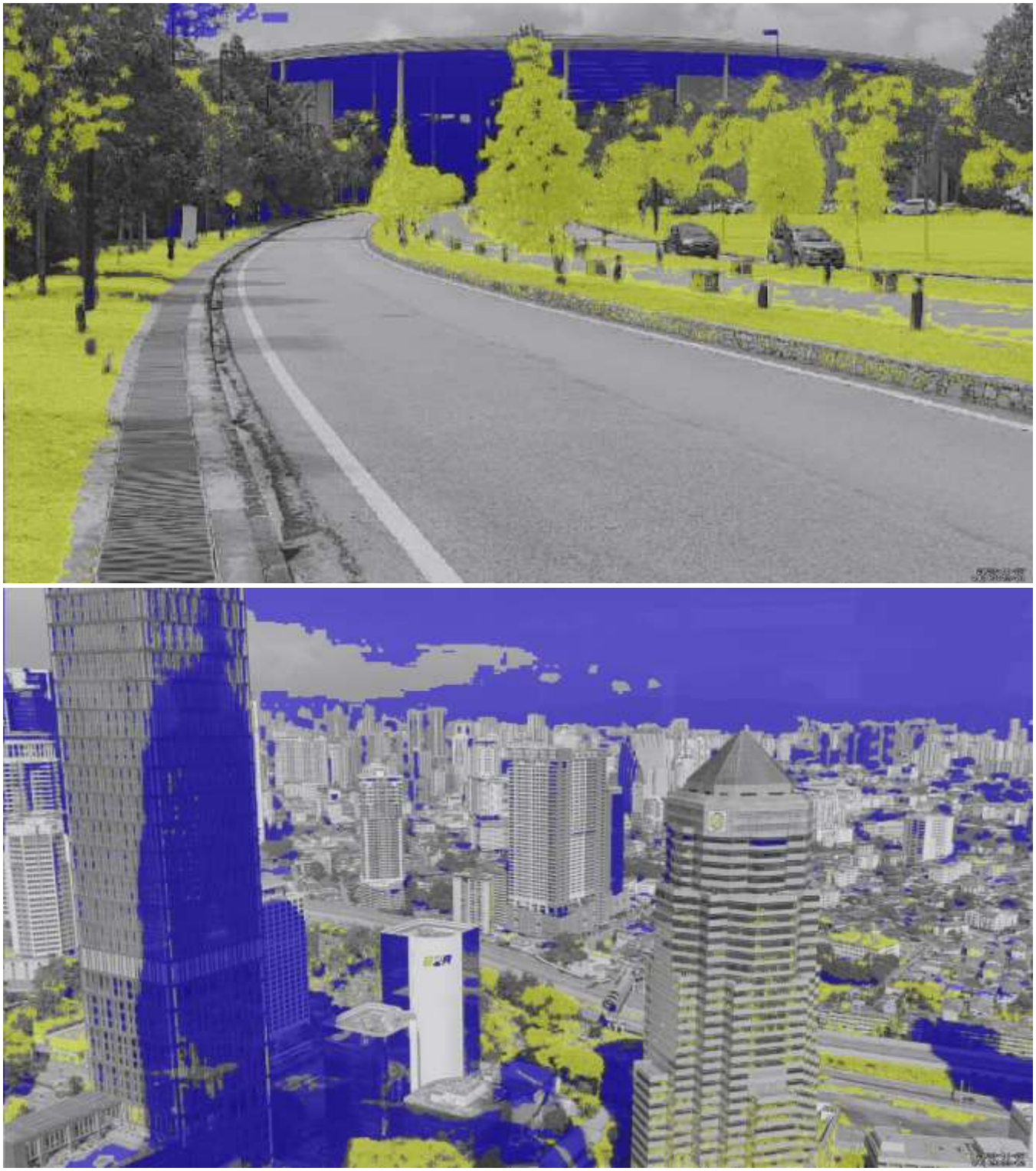
You are working on FPGA:CIV1

You are working on FPGA:CIV1

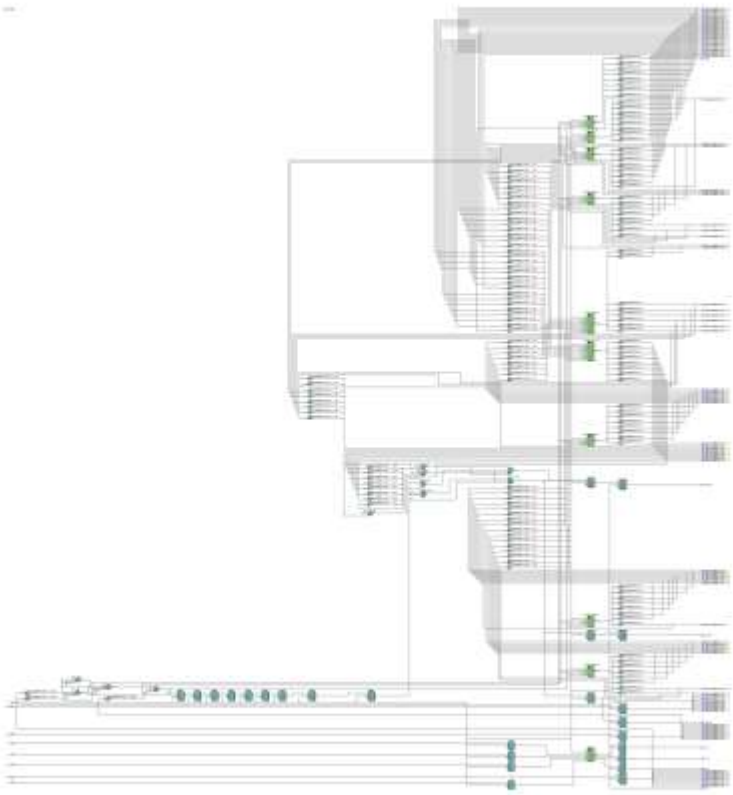


Input image:

Output Image:



Schematic Diagram:



Conclusion:

Verilog has established itself as a powerful tool for detecting yellow and blue light in images. By harnessing Verilog's efficiency and versatility, researchers have devised innovative hardware architectures that can effectively identify and extract these specific colors from diverse image sources. The ability to accurately detect yellow and blue light in images holds significant implications for various fields, including medical imaging, security, and robotics.

Within the realm of medical imaging, detecting yellow and blue light can assist in the diagnosis and treatment of various ailments. For instance, identifying yellow lesions on the skin can aid in detecting jaundice, while recognizing blue hues in X-rays can help pinpoint bone fractures.

In the context of security, detecting yellow and blue light can enhance surveillance systems and improve threat identification. For example, distinguishing yellow from orange in traffic signals can optimize traffic flow management, while recognizing blue lights in emergency vehicles can facilitate faster response times.

In the field of robotics, detecting yellow and blue light can enable robots to navigate their environments more autonomously and interact with objects more effectively. For instance, recognizing yellow markings on floors can guide robots along designated paths, while identifying blue lights on machinery can alert robots to potential hazards.

As image processing techniques continue to evolve, Verilog's role in detecting yellow and blue light is expected to expand further. With advancements in hardware architectures and algorithm optimization, Verilog-based solutions will play an increasingly crucial role in various applications, leading to enhanced image analysis capabilities and improved decision-making processes.

Research Gap:

Some Optimization is required in detecting small red light in an image and light green color because it is detected as yellow.

References:

A Hardware Architecture for Color Detection in Images Using Verilog", by S. S. Kumar, K. S. Reddy, and P. S. Reddy, in International Journal of Scientific Research and Engineering, vol. 10, no. 11, pp. 1-5, 2014.
FPGA-based Color Detection System Using Verilog", by P. B. Patil, S. S. Agrawal, and M. R. Ingle, in International Journal of Advanced Research in Computer Science and Engineering, vol. 4, no. 5, pp. 113-121, 2014.
Color Image Processing Using Verilog HDL", by P. R. Nallur and M. M. Ranganath, in International Journal of Scientific and Engineering Research, vol. 5, no. 10, pp. 2973-2976, 2014.
FPGA Implementation of Color Detection Algorithm for Skin Detection", by S. S. Kumar, K. S. Reddy, and P. S. Reddy, in International Journal of Computer Applications, vol. 110, no. 2, pp. 9-12, 2015.

FPGA-based Color Detection for Object Identification", by R. K. Srivastava, D. K. Shukla, and S. K. Srivastava, in International Journal of Computer Applications, vol. 112, no. 2, pp. 24-27, 2015.