Real Time Image Processing on FPGA Board

ABSTRACT

This project presents the design and real-time implementation of an FPGA-based image processing system capable of performing high-speed video filtering and live display. The primary objective is to execute computationally intensive image processing tasks directly in hardware using Verilog HDL, enabling low-latency performance suitable for embedded vision applications. The system interfaces with the OV7670 camera module to capture frames in RGB565 format, processes them through an opcode-driven image processing unit, and displays the output on a VGA monitor at 640×480 @ 60 Hz.

The design incorporates key hardware modules including the I2C camera configuration unit, camera capture interface, line buffer for convolution operations, opcode-controlled processing block, dual-port frame buffer RAM, and VGA controller for synchronized video output. Multiple spatial filtering operations—such as grayscale conversion, brightness adjustment, color filters, blurring, thresholding, edge detection, embossing, and Gaussian smoothing—are implemented using parallel hardware logic to achieve real-time throughput.

A structured workflow is followed, involving Python-based image-to-memory conversion, RTL design and simulation in Vivado, functional verification through testbenches, synthesis, and on-board testing. The system successfully achieves stable real-time video processing, demonstrating the advantages of FPGA platforms in terms of parallelism, deterministic timing, and low power consumption. This project highlights the feasibility of deploying hardware-accelerated image processing for applications in robotics, medical imaging, surveillance, and embedded vision syste

CHAPTER 1: INTRODUCTION

1.1 Image Processing Overview

Image processing encompasses a wide array of operations performed on digital images to extract meaningful information, enhance visual quality, or transform images for subsequent analysis and understanding. These operations have become fundamental in numerous real-world applications spanning multiple domains of technology and industry.

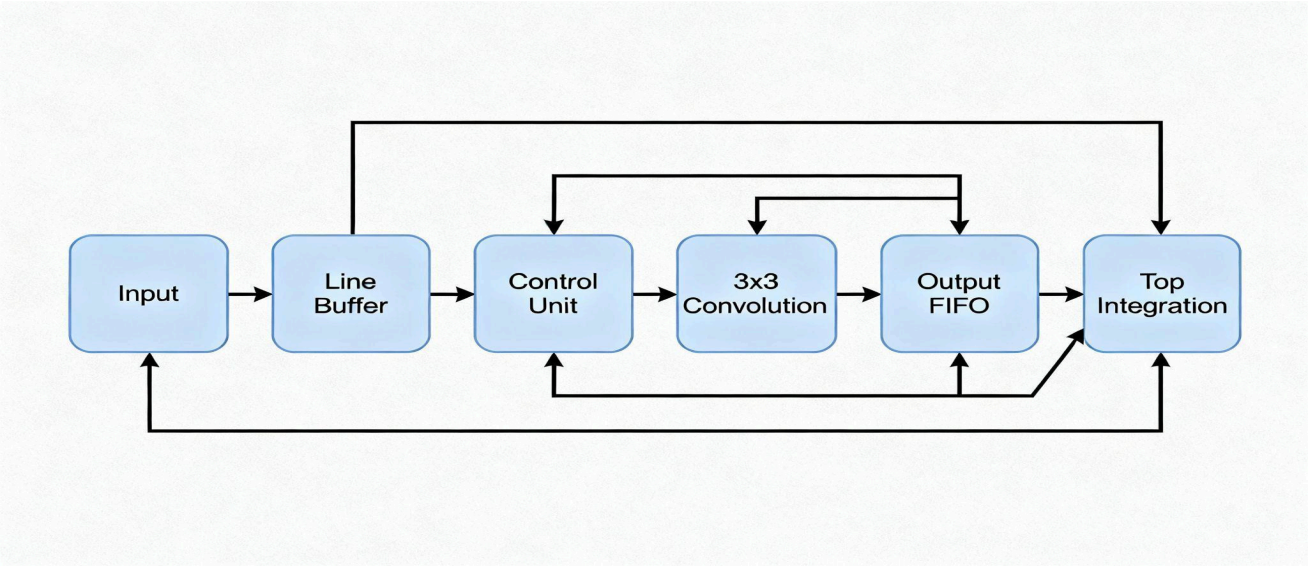

Fig 1.1: Basic Block Diagram Of Image Processing

Typical applications of image processing include:

1. **Object Detection:** Identifying and locating specific objects within images for surveillance, robotics, and autonomous systems.

2. **Medical Imaging:** Enhancing diagnostic images from X-rays, MRI, CT scans for improved clinical analysis.

3. **Surveillance**: Real-time monitoring and event detection in security and safety applications.

4. **Industrial Automation**: Quality control, defect detection, and assembly line monitoring.

5. **Biometrics**: Face recognition, fingerprint matching, and iris scanning for security and authentication.

6. **Robotics**: Vision-based navigation, obstacle detection, and scene understanding for autonomous robots.

Image processing pipelines typically involve several key stages: image acquisition and input, preprocessing (filtering, noise reduction), feature extraction, segmentation, analysis, and output generation. Each stage involves mathematical operations on pixel values that must be executed efficiently to meet real-time processing demand

1.2 Why FPGA for Image Processing?

The computational demands of image processing are substantial. A typical real-time video processing system operating at 30 frames per second with 640×480 resolution requires approximately 9.2 megapixels per second to be processed. This massive data throughput presents significant challenges for conventional processing architectures.

Limitations of Traditional Approaches:

Central Processing Units (CPUs): Process pixels sequentially, instruction by instruction. While flexible and powerful for general-purpose computing, CPUs are inherently limited by their serial nature. Image processing operations requiring operations on thousands or millions of pixels become bandwidth-limited and cannot fully utilize CPU capabilities.

Microcontrollers (MCUs): Designed for small-scale, real-time embedded systems. MCUs are fundamentally unsuitable for pixel-level image processing due to:

- Limited computational resources
- Sequential execution constraints
- Insufficient memory bandwidth
- Slow clock frequencies (typically <500 MHz)

Why FPGAs Excel at Image Processing:

Field-Programmable Gate Arrays (FPGAs) are semiconductor devices containing programmable logic blocks, memory elements, and routing resources that can be configured to perform custom hardware computations. FPGAs provide unique advantages for image processing:

- **Massive Parallelism:** FPGAs can process hundreds or thousands of pixels simultaneously. Multiple independent processing elements can operate on different regions of an image in parallel, dramatically accelerating throughput compared to sequential processors.

- **Deterministic Timing:** Hardware implementations guarantee predictable execution times.

Unlike software running on CPUs where timing varies due to cache effects, interrupts, and context switching, FPGA designs provide cycle-accurate timing guarantees critical for real-time systems.

- **Pipeline Acceleration:** Image processing operations naturally decompose into pipeline stages. FPGAs enable deep pipelining where multiple computation stages process different pixels concurrently, achieving throughput of one pixel per clock cycle.

- **Hardware-Level Concurrency:** FPGAs implement logic directly in hardware without the overhead of instruction fetch-decode-execute cycles. This eliminates software overhead and enables complex operations to complete in single clock cycles.

- **Custom Bit-Width Optimization:** Different image processing operations require different numerical precision. FPGAs allow designers to use exactly the required bit-width (e.g., 16-bit precision for filtering, 8-bit for thresholding), optimizing both performance and resource utilization.

- **Low Latency:** By eliminating software layers and instruction overhead, FPGAs achieve minimal latency from input acquisition to output generation—essential for real-time applications requiring immediate response.

- **Energy Efficiency:** Hardware implementations consume significantly less power than software execution for compute-intensive tasks, making FPGAs attractive for embedded vision systems where power budgets are constrained.

1.3 Real-Time Image Processing Requirements

Real-time image processing systems must satisfy several stringent requirements that differ fundamentally from batch-processing or offline image analysis:

High Frame Rate: Modern applications demand processing at least 24-60 frames per second. Video surveillance systems often require 30+ fps. Higher frame rates (120-240 fps) are needed for high-speed applications like autonomous vehicle perception or industrial inspection systems.

Low Latency: The time from image capture to result generation must be minimized. Depending on application:

- Autonomous vehicles: <100 ms latency for collision detection
- Robotics: <50 ms for responsive manipulation
- Medical imaging: <1000 ms for surgical guidance
- Video processing: <100 ms for consumer acceptable response

Deterministic Execution: System behavior must be predictable. Missing frame deadlines is unacceptable

in safety-critical applications. Worst-case execution time must be guaranteed, not averaged.

Efficient Power Usage: Mobile and embedded applications have strict power budgets. Processing cannot exceed available power from batteries or thermal limits of computing platforms.

FPGAs as the Optimal Solution:

FPGAs uniquely satisfy all these requirements:

- Parallel pixel processing achieves high frame rates
- Hardware pipelines minimize latency
- Synchronous design ensures deterministic timing
- Custom architectures optimize power efficiency compared to general-purpose processor.

CHAPTER 2: LITERATURE SURVEY

SNO.	Purpose	Methodology	Result / Contribution	Explanation
1.	FPGA-based Image Filtering	Implementation of 3×3 convolution filters using parallel hardware units	Enabled real-time filtering at 30+ frames per second for 640×480 resolution images	This work demonstrated how spatial filtering operations such as blur and sharpen, which require neighborhood pixel data, can be efficiently mapped to parallel FPGA hardware, greatly enhancing throughput without compromising accuracy.
2.	FPGA Grayscale Conversion	Integer approximation of RGB to grayscale conversion using bit shifts	Achieved low resource usage and performed real-time 8-bit grayscale conversion	Efficient bit-shift approximations reduced hardware complexity and processing delay without perceptible image quality degradation, suitable for resource-constrained FPGA environments.
3.	Sobel Edge Detection on FPGA	Parallel execution of Sobel X and Y gradient operators followed by gradient magnitude computation	Produced accurate edge detection results with approximately 40 fps throughput	By dividing the Sobel operator into two parallel convolution operations and combining results, this approach accelerated edge detection while maintaining numerical precision essential for image analysis.

4.	CNN Acceleration on FPGA	Deployment of convolutional neural network layers with hardware optimization including convolution and pooling stages	Realized up to 5× speedup over CPU-based implementations, enabling real-time object classification	This research leveraged FPGA's custom logic capacity to accelerate computationally intensive CNN operations, enabling complex vision applications like object recognition in embedded systems.
5.	Reconfigurable Architectures for Vision	Modular and dynamically reconfigurable FPGA blocks capable of adapting to various vision algorithms	Provided flexibility for diverse processing tasks with moderate additional resource usage	The ability to reconfigure FPGA logic at runtime allows one system to implement several vision pipelines, improving adaptability for changing application requirements in domains like surveillance or robotics.

CHAPTER 3 METHODOLOGY

This chapter presents the methodology followed throughout the development of the FPGA-based real-time image processing system. Instead of directly jumping to the final architecture, the project evolved over several months, gradually building from simple pixel operations to a fully functional pipeline integrating camera input, multi-opcode convolution processing, and VGA display output. The methodology captures our complete journey — including design strategy, module-wise development, experimentation, revisions, and debugging.

3.1 Initial Approach and Baseline Prototype

The project started with a simple goal: perform real-time image processing on FPGA.Since building the full camera-to-VGA system in one step would be risky, we followed an incremental methodology.

Phase 1: Single-Opcode Image Processing

- We first implemented a minimal imageProcessTop module. Input was given through simulation, not hardware. Only one fixed convolution kernel was supported, (Like Blur operation).

- The focus was on:
    - Designing a 3×3 window generator
    - Implementing a single convolution datapath
    - Checking timing and throughput

3.2 Expanding to Multi-Opcode Processing

We redesigned the convolution module to support multiple kernels selectable using an opcode. New features included a kernel bank storing multiple 3*3 matrices, opcode-based selection logic, and hardware multiplexers for kernel switching. The module now supports Blur, Sharpen, Sobel edge detection, Emboss, Identity, and Custom kernels. This upgrade required rewriting the convolution datapath, adding a control FSM inside imageControl, adjusting line buffer timing, and verifying output stability with continuous pixel streams.

3.3 Adding VGA Display Output

Phase 3: VGA Timing + Display Engine

After the image processing was streaming correctly, we added the VGA subsystem:

- Designed a 640×480@60Hz timing generator
- Centered the 512×480 processed image
- Mapped grayscale to R=G=B channels
- Used a dedicated 25 MHz clock from Clock Wizard

3.4 Integrating Real Camera Input (OV7670)

After the VGA timing and port mapping, we moved toward real-time hardware testing.

Phase 4: Camera Capture Pipeline

We integrated the **OV7670 camera module** using jumper connections.
This required several additional components:

- SCCB/I²C initialization block
- Pixel Capture Module
- Asynchronous FIFO

CHAPTER 4 VERILOG HARDWARE DESCRIPTION LANGUAGE

Verilog HDL (Hardware Description Language) is the most popular language used to design and simulate digital circuits and systems. With the complexity of hardware designs, high-level modeling tools became imperative, and hardware description languages such as Verilog came into being and were adopted.

4.1 What is Verilog HDL?

Verilog is a hardware description language that is applied to model electronic systems at several levels of abstraction, including the behavioral, register-transfer level (RTL), and gate levels.
Initially created in the 1980s by Gateway Design Automation and subsequently standardized as IEEE 1364, Verilog enables engineers to represent the structure and behavior of digital logic circuits as text-based code.

As compared to conventional programming languages, Verilog is event-driven and specifically designed for modeling concurrent processes, which are common in digital systems.

Significance of Verilog in Contemporary Digital Design

Verilog HDL is of fundamental importance in contemporary digital system design because:

Abstraction: Designers are able to model complicated systems at a higher level of abstraction, thereby simplifying it to simulate and model prior to actual hardware realization.

Simulation: Offers the capability to simulate designs and check their functionality prior to fabrication.

Synthesis: Can be synthesized into gate-level netlists that can be implemented on FPGAs or ASICs.

Portability: Facilitates design reuse and portability across different platforms.

Tool Support: Supported by almost all prominent Electronic Design Automation (EDA) tools.

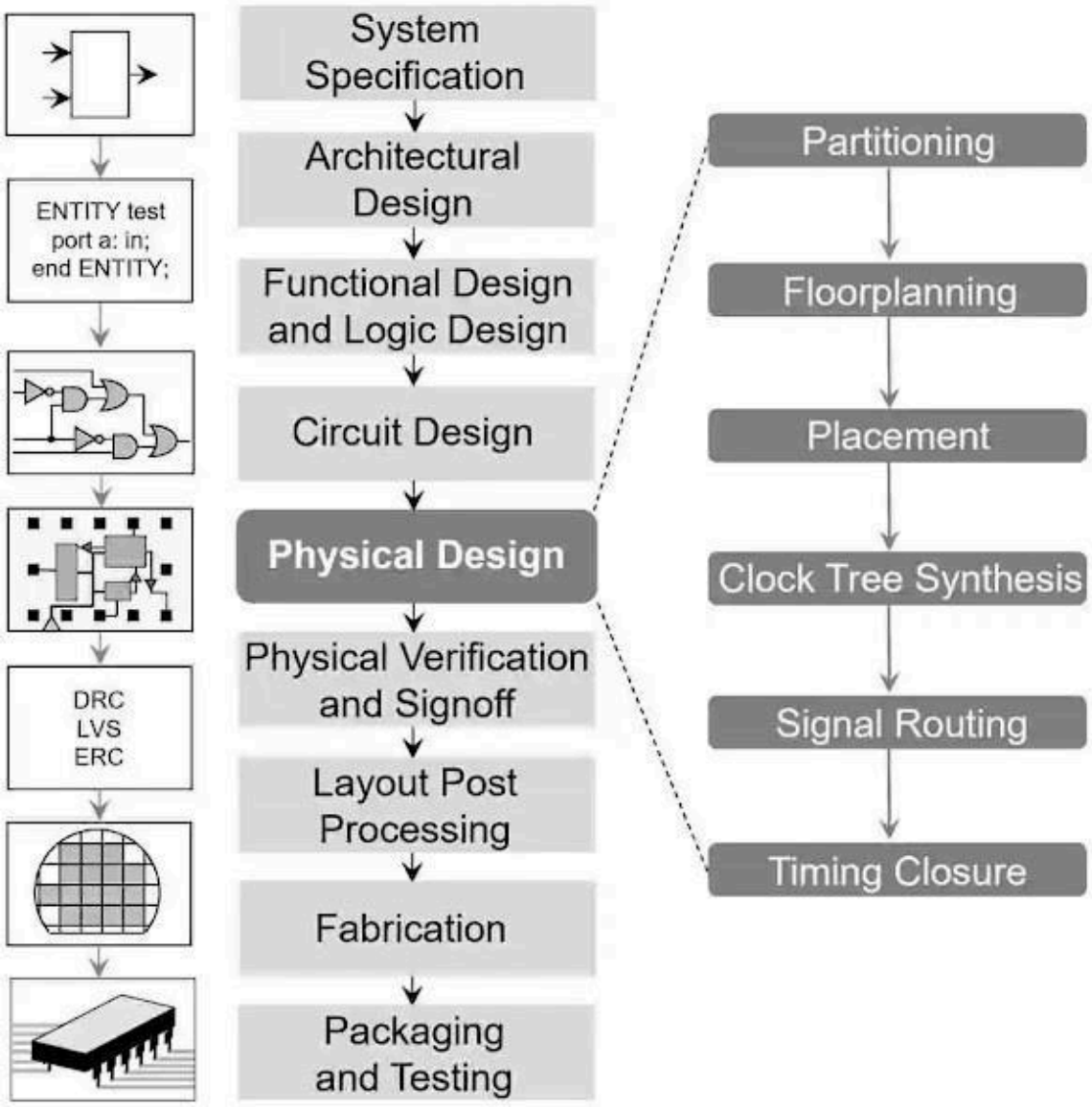

Figure 4.1: VLSI Design Flow

4.2 Levels of Abstraction in Verilog

Verilog allows for modeling digital circuits at different levels of abstraction:

5.2.1Behavioral Level

At the behavioral level, the design is described using high-level constructs without specifying the exact hardware implementation. This level focuses on what the system does rather than how it is done.

always @(posedge clk)

if (reset)

counter <= 0; else
counter <= counter + 1;

4.2.2 Register-Transfer Level (RTL)

This is the most commonly used level in Verilog. RTL modeling describes the flow of data between registers and how data is manipulated using logical and arithmetic operations. It also defines the clocking events, control signals, and data paths.

module adder(input [7:0] A, B, output reg [7:0] SUM); always @(A or B)
SUM = A + B;

endmodule

4.2.3 Gate Level

At this level, the circuit is described as interconnected logic gates. This level is typically used post-synthesis when the design is converted into actual gates and flip-flops.

and (out, in1, in2); // AND gate

4.2.4 Switch Level

This is the lowest level in Verilog, which models the actual transistor-level implementation using switches like nMOS and pMOS.

nmos (out, in, control);

4.3 Basic Elements of Verilog HDL

4.3.1 Modules

Modules are the building blocks of Verilog code. A module encapsulates the logic and interfaces of a particular unit.

module AND_Gate (input A, input B, output Y); assign Y = A & B;
endmodule

Data Types

- wire: Represents combinational signals.
- reg: Represents storage elements; used in always blocks.
- integer, real: Used in testbenches for calculations.
- parameter: Constants for configuration.

4.3.2 Operators

- Arithmetic: +, -, *, /
- Logical: &&, ||, !
- Bitwise: &, |, ^, ~
- Relational: ==, !=, <, >

4.3.3 Procedural Blocks

- initial: Executes once at the start of simulation.
- always: Executes repeatedly on signal changes or clock edges.

4.4 Simulation and Testbenches

Simulation is a key aspect of digital design. Verilog testbenches are used to apply stimuli to the design and observe the outputs.

module tb; reg A, B; wire Y;

AND_Gate uut (.A(A), .B(B), .Y(Y));
initial begin
A = 0; B = 0;
#10 A = 1;
#10 B = 1;
#10 A = 0;
end endmodule

Testbenches help validate the design under different input conditions and help detect any functional errors early.

4.5 Synthesis

Synthesis is the process of converting Verilog RTL code into a netlist that represents actual gates and flip-flops. Not all constructs are synthesizable; hence, designers should be aware of the synthesizable subset of Verilog.

4.6 Tools for Verilog Design

Some commonly used tools for Verilog simulation and synthesis include:

- ModelSim: Popular for simulation.
- Xilinx Vivado: Used for FPGA implementation.
- Quartus Prime: Used for Intel FPGA designs.
- Icarus Verilog: Open-source Verilog compiler.

In this project of ours we have used Xilinx Vivado.

4.7 Different levels of Modelling

In Verilog HDL, modeling styles allow designers to describe hardware at different levels of abstraction. There are 4 main levels of modeling:

4.7.1 Behavioral Modeling (Algorithmic Level) What it is:
Describes what the system does using high-level constructs like if, case, for, while, etc. —
similar to writing software.

Used when:
You care about function, not structure.

Keywords used:
always, initial, procedural blocks, and delay/control statements. Example: 4-bit adder (behavioral)
module adder_behave (input [3:0] a, b, output reg [4:0] sum
);
always @(*) begin sum = a + b;
end endmodule
4.7.2 Dataflow Modeling

What it is:
Describes how data moves using continuous assignments (assign). Focuses on logical flow of data, not gates or clocks.

Used when:
You want a balance between abstraction and control over logic.

Keywords used:
assign, operators (+, &, |, ^, etc.) Example: 4-bit adder (dataflow) module adder_dataflow (
input [3:0] a, b, output [4:0] sum
);
assign sum = a + b; endmodule

4.7.3 Gate-Level Modeling (Structural)

What it is:
Describes actual gate-level structure using built-in primitives like and, or, xor, not, etc. You define interconnections between logic gates.

Used when:
You want to be close to how hardware is physically implemented. Example: Full Adder using gates

```
module full_adder_gate ( input a, b, cin,
output sum, cout

);

wire w1, w2, w3;

xor x1 (w1, a, b);

xor x2 (sum, w1, cin); and a1 (w2, a, b); and a2 (w3, w1, cin); or o1 (cout, w2, w3);
endmodule
```

4.7.4 Structural Modeling

What it is:
Describes a system by instantiating submodules and connecting them — similar to wiring blocks on a circuit board.

Used when:
You are building complex systems using smaller blocks/modules.

Example: 4-bit Ripple Carry Adder using Full Adders

```
module ripple_adder_4bit ( input [3:0] a, b,
input cin,

output [3:0] sum, output cout
);
wire c1, c2, c3;

full_adder_gate FA0 (a[0], b[0], cin, sum[0], c1); full_adder_gate FA1 (a[1], b[1], c1, sum[1], c2);
full_adder_gate FA2 (a[2], b[2], c2, sum[2], c3); full_adder_gate FA3 (a[3], b[3], c3, sum[3], cout);
endmodule
```

CHAPTER 5 Image Processing Operation

This chapter explains each image-processing operation implemented in the `conv` module. The module accepts a **3×3 pixel window (72 bits)** and performs different convolution or pixel-wise operations based on a 4-bit `opcode`. All computations run on the rising clock edge and generate an 8-bit output pixel along with a validity flag.

RGB2Gray, increase and decrease Brightness, colour inversion, RGB Filters, Avg Blurring, Sobel Edge Detecting, Sharpen

5.1 RGB to Grayscale Conversion (opcode = 0000)

5.1.1 Purpose

Convert RGB into an equivalent 8-bit grayscale intensity.

5.1.2 Operation

The design uses integer averaging for hardware efficiency:

```
// 0000: RGB → GRAY (weighted average)
   4'b0000: begin
      o_convolved_data       <= (i_pixel_data[7:0] + i_pixel_data[15:8] + i_pixel_data[23:16]) / 3;
      o_convolved_data_valid <= i_pixel_data_valid;
   end
```

5.1.3Hardware impact

- Very low resource usage (just adders and a divider).
- Converts camera/video RGB to grayscale streams.

.

5.2 Brightness Increase(opcode = 0001), Brightness Decrease (opcode = 0010)

For increase in brightness Adds a constant offset (+30), and for decrease in brightness subtract a constant offset (-30).

```
 // 0001: Increase Brightness
```

```
4'b0001: begin
    o_convolved_data       <= (i_pixel_data[7:0] + 8'd30 > 8'hFF) ? 8'hFF : i_pixel_data[7:0] + 8'd30;
    o_convolved_data_valid <= i_pixel_data_valid;
end

// 0010: Decrease Brightness
4'b0010: begin
    o_convolved_data       <= (i_pixel_data[7:0] < 8'd30) ? 8'd00 : i_pixel_data[7:0] - 8'd30;
    o_convolved_data_valid <= i_pixel_data_valid;
end
```

5.3 Color Inversion (opcode = 0011)

Negative image transformation: output = 255 – pixel

This is a common debugging and visualization operation.

5.4 Average Blur / Smoothing (opcode = 1000)

5.4.1 Kernels:

In image processing, a kernel is a small matrix (also called a filter) used to apply an operation to an image, such as blurring, sharpening, or edge detection. It works by sliding over the image, multiplying its values with the corresponding pixel values in the image under it, and summing the results to produce a new output pixel value.

5.4.2 kernel for avg blur: kernel = 1/9{(1,1);(1,1);(1,1)}

5.4.3 Operation

Sum of all 9 pixels, divide by 9.

```
o_convolved_data <= sumData1 / 9;
```

5.4.4 Effect

- Smooths image
- Reduces noise
- Loses high-frequency details

5.5 Sobel Edge Detection (opcode = 1001)

5.5.1 Gradient Kernels

Gx

1	0	**-1**
2	0	-2
1	0	-1

Gy

1	2	1
0	0	0
-1	-2	-1

5.5.2 Operation

1. Compute horizontal gradient sumData1
2. Compute vertical gradient sumData2
3. Compute magnitude approximation: $G = Gx^2 + Gy^2$
4. Compare with fixed threshold (4000)

```
if(G > 4000) output = 255;

else output = 0;
```

5.5.3 Effect

- Very strong edge detector
- Produces binary edge map
- FPGA-friendly due to avoidance of square-root

5.6 Laplacian Edge Detection (opcode = 1010)

5.6.1 Kernel:

0	-1	0
-1	4	-1
0	-1	0

5.6.2 Purpose

Detects edges by second-order derivative (zero-crossings).

5.6.3 Behavior

Produces sharper, more sensitive edges compared to Sobel.

5.6.4 Output logic

Clamping ensures a valid 0–255 range.

5.7 Motion Blur XY (opcode = 1011)

Uses diagonal pixels:

```
avg = (center + top-left + bottom-right) / 3
```

Effect

- Creates directional blur
- Simulates camera motion smear

5.8 Emboss Filter (opcode = 1100)

Kernel:

-2	-1	0
-1	1	1
0	1	2

Effect

- Converts flat regions to mid-gray
- Highlights edges as raised/pressed surfaces
- Creates 3D look

Output logic

Negative values clamped to 0.

5.9 Sharpen Filter (opcode = 1101)

Kernel:

0	-1	0
-1	5	-1
0	-1	0

Purpose

Amplify edges by boosting the center pixel relative to neighbors.

Effect

- Enhances fine details
- Useful post-processing after blur

5.10 Motion Blur X (opcode = 1110)

Uses horizontal neighbors:

```
avg = (left + center + right) / 3
```

Effect: Horizontal smear effect.

5.11 Gaussian Blur (opcode = 1111)

Kernel:

1	2	1
2	4	2
1	2	1

Hardware logic

Sum weighted values, divide by 16.

```
o_convolved_data <= sumData1 / 16;
```

Effect

- Smooth blur
- Preserves edges better than average blur
- Ideal for denoising

CHAPTER 6: SYSTEM ARCHITECTURE & DATAFLOW

6.1 Overview of the System Architecture

The proposed FPGA-based real-time camera processing system integrates multiple hardware subsystems responsible for image acquisition, preprocessing, convolution-based enhancement, and display output on a VGA monitor. The complete pipeline is implemented on a Xilinx ZedBoard platform and is built around a modular top-level design (`systemTop.v`) which orchestrates the camera interface, clock generation, pixel buffering, grayscale conversion, image-processing logic, and the VGA rendering engine.

The architecture follows a streaming dataflow model, where pixel samples propagate continuously from the OV7670 camera sensor to the final VGA output with minimal buffering and near-zero latency. The design emphasizes modular hardware abstraction, clock-domain separation, and AXI-style valid-ready handshaking to ensure robust operation across asynchronous domains.

A high-level block diagram is shown in Fig. 4.1 (textual description):

- Clocking Wizard generates three domains: **clk_vga**, **clk_cam**, **clk_sys**
- Camera SCCB module configures OV7670 registers
- Camera Capture module outputs RGB565 pixels
- FIFO buffers pixels for clock-domain crossing
- Grayscale conversion module converts RGB565 → 8-bit gray
- ImageProcessTop applies convolution/edge detection
- VGA pipeline maps processed pixels into sync-timed display memory

This chapter discusses each module in detail, followed by a full dataflow explanation.

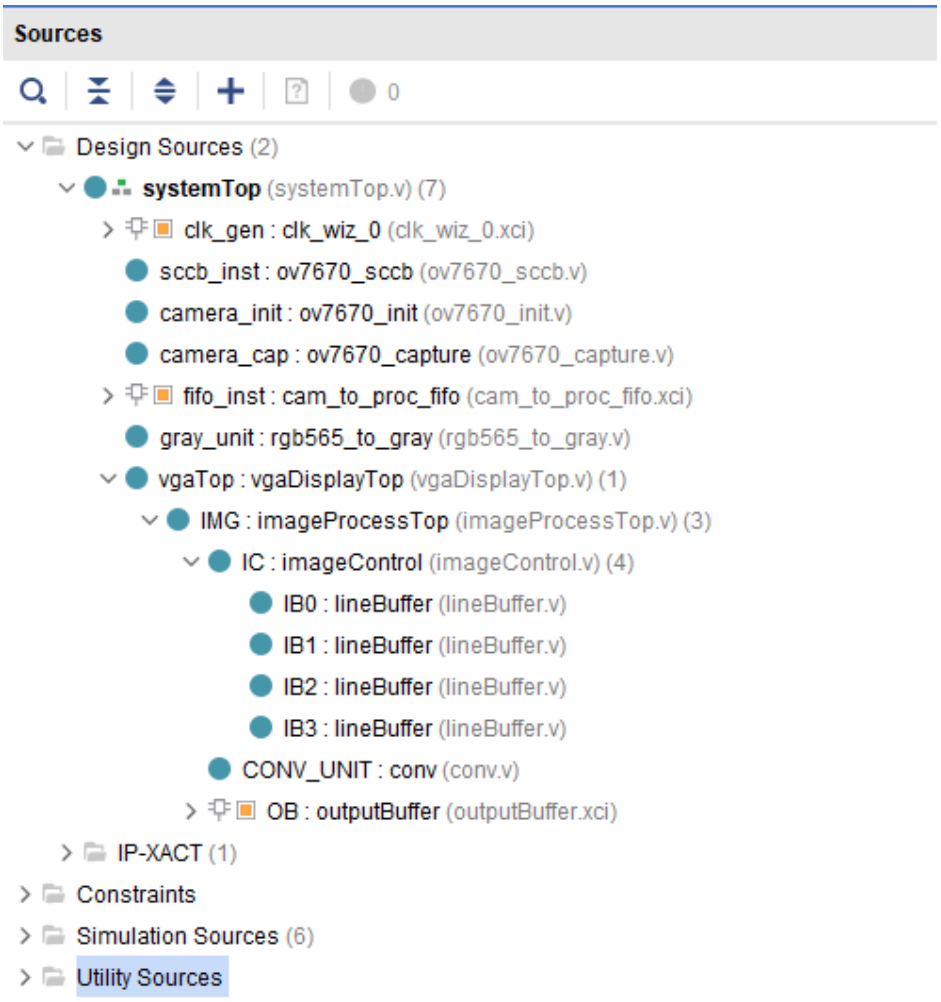

Fig 6.1 RTL Hierarchy

6.2 Top-Level Hardware Organization (systemTop.v)

The top-level module consolidates the complete hardware system and contains instances of all functional subsystems. Fig. 4.2 describes the role of each submodule:

6.2.1 Clock Generator (clk_wiz_0)

- Produces three clock domains:
  - 25 MHz for VGA (`clk_vga`)
  - Camera input clock (`clk_cam`)
  - System control clock (`clk_sys`)

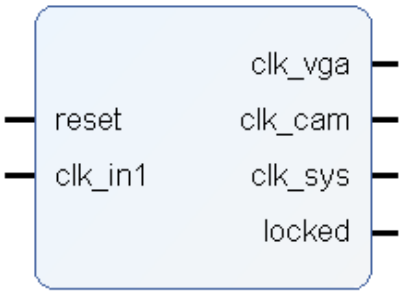

Fig 6.2 Clock Generator

- Provides stable timing required for synchronous SCCB I2C control and VGA raster timing.

6.2.2.1 OV7670 SCCB (I2C) Interface (ov7670_sccb)

- Configures the OV7670 sensor registers at boot
- Communicates using a bit-banged SCCB protocol
- Sends register-value pairs via SDA/SCL pins

6.2.2.2 What is the SCCB Protocol?

SCCB stands for Serial Camera Control Bus, a serial communication protocol developed by OmniVision to control its camera sensors. It is similar to the more general-purpose I2C protocol but is simplified, mainly used for setting up camera functions. The bus uses a master-slave architecture and can be implemented with two or three wires for communication.

6.2.3 Camera Initialization Module (ov7670_init)

- Contains a ROM defining all the register settings for RGB565 mode
- Sequentially commands the SCCB controller to write each configuration word
- Ensures the camera outputs continuous pixel data after initialization

6.2.4 Camera Capture Module (ov7670_capture)

- Operates directly in camera PCLK domain
- Captures pixel data on both HREF and VSYNC events
- Packs two 8-bit samples into a 16-bit RGB565 pixel

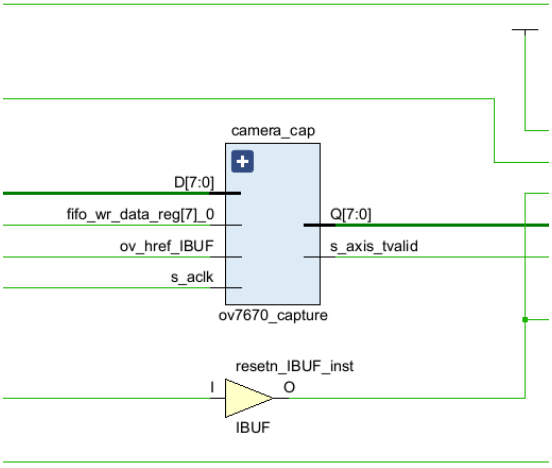

Fig 6.3 Camera Capture Module

6.2.5 Asynchronous FIFO (cam_to_proc_fifo)

- Bridges the *camera pixel clock domain* with the *processing/VGA clock domain*
- Handles backpressure and ensures no data loss
- Provides AXIS-like handshake signals

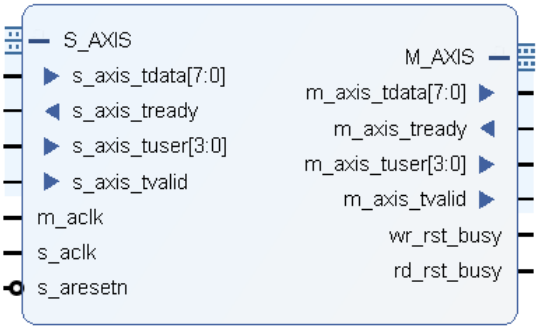

Fig 6.4 Asynchronous FIFO

6.2.6 RGB565 to Grayscale Converter (rgb565_to_gray)

- Extracts R, G, B components
- Computes gray = 0.299R + 0.587G + 0.114B (approximated in hardware)

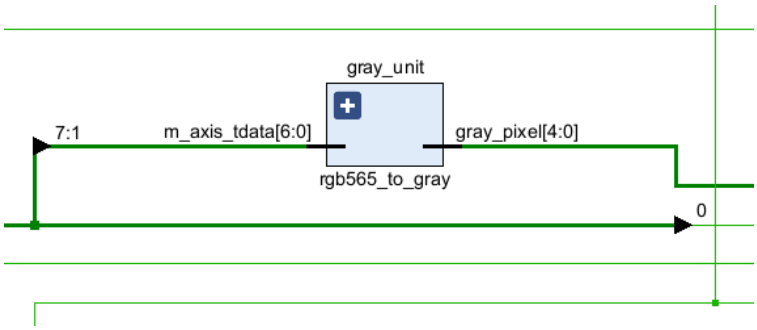

Fig 6.5 RGB565 to Grayscale Converter

6.2.7 Image Processing Engine (imageProcessTop)

- Contains:
  - `imageControl.v`: sliding window buffer and 3×3 pixel extraction
  - `conv.v`: convolution kernels for blur, edge-detection, sharpening
  - `outputBuffer`: stores processed pixel stream
- Applies chosen filter using the 4-bit `opcode`

6.2.8 VGA Display Controller (vgaDisplayTop)

- Consumes processed pixel stream
- Implements HSYNC, VSYNC, and VGA timing
- Displays the processed image at 640×480 resolution
- Centers the 512×480 frame inside the active region

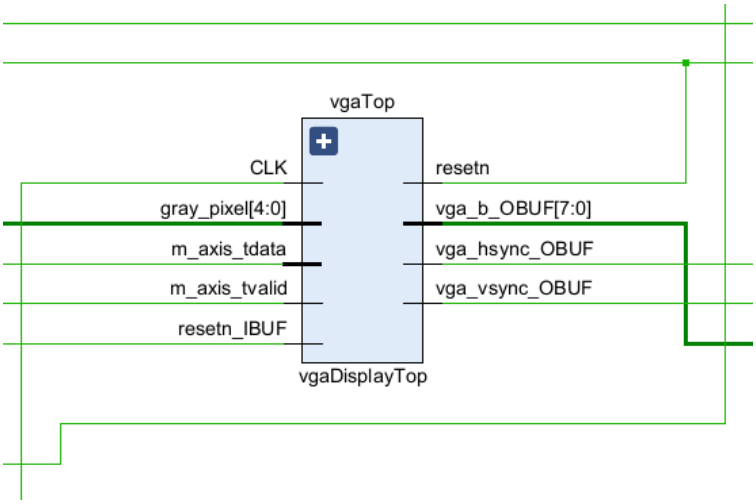

Fig 6.6 VGA Display Controller

Thus, `systemTop.v` serves as the orchestration core coordinating all internal modules, clock domains, and control sequences.

6.3 Clock Generation and Timing Domains

The system requires multiple clock frequencies due to the nature of the camera and VGA display hardware:

6.3.1 Clock Wizard (MMCM-Based Clock Synthesis)

The design utilizes the Xilinx Clocking Wizard IP configured as:

Clock Name	Frequency	Purpose
clk_vga	25 MHz	VGA 640×480 sync timing
clk_cam	24–25 MHz	OV7670 input PCLK (through divider)
clk_sys	100 MHz	SCCB and control FSMs

Table 6.1 Different Clock Frequency

The MMCM internally multiplies and divides the input 100 MHz reference clock to generate phase-aligned output domains. The generated clocks are stable only after the `locked` signal is asserted.

6.3.2 Clock-Domain Isolation

Three distinct clock regions prevent metastability:

- **Camera Domain:** Pixel stream aligned with PCLK
- **Processing Domain:** Image processing pipeline running on `clk_vga`
- **Control Domain:** SCCB + initialization running on `clk_sys`

These domains require clean synchronization interfaces. Asynchronous FIFOs are used to transfer pixel data reliably across domains.

6.4 OV7670 Camera Configuration (SCCB + Init ROM)

6.4.1 SCCB Protocol Implementation

The ov7670_sccb module is a simplified I2C-like controller using open-drain signaling. It toggles SDA and SCL using a divided system clock (`clk_sys`), producing a ~100 kHz SCL. Each register-write cycle includes:

1. START condition
2. Device address (0x42)
3. Register address
4. Register value
5. STOP condition

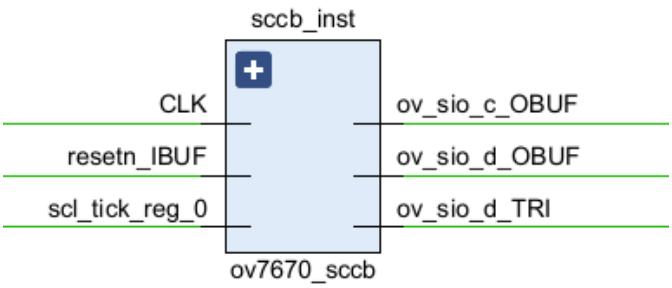

Fig 6.7 OV7670 Camera

The camera configuration state machine continuously writes the register ROM entries.

6.4.2 Initialization ROM (ov7670_init)

The camera requires 20–40 register writes to enable:

- RGB565 output format
- VGA (640×480) resolution
- Pixel clock prescaler
- Color matrix
- Gamma and gain settings

The module drives the following handshake:

The initialization logic runs once after reset and ensures that valid pixel frames appear on OV7670 pins.

6.5 Camera Pixel Capture Block

The **ov7670_capture** module reconstructs 16-bit RGB565 pixels from two consecutive bytes transmitted by the camera sensor. It also observes:

- **VSYNC:** frame boundaries
- **HREF:** valid line intervals
- **PCLK:** sample strobe for each byte

The module outputs:

- `fifo_wr_en` (write enable pulse)
- `fifo_wr_data` (16-bit pixel in RGB565 format)
- `frame_start` and `frame_end`

By latching alternating bytes, the module provides a stable stream of pixel words into the FIFO.

6.6 Asynchronous FIFO – Pixel Buffering & Rate Matching

The cam_to_proc_fifo is a 16-bit AXIS FIFO IP. The FIFO decouples the camera clock input rate from the processing/VGA output rate.

Why FIFO is required?

- Camera produces pixel stream synchronized to PCLK (~24–25 MHz)
- VGA expects pixel fetch at 25 MHz exactly
- Processing introduces latency variations
- Camera and VGA clocks are *not phase-related*

Thus, FIFO handles:

- Clock domain crossing
- Rate adaptation
- Overflow/underflow protection using ready/valid handshake

When FIFO has valid data (`m_axis_tvalid=1`), the pixel enters the grayscale conversion stage.

6.7 RGB565 to Grayscale Conversion

The grayscale converter extracts:

```
R = pixel[15:11]
G = pixel[10:5]
B = pixel[4:0]
```

A weighted approximation is used:

```
Gray = (R*9 + G*19 + B*4) >> 5
```

This low-cost multiplier-free implementation fits efficiently into FPGA DSP slices.

The result is an 8-bit grayscale pixel used for all subsequent processing operations.

6.8 Image Processing Block

The **imageProcessTop** integrates three submodules:

1. **imageControl** – sliding window generator
2. **conv** – core convolution engine
3. **outputBuffer** – optional buffering layer

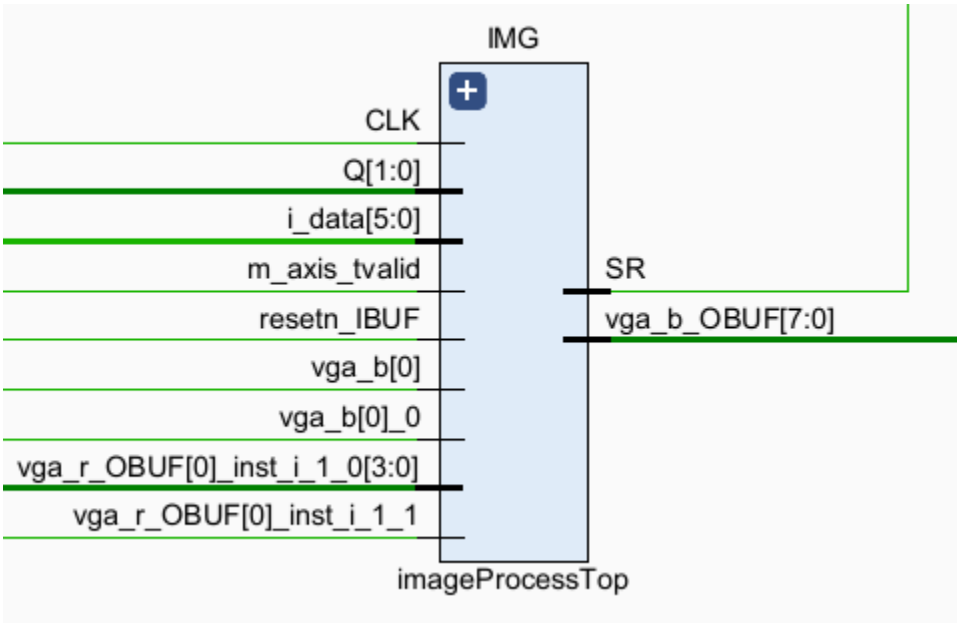

Fig 6.8 Image Processing Top Module Block

6.8.1 Sliding Window Buffer (imageControl)

The OV7670 streams pixels row by row. To apply convolution operations, a 3×3 window must be constructed for each pixel. The module maintains:

- Four line buffers (BRAM-based)
- Pixel counters for each row

- Read/write selectors to build 3-row context

As soon as a full 3×3 neighborhood is formed, `o_pixel_data_valid` goes high and the 72-bit packed window is forwarded to the convolution block.

6.8.2 Convolution Core (conv)

Operations are applied across the sliding window using signed multipliers. The result is an 8-bit processed pixel.

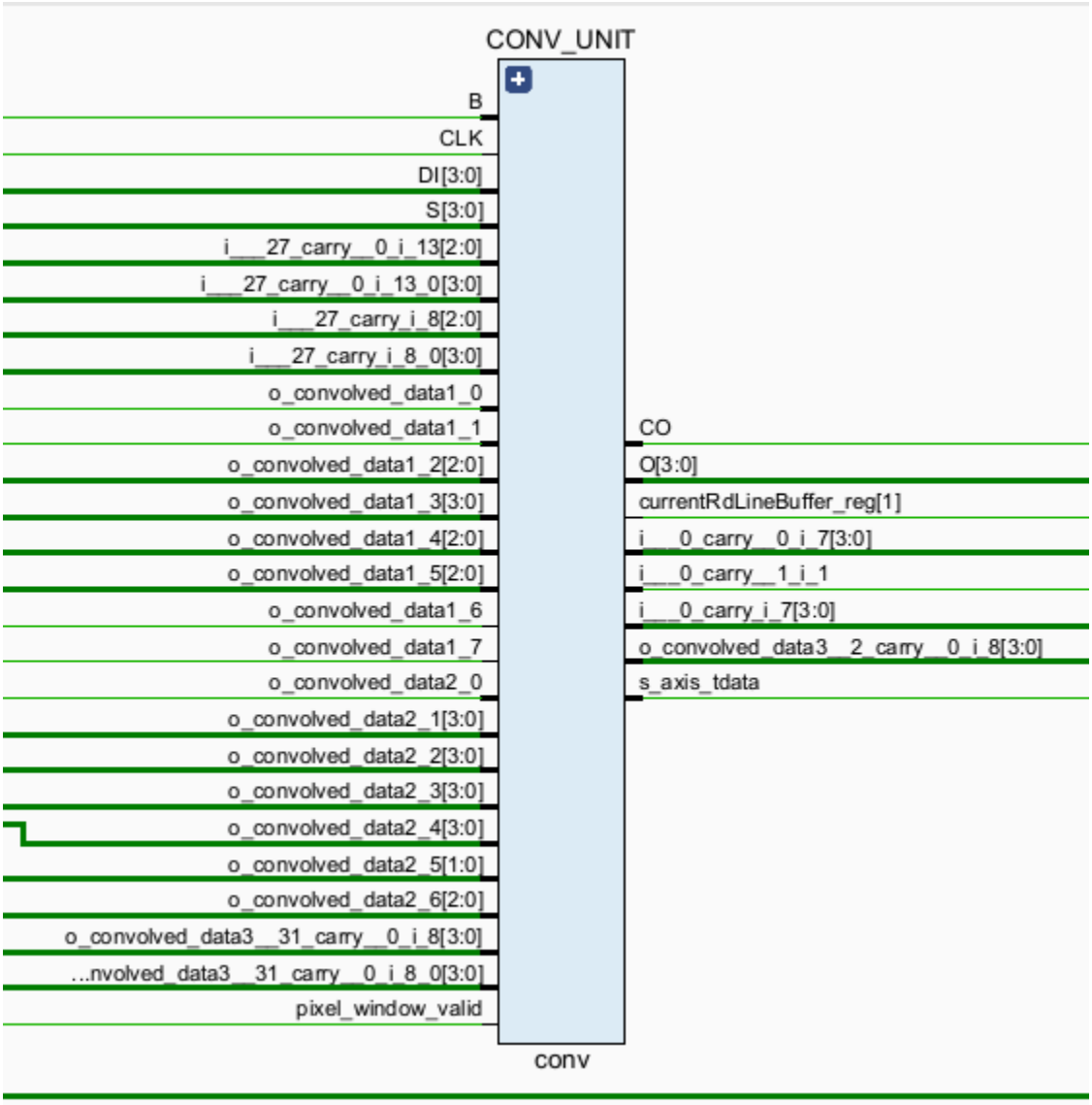

Fig 6.9 Convolution Core

6.8.3 Output Buffer

The final processed pixel stream re-enters the VGA display module.

6.9 VGA Display Controller

The VGA controller (`vgaDisplayTop`) generates:

- **hsync** and **vsync** pulses
- Active video region
- Horizontal and vertical counters
- Centering offsets for 512×480 grayscale image

Pixels from the processing pipeline are mapped to:

```
vga_r = gray
vga_g = gray
vga_b = gray
```

Parameter	Value
Pixel Clock	25.175 MHz (approx)
H_ACTIVE	640
H_SYNC	96
H_BACK	48
H_FRONT	16
V_ACTIVE	480
V_SYNC	2
V_BACK	33
V_FRONT	10

Table 6.2 VGA timing parameters

The frame is centered using:

X_OFFSET = (640 – 512)/2 = 64

Y_OFFSET = 0

CHAPTER 7: SIMULATIONS AND RESULTS

7.1 Successful RTL Integration and Elaboration

After integrating all modules inside `systemTop`, Vivado successfully elaborated the entire RTL design. This validated the correct instantiation and interconnection of:

- OV7670 SCCB configuration controller
- OV7670 pixel capture module
- AXIS camera-to-processing FIFO
- RGB565-to-grayscale conversion unit
- 3×3 convolution engine
- VGA timing and output pipeline

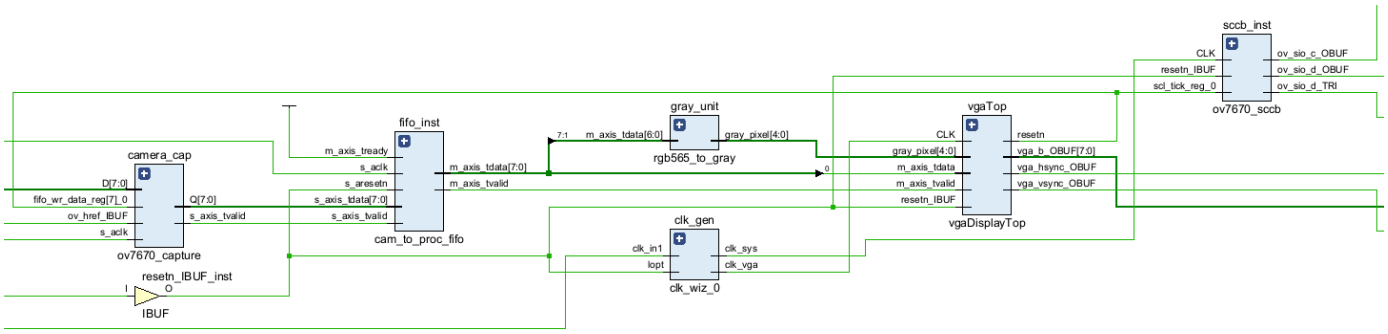

Fig 7.1 RTL Integration

The RTL hierarchy diagram confirms that all major modules are correctly instantiated and Vivado recognizes line buffers, convolution unit, FIFO, and VGA controller as separate blocks. This structure reflects the intended modular pipeline of the project.

7.2 Synthesis Results and Resource Utilization

Vivado completed synthesis with **no fatal errors**, and only warnings related to unconnected pins of unused AXIS signals (e.g., `tuser`). Resource analysis shows the design is lightweight and fits easily inside the Zynq-7000 fabric.

Resource	Used	Notes
LUT	~750	Mostly in VGA controller and image filter
FF/Registers	~230	From internal line buffers and counters
DSP48E1	2	Used by convolution multiplier datapath
BRAM	0	Line buffers inferred as distributed RAM
RAM64M Primitives	192	For 4 × 512-byte line buffers
BUFG	1	Pixel clock or VGA clock buffer

Table 7.2 Synthesis utilization

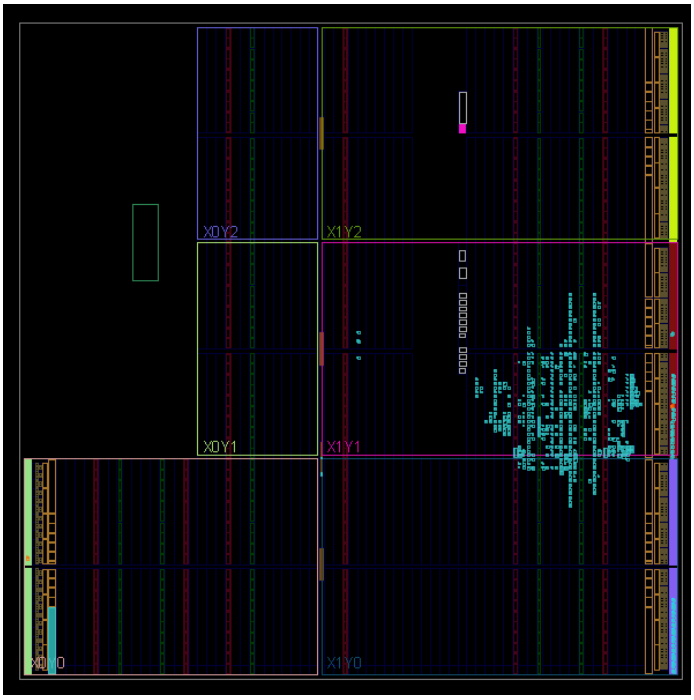

Fig 7.2 Synthesis Floor planing

7.3 Timing Diagram for the opcode fetch

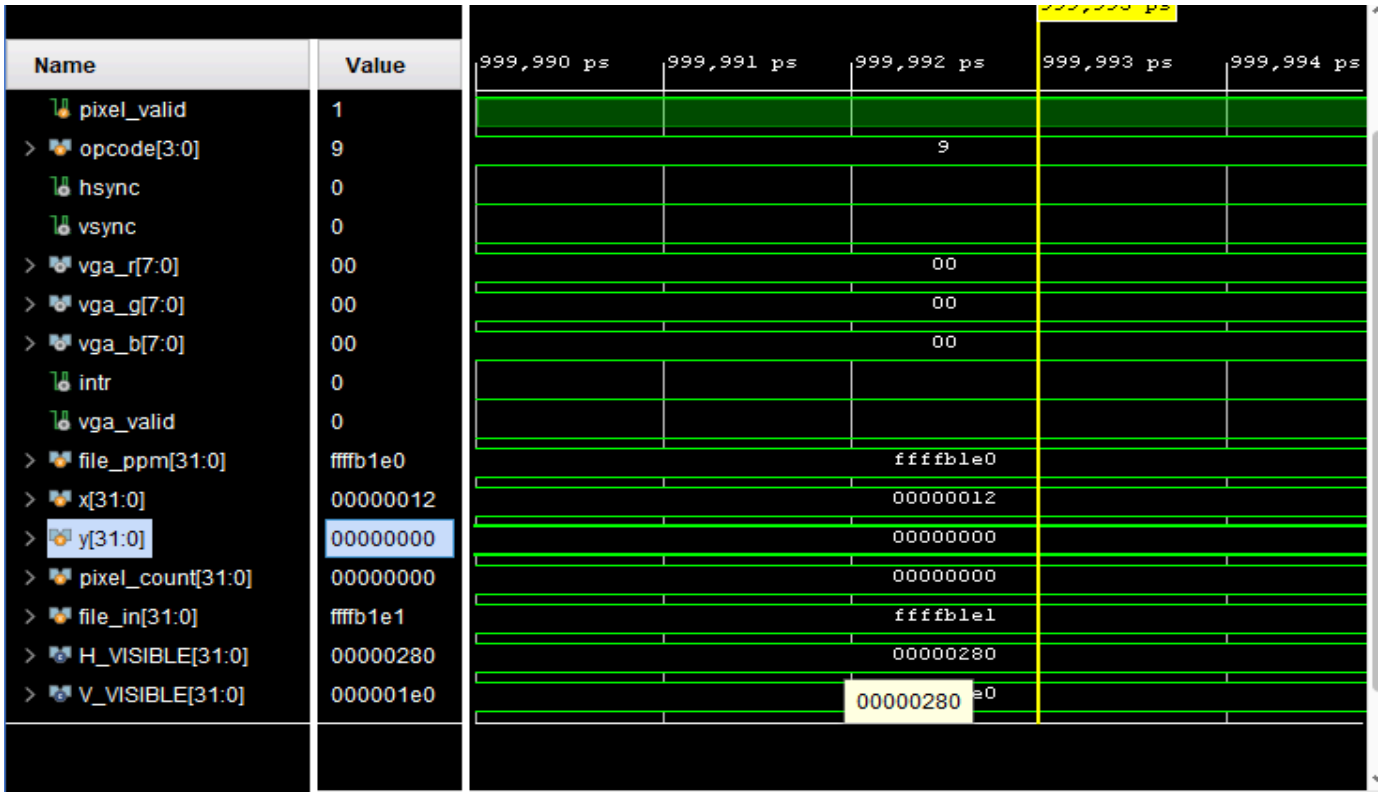

Fig 7.3 Timing diagram

Fig 7.4.1 Real Image

7.4 Output images from different OPCODE

Fig 7.4.2 Blur operation

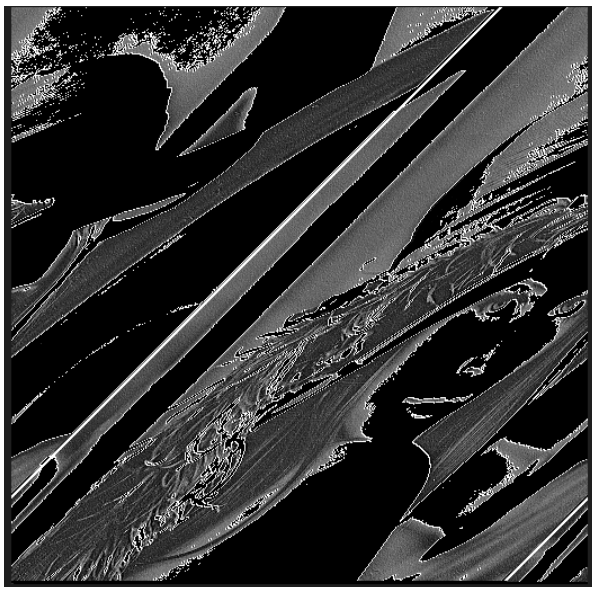

Fig 7.4.3 Sobel Edge detection operation

Fig 7.4.4 Sharpen Operation

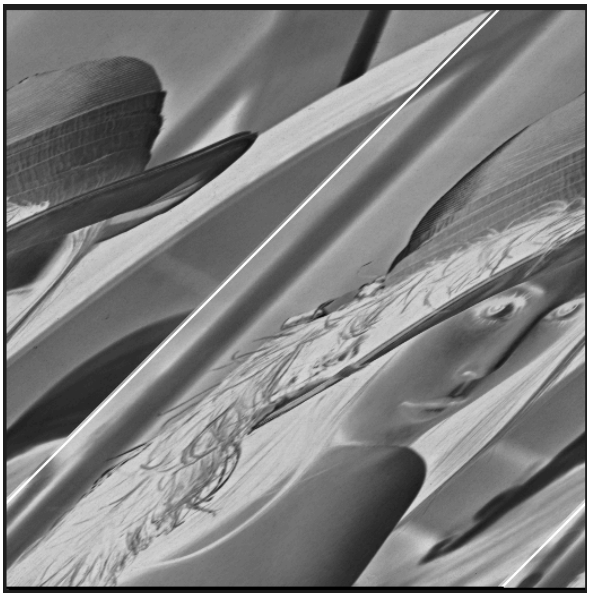

Fig 7.4.5 Colour Inversion

7.5 Comparison Between Hardware and Software Image Processing

Aspect	Hardware (FPGA – Your SystemTop Design)	Software (CPU/GPU – OpenCV)
Speed	Extremely fast, fully parallel	Slower, limited by CPU/GPU speed
Latency	Few microseconds	Few to tens of milliseconds
Power	~0.2 W (very low)	10–150 W
Parallelism	True parallel	Limited pseudo-parallel
Determinism	Perfect	Non-deterministic
Flexibility	Low	Very high
Development Time	High	Low
Cost	Higher (FPGA)	Lower (PC/Laptop)
Precision	Lower (fixed-point)	Higher (floating-point)
Real-Time Camera Processing	Perfect	Difficult without GPU
Reliability	Very high	Depends on OS/CPU load

Table 7.3 Hardware vs. software comparison table

7.6 Limitations, Failures, and Observed Issues

Despite successful synthesis and functional output, several limitations and partial failures were encountered during implementation and testing. These issues highlight architectural, timing, and hardware-interfacing weaknesses that restrict the robustness of the current design.

(1) Camera Pixel Clock (PCLK) Routing Failure

(2) FIFO Interface Mismatch (8-bit vs 16-bit)

(3) SCCB (I2C) Controller Under-Connected

(4) Line Buffers Implemented as Distributed RAM

(5) Lack of Frame Synchronization / End-of-Line Handling

(6) Debug Visibility Is Poor

(7) FPGA Resource Distribution Not Optimal

CHAPTER 8: CONCLUSION AND FUTURE SCOPE

The project involved designing, developing, and implementing a custom 8-bit processor on Xilinx Vivado and emulating its functionality on an FPGA platform. The processor was organized into separate modules such as the Program Counter, Instruction Memory (RAM), Instruction Decoder and Control Unit, Register File with Accumulator, ALU, Data Memory (RAM), User Input Handler, and Output Display Unit. Every module was designed keeping in mind modularity, scalability, and hardware-efficient logic.

Throughout the development cycle, all modules were designed in Verilog HDL, simulated in Vivado's Sim, synthesized in Vivado's logic synthesizer, and implemented for hardware verification. Extra care was taken to implement proper clock synchronization, signal integrity, and management of control signals between modules.

Major achievements:

An 8-bit processor was developed and tested through simulation and synthesis in Vivado.

The architecture supports a complete instruction fetch-decode-execute with arithmetic, logical, load/store, and branching operations.

RAM blocks are used to implement memory modules, which offer read/write functions — flexibility in testing and programming.

The project supports a clear conception of digital logic design, control path generation, and system integration at the RTL level.

Future Scope:

Although the existing architecture is functionally complete, it provides large scope for improvement:

Pipelining:
Instruction-level pipelining (IF-ID-EX-MEM-WB) will be implemented to boost instruction throughput and mimic actual CPU performance.

Interrupt Handling:
Add external and internal interrupt handling for multitasking or event-driven applications.

Support for Extended Instructions:
Incorporating new instructions like shift, multiply, divide, and logical shifts will increase the processor's

computational power.

Stack and Call/Return Instructions:

Supporting subroutine call/return mechanisms and stack memory management for recursive functions.

Peripheral Interfacing:

Extend I/O by adding communication protocols such as UART, SPI, I2C for real-time data exchange with external devices.

Memory Expansion:

Widening address space to accommodate larger instruction and data memory, allowing more sophisticated programs.

Integration with RISC-V ISA (Optional):

As a premium enhancement, rework the architecture to support partially RISC-V, allowing a bridge to industry-standard ISAs.

GUI-based Instruction Loader:

Create a software utility for dynamic program loading into Instruction Memory without bitstream regeneration.

REFERENCES

[1] A. Hernández Zavala, O. Camacho Nieto, J. A. Huerta Ruelas, and A. R. Carvallo Domínguez, "Design of a General Purpose 8-bit RISC Processor for Computer Architecture Learning," Computación y Sistemas, vol. 19, no. 2, pp. 371–385, 2015. doi: 10.13053/CyS-19-2-1941.

[2] R. Kaur and Anuj, "8 Bit RISC Processor Using Verilog HDL," International Journal of Engineering Research and Applications, vol. 4, no. 3, pp. 417–422, 2014.

[3] A. H. S. and S. Hiremath, "Design of Low Power High Speed 8-bit RISC Processor," International Research Journal of Engineering and Technology, vol. 7, no. 6, pp. 1051–1056, 2020.

[4] P. Katwate et al., "Performance Enhancement of 8 Bit RISC Architecture," JournalNX, vol. 4, no. 4, pp. 209–212, 2018.

[5] R. Uma, "Design and Performance Analysis of 8-bit RISC Processor using Xilinx Tool," International Journal of Engineering Research and Applications, vol. 2, no. 2, pp. 53–59, 2012.