Image Processing in FPGA

SUMMER PROJECT 2025

14 July 2025

Contents

1 Introduction

The **Image Processing in FPGA** project aims to implement real-time edge detection using the Sobel filter on an FPGA board. This project explores the intersection of image processing and digital hardware design, leveraging the capabilities of FPGAs to perform parallel data processing with ultra-low latency. By developing a complete hardware-software co-design system, this project not only enhances understanding of Verilog and FPGA workflows but also demonstrates how image data can be manipulated efficiently in real time.

Figure 1: Results of a Sobel edge filter

1.1 Project Objectives

The primary objective of this project is to design and implement a real-time **Sobel edge detection** algorithm on an FPGA using Verilog. The system is designed to read grayscale image data from a host PC, process it using the Sobel filter within the FPGA fabric, and return the output image highlighting edges.

This real-time image processing capability is achieved by leveraging the inherent parallelism and deterministic timing of FPGAs. The project involves writing custom Verilog modules for pixel processing and gradient computation, and designing data pipelines for continuous image stream processing.

A key design strategy is the use of **multiple 3×3 convolution windows in parallel**, enabling the FPGA to apply the Sobel filter to several pixels simultaneously. This parallelism significantly accelerates edge detection and showcases the advantages of hardware-level concurrency.

Communication between the PC and the FPGA is handled through a UART interface, using the **TeraTerm** serial communication software for sending input data and receiving the processed image. This setup allows seamless transmission and visualization of processed frames, and illustrates how FPGAs can interface with external systems in real-world applications.

- Real-time edge detection using the Sobel operator
- Hardware-level parallelism using multiple 3×3 windows

- Efficient resource utilization on an FPGA
- Communication protocol design (UART) for PC–FPGA interfacing using TeraTerm

1.2 Technical Highlights

FPGA-based Acceleration: Harnessing Parallelism

FPGAs provide unparalleled parallelism, which allows for simultaneous processing of multiple pixels—drastically reducing the time required to apply edge detection filters like Sobel. Unlike CPU-based sequential processing, the FPGA processes incoming image data in a pipelined fashion, significantly improving throughput. This is achieved by applying multiple 3×3 Sobel kernels in parallel across the image frame.

This hardware-based implementation is particularly well-suited for applications requiring real-time video processing, such as in robotics, surveillance, or industrial vision systems.

Sobel Filter Implementation

The Sobel operator is implemented directly in hardware using Verilog. The filter computes horizontal and vertical gradients on the grayscale image and combines them to detect edges. A sliding 3×3 window mechanism is developed within the FPGA to apply the Sobel kernel to each pixel in real time, and multiple such windows are instantiated in parallel for increased throughput.

The system design includes:

- Line buffers to hold intermediate pixel rows
- Convolution logic for gradient computation
- Thresholding module to highlight strong edges

Communication Interface and Visualization

To visualize the results, the processed image is sent back to a PC over a UART interface. The data transmission and reception are handled using Tera Term, a terminal emulation program that allows serial communication with the FPGA board. This closed-loop communication and visualization setup demonstrates an effective hardware–software co-design pattern, reinforcing concepts of embedded systems and peripheral interfacing.

1.3 Learning Objectives

Through this project, the following key skills and concepts are aimed to be developed:

- **FPGA Programming**: Writing synthesizable Verilog code and managing timing constraints.
- **Digital Logic Design**: Implementing real-time data pipelines and filters at the gate level.
- **Image Processing Fundamentals**: Understanding edge detection and spatial convolution techniques.

- **Toolchain Proficiency**: Utilizing Xilinx Vivado for synthesis, simulation, and hardware debugging.
- **Hardware-Software Co-design**: Designing communication protocols (UART) and integrating with tools like Tera Term for end-to-end testing.

This project serves as a foundation for further exploration into advanced image processing tasks on reconfigurable hardware platforms and real-world embedded vision applications.

2 Digital Circuits and HDL

2.1 Digital Circuits

2.1.1 Introduction

Digital circuits form the backbone of image processing systems implemented on FPGAs (Field-Programmable Gate Arrays). In the project "Image Processing using FPGA", digital circuits are designed and configured to perform parallel, high-speed processing of image data. Unlike software-based approaches that run on general-purpose CPUs, FPGA-based image processing utilizes custom hardware logic to execute tasks like filtering, edge detection, and enhancement in real time. This not only accelerates processing speed but also offers lower latency and power efficiency, making it ideal for applications such as surveillance, medical imaging, and autonomous systems.

2.1.2 Logic Gates in Digital Circuits

In this project, fundamental logic gates—AND, OR, NOT, XOR, NAND, and NOR—are essential building blocks of the digital circuits implemented on the FPGA. These gates are used to create more complex components like adders, multiplexers, and comparators, which are vital for processing pixel data. For example:

- AND/OR gates are used in masking and combining pixel values.
- XOR gates help in edge detection by highlighting differences between neighboring pixels.
- NOT gates are used for pixel inversion or thresholding.

By combining these gates, efficient hardware-based image processing operations can be executed in parallel, leading to faster and real-time performance.

2.1.3 Combinational circuits

Combinational circuits play a key role in FPGA-based image processing, as they provide fast, deterministic outputs based solely on current input values. In this project, circuits such as adders, subtractors, comparators, multiplexers, and encoders are used extensively.

- **Adders/Subtractors** are used for arithmetic operations like brightness adjustment or convolution.
- **Comparators** help in operations like thresholding, where pixel values are compared to a set value.
- **Multiplexers** are used for selecting between different data paths or image channels.

These circuits enable pixel-wise processing without the need for clock cycles or memory storage, making them ideal for tasks that require high-speed, real-time computation.

2.1.4 K-Map: Karnaugh Map

Karnaugh Maps (K-Maps) are a simplification tool used in digital logic design to minimize Boolean expressions. In this image processing project using FPGA, K-Maps are used to optimize the logic functions that control specific operations, such as pixel classification, thresholding, or control logic.

By reducing complex Boolean expressions into simpler forms using K-Maps, we can design more efficient combinational circuits with fewer logic gates. This optimization leads to lower resource usage on the FPGA, faster processing, and reduced power consumption — all of which are crucial for high-performance, real-time image processing tasks.

2.1.5 Sequential circuits

Sequential circuits are crucial in FPGA-based image processing when operations require memory or timing control. Unlike combinational circuits, sequential circuits depend on both current inputs and past states, making them suitable for handling image frames, pixel streams, and synchronization.

Common sequential components used in the project include:

- **Flip-flops and Registers** – to store pixel values, intermediate results, or flags across clock cycles.
- **Counters** – for scanning rows and columns of an image or controlling timing in pipelines.
- **Finite State Machines (FSMs)** – to manage control logic for tasks like frame buffering, filtering sequences, or data flow coordination.

These circuits allow for synchronous and orderly processing of image data, especially in real-time systems where precise control over timing and sequence is essential.

2.2 HDL - Hardware Description Language

2.2.1 Introduction

Since FPGAs are reconfigurable, we need a tool that allows users to create custom digital logic hardware. For this, we use HDL (Hardware Description Language). HDLs are essential for designing digital hardware because they provide a structured, programmable way to describe how circuits should behave, both in logic and timing. With HDL, we can

- Design Custom Digital Hardware HDL allows you to design **any digital system**, from simple gates to complex processors. You can precisely define how data flows, how logic is applied, and when signals change.
- Simulate Before Fabrication With HDL, you can **simulate your design** and verify functionality *before* implementing it on physical hardware like an FPGA.

- Synthesis to Real Hardware HDL code can be **synthesized** into actual hardware. HDL is the bridge between abstract logic and real circuits.
- Parallel Execution Unlike software code, HDL inherently describes **parallel** behavior — multiple modules operate **simultaneously**.

2.2.2 Verilog

Verilog is a hardware description language with **C-like syntax**, making it easy to learn. It's ideal for writing **RTL (Register Transfer Level)** logic and is widely supported for FPGA synthesis and simulation.

We used HDLbits platform to start learning verilog:

What We Learned from the First Three Sections of HDLBitsGetting Started

- – Introduction to the HDLBits platform and simulation interface.
  - Basic syntax of **Verilog**.
  - Understanding how to define **modules, inputs, outputs,** and simple assignments.
- Verilog Language (Basics)
  - Declaring **wires** and **assign** statements.
  - Building simple **combinational circuits** using logic operator
  - Using **bit slicing** and **concatenation** to manipulate vectors.
  - Understanding how to create reusable and modular code structures.
- Combinational Logic
  - Implementing more complex logic functions: **muxes, decoders, encoders, priority logic, and comparators**.
  - Using **if-else** and **case** statements for conditional logic.
  - Structuring logic to be **synthesizable** and **deterministic**.

2.2.3 An example problem statement

To deepen our understanding of sequential circuits and module instantiation, we solved the "Module shift8" problem on HDLBits. The objective was to implement a shift register with 8-bit wide data, chained through three D flip-flop modules, and use a 4-to-1 multiplexer to select the delayed output based on a 2-bit selector.

Problem Description: You are given a module `my_dff8` that implements 8 D flip-flops. Your task is to instantiate three such modules and connect them in series to create a shift register of depth 3. Additionally, you must use a 4-to-1 multiplexer that selects which stage's output to forward based on a 2-bit input `sel[1:0]`.

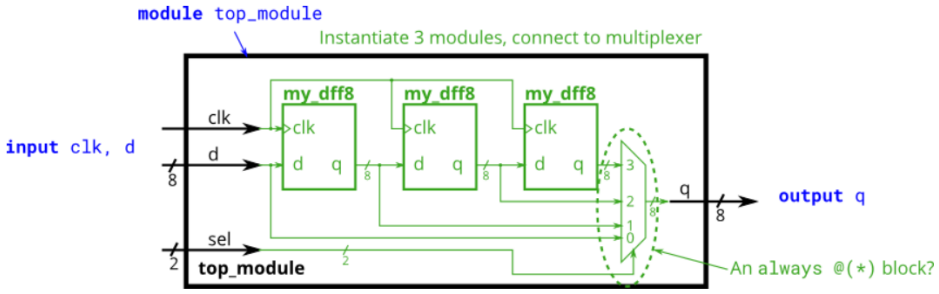

Figure 2: HDLBits Problem Statement Circuit for Module shift8

Verilog Solution:

```
module top_module (
    input clk,
    input [7:0] d,
    input [1:0] sel,
    output reg [7:0] q
);
    wire [7:0] q1, q2, q3;

    my_dff8 d1(clk, d, q1);
    my_dff8 d2(clk, q1, q2);
    my_dff8 d3(clk, q2, q3);

    always @(*) begin
        case (sel)
            2'd0: q = d;
            2'd1: q = q1;
            2'd2: q = q2;
            2'd3: q = q3;
        endcase
    end
endmodule
```

Explanation:

- `my_dff8` is instantiated three times and chained to form a 3-stage shift register.
- A combinational `always @(*)` block implements the 4-to-1 multiplexer using a `case` statement.
- The `sel` input determines how many clock cycles to delay the input `d` before outputting it as `q`.

Simulation: The HDLBits platform automatically validates the design using a set of predefined test cases. Upon simulation, all test cases passed successfully, confirming the correctness of the implementation.

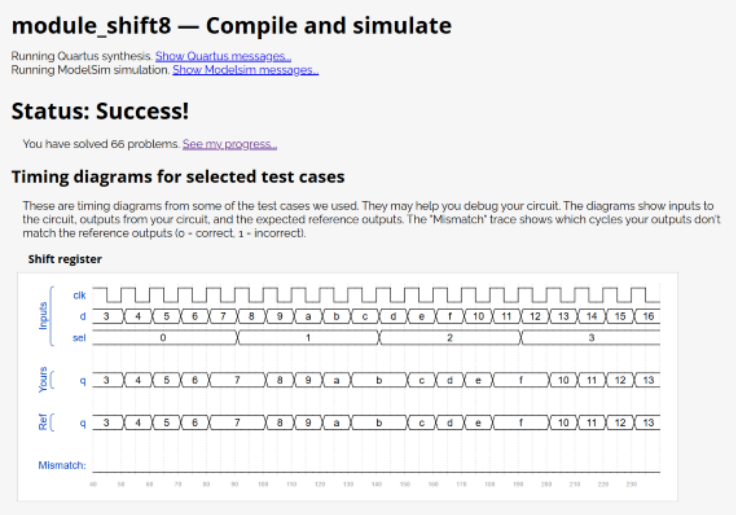

Figure 3: Simulation Result for shift8: All Test Cases Passed

3 Software setup and platforms used

3.1 Icarus Verilog

Icarus Verilog is used primarily for early-stage Verilog simulation and testing. It is an open-source Verilog simulation and synthesis tool that supports a wide range of Verilog features.

1. Install Icarus Verilog using package managers (e.g., ‘sudo apt install iverilog‘ on Ubuntu).
2. Example used: a simple XOR gate module. Code file `xor.v`:

```
module xor\_gate(
input a, b,
output y
);
assign y = a ^ b;
endmodule
```

3. A corresponding testbench was written and simulated using:

```
iverilog -o xor\_tb xor.v xor\_tb.v
vvp xor\_tb
```

4. Use ‘gtkwave‘ to view the simulation waveforms.

3.2 Xilinx Vivado

Vivado is used for synthesis, implementation, and bitstream generation for the Cora Z7 FPGA.

1. Create a Vivado project and add Verilog source files.
2. Use the Block Design feature to graphically integrate IP blocks, including the Zynq Processing System, AXI interfaces, and custom logic.
3. Elaborate the design to analyze the structure and check for correctness.
4. Run synthesis to translate the high-level design into gate-level logic.
5. After synthesis, run implementation to map and place the logic onto FPGA resources.
6. Generate the bitstream file to configure the FPGA.
7. Use Vivado’s Hardware Manager to program the FPGA and verify the configuration.

3.3 Xilinx Vitis

Vitis is used for software development and deployment on the ARM processor in the Zynq SoC of the Cora Z7.

1. Create a new platform project by importing the hardware description (.xsa) generated from Vivado.
2. Develop the software application (e.g., "Hello World") in C.
3. Configure build options, including processor selection and optimization levels.
4. Build the application and create bootable files (.elf, .bin).
5. Use the Vitis terminal or serial monitor to interact with the application over UART.

3.4 Integration and Deployment

Vivado and Vitis are tightly integrated to support hardware-software co-design.

- After completing the hardware design in Vivado and exporting the hardware (.xsa file), the project is imported into Vitis for software development.
- The bitstream and compiled software application are bundled into a single boot image.
- The boot image is loaded onto the Cora Z7 board either via JTAG or SD card.
- During execution, image data is sent to the FPGA through UART. The FPGA performs Sobel edge detection and sends back the processed image data.
- A software called Tera Term running on the PC is used to send and receive the image data in real time.

4 Week Wise Progress

Week Wise Progress

The initial phase of the project (Week 1) focused on building a strong foundation in basic electronics, which is essential for understanding digital systems. This included studying the behavior of key components such as resistors, capacitors, diodes, and transistors, along with fundamental laws like Ohm's and Kirchhoff's Laws. This knowledge provided the necessary background for interpreting and designing digital logic circuits.

During the second week, the focus shifted to digital logic and Verilog programming. This involved learning how to represent numbers in binary, octal, and hexadecimal, as well as understanding Boolean algebra, Karnaugh maps (K-maps), and the principles of combinational and sequential logic circuits. Practical sessions included writing and simulating basic Verilog modules to reinforce these concepts.

The third week involved the setup and familiarization with industry-standard tools—Vivado and Vitis by Xilinx. These tools were installed and tested, enabling the creation, synthesis, and simulation of digital designs. Hands-on exploration of their interfaces prepared the groundwork for working with actual FPGA hardware.

In the fourth week, the knowledge from previous weeks was applied to program and test logic designs on a physical FPGA board. Basic Verilog designs were implemented, pin configurations were mapped, and outputs were verified using onboard LEDs. This marked the transition from simulation to real-world testing and laid the foundation for more advanced hardware design tasks.

4.1 Week 1: Introduction to Electronics

- **Goals:** Understand basic electronic components and circuit behavior.
- **Tasks Completed:**
  - Watched lecture videos on Introduction to Electronics
- **Challenges:** Identifying component behavior in complex circuits. .

Basic electronics and digital logic involve understanding how information is represented and processed in digital systems. Number representation forms the foundation, using binary and other bases to encode data. Karnaugh Maps (K-Maps) are used to simplify Boolean expressions, helping design efficient logical circuits. Digital circuits are built from basic gates like AND, OR, and NOT, which are then combined to form more complex **combinational circuits**, where the output depends only on current inputs. In contrast, **sequential circuits** include memory elements like flip-flops, making outputs dependent on both current inputs and past states. These are essential for building counters, registers, and control units. Finite State Machines (FSMs) model complex systems with defined states and transitions, enabling structured design of responsive and predictable digital behavior.

4.2 Week 2: Introduction and Download of Verilog

- **Goals:** Learn basics of HDL and install Verilog tools.
- **Tasks Completed:**
  - Installed Icarus Verilog and GTKWave.
  - Wrote basic Verilog modules (AND, OR, NOT gates).
- **Challenges:** Understanding simulation vs. synthesis.

Verilog is a hardware description language used to model electronic systems. It allows designers to describe the structure and behavior of digital circuits. Installing simulation tools like Icarus Verilog helps in testing modules before deploying them on hardware.

4.3 Week 3: Downloading and Installing Vivado/Vitis and Their Operation

- **Goals:** Set up development tools by Xilinx and explore the interface.
- **Tasks Completed:**
  - Installed Vivado and Vitis IDE.
  - Ran sample projects and explored block design tools.
- **Challenges:** Software installation size and license issues.

Vivado is used for synthesizing and implementing digital designs on Xilinx FPGAs, while Vitis is used for embedded software development. These tools offer graphical and scripting interfaces to design, analyze, and simulate systems efficiently.

4.4 Week 4: Working with FPGA

- **Goals:** 1. Software setup on Vivado, programming a simple logic circuit on an FPGA board.
  2. programming the inbuilt processor using Vitis.
- **Tasks Completed:**
  - Uploaded Verilog design to the FPGA board.
  - Tested output using onboard LEDs.
  - Uploaded a code on processor to invert a grayscale image.(without fpga).
- **Challenges:** Debugging pin mapping errors.

Field Programmable Gate Arrays (FPGAs) are integrated circuits that can be configured after manufacturing. Using tools like Vivado, we can program FPGAs to perform custom digital logic operations. This week focused on writing a basic Verilog module and implementing it on the FPGA for real-world verification.

5 Summary

Project Overview

This project aims to implement real-time edge detection using the Sobel filter on an FPGA, leveraging hardware parallelism for high-speed image processing. The system processes grayscale images from a PC through a custom Verilog implementation, returning edge-detected results via UART interface visualized with TeraTerm.

Key Achievements

- Established fundamental electronics knowledge including component behavior (resistors, capacitors, diodes, transistors) and circuit laws (Ohm's, Kirchhoff's)
- Learned digital logic concepts: number systems (binary, octal, hexadecimal), Boolean algebra, and Karnaugh Maps
- Developed practical Verilog skills by implementing and simulating basic logic gates (AND, OR, NOT)
- Successfully installed and configured the complete Xilinx toolchain (Vivado and Vitis)

Technical Implementation

1. **Electronics Foundation:**
    - Analyzed basic electronic components and their behavior
2. **Digital Logic Design:**
    - Implemented combinational circuits using logic gates
3. **Verilog Development:**
    - Created synthesizable modules for basic logic operations
    - Set up simulation environment using Icarus Verilog and GTKWave
    - Resolved simulation vs. synthesis implementation challenges
4. **FPGA Toolchain:**
    - Configured Vivado for design synthesis and implementation
    - Established Vitis environment for processor programming
    - Overcame installation and licensing hurdles

Development Tools

- Icarus Verilog for initial simulation and testing
- Xilinx Vivado for FPGA synthesis and implementation
- Xilinx Vitis for ARM processor software development

Conclusion

The project successfully aims to demonstrate FPGA's superiority in real-time image processing through parallel hardware implementation.It also helps in gaining valuable experience in digital system development, establishing a foundation for more complex computer vision applications.

List of Figures