EC3073, Team D10
Milestone 3 Report

Sam French - *32488750* | Evan Polyzos - *33113939*
Miles Nathan - *33155461* | Kabeeshan Gunasekaran - *31309496*

1 Introduction

The objective of this technical report is to document the design, implementation, and evaluation of performance improvements achieved by transitioning from a single-processor to a multiprocessor embedded vision system on the DE10-Lite FPGA platform. The system is designed to acquire image data from an ESP32-CAM over SPI, apply visual filters, and render the output in real time via VGA. This project is motivated by the need to explore parallelism and interprocessor communication within a resource-constrained embedded environment, where traditional single-core designs often struggle to meet the computational and timing demands of real-time image processing and I/O operations.

To address these challenges, the system employs shared SDRAM, hardware mutexes, and PIO-based interrupts to enable workload distribution across processors while ensuring data consistency and synchronisation. It improves upon an existing single-processor design (described in the following section) by incorporating multiple Nios II processors, custom instruction extensions, and enhancements to the image acquisition pipeline, including RGB support and a more efficient SPI data transfer method.

This report outlines the system architecture, processor roles, implementation approach, and the evaluation of performance and feature improvements over the baseline.

The final system must satisfy the following high-level design requirements:

- Achieve at least **50% performance improvement** over the baseline 2 FPS
- Use a **dedicated processor** for all SPI communication
- Maintain all features of the previous baseline system
- Justify all additional features with **clear analysis of their impact on system performance**

2 Base System Overview

The base system is built around the DE10-Lite FPGA platform, featuring a single Nios II processor to manage the acquisition, processing, and display of grayscale images from an ESP32-CAM module. This processor handles all functionality, including SPI communication with the ESP32-CAM and ADXL345 gyroscope, image processing, storage and output to a display through VGA. A general overview of the different components and the connections between can be seen in Figure 1 below.

At startup, the system requests and receives image data from the ESP32-CAM, storing frame data in SDRAM. The frame data is sent from SDRAM to the pixel buffer which is connected to the VGA controller, displaying the image. It uses interrupts to change between single/quad mode and also the image filter applied, determining which image data is displayed. This architecture enables clear task separation but introduces some latency due to the lack of parallelisation.

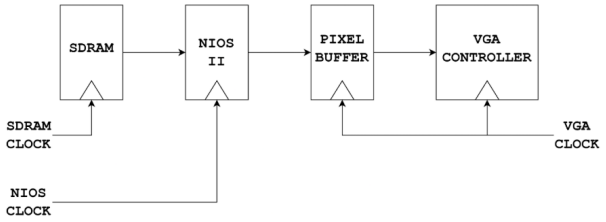

Figure 1. Hardware Flow Block Diagram (Exc, Keys, Switches and Gyro based interrupts).

The system displays images on a VGA monitor with filter switching and orientation control, supporting the following user interface features: a double tap detected by the gyroscope toggles between single and quad display modes, with each quadrant in quad mode allowing independent filters; normal, flip, blur, or edge detection which is selectable via switches. Another key cycles through filters, and gyroscope input enables dynamic rotation of images in quad mode.

However, performance metrics show limitations, with typical frame rates as displayed in Table 1, and noticeable delays during mode transitions. These baseline figures serve as a reference for evaluating further system optimisations. Overall, while the base system is functional and modular, it lacks the efficiency and responsiveness required for more demanding real-time applications, such as an RGB display.

Table 1
Benchmarking of Original System Performance (Frame rate)

Task (Grayscale)	FPS
Display an unaltered video frame in single-image mode	1.555
Display a flipped video frame in single-image mode	1.515
Display a blurred video frame in single-image mode	0.909
Display an edge-detected video frame in single-image mode	0.843
Display 4 of the same video frames at once in quad-image mode	3.608
Display 4 unique video frames at once in quad-image mode	1.881

3 Multiprocessing Implementation

To support multiprocessing, the system was restructured so that the overall workload could be split into separate, parallel tasks, with each task handled by its own dedicated processor. This shift was motivated by the need for better performance and responsiveness, especially given the real-time demands of image

processing, sensor input handling, and display output. By dividing responsibilities across processors, the system makes much better use of the available computing resources and allows the system to handle data acquisition, processing, and communication all at the same time.

The system uses three processors, each assigned to a specific role. The first processor, called the master processor, is in charge of most of the communication between processors. It also handles image manipulation tasks, keeps track of which display mode is active, and applies any filters that need to be shown on the screen. Essentially, it acts as the central controller for the whole system.

The second processor, known as the SPI processor, is focused entirely on SPI communication as per system requirements. It communicates with the ESP32-CAM to receive image data and also interacts with the onboard gyroscope/accelerometer to get orientation data. By giving this processor its own dedicated task, we make sure that SPI communication runs smoothly without being delayed by other parts of the system.

The third processor is the display processor, which takes care of sending frame data to the pixel buffer so that it can be shown on the VGA display. Since this processor only focuses on moving and displaying image data, it helps keep the output consistent and responsive, even if other parts of the system are busy.

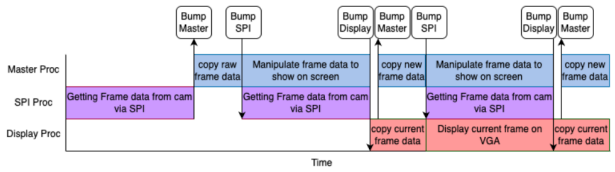

Figure 2. Timing Diagram For Multiprocessor System.

As shown in the flowchart above, the process begins when the camera finishes capturing a frame and marks it as ready. The SPI processor reads the frame data over SPI and writes it into a shared section of SDRAM. Once the data is written, it sends a message to the Master processor to let it know that the frame is available. After that, the SPI processor waits for the Master to signal that it's done with the frame before grabbing the next one.

When the Master processor receives the "frame ready" message, the first thing it does is copy the frame data from SDRAM into its own working buffer or memory space. This ensures that the Master has a consistent snapshot of the frame to work with, and frees up the shared SDRAM for the next incoming frame. Right after copying the frame, the Master sends a message back to the SPI processor, telling it that it's done with the current frame and is ready for the next one.

Only after that does the Master start processing the image. Based on the current system mode, it performs any required image manipulations. For instance, in quad mode, the Master duplicates the image three times to create a 2x2 layout and calculates any filters that need to be applied to each section. These operations depend on the user-selected settings or sensor input from the accelerometer.

Before writing the newly processed frame back into SDRAM, the Master checks with the Display processor to make sure it has finished displaying the previous frame. This avoids overwriting data that's still in use. Once the Display processor signals that it's ready, the Master writes the new image into SDRAM and sends two messages: one to the Display processor to tell it to start displaying the new frame, and one to the SPI processor to let it know it can begin reading the next frame from the camera.

The Display processor then copies the new frame from SDRAM into the pixel buffer, which is connected to the VGA output. The image is shown on the screen, and once the display is complete, the Display processor sends a message back to the Master to confirm it's done. This lets the system safely move on to the next frame without conflicts or glitches.

This entire flow is carefully managed using message passing and synchronization checks, which act as concurrency control methods. By relying on processor-to-processor messages rather than direct shared access, the system avoids race conditions and data hazards. Each processor only acts when it's safe to do so, based on status flags or explicit confirmations, ensuring that shared resources like SDRAM are accessed in a controlled and predictable way. The structure of this shared SDRAM can be seen in Figure 3 below, alongside the overall performance metrics in Table 2. It can bee seen that the system achieves the minimum 3 FPS as required by the brief.

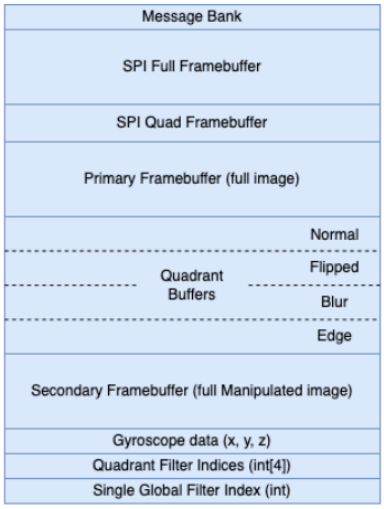

Figure 3. Shared SDRAM Structure.

4 General System Improvements

In addition to implementing multiprocessing, several key system-level enhancements were introduced to improve performance and functionality beyond the capabilities of the base system. These improvements include efficient data packing for faster communication, support for RGB image data, and hardware-accelerated custom instructions for computationally intensive image processing tasks.

Table 2
Performance (FPS) after multiprocessor implementation

Task	FPS
Display an unaltered video frame in single-image mode	3.063
Display a flipped video frame in single-image mode	2.692
Display a blurred video frame in single-image mode	1.834
Display an edge-detected video frame in single-image mode	1.576
Display 4 of the same video frames at once in quad-image mode	7.856
Display 4 unique video frames at once in quad-image mode	2.508

4.1 Packed Data

To optimise communication bandwidth between the ESP32-CAM and the SPI processor, the system was configured to utilise the camera's built-in packed data transmission mode, enabled by setting Bit 2 in the configuration byte sent via SPI. This feature allowed the camera to transmit grayscale pixels two per byte instead of one, effectively halving the volume of data transmitted over the SPI interface

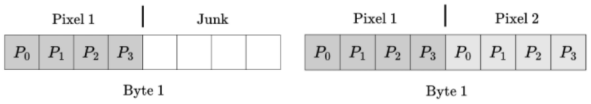

Figure 4. Left - Unpacked Data | Right - Packed Data

While this mode offers significant performance benefits—most notably, a substantial reduction in SPI transfer time due to the halved data size—these gains were not fully realised in practice. This is because our system was designed to immediately unpack the received data within the SPI processor before writing it to shared SDRAM.

This unpacking step doubled the memory requirement and introduced additional computational overhead during data transfer. However, this approach simplified the integration with existing components by maintaining compatibility with the one-byte-per-pixel format expected by the master and display processors. Additionally, it provided a more straightforward path for future upgrades, such as support for RGB image formats, which inherently demand more memory.

The decision thus prioritised code simplicity, maintainability, and extensibility, accepting a trade-off in memory efficiency in exchange for system-wide consistency. Overall, it still led to an increase in Frame Rate as is evident from the discussion above, the results can be found in Table 3.

4.2 RGB Image

To support RGB image processing, the system was configured to request RGB frames from the ESP32-CAM by enabling Bit 0 in the SPI configuration byte. This caused the camera to transmit each pixel in RGB format using the packed 4-4-4 data mode.

Each RGB pixel consists of 12 bits (4 bits each for Red, Green, and Blue), which were unpacked in the SPI processor and stored in the shared memory buffer in a triple-byte format: one byte for each color channel (R, G, and B). This was done to align with existing VGA display and processing pipelines that expected one byte per channel.

Instead of tightly packing RGB data, which would complicate processing, the decision was made to store the RGB pixels in three parallel memory arrays—one each for R, G, and B. This increased the memory footprint by a factor of three compared to grayscale but simplified the codebase. Image processing functions could now access pixel color components directly and independently.

The primary trade-off of supporting RGB was a decrease in performance due to increased memory access and SPI transfer size. RGB frames required more bandwidth to transfer and more space to store, leading to longer read/write times and higher SDRAM usage. Nonetheless, the implementation prioritized clarity and maintainability.

Visual evidence of RGB support can be seen in the edge detection output (see Figure 5), where detected edges are overlaid in blue on the original RGB image, providing better visual contrast than grayscale-only output. However, this enhancement resulted in a decrease in framerate, as the volume of image data transmitted over SPI and accessed from shared memory has effectively tripled compared to the grayscale implementation, this is evident in Table 4.

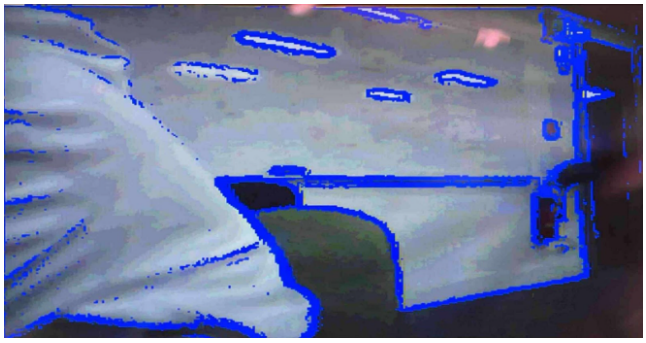

Figure 5. RGB Edge Detection

4.3 Custom Instructions (Inc. Convolution Improvement)

To reduce processing latency in computationally intensive tasks, three custom hardware instructions were developed, each implemented as a separate Verilog module and connected to the Nios II processor through Platform Designer's custom instruction interface. The following custom instructions were included in the final system:

- `custom_rgb_to_gray` – Grayscale conversion
- `custom_blur` – 3×3 convolution for Gaussian blur
- `custom_sobel` – 3×3 convolution for Sobel edge detection

By implementing these operations as combinatorial custom instructions, they could be executed directly in hardware in a single-cycle evaluation. This approach bypassed the overhead associated with equivalent software routines written in C, which required many more cycles per pixel due to nested loops and arithmetic. These hardware instructions significantly accelerated the image

processing pipeline, enabling both the edge detection and blur modes to run at the same frame rate as the flipped image mode.

4.3.1 Grayscale Conversion

This instruction receives a 12-bit RGB input in 4-4-4 format and returns an approximate 4-bit grayscale value, calculated using a weighted average method.

$$Pixel_i = 0.299 \cdot R_i + 0.587 \cdot G_i + 0.114 \cdot B_i$$

4.3.2 Custom Blur and Custom Sobel (Convolution Improvements) Both of these instructions accept two 32-bit inputs:

- `dataa` – Contains 8 grayscale pixels, each packed into 4-bit segments
- `datab` – Contains the final 9th grayscale pixel

Together, these inputs form a 3×3 pixel window needed for convolution. The convolution kernels (Gaussian and Sobel) are hard-coded inside the Verilog modules, as they remain fixed throughout all computations. The final result is clamped to a 4-bit value and returned in the lower byte of the 32-bit output.

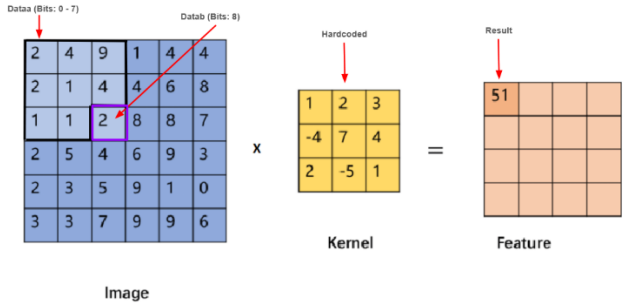

Figure 6. Convolution Custom Instruction - Verilog Inputs and Outputs.

4.4 Qauntiative Metrics of Performance

Table 3
Performance with Multiprocessor Baseline vs Packed

Task	Packed	Δ vs Baseline
Display unaltered frame (single-image)	3.057	-0.006
Display flipped frame (single-image)	2.690	-.0.002
Display blurred frame (single-image)	1.881	+0.047
Display edge-detected frame (single-image)	1.570	-0.006
Display 4 same frames (quad-image)	7.828	-0.028
Display 4 unique frames (quad-image)	2.490	-0.018

Table 4
Isolated Δ FPS Improvements: RGB vs Packed

Task	RGB	Δ vs Packed
Display unaltered frame (single-image)	2.650	-0.407
Display flipped frame (single-image)	2.293	-0.397
Display blurred frame (single-image)	0.810	-1.071
Display edge-detected frame (single-image)	1.455	-0.115
Display 4 same frames (quad-image)	3.625	-4.203
Display 4 unique frames (quad-image)	1.523	-0.967

Table 5
Isolated Δ FPS Improvements: Custom Instruction vs RGB

Task	Custom Instr.	Δ vs RGB
Display unaltered frame (single-image)	2.650	0
Display flipped frame (single-image)	2.619	+0.326
Display blurred frame (single-image)	2.618	+1.808
Display edge-detected frame (single-image)	2.617	+1.162
Display 4 same frames (quad-image)	3.763	+0.138
Display 4 unique frames (quad-image)	3.715	+2.192

5 Conclusions and Future Work

The system underwent a comprehensive transformation through the integration of multiprocessing and targeted performance optimisations. Distributing functionality across multiple processors significantly enhanced system responsiveness and reliability. One processor was dedicated to managing sensor communication via SPI, while another handled image filtering and rendering, enabling concurrent data acquisition and processing. A third processor was introduced to exclusively manage VGA video output, decoupling rendering from computation and leading to smoother image output and improved frame rates.

Further improvements were made such as transition from grayscale to 12 bit color data providing a richer visual output. Whilst this posed a deficit in performance, coupling this with packed data from SPI enabled this to be a viable option that provided a better user experience without rendering the system unusable. In addition to this, the optimisation of the convolution process using custom instructions meant that the performance of those filters was on par with the others. Lastly, Advanced techniques for handling SDRAM communication and frame buffer access played a key role in maintaining correct and stable operation whilst pushing the system to its limits. The quantitative result of these changes is seen in Table 6.

Despite the improved performance some limitations remain, such as the overhead introduced by the synchronization mechanisms and complexity of maintaining data consistency across processors. Future improvements include the abstraction of different image filtering techniques to separate processes to again reduce individual load per processor.

Table 6
Frame rates (FPS) for final System.

Task (Custom Instruction)	FPS
Display an unaltered video frame in single-image mode	2.624
Display a flipped video frame in single-image mode	2.619
Display a blurred video frame in single-image mode	2.618
Display an edge-detected video frame in single-image mode	2.617
Display 4 of the same video frames at once in quad-image mode	3.763
Display 4 unique video frames at once in quad-image mode	3.715