Motion Sentry

Final Report

Ezekiel Daye
Department of Electrical Engineering and Computer Science
Department of Physics
Massachusetts Institute of Technology
Cambridge Massachusetts
egdaye@mit.edu

Ryan Hourican
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge Massachusetts
ryanh24@mit.edu

Makar Kuznietsov
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge Massachusetts
makark@mit.edu

***Abstract*—Motion Sentry: A Real-Time 3D Object Tracking System Using FPGA**

The Motion Sentry is an advanced tracking system that is made up of camera-based motion detection, servo motor control, and laser sensing to allow for real-time 3D object tracking. The full system utilizes an FPGA to run a pipelined background subtraction algorithm to identify motion in the x-y plane. With this data, the FPGA relays the position to a servo motor which translates the position into motion using pulse-width modulation (PWM). The servo adjusts the orientation of a LiDAR sensor to capture z-axis depth information about the detected object and sends that back to the FPGA through a UART interface.

The system will take the depth information and display the results via 8-bit numerical images that represent key parameters such as object distance or velocity, providing clear and concise monitoring of the object and the systems performance.

The Motion Sentry effectively uses the FPGAs ability to handle high-bandwidth, computationally intensive tasks such as video processing and real-time pixel classification to create a tracking system. By combining parallel processing, external DRAM for memory management, and integrated control of servo and LiDAR systems, the Motion Sentry demonstrates a sophisticated design optimized for digital systems and advanced motion tracking applications.

I. Introduction

Field-Programmable Gate Arrays (FPGAs) have long been used with advanced sensor technology to track objects in a variety of scenarios. Due to the FPGAs parallel processing capabilities and reconfigurability, they are ideal for applications requiring high-speed data processing and low latency. When combined with Light Detection and Ranging (LiDAR) sensors, to provide precise depth information, the combined system of FPGA, servo, and LiDAR is capable of achieving a comprehensive understanding of a multitude of environments.

In 2012, a study demonstrated the implementation of real-time background subtraction on an FPGA using the Horprasert model, achieving effective motion detection for video surveillance applications. The authors noted that "the proposed hardware-friendly model achieved robust segmentation and shadow detection in real time, with a low hardware cost" [1]. This work highlights the capability of FPGAs to efficiently execute a complex mathematical algorithm, such as background subtraction, while maintaining real-time performance.

Additionally, the defense sector has extensively adopted FPGA technology for object tracking and detection. Since an FPGA can handle large data streams and perform complex algorithms in real-time, they are ideal tools for usage in radar and signal processing applications. This allows for precise target detection, tracking, and discrimination capabilities in aerospace surveillance and defense systems [2].

Building and borrowing from these ideas, the **Motion Sentry** project aims to develop a small-scale real-time 3D object-tracking system. By using an FPGA-based background subtraction for x-y plane motion detection and integrating servo-controlled LiDAR for z-axis depth information, this system attempts to achieve efficient and accurate tracking in most environments.

II. Parts of System

A. Background Subtraction (Ezekiel)

Background subtraction aims to identify moving objects in a static scene by comparing each pixel in incoming frames to a reference background model. The process of creating this requires mathematical modeling, efficient memory management, and pipelining to meet the FPGAs restraints in timing and resources. To create a background subtraction algorithm for the FPGA is a complicated task and requires meticulous design and coordination between modules. The algorithm used in this implementation is adapted from Horprasert et al.'s work, which introduces a robust approach for background modeling and shadow detection by incorporating metrics like brightness distortion and chromaticity distortion [3]. This algorithm is well-suited

for FPGA implementation because can be parallelized in hardware, making it efficient for real-time applications.

To classify the pixels as foreground, background, shadow, or highlight each pixel is compared against a statistical background model, and a set of two other metrics are calculated using the data from the background model – brightness distortion (α) and chromaticity distortion (CD). With all of this information, the system can tell which objects are moving in the static frame. This stage is pivotal for achieving accurate foreground extraction to track objects.

1. **Pixel Classification Logic**:
   The **pixel classification module** uses the calculations of brightness (α) and chromaticity (CD) distortions for each pixel using the following equations:

$$\alpha = \frac{I}{E} \quad (1)$$

$$CD = \sqrt{(I_R - \alpha E_R)^2 + (I_G - \alpha E_G)^2 + (I_B - \alpha E_B)^2} \quad (2)$$

   - Thresholds are applied to α and CD to classify pixels:
     - If α and CD fall within certain thresholds, the pixel is classified as part of the static background.
     - If α is low but CD is within the range, the pixel is categorized as a shadow.
     - Pixels with high α may be classified as highlights.
     - High α and CD indicate a foreground object.
   - This module outputs the classification result for each pixel, which can be used to generate a foreground mask.

For example:

```
if (alpha < THRESH_ALPHA && cd < THRESH_CD)
begin
    classification <= BACKGROUND;
end
```

2. **Statistical Background Modeling** :
   - Mean (E) and SD (σ) are computed for each pixel channel (Red, Green, Blue) over multiple frames. These metrics are calculated before tracking can begin and are stored in the DRAM.
3. **Newton-Raphson Square Root Module**:
   - The square root operation, critical for computing σ and CD, uses the Newton-Raphson method for iterative approximation.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \; f(x) - x^2 - a \quad (3)$$

4. **DRAM Storage and Access**:
   - All intermediate values, such as sums ($\sum$ I), squared sums ($\sum I^2$), and statistical results (mean, variance), are stored in DRAM for every pixel. These values are accessed and updated for each new frame.
5. **Pipeline and Synchronization**:
   - The system is pipelined to ensure that accumulation, model update, classification, and output occur concurrently. Proper synchronization between stages is critical to avoid timing conflicts and data overwrites.

Memory Considerations for FPGA-Based Background Subtraction

Implementing a background subtraction algorithm for 720p video resolution on an FPGA requires extensive memory resources to store and manage pixel-level data efficiently. Each frame at 720p resolution consists of 921,600 pixels, and the algorithm demands both intermediate and final values to be stored for every pixel. The memory requirements stem from the need to maintain running sums, sums of squares, and distortion values for chromaticity and brightness, which are critical for real-time foreground-background classification.

Background Model Memory Analysis

The memory requirements for each pixel in the background model include:

1. **Running Values (Stored Temporarily for Ongoing Calculations):**
   - **Running Sums (`sum_R`, `sum_G`, `sum_B`):**
     - Each channel (R, G, B) uses 48 bits per pixel.
     - Total memory for running sums: 48 bits/pixel/channel × 3 channels × 921,600 pixels ≈ 132.7 Mb.
   - **Sums of Squares (`sum_sq_R`, `sum_sq_G`, `sum_sq_B`):**
     - Each channel (R, G, B) uses 64 bits per pixel.
     - Total memory for sums of squares: 64 bits/pixel/channel × 3 channels × 921,600 pixels ≈ 176.9 Mb.
   - **Distortion Metrics (`sum_CD`, `sum_BD`, `sum_sq_CD`, `sum_sq_BD`):**
     - Distortion sums (`sum_CD`, `sum_BD`) use 48 bits each, and sums of squares of distortions

(`sum_sq_CD`, `sum_sq_BD`) use 64 bits each.
        - Total memory for distortion metrics:
          (48 bits + 64 bits)/pixel × 2 metrics × 921,600 pixels ≈ 103.7 Mb.
2. **Total Memory for Running Values:**
   132.7 Mb + 176.9 Mb + 103.7 Mb ≈ 413.3 Mb.
3. **Final Stored Values (Mean, Standard Deviation, Square Root Outputs):**
    - **Means (`mean_R`, `mean_G`, `mean_B`):**
        - Each channel (R, G, B) uses 32 bits per pixel.
        - Total memory for means:
          32 bits/pixel/channel × 3 channels × 921,600 pixels ≈ 88.4 Mb.
    - **Standard Deviations (`SD_alpha`, `SD_CD`):**
        - Total memory for square root outputs:
          32 bits × 2 metrics × 921,600 pixels ≈ 7.4 Mb.
4. **Total Memory for Final Stored Values:**
   88.4 Mb + 7.4 Mb ≈ 95.8 Mb.

Summary

- **Memory for Running Values:** ≈ 413.3 Mb
- **Memory for Final Stored Values:** ≈ 95.8 Mb
- **Total Memory Usage:** ≈ 509.1 Mb

DRAM Integration

The FPGA interfaces with DDR3 DRAM using a memory controller to handle the large data volume. The traffic generator, stacker, and unstacker modules manage read and write operations to and from DRAM. For each frame, these modules coordinate the retrieval of pre-computed background model values (e.g., mean and standard deviation) and store updated values for subsequent frames.

Module Latency Analysis

Timing and pipelining are critical for operation of the background subtraction algorithm on FPGA hardware. This section summarizes the latency of key modules involved in the design.
1. **Newton-Raphson Square Root (Sqrt)**

The `Sqrt` module utilizes the Newton-Raphson method to calculate square roots for fixed-point values in Q16.16 format. The iterative algorithm requires 16 iterations to converge for this precision level. In simulation division happens in one cycle so the latency is only 18 cycles but given division takes ~ 16 cycles the total latency is:

16 *(Initialize (1) + Division (16) + Finalize (1)) = 288 Cycles

2. Chromaticity Distortion

The `ChromaticityDistortion` module calculates chromaticity distortion (CD) by determining deltas for the red, green, and blue channels, squaring them, summing them, and computing the square root. The operations include:

- **Delta and Delta-Squared Calculation:** These involve additions, multiplications, and divisions for each channel. The division latency for each channel contributes **16 clock cycles**, totaling **48 clock cycles**.
- **Summing the Deltas:** Adding three delta-squared values contributes to 1 **clock cycle**.
- **Total Latency:** 48 (channel-wise delta calculation) +1 (summing) +288 (square root)= 337 clock

3. Brightness Distortion

The `BrightnessDistortion` module calculates brightness distortion (α) by normalizing the input pixel values and performing a division to determine the brightness distortion ratio. Key operations include:

- **Numerator and Denominator Calculation:** These involve multiplications and divisions for each channel. The division latency for each channel is estimated to be **16 clock cycles**, resulting in **48 clock cycles** for all three channels.
- **Final Division:** The calculation of α=N/D involves one more division, taking **16 clock cycles**.
- **Total Latency:** 48 (channel-wise division) +16 (final division)=64 clock cycles

4. Pixel Classification

The `PixelClassification` module determines whether a pixel belongs to the background, foreground, shadow, or highlight. The operations include:

- Buffering input values (1 cycle)
- Threshold calculation for α and CD (2 cycle)
- Classification logic based on thresholds (1 cycle)
- Output classification and validation (1 cycle)

The total latency is: 5 cycles

5. Background Model Latency

Since the background model calculations are performed prior to real-time operations, its latency does not directly impact the system's performance during tracking. The module can take as many clock cycles as needed to complete its operations without imposing strict timing constraints. This design decision allows for a more comprehensive and resource-efficient calculation of the background statistics, leveraging the available memory and computational bandwidth without compromising the real-time aspects of the overall system.

B. Servo-Control (Ryan)

The servo control subsystem is responsible for the dynamic orienting of the LiDAR sensor to track the detected object in real-time. This involves processing the center of mass (CoM) coordinates from the background subtraction module, converting these coordinates into precise PWM signals, and ensuring accurate servo actuation to keep the LiDAR aligned with the detected object.

1. CoM to PWM Mapping logic

The CoM to PWM mapping module serves as the bridge between the 2D pixel coordinates of the CoM in the video frame and the corresponding angular positions required to reorient the servo motors. This process ensures the LiDAR remains focused on the moving object while accounting for the camera's field of view (FoV) and the mechanical constraints of the servos.

The mapping process translates the pixel range of the CoM, x: 0 to 1280 and y: 0 to 720, to the calibrated servo angles ranges $[min_x, max_x]$ and $[min_y, max_y]$. The linear transformation ensures predictable servo motion across the entire camera FoV.

The Linear mapping equations are defined as:

$$x_{pwm} = CoM_x \frac{max_x - min_x}{1280} + min_x \quad (4)$$

$$y_{pwm} = CoM_y \frac{max_y - min_y}{720} + min_y \quad (5)$$

2. PWM Signal Generation

The servo control module generates PWM signals to drive the motors based on the computed x_{pwm} and y_{pwm} values. A high-resolution counter ensures precise timing for PWM signals.

Key specs:

- **Frequency:** 50Hz (20 ms period)
- **Duty Cycle Range:** 1 ms to 2 ms pulse width (5-10% duty cycle)
- **FPGA Clock frequency:** 200 MHz
    - Using 100 MHz to create this frequency for the camera limits our further use of it
- **Angular precision:** Achieves accuracy within 0.001°, leveraging high clock resolution for duty cycle adjustments.

Pseudocode:

```
always_ff @(posedge clk_camera) begin
    if (counter+1 == (20_000_000)) begin
        counter <= 0; // Reset after 20ms
    end else begin
        counter <= counter + 1;
    end
end
assign signal_out = counter < pwm_signal;
```

Calibration Process:

The calibration process ensures that the servo motors accurately align with the camera's field of view and track moving objects effectively. This was achieved through a systematic approach:

1. **Midpoint Alignment**: Identifying the PWM values that positioned the servos to point at the exact center of the camera's frame. This served as the baseline for calibration.
2. **Boundary Identification**: The PWM values corresponding to the topmost, bottommost, leftmost, and rightmost points of the camera's field of view were determined. These boundary values were carefully selected to ensure they were symmetrically spaced from the midpoint values.
3. **Validation**: Multiple iterative tests were conducted to verify that the mapped PWM values consistently directed the servos to track moving objects across the entire range of the camera's field of view. Adjustments were made to fine-tune the calibration and ensure robust tracking performance.

3. Synchronization with LiDAR

Synchronization between the servo control subsystem and the LiDAR sensor is crucial for maintaining accurate tracking and depth measurement of the moving object. The system operates in real-time, requiring seamless coordination to ensure the LiDAR sensor is always correctly oriented toward the target as it moves within the camera's field of view.

Importance of Synchronization:

- **Accurate Depth Data Acquisition:** The LiDAR must focus directly on the tracked object. Misalignment can result in incorrect distance measurements or failure to track an object.

- **System Efficiency:** Proper synchronization ensures that every depth measurement is meaningful, helping to optimize the overall performance of the system.

4. Pipeline Integration

The servo control subsystem is seamlessly integrated into the FPGA's processing pipeline, ensuring real-time performance and efficient operation by overlapping tasks across multiple stages. Data from the background subtraction module is continuously processed to identify moving objects and determine their center of mass (CoM) coordinates, which are then translated into servo angles and synchronized with the LiDAR system for depth measurements. Each component operates in parallel, minimizing delays and enabling precise tracking in a 3D space.

C. Lidar and Display (Makar)

To establish communication between the LiDAR and FPGA, the UART protocol was employed with a baud rate of 115200, 8 data bits, and 1 stop bit. Each data frame transmitted by the LiDAR consists of 9 bytes, containing information such as the distance value, signal strength, chip temperature, and a checksum for error detection.
For this project, the **TFmini-S LiDAR module** from Benewake (Beijing) Co., Ltd. was selected due to its excellent tracking precision, wide range, and cost-effectiveness. The module uses a straightforward interface with two signal wires and two power wires. The wiring configuration is as follows:

- **Black wire**: Ground, connected to the GND pin on the PMOD-B row.
- **Red wire**: Power, connected to the +5V on the FPGA.
- **Green wire**: Data transmission from FPGA to LiDAR or clock signal, depending on the mode.
- **White wire**: Data transmission from LiDAR to FPGA.

The TFmini-S supports three operational modes:

1. **UART mode**: Data is transmitted as a string.
2. **I²C mode**: Suitable for lower pin count requirements.
3. **Custom output settings**.

Experimental analysis revealed that the UART mode consistently transmitted frames in a little-endian format, with each frame consisting of 9 bytes sent consecutively without interruption. The format of the bytes is as follows:

- **Byte 0**: Frame header (0x59).
- **Byte 1**: Frame header (0x59).
- **Byte 2**: Lower 8 bits of distance.
- **Byte 3**: Higher 8 bits of distance.
- **Byte 4**: Lower 8 bits of signal strength.
- **Byte 5**: Higher 8 bits of signal strength.
- **Byte 6**: Lower 8 bits of temperature.
- **Byte 7**: Higher 8 bits of temperature.
- **Byte 8**: Checksum (sum of previous 8 bytes, lower 8 bits).

To handle this communication, a **LiDAR UART receiver module** was developed. This module ensures alignment with the incoming frames by verifying the frame headers (0x59) and confirming the checksum. If the validation fails, the module discards the entire frame.
Additionally, the LiDAR supports command-based communication via UART, with frames structured as follows:

- **Byte 0**: Frame header (0x5A).
- **Byte 1**: Frame length (including header and checksum).
- **Byte 2**: Command identifier.
- **Bytes 3 to N-2**: Data segment (little-endian format).
- **Byte N-1**: Checksum (sum of all preceding bytes, lower 8 bits).

A dedicated module was implemented to send commands to the LiDAR, enabling functionalities such as:

- **System reset**: 5A 04 02 60.
- **Set frame rate**: 5A 06 03 LL HH SU.
- **Set baud rate**: 5A 08 06 H1 H2 H3 H4 SU.
- **Obtain firmware version**: 5A 04 01 SF.

After evaluating all communication options, UART mode was selected for its simplicity and robustness. A logic analyzer was used to validate the consistency of UART packets and ensure adherence to protocol specifications.

Display System

The display system communicates with the FPGA via the HDMI protocol, with the goal of overlaying text such as

distance measurements onto a live video feed. Initially, the implementation used sprite sheets to display digits (0-9). The sprite sheet approach involved converting an image into a .mem file in binary colors, minimizing the data file size using a custom script. However, issues with BRAM utilization arose during this process.
To address these challenges, the team transitioned to a **vector-based graphics approach**. Instead of using pre-rendered sprites, the module hardcoded the line segments necessary to render each digit. When displaying data, pixel values from the camera feed were dynamically replaced with white pixel values to represent the digits. This approach proved more efficient and avoided BRAM limitations, successfully enabling the display of numerical data on the video output.

To display numerical digits, the seven-segment digit format was employed, similar to the design used on the built-in LED seven-segment displays available on the FPGA. The vertical segments occupy 80% of the total height, leaving 10% margins at both the top and bottom. Similarly, the horizontal segments span 40% of the total width, with 30% margins on the left and right sides. This layout ensured a clear and proportional digit representation on the display.

Binary to Binary-coded decimal conversion
The LiDAR outputs data in a 16-bit binary format, which needs to be converted into a human-readable decimal format for display. To achieve this, a double-dabble algorithm module was implemented to convert the binary numbers into binary-coded decimal (BCD) format. In BCD, each digit of a decimal number is represented by a 4-bit binary equivalent.
Among the various coding schemes available for BCD, the project utilized the 8-4-2-1 coding method, where each decimal digit is directly represented by its binary counterpart as follows:

- 0: 0000
- 1: 0001
- 2: 0010
- 3: 0011
- 4: 0100
- 5: 0101
- 6: 0110
- 7: 0111
- 8: 1000
- 9: 1001

The double-dabble algorithm is a methodical approach to convert binary numbers into Binary-Coded Decimal (BCD) format. It operates by recursively shifting the binary number string to the left while simultaneously adjusting the BCD representation. The process begins with the least significant bit (LSB) of the binary number, and as each bit is shifted, it is appended to the BCD representation.

To ensure accurate conversion, the algorithm applies a critical step: whenever a digit in the BCD representation reaches or exceeds the value of 5, 3 is added to it. This addition ensures that the BCD representation aligns with the 8-4-2-1 coding scheme after the next shift, as shifting a digit greater than or equal to 5 by one position would otherwise produce an invalid BCD digit.
The steps of the algorithm can be summarized as follows:

1. **Initialization**: Start with the binary number and an empty BCD register.
2. **Bit Shifting**: Shift the binary number left by one bit, moving the shifted bit into the appropriate position in the BCD register.
3. **Adjustment**: Check each digit of the BCD. If a digit is greater than or equal to 5, add 3 to it to correct the representation for the next shift.
4. **Repetition**: Repeat the shifting and adjustment process for all bits in the binary number.
5. **Completion**: Once all bits are processed, the BCD register contains the equivalent decimal representation in 8-4-2-1 format.

This algorithm ensures a consistent and efficient conversion from binary to BCD, resulting in a 20-bit number that represents BCD for initial 16-bit number.

III. Challenges

A. Background Subtraction (Ezekiel)

Overview

A robust background subtraction algorithm implementation presents many significant challenges, which mainly stem from the intricate nature of the algorithm and the technical barriers of working with hardware-level constraints. While the core mathematical operations behind background subtraction are conceptually straightforward in software, the limitations of FPGA hardware—especially when performing resource-intensive operations such as division and square root in fixed-point arithmetic—required a steep learning curve and substantial time investment.

Integrating the modules added further complexity to the system. The classification process requires inputs from both the current frame and the background model, increasing the necessity of DRAM access and module synchronization. The following challenges arose during implementation:

- **Increased Memory Requirements**:
    - The classification stage requires not only the mean and variance but also derived metrics like α and CD, necessitating additional storage and computational overhead.
- **Timing and Latency**:
    - The classification module must receive outputs from prior stages (e.g., statistical modeling and square root computation) in a timely manner. Any delay in these modules propagates to the classification stage, potentially stalling the system.
- **Fixed-Point Arithmetic in Classification**:
    - Implementing the $\sqrt{\cdot}$ and $\div$ operations in fixed-point required careful design to balance precision and resource utilization (see *Appendix)*.

Challenges

The first major challenge was the lack of experience with designing such a complex and interconnected system on an FPGA, particularly in managing external memory (DRAM). The background subtraction algorithm, as described in [*1,3*], requires maintaining a per-pixel running sum and sum of squares for each of the RGB color channels. For high-resolution video, this results in a significant volume of data. Each pixel's running sum requires 48 bits per color, while each sum of squares requires 64 bits per color. For a 720p resolution (1280x720), the data demands quickly scale into megabytes (*Background Model Memory Analysis*), necessitating the use of DRAM. However, integrating DRAM into the system using the MIG DDR protocol proved to be a considerable challenge due to the lack of easily accessible testbenches with representations of latency, and memory access synchronization.

The second major challenge was the complexity of implementing core mathematical operations with the hardware constraints. For instance, operations such as square root and division, which are essential for calculating brightness distortion, chromaticity distortion, and pixel classification, are not natively efficient on FPGA hardware. Developing a fixed-point implementation of the square root using Newton-Raphson approximation required nearly a week of focused work to ensure both accuracy and compatibility within the pipelined system. Following this, significant time was spent integrating the square root module with the larger system, ensuring it operated correctly within the constraints of the pipeline. This integration had to account for timing synchronization with other modules such as the background model, brightness distortion, chromaticity distortion, and pixel classification. The effort to pipeline these modules properly consumed another week, as each module needed to independently verify its functionality before integration.

A further complication arose when testing the algorithm. Unlike software, where modular components can be tested independently and easily, the full functionality of this system on the FPGA requires all modules to work seamlessly together. The interconnected nature of the modules meant that partial hardware testing was unfeasible until a working top-level design was completed. To simulate the system in isolation, a "magic memory" abstraction was used to simulate DRAM with no protocol overhead or latency, allowing the system to function as if unlimited resources were available. However, moving from this idealized simulation environment to the FPGA's MIG DDR protocol introduced new challenges. Testing and debugging the DRAM interactions, particularly the timing and memory access patterns, required the creation of extensive Python testbench classes to simulate memory behavior and latency.

Here, Kiran's (LA) assistance was invaluable. Their development of MIG DDR and traffic generator simulation classes provided the framework for working with the simulated DRAM. Without this, it would have been exceedingly difficult to implement and test the background model at scale. Despite Kiran's contributions, resolving memory access timing and ensuring compatibility with the algorithm required many iterations of debugging and testing.

Despite making progress on the background model, which successfully computed the running sums and sum of squares for each pixel, integrating this module with downstream components like chromaticity distortion, brightness distortion, and pixel classification required substantial effort. These modules not only needed to retrieve data from DRAM efficiently but also required correct synchronization and pipelining to avoid delays or stalls in processing incoming pixels. For example, the pixel classification module required the mean and standard deviation of each pixel (stored in DRAM) to be used in conjunction with the calculated brightness and chromaticity values of incoming pixels. This required precise timing, memory coordination, and the implementation of a buffer system to minimize latency.

Stephen's (LA) BRAM management module was essential in helping develop a "magic" memory buffer to address these latency concerns. The BRAM management module ensured that data could be prefetched and staged in a way that appeared instantaneous to the requesting modules. This abstraction allowed progress to be made on the integration of the chromaticity distortion and brightness distortion modules with the pixel classification system. However, fully integrating these modules into the top-level design proved challenging due to timing constraints and DRAM dependencies. Additionally, real-time integration with external systems, such as the lidar and servo systems, presented further complexities beyond the scope of this specific algorithm.

Reflections and Future Improvements

Developing this algorithm demonstrated the substantial effort required to translate a computationally intensive algorithm into a hardware-friendly implementation. While the background subtraction algorithm itself is conceptually straightforward and has been extensively studied (*see reference* [1,3]), its implementation on an FPGA posed significant challenges. The iterative nature of the algorithm—requiring calculations for every pixel and every frame—created dependencies that compounded the complexity of the system. Trying to synchronize between modules, manage large data volumes in DRAM, and deal

with hardware-level constraints added layers of difficulty that were underestimated at the start of the project.

Given the constraints of time, resources, and experience, it was not feasible to complete the full integration of the algorithm. The project as it is now required approximately four hours of daily work for over three weeks, much of which was spent addressing hardware-specific challenges such as pipelining, memory management, and mathematical approximations. These challenges were further magnified by the need to integrate with external systems for the final motion-sentry device. In hindsight, this algorithm would have been better suited as a two-person project, with one individual focusing exclusively on the top-level integration and memory management, and the other dedicated to implementing the mathematical operations and ensuring their accuracy.

Despite these difficulties, the work completed laid a solid foundation for future development. The background model, chromaticity distortion, brightness distortion, and pixel classification modules have been thoroughly tested in simulation and are functionally correct in isolation. Moving forward, efforts should focus on resolving the DRAM timing issues, optimizing the memory pipeline, and integrating the system on hardware. During the final stages of the project, efforts were redirected toward collaborating with team members to integrate the full system, which included modifying the existing color tracking implementation to provide velocity and direction information in the x-y frame. This experience has provided invaluable insights into the challenges of FPGA development, and the lessons learned will inform future projects in this field.

B. Servo-Control

Integrating the servo-control system into the overall design posed significant challenges, particularly due to the dynamic and mechanical nature of the component. These challenges primarily stemmed from timing complexities, synchronization issues, and the reliance on limited input data:

- **Mechanical Delay in Sero Motion:** Unlike electronic components, servos introduce inherent mechanical delay as they physically move to the commanded angle. This delay required careful consideration in the system's real-time pipeline, as new position commands could not be issued until the servo completed its motion. Failure to account for these delays would have led to misalignments between the LiDAR's orientation and the target's actual position.
- **PWM Signal Timing and Resolution:** The servo's 50 Hz duty cycle demanded precise synchronization with the Camera's 200 MHz clock. Generating PWM signals with sufficient resolution while maintaining real-time performance introduced latency into the control loop. This latency limited how quickly the system could respond to rapid object movements, reducing tracking fidelity under dynamic conditions.
- **2D Positional Dependency:** The lack of direct depth feedback during initial tracking stages forced the servo-control system to rely solely on 2D positional data from the camera. This limitation required assumptions about the target's movement and distance, reducing precision during high-speed or erratic object motion.

Reflections and Future Improvements

There are several areas for improvement within the servo-control sub-system. Implementing these changes would require additional time as many refinements, such as integrating depth-driven feedback, optimizing the timing pipeline, and automating calibration, demand extensive testing and development cycles. A longer timeline would allow for more robust iteration, ensuring that each component is thoroughly validated and better integrated into the overall system.

IV. Enhanced Calibration: While calibration was effective, this process can be streamlined and automated in future iterations. This could be done by incorporating a feedback mechanism using LiDAR or a secondary camera, allowing for real-time verification and adjustment of the servo's range and alignment, ensuring greater accuracy and consistency.

V. Depth-Driven Feedback Integration: Due to our current reliance on 2D positional data, the system is not able to respond dynamically to changes in depth. A direct integration of depth feedback from the LiDAR could enable adaptive servo adjustments helping to improve tracking precision.

VI. Improved Timing Pipeline: Addressing the latency introduced by the mechanical delays and the 50 Hz PWM duty cycle would be a priority. Predictive algorithms to preemptively calculate servo positions could help mitigate these delays, maintaining alignment with the moving target.

A. LiDAR and display output

- **LiDAR Integration**: Working with a device that operates on a custom protocol, albeit based on UART, introduced significant challenges due to its unique frame structure. Initially, debugging this communication was cumbersome, as the LiDAR's data frames required precise alignment and validation. The introduction of a logic analyzer greatly facilitated the debugging process, enabling reliable data acquisition and consistent communication with the LiDAR module.

- **Screen Output**: Displaying text digits on the screen posed significant technical hurdles. The initial approach involved using a preloaded digit sprite sheet stored in BRAM, dynamically selecting the appropriate sprite segment based on the desired digit. However, discrepancies arose between the expected and retrieved data at specific BRAM locations. After extensive troubleshooting, an alternative approach was adopted. Instead of relying on preloaded sprites, digits were rendered directly onto the screen as 7-segment stencils by programmatically drawing lines in the appropriate positions. This solution not only circumvented the BRAM issue but also provided a reliable and efficient method for screen output.
- **Integration:** Integrating the LiDAR module with the display output introduced several challenges. One significant issue involved the LiDAR module occasionally falling out of sync with the data order sent by the LiDAR device. Another challenge arose from discrepancies between the clocks used across various modules. To address the synchronization issue, communication with the LiDAR was enhanced, enabling the ability to reset the LiDAR on demand, effectively realigning the data stream. The clock discrepancy was resolved by carefully ensuring that all components of the LiDAR system operated on the same clock, providing consistent timing and functionality throughout the integration.

6. Appendix

Fixed-Point Arithmetic in Verilog *(Ezekiel)*

Fixed-point arithmetic is a numerical representation used in hardware design to perform arithmetic operations with fractional precision, offering a balance between the simplicity of integer arithmetic and the precision of floating-point operations. In Verilog, fixed-point arithmetic is particularly advantageous for FPGA designs, where hardware constraints such as area, power, and latency are critical considerations.

Fixed-Point Representation

In fixed-point arithmetic, numbers are represented as integers with an implicit radix point. The position of the radix point is determined by the number of fractional bits, commonly denoted as Qm.n, where:

- m represents the number of bits in the integer part.
- n represents the number of bits in the fractional part.
- The total bit-width W=m+n+1 includes one bit for the sign-in signed representations.

For example, a Q16.16 fixed-point format allocates 16 bits for the integer part and 16 bits for the fractional part.

Square Root: Fixed-point square root is more complex, as it often involves iterative algorithms such as Newton-Raphson or CORDIC. The result remains in the same format, but intermediate results must be managed carefully to avoid overflow or underflow.

Implementation in Verilog

Fixed-point arithmetic in Verilog is implemented using standard integer data types, with careful attention to bit-widths and scaling. Consider the following example of a simple Q16.16 multiplication:

```
module FixedPointArithmetic (

    input logic signed [31:0] a,  // Q16.16
format

    input logic signed [31:0] b,  // Q16.16
format

    output logic signed [31:0] product    //
Q16.16 format

);

    assign product = (a * b) >>> 16;  //
Multiplication with scaling

endmodule
```

Advantages of Fixed-Point in Hardware

- Hardware Efficiency: Fixed-point operations require fewer resources (e.g., LUTs, DSPs) compared to floating-point, making them ideal for FPGA implementations.
- Predictable Latency: The deterministic nature of fixed-point arithmetic simplifies timing analysis and pipelining.
- Custom Precision: Designers can tailor the bit-widths to meet the application's precision and range requirements, optimizing resource utilization.

Challenges in Fixed-Point Arithmetic

- Overflow and Underflow: Fixed-point arithmetic lacks dynamic range, leading to potential overflow or underflow if values exceed the representable range. This must be managed through scaling or saturation logic.

- Precision Loss: Right shifts during scaling can result in truncation errors, necessitating careful design to balance precision and performance.
- Debugging Complexity: Fixed-point calculations require careful verification, as errors in scaling or alignment can propagate through the system.

Code:
https://github.com/EzekielDaye/Motion-Sentry

7. References

[1] S. Chen, Y.-P. Hsu, and Y.-T. Tsai, "FPGA-Based Implementation of Real-Time Background Subtraction for Video Surveillance Using the Horprasert Model," *Sensors*, vol. 12, no. 1, pp. 585–605, 2012.

[2] Xilinx, *FPGA Applications in Radar and Defense Systems*, 2017.

[3] T. Horprasert, D. Harwood, and L. Davis, "A Robust Background Subtraction and Shadow Detection," *Proc. Computer Vision Laboratory*, University of Maryland, 2000.

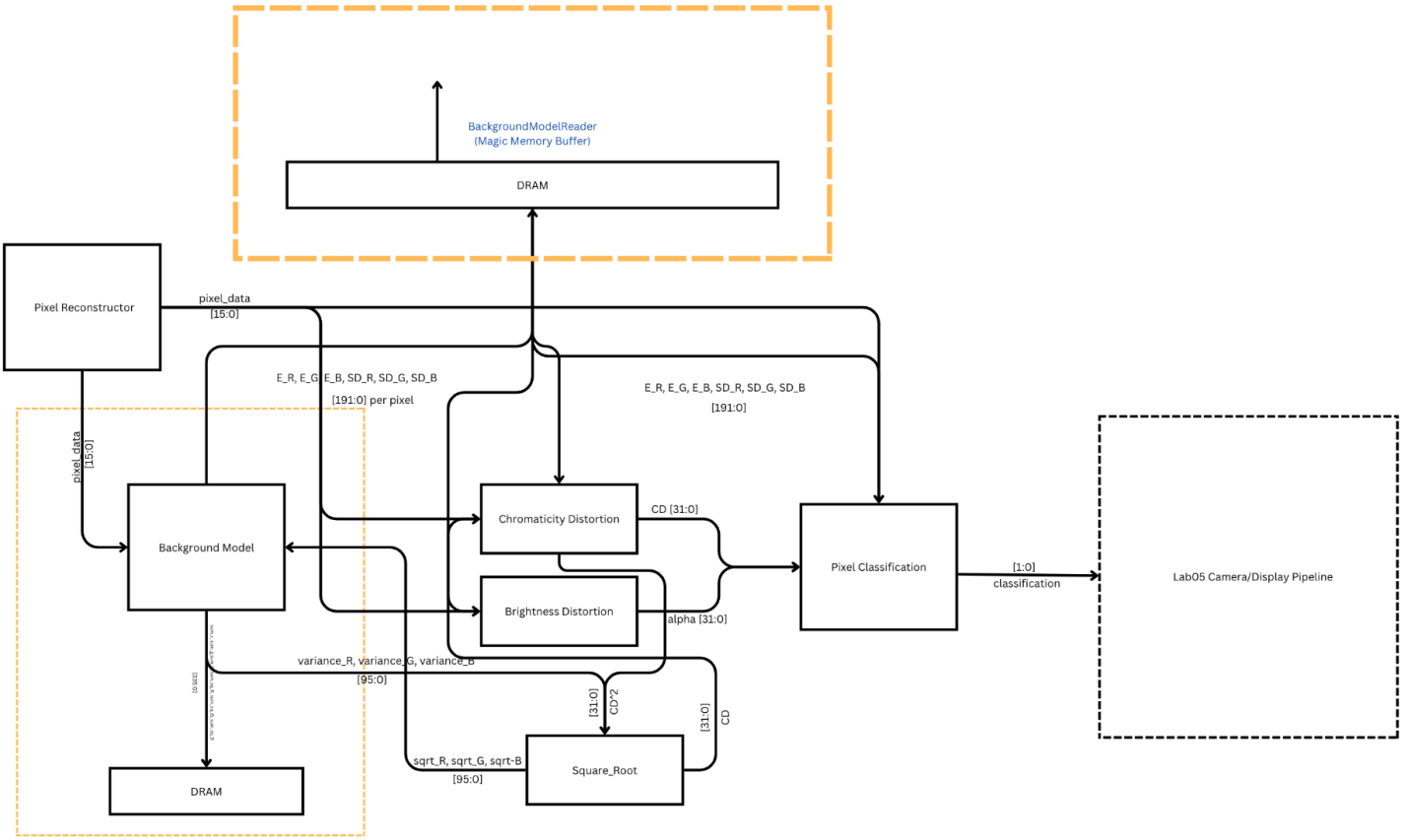

Fig 1. Top Level Overview of Background Subtraction

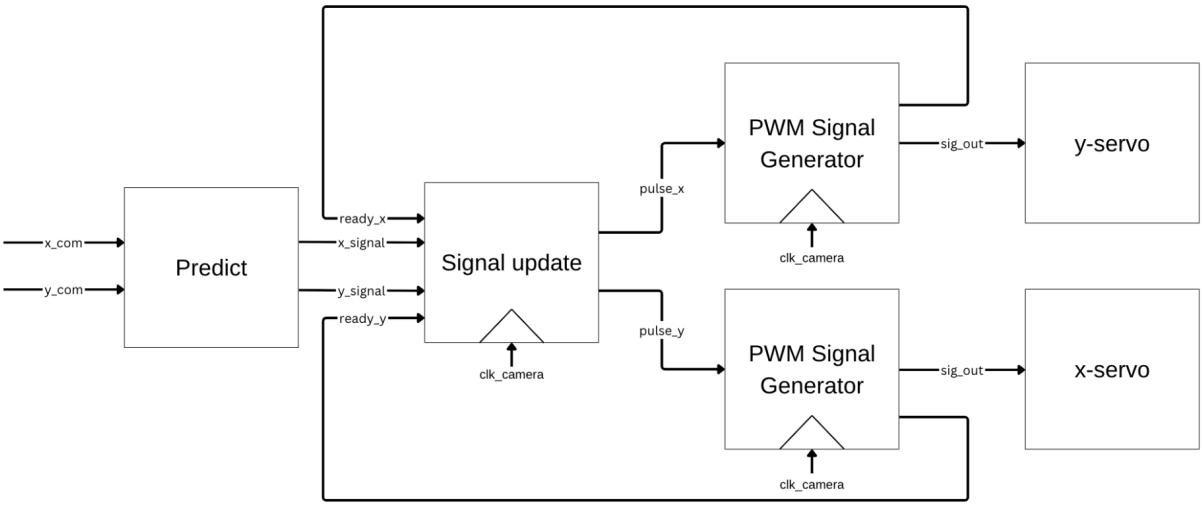

Fig 2. Block Diagram of Servo Control System

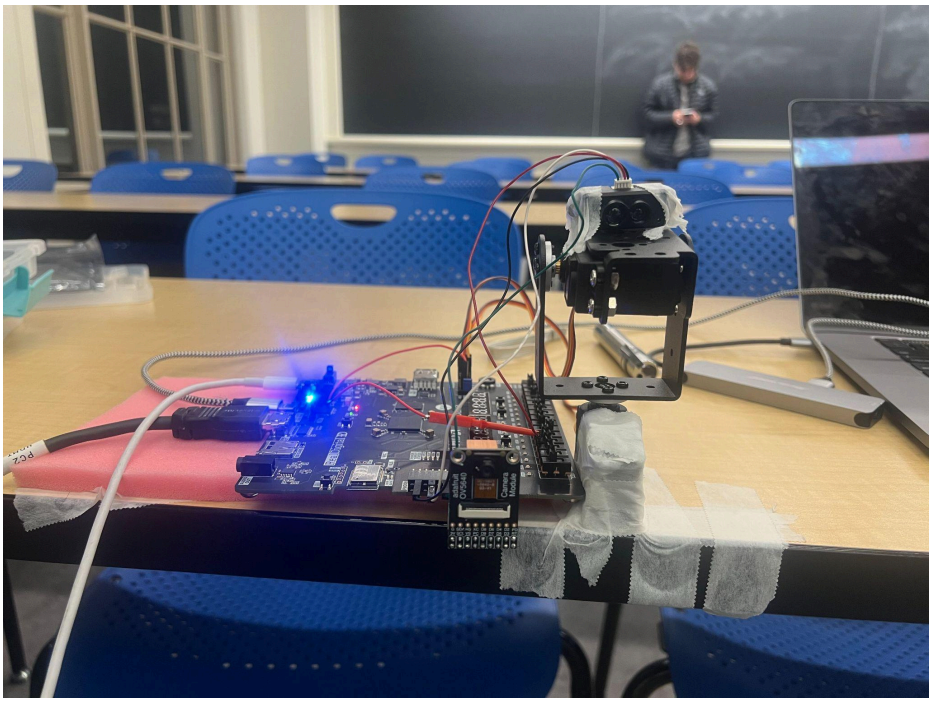

Fig 3. Image of the FPGA, Camera, and LiDAR systems connected