Embedded System Design (CS-476)

# Project: image recognition and tracking on FPGA

**Authors:**

Andrea Grillo, 370780

Riccardo Lionetto, 370934

**Professor:**

Ties Jan Henderikus Kluter

June 2024

# Contents

# 1 Introduction

The project consists in creating an object detection system on a virtual embedded system on FPGA, paired to a tilt-pan tracking structure that allows the camera to follow and re-align on the object center.

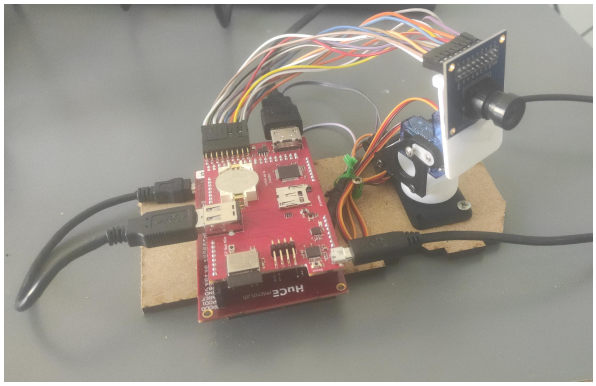


Figure 1: Complete system

## 1.1 Tracking system development

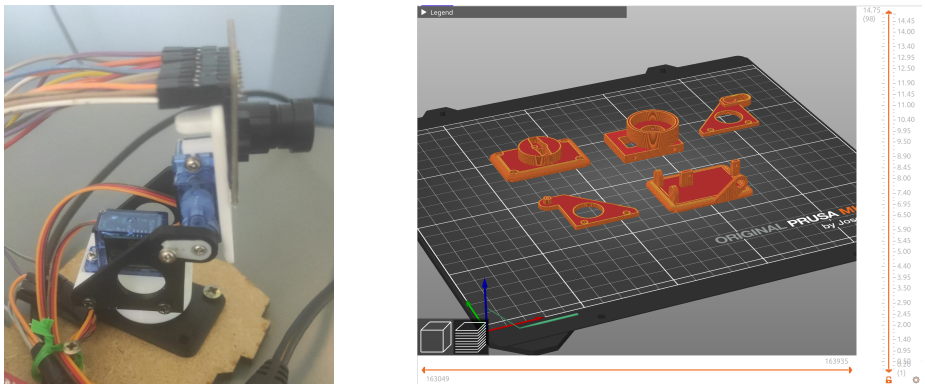
The building steps for the pan-tilt tracking system are outlined in detail below. This system is responsible for allowing the camera to follow and re-align on the object center dynamically.

### 1.1.1 Pan-Tilt Mechanism

The structure that allows for the camera movements is a pan-tilt mechanism with two degrees of freedom. This setup enables panning(horizontal movement of the camera) and tilting(vertical movement of the camera), providing a convenient full range of motion necessary for effective tracking.
To add stability, the entire mechanism is mounted on a wooden base, which ensures that the movements are precise and that the camera remains steady during operation.

### 1.1.2 3D Printing

To create the pan-tilt structure, 3D models were downloaded and properly modified to fit our project objectives using CAD software. These models were then prepared for the 3D printing process by means of a Slicer. A proper filling value was chosen for each part to ensure they were durable and steady enough. This approach allowed us to create a tailored solution for the tracking system, making easy custom fittings and adjustments.



(a) (b)

Figure 2: Pan-tilt mechanism and slicing of its 3D models

### 1.1.3 Servo Motors

The actuation of the pan-tilt mechanism is achieved using servo motors. Specifically, the servo motors employed have 180° of rotation and Pulse Width Modulation (PWM) control. These motors are ideal for this application for two reasons. Firstly, they provide precise control over the rotation angles, leading to smooth and accurate movement of the camera. Secondly, the PWM controller does not require a high level of complexity, allowing for a relatively easy integration with the FPGA.

### 1.1.4 Extended Connections for Camera-Board Integration

To enhance the camera's freedom of movement, it was necessary to extend the connections between the camera and the board. Attaching the camera directly to the board would have required mounting the entire FPGA-board-camera assembly on the pan-tilt mechanism, requiring stronger motors and a more complex and robust system.
On the other hand, this wire extension modification allows the camera to be positioned independently on the pan-tilt mechanism, improving the overall flexibility of the tracking system and decreasing the complexity.

### 1.1.5 Power supply

The power requirements of the motors exceed what the FPGA board alone can supply. For this reason, an external power supply of 5V was used.
The FPGA, motors, and power supply were all connected to a common ground to ensure the motors operate reliably and efficiently.

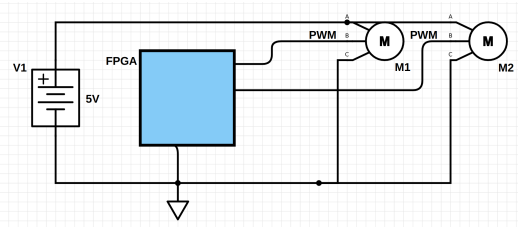



Figure 3: Slicing of the 3D models

# 2 Preliminary validations

For what it concerns the object recognition algorithm, we firstly analyzed common methods to perform object detection and identified the fundamental steps as follows:

- **Conversion from RGB to HSV:** HSV format is more robust to changes in lighting conditions compared to the RGB color model.
- **Thresholding:** this step involves applying a threshold to get the mask of the object, isolating the pixels that match the desired color range
- **Contour Detection:** this process detects the outlines of the object within the mask.
- **Isolation of the Desired Contour:** this step ensures that only the desired object is detected, based n the area of the contour or on the shape (numbers of sides of the polygon), filtering out other objects of the same color.
- **Center of the Contour:** calculating the center of the detected contour to determine the object's position.

However, it would have been computationally expensive to run all these steps on our FPGA-based architecture. For this reason, we strove to simplify the approach and define the following steps:

- **RGB Thresholding:** directly thresholding in the RGB color space to identify the object.
- **Averaging of the points of the right color:** calculating the average position of the points that match the desired color to determine the object's center.

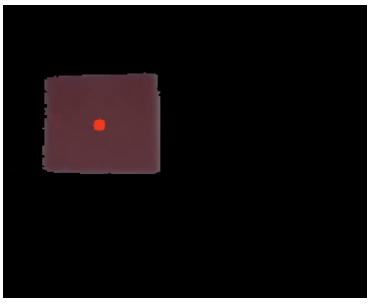
The first step has the main advantage of eliminating the need of floating point computations. The conversion from RGB to HSV requires mathematical operations that often result in fractional values indeed.
It is true that floating-point operations can be implemented as fixed-point, making them more suitable for embedded systems, but many multiplications would still be required, therefore degrading performances. For this reason, it was worth it to investigate if RGB was suitable as well for our application.

## 2.1 OpenCV algorithm testing

To assess the feasibility of this simplified approach, we used the Python library OpenCV as a proof of concept. OpenCV provided a flexible and powerful set of tools to implement and test our object detection algorithm in a fast and convenient way.

```
# Threshold the RGB image
mask = cv2.inRange(frame, lower_red, upper_red)

# Find the coordinates of all 1 values
coordinates = np.argwhere(mask == 255)

frame = cv2.bitwise_and(frame, frame, mask=mask)

if len(coordinates) != 0:
    # Calculate the mean of the coordinates along each axis (rows and columns)
    average_coordinates = np.mean(coordinates, axis=0)
    print("avg coord", int(average_coordinates[0]), int(average_coordinates[1]))

    cv2.circle(frame, (int(average_coordinates[1]), int(average_coordinates[0])), 5, (0, 0, 255), -1)
```

(a)



(b)

Figure 4: Code snippet (a) and detected object with marked center (b)

The algorithm performance resulted to be great, working in every environment that was tested.

It is however important to note the limitations that our choice produced:

- the center of only one object at a time can be computed and displayed.
- performance is higher when the color is clearly distinct from the around environment
- the use of RGB instead of HSV color format leads to less robustness in case of lighiting perturbations.

## 2.2 Hardware testing

Before diving into the complete software implementation on the FPGA, the hardware was studied and preliminary tests were conducted to better understand its requirements in terms of control and signals. In this case, the hardware components were the two servo motors.
These motors are controlled with a PWM signal, where the duty cycle defines the angle to which the motor moves. To establish the relationship between angle movement and duty cycle, the a priori knowledge gained from the datasheet on the complete range of angles (0° to 180°) and duty cycles (2.5% to 12.5%) was leveraged.

- **Arduino test:**

An Arduino testbench was used to see the servos in motion, as the existing libraries for Arduino facilitate fast development. This preliminary testing phase ensured a clear understanding of the control signals needed for the servo motors, which was crucial for the subsequent FPGA implementation.

```
Servo servo1, servo2;
int pos = 90;

void setup() {
  servo1.attach(9);
  servo2.attach(10);
  servo1.write(pos);
  servo2.write(pos);
}

void loop() {
    for (pos = 55; pos <= 125; pos += 1) {
    servo1.write(pos);
    servo2.write(pos);
    delay(15);
  }
  for (pos = 125; pos >= 55; pos -= 1) {
    servo1.write(pos);
    servo2.write(pos);
    delay(15);
  }
}
```

(a)

(b)

Figure 5: Code snippet (a) and working testbench (b)

- **FPGA implementation:**

The FPGA documentation indicated that the suitable pins for PWM generation are the 'PMOD' pins. To use these pins, the Verilog top-level entity had to be modified to add a 2-bit output signal (one bit per motor). Additionally, the TCL file was updated to define the pin assignments.

t this point, a problem was faced. The output voltage of these PMOD pins was 3.3V, whereas the PWM pin on the motors required 5V. Before creating a level shifter circuit to boost the voltage, the motors were tested with the 3.3V signals. Surprisingly, the motors worked with these signals. This indicates that the threshold level the motors used to distinguish high/low signals was lower than 3.3V.

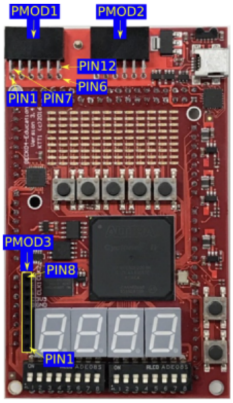



Figure 6: PMOD pins location on the FPGA

# 3 Software solution

The complete software implementation on the FPGA left all the processing to the CPU. The key steps are:

1. Acquire a single camera frame.
2. Save the frame into a RAM.
3. CPU reads the memory and:
   - applies thresholding and conversion to grayscale.
   - averages the coordinates of the masked points.
4. CPU computes the position commands and generates PWM signals.

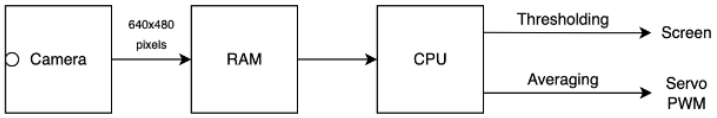
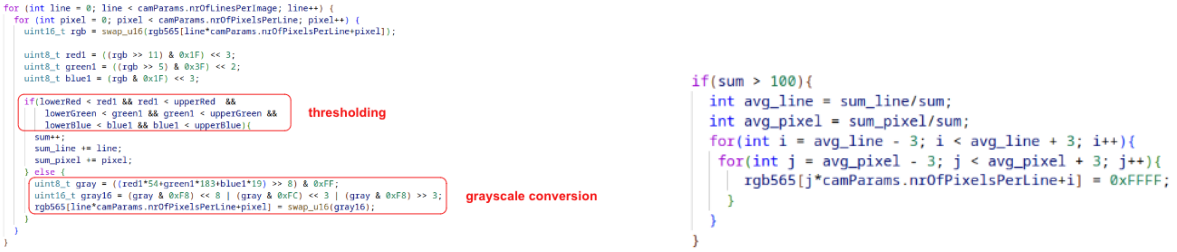



Figure 7: Software processing iterations

It is important to note that the boundary values for the thresholding process previously found through the OpenCV pre-evaluation were not suitable anymore. The reason behind lays in the fact that different cameras were used, each with different sensibility and characteristics.
Therefore, a new process had to be found to compute new boundaries. The solution involved the following passages:

1. place the object to be detected in front of the camera.
2. print the collected RGB values of the object.
3. choose boundary values around the collected ones.




(a) (b)

Figure 8: Software code snippet: thresholding and grayscale conversion (a), averaging (b)

In the next section performance studies of this software implementation will be discussed that will lead to custom hardware acceleration.

# 4 Hotspots detection

A crucial step in the process of creating a final real-time version of the object detection and tracking system involved the analysis of the performance of the software solution.
Clock cycles were profiled and studied to extract bottlenecks and weak points of the CPU processing.In the table below the profiling results of the thresholding and averaging processes are listed.

| | | | |
|---|---|---|---|
| CPU cycles | 44 millions | Execution time | 0.6 seconds |
| CPU stall cycles | 26 millions | CPU waiting time | 60% |
| BUS idle cycles | 23 millions | BUS busy time | 48% |

Table 1: Profiling results of the thresholding and averaging processes

From these values important considerations can be obtained. It appears that the CPU is waiting for 60% of the time and therefore working for only 40% (18 million cycles).

- **Hotspots**

The high amount of data that had to written/read multiple times from the RAM appeared to be the main limitation. All pixels of the camera feed get saved in the RAM memory: $640x480pixels * 16bits = 614,4kByte$. From there, the CPU reads them, perform the thresholding operation (leaving the detected object to its original color) and write them back in the RAM so that they could be displayed.

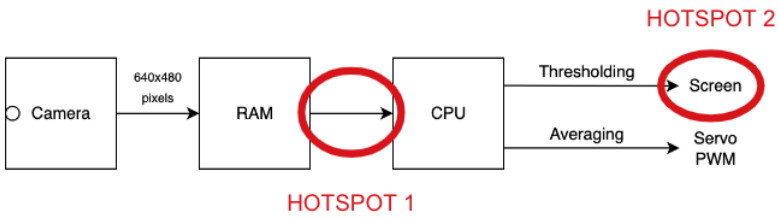



Figure 9: Hotspots in the software implementation

We also thought about how a data cache would affect the performance of this solution. The C code has to go through the whole image, which is contiguous in memory. This means that a data cache would improve a bit the performances by loading the whole line of the cache, therefore reducing the number of bus requests and transactions, but the improvement would not be so big as the considerable size of the frame would lead to a very big number evictions and reloading of the data in memory.

# 5 Hardware solution

After developing a complete and functioning software solution, and having analyzed its hotspots, we moved to accelerate the implementation by creating custom hardware solutions.

It was crucial to firstly make sure to have clear all the key features we required our architecture to have. These were defined as follow:

1. Maintain the camera feed visualization.
2. The camera feed should display a grayscale background while keeping the detected object in its natural color. This aspect has the benefit of serving as a powerful debugging feature too.
3. The center of the detected object should be masked with white pixels.
4. Motors should be properly PWM controlled to have object tracking in action.

All the above features were implemented with different mixed strategies:

- **camera custom verilog module**: its purpose was to perform an in-stream thresholding on the camera feed and a grayscale conversion.
- **software processing performed by the CPU**: its role was averaging applying specifically designed formulas to compute the detected object center and compute the motors duty cycles.
- **custom verilog module:** its purpose was to generate the PWM signals needed to control the servo motors.

## 5.1 Camera custom module

The camera module makes use of different internal blocks to perform its actions, as it can be visualized in Figure 10.

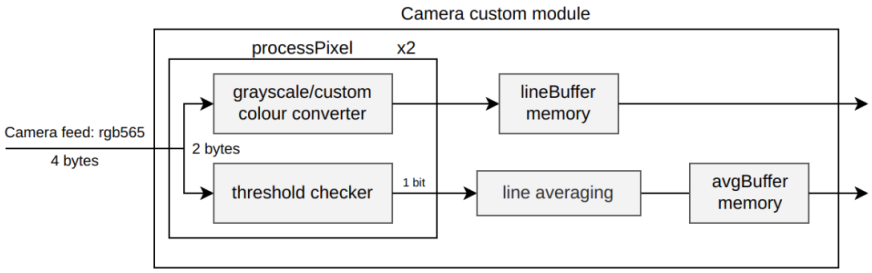



Figure 10: Structure of the in-stream pixel processing of the camera custom block

- **Thresholding block**

This block has the purpose of checking for every pixel if its related value is correspond to the color to be detected. It receives one pixel (2 bytes in RGB565 format) and push the same pixel in output with associated a single bit (1 or 0, depending if the pixel is considered part of the target object or not).
For the software implementation the best value to maximize the success rate of the object detection were found, going over a process of fine-tuning.

**Implementation with comparators:**
In its first implementation, the thresholding block checked against those very same values. In order to perform such action, the synthesizer had to make use of comparator logic. Despite the fact that the latter had an impact of used gates negligible, compared to the overall architecture in place, it looked interesting to try a novel approach to better highlight the omnipresent tradeoff between cost and performances.

**Implementation with no comparators:**
For this reason, a second version of the thresholding block came from the effort of substituting all

| | fine-tuned | approximated |
|---|---|---|
| red | 15-25 | 8-15 |
| green | 0-10 | 0-7 |
| blue | 5-15 | 3-7 |

Table 2: Fine-tuned thresholding values and approximated solution

comparators with a cheaper and more hardcoded solution. The previous fine-tuned values were properly approximated in ways that could allow using simple wires and inverters. The table below summarizes those changes.

```
assign s_rok = s_red > 5'd15 & s_red < 5'd25;
assign s_gok = s_green < 6'd10;
assign s_bok = s_blue > 5'd5 & s_blue < 5'd15;
```

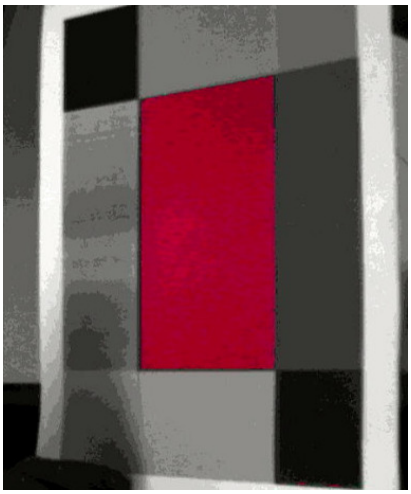
(a)

```
// Check with no use of comparison operators
assign s_rok = ~s_red[4] & s_red[3];
assign s_gok = ~s_green[5] & ~s_green[4] & ~s_green[3];
assign s_bok = (s_blue[2] | (s_blue[1] & s_blue[0])) & ~s_blue[4] & ~s_blue[3];
```

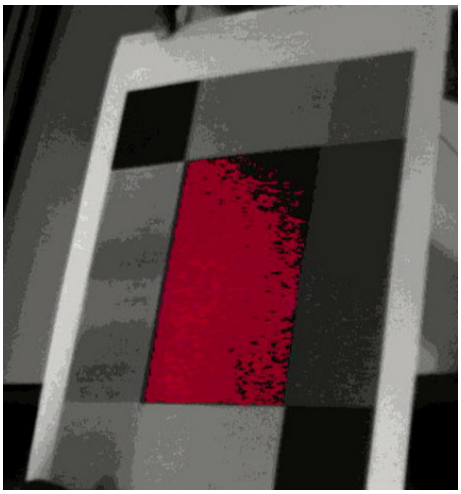
(b)

Figure 11: Code with comparators (a) and no comparators (b)

As expected, the use of comparator led to more reliable performance at the cost of ideally more 'expensive' logic, whereas approximated boundaries produced worst result with a gain in 'less' used logic. The system became indeed more prone to detect very similar colors and to miss some of the true points.



(a)



(b)

Figure 12: Performances with use of fine-tuned values (comparators) and chip usage optimization (no comparators)

- **Grayscale + custom color converter**

This block was responsible of applying to the camera feed pixels an in-stream conversion to grayscale for all the pixels that were not detected as belonging to the target object. It processes one pixel per time, receiving two bytes in input and giving in output the same amount of bytes. The output signal could be smaller if the conversion was purely to grayscale format, but in this case the detected object color is leaved, forcing the RGB565 format to remain.

- **PWM generator**

This instance handles the conversion from the instructions received by the software positioning code to PWM signals needed to control the two servos actuating the pan-tilt system.
It requires the value of two duty cycles (one per motor) to be compared against a single internal counter:

- when count $<$ duty_cycle value: pin output on high
- when count $\geq$ duty_cycle value: pin output on low

The working frequency of the motor was initially set to be 50Hz. However, to achieve such precise value additional actions had to be designed. For this reason, simplifying the implementation without actually impacting performances seemed to be a good choice. The idea was therefore to set the count register size to a value that would allow an independent 'overflow' reset with no need of an active reset operation. The closest working frequency appeared to be 69Hz, obatined with a count register of 20 bits:
$clock/count\_register = 72MHz/2^{20} = 69Hz$

## 5.2 CPU averaging

The main challenge we have faced during the development of the hardware solution has been on the choice of the strategy to do the averaging to find the center point of the object. We have come up with different ideas, and for each of them we have analysed first the feasibility, and then the pros and cons.

- **Mask Buffer Memory in camera module.**

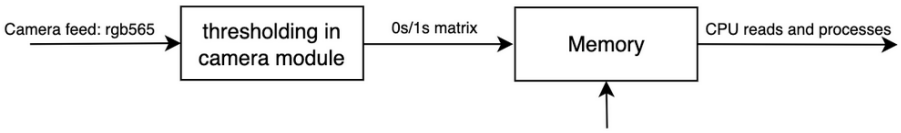



Figure 13: First HW idea

The problem of this idea is the size of the mask to be generated. As the frame is 640x480 pixels, the needed memory is 38k, which is to big to be fit in the FPGA.

- **Custom module with thresholding and averaging in hardware with DMA transfers.**

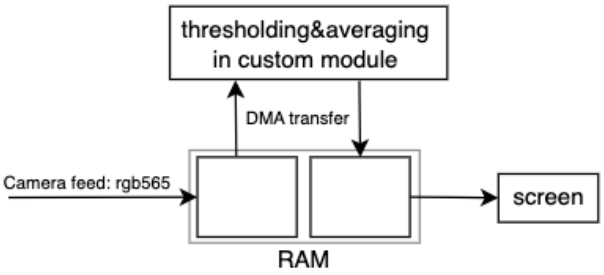



Figure 14: Second HW idea

This solution was considered feasible, but has a big disadvantage. The image frame has to be moved first to the camera module to the RAM, then it is read by the custom module, than it goes back to RAM to be finally read by the screen module. Therefore the frame has to be transferred 4 times on the bus, which would lead to big delays and heavy use of the bus.

- **Sparse mask matrix.**

  As the number of pixels which are in the object is usually quite smaller if compared to the total number of pixels, we had the idea of saving the the mask as a sparse matrix, saving only the coordinates of the pixels which are part of the object.

  The main drawback of this approach is that the size of the memory needed is variable, as the number of detected pixels change at every iteration, which makes the handling of the memory very difficult to be done in hardware.

- **In-stream thresholding and split averaging between hardware and software.**

  To overcome all the drawbacks found in the previous ideas, we wanted to find an intermediate solution, in which a part of the computation would be done in hardware and the rest in software.

  The main challenge in this is how to generate some intermediate results that can be recombined to do the final average in software. To do this we played a bit with the maths of the averaging.

  Given the list of the points that are in the object: $(x_i, y_i) \qquad \forall i \in 1, .., n$

The center is computed as:

$$\bar{x} = \frac{\sum_i x_i}{n} \qquad \bar{y} = \frac{\sum_i y_i}{n}$$

The averaging can be split over the rows and the columns in the following way:

For each line $j$ compute:

1. number of pixels in the object $n_j$
2. sum of the indexes of those pixels $s_j$

Therefore the coordinates of the center point are finally computed as follows:

$$\bar{x} = \frac{\sum_j s_j}{n} \qquad \bar{y} = \frac{\sum_j j \cdot n_j}{n} \qquad \text{where} \quad n = \sum_j n_j$$

The intermediate results to be stored for each line are:

1. $n_j \in 0, .., n_{pix}$ $\rightarrow$ fits in 10 bits
2. $s_j \in 0, .., k$ $\quad k = \frac{n_{pix}(n_{pix}+1}{2}$ $\rightarrow$ fits in 18 bits

This means that the results needed for each line fit in a 32 bit memory word (actually in 28 bits, but we have assigned the remaining 4 bits to be split between the 2 numbers, they would be wasted anyway if we want to keep the default size of the memory word).
As we need to store those results for each line, we need a memory with at least 480 32 bits words. This means that we can use the already present 2K memory module.

We have defined two counters to compute the intermediate results in hardware:

```
reg [11:0]  sum_1Pixels;
reg [19:0] sum_idxPixels_line;
wire [19:0] idx_Pixel= {10'd0, s_pixelCountReg[10:1]};

always @(posedge pclk)
begin
  sum_1Pixels <= (reset == 1'b1 || s_hsyncNegEdge) ? 12'd0 :
                 s_weLineBuffer ? (sum_1Pixels + {11'd0,outputMask1} + {11'd0, outputMask2}) : sum_1Pixels;
  sum_idxPixels_line <= (reset == 1'b1 || s_hsyncNegEdge) ? 20'd0 :
                s_weLineBuffer ? (sum_idxPixels_line +
                          (outputMask1 ? (idx_Pixel-20'd1) : 20'd0) +
                          (outputMask2 ? idx_Pixel : 20'd0)) : sum_idxPixels_line;
end
```

Figure 15: HW line counters

And the final averaging is done in software as follows:

```
for (int i = 0; i < camParams.nrOfLinesPerImage; i++) {
  asm volatile ("l.nios_rrc %[out1],%[in1],%[in2],0x7":[out1]"=r"(result):[in1]"r"(16),[in2]"r"(i));
  index_pixels_per_line = result >> 12;
  pixels_per_line = result & 0xFFF;
  sum_pixels += pixels_per_line;
  sum_index_pixels += index_pixels_per_line;
  sum_lineindex_pixels_per_line += pixels_per_line * i;
}

if(sum_pixels > 100){
  int avg_line  = sum_index_pixels / sum_pixels;
  int avg_pixel = sum_lineindex_pixels_per_line / sum_pixels;
}
```

Figure 16: SW final averaging

# 6 Performance analysis

## 6.1 Assembly code analysis

Before considering the implementation as completed and proceeding with the performance tracking, an important step was to read the assembly code generated from the compiler to spot eventual implementation errors in the C code.
For example, there were some useless accesses to memory due to *volatile* qualifiers defined for variables that actually did not need it.

### 6.1.1 Software averaging analysis

For the averaging operation, we made sure that all the computations could be made on the CPU using registers instead of having accesses to memory.
Achieving this result was very important, as the effect was removing the source of uncertainty coming from the bus, leading to a constant number of clock cycles each execution step.


(a) (b)

Figure 17: Software averaging: C code (a) and respective assembly (b)

In the cycle 13 instructions are repeated for exactly 480 (number of lines) times, therefore a total number of instructions of 6240.
We also know that each custom instruction call takes 2 cycles, as the memory read operation stalls the CPU for 1 cycle. Therefore we have to add at least 480 stall cycles.
We then measured the actual number of cycles using the profiling counters and obtained the following:

| CPU cycles | 7745 |
|---|---|
| CPU stall cycles | 536 |

Table 3: Averaging profiling

As we can see, the numbers are very close to the number we predicted by analysing the assembly code.

### 6.1.2 Center computation and masking, PWM commands

On the other hand, for other parts of the code it was difficult to have a precise prediction of the number of cycles needed at each execution. An example is the C code responsible for the center coordinates computation, its masking and the generation of the PWM control commands. This is due to the fact that it makes memory accesses and have multiple branches depending on the value of some variables.

## 6.2 Profiling trend analysis

| | software | optimized hardware |
|---|---|---|
| CPU cycles | ~44 millions | ~10k |
| CPU stall cycles | ~26 millions | ~2k |
| BUS idle cycles | ~23 millions | ~5k |
| **Execution time** | **~0.6 seconds** | **~240 microseconds** |
| CPU waiting time | ~60% | ~20% |
| BUS busy time | ~48% | ~50% |

Table 4: Comparison between software vs. optimized hardware solution

# 7 Possible improvements

During the development of our project and solutions, other ideas came to mind that could be implemented to further improve the system.

## 7.1 Tracking without visualization

The current implementation was conceived by having as one of the fundamental key points the will of keeping the camera feed visualization, in which the image is grayscaled with the exception of the detected object, whose color is maintained in its original form.
To achieve an even faster system, in which main the objective would be the movement and tracking of the pan-tilt system, the **camera feed could be eliminated** and all computations could be done on stream as soon as a frame is received.

## 7.2 Ping-pong optimization

If we want to maintain the camera visualization and improve the performances, another idea we had was to use a **ping-pong strategy** for the final averaging. By adding another buffer memory to the camera module, we could start the DMA transfer of the next frame during the software computation. The CPU will not interfere with the DMA transfer as it has to access the memory only once to write the white pixel at the center of the object.
However, this further optimization would not generate a big gain in performances, as the number of cycles used for the final averaging is small compared to the cycles needed for the transfer of the frame.

## 7.3 Improve tracking stability

As it was previously discussed, extended connection between the camera and board was necessary to enhance the camera's freedom of movement. The current solution consists in the use of regular jumpers. A major drawback is the increase in resistance and capacitance due to the wires. This aspect, together with some slightly loose contact at the pins level, leads sometimes to a failure in a reliable transportation of the signal from the camera to the FPGA.
Therefore, better wires and connectors could be leveraged.

# 8 Conclusions

This project allowed to get a deep knowledge on all the architectural levels of an embedded system, starting from the lowest level of logic gates to the higher level of C code.
This knowledge allowed to spot and understand the limitations of an implemented solution and to perform optimizations to it.
An important takeaway is the need of thinking in advance about all the consequences of an implementation choice, as the latter will surely have an impact on other parts of the system. A key aspect is the delicate tradeoff between the performances of the system and the use of the available hardware.
It is therefore important to have clear the priorities the project has, that could range for example from highest possible speed to lowest FPGA chip consumption.