AEP Summer 15: Final report

Shared Robo-Vision

Team: Alberto Vaccari, Dapeng Lan and Yu Liu.

Introduction

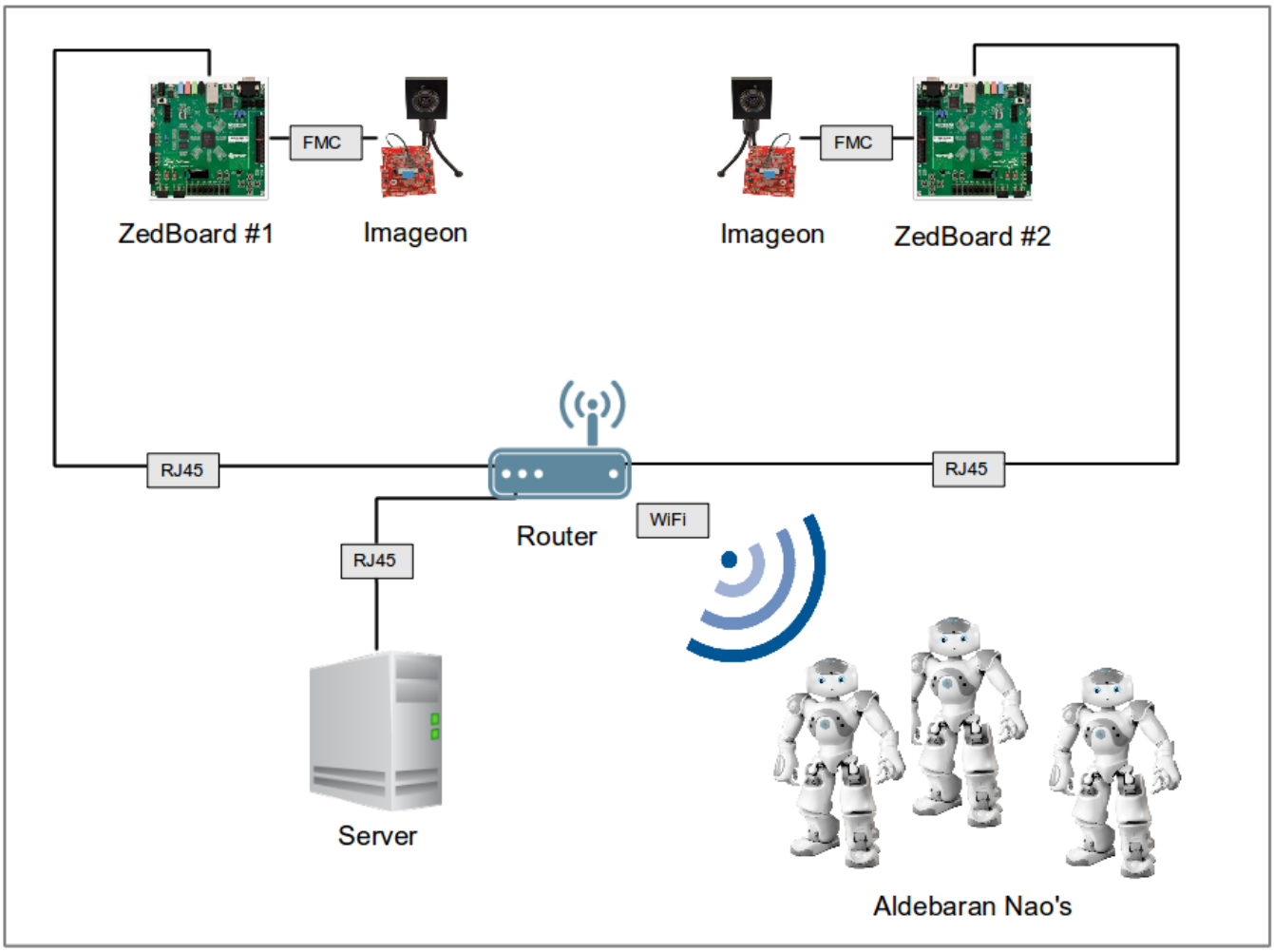
RoboCup is an annual international robotics competition with the aim to promote robotics and AI research.

The AOT department of TU-Berlin has been working and participating in this competition since 2003.
After discussing with some people involved in the project, an interest in a “tool for logging matches and the behaviour of the robots” was shown.
This project was chosen as a way to improve their workflow and support their useful contribution in the field of robotics and artificial intelligence.

The main idea is to position 2 cameras overseeing the 2 sides of the football pitch, where the match will be held. Each robot will have a marker on their head to identify them, also used for tracking.
The 2 camera will be connected to a server for tracking and logging. The server will also be able to communicate with the robots through the network.

The operator will be able to see the current position of the ball and of each robot in the field, record the match and directly control a specific robot.

System Diagram



Objectives

The objectives of the project are the following:

- Transmit HD images (1920*1080) from 2 ZedBoards to a computer
- Perform some degree of image processing directly on the ZedBoards
- Track the position of the robots on the field
- Track the position of the ball on the field
- Record a match
- Replay a recorded match
- Control the movements of the robots from the server application

Roles

Alberto:

- Tracking of the robots
- Tracking of the ball
- Recording a match
- Replaying a saved match
- Communication and control of the robots
- Writing progress and final reports
- Preparing Demo and Presentation

Dapeng:

- Fetching HD images from FMC-Imageon V2000C camera
- Transmission of HD images from 2 ZedBoards to the server
- Image pre-processing on the ZedBoards

Yu:

- Receiving images from the ZedBoard
- Image pre-processing on the ZedBoards

Current Status

Here is the current status of the tasks:

- ☑ Tracking of the robots
- ☑ Tracking of the ball
- ☑ Recording a match
- ☑ Replaying a saved match
- ☑ Communication and control of the robots
- ☑ Fetching HD images from FMC-Imageon V2000C camera
- ☐ Transmission of HD images from 2 ZedBoards to the server
- ☑ Image pre-processing on the ZedBoards
- ☑ Receiving images from the ZedBoard

Completed | Partially Completed | Started | Not Started

Image Processing – Overview

This section covers the image processing, and the reasons behind some of the choices made.

Initially, it was thought that using already existing patterns on the robots (Aldebaran Nao's) for their localization would be the best choice.

By simply colouring the almost semi-circular front area on the head of the Nao, it is possible to get both orientation and rotation of the robot in the field.

Image 1: Original

Image 2: With coloured front, used for testing

Locating the shape was a fairly easy task, although calculating its difference in rotation from the template shape ended up being a fairly challenging task, as OpenCV doesn't provide rotation information when performing a template matching (searching an image template within another image).

Since the task was not only to detect the position and the rotation of each robot on the field but also to identify it, the suggestion of looking into ArUco markers was followed.

Image 3: Template for shape

ArUco markers have been created to facilitate Augmented Reality applications, providing an easy and light-weight library for detecting 2D markers and returning their 3D coordinates in the real world. Even if they were initially created for AR projects, their use rapidly extended to robotics and computer vision ones as well.

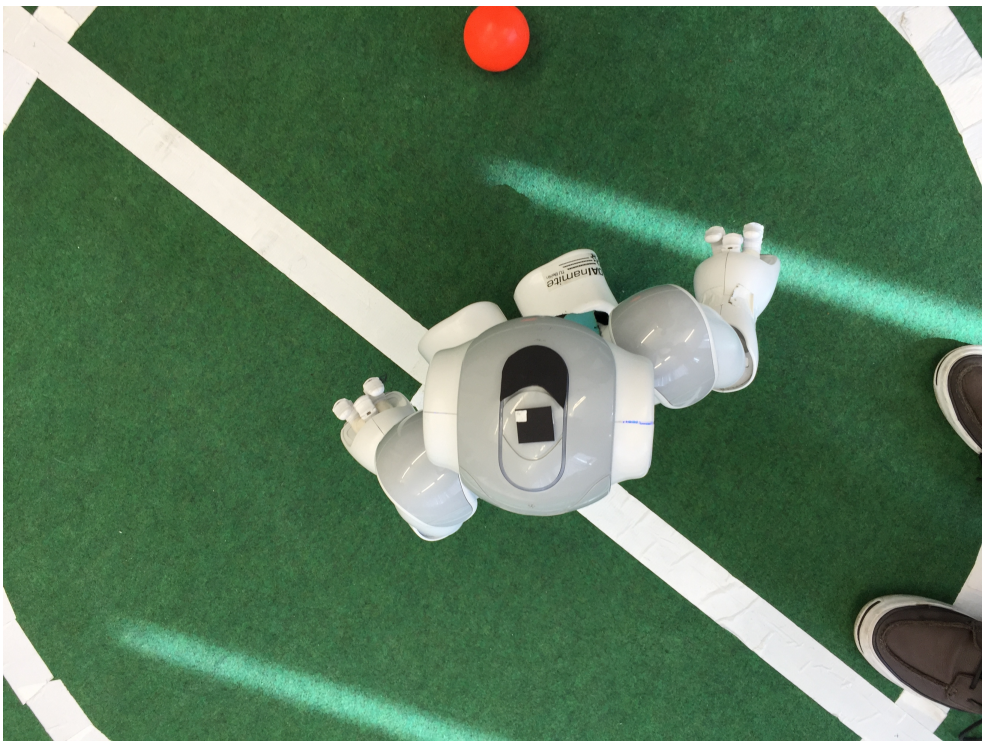


Image 4: Proposed, later discarded, solution for the robot identification

ArUco markers have the characteristic to also contain an ID, meaning that a detected marker contains its 3D coordinates
(in regards to the camera) and an identification number, making it easy to map an ID to a specific robot.

The component locating the ball relies on the assumption that the ball has a regular (by rules) and constant size and it is red in colour.

The following section will cover the components performing image processing, presenting and explaining their workflow for a better understanding.

Image 5: Example of ArUco marker

Image Processing – System Workflow

1. Main.cpp

– Clean 'field' image

– Fetch image from Left camera

– Fetch image from Right camera

– Make a copy of the camera images (for showing the unchanged version as stream, for debugging)

On both images:

- Detect ArUco markers (***Markerdetector.cpp**)
- Detect ball (***BallFinder.cpp**)
- Draw ball on the 'field' image
- Go through each marker found
- Get center and rotation of each marker
- Retrieve the team of the current player (dictionary lookup)
- Retrieve the team colour and the player's number
- Calculate and draw the player on the field
- If a movement command has been issued to a specific robot:
    * Check if target location has been reached
    * If not: show target location and line connecting it to the player
    * Send command to robot, if it has not already been sent

2. Markerdetector.cpp

– If image is not a grayscale (8UC1), convert it to grayscale (BGR2GRAY)

– Threshold image:

> *# Using an adaptive threshold (ADAPTIVE_THRESHOLD_MEAN_C): the threshold value is the mean of the block neighborhood minus a given C value.*

– Detect all the rectangles in the image:

- Calculate min/max size of a valid contour
- Find contours:

    # Get absolutely all the contour points (no compression/approximation)
- Check that each contour found (e.g. candidate markers):
    - Has 4 sizes (points)
    - Is convex
    - Its min size is greather than 1x10^-10
    - Its min size is greather than 10 (remove points that are too close to each other)
- Sort the points of each contour in anti-clockwise order
- Remove candidates that are too small (checking the perimeter)
- Identify each marker (get its ID) (***Arucofidmarkers.cpp**)
- Remove duplicates (if any)

3. Arucofidmarkers.cpp

– If image is not a gray-scale (8UC1), convert it to gray-scale (BGR2GRAY)

– Threshold image:

 # Uses a binary threshold (THRESH_BINARY) together with the Otsu's algorithm (THRESH_OTSU) to determine the optimal threshold value

– Check if border is black

 # Markers are 7x7 blocks with 1 black block as border (quiet zone)

– Check if inner blocks are black or white

– Check every possible rotation

 # Get the hamming distance to the nearest possible word (calculating the minimum number of errors that could have transformed a word into another)

– Convert bits (black/white) into an integer

4. BallFinder.cpp

– Convert image to HSV (Hue, Saturation, Value)
– Threshold the image
    # *Removing anything not red*
– Remove small objects from the foreground
    # *Morphological opening (erode-dilate)*
– Fill small holes in the foreground
    # *Morphological closing (dilate-erode)*
– Apply Gaussian blur (to avoid false circle detection)
– Find circles within a certain range
    # *Using HoughCircles*

5. Using the ZEDBOARD to do the pre-image processing

In this part of design, the goal is to read the HD video stream data from FMC imageon module and store the stream data in the DDR. Then using the steam data to do some hardware processing. And finally output the video stream through HDMI to the screen.

Image 6. FMC-IMAGEON

Hardware Structure

-The configuration in Vivado

The top system in Vivado as show below:

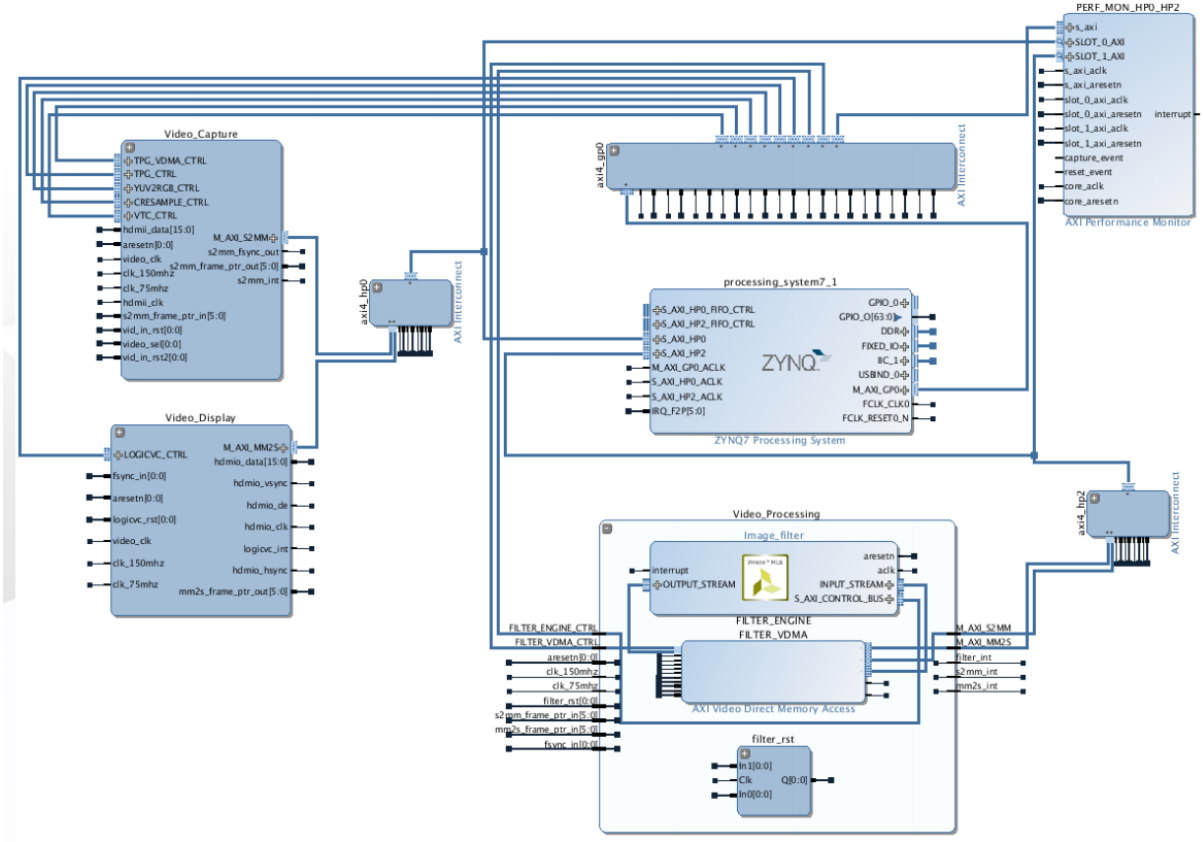


Image 7. Top system in Vivado

The **Video_Capture** part as show below:

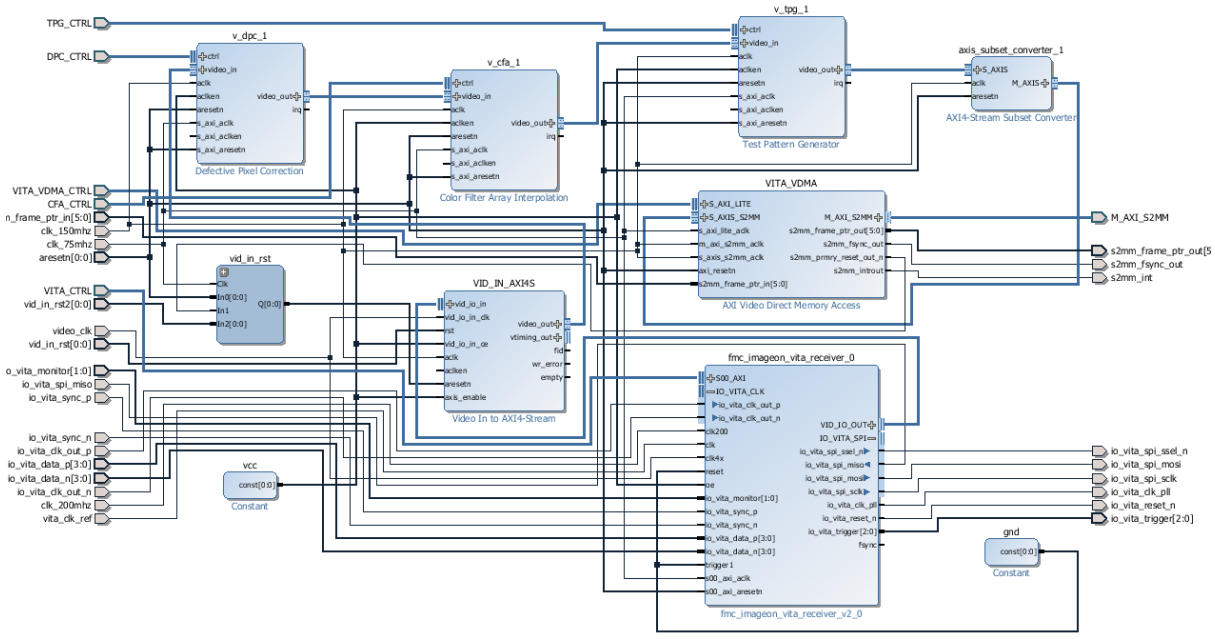


Image 8. Video_Capture part in Vivado

The **Video_Process** part show as below:

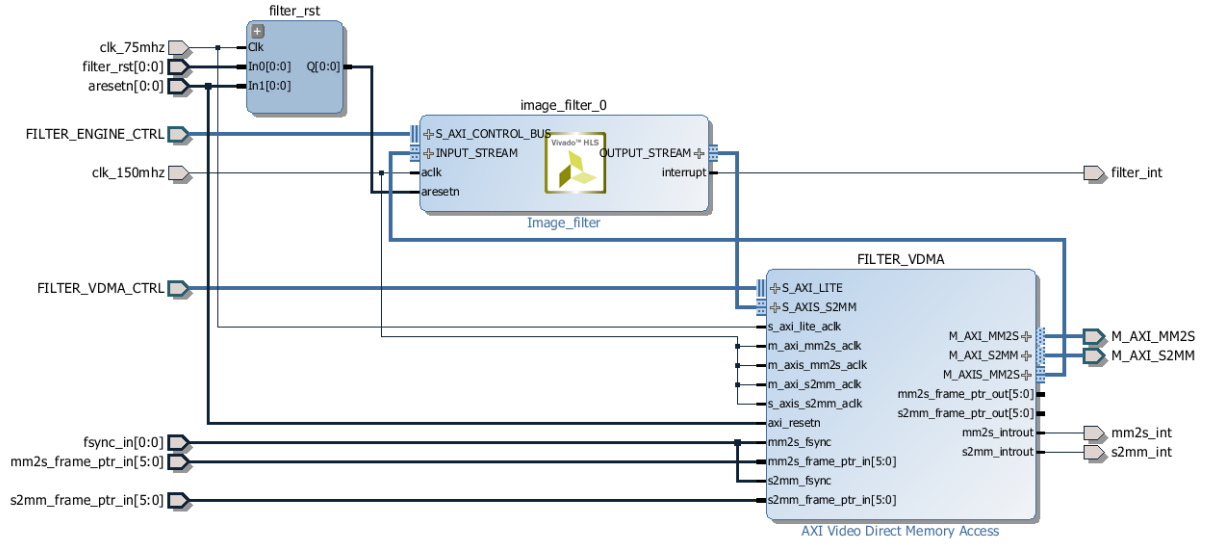


Image 9. Video_Process part in Vivado

-Hardware Process

Video Processing Components from Xilinx generally use a common AXI4 Streaming protocol to communicate pixel data. This protocol describes each line of video pixels as an AXI4 packet, with the last pixel of each line marked with the TLAST signal asserted. In addition, the start of the video frame is indicated by marking the first pixel of the first line with the USER[0] bit asserted.

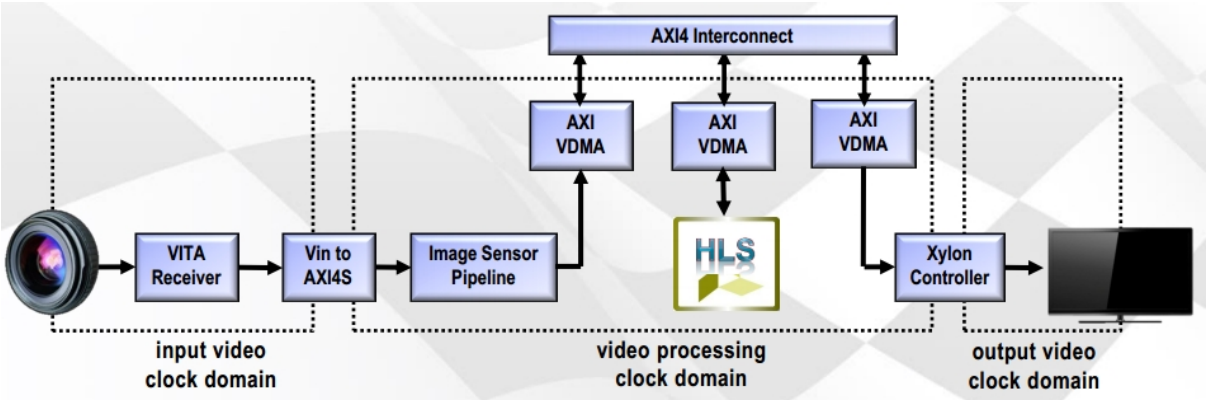


Image 10. Video Processing Designs

This structure refer to as “frame-buffer streaming”, pixel data is first stored in external memory before being processed and stored again in external memory. A video display controller is then required to output the processed video. A frame-buffer streaming architecture allows more decoupling between the video rate and the processing speed of the video component, but it requires enough memory bandwidth to read and write the video frames into external memory.

- AXI4-Lite for processor control of various IP cores (not shown): **50 MHz**
- AXI4-Stream for point-to-point video connections :**148.5MHz**
- AXI4 for high-performance access to external memory

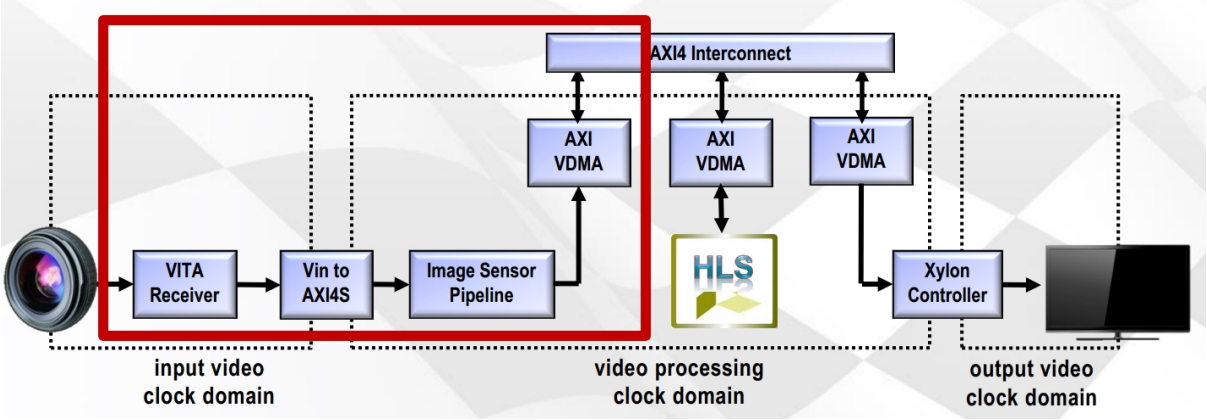


Image 11. Video Input Designs

The function of this part is:

- Add FMC-IMAGEON cores to IP Catalog
- Create Video Pipeline
    - o VITA Receiver core + Video to AXI4-Stream bridge
    - o Image Processing cores : DPC, CFA
    - o AXI Video DMA

Video Input image process

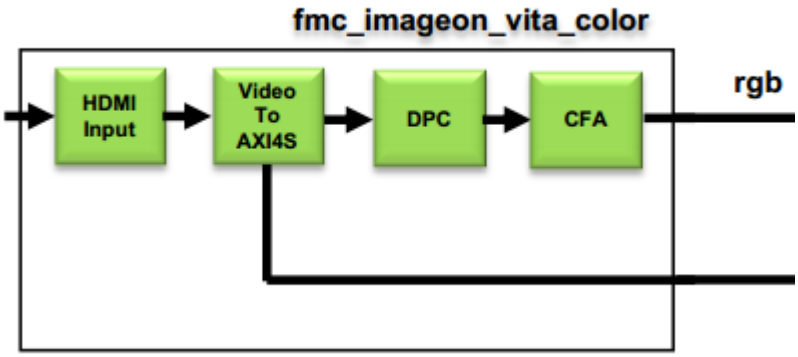


Image 12. Video Input image process

DPC: Defect Pixel Correction

Image Sensors are manufactured with defect pixels: Defective pixels are typical for image sensors. Image sensor manufacturers tolerate certain types of defective pixels, assuming that they can be corrected downstream by an image processing pipeline.

CFA: Color Filter Interpolation

Most widespread and cost effective implementation involves placing color filters on top of each pixel in a Bayer pattern arrangement. With this arrangement of color filters, each pixel captures only one of the three primary colors. The Color Filter Array interpolation (CFA) core restores the missing two colours based on neighbour pixels.

-Hardware Process

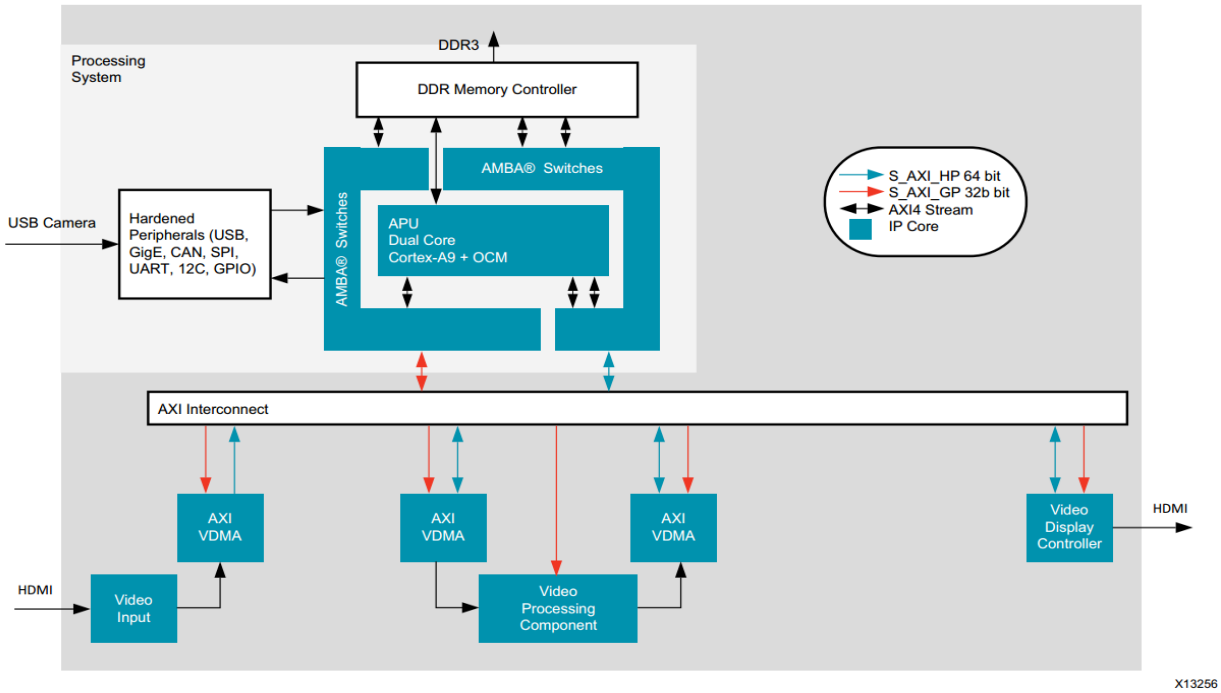


Image 13. Video Processing Designs

This structure refer to as “frame-buffer streaming”, pixel data is first stored in external memory before being processed and stored again in external memory. A video display controller is then required to output the processed video. A frame-buffer streaming architecture allows more decoupling between the video rate and the processing speed of the video component, but it requires enough memory bandwidth to read and write the video frames into external memory.

Video Processing in Vivado HLS

Vivado HLS contains a number of video libraries, intended to make it easier for you to build a variety of video processing. These libraries are implemented as synthesizable C++ code and roughly correspond to video processing functions and data structures implemented in OpenCV. Many of the video concepts and abstractions are very similar to concepts and abstractions in OpenCV. In

particular, many of the functions in the OpenCV imgproc module have corresponding Vivado HLS library functions.

For instance:

OpenCV	HLS Video Library
cv::Point_<T>, CvPoint	hls::Point_<T>, hls::Point
cv::Size_<T>, CvSize	hls::Size_<T>, hls::Size
cv::Rect_<T>, CvRect	hls::Rect_<T>, hls::Rect
cv::Scalar_<T>, CvScalar	hls::Scalar<N, T>
cv::Mat, IplImage, CvMat	hls::Mat<ROWS, COLS, T>
cv::Mat mat(rows, cols, CV_8UC3);	hls::Mat<ROWS, COLS, HLS_8UC3> mat (rows, cols);
IplImage* img = cvCreateImage(cvSize(cols,rows), IPL_DEPTH_8U, 3);	hls::Mat<ROWS, COLS, HLS_8UC3> img (rows, cols);
	hls::Mat<ROWS, COLS, HLS_8UC3> img;
	hls::Window<ROWS, COLS, T>
	hls::LineBuffer<ROWS, COLS, T>

Image 14: OpenCV correspond to Vivado HLS library functions

In our implementation, we implement following functions:

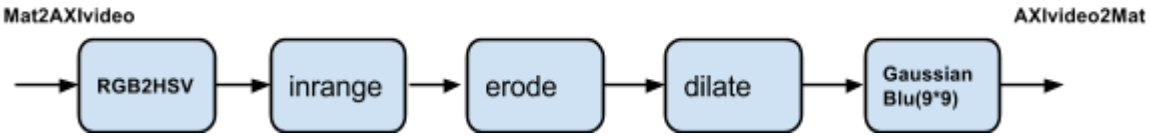


Image 15: HLS functions

RGB to HSV conversion algorithm as follow: (code see in the top.cpp file)

$$MAX = \max(Red, Green, Blue)$$

$$MIN = \min(Red, Green, Blue)$$

$$H = \begin{cases} 0 & if \quad MAX = MIN \\ 42 \times \dfrac{Green - Blue}{MAX - MIN} + 42 & if \quad MAX = Red \\ 42 \times \dfrac{Blue - Red}{MAX - MIN} + 127 & if \quad MAX = Green \\ 42 \times \dfrac{Red - Green}{MAX - MIN} + 213 & if \quad MAX = Blue \end{cases}$$

$$S = \begin{cases} 0 & if \quad MAX = 0 \\ 255 \times \dfrac{MAX - MIN}{MAX} & Otherwise \end{cases}$$

$$V = MAX$$

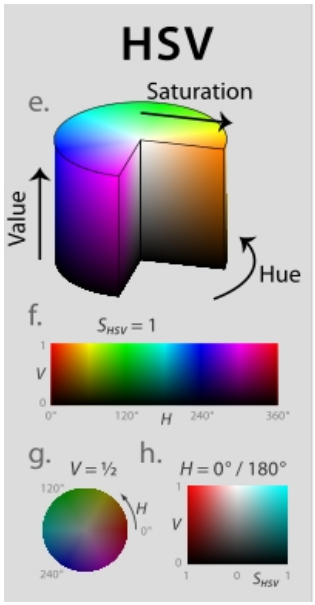


Image 16. the HLS C code of the RGB to HSV conversion

div_tab is an array of 256 values (div_tab[0] is set to 0 since it doesn't affect the results) which stores the division results of 1/i for i = 1, …, 255 with a precision of 16 bits.

```
dRG = Red − Green;
dGB = Green − Blue;
dBR = Blue − Red;
min_t = MIN(Red, Green);
max_t = MAX(Red, Green);
mini = MIN(Blue, min_t);
maxi = MAX(Blue, max_t);
diff = maxi − mini;
div_m = 42 * div_tab[diff];
ddRG = div_m * dRG;
ddGB = div_m * dGB;
ddBR = div_m * dBR;
if (maxi == mini) H = 0;
else if(maxi == Red) H = (ddGB ≫ 16) + 42;
else if(maxi == Green) H = (ddBR ≫ 16) + 127;
else H = (ddRG ≫ 16) + 213;
if (maxi == 0) S = 0;
else S = (255 * diff * div_tab[maxi]) ≫ 16;
V = maxi;
```

Image 17. HLS C code for RGB to HSV conversion.

We compare the HSV values of the pixel with an input value (Hin) chosen by the user. If H is close enough to that input value and S and V are big enough, then the pixel will be tracked. The boundary information is obtained depending on the color (Hin) selected by the user (see Fig. 18).

```
unsigned int xmin_temp = cols − 1;
unsigned int xmax_temp = 0;
unsigned int ymin_temp = rows − 1;
unsigned int ymax_temp = 0;
temp = H − Hin;
if(S > 63 &V > 127){
   if (ABS(temp) < Threshold){
      if (i < xmin_temp) xmin_temp = i;
      if (i > xmax_temp) xmax_temp = i;
      if (j < ymin_temp) ymin_temp = j;
      if (j > ymax_temp) ymax_temp = j;}
}
```

Image 18. HLS C code for display of positions of object.

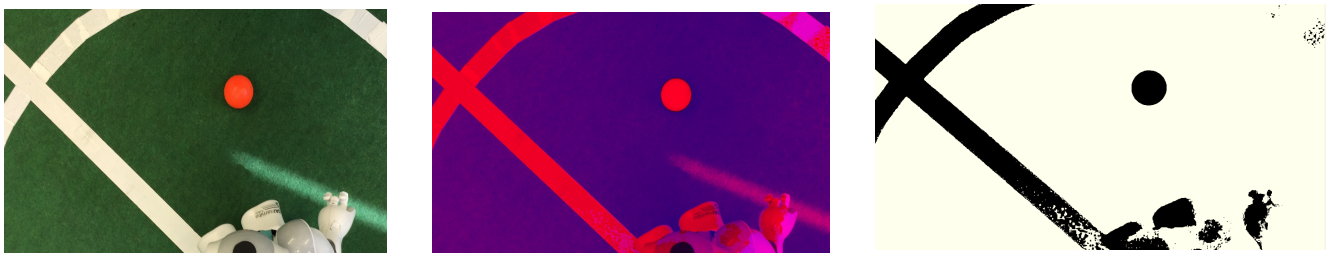


Image 19. Orgin(left), RGB2HSV(middle), Hin threshold(right)

Erode:

The basic idea of erosion is like soil erosion only. Simply white region decreases in the image. It is useful for removing small white noises.

Dilate:

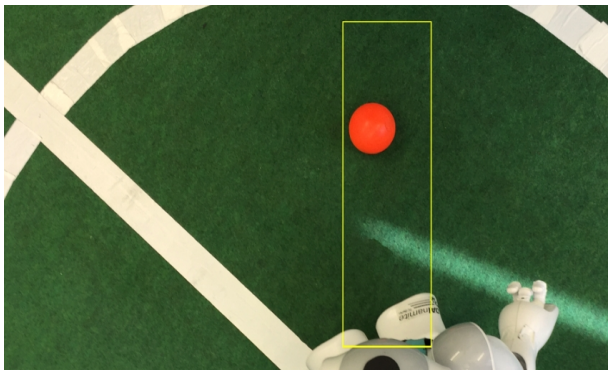
It is just opposite of erosion. It increases the white region in the. Normally, in cases like noise removal, erosion is followed by dilation. Because, erosion removes white noises, but it also shrinks our object. So we dilate it.

GaussianBlur

Gaussian blur to reduce image noise and reduce detail.

HoughCircle(To be done)

To find the circle in the image.



The result of HLS synthesized- **change rgb to hsv, do the Hin threshold.** It can seen from the form that it use 19% of the LUT.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	8
FIFO	0	-	95	540
Instance	-	36	7327	9817
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	25	-
ShiftMemory	-	-	-	-
Total	0	36	7447	10365
Available	280	220	106400	53200
Utilization (%)	0	16	6	19

Image 19 Utilisation Estimates

Building UDP client and server

We tried to use UDP protocol for image transmitting and receiving and managed to build a UDP client and server to realize this function and test it on PC. The client can read videos from webcam or images from file and then fragment them into several packets and send them through UDP. The server part can receive fragmented packets, reassembly them and show the original videos or images. We also set frame checking algorithm to make sure the integrity of the transmitted videos and images.

Running this part of the projects

The detail of the program and execution steps can be found in the project folder README.txt.

Reference:

[1] C. Desmouliers, E. Oruklu, S. Aslan, J. Saniie F. Martinez Vallina and F. Martinez Vallina Image and Video Processing Platform for FPGAs Using High-Level Synthesis.