ECE532

Group 10 Final Report

Nicholas Cusimano

Jia Le (Gallop) Fan

Wen Jie (Jeff) Feng

Project: Environmentally Aware Camera System on Vehicle

Table of Contents

Introduction

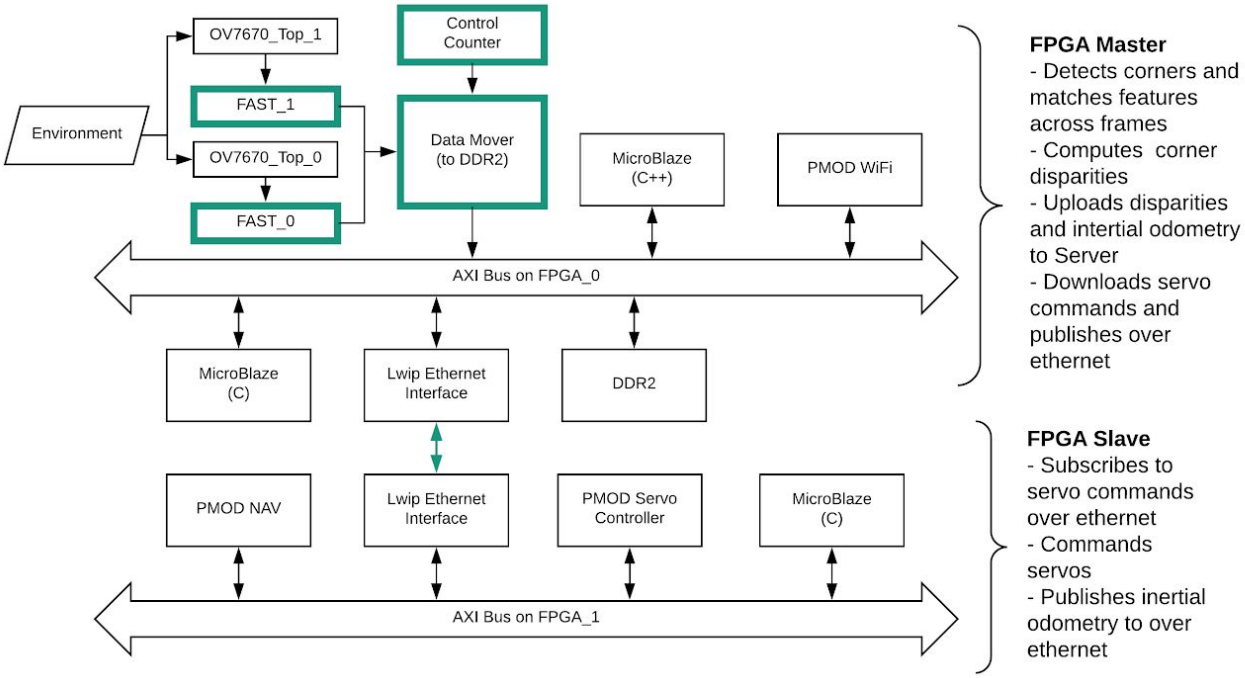
Autonomous path planning requires environment detection; common sensors for this task are: range sensing (eg. LIDAR), machine vision, GPS, kinematic odometry, and IMU. Of these sensors, our team believes machine vision performs the best. Range sensing can be sensitive to electromagnetic interference. GPS is unavailable underground, underwater, in the air, and sometimes unavailable indoors as well. Kinematic odometry suffers from wheel slip; IMU measurements suffer from drift [1].

Researched by companies like Google, Tesla, and Uber, autonomous path planning is far from being a solved problem; Google and Uber self-driving car projects use (among other sensors) LIDAR which may become impractical. If Google's or Uber's projects are scaled up for the general population, each self-driving cars' LIDAR could potentially drown out each other cars' signals; sensor engineers will have to solve this interference problem[2]. A machine vision solution to detecting the environment is required.

Our team anticipates that not only will FPGAs be integrated into present and future imaging systems but also that FPGAs will be reconfigured depending on the imaging environment; this is because different feature detectors perform better in different environments. For example, edge and corner detectors work well in man-made environments but suffer in natural environments because the latter environments intrinsically lack straight lines. Similarly, blob detectors work well in natural environments; this is why projects like the Mars Exploration rovers use blob detectors rather than edge detectors [3]. We envision that future vehicles will travel in both man-made and natural environments. FPGAs are suitable for this purpose because they can dynamically optimize their performance in each of these environments.

System Overview

The team initially proposed to create a robot which can detect a feature disparity map. This map can then be uploaded to a remote server which was running a navigation stack. These details are outlined in Appendix A. The following block diagram shows the proposed system:



We proposed using two nexys boards because it was determined that one board could not support all of the necessary PMOD devices for the project. We thought of using the Nexys Video board and making the FMC connector accessible through a custom PCB, however we decided not to pursue this idea due to the risk in lead time and possible failure. We came to these conclusions after being inspired by this blog post [4] and later this two-part paper: Davide Scaramuzza's Tutorial to Visual Odometry [5] published in the IEEE Robotics and Automation Magazine.

By the end of the project, the system consisted of the following features:

FAST Algorithm - Overview

Features from Accelerated Segment Test, originally invented by Edward Rosten [6], has been reproduced in various tools such as openCV and its variants; source code is readily available in a variety of wrappers and even in machine code. Originally our team was planning to use PutVision Lab's version [7] of the FAST Corner Detector in the design; however, our team chose to reimplement their architecture because it would be a good exercise in hardware programming. As one might expect, the paper did not contain every detail; in fact, there were even errors in their diagrams (on the free web sources we could find). A Xilinx Application Note (XAPP953) on a rank filter, one post of a multi-part blog post from element14 [8] on building a

gradient filter and clues from an assignment handout from Berkeley [9] on building a median filter were helpful for HDL coding.

This algorithm, described in PutVision Lab's implementation paper [7], is used for corner detection. This is the main custom IP block that the team worked on. The team made custom IP for this algorithm based off of the remarks and diagrams in the PutVision Lab's implementation paper. The output of the fast algorithm is a map containing the coordinates of all of the detected corners in a frame. This corner map can then be compared against the maps of consecutive frames to determine corner disparities and thus navigational information. Note that this IP block is not running within a microblaze processor; all of the counters, buffers, and comparators are implemented externally.

PMOD Servo Controller - Overview

This IP block is responsible for driving a set of continuous rotation servo motors to enable robot movement.

OV7670 - Overview

The original OV7670 module was modified in the earlier milestone to output for grayscale pixels. This was done by making a seperate latch to capture the Y pixel in the YCrCb protocol and by providing it at the output every other pixel clock cycle. Later on when VDMA was considered for the final design, additional control signals were added in order to properly communicate with the VDMA ip block.

Frame buffer (originally VDMA) - Overview

The VDMA was tested to be functional in loopback mode, but was never integrated into the final design. Since our simulation testing used bitmap images instead for testing, a hardware implementation that allowed for switching between these bitmap images was instead implemented.

The frame buffer was instantiated in BRAM with enough address space to hold 3 separate 320 x 240 grayscale bitmap images. Switches on the FPGA board provides an interface to switch between the images, and the images are displayed on the VGA after the FAST algorithm is applied.

Outcome

The following features from the proposed diagram were not implemented:

- PMOD Wifi
    - There was some initial work done on it to verify connectivity but further work was minimal
- PMOD Nav (IMU)
    - Development was cancelled due to other components being more critical for functionality (i.e. the FAST algorithm)
- LwIP interface
    - Because we didn't implement the IMU, and we were focusing on getting FAST to work on a single camera on a non-moving robot, we did not see the need to develop a communication interface between two boards.

The project does not work as intended. In the end, there was not a wholly integrated system. The FAST algorithm was near integration and completion, but took longer than expected to integrate and test. It can currently detect corners, but has imperfections across different sample images. It was also never tested with live camera input, as we were focusing on getting to get the algorithm to work with images before video.

Suggestions for Future Development

To improve the system, we would have based our implementation of corner detection off of a readily available software. Ideally, this software would be discretized into different modules for gradual hardware implementation. A proven software would help to verify concerns and problems with our own IP.

If we had the opportunity to start over, we would have first attempted to have a working version of FAST before investigating the IMU, WiFi and Servo IP. This would have given the team 2 - 3 extra weeks in the beginning of the timeline to relieve any concerns/confusion of FAST. Development of the FAST algorithm should have been based off of multiple sources, such as open source, verifiable software, rather than a couple of research papers.

If the project were to be taken over, we do not recommend discarding the existing FAST implementation, but rather comparing the FAST modules against proven FAST implementations to determine the problem. After FAST is working, we suggest that the developers implement feature matching and disparity mapping for a single camera peripheral, rather than working on PMOD devices. This work can be based off of already existing, open source implementations. Essentially, the most important/complex tasks should be given the highest priority and the largest schedule allocation. The whole group should be working on getting the complex parts to work - one or two people can focus on developing new IP, and the other group member(s) can focus on interpreting open source software for the purpose of this project.

Project Schedule

The proposed schedule is in Appendix B. The actual schedule (assembled from the milestone reports) is shown below:

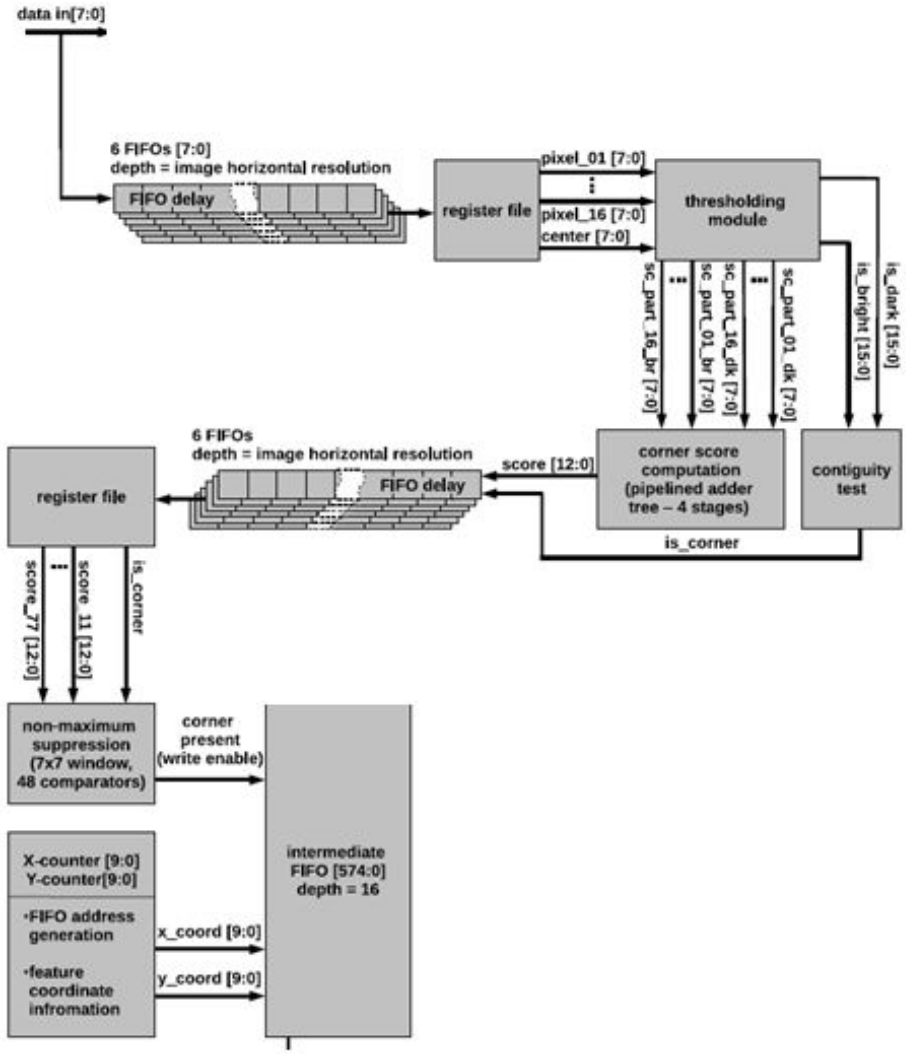
Milestone	Task Description/Status	Person Responsible
M2	OV7670 input displayed on VGA monitor	Jeff
	Basic server code written (to confirm connectivity aspect of WiFi module)	Jeff
	Servo IP written - motors can be driven in both directions	Nick
	WiFi and NAV PMOD tested successfully, but not integrated	Gallop
M3	Parse greyscale from video feed	Jeff
	Integrate camera IP with Lwip stack	Jeff
	RTL written for FAST modules	Gallop
M4	Simulation and verification of threshold module (FAST algorithm)	Jeff
	Simulation and verification of Circle pixel buffer and NMS buffer modules (FAST algorithm)	Nick
	Simulation and verification of the NMS block, contiguity module, and corner score module	Gallop
M5	Block diagram with VDMA has been made - currently under test	Jeff
M6	Bitmap image test bench set up	Nick and Gallop
	set up VGA and confirmed working with interface that can connect to M_AXIS_STREAM out port	Jeff
	Module integration of the FAST algorithm - under test	Gallop
M7	FAST is outputting corner data with the test bench, but the output data is not exactly correct	Nick and Gallop
	VDMA loopback works, could not connect with FAST algorithm	Jeff

It is evident that the actual schedule diverged from the proposed schedule at an early stage in the project. This is likely attributed to the lack of understanding of certain technologies involved in the project (FAST, VDMA, etc). In addition, the assignments, which were not accounted for in the proposed schedule, caused delays towards the final milestones. Most of the PMOD development was abandoned at an early stage of the project. However, even the initial PMOD development fell behind schedule. The proposed schedule highly underestimated the time required to fully implement and verify the FAST algorithm.

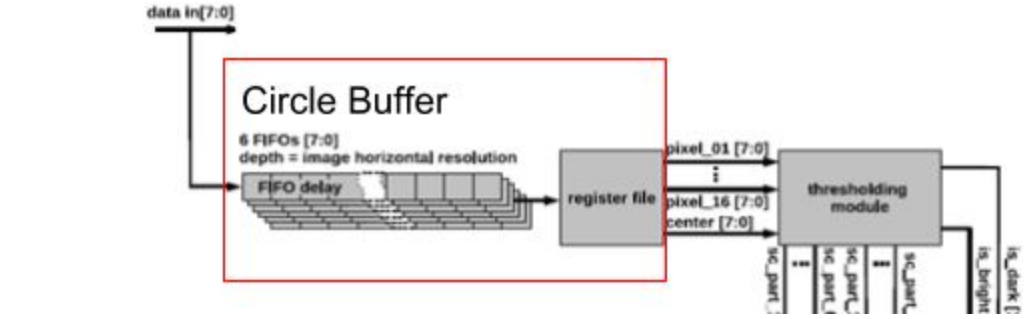
Design Description

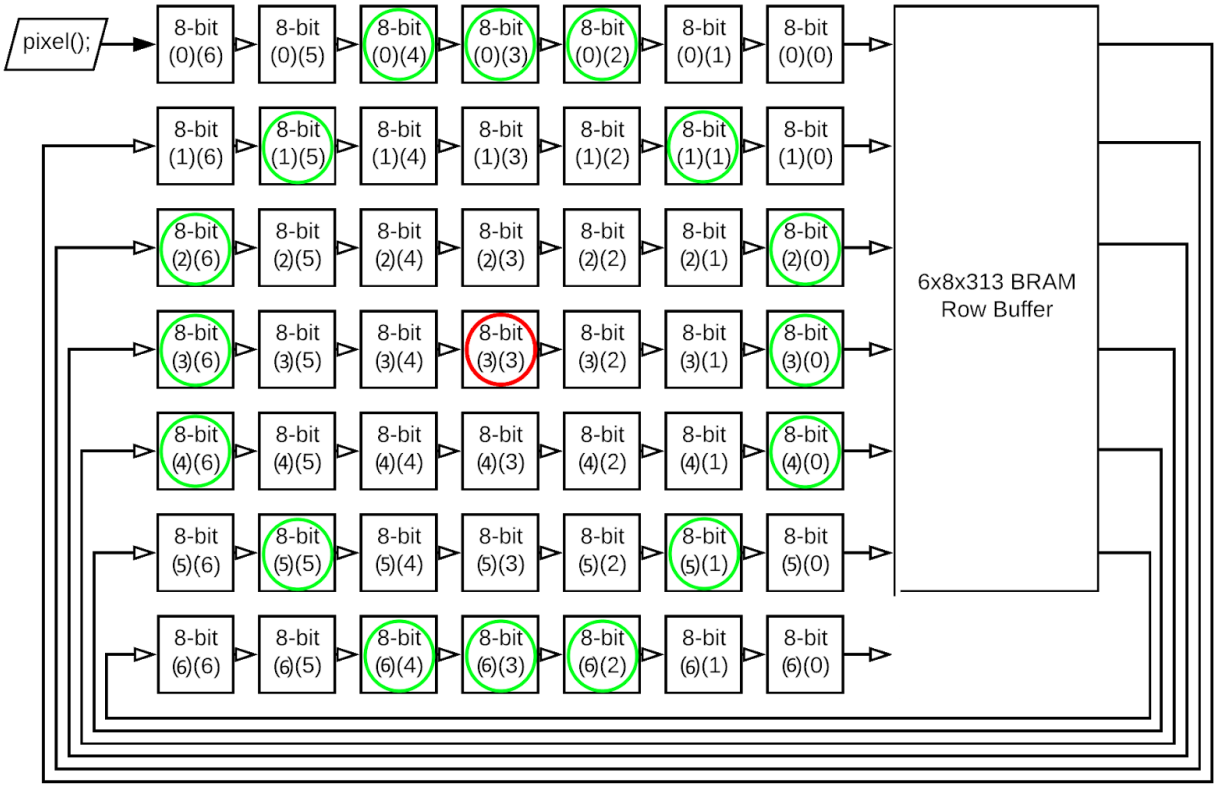
FAST algorithm

The diagram of the FAST datapath is shown below, and repeated in Appendix C. This is what the PutVision Lab's implementation used [7]. The following sections will describe all of the elements of the datapath.

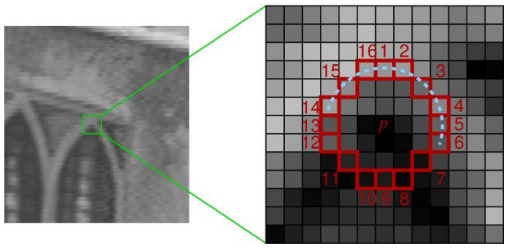


Circle Pixel Buffer

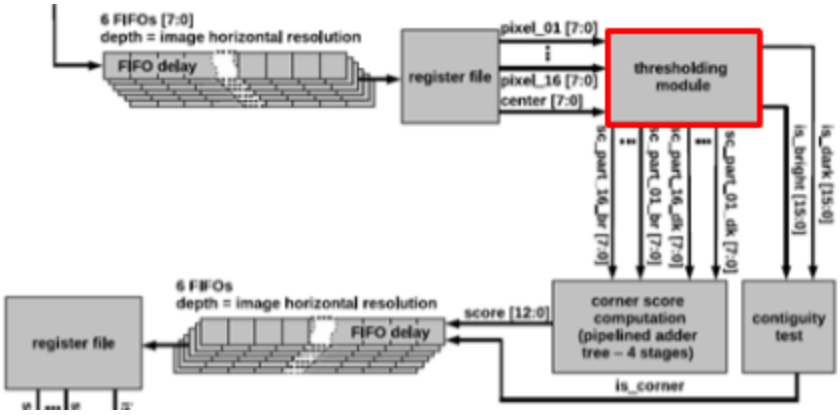




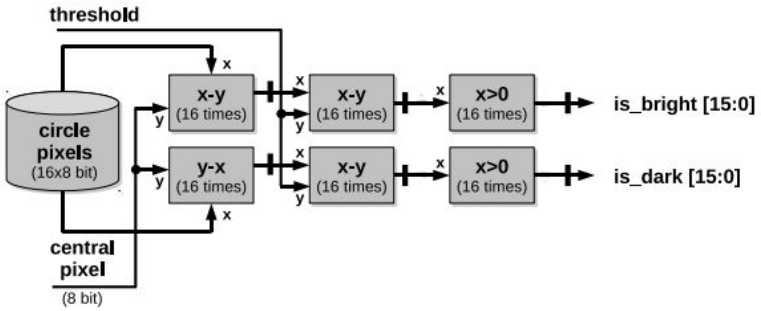
This module buffers a serial datastream such that multiple rows of an image can be viewed at once. Specifically, the initial set of FIFO buffers + register file have an input of raw image data (which comes in by row). It is buffered into a 7 x 7 pixel window of the image. This is necessary to have a circle of pixels to compare the intensity around a central pixel. This circle is called the bresenham circle. An example of the bresenham circle is shown below. The output of this module is the intensity values of the 16 points of the bresenham circle and the center pixel. Having this output for each clock cycle is necessary to easily calculate the corner score of a specific pixel.



Thresholder

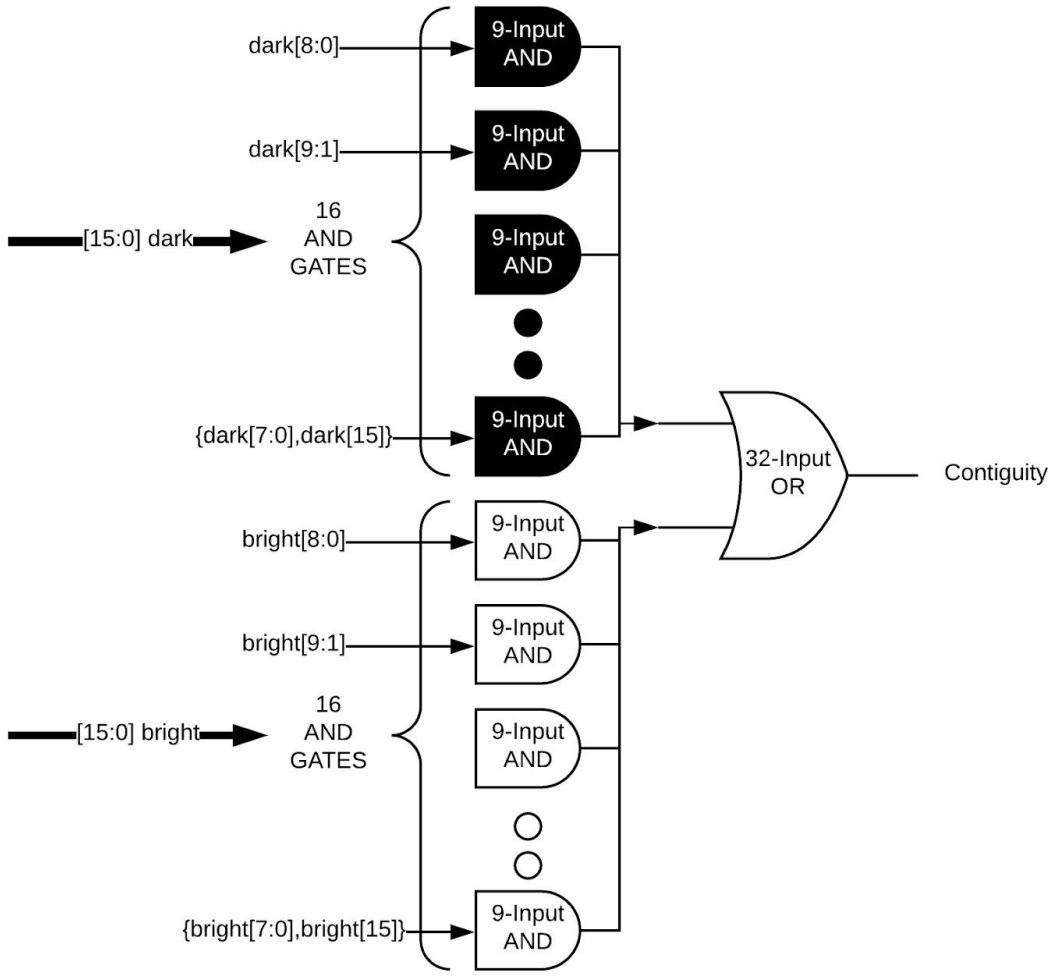


The purpose of the thresholder module is to compute the relative *brightness* or *darkness* of each of the bresenham circle pixels with the center pixel. The intensities are subtracted, and then this result is subtracted by a threshold value, according to the following block diagram:



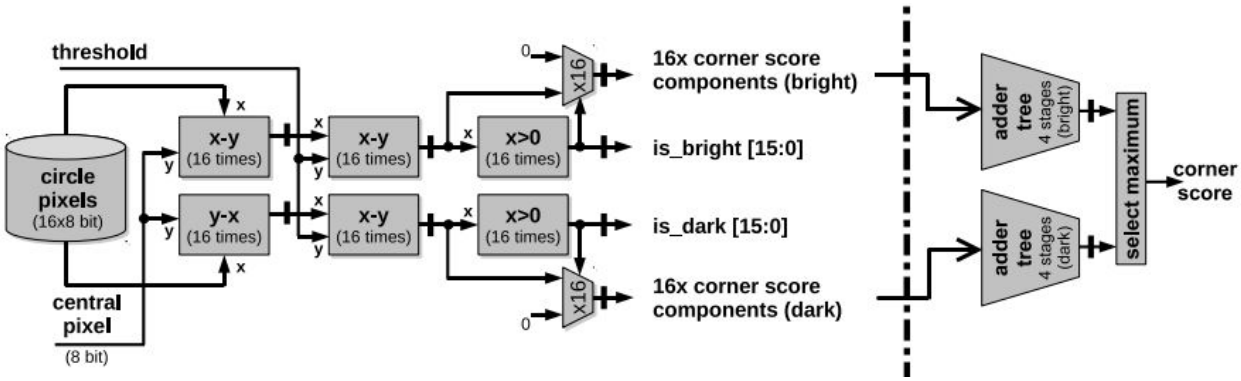
Contiguity Test

The contiguity module tests for uninterrupted segments on the aforementionned bresenham circle; if there are 9 or more consecutive circle pixels that are all dark or all bright, then the center pixel passes the contiguity test.

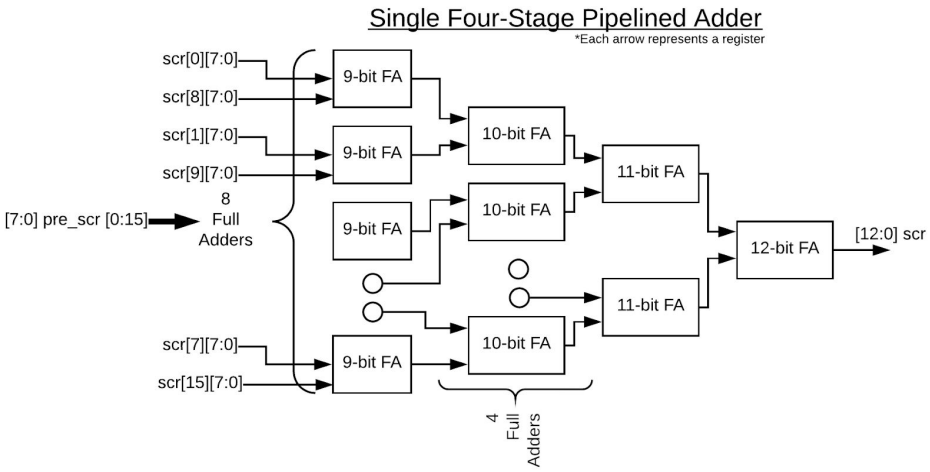


Corner Score Computation

Corner score computation is done with score components generated by the thresholder module; the corner score represents the Sum of All Differences between the center pixel and each circle pixel. The corner score components need to be summed together to get the corner score.



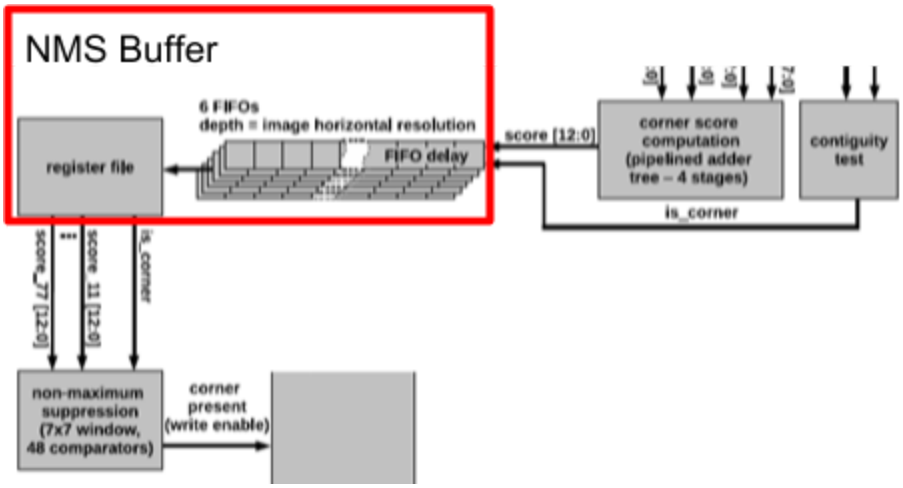
The adder stage takes sixteen 13-bit score components and adds them all together in 4 stages.



A pixel with a higher corner score than all its neighbours means that this pixel has the most variance in intensity in its vicinity; this means that the pixel's neighbours have a lower intensity variance in its vicinity. These corner scores allow a non-max-suppression to be applied such that only the best possible corners are chosen.

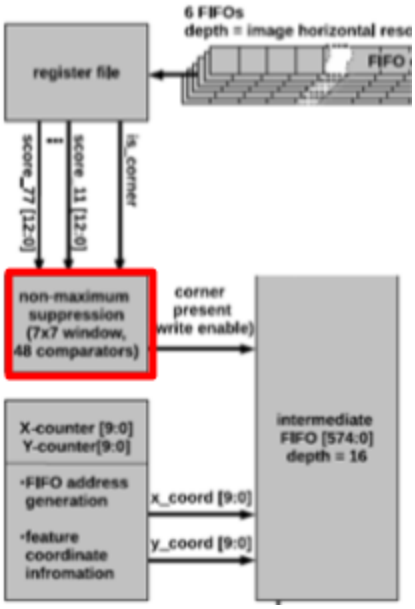
Non Maximum Suppression Buffers

Similar to the circle pixel buffer, the Non Maximum Suppression (NMS) buffers are responsible for buffering a serial datastream such that multiple rows of an image can be viewed simultaneously by the next module in the datapath. The output of the NMS buffer is a 7 x 7 window of corner scores

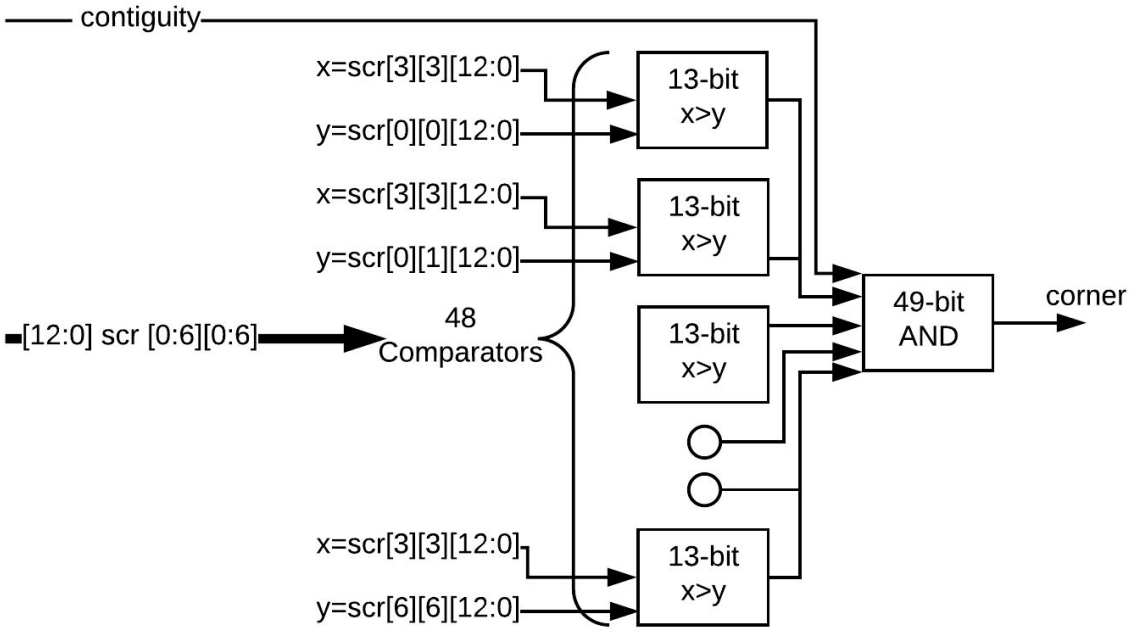


Non Maximum Suppression Block

The NMS block consists of 48 comparators. The module compares the corner score of the pixel which is at the center of the 7 x 7 window with the surrounding pixels' corner scores. The center pixel will be considered to be a valid corner if its corner score is greater than the surrounding pixel values. This non maximum suppression reduces the amount of noise in an output image.



Non-Maximum-Suppression Module



Address Counters

The address counters produce the memory location to pixel location mapping; the vsync signal from pixel data pipeline signifies a new frame which resets the counters and then a pixel data clock steps the counters. Generalizing, we rely on an input counter and an output counter; the input counter flags each data byte as valid or not and then the output counter counts the valid and invalid bits to generate an address. The output counter also generates a vsync_out signal when the number of consecutive invalid bits is greater than the number of invalid bits in a single row (ie. 6). In the submitted non-functional code, we initially thought this may have been wrong and opted to transmit the entire address of each pixel with each data byte through the entire FAST pipeline; the alternative of using only one bit for validity worked the same but recent modifications to the work was not conducted with valid bits. In the submitted code, the address is transmitted through the pipeline and output_counter is merely a pass-through for the address.

The input counter counts after vsync_in assertion; it uses a counter from 0-319 for the x dimension and a counter from 0-239 for the y dimension. The FAST pipeline, for its entirety, takes an extra bit: valid; valid is assigned as (xcount>2) && (xcount<317) && (ycount>2) && (ycount<237). The following pseudocode shows the structure of an input pixel:

```
typedef struct {
	logic [7:0] data;
	logic first_valid;
} input_pixel;
```

These input_pixels get propagated through the pipeline via the circle pixel buffer; the nms buffer reassigns the valid signal before passing it through the buffer. The nms buffer module needs to count the valid and invalid bits from the circle pixel buffer and reassign the valid signal; regardless of how the buffer reassigns the valid signal, if the input to the nms buffer had been valid, then it is passed through the buffer. The reassigned valid signal is counted by the output counter; these two steps allow us to ignore the wrap-around effect seen in line buffers for the edges of a frame. The following pseudocode describes the NMS buffer

```
module nms_buffer(input first_valid, output second_valid);
	// expected: 3*320 invalid, 234*{3 invalid, 314 valids, 3 invalids,}, 3*320
invalid, repeat
logic [8:0] valid_counter;		// keeps track of x
logic [3:0] invalid_counter;
logic signed [7:0] out_row_counter;		// keeps track of y
logic module_vsync;
always @ (posedge clk) begin
	second_valid <= 1'b0;
	if(invalid_counter == 15) begin
```

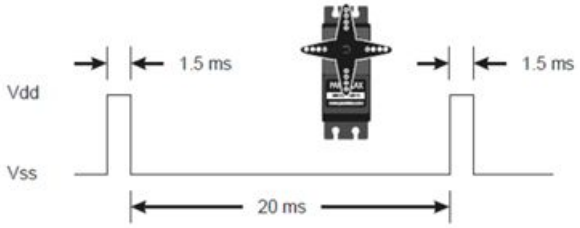
```
module_vsync <= 1; out_row_counter <= 8'd255;
Invalid_counter <= 0; valid_counter <= 0;
end
if(module_vsync) begin
	if(first_valid&& invalid_counter != 0 && out_row_counter != 8'd254) begin
		out_row_counter <= out_row_counter + 1;
end
valid_counter <= first_valid ? valid_counter+1 : 0;
invalid_counter <= first_valid? 0 : invalid_counter + 1;
if(out_row_counter > 8'd2 && out_row_counter < 8'd234
&& valid_counter > 2 && valid_counter < 314)) begin
second_valid <= 1'b1;
end
end

end
endmodule

typedef struct {
	logic [12:0] scr;
	logic corner_possibility;
	logic second_valid;
} scr;

The output counter then counts the valids to generate the address.

module output_counter(i_corner_pos, o_corner_we, o_address, vsync_out);
	// expect: 3*314 invalid, 228*{3 invalid, 314 valids, 3 invalids,}, 3*314
invalid, repeat
	logic [8:0] valid_counter;
	logic [4:0] invalid_counter;
	logic [7:0] ycount;
always @ (posedge clk) begin
	valid_counter <= valid ? valid_counter+1 : 0;
	invalid_counter <= (valid && invalid_counter != 3'd31) ? 0:
invalid_counter+1;
	if(valid_counter == 314 && ycount != 8'd255) ycount <= ycount + 1;
	if(invalid_counter == 5'd31) ycount <= 0;
end
assign o_corner_we = i_corner_pos && (valid_counter > 2) && (valid_counter < 314)
&& ycount > 2 && ycount < 228;
	assign o_address = '{valid_counter+6,ycount+6};
	assign vsync_out = (ycount == 228) ? 1 : 0;
endmodule
```

PMOD Servo Controller

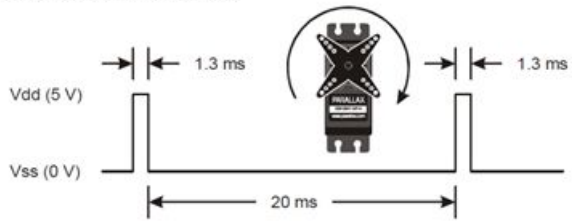
We use the Parallax Continuous Rotation Servo Motor (#900-00008). Below is the section of the servo motor datasheet which describes the communication protocol. It specifies that the servo needs a pulse between 1.3ms and 1.7ms, followed by a 20ms pause between consecutive pulses, to operate.

Communication Protocol

The Parallax Continuous Rotation Servo is controlled through pulse width modulation. Rotational speed and direction are determi ned by the d uration of a high pulse, i n the 1.3–-1.7 ms range. In order for smooth rotation, the servo needs a 20 ms pause between pulses. Below is a sample timing diagram for a centered servo:



As the length of the pulse decreases from 1.5 ms, th e servo will gradually rotate faster in th e clockwise direction, as can be seen in the figure below:



Likewise, as the length of the pulse increases from 1.5 ms, the servo will gradually rotate faster in the counter-clockwise direction, as can be seen in the figure below:

The IP also controls both motors at the same time, with the same speed setting. Currently the IP supports 5 settings:

1. Both motors move counter clockwise at full speed
2. Both motors move counter clockwise at full speed
3. Motor A moves counter clockwise at full speed, motor B moves clockwise at full speed
4. Motor B moves counter clockwise at full speed, motor A moves clockwise at full speed
5. Idle (motors do not move)

If an addition were to be made to this IP, it is suggested to accommodate for multiple speed settings. In addition, if more modularity is desired, it is recommended to have one IP block drive one motor, and repeat that block for the desired number of servos.

OV7670

Below is the configuration we planned for our final design of the OV7670 camera capture block.

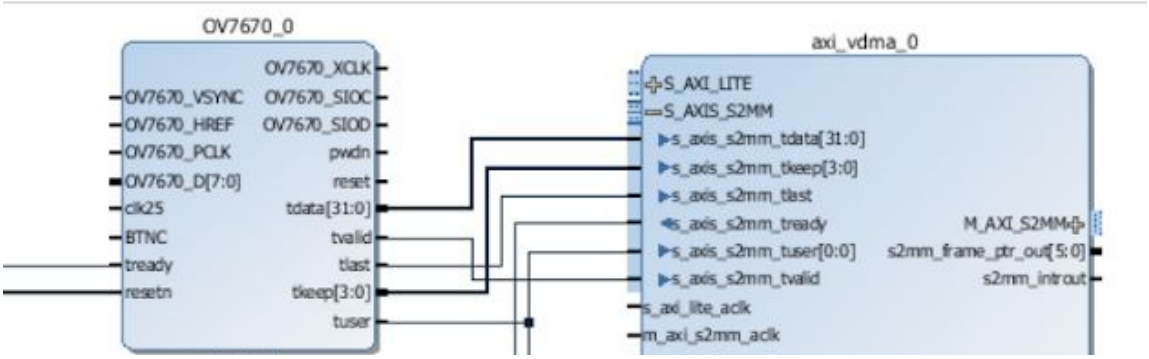
The output *tdata* port would be would be 1-byte wide in our design, as we would only need to stream in 1 grayscale pixel per cycle.

Additional ports are added to the original OV7670 in order to work with the VDMA

1. The *tuser* output signals the start of the frame
2. The *tready* input signals that the vdma is ready for reading
3. The *tvalid* output signal is asserted with the tdata
4. The *tlast* output signals the end of a line

The axi_vdma ip block would be configured to receive the video stream as input, and then write to DDR2 block through the M_AXI_S2MM port.

However, as previously stated, due to time restrictions the video feed from OV7670 was never integrated with the custom IP block we designed.

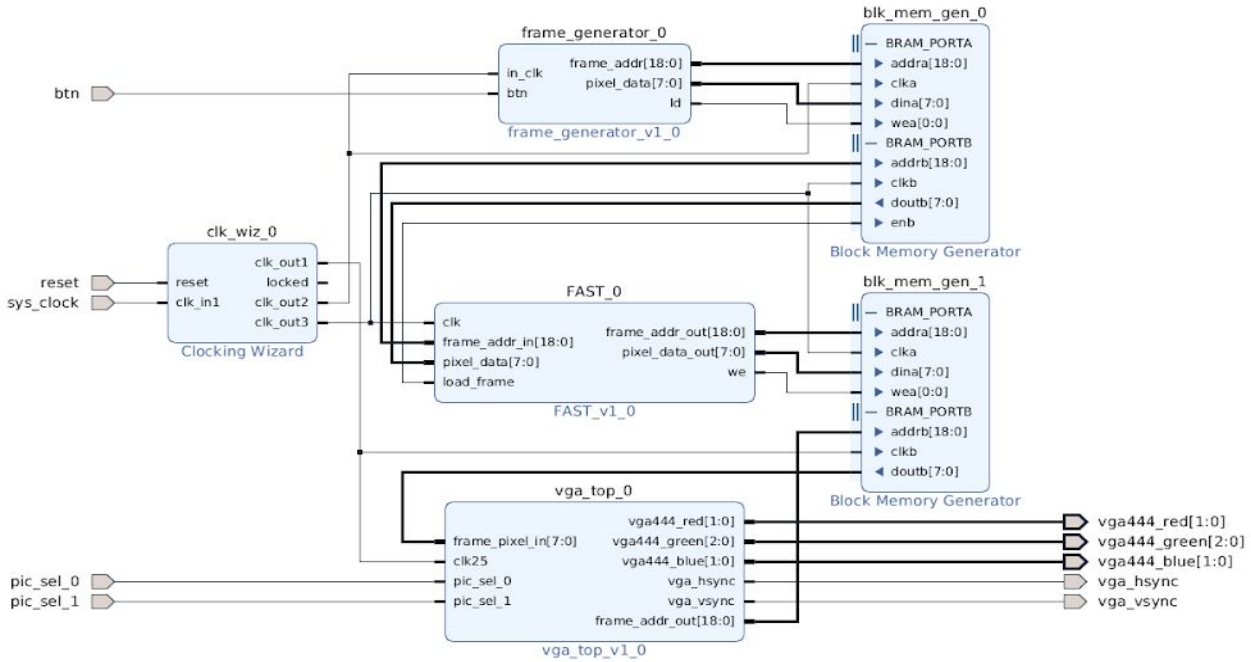


FRAME BUFFER (Originally VDMA)

Below is the configuration we used for our final hardware design. This was our design goal as we pivoted from our initial use of video data to still images. We were unable to fully implement this as we were pressed for time and the FAST block itself was not completely functional.

The **frame generator** is responsible for reading a image from a text file (preferably .bmp) and then writing the contents of the image into BRAM_0. It would assert a *we* signal and offset it by 19th and 18th bit of the address to identify the first, second and third frame, due to 1 frame being 320 x 240, 17-bits of address space was enough to fit a single frame.
The FAST custom ip block would then read the BRAM_0 contents and then compute for corners. white boxes to detect corners would be drawn directly onto the data, and then written into BRAM_1.

The **VGA** module will then read a single from BRAM_1, depending on the user switch input. The VGA module remains mostly the same as the module from the ece532_pmod_camera file, but has address and data widths modified to match our design.



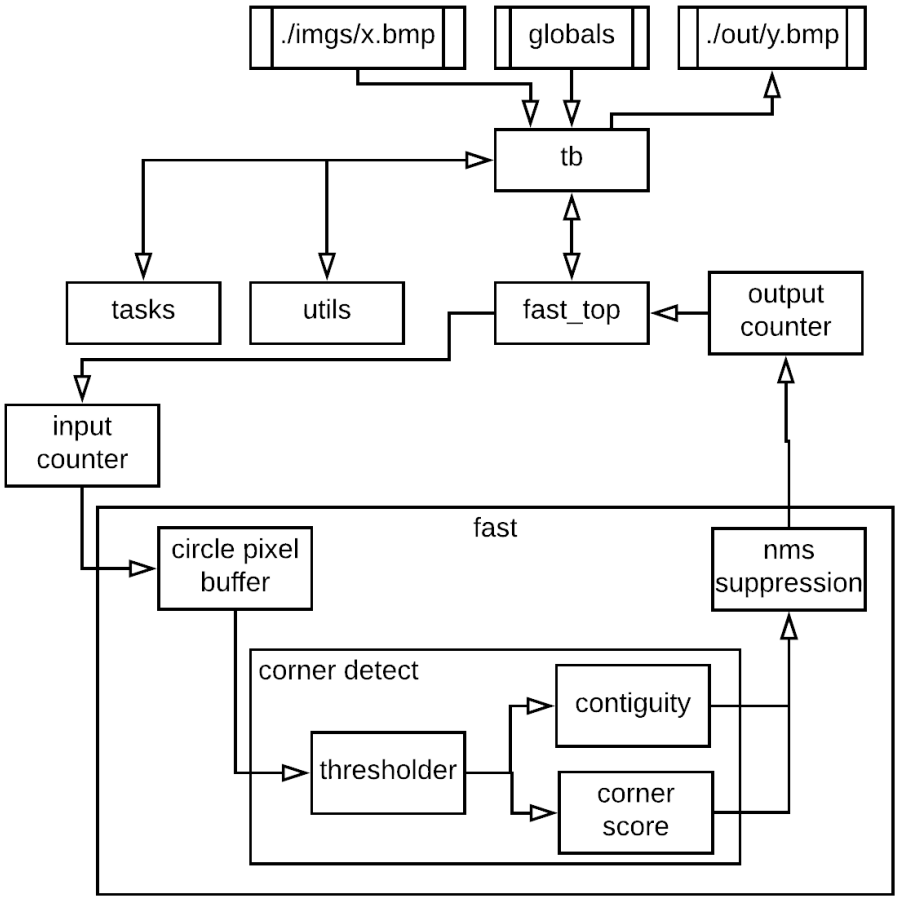
Description of the Design Tree

The source code for the project can be found at
https://github.com/nick-cusimano/ECE532-FAST-corner-detector

The repository contains three directories:

1. Imgs:contains input images for the FAST algorithm
2. Out: contains output images for the FAST algorithm
3. Src: contains source code for the FAST algorithm (shown in the figure below under the *fast* block), as well as a top level module, test bench file, and counters to perform pixel addressing. This is visualized in the below image

The bitmap test bench was based off of contributions from a prior ECE532 project test bench (https://github.com/ngemily/sample-bmp-tb).



Appendix A: Proposed Features

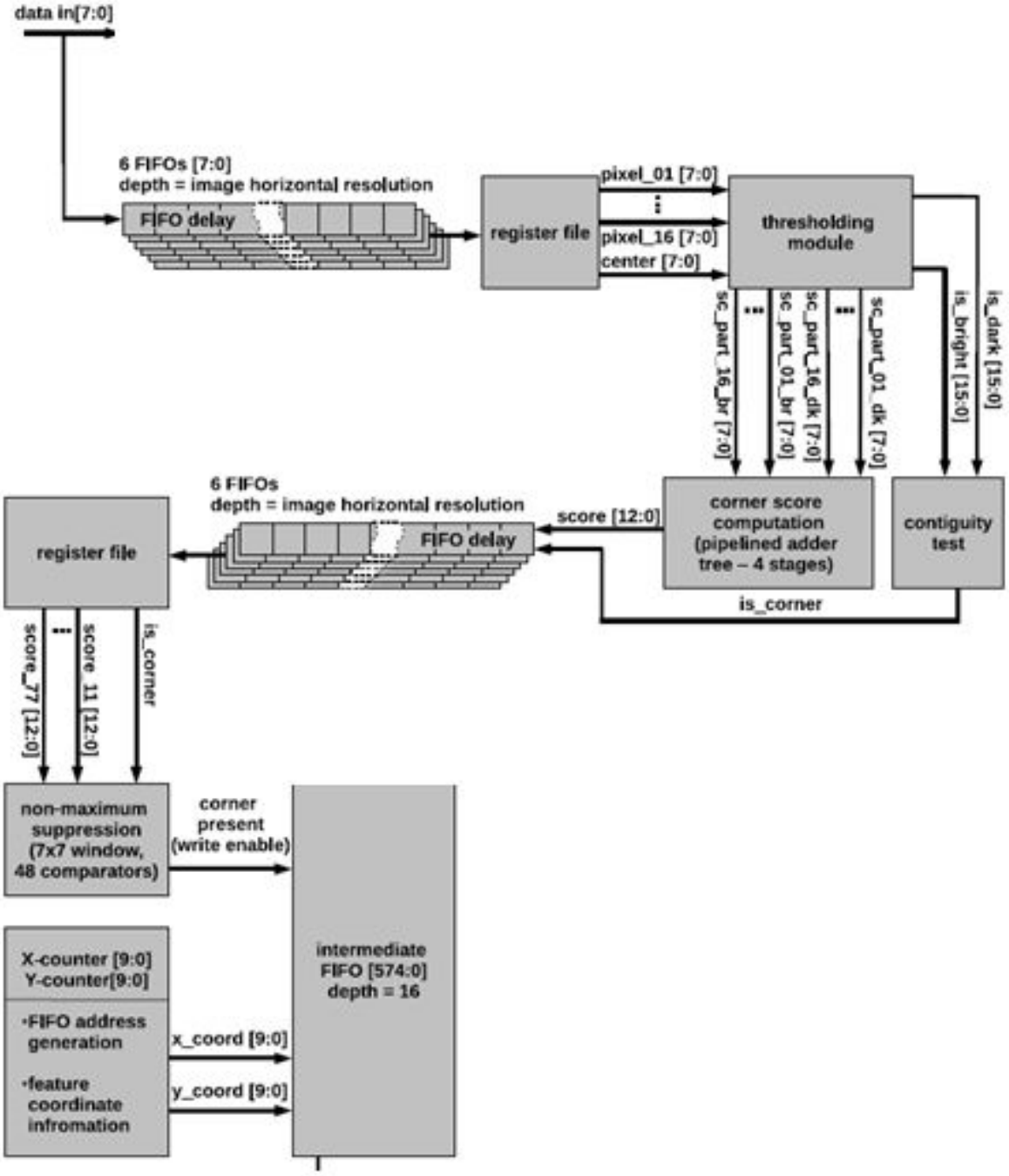
The team initially proposed to create a robot which senses the environment, broadcasts environment descriptors to a compute server, and subscribes to motion commands on the compute server. The custom IP in this project will be the camera peripheral which converts image frames to time stamped environment descriptors.

Microblaze System	Compute Server	Camera Peripheral
- Communicate with compute server over WiFi. Advertise environment descriptors; subscribe to motion commands - Command servo controller to actuate on environment - Make inertial measurements using IMU and SPI communication	- Subscribe to environment descriptors from FPGA - Process odometry using time stamped environment descriptors - Compute motion commands using the open source Robotics Operating System Navigation Stack [7] (ie. this will be used because it is easy to scale up features for this stack) - Advertise motion commands to FPGA	- Capture stereo image stream in FIFO - Distortion correction and rectification - Feature detection (Reimplement Putvision's FPGA-FAST) [8] - Feature matching - Stereoscopic projection (More features will be implemented and features will be removed from compute server if time is available)

Appendix B: Milestones From Project Proposal

Milestone #	Task Description	Person Responsible
M1 - Feb 5th	Servo PMOD - can actuate motor using	Nick
	Wifi PMOD / AXI SPI controller - can communicate with a server	Gallop/Jeff
	Camera to output to on-board VGA	Jeff
	Robot mobility test: can the robot carry the weight of all the required resources?	Nick/Gallop
	IMU PMOD / AXI SPI controller - can poll IMU data	Gallop/Jeff
M2 - Feb 12	Web server - can wirelessly communicate with FPGA client	Jeff
	Implement Putvision's FPGA - FAST (no modifications)	Gallop
	Integrate IMU, WiFi, Servo hardware (being able to read basic messages)	Nick
M3 - Feb 26	Modify FPGA-FAST for our use and set up image test bench	Gallop
	Implement camera distortion correction and rectification	Jeff
	2D-3D transform (stereoscopic projection)	Nick
M4 - Mar 5th	Feature matching	Nick
	Server: visual odometry + navigation stack	Gallop
	Testbench for feature matching and vision peripherals	Jeff
M5 - Mar 12th	Integrate feature match module with FAST module, then test	All
M6 - Mar 19th	Integrate all parts together	All
M7 - Mar 26th	Integrate all parts together	All

Appendix C: FAST Datapath



References

1. D. Scaramuzza et.al (2011, December). *Visual Odometry. Part I: The First 30 Years and fundamentals* [Online]. Available http://rpg.ifi.uzh.ch/docs/VO_Part_I_Scaramuzza.pdf
2. J.Hecht, OSA | The Optics Society (2018, Janurary) *LIDAR for self-driving cars* [Online]. Available https://www.osa-opn.org/home/articles/volume_29/january_2018/features/lidar_for_self-driving_cars/
3. D. Scaramuzza et.al (2011, December). *Visual Odometry. Part 2: Matching, Robustness, and Applications [Online]. Available* http://www.zora.uzh.ch/id/eprint/71030/1/Fraundorfer_Scaramuzza_Visual_odometry.pdf
4. A. Singh, “Visual Odometry from scratch - A tutorial for beginners,” *Avi Singh's blog*. [Online]. Available: https://avisingh599.github.io/vision/visual-odometry-full/.
5. “Department of Informatics,” *Tutorial on Visual Odometry - Davide Scaramuzza*. [Online]. Available: http://rpg.ifi.uzh.ch/visual_odometry_tutorial.html.
6. *FAST Corner Detection -- Edward Rosten*. [Online]. Available: https://www.edwardrosten.com/work/fast.html
7. “PUTvision/FPGA-FAST,” *GitHub*. [Online]. Available: https://github.com/PUTvision/FPGA-FAST.
8. “Gradient Filter implementation on FPGA : Part 2 Implementing gradient Filter,” *Recent Posts*, 26-Mar-2015. [Online]. Available: https://www.element14.com/community/groups/fpga-group/blog/2015/05/27/gradient-filter-implementation-on-fpga-part2-first-modules.
9. R. Avizienis. *Building your First Image Processing ASIC*. [Online] Available: https://inst.eecs.berkeley.edu/~cs250/fa12/handouts/lab2-median-filter.pdf