Grand Theft Auto Final Report

Students:
Milos Nikolic,
Neil Veira,
Yaron Milwid
Group #: 8
TA: Charles Lo
Professor: Paul Chow
Course: ECE532 H1S

Overview

High Level Description

The goal of this project was to create a system whereby a remote-control car would drive itself based on a video camera feed. The car was constrained to drive inside a user-defined track and avoid obstacles. Both the track and obstacles were defined by red tape. Upon reaching the target, signified by white, the car would stop.

Motivation

Currently, self-driving cars are often in the news, and many companies are attempting to develop self-driving vehicles of different types. While we knew we would be unable to develop an understanding of all issues experienced while developing self-driving vehicles, we wanted to learn about some of the considerations involved. The main considerations we were interested in include: path finding, processing of sensor data, transmission of sensor data and performing some type of actuation based on the the path finding algorithm's decision.

Block Diagrams and Flow Diagrams

Figures 1 and 2 below show the initial block and flow diagrams for this project, while Figures 3 and 4 show the final block and flow diagrams for this project. While the flow diagram became simplified due to the removal of uncertainty and the linearization of the dataflow, the block diagrams got significantly more complicated. For example, in the updated block diagram, instead of using 1 AXI-interconnect, we have 3 AXI-interconnects, and instead of using 1 Microblaze controller, we have 2. We also have 3 clock domains instead of 1, thus introducing the need for synchronization.

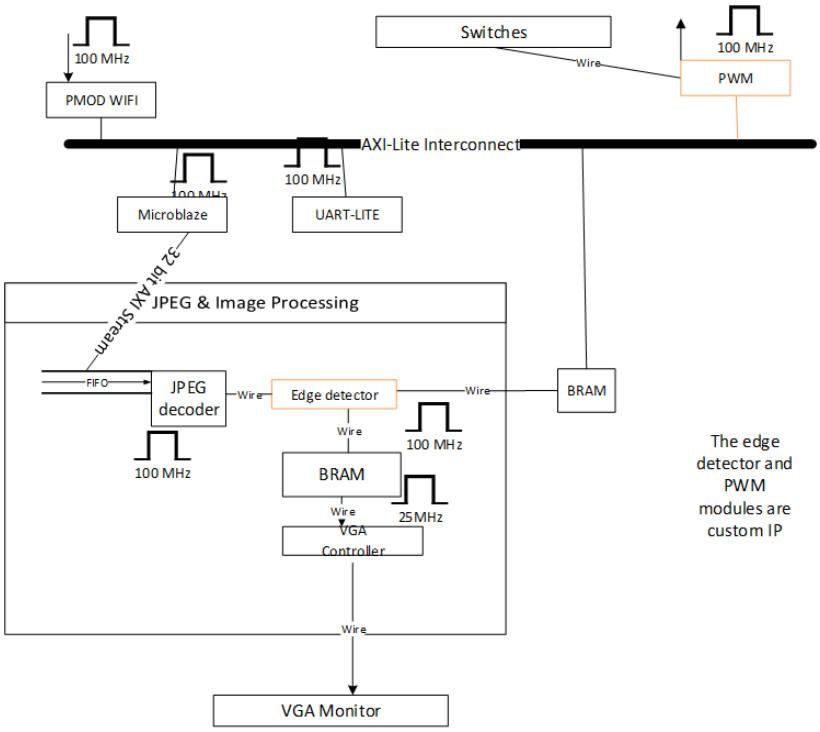

Figure 1 Initial block design

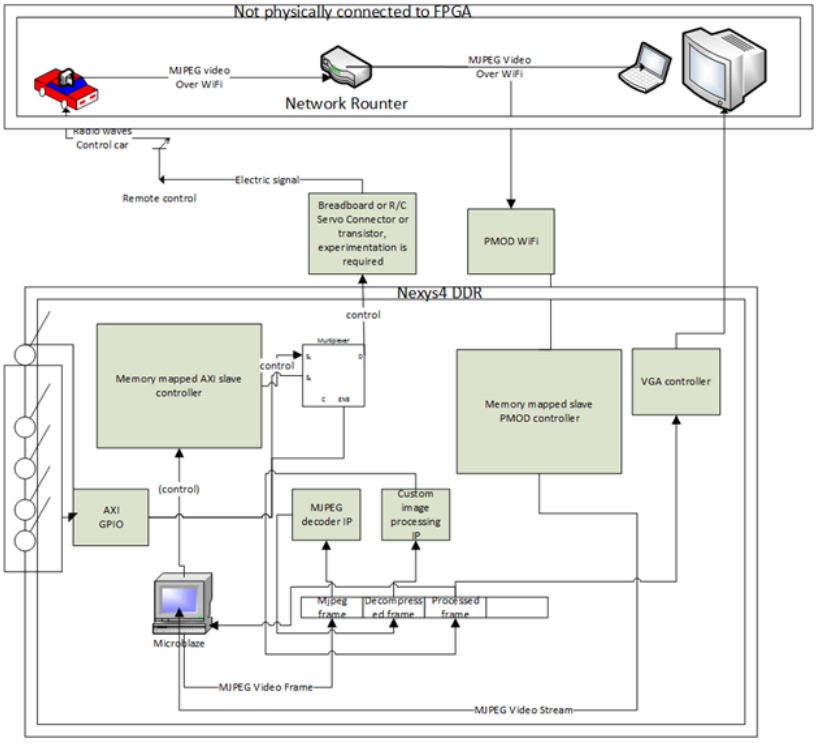

Figure 2 Initial flow diagram

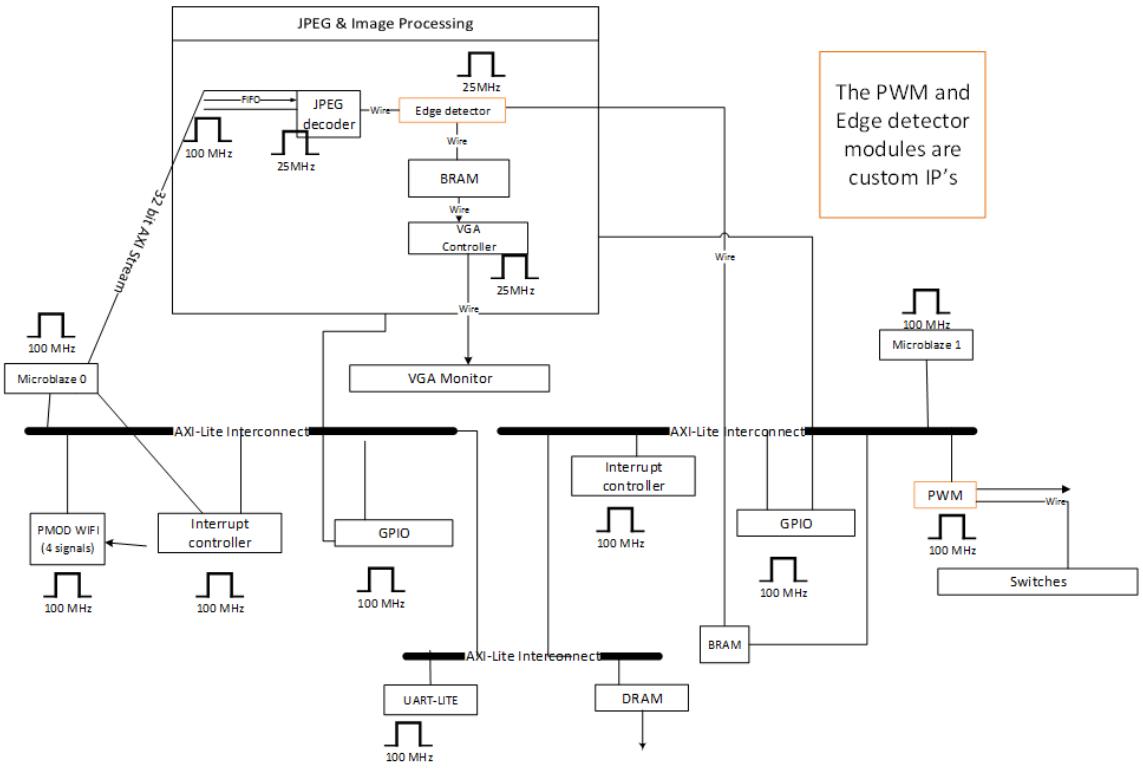

Figure 3 New block design

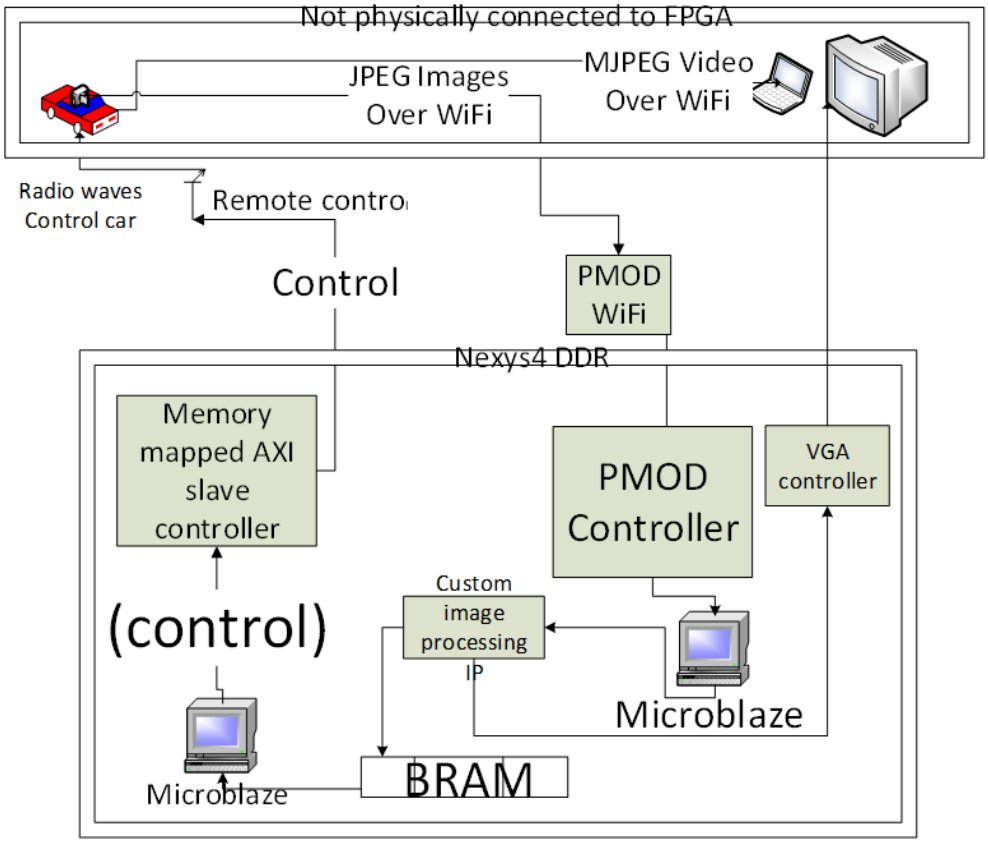

Figure 2 Initial flow diagram

IP Created

We created two custom verilog/vhdl blocks as well as one software block.

1. The first custom IP we made was an image processing block. This block took in an image in jpeg format and output two things: a list of (x,y) coordinates of pixels which are considered edges in the image, and RGB values for every pixel to be displayed on the VGA monitor. The block contained a jpeg decoder module which was not created by us, but there was some additional wrapper logic to work around bugs in the jpeg decoder (see final functionality section). After decoding the image into raw RGB values the block performed colour thresholding to identify pixels either as “red” or “not red” and edge detection to identify pixels as edges of the red sections. A second instance of this edge detector was used to detect white sections for the car’s destination. Also within the image processing block is a VGA driver module which generates the Hsync and Vsync signals for a VGA monitor. This part was not developed by us, but we wrote a wrapper module on it to control what RGB values to output for a given (x,y). This controller displays the red and white sections and the edges in the processed image on the VGA.
2. The second custom IP block was a pulse width modulation (PWM) block used to control the remote controller. Four outputs of this IP are soldered to the remote controller to drive the direction and speed of the car. The block operates at 1.5 kHz and 2 input registers specify the duty cycle of the 4 outputs, half a register each. The block was encapsulated in a wrapper that included a switch to multiplex between microblaze and manual control buttons the Nexys4 DDR board. The output of the block in negated, to match the controller operation.
3. The control algorithm consists of 2 stages and was applied in software on microblaze. In first stage weighted A* star was used to find a possible path. If no goal was found by the custom IP, middle top of the frame was selected. If no path was found all nodes would have been explored and the closest border node to target is selected. In the second stage, nodes on the path were examined and the farthest one that can be reached without hitting any obstacles is selected as a waypoint. Heading and speed are adjusted accordingly. Communication with the custom IP is done through a dual port BRAM block. The block contains 3 sections, each containing a list of edges for one frame. One of sections is used for the microblaze to read from, one is used for the image processor block to write to, and one is used to store the next most recent frame. The microblaze and image processor send signals to each other to communicate which of the three sections they are currently reading to/writing from. At the beginning of each frame the microblaze and image processor both check which section the other is using and switch to a different section. This protocol ensures that the microblaze is always able to read the most recent fully processed frame while never reading from the same frame that the image processor is writing to.

Third party IP used/modified

The main Xilinx IP we used were the FIFO, DRAM, BRAM and clock generators, as well as the Microblaze and AXI infrastructure.

We also used the Digilent WiFi PMOD IP, which included a sample project for creating a TCP echo client. We modified this client to repeatedly query a server, trim the response header, and send the response to the image processing IP.

In order to send the images captured by the car to the WiFi PMOD, we used the IP Webcam application for Android.

Since the image sent over WiFi was in JPEG format, we used a JPEG decoder that we found of Opencores.org. We wrapped this IP in an AXI-Stream wrapper with a FIFO block and some logic to convert the protocol used by the IP to the AXI protocol and to work around some bugs in the JPEG decoder itself.

Finally, we display the processed image using a VGA driver developed by Rob Chapman for the Altera University Program. We chose to use this module rather than something like the TFT controller because it is small, simple, easy to use, and is better suited for our purposes. Specifically, because our image processing was done in verilog, it was more convenient to take the RGB values from the verilog module rather than having the microblaze do it. Moreover, we wanted to display all the red pixels as well as the edges which would be too much for the microblaze to process.

Project Complexity

Component	**Type**	**Points**	**Justification**
Microblaze #1	Xilinx IP	1	Given
Microblaze #2	Xilinx IP	1	Given
WiFi	PMOD	3	Comparable to bluetooth
JPEG decoder	Third party IP	3	This turned out to be the most troublesome and difficult to use component of the entire project
VGA display	Feature / third party + custom IP	1	Given
Edge detection	Custom IP	1	Given
PWM remote controller	Custom IP	1	Given
Remote control car	External hardware	0	
Phone camera with IPwebcam app	External hardware	0	

DDR	Feature	1	Given
Microblaze interrupts	Feature	1	Given
Pathfinding	Software	0	
Total		**13**	

Outcome

Final Features

At the time of our demonstration, our project had all of the features implemented that we originally planned to implement, including the optional features. Unfortunately, not all of them worked perfectly, as will be discussed in the next section. The successful features include:

- Receiving images from wifi using the wifi pmod
- Identification of objects as obstacles, destinations, or neither based on their colour in the image
- Finding a path to reach the destination while avoiding the obstacles
- Continuous travel until reaching the destination, or determining inability to continue
- Display of the processed image on the VGA monitor with the obstacles in red, destination in white, and edges of both in green.
- Override switch for manual control of the car by a human using the Nexys4 DDR board

The following features were implemented but not working perfectly:

- Once placed in the obstacle course, navigation without human intervention:
    - We had issues decoding certain JPEG images, therefore some inputs confused our algorithm and it was unable to recover
- Ability to handle any number of objects of any shape or size:
    - For the same reason as above, at certain points the project stopped working

Final Functionality

The following aspects of the design were functioning correctly:

- Receiving valid jpeg images from wifi into the microblaze
- Colour thresholding and edge detection of decoded jpeg images.
- Calibrate mode for the image processor which dynamically sets the target colour of the obstacles to the colour in the center of the image.
- Displaying the processed image on the VGA monitor with obstacles in red, destinations in white, and edges in green.
- Communication between the image processor and the second microblaze. We confirmed that the microblaze was able to read the correct edge list written into ram by the edge detector, and it was able to correctly receive a goal coordinate.
- Pathfinding algorithm to navigate to the goal while avoiding obstacles (edges)

- Selecting a direction to move the car for the current frame
- Moving the car in the desired direction using the PWM module
- Override switch to control the car directly from the buttons on the Nexys4 DDR board

The following aspects of the design were not functioning correctly:

- Decoding JPEG images when the bytes are sent to the JPEG decoder in an intermittent manner (ie. valid data is given for only 1 cycle out of every N cycles, where N >> 1). When we first encountered this issue we found (in simulation) that there was a bug in the JPEG decoder which caused it to incorrectly identify the bytes belonging to the huffman tables in the image. This caused the huffman decoder to read invalid data and throw an error, at which point the entire JPEG decoder would stop completely until reset. Because the JPEG decoder is a large and complex system which was written by a third party we tried to work around the bug rather than fix it. Our work-around was to add a fifo queue between the microblaze and the jpeg decoder. The bytes sent from the microblaze were written to the fifo and were not read out until there were sufficiently many piled up (~512 words). At that point the entire fifo would be emptied out in one go, sending one word to the JPEG decoder per cycle. Using this method we were able to correctly decode, process, and display some images sent from the microblaze. Unfortunately this still did not work for *all* images. For certain images the jpeg decoder would start decoding and give correct RGB outputs up to some point, and then suddenly crash and give garbage output. We were able to consistently reproduce this issue in simulation and on hardware but were unable to determine the cause or find a work-around. We also confirmed that the images fed to the decoder were valid JPEG images.
- We also had additional issues decoding wifi images. We confirmed that the wifi images were valid JPEGs by printing out the bytes from the microblaze, copying them to a binary file, and opening it as a JPEG. However, the jpeg decoder would consistently crash on the second image sent for reasons which we were unable to determine.
- Even when the images were decoded successfully we had difficulty adjusting the colour thresholding to the varying lighting conditions. The calibrate mode feature was added to try to solve this problem, but we never reached the stage where we could make use of it due to the above issues with the wifi and JPEG decoding. .

Unfortunately, the JPEG decoder component is at the beginning of the pipeline. Without it functioning perfectly the design is unable to demonstrate much functionality.

<u>Changes we would make if we could start over</u>

Given the chance to start over, we would redesign the system so as to remove the JPEG and WiFi components. To do so we would use a much bigger and more powerful car that is capable of carrying the Nexys4 DDR board, camera, power supply, and remote controller. This would allow us to send raw video directly from the camera into the image processor, thus removing the need for the wifi component and the JPEG decoder.

Such a design would be far more successful since most of our issues arose from the JPEG decoder block, and it was ultimately responsible for our design's malfunction during the demonstration. Removing the wifi component would also be beneficial since we were unable to

find a method of sending images over WiFi without using compression. Additionally, we found that using Vivado 2016.2, this portion of our project was extremely slow to connect to the server and retrieve an image. Using Vivado 2016.4, once the WiFi was connected, the speed was acceptable, but for testing and development purposes, the time required to connect to the WiFi network was too long. Additionally, the only computers available to us that had Vivado 2016.4 installed were significantly slower than those using Vivado 2016.2. As such, development on these computers was extremely difficult and time-consuming.

In order to remove the two components above, we would need to find another way of transferring the video feed to the development board. The easiest way of doing this is to connect the video directly to the development board. However, this requires having a power supply connected to the board as well, and therefore the remote control truck would need to be larger and more powerful.

Next Steps

1. The immediate next step of the project would be to debug the jpeg decoder. It is a critical stage of the pipeline and nothing more can be done until all of its issues, as described in the previous section, are resolved.
2. One would then need to find a method for speeding up the wifi component. As explained previously, retrieving a single JPEG image via http was unacceptably slow (on the order of 10-30 seconds). In order for the design to be of any practical use this would need to be sped up by at least an order of magnitude.
3. Depending on the final speed of WiFi, the microblaze with the control algorithm might also need a speed up, either by reducing image size (increasing A* step size) or by speeding up of the microblaze itself. If execution time needs to be reduced to below 1 second, part of the pathfinding algorithm might need to be implemented in hardware.
4. Testing: although the other components of our design appear to be working together correctly, we did not have much opportunity to test them thoroughly as our main focus was on resolving the above issues. In particular, more testing would be needed for the communication protocol between the image processor and the path-finding microblaze. This protocol was designed to allow these two components to read and write edge lists in different frames out of sync with each other (see “IP we created” part 3).
5. Fine-tuning: even once all the components are functioning together correctly, parameters such as the car speed, colour thresholds, and reactions to different situations would need to be adjusted for the car to successfully navigate the obstacle course.

Project Schedule

For most of our early milestones, we achieved more than expected. Additionally, due to the testbench demonstration being postponed, we changed the order of certain milestones. When we began integration, our milestones were seldom achieved since we had issues integrating the JPEG decoder into the rest of the project. Many of these issues were initially due to black box errors, which we solved by using a manual compile order and generating certain blocks out of context, and certain blocks globally. After finally solving this issue, we had issues due to the different protocols used by the JPEG decoder and the Microblaze and bugs in the JPEG decoder. When the images sent via WiFi were finally decompressed successfully and the edges displayed on the VGA monitor, we found that we had significant timing issues due to the size of the AXI-interconnect. As a result, we had to redesign the architecture using a hierarchical AXI interconnect.

The following table gives a detailed breakdown of the project timeline – what was planned and what was accomplished for each milestone.

Milestone # (MM/DD/YY)	**Original Milestone**	**Actual proposed Milestone: Yaron**	**Actual Achieved Milestone: Yaron**	**Actual proposed Milestone: Neil**	**Actual Achieved Milestone: Neil**	**Actual proposed Milestone: Milos**	**Actual Achieved Milestone: Milos**	**Actual proposed Milestone: Group**	**Actual Achieved Milestone: Group**
1 (02/10/17)	Hijack the controller ● Test ability to send signals for the 7 possible directions of movement (see testing section). ● Direction defined in a testbench Microblaze code and converted into 4 output signals ● Signals output through the PMOD output pins on the Nexys4 DDR board ● Disassemble the controller hardware and connect it to the output pins on the Nexys4 DDR board. If necessary, build circuit to interface them ● Ability to override Microblaze and control car with the switches ● Verify that the testbench code	● Write tests to validate car remote hijacking	● Worked with Neil to find IP for MJPEG decoding that does not require extra licenses ● Wrote routing Verilog code o Manual control of car (directional switches) o Selector (SW[0]) to determine whether to use manual control or internally generated signals o LED's displaying status of switches, signal from microcontroller, output signal (which direction to move, not actual signal) ● Wrote C test bench for car remote hijacking ● Worked with Milos to	● Acquire tape to make obstacles ● Research image processing and decide algorithm	● Acquired tape ● Obtained jpeg decoder from opencores and started working on porting it to Vivado 2016.2 ● Designed the image processing algorithm and implemented it in Python o colour thresholding to identify red pixels o 9 x 9 sliding window which counts the number of red pixels to identify edge pixels ● Tested image processing on sample images and tuned parameters until it worked well	● design a PWM module to allow us to control the speed of the car	● designed, implemented and tested a custom IP PWM module	● Hijack Car	● Determined how controller works

	causes the car to move as desired		integrate his PWM IP ● Includes adding test cases to test PWM (testing partial speed, partial turning) ● Worked with Milos to hijack controller ● Verified hijacking of controller o Determined output signals correct o Determined car moves correctly o Determined car can move with phone on it ▪ Placing phone on back of car removes too much friction from front of car, thereby preventing turning ▪ Will solve this issue by taping phone holder to front of car or top of car o Determined placing wires in contacts creates contact issues						

			▪ Will solder wires on once have determined whether we need longer wires						

2 (02/17/17)	• Testbench demo: Testbench for custom image processing IP Create sample data to test algorithm and expected results Several obstacle courses encoded in text files in similar format as will be output by the custom image processing IP. Test Cases for MJPEG decompression Test for WiFi	• Port WiFi API over to work with Microblaze	• Found IP for the WiFi PMOD o Digilent ported WiFi API over to be used with Microblaze • Created project to be used to test WiFi PMOD • Can send/receive data via UDP with phone • Modified TCP client sample code to get data from website • Beginning to investigate getting images from camera o Need to improve understanding of structure of server for how to get particular pages from server, not landing page • Soldered longer wires onto controller	• Have the jpeg decoder synthesized and simulate a few images on it. Understand how to interface with it • Implement image processing in Verilog	• Continued working on converted JPEG project to Vivado. Didn't realize that the version I had downloaded had many files missing so I got stuck trying to convert the old IPs to Xilinx IPs. • Wrote first implementation of image processor in Verilog using shift register to store 9 x width pixels. • Wrote testbench for it which sends it raw RGB pixel data from a file, reads the output edges, and compares them to a list of expected edges in a file. • Simulated the module on a small simple image.	Control algorithm for Microblaze • Research possible ideas • Define inputs • Start coding • Have basic functionality of the control algorithm	• Finished coding basic functionality • Started coding the path finding algorithm, no results yet	• Mount camera • Create test data for MJPEG and image processing	• Camera mounted • Took photos of different colored objects on carpet using camera to be used in project, in order to test image processing algorithms

3 (03/03/17)	● MJPEG decompression IP selected and working (ensure output is the same as software version of decompression to within threshold)Image processing IP designed Use ImageJ to try different object recognition methods in order to select best and determine output format for path-finding algorithm	● Use ILA to implement test-bench for PWM and controller hijacking code	● Used ILA to probe switches, microblaze instructions and output ports ● Found correct structure for HTML Get request to get JPEG images from the phone	● Get the JPEG decoder working ● Integrate it with the edge detector	● Realized my previous error with the JPEG decoder having missing files. Recreated the project using post-synthesis netlists from the SVN version of the JPEG project. Synthesized and simulated it successfully. ● Integrated the edge detector with a microblaze to make sure it could read the edge list correctly. ● Wrote a testbench for the above design for the testbench demo.	●	●	●	●

4 (03/10/17)	• WiFi receives video and places in correct memory location Path finding algorithm implemented and tested Camera affixed to car Custom image processing IP implemented and tested	• Figure out storage of images retrieved from the server	• Changed PWM to use high impedance pins • Changed wireless code to repeatedly retrieve JPEG images • Wrote code to trim header o Unable to test yet • Wrote code to send data retrieved from camera to FIFO channel o Unable to test yet • Added FIFO channel to design • Attempted to add Neil's JPEG decoder to empty the FIFO channel o Did not work	• Inte grate JPEG decoder with the edge detector	• Tested the edge detector on images up to 256 x 256 and confirmed it works. • Integrated JPEG decoder with the edge detector. o Discovered that the JPEG decoder outputs pixels in 16 x 16 blocks which is very inconvenient for edge detector. Also does not output pixels at an even rate. Had to add a buffer and quite a bit of logic to allow the two components to interface correctly. • Modified edge detector to write edges to ram instead of fifo. Should be faster for the microblaze to read.	• Estimate the size of the car in any position on the image • Select an intermedi ate goal when the final goal is not in the image • Explore modifying the heuristic to push the path further away from the obstacles and tracks •	• Derived formula for size estimation • Implemented and tested size estimation function • Implemented modified heuristic • Implemented function to follow selected path •	•	•

5 (03/17/17)	• Mid-Project Demo: Integration of MJPEG decompression, WiFi, and Algorithm	• Get JPEG decoder into project	• JPEG decoder, edge detector, VGA output now in project as AXI-Stream peripheral • Project synthesizes, implements and bitstream generates • Endianness issue with to AXI-Stream peripheral fixed • AXI-Stream peripheral doesn't assert ready signal • Edges not shown on monitor	• Integrate JPEG decoder + edge detector with wifi • Add VGA output for processed image	• Added VGA driver module which I found on the web and integrated it with the edge detector o Required ram to buffer pixels and some control logic. o Looked into TFT but decided it was not suited for my purposes • Tested the image processor + VGA output on hardware by initializing a ram with a jpeg image and wrote a wrapper module to send data from ram to JPEG decoder o Somewhat worked, but displayed image had some garbage pixels (later found to be due to bug in JPEG decoder) • Tested image processor on 640 x 480 image and discovered that the shift register became much too	• Select an intermediate goal when the final goal is not in the image • Deploy and test on Microblaze • Interface with IP for edge/barrier detection •	• Selection of intermediate goal when the final is not present • Selection of intermediate goal when the final one is not reachable • Fixed way-point selection • Algorithm is now complete • Started deployment on Microblaze •	•	•

					big. Had to reimplement a significant portion of the edge detector using different approach. ● Worked with Yaron to integrate image processing with wifi (see Yaron section)				
6 (03/24/17)	● Integration of milestone 5 and hijacked controller Testing Output processed image to VGA monitor			● See group section	● Ran lots of simulations and tests to debug image processing component alone. o Discovered bug where JPEG decoder sends ready signal high before it is actually ready, causing garbage output. o Worked around bug and got image processor + VGA working perfectly on hardware <u>when data is sent one word per cycle</u> ● More simulations revealed the bug where JPEG decoder errors if not given one word per cycle.	● Finish deployment and test on Microblaze ● Interface with IP for edge/barrier detection ●	● Path finding algorithm deployed and tested on Micorblaze ● Interface with the PWM output module done ● Designed simple AXI slave to communicate with edge detection ● Connected synthesized and implemented interface with edge/barrier ●	● Get JPEG decoder to work in overall design ● Begin integrating Milos' algorithm into design	● JPEG decoder, edge detector, VGA output now in project as AXI-Stream peripheral ● Project synthesizes, implements and bitstream generates ● Able to send an image to display on the monitor ● Changed WiFi to disconnect from the server upon reading 0xFFD9, rather than based on the time since transmission began o This speeds

					o Added fifo + control logic to work around ● Added calibration mode feature which allows us to set the target colour to adjust to different lighting conditions. ● Worked with Milos and Yaron to integrate with their parts				up the process o This prevents images from being cut off
7 (03/31/17)	● Final Demo: have the entire system integrated and working			●	● Debugging with the rest of the group	●	●	● Finalize the integration	● Got all parts working individually ● Switched to Vivado 2016.4 ● Had issues with the JPEG decoder

Description of the Blocks

The following is a list of all the main blocks in our top-level block design.

- IP Webcam (V1.12.5r) android app, from (https://play.google.com/store/apps/details?id=com.pas.webcam&hl=en)
    - Photo resolution set to 640x480
    - Quality set to 12
    - LED toggled on
- PMOD WiFi IP (V1.0 Rev 25) from Digilent, can be accessed at (https://github.com/Digilent/vivado-library/archive/master.zip)
    - Pmod Bridge (V1.0 Rev 7)
    - AXI Quad SPI (V3.2 Rev 10)
    - AXI GPIO (V2.0 Rev 13) (2 of these)
    - AXI Timer (V2.0 Rev 13)
- TCPEchoClient example bundled with PMOD WiFi IP
    - Changes:
        - Server, port changed to point to IP Webcam
        - SzSsid changed to point to hotspot, and password added
        - All connection modes except WPA2 Passphrase removed
        - tStart and tWait removed and replaced by checking for a specific series of bytes
        - Get request changed to request image from IP Webcam
        - Function added to display title page while connecting to WiFi network
        - Logic added to remove the header of the HTTP response
        - Much of echoing removed
        - Data received forwarded via FSL
        - Upon closing connection to a server, the code now reconnects and re-sends the Get request
- Microblaze_0 (V10.0 Rev 1) for the wifi component
- Microblaze_1 (V10.0 Rev 1) for the path-finding algorithm
- AXI Interconnect (V2.1 Rev 12) x3. Three of these were used to build a hierarchical AXI interconnect (due to timing requirements). They were used as follows:
    - Connecting microblaze_0 to its slaves
    - Connecting microblaze_1 to its slaves
    - Connecting the first two AXI's to the slaves of both microblazes
- MicroBlaze Debug Module (MDM) (V3.2 Rev 8)
- AXI Interrupt Controller (V4.1 Rev 9) x2 (one for each microblaze)
- Processor System Reset (V5.0 Rev 10) x2 (one for each microblaze)
- Local Memory Bus (LMB) 1.0 (V3.0 Rev 9) x2 (one for each microblaze)
- LMB BRAM Controller (V4.0 Rev 10) x2 (one for each microblaze)
- Stream_jpg_yy_nv_mn_v1_0_v1_0_wed2 (V1.0 Rev 36). Custom IP to perform image decompression, thresholding, edge detection, and VGA output. Uses:

    - FIFO Generator (V13.1 Rev 3) to work around a bug in the jpeg decoder which requires data to be sent one word per cycle (see description final functionality section).
    - Image_processor: contains logic for interfacing between the jpeg decoder, the edge detector, and the VGA controller. Submodules:
        - Jpeg: JPEG decoder top-level, taken from https://opencores.org/project,mjpeg-decoder
        - Block Memory Generator (V8.3 Rev 5) for buffering pixels (RGB) between the jpeg decoder and the edge detector. This is needed because the jpeg decoder outputs pixels in 16 x 16 blocks, whereas the edge detector uses a 9 x 9 window that slides through the pixels in row-major order.
        - Edge_detector x2: custom IP to perform colour thresholding and edge detection. One instance detects red, the other detects white. Uses:
            - Block Memory Generator (V8.3 Rev 5) within the edge detector for storing intermediate pixel sums as the window slides.
        - vga_controller: Wrapper containing logic to output edges and coloured pixels to VGA port. Uses:
            - Block Memory Generator (V8.3 Rev 5) within the vga controller for storing pixel values output to the VGA. This is needed because the vga driver (which generates the Hsync and Vsync signals) runs independently of the edge detector.
            - Vga_driver: module by Rob Chapman, Altera University Program to generate the Hsync and Vsync signals. Can be accessed at http://www.ece.ualberta.ca/~elliott/ee552/studentAppNotes/1998_w/Altera_UP1_Board_Map/vga.html.
- Block Memory Generator (V8.3 Rev 5). Used for storing edge list between stream_jpg_yy_nv_mn_v1_0_v1_0_wed2 and microblaze_1
- AXI BRAM Controller (V4.0 Rev 10)
- AXI GPIO (V2.0 Rev 13) x2, used for sending individual signals between the microblazes to other modules.
- AXI Uartlite (V2.0 Rev 15)
- Switch_led_out (V1.0 Rev 4)
    - Custom IP to perform PWM and select between automatic mode, or user control of car
- Clocking Wizard (V5.3 Rev 3)
- Control Algorithm
    - SEARCH - finds a path by weighted A*
    - NAVIGATE - Set heading and speed according to the selected waypoint
    - GENERATE_EDGE_LIST - gathers all edges from BRAM
    - PRINT_RESULT - selects a waypoint from the path
    - INIT - initializes microblaze for searching
    - PROBE - probes the node to estimate cost to goal from the probed node
    - EDGE_DISTANCE - applies penalty to cost to goal if too close to an edge
    - BEST - Select the best node from the open nodes list

- EDGE_DISTANCE - estimates size of car at a distance based on height of pixel
- GET_EDGE, GET_NODE - get nodes or edges from the free list
- PREALLOCATE_EDGES, PREALLOCATE_NODES - preallocate edges and node to improve performance

Description of Your Design Tree

All the project and related files can be found at https://github.com/yaronm/G8_Self_Driving_Remote_Control_Car. The file README.md contains a full project outline. The following is our design tree as described in this file.

- src: Vivado project files
--- project_1: Main project
----- project_1.sdk: software files
------- MB0_n: SDK project for wifi microblaze
--------- src/main.cc: Contains all relevant code for this component
------- MB1: SDK project for path-finding microblaze
--------- src/helloworld.c: Contains all relevant code for this component
----- project_1.srcs: hardware source files

--- img_proc_full: Packaged project containing the image processing IP
----- jpeg.srcs/sim_1/imports/vhdl/jpeg_testbench.vhd:
------- sources_1: verilog & vhdl source files
--------- stream_jpg_yy_nv_mn_v1_0.v: top-level source file for sending data from this microblaze
--------- image_processor_wrapper.v: top-level file for testing only. It gives input data from a pre-initialized bram.
--------- image_processor.v: top-level file of image processor
--------- imports/mjpeg/: files for jpeg decoder
--------- edge_detector.v: self-explanatory
--------- vga_controller.v: controls RGB values to display
--------- vga_driver.vhd: IP from Rob Chapman, Altera University Program
--------- ip/: Xilinx ip

------- sim_1: simulation files
--------- imports/vhdl/jpeg_testbench.vhd: testbench for jpeg decoder
--------- wrapper_tb.v: testbench for image_processor_wrapper

--- PWM: Packaged project containing the PWM controller IP. Used for testing the IP with oscilloscope and remote control.

--- vivado-library-master: Files needed for the wifi module

- doc: Documentation including final report and presentation slides

Tips and Tricks I hope to have a page online with Project Tips.

Our main suggestions to future students are:

- If you are using open source IP's, check exactly what functionality is verified by the test benches that come with them. In our experience, the test bench was not thorough, and it would be faster to fix the issues before integrating the component into the larger project to avoid needing to constantly re-synthesize and implement a large project.
- For post-synthesis functional simulations, apply the (* dont_touch = "true" *) to all signals whose value you want to see in simulation.
- Always check your pin assignments as early in the flow as possible; Vivado does not check their validity until generate_bitstream.
- Ensure everyone is using the same version of Vivado; using different versions makes working together as a group difficult.
- Use the lab computers if you can; they are likely to be an order of magnitude or more faster than your laptops.
- If your microblaze is very slow, try using a different version of Vivado.
- If you have black box errors, try adding IPs manually and generating those IP's out of context and generate your block design globally.
- Always generate the Xilinx fifo_generator IP out-of-context. For some reason generating it globally causes the empty signal to be stuck high.
- If you made a change to the RTL and had to resynthesize, always close Vivado and then open it again before running a post-synthesis functional simulation.
- If you can't figure out what is causing an error in SDK/Vivado, reopen the project or even better restart computer