CE392

Project Report

FPGA Based 3D Tracking System

Jerry Xu   Yan Xu   Brian Solmos

Table of Contents

Table of Figures 2

Executive Summary 3

Introduction 3

Broader Considerations 4

Design Constraints and Requirements 4

Design Description 5

D8M Camera Module with Location Tracking 5

RFS Wi-Fi Module 6

Top Level Design 7

Design Optimizations 8

Testing & Simulation 8

Implementation & Synthesis 10

Conclusion 15

Appendices 15

Table of Figures

FIGURE 1 INITIAL SKETCH OF POSSIBLE CAMERA IMPLEMENTATION 3
FIGURE 2 CAMERA AND TRACKING DATAFLOW MODEL 5
FIGURE 3 GRAPHICAL EXPLANATION FOR Z-COORDINATE CALCULATION 6
FIGURE 4 WI-FI MODULE DATAFLOW MODEL 7
FIGURE 5 OVERALL DESIGN DATAFLOW PLOT 7
FIGURE 6 SAMPLE TEST IMAGE 8
FIGURE 7 WAVEFORM FOR THE MULTI-FRAME TESTING 9
FIGURE 8 TESTBENCH OUTPUT FOR THE MULTI-FRAME TESTING 9
FIGURE 9 SCHEMA OF LOCATION DETECTION MODULE (LEFT) 10
FIGURE 10 SCHEMA OF LOCATION DETECTION MODULE (RIGHT) 11
FIGURE 11 TIMING INFORMATION FOR TRACKER MODULE 11
FIGURE 12 TOP LEVEL SCHEMA OF THE VIDEO MODULE 11
FIGURE 13 WI-FI MODULE DETAILED SCHEMATICS 12
FIGURE 14 TOP-LEVEL SCHEMATIC 13
FIGURE 15 MEMORY USAGE BY MODULES 13
FIGURE 16 TRACKER XY AND XZ PLOTS 14
FIGURE 17 TRACKER 3D OUTPUT 15

Executive Summary

The goal of our design is to track a stylus using a 3D coordinate system. Most existing solutions involve expensive multi-sensor systems, and we hope to achieve comparable performance using a single camera tracking a green marker. The core tracking component uses FPGA streaming architecture, which lowers hardware cost and latency. To make the device mobile, the coordinates are sent using a Wi-Fi module, and the verification is performed using Python plotting. The device also supports video output via HDMI to check the camera setup and visualize tracker results. The design has some limitations. We are using a large portion of the onboard memory. That means we cannot use the NIOS II processor to add HUD data display to the video output or achieve higher resolution, both of which would require additional frame buffer space. Additionally, the system might benefit from some noise reduction as the tracker is highly sensitive.

Introduction

Our goal in this project is to build a 3D tracking system that only uses a camera as the source of information. By tracking the location of a green marker and performing some geometric derivations, we should be able to obtain the x, y, and z coordinates of a stylus. Compared to existing implementations on VR devices, this approach greatly reduces the cost and complexity of the system. We will evaluate the performance of this design and make improvements to it.

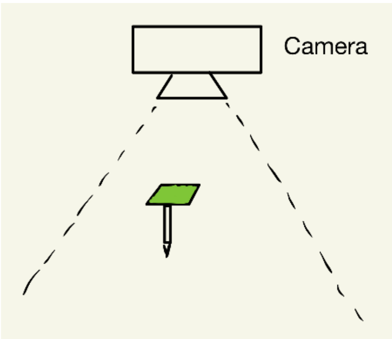

Figure 1 Initial Sketch of Possible Camera Implementation

The FPGA device we have chosen is the DE-10 Nano board. It has an ARM processor for running Linux, two GPIO ports for connecting the camera and RFS Wi-Fi module, and a NIOS II processor used for controlling the Wi-Fi module and interfacing different components. It provides great flexibility and processing power for our project.

The FPGA portion of the project resembles the Sobel unit we designed before. Reading the pixel stream from the camera, our location detection module will locate the green pixels and find the center of the green box. Based on the size of the green box with respect to a reference point, we can derive the z-coordinate of the pen.

We run C code on the NIOS II processor on the DE10 board. The processor allows us to easily control the RFS module, which has a built-in Wi-Fi unit. The processor interfaces with the onboard FPGA, on top of which we build our streaming architecture to get the coordinates. It then sends out the results via UDP packets, which are then captured by a Python program running the drawing code.

Broader Considerations

Our design mainly targets the need for location tracking in VR and AR devices. Most of the current devices rely on complicated AI algorithms or expensive hardware. By replacing those things with a single camera, we want to explore whether we can achieve similar effects with more affordable hardware if we are willing to sacrifice some accuracy. If this approach works well, it could make VR devices more widespread given how cheap cameras have become.

Furthermore, if our design works on an FPGA, it shows that tracking can be performed with small power consumption and low computational requirements. Therefore, some IoT devices like security cameras can also benefit from similar approaches.

Design Constraints and Requirements

For the camera, our only option is the D8M module, which uses the GPIO port for data transmission and can be configured with I2C. We found a sample code that can drive the camera, buffer pixel data, and produce HDMI output. On top of that, we added our location detection module to generate the coordinates. We noticed that because of the limited bandwidth, increasing resolution leads to a reduction in frame rate. Since we are only aiming for crude location detection, we decide to aim for a higher frame rate instead of a higher resolution. Furthermore, because higher resolution requires a larger frame buffer, we want to avoid exceeding the resource limit. In the end, we choose the resolution 640*480 at 60Hz.

For the Wi-Fi module, we chose RFS from Terasic. It uses also uses the GPIO port, which makes the DE10 Nano our only viable FPGA choice given that DE2 only has one GPIO port. An important reason why we chose this hardware is that it comes with the drivers for NIOS II development. Since the Wi-Fi module needs to interface with the FPGA, we have to develop our code with the platform designer tool to allocate the I/O, and that prevents us from using the ARM processor.

For video output, our current design uses HDMI because it is the only option on the DE-10 board. It has the advantage of being able to output at a higher resolution than VGA, which makes it possible to display both a high-quality video stream and captured data. Furthermore, the interface of HDMI is similar to that of VGA, which allows us to use the VGA controller to generate control signals.

Design Description

D8M Camera Module with Location Tracking

The design of the camera module contains the following components. The streaming data from the camera first comes into D8M_LUT for data correction. After that, the data is stored in BRAM temporarily. The reading of video outputs is controlled by the VGA controller. When the controller requests data, the RAW2RGB module converts the raw output to RGB data. The data then gets set to FOCUS_ADJ, which drives the motor for the camera lens to perform auto focus. There is no need for FIFO between them because the data is stored in registers and the processing only takes one cycle in each module. Finally, the data from FOCUS_ADJ goes to our location tracking module. Our tracking module will output the x and y coordinates as well as the width and height of the green region. It has a pixel stream output, which highlights the position of the center of the green region with a red marker. The regular and modified pixel streams can be selected using a switch.

Figure 2 Camera and Tracking Dataflow Model

Next, we are going to explain more implementation details of the location tracking module. This module contains a basic FSM for performing the green-dot detection and producing a pair of center coordinates as well as the width and height of the green region. We first need to detect the green region in our image. The thresholds we use are green channel greater than 100 and red and blue channel values smaller than half of the green channel value. In practice, this threshold works well with a wide range of green colors. If a green region is detected in a frame, we will raise the valid signal when the final output is produced.

The detection part recognizes the leftmost, rightmost, topmost, and bottommost green pixels and does the following calculations. By taking the average of the column numbers of the leftmost point and rightmost point, we can get the x coordinate, and by taking the average of the two row numbers of the topmost and bottommost pixels, we can get the y coordinate. The width and height are calculated by taking the difference between the column numbers and row numbers respectively. The z-coordinate can be found by comparing the area of the square with a reference value. As shown in the figure below, as the object moves, the distances from the object and reference to the camera d and d_{ref} and the width of the reference and the that of the current image l and l_{ref} follow the relation $\frac{d_{ref}}{d} = \frac{l}{l_{ref}}$. This allows us to express the current distance d as $d = \frac{l_{ref} d_{ref}}{l}$. Here, the distance d would be z, and the ratio $\frac{l_{ref}}{l}$ can be found based on the ratio of x and y with reference values. In our design, we used the relationship $\frac{l_{ref}}{l} = \sqrt{\frac{w_{ref}h_{ref}}{wh}}$ to combine the impact of height and width. It assumes that the area should be constant and use the ratio of the area to derive the ratio of the dimensions. The z-calculation will be performed in the Python code to save FPGA resources.

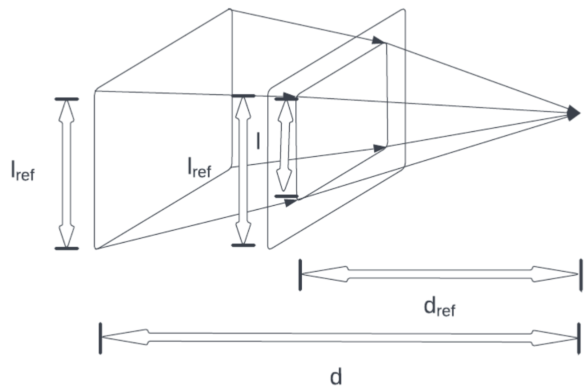

Figure 3 Graphical Explanation for z-Coordinate Calculation

The detection module also outputs a pixel stream. Based on the coordinates detected in the previous cycle, it will place a red square at the center of the green region. The value from the previous cycle is used because this eliminates the need for an output buffer. Furthermore, since the frame rate is 60 Hz, we are unlikely to notice the difference. This allows us to better verify the output of our code. We can choose between this modified data stream and the original pixel stream using a switch.

RFS Wi-Fi Module

The Wi-Fi module is designed with the Platform Designer tool in Quartus. We can instantiate the processor, allocate on-chip memory, and interface with the rest of the FPGA in this tool. A key advantage is that it organizes the memory locations of all the modules and generates a board support package that can be imported to the NIOS II programmer. Using the header in the BSP, we can read and control FPGA using C-style file functions.

In the NIOS II programmer, we can write and run C code on the on-broad processor. The program and its variables will be stored in the on-chip memory. The commands for the Wi-Fi modules are written in the form of strings, and the standard C file interface is used to read data and write commands. We chose the UDP protocol because our location update frequency is high enough and occasional packet drops have no noticeable impact. Compared to TCP, it also results in smaller packet sizes and eliminates the need for a handshake. The RFS module has built-in functions for creating the UDP packet. Therefore, we do not need to worry about that in our code.

Our code first establishes a Wi-Fi connection. It does so by sending instructions to the NIOS terminal, prompting the user to choose the SSID and enter the password. It needs to be under the same network as the receiver PC to communicate as we are using LAN. The IP address of the PC and the port number of the listening program are hardcoded into the C code so that the packets only get sent to one receiver. Since we are testing under our mobile network, hardcoded IP did not present a problem for us because the IP address of our PC is static. After the connection is established, the program will first send the phrase "hello world" to the PC to test the connection. Then, it starts to read the coordinates, width, and height from the FPGA. After getting those values as integers, we combine them into a single string, pack the payload with instructions for the RFS, and deliver the instruction through the driver.

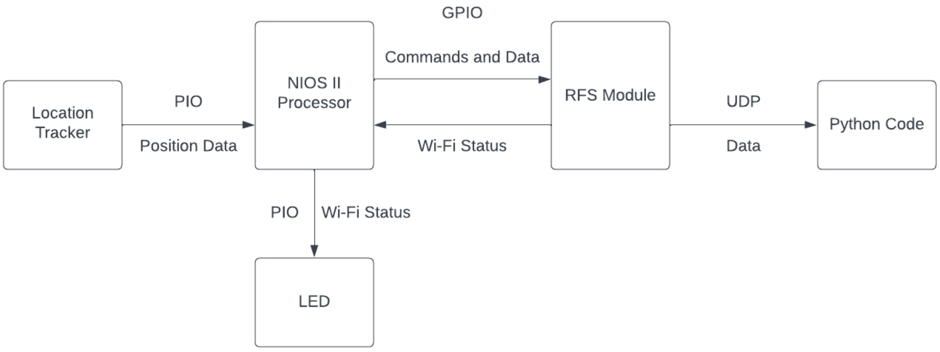

Figure 4 Wi-Fi Module Dataflow Model

The flow of data is shown in the figure above. The location tracker and LED communicates with the processor using PIO, and the RFS module is connected via GPIO. The UDP packets are delivered using Wi-Fi.

Top Level Design

The overall flow of data in our design is shown in the diagram below. The camera sends raw data to the camera decoder, which are RTL designs we got from Terasic. Then, the pixel stream is sent to the location detection module on FPGA, which produces the coordinates for the NIOS II processor. The tracker module also produces a new stream with the center of the green region highlighted. The video stream being displayed is selected with a switch. The final output will be sent via HDMI. The processor will transfer data to our PC through the Wi-Fi module.

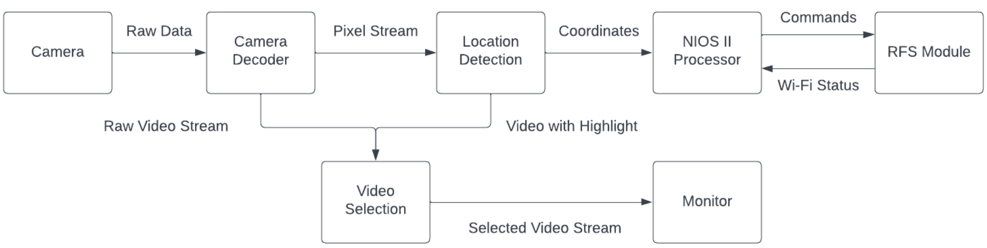

Figure 5 Overall Design Dataflow Plot

Design Optimizations

In the location tracking module, several optimizations are made for presenting more smooth videos and tracker outputs. We observed that the camera writes pixel data at 25MHz, and the state machine is driven by a 50 MHz clock. Therefore, to avoid data buildup in the FIFO serving as the input buffer, our state machine must be able to process one pixel in 2 cycles. Through some optimizations, we were able to reduce the state machine to 2-states, which prevents the FIFO from filling up. We verified that by adding an if-condition in the state machine that produces invalid outputs when the FIFO is full, and we did not observe the output we chose after running the program for 5 minutes. With that optimization, we can reduce the size of our FIFO without worrying about losing data.

The detection algorithm was also improved. Our initial choice of thresholds for RGB were both R-value and B-value should be less than 100 and G-value should be greater than 100. However, we observed that this choice limits our ability to handle different lighting conditions. Therefore, we changed our thresholds to G-value greater than 100 and G and B values less than half of the current G-value. The new flexible thresholds make the output more stable and accurate under a range of lighting conditions. Furthermore, we also used shifters for performing the division, preventing long delay paths caused by dividers.

We also performed some optimization in the Wi-Fi module. We built the code under a range of different system memory capacities to minimize memory consumption. We packed all four data into one packet to reduce the load on the RFS module to enable faster updates.

Testing & Simulation

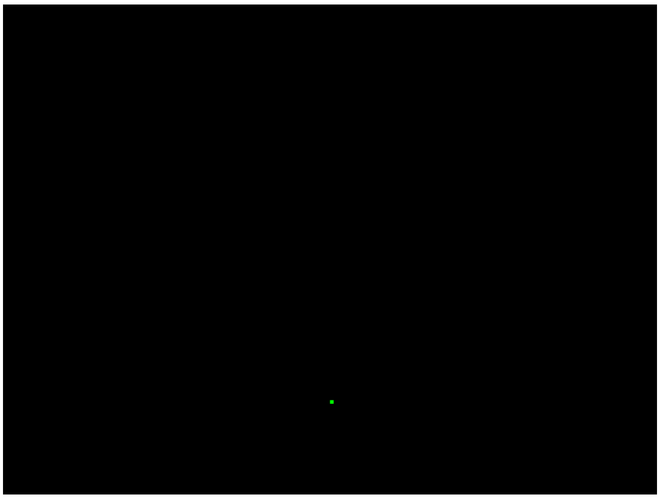

Figure 6 Sample Test Image

We generated our test cases using the file utilities provided in System Verilog. The figure above is an example of our generated image. The background is set to completely black to prevent noise from affecting our results. This image contains a 4-by-4 square at point (362, 101), and it is used in our tests below.

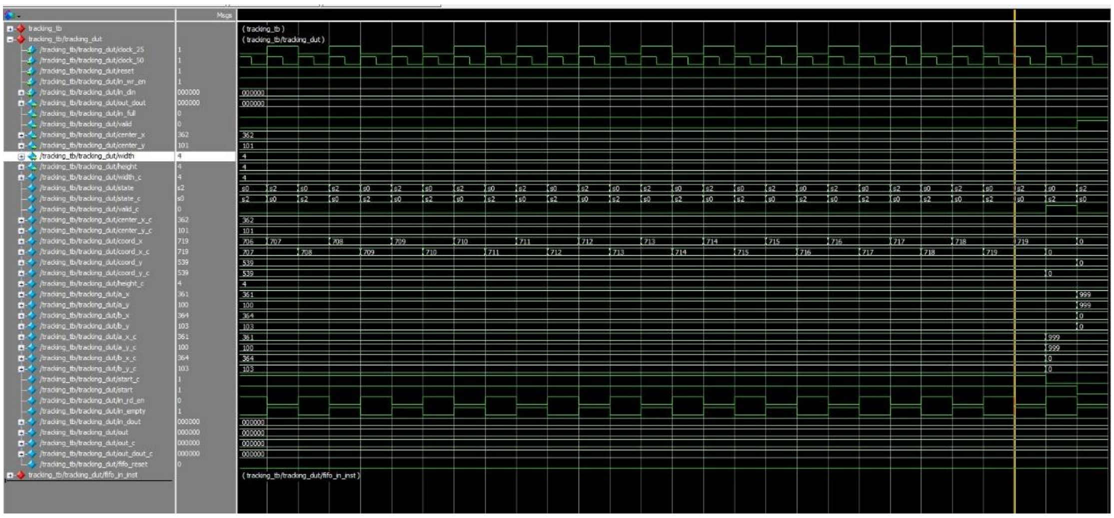

Figure 7 Waveform for the Multi-Frame Testing

```
# @ 20: Loading file green_dot.bmp...
# @ 7776050: center_x is  362...
# @ 7776050: center_y is  101...
# @ 7776050: width_out    4...
# @ 7776050: height_out    4...
# @ 15552050: center_x is  362...
# @ 15552050: center_y is  101...
# @ 15552050: width_out    4...
# @ 15552050: height_out    4...
# @ 23328050: center_x is  362...
# @ 23328050: center_y is  101...
# @ 23328050: width_out    4...
# @ 23328050: height_out    4...
# ** Note: $finish    : ./tracking_tb.sv(222)
#    Time: 23328050 ns  Iteration: 1  Instance: /tracking_tb
```

Figure 8 Testbench Output for the Multi-Frame Testing

Our initial testbench only contains a single-image test case with a 50MHz clock used as both read and write clock. After integrating the module into our main design, we created a multi-frame testbench for the tracker. It tests the module's ability to handle continuous frames without resets. This testbench also has separate clocks for reading and writing to simulate the impact of the system clock and VGA clock. As shown by the testbench output, the module can produce consistent and correct outputs for the continuous inputs, which indicates that it should work in our larger design.

The performance of the Wi-Fi module is limited by the RFS module and the Wi-Fi connection. Our testing shows that if we exceed 10 packets per second, the RFS module will not be able to keep up with the processor. Furthermore, the UDP packets might also get dropped or lost in transmission if we raise the frequency. Therefore, we choose 10Hz as the final frequency for position updates.

As discussed previously, the test we have carried out involves building a connection with the wireless access point, sending an initial greeting, and then updating tracker output at 10 Hz. The result shows that RFS can recognize and connect to our AP created by a Samsung phone at 2.4GHz reliably. The IP address of our PC is static, and the port number can be picked arbitrarily, which allows us to hardcode them into the NIOS II code. Although we are using UDP, we observed no packet loss, which justifies our design decision.

Implementation & Synthesis

Camera Module

First, we will check the design of the location tracker module. In the schema below, we can see that the state machine is correctly instantiated. The comparators are used to check pixel color and highlight center in the output stream, and the adders and shifters are used to perform coordinate, width, and height calculations. All the intermediate variables used in the state machine and output are stored in registers, which help guarantee the timing. Overall, the generated schema fits our expectations.

We also performed the timing analysis on the tracker module. Because the pixels are returned at a frequency of 25MHz and our tracker takes 2 cycles to process one pixel, the tracker only needs to reach 50MHz on its write clock to satisfy the requirement of our design. As shown by the Synplify tool's output, our write clock can be driven with frequencies up to 181.5MHz, which shows that our design has a large overhead.

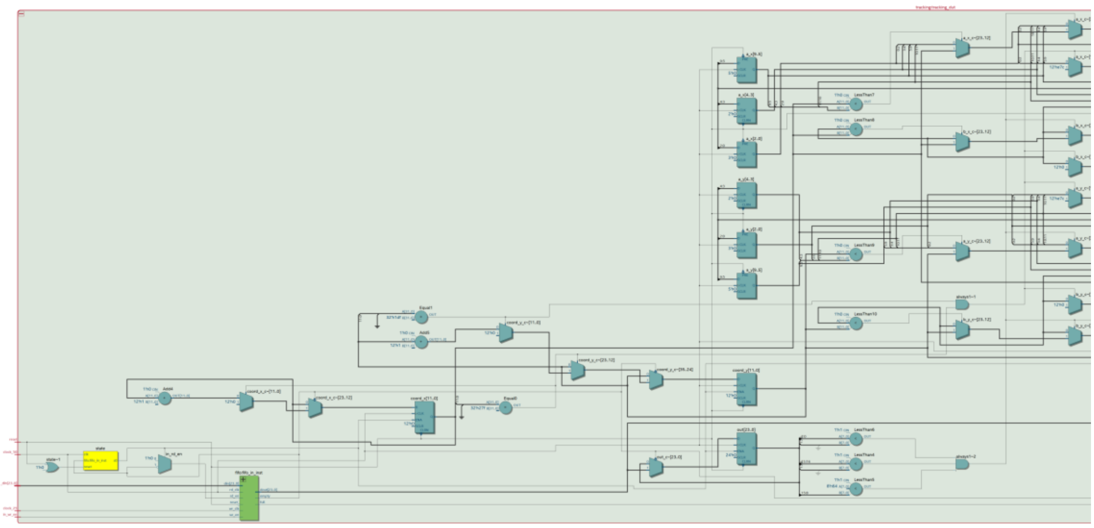

Figure 9 Schema of Location Detection Module (Left)

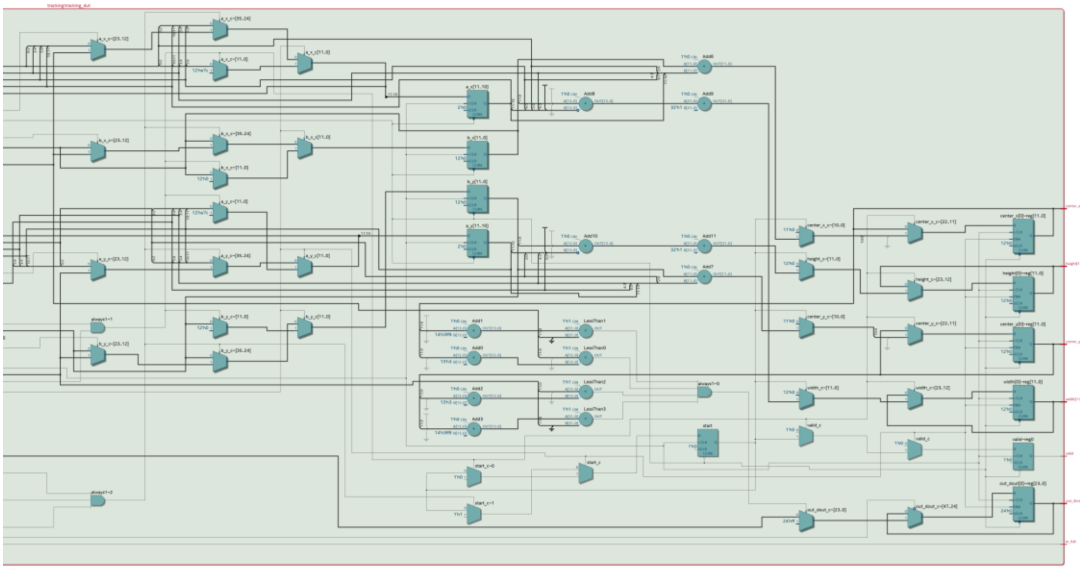

Figure 10 Schema of Location Detection Module (Right)

Area Summary			
LUTs for combinational functions (total_luts)	441	Non I/O Registers (non_io_reg)	189
I/O Pins	102	I/O registers (total_io_reg)	0
DSP Blocks (dsp_used)	0 (15)	Memory Bits	12288
Detailed report		Hierarchical Area report	

Timing Summary			
Clock Name (clock_name)	Req Freq (req_freq)	Est Freq (est_freq)	Slack (slack)
tracking|clock_25	50.0 MHz	288.1 MHz	16.529
tracking|clock_50	50.0 MHz	181.5 MHz	14.491
Detailed report		Timing Report View	

Figure 11 Timing Information for Tracker Module

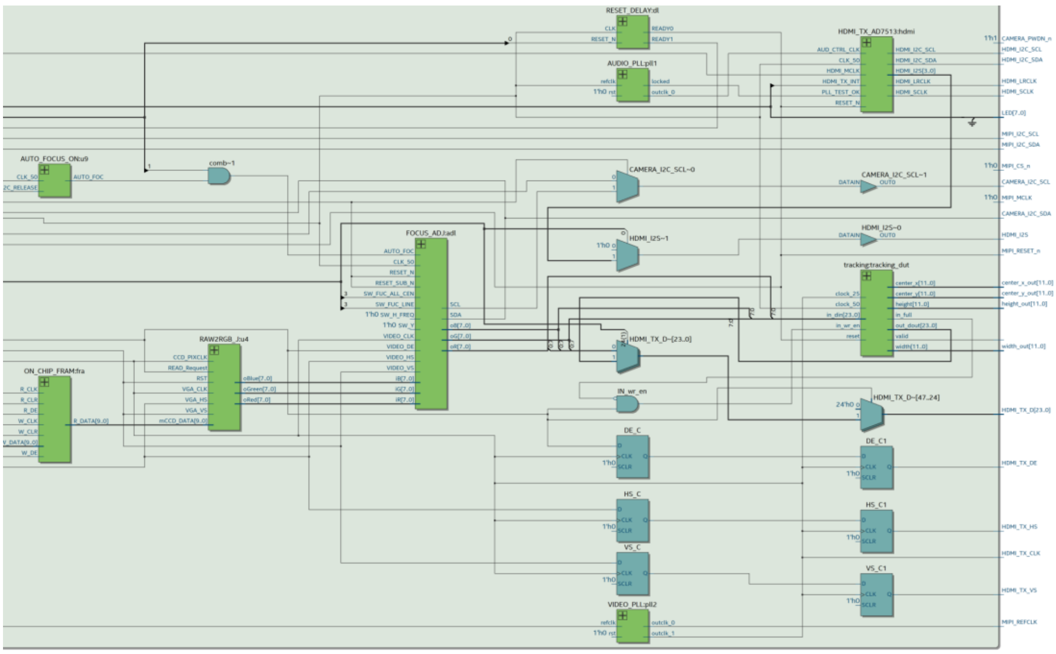

Figure 12 Top Level Schema of the Video Module

The figure above shows the top level of the fully connected video module. All the important modules we discussed are correctly instantiated and connected. We can see that there are two choices for HDMI_TX signal. The pixel can either come from FOCUS_ADJ, which does not have the marker, or tracking, which has the marker. Note that since tracking introduces 2 cycles delay to the pixel values, we need to use two registers to delay the control signals vertical sync, horizontal sync, and HDMI_DE for the display to work. The RESET_DELAY module produces the global reset signal used to make sure that the tracking module does not miss pixels, which can cause pixel distortion.

RFS Wi-Fi Module

In the schema below, we can see that the NIOS II processor, on-chip memory, PIO, RFS control module, Wi-Fi UART, and JTAG UART are all instantiated correctly. We can see that all the data are passed into the interconnected before being sent to the processor, which fits our expectations.

The RFS Wi-Fi module also uses some of the Altera IPs. The QSys design uses ST adapters, error adapters, IRQ, master translator, master agent, slave translator, slave agent, reset controller, and SC FIFO. The processor uses some NIOS II IP, timer, and traffic limiter. The PIO and JTAG UART interfaces both have their own IPs. The interconnect uses demultiplexers, multiplexers, and merlin routers. However, these IPs do not prevent us from generating sof for verification.

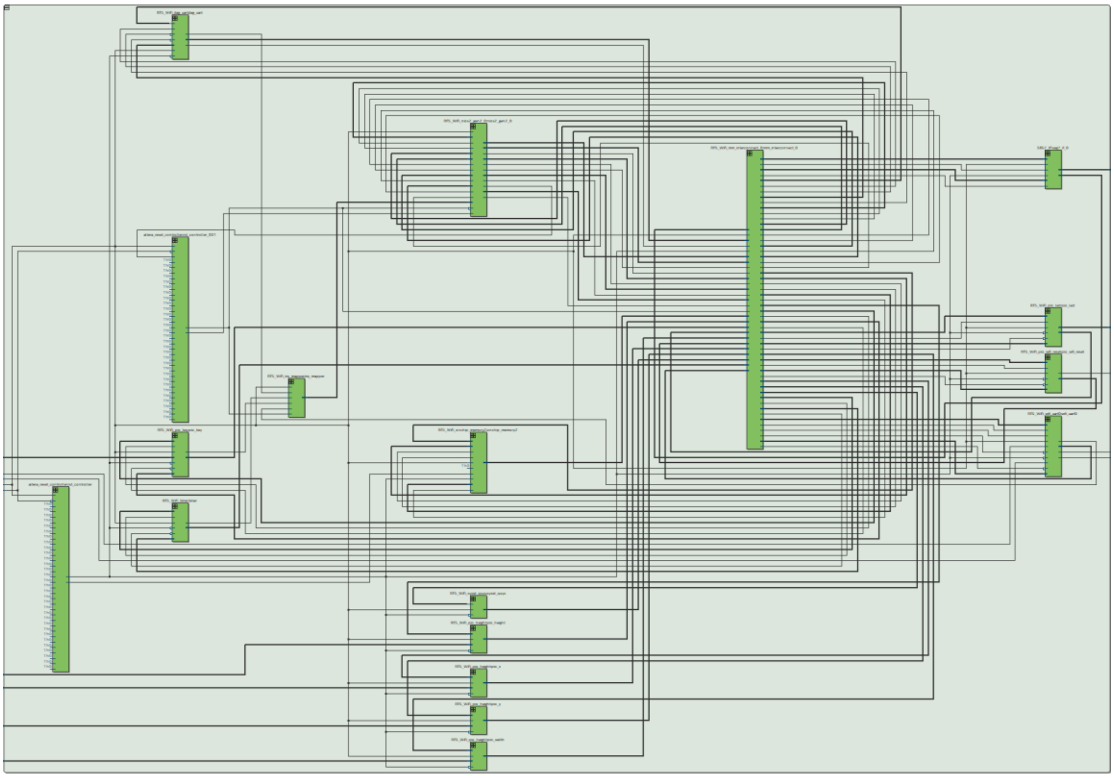

Figure 13 Wi-Fi Module Detailed Schematics

Top-Level Design

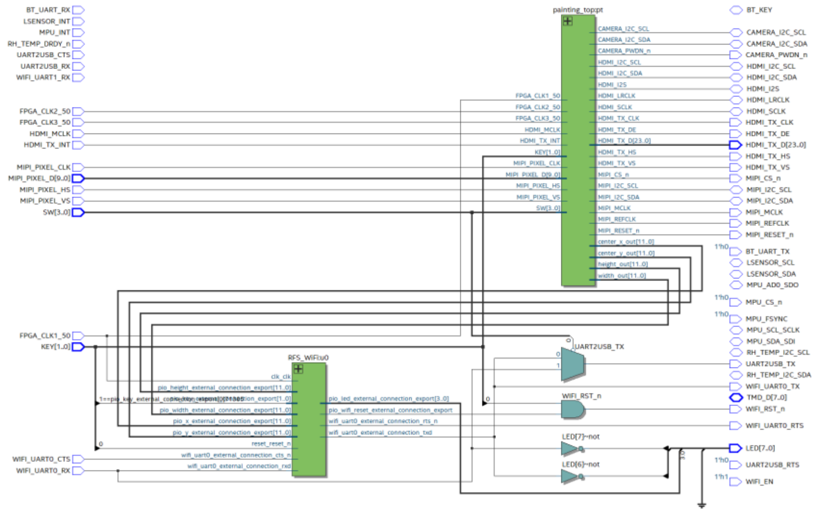

Figure 14 Top-Level Schematic

The top-level schema is shown in the figure above. We packed all the camera-related modules and the tracker in the painting top module. It sends the location data out to the Wi-Fi module, which forwards data to the Python plotter. The necessary connections are instantiated as expected.

Based on the report generated by Quartus, we used 9,638 logic elements, which takes up 23% of the available resource. We used 108 pins, which is 34% of the available pins. Our bottleneck is the block memory. We used 4,317,376 bits, which is 76% of all the available memory. We also used 5 DSP blocks (6%), 16,829 registers, and 2 PLL (33%).

<table>
<tr><th></th><th>Name</th><th>Type</th><th>Mode</th><th>Port A Depth</th><th>Port A Width</th><th>Port B Depth</th><th>Port B Width</th><th>Size</th><th>MIF</th></tr>
<tr><td>1</td><td>RFS_WiFi:u0|RFS_WiFi_jtag_uart:jtag_uart|R...:dpfifo|altsyncram_9bo1:FIFOram|ALTSYNCRAM</td><td>AUTO</td><td>Simple Dual Port</td><td>64</td><td>8</td><td>64</td><td>8</td><td>512</td><td>None</td></tr>
<tr><td>2</td><td>RFS_WiFi:u0|RFS_WiFi_jtag_uart:jtag_uart|R...:dpfifo|altsyncram_9bo1:FIFOram|ALTSYNCRAM</td><td>AUTO</td><td>Simple Dual Port</td><td>64</td><td>8</td><td>64</td><td>8</td><td>512</td><td>None</td></tr>
<tr><td>3</td><td>RFS_WiFi:u0|RFS_WiFi_nios2_gen2_0:nios2_...ltsyncram_rvc1:auto_generated|ALTSYNCRAM</td><td>AUTO</td><td>Simple Dual Port</td><td>256</td><td>2</td><td>256</td><td>2</td><td>512</td><td>None</td></tr>
<tr><td>4</td><td>RFS_WiFi:u0|RFS_WiFi_nios2_gen2_0:nios2_...ltsyncram_66f1:auto_generated|ALTSYNCRAM</td><td>AUTO</td><td>Simple Dual Port</td><td>512</td><td>32</td><td>512</td><td>32</td><td>16384</td><td>None</td></tr>
<tr><td>5</td><td>RFS_WiFi:u0|RFS_WiFi_nios2_gen2_0:nios2_...ltsyncram_7bc1:auto_generated|ALTSYNCRAM</td><td>AUTO</td><td>Simple Dual Port</td><td>64</td><td>11</td><td>64</td><td>11</td><td>704</td><td>None</td></tr>
<tr><td>6</td><td>RFS_WiFi:u0|RFS_WiFi_nios2_gen2_0:nios2_...ltsyncram_dsc1:auto_generated|ALTSYNCRAM</td><td>AUTO</td><td>Simple Dual Port</td><td>8</td><td>32</td><td>8</td><td>32</td><td>256</td><td>None</td></tr>
<tr><td>7</td><td>RFS_WiFi:u0|RFS_WiFi_nios2_gen2_0:nios2_...ltsyncram_ubd1:auto_generated|ALTSYNCRAM</td><td>AUTO</td><td>Simple Dual Port</td><td>1024</td><td>32</td><td>1024</td><td>32</td><td>32768</td><td>None</td></tr>
<tr><td>8</td><td>RFS_WiFi:u0|RFS_WiFi_nios2_gen2_0:nios2_...ltsyncram_v2d1:auto_generated|ALTSYNCRAM</td><td>AUTO</td><td>Simple Dual Port</td><td>128</td><td>16</td><td>128</td><td>16</td><td>2048</td><td>None</td></tr>
<tr><td>9</td><td>RFS_WiFi:u0|RFS_WiFi_nios2_gen2_0:nios2_...ltsyncram_s471:auto_generated|ALTSYNCRAM</td><td>AUTO</td><td>Single Port</td><td>256</td><td>32</td><td>--</td><td>--</td><td>8192</td><td>None</td></tr>
<tr><td>10</td><td>RFS_WiFi:u0|RFS_WiFi_nios2_gen2_0:nios2_...ltsyncram_1bc1:auto_generated|ALTSYNCRAM</td><td>AUTO</td><td>Simple Dual Port</td><td>32</td><td>32</td><td>32</td><td>32</td><td>1024</td><td>None</td></tr>
<tr><td>11</td><td>RFS_WiFi:u0|RFS_WiFi_nios2_gen2_0:nios2_...ltsyncram_1bc1:auto_generated|ALTSYNCRAM</td><td>AUTO</td><td>Simple Dual Port</td><td>32</td><td>32</td><td>32</td><td>32</td><td>1024</td><td>None</td></tr>
<tr><td>12</td><td>RFS_WiFi:u0|RFS_WiFi_onchip_memory2:onc...tsyncram_06d1:auto_generated|ALTSYNCRAM</td><td>AUTO</td><td>Single Port</td><td>35000</td><td>32</td><td>--</td><td>--</td><td>1120000</td><td>None</td></tr>
<tr><td>13</td><td>painting_top:pt|DE10_NANO_D8M_RTL:DE10_...tsyncram_kmj1:auto_generated|ALTSYNCRAM</td><td>AUTO</td><td>Simple Dual Port</td><td>307200</td><td>10</td><td>307200</td><td>10</td><td>3072000</td><td>None</td></tr>
<tr><td>14</td><td>painting_top:pt|DE10_NANO_D8M_RTL:DE10_...tsyncram_87k1:auto_generated|ALTSYNCRAM</td><td>AUTO</td><td>Simple Dual Port</td><td>4096</td><td>10</td><td>4096</td><td>10</td><td>40960</td><td>None</td></tr>
<tr><td>15</td><td>painting_top:pt|DE10_NANO_D8M_RTL:DE10_...tsyncram_87k1:auto_generated|ALTSYNCRAM</td><td>AUTO</td><td>Simple Dual Port</td><td>4096</td><td>10</td><td>4096</td><td>10</td><td>40960</td><td>None</td></tr>
<tr><td>16</td><td>painting_top:pt|DE10_NANO_D8M_RTL:DE10_...tsyncram_87k1:auto_generated|ALTSYNCRAM</td><td>AUTO</td><td>Simple Dual Port</td><td>4096</td><td>10</td><td>4096</td><td>10</td><td>40960</td><td>None</td></tr>
</table>

Figure 15 Memory Usage by Modules

After using our test bench to verify the tracker and testing the Wi-Fi module separately by letting it read switches, we combined our design and put it on the board. We performed a system test to find the problems in our design. We set up the connection to have the camera send live data to our computer. Then, we move a green marker around the screen to see if the plotted locations in Python visualization match the motion of the marker.

Initially, we discovered that the FIFO is reading data, but its empty signal is always high. This was caused by the FIFO using a positive reset signal instead of a negative one used throughout our testbench, and we fixed it by inverting the global reset. Then, we realized that the FIFO filled up too quickly. Two reasons caused that: the FIFO write should use the slower VGA clock at 25MHz, and the state machine is using too many cycles. We changed the write clock and optimized the state machine, solving that problem. We then realized that while the height data is correct, the width is incorrect. Because the width is taken from the row number, which is more sensitive to misaligned frames, we checked the VGA controller and discovered that it does not use our global reset signal. Therefore, we start to read before the buffer is filled with meaningful data, causing our coordinate counter to start too early. After fixing all those issues, our design was fully functional. The coordinates produced match our motion and there is no delay.

We performed some real-world testing and included the results below. The first plot shows the x and y coordinate outputs as we move the marker in a circular path. Considering the human errors included in the testing, we believe that the results indicate that the output of our design is valid. The second plot shows the x and z coordinates as move the marker in an elliptical path in the xz-plane. The path is not spherical because it is hard to control movement in the xz-plane. Nevertheless, the results show that the z-coordinate tracking is functional. Its accuracy can be further improved with additional calibration. The last plot shows the captured points during spiral movements. It shows that our design can generate x, y, and z coordinates simultaneously. With those tests, we are confident about the performance of our design.

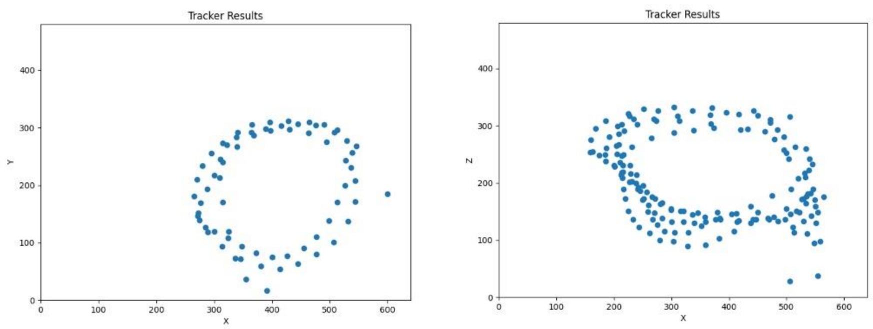

Figure 16 Tracker XY and XZ Plots

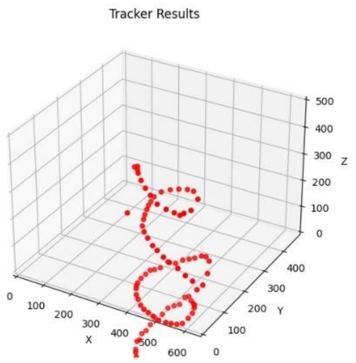

Figure 17 Tracker 3D Output

Conclusion

Overall, our design has fulfilled the design goals we proposed initially. The location tracking is based on the 60Hz 640*480 video stream we planned, and it generates x, y, and z coordinates with reasonable accuracy. Most importantly, the design fits in the DE10-Nano board and the camera and RFS Wi-Fi modules both work as expected.

Some future improvements to this design include adding a data display using the NIOS II processor and supporting higher quality video stream, which would allow for more accurate tracking. However, those improvements would require more advanced hardware. We can also add additional sensors like gyros that can account for tilted markers. Additionally, it is possible to improve our coordinate calculation code to reduce the impact of noise. Some possible modifications include requiring continuous green pixels and ignoring smaller regions.

Appendices

D8M Streaming Design Reference:
https://github.com/grant4001/Sobel_DE10

D8M Camera Specs and Drivers:
http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1011

VGA Timing Guide:
https://projectf.io/posts/video-timings-vga-720p-1080p/

RFS Module Specs and Starter Codes:
https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=65&No=1025

DE10-Nano Specs:
https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1046