Waveshine

A Gesture-Controlled Lighting System

Final Report

Jonas Cameron
MIT EECS
jonasc@mit.edu

Aklilu Aron
MIT EECS
arona@mit.edu

***Abstract*—We present Waveshine, a gesture-controlled lighting system that utilizes active, buffered filtering of video input to perform decision making by interfacing with an array of individually-addressable LEDs. A user of this system is able to select either red, green, or blue channels by displaying a specific shape in front of the camera (i.e. a triangle shape to command the green channel), and then adjust the brightness of that channel by maneuvering the object in specific directions, all in real time. Due to its implementation on an FPGA and direct communication with individual LEDs, Waveshine offers a seamless, high-performance, and efficient experience that highlights the advantages of a hardware implementation as compared to a software control scheme.**

I. System Overview

A. Physical Layer

The physical component of this project is relatively simple. A camera and processing board is connected to the Digilent Nexys A7-100T FPGA Trainer Board via two of the board's Pmod ports. The VGA output port is connected to the lab monitor. The data line on the 8x32 WS2818B LED board is connected to the fourth pin on Pmod port "jc" (jc[3] in the .sv file). The LED array is plugged in to a lab power supply set to 5V to supply adequate power. Common ground is connected to the Nexys board and the LED panel. The physical setup is shown below in figure 1.

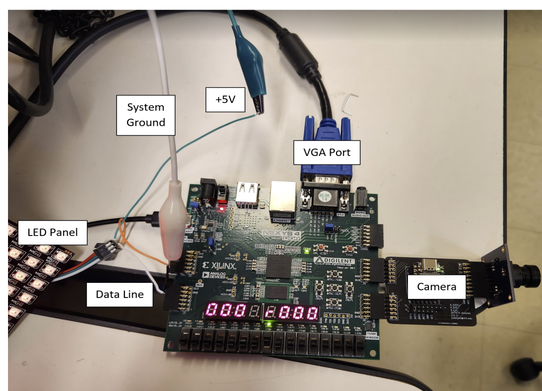


Fig. 1. Physical setup with camera board, FPGA board, VGA monitor and LED array

B. Hardware Implementation

The implementation for Waveshine can be thought of on a high level as containing two major structures: the video processing engine and the LED interface. These sections are connected as indicated in the high-level block diagram shown at the top of the next page (Figure 2). The video processing engine includes existing overhead which was modified from 6.205 lab 4. This includes several modules which translate the input of the camera to a sensible, upright, scaled VGA output. It also includes a *buffer* module which was modified to implement a 10-line rolling buffer of pixel data, effecting a 10x10 sliding window as pixel data is retrieved from the camera. The main piece of the video processing engine is the *detect* module, which takes in 10 pixels at a time from the rolling buffer and tracks how well those pixels match a predetermined hard-coded template. When one of three target shapes is recognized, *detect* sends its identity and position down the pipeline to the next module.

The *tracker* module watches this position over time. When a shape is detected more than once within the span of a few frames, and the position of these detections is different, *tracker* adjusts a 24-bit rgb output accordingly. The *LEDcomm* module continuously monitors this rgb value and translates it to WS2818B serial communication. These modules will be detailed in the following sections.

II. Video Processing Engine

The general purpose of the video processing engine is to take in the raw camera input, filter it appropriately, and extract the necessary information to track shapes. This includes outputting the current pixel location, and if there has been a valid shape detected, then outputting which channel it corresponds to (red, green, blue), and the position of the letter. This output is then sent to the tracker module which will make decisions based on any shape movement. This video processing engine can be broken down into its components. The first portion is the camera pipeline, adopted from the 6.205 course materials [1]. This pipeline includes many small modules important for interfacing with the camera board attached to the FPGA board and appropriately routing the incoming pixels downstream, as well as properly displaying the desired output on a VGA-compatible monitor. This engine includes a redesigned buffer module to handle line buffers of size 10, and the critical

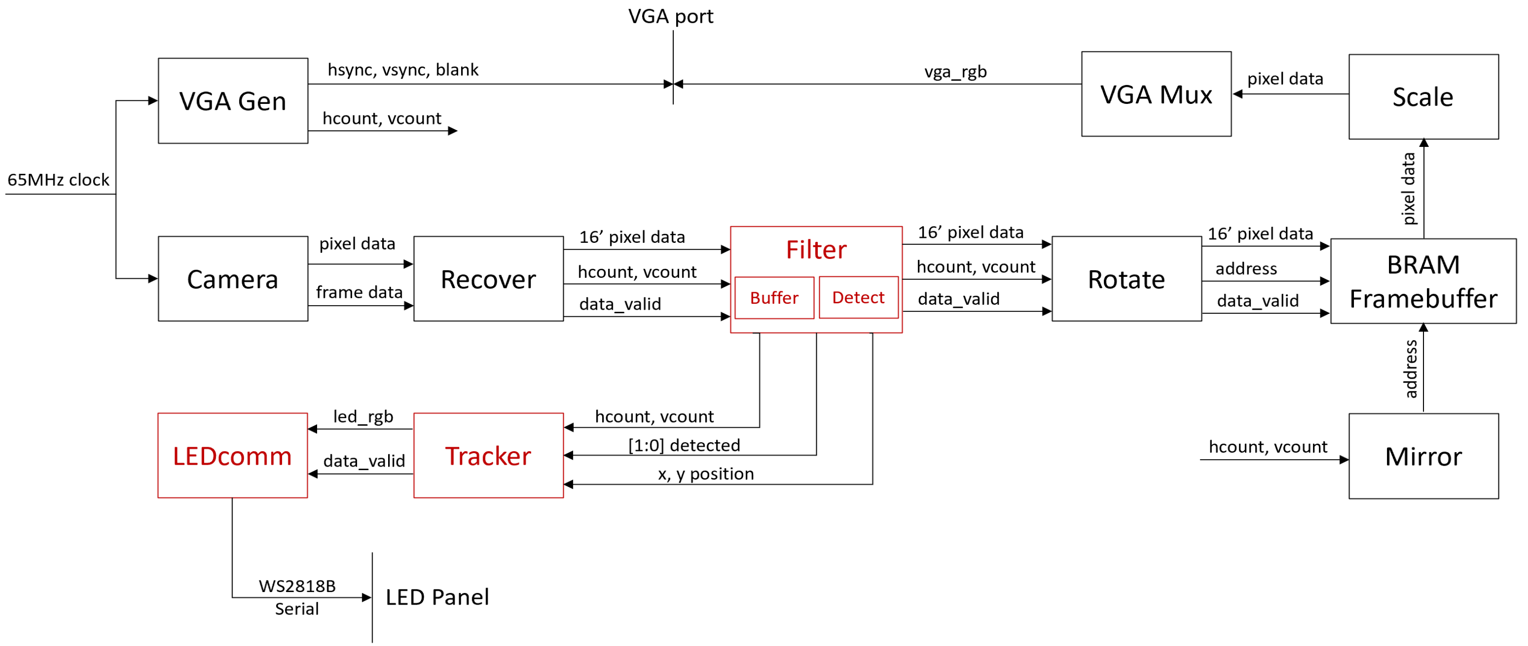


Fig. 2. Mid-level block diagram of the Waveshine system. Original modules in red.

matching filter, which is responsible for interpreting the frames and acting accordingly.

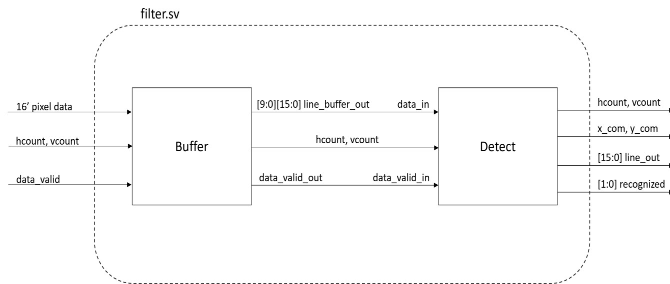


Fig. 3. Internal block diagram of the filter module

A. Buffer Module

The *buffer* module takes inspiration from a smaller version developed by the authors as part of 6.205 lab 4b [2]. The purpose of the buffer is to always have access to the last 10 lines of pixels that were read in. Each time a new pixel is read in, the current line buffers ”roll” over so that the oldest line is dropped and the newest is read in. In this way, at any given point in time, the downstream module has access to a given pixel’s immediate 10x10 surroundings. In order to implement this, 11 BRAMs were used in an 11-state finite state machine (FSM). This is necessary because each buffer’s role changes depending on which stage in the rolling process it is in (i.e. it may be accepting a new input in one state, while it may be simply shifting over data in another). Each BRAM is addressed by the incoming pixel’s hcount so that all off the appropriate pixels can be read out of memory. Finally, the pixels are sent out in buffers of size 10 to be used in the *detect* module.

B. Detect Module

The *detect* module incorporates the main filtering functionality which is responsible for shape detection. The main goal of the module is to reliably detect when a command shape appears in the camera’s view. In order to accomplish this, an appropriate buffer has to be used in order to draw conclusions about groups of pixels. This implementation makes use of the rolling line buffer described above to break up a potential character into a two-stage 100x100 pixel unit and evaluate its contents. Due to limitations in hardware memory and timing, it is not feasible to try and consider all 10,000 pixels at once to determine if we have a shape match. Thus, this system breaks down a given frame into 10x10 pixel blocks. Our template shapes are constructed from 10x10 units of these uniformly white or black 10x10 pixel blocks. Thus, we can consider each 10x10 unit at a time, while at the same time remembering what blocks we have seen before. In this way we can determine whether or not we are seeing a valid shape.

For example, consider trying to identify a triangle shape contained within a 100x100 pixel area on the camera input. If the shape is black on a white background, every pixel could be checked on whether it is a very dark (low value) or very light (high value), and the entire range could be given a score which is then subject to a threshold to determine whether a ’triangle’ is present. However, this robust implementation would require a 100-line rolling line buffer in order to store the necessary info to be sent to the following module. We found this to be unreasonable in code and in memory and resource utilization.

The simplified implementation divides this 100x100 space into 10 10x10 squares, each of which is checked against a uniform black or white template square. The result is a template for the triangle shape as shown below:

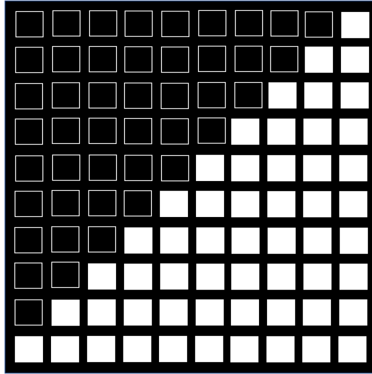


Fig. 4. Possible 100x100 template for the triangle shape (used for controlling green LED levels), split into 10x10 chunks

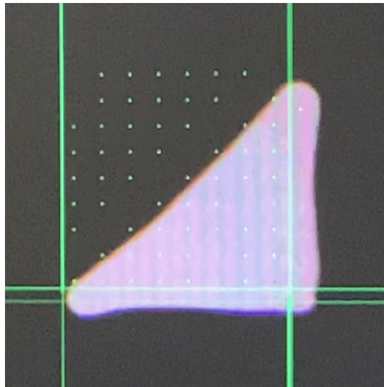


Fig. 5. The triangle (green shape) on the verge of being detected. Each green dot represents a 10x10 square that has matched the template.

It is also important to note here that there is some flexibility in terms of what we consider a ”match”. For each of the 10x10 sub-unit blocks, we can set a threshold count for which if there are at least that many sufficiently (another threshold value) bright or dark pixels, that is a valid block. This flexibility is very useful because in most cases the orientation or distance of the shape will not be perfectly aligned with what the filter is looking for. Thus, there is room for adjusting tolerance to allow for smoother operation. Once we reliably detect if there is a shape, we can record the position of the shape and send it onwards to the *tracker* module to keep track of how the shape moves over time.

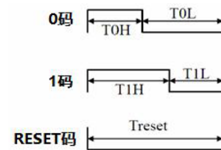
III. Light Interface Module

The *tracker* module takes several key inputs from the detect module: pipelined hcount and vcount, the detected horizontal and vertical position, and a 2-bit register which indicates which shape, if any, has been detected. When a particular shape is detected, its position is stored and the module enters a state corresponding to this shape. The module begins to count frames. If at any point the number of frames counted exceeds a parameterized value, the module returns to its rest state. If the same shape is detected before this frame count, the frame count is reset to zero, the rgb output is altered according to the difference in position, and the new position is stored. In design and practice, we found that a 1:1 scaling between the pixel position and the 8-bit LED color channel was appropriate, both being on the order of about 100. Thus traversing the entire camera range results in a full range of red, green, or blue values. If a detection of one shape is followed immediately by a detection of a different shape in the next frame, the new detection cannot be processed until the parameterized frame count has been exceeded. This occurrence creates the longest pipelined path in the Waveshine system and thus is the extreme of latency. This delay is not noticeable visually.

The role of the *LEDcomm* module is to transform the final desired RGB data into a serial form which can be effectively received by the 8x32 LED panel. This is designed in accordance with the WS2818 communication protocol used by the integrated circuits built into the panel. The module must continually stream data to the LED drivers, and update this stream based on the output of the filter module.

The WS2818 protocol uses a Single-Line Return-to-Zero standard. In order to transmit a bit of value 1, the protocol specifies that a high signal must be sent for a specific amount of time, followed by a low signal for a specific amount of time. Transmitting a 0 involves the same scheme with different timing. These timing specifications can be seen in Figure 6.

The detection scheme being quite stable from frame to frame, *LEDcomm* was modified to not require a rising edge in data-valid-in. Instead, it waits until the previous rgb value has been sent and then looks for any high valid data signal. This technically reduces throughput, as the module cannot take a new rgb input while the previous is being sent. However, given the timing specifications of the WS2818 protocol, an entire LED panel's worth of data can be transmitted within the span of one frame. Thus, with respect to the incoming detection updates, *LEDcomm* has 100 percent throughput.



Sequence Time

T0H	0-code, High-level time	220ns~380ns
T1H	1-code, High-level time	580ns~1.6µs
T0L	0-code, Low-level time	580ns~1.6µs
T1L	1-code, Low-level time	220ns~420ns
RES	Frame unit, Low-level time	> 280µs

Fig. 6. WS2818 Serial Timing Specification

In order to communicate, then, the interface module sends

each bit of the 24-bit RGB input at a time, using the above codes for each bit. To time the signals correctly, the module assumes a 65MHz clock (obtained from an upstream clock transformation module) and waits the appropriate number of clock cycles when holding the output high or low. For example, to send the high portion of a ”1” bit, keep the output high for about 64 65MHz clock cycles, equating to about 0.985 microseconds. This falls comfortably in the 580ns-1.6us range specification. The interface module completes this communication for every bit in the RGB data and repeats the process continually.

IV. Evaluation

When it comes to system speeds, Waveshine is built to perform very well. The throughput of the system is limited by how quickly new pixels can move through the video processing engine. Due to fixed overheads built into the camera pipeline (i.e. 2 cycles for BRAM access return), it takes several clock cycles for a new valid pixel to be presented to the engine. Despite this, because all of the template comparison happens in one cycle, there is no worry of negatively impacting throughput. That being said, the implementation for checking if the current 10x10 is ”white” or ”black” requires doing comparisons and additions on each channel (R,G,B) of 100 pixels in one cycle. We believe Vivado is building distributed RAM or BRAM blocks in order to accomplish this large combinational task, according to the presence of 22 otherwise unexplained 320x16 bit RAMs in addition to 4 (expected) 7-bit 100-input adders. While this was still able to pass timing requirements due to Vivado's superb optimization abilities, its possible that this implementation could be improved by sacrificing some clock cycles to free up some combinational logic, especially because the *detect* module is not the limiting factor on the throughput. However, it is also important to note that the RAM resources on the FPGA board remain as unused space unless they are built, so it is not necessarily a bad thing if the logic takes up more of these resources.

A. Timing

Vivado was able to optimize our design so that the worst negative slack (WNS) in the system is 0.286, with zero total negative slack. WNS was reported as positive in every phase of routing. We expect that the path which limits the speed of the system is through the buffer and detect modules, including the 100-input adders, then through the tracker and LEDcomm modules. The limiting factor is likely the combinational block present in the detect module, and if this were optimized or prioritized differently to sacrifice latency, a clock faster than 65MHz could be used.

B. Resource Utilization

For the most part, Waveshine is not that straining on system resources. According to the build logs (see repo link right before appendix), none of the memory elements were close to being fully used. As mentioned previously, the only portion of the design that does use a significant amount of resources is the fairly large combinational math used in *detect*. This could be an opportunity to use up any extra space to further pipline the math and improve the speeds of the overall system, so long as the resources aren't hogged to the point that they begin to affect performance. For specifics on exact amounts of BRAMs, LUTs, etc, see the Vivado reports in the appendix.

C. Goals and Extensions

Our current implementation met all of our commitments, and nearly all of our goals. We decided to forgo simple color detection for a more sophisticated object recognition system, and were able to get this successfully implement this. The downside was that the shape detection requires a fairly stable camera environment; the best results were obtained with the white shape on an all black surface. For this reason, it did not make much sense to incorporate the auto-brightness feature, because there would not be much of an ambient background from which to draw from. As far as the shape detection, Waveshine can detect the red, green, and blue shapes reliably given the proper conditions. Specifically, the distance from the camera and orientation of the shapes are very important in order to match what the template is expecting. Using multiple different thresholds allowed us to leave some room for error, but it is possible we could achieve better results if we explored a more sophisticated detection technique (true convolution, edge detection, etc). That being said, the current system is very effective, and also is built in a way that would allow for a wide range of expansions. Adding new shapes is as simple as dropping in a 100 bit shape template and corresponding threshold. Furthermore, we are currently only tracking vertical motion to affect LED channel intensity. We could also make use of horizontal movement to incorporate more features, such as sweeping across which LEDs are on, and other neat tricks.

V. Discussion

In retrospect, we have several thoughts:

- As opposed to color detection using robust color spaces and masks, we opted for the more complex option of shape detection. A color detection implementation would undoubtedly be simpler in resources and timing.
- Scaling the VGA output was causing unwanted behaviors in our live shape tracking. Due to time constraints we simply chose not to scale up the output, though with time we could reconcile our modules with this existing functionality.
- Much of the existing camera processing pipeline (mirror, rotate, scale, masking, muxing, etc.) cause issues in our design process. With more time we might have built out our own custom pipeline instead of editing.
- We found that using bright white emitted light on a dark background worked best for our shape detection algorithm. The surrounding environment slightly affected our results through reflections. Some filtering or unique coloring (i.e. pink shapes) could possibly alleviate these issues and enable a solution more robust to background noise.

- There are other methods of shape detection, such as edge detection using convolved kernels, that might be interesting to explore and compare in the future.
- An 8x8 or 16x16 sliding window, as opposed to 10x10, might have been more ”Vivado friendly” and resulted in a slimmer build.

VI. Contributions

In terms of implementing the Waveshine system, Aklilu was responsible for developing the video processing engine. This included revamping the *buffer* module, writing the *detect* module, and adjusting the entire camera pipeline as necessary to support Waveshine, as well as drawing the template shapes and experimenting to find successful threshold values. Jonas was responsible for developing the LED interface. This included researching the WS2818 protocol to understand the operation of the LED panel, writing the *LEDcomm* module to properly implement communication with the LEDs, and writing the *tracker* module to interpret position information from the video engine and send commands to *LEDcomm*.

For the report, both authors were generally responsible for writing about the portions of Waveshine that they developed, as well as relevant evaluation topics. In addition, Jonas created some figures to be displayed in the report and edited together the final video.

The full source repository for Waveshine can be found at https://github.com/akliluaron/Waveshine

References

[1] Author: 6.205 Fall 2022 Teaching Staff, “Lab 04A: Video Processing”, 6.205 Website October 2022.
https://fpga.mit.edu/6205/F22/labs/lab04

[2] Author: 6.205 Fall 2022 Teaching Staff, “Lab 04B”, 6.205 Website October 2022.
https://fpga.mit.edu/6205/F22/labs/lab04b

[3] Author: Worldsemi, “WS2818A Single-line 256 Gray-level 3-channel Constant Current LED Driver IC”,
https://www.tme.com/Document/1d930d9b83e8cce43e5d1c490ab0fbe8/WS2818A.pdf

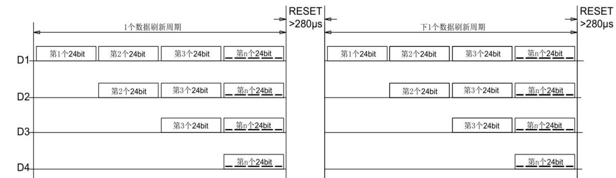
VII. Appendix A: Vivado Report on Resource Utilization

```
Report Cell Usage:
+------+-----------+------+
|      |Cell       |Count |
+------+-----------+------+
|1     |BUFG       |     2|
|2     |CARRY4     |   130|
|3     |DSP48E1    |     9|
|8     |LUT1       |    52|
|9     |LUT2       |   527|
|10    |LUT3       |   663|
|11    |LUT4       |   711|
|12    |LUT5       |  1514|
|13    |LUT6       |  5315|
|14    |MMCME2_ADV |     1|
|15    |MUXF7      |   836|
|16    |MUXF8      |   154|
|17    |RAMB18E1   |    22|
|18    |RAMB36E1   |    48|
|21    |SRL16E     |    58|
|22    |FDRE       |  4353|
|23    |FDSE       |    32|
|24    |IBUF       |    29|
|25    |OBUF       |    51|
+------+-----------+------+
```

```
---------------------------------------------------------------------------------
Detailed RTL Component Info :
+---Adders :
	   2 Input   32 Bit       Adders := 1
	   3 Input   24 Bit       Adders := 3
	   2 Input   24 Bit       Adders := 3
	   4 Input   22 Bit       Adders := 2
	   2 Input   17 Bit       Adders := 1
	   3 Input   16 Bit       Adders := 1
	   3 Input   13 Bit       Adders := 3
	   3 Input   12 Bit       Adders := 3
	   2 Input   11 Bit       Adders := 8
	   3 Input   10 Bit       Adders := 2
	   2 Input   10 Bit       Adders := 6
	   2 Input    9 Bit       Adders := 2
	   3 Input    8 Bit       Adders := 2
	   2 Input    7 Bit       Adders := 16
	 100 Input    7 Bit       Adders := 4
	   2 Input    5 Bit       Adders := 2
	   2 Input    4 Bit       Adders := 2
	   2 Input    2 Bit       Adders := 1
+---Registers :
	               32 Bit    Registers := 12
	               24 Bit    Registers := 2
	               22 Bit    Registers := 2
	               19 Bit    Registers := 2
	               17 Bit    Registers := 1
	               16 Bit    Registers := 324
	               12 Bit    Registers := 4
	               11 Bit    Registers := 24
	               10 Bit    Registers := 35
	                9 Bit    Registers := 1
	                8 Bit    Registers := 7
	                7 Bit    Registers := 14
	                5 Bit    Registers := 2
	                4 Bit    Registers := 5
	                2 Bit    Registers := 4
	                1 Bit    Registers := 110
+---RAMs :
	             1200K Bit   (76800 X 16 bit)          RAMs := 1
	              512K Bit   (65536 X 8 bit)          RAMs := 1
	                5K Bit   (320 X 16 bit)          RAMs := 22
	                3K Bit   (256 X 12 bit)          RAMs := 1
+---Muxes :
	   2 Input   32 Bit        Muxes := 1
	   2 Input   24 Bit        Muxes := 10
	   4 Input   24 Bit        Muxes := 1
	   3 Input   16 Bit        Muxes := 3
	   2 Input   16 Bit        Muxes := 12
	   4 Input   16 Bit        Muxes := 2
	   2 Input   12 Bit        Muxes := 4
	   4 Input   12 Bit        Muxes := 3
	   2 Input   11 Bit        Muxes := 11
	   2 Input   10 Bit        Muxes := 3
	   2 Input    9 Bit        Muxes := 2
	   3 Input    9 Bit        Muxes := 1
	   3 Input    7 Bit        Muxes := 2
	   2 Input    7 Bit        Muxes := 6
	   5 Input    7 Bit        Muxes := 2
	   8 Input    7 Bit        Muxes := 1
	   2 Input    6 Bit        Muxes := 6
	   8 Input    6 Bit        Muxes := 1
	   2 Input    5 Bit        Muxes := 2
	   3 Input    5 Bit        Muxes := 1
	   2 Input    4 Bit        Muxes := 5
	  20 Input    2 Bit        Muxes := 1
	   4 Input    2 Bit        Muxes := 3
	   8 Input    2 Bit        Muxes := 2
	   2 Input    2 Bit        Muxes := 14
	   2 Input    1 Bit        Muxes := 275
	   3 Input    1 Bit        Muxes := 8
	   4 Input    1 Bit        Muxes := 18
	   7 Input    1 Bit        Muxes := 5
	  11 Input    1 Bit        Muxes := 6
	   6 Input    1 Bit        Muxes := 4
	   8 Input    1 Bit        Muxes := 8
	  10 Input    1 Bit        Muxes := 4
---------------------------------------------------------------------------------
Finished RTL Component Statistics
---------------------------------------------------------------------------------
```

VIII. Appendix B: WS2818B Protocol

Data Transmission Method



Note: D1 is the data from MCU, and D2, D3, D4 are from Cascade Circuits.

Composition of 24bit data

R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0

Note: Data transmit in order of RGB, high bit data is first.