University of Central Florida

Senior Design I

Final Report

DeepGate Speech Recognition System

Group 3:
Lindsay Davis
Estella Gong
Michael Lopez-Brau
Cedric Orban

Supervisor:
Dr. Lei Wei
Associate Professor
Department of Electrical and
Computer Engineering

Sponsors:
SoarTech

Submitted April 27, 2017

Contents

List of Figures

List of Tables

1 Executive Summary

People have the remarkable capability of being able to make vast inferences about the world with little information, as well as being able to communicate the information gained from these inferences to others. How are humans able to effectively use language, an unfathomably flexible and loose communication tool, to exchange and develop ideas? Is it possible to train machines to communicate or at least understand the vagueness of natural language? These questions and many like them have piqued the interest of many linguists, computer scientists, and engineers, leading to research thrusts in computational linguistics and natural language processing.

At the intersection of computational linguistics and natural language processing, a variety of algorithms have been developed in the search to find the best one for speech recognition. Recent advances in hardware have caused a resurgence of development in deep neural network modeling, now commonly known as deep learning. In the past, neural network models with many hidden layers were unfeasible due to their computational complexity but now many of the most common deep learning algorithms can be run on a middle-to-high end laptop. Deep learning approaches have become very popular as of late due to their ability to dominate other machine learning algorithms on almost every benchmark. We have noticed that these algorithms also perform exceptionally well in the language domain, particularly in speech recognition.

Speech recognition systems generally deal with several important steps, namely: data acquisition and pre-processing, recognition/decoding, and application interface (e.g. moving a prosthetic arm). Conventional speech recognition systems extract expert features in their pre-processing step and use some variant of hidden Markov models as their speech recognizer/decoder. With the popularity and success of deep learning, we utilize a deep learning framework with the goal of learning the best features for speech recognition, instead of hand-crafting expert ones. We designed a system that mirrors current research thrusts by implementing this algorithm on an FPGA chip. FPGA chips have the advantage of being significantly cheaper than ASIC variants for small-scale applications. They also have the ability to run a neural network faster than a CPU and with less power consumption than a GPU.

In this project, we focus on the design and application of DeepGate, a speech recognition system that is affordable, energy-efficient, low-cost, and portable while being able to maintain the computational sophistication needed for keeping the word error rates as low as possible. Implementing DeepGate on a low-cost FPGA allows us to meet these efficiency benchmarks while the deep learning framework allows us to achieve a reasonably low word error rate. DeepGate has a small vocabulary of 14 pre-registered words, corresponding to the numbers 0-9 and the cardinal directions: north, south, east, and west. The speech signal will first be pre-processed by a desktop or laptop computer before sending the pre-processed data over to the FPGA. The FPGA will perform the classification task and one of the

14 LEDs corresponding to our 14 words will illuminate. The FPGA will also send the classification results back to a GUI. We developed a GUI for interfacing with DeepGate that displays the speech converted to text and 14 buttons to simulate the LEDs on the FPGA PCB. Generally, our motivation for this project stems from its interdisciplinary nature. Speech recognition technologies often employ tools from a myriad of disciplinary areas. Among them are computer hardware, linear algebra, signal processing, and machine learning.

2 Project Description

2.1 Project Motivation

It is clear that the future of the field-programmable gate-array has only begun to unravel. This is evidenced by Intel's recent acquisition of Altera and the reports of Microsoft using FPGAs to accelerate their search engine and server infrastructure. Furthermore, researchers across the world have begun exploring the possibility of adapting machine learning algorithms to programmable logic devices, increasing throughput and lowering power consumption relative to CPU-based implementations of the same algorithms. These trends, coupled with the recent unveiling of Google's tensor processing unit (a specially designed integrated circuit optimized for machine learning) have inspired us to create our own FPGA-accelerated deep learning system. By coupling our knowledge of digital systems design, machine learning, software development, and printed circuit boards, we aim to implement a fully functional speech recognition tool using an FPGA-based neural network algorithm.

The end-result of this project is likely not commercially viable. There exist many speech recognition systems that utilize machine learning and have a state-of-the-art accuracy level and latency. Nonetheless, we aim to achieve the highest possible through-put and accuracy possible, showcasing the capabilities of FPGAs as power-efficient compute engines in the process. In this way, the project serves more as a proof-of-concept than anything else. We believe our project has the opportunity to be more interesting than the microcontroller/custom PCB designs that are more typical in senior design. Although taking digital design and computer architecture courses is necessary to graduate from the college, rarely do the senior design instructors push students to apply the concepts learned in these classes to FPGAs or purpose-built PCIe cards.

Additionally, this is a capstone project required by the University of Central Florida's College of Engineering and Computer Science and the ABET accreditation board. In order to obtain our Bachelor of Science in Electrical Engineering/Computer Engineering degrees, we must show the successful application of electrical and computer engineering skills in a large, group-based project. These include, but are not limited to, soldering, printed circuit board design, circuit prototyping, digital design, FPGA programming, algorithmic design, and software engineering. We believe the scope of our project encompasses all the of the aforementioned skills while simultaneously allowing us to exercise engineering judgement, showcase our ability to collaborate as team members, and utilize our technical writing skills. Moreover, the project requires us to produce something within design, financial, and ethical constraints, pushing us to further our knowledge.

Our group consists of three electrical engineering students and one computer engineering student. After careful deliberation, we decided to go with a USB-peripheral PCB containing an FPGA (similar to an FPGA development board). This system allows us to integrate the concepts we find most important or useful for future em-

ployment while allowing all of us to contribute individually in a meaningful way.

2.2 Goals and Objectives

Table 1: DeepGate Objectives

Objective	**Area**	**Priority**
Process speech audio data fast enough for real-time recognition.	O	High
Implement the deep neural network algorithm on a low-cost FPGA (<$100).	FW	High
Store weights using on-chip block RAM only.	FW	Low
Implement a serial communication protocol with a high data transfer rate.	SW/FW	Medium
Keep printed circuit board's power consumption under 10 W.	HW	Low
Develop a highly flexible digital architecture that allows for quick neural network modification.	FW	High
Implement design on a printed circuit board with an area less than 8 inches squared.	HW	Medium
Keep circuit board layer count less than or equal to 4.	HW	Medium
Implement GUI to allow for testing Integer and Cardinal Direction speech recognition	SW	High
Implement Status Logging for GUI feedback	SW	Low
Regular weekly check in with members	O	Medium
Regular one-on-one meetings with each member	O	Medium
Biweekly schedule review and update	O	Medium

Key:

- O - Overall goal
- HW - Hardware goal
- FW - Firmware goal
- SW - Software goal

2.3 Specifications

- Hardware
  - PCB with dedicated memory and power supply
  - PCB shall include:
    * FPGA
    * SDRAM
    * JTAG-SMT2 Programming Chip
    * LEDs
    * Passive Circuit Elements (Resistors, Capacitors)
    * Power Supply Circuit
    * DC Power Jack
    * Reset Button
    * Crystal Oscillator
    * RS232 Serial Port
  - PCB shall be compact (< 8 inches squared surface area.)
  - PCB shall consume power efficiently, under our maximum of 20 W.

- FPGA
  - The FPGA shall be within our budget range ideally between 10 and 100.
  - FPGA will need to process data fast enough for real-time voice recognition.
  - FPGA firmware will need to be flexible enough to accommodate changes in neural network algorithm.
  - Firmware shall use both on-chip and off-chip RAM for data/weight storage.

- Software
  - Speech signal dataset for training the deep neural network
  - Command set for when it should start
  - Deep neural network shall decode the speech of at least one person per second
  - Deep neural network shall correctly decode speech signals based on the type of hardware and size of the network. Currently we will set this threshold at 0.5.
  - Detect and filter out any background noise
  - PC with microphone to analyze voice signals

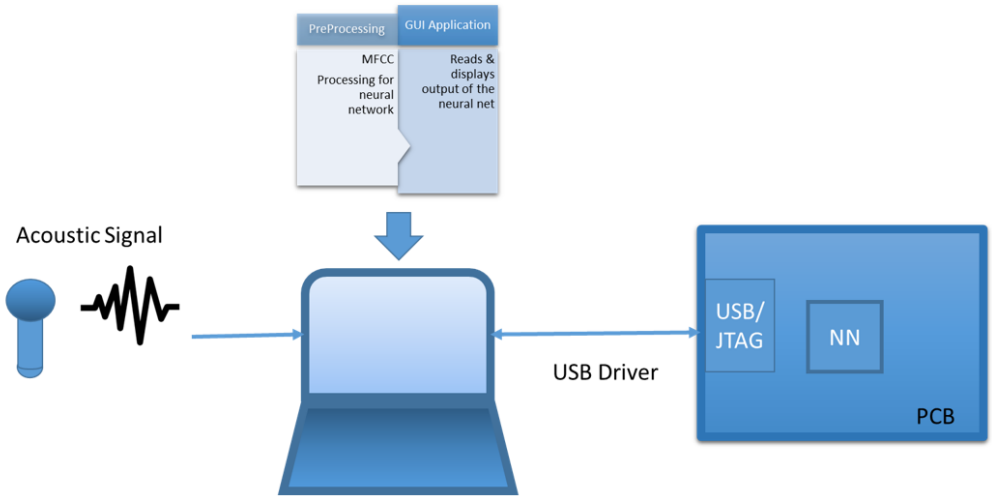
2.4 System Block Diagram

Our project can be broken down into 4 parts:

- Acoustic input, which involves hardware (mic) and pre-processing (filtering, maybe some DSP).
- Recognition, or decoding stage. This includes developing the DNN model and coding it on the FPGA.
- Designing the PCB for the FPGA, mic, and any other peripherals we decide to use, like USB ports.
- Application we decide to use the speech input on.
  - Accurate display/confirmation of spoken integers
  - Accurate display/confirmation of spoken directions (forward, backward, left, right)

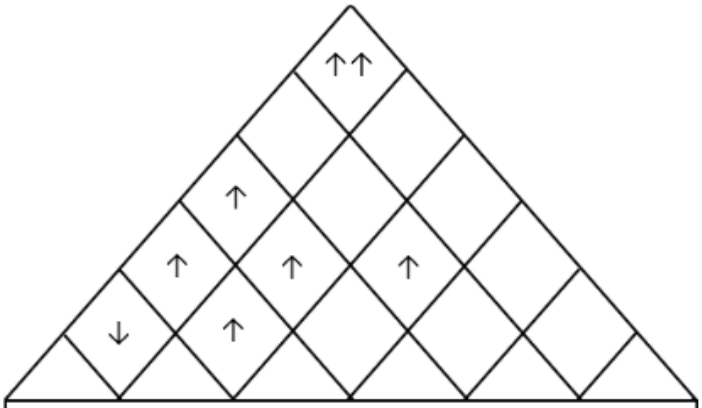
The figure below depicts our overall system block diagram.

Figure 1: System Block Diagram



2.5 House of Quality Analysis

Figure 2: DeepGate House of Quality



			Engineering Characteristics					
			FPGA Size	Latency	Power Consumption	PCB Dimensions	Data Rate	Neural Network Algorithm Flexibility
			+	+	+	+	+	+
Marketing Requirements	Performance	+	↑↑↑	↓↓	↑	↑↑	↑↑↑	
	Power Consumption	+	↑↑	↑		↑	↑	
	Accuracy	+	↑↑		↑			↑
	Cost	+	↑↑↑	↓↓	↑	↑↑		
	Ease-of-Use	+					↑	↑↑↑
Objective Measures	Measurement Unit		# of Slices	ms	W	in^2	kB/s	Hrs/ Change
	Target		< 1500	10	10	8	100	0.5
	Technical Difficulty (1-10)		8	6	3	6	9	9.5
	Relative Importance (1-10)		7	9	2	4	8	8

FPGA Size – The amount of logic available for use on the FPGA. Cost typically goes up as the number of logic resources increase. This directly correlates to the size of our neural net and whether or not signal processing will be performed on the FPGA. Size is measured in logic blocks. These logic blocks have different names depending on the manufacturer. Altera uses Adaptive Logic Modules (ALMs) while Xilinx uses Configurable Logic Blocks (CLBs). The relation between cost and size is not exactly linear. For example, a chip with 50000 CLBs goes for $300 while a chip with 60000 CLBs could go for five times that. It depends greatly on factors like power consumption, speed, and the chip's thermal characteristics.

Performance – How quickly our neural network can make inferences based on speech data. We aim for a minimum speed of 100 frames per second with our neural network. Real-time speech recognition typically uses a 10 ms frame size, meaning our design would be capable of decoding the speech of at least one person per second.

Accuracy – The ability of our device to correctly decode speech signals. This is almost directly correlated with the size of our neural network, which is in turn correlated with the size of the FPGA we use and the amount of money we are willing to spend on a chip. Metrics relating accuracy and FPGA size are hard to come by, as not much research has been done on this topic. This is something that we will update continuously as we design a neural network prototype.

PCB Dimensions – PCB size, including depth, width, and layers, are heavily correlated with cost. Every 2 layers added to our PCB comes with an increase in manufacturing costs of around $200. Greater size implies greater power consumption as well, as traces are longer and signals need to travel farther. Different FPGA packages require different dimensions. For example, using a ball-grid array package would require at least 4 layers to implement on a PCB while a quad-flat package would possibly let us get away with using 2 layers.

Power Consumption – FPGA vendors provide tools that allow you to estimate the power consumption of your design. It is difficult to say what our numbers looked like as we have not fleshed out a full prototype neural net. Typically power consumption is linearly correlated with the clock frequency on the FPGA and the amount of logic resources utilized. For example, using 10000 CLBs clocked at 100 MHz uses 4 times as much power as using 5000 CLBs clocked at 50 MHz.

Ease-of-Use – Ideally, the end-user will be able to specify custom convolutional neural network parameters using our GUI interface and have the FPGA development software generate programming files based on those parameters in less than half-an-hour. Additionally, the neural network should be minimally dependent on vendor-specific primitives to encourage portability across different chips.

3 Research Related to Project Definition

3.1 Existing Similar Projects and Products

Speech recognition systems have been around since the 1950s, where the technology was limited to single-speaker systems. Since then, much research has been done on the science of speech perception and production, and new algorithms have been developed and applied to improving speech recognition systems, such as dynamic time warping (DTW), Hidden Markov models (HMM), and neural network models. Our project specifically uses a deep neural network architecture known as a deep feedforward neural network. The efficacy and shortcomings of each algorithm are discussed in a later section.

With respect to software implementations of these algorithms, field-programmable gate arrays (FGPA) have been rising in popularity, showing advantages over previous CPU and GPU implementations. FPGAs offer the flexibility of being reprogrammable, something that application-specific integrated circuits (ASIC) cannot boast. Additionally, FGPAs manage to be faster than CPU implementations while being significantly more power-efficient than GPU implementations.

Our project takes recent developments in both deep learning and FPGAs to design an implementation that is low-cost, power-efficient, fast, and accurate. Due to the nature of our project, projects and products that are similar are few and far between. With these metrics in mind, we aimed to find existing projects and products that had some of these qualities for comparison.

3.1.1 FAIR's Wav2Letter

The Facebook Artificial Intelligence Research (FAIR) team recently uploaded a paper on arXiv, *Wav2Letter: an End-to-End ConvNet-based Speech Recognition System*, based on an improved version of the algorithm discussed in the previous section.

3.1.2 Microsoft Catapult

Working with FPGAs in the area of artificial intelligence is exciting and also new. A recent Microsoft "moonshot project" known as Catapult involved expanding their use of field programmable gate arrays (FPGAs) to enhance their Bing and Azure products. Their system has allowed them to implement algorithms directly on to the hardware. This was a huge success for the team as not only can it be more efficient than intermediary software, but also highly responsive to the latest innovations in artificial intelligence.

The inspiration for the project came from Doug Burger, Microsoft Research engineer, who was looking to utilize technologies that were not afflicted by the ever slowing improvement in silicon chips as predicted by Moore's Law. Utilizing FPGAs for cloud computing is unprecedented at this scale. FPGAs are used to make data flow faster and more efficient by using it to interface with the network and the

servers. There the traffic can be managed as well as communications to other FPGAs or servers can be made.

As Derek Chiou, head of the Microsoft Azure's Cloud Silicon Team said, "I think a lot of people don't know what FPGAs are capable of." Our project can provide inside and a glimpse into those capabilities.

3.1.3 Hardware-Accelerated Neural Network

Our design is influenced by the architecture published in "FPGA based implementation of deep neural networks using on-chip memory only," a paper written by Jinhwan Park and Wonyung Sung presented at the 2016 IEEE International Conference on Acoustics, Speech and Signal Processing. Park and Sung make use of the processing unit/tile scheme our system uses but leave out the majority of the technical details concerning structural connections, data flow, and control patterns. Finding their results satisfactory (character/phoneme recognition accuracy) we make use of a 3-bit weight scheme and sig_337p sigmoid function implementation. Their design serves as a good indicator of what can feasibly be done using a field-programmable gate array. We add to their base design by developing our architecture from the ground-up with the goals of higher accuracy and more flexibility in mind (while having less hardware resources at our disposal). In addition, certain assumptions had to be made about the way they performed their computation. For instance, they did not explain if input data and intermediate/output node values pre-sigmoid were normalized to a certain range. The major difference between our design and theirs is that they implement a feed forward neural network where as we aim to construct a convolutional neural network. However, the systolic array based architecture can work well for both situations, even if some minor nuances must be addressed.

According to their paper, an accuracy comparable to CPU and GPU-based neural network implementations was observed while experiencing a huge decrease in power consumption moving from a GPU to an FPGA for computation.

3.2 Relevant Technologies

3.2.1 Digital Systems

Digital systems are electronic circuits that handle discrete signals (signals whose voltages fall in discrete bands). The vast majority of digital circuits today deal with binary signals (signals whose voltages fall into one of two voltage bands typically called high and low, or true and false). A "true" signal has a voltage near the circuit's supply voltage, Vcc, and a "false" signal's voltage holds a value nearly at ground. Using only two voltage bands allows circuits to be built at a reduced cost while improving reliability. It is easier to get transistors to produce two distinct voltages near Vcc and ground then it is to get them to produce voltages in a continuous range, especially in modern integrated circuits that contain billions of transistors.

At the most basic level of abstraction, a modern digital circuit is implemented using MOSFETs that are arranged in such a way as to produce logic gates and flip-flops.

Gates can implement basic Boolean functions such as AND and OR while flip-flops form the memory elements of the circuit (they store 1-bit at a time). Certain sets of gates such as AND-NOT, NAND, and NOR are said to be functionally complete. That is, using only these gates any Boolean expression can be realized. By manipulating the connections between gates, logic circuits with higher-level functionality can be created. These circuits include multiplexers, priority decoders, arithmetic logic units, all the way up to circuits that exhibit the extreme levels of complexity seen in modern processors. Digital systems can be categorized into two divisions that will be explained in the following section.

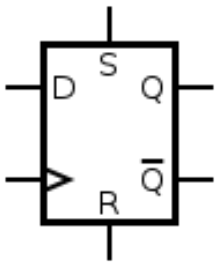
3.2.1.1 Combinational and Sequential Logic

A combinational circuit is a system in which the current output is solely dependent on the present inputs. These circuits are used to perform Boolean algebra. Examples of combinational circuits include arithmetic logic units, multiplexers, and the sig_337p circuit we use for our neural network. This class of logic can be contrasted with sequential systems, where the output of the circuit depends not just on the present inputs but on previous inputs as well. In other words, sequential logic has memory usually implemented using flip-flops that stores the state of the system. Sequential logic can be employed to create finite state machines and registers. An example of a finite state machine is an SDRAM memory controller or the tile control circuit used in the hardware implementation of our convolutional neural network. Virtually all digital systems in use today are composed of at least some sequential logic. Furthermore, sequential logic can be broken down into two subcategories, synchronous and asynchronous logic.

3.2.1.2 Synchronous and Asynchronous Sequential Logic

Synchronous sequential logic uses an oscillator called a clock to synchronize changes in outputs among all the flip-flops that constitute the circuit. These changes typically occur during clock transitions. That is, a flip-flop (Figure 3) reads and stores the value at its input, D, only at the moment in time that the clock transitions from low to high.

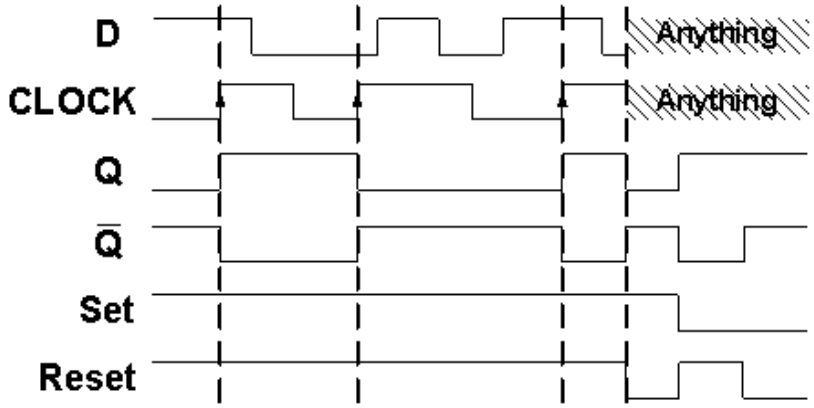
Figure 3: Positive Edged D Flip-Flop - Permission for Reprinting Obtained from Wikipedia (Image Available in the Public Domain)



After some short time, the values Q and Q‘ will update to accurately reflect the input value at the clock transition. The port with the arrow on the flip-flop denotes

where the clock signal is connected to. Furthermore, an arrow pointing into the flip-flop denotes that it reads the input on positive clock transitions (low to high). An arrow pointing the opposite direction would imply sampling occurs during negative clock transitions. The S and R ports exist to allow setting and resetting the flip-flop independent of the other inputs. Ideal D flip-flop functionality is shown in Figure 4.

Figure 4: Positive Edged D Flip-Flop Timing Characteristics - Permission for Reprinting Obtained from the Macao Museum of Communications



Asynchronous logic does not require the use of a clock. Outputs change directly in response to changes in inputs. Flip-flops are not used in this case and memory is implemented using level sensitive latches. The advantage of using asynchronous logic is that it pushes a design to be as fast as physically possible (execution speed is limited by the propagation delays of logic gates, not the frequency of a clock). Even with this major advantage, the overwhelming majority of digital circuits in use today employ synchronous logic to reduce complexity and timing variability. FPGAs are capable of implementing both synchronous and asynchronous sequential logic, but are designed with synchronous logic usage in mind. They have an abundance of flip-flops, clock trees, and PLLs that make synchronous logic usage more favorable.

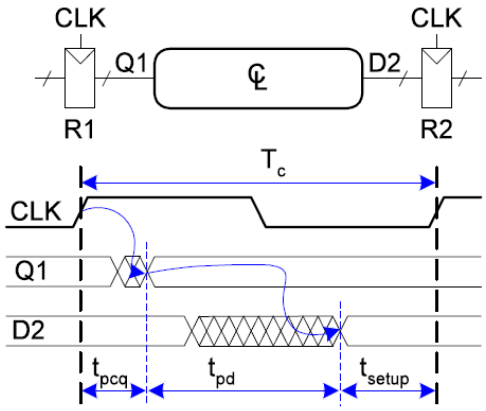
3.2.1.3 Digital Circuit Timing

Ideal flip-flops sample their inputs at the exact moment in time a clock transition occurs. In practice, however, the input signal must remain constant for a certain amount of time before the clock edge, called the setup time, and a certain amount of time after the clock edge, called the hold time. If setup time or hold time is violated, the flip-flop could be forced into a metastable state, where its output is unpredictable for an indeterminate amount of time.

To avoid setup time violations, signals must be given ample time to travel from register to register. Maximum propagation delay between flip-flops is dependent on the amount of combinational logic present. If it takes too long for a signal to travel through a combinational circuit, the input to the following flip-flop will not be stable

for a long enough time to avoid meta-stability. Setup time violations can be corrected by reducing the frequency of the circuit's clock or performing combinational calculations over several clock cycles. Setup time constraints are depicted in Figure 5, where R1 and R2 are registers (made up of flip-flops) and CL is short for combinational logic.

Figure 5: Setup Time Waveform Diagram - Permission from Elsevier for Reprinting Pending



The following equations 1 and 2 must be satisfied to avoid setup time violations:

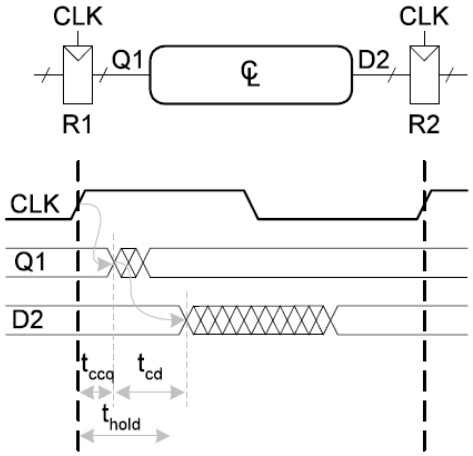
$$T_c \geq t_{pcq} + t_{pd} + t_{setup} \tag{1}$$

$$t_{pd} \leq T_c - (t_{pcq} + t_{setup}) \tag{2}$$

Where tpcq is the amount of time after the clock edge necessary for the output to become fully-stable, tpd is the maximum propagation delay through the combinational logic (some combinational paths are longer than others), Tc is the period of the system clock, and tsetup is the required flip-flop setup time.

Hold time violations occur when signals propagate too quickly from flip-flop to flip-flop. In this case, the signal does not maintain its value for long enough after the clock edge. Figure 6 below depicts hold time constraints.

Figure 6: Hold Time Waveform Diagram - Permission from Elsevier for Reprinting Pending



In the figure above, thold is the flip-flop minimum hold time, tccq is the time after the clock edge when the output of the flip-flop begins changing, and tcd is the delay of the shortest combinational path. Equations 3 and 4 must be satisfied to avoid hold time violations:

$$t_{hold} < t_{ccq} + t_{cd} \tag{3}$$

$$t_{cd} > t_{hold} - t_{ccq} \tag{4}$$

Hold time closure cannot be achieved by modifying the clock. The minimum combinational delay must be increased in this case (assuming the physical characteristics of the flip-flops cannot be modified).

Timing closure is a significant factor in FPGA development. If timing is not met, the system can and will exhibit unexpected behavior. Fortunately, tools exist that can help diagnose where setup and hold time violations will occur, and the designer can take appropriate steps to minimize them. Setup time violations are common when developing a circuit for an FPGA and can be dealt with by inserting registers between combinational logic or decreasing the frequency of the system clock. Additionally, place and route tools can be instructed to run as long as they need to find a configuration that meets timing specifications. Hold time violations are a lot less common in FPGA programming as compilation tools can remove them automatically by inserting buffer logic between registers.

In the following sections, several timing characteristics such as Fmax, the maximum clock frequency allowable for a design (dependent on maximum combinational delay), will be used to describe the FPGA-specific constraints placed on our neural network.

3.2.1.4 Hardware Description Language

The place and route process must have a register-transfer level netlist as input in order to work correctly. There are several different ways of generating netlists, including using hardware-description languages (HDLs), high-level synthesis tools, and even schematic diagrams. Schematic diagrams were phased out years ago by Xilinx, leaving us with a choice between using an HDL or high-level synthesis tool.

These behavioral or algorithmic synthesis tools can take a description of the functionality of the circuit written in a high-level programming language like C (albeit slightly restricted) and generate netlists automatically, saving a lot of time and easing the development process. The downside of using this software is that it generally produces an inefficient circuit relative to one handcrafted in a hardware-description language. This is because there may be overhead in the data-path or control logic, increasing the circuit's latency and logic utilization. In certain cases, they can generate logic that is equivalent to, or even faster than an RTL netlist described using an HDL. We could not be sure before starting the project that this would be the case for our design, thus to ensure our success and take advantage of our familiarity with HDLs, we opted to make a choice between Verilog and VHDL.

While others exist, the most heavily used hardware-description languages are Verilog and VHDL. VHDL, being a product of a US Department of Defense research program in the 1970s, is heavily used in the defense industry and government sector. It is extremely verbose and strongly typed, thus for the reasons of familiarity and accessibility we chose to implement our design in Verilog. Verilog has C-like syntax, simplified vector-bit operations, and is much easier to utilize for rapid prototyping and debugging. SystemVerilog, an extension to the Verilog-2005 standard, would have been ideal as it is the industry standard when it comes to test benches and circuit simulation, but Xilinx ISE does not support its compilation. Thus the choice is largely a personal one, as all three languages can usually implement the same hardware with very little variability in efficiency (post-synthesis at the RTL level).

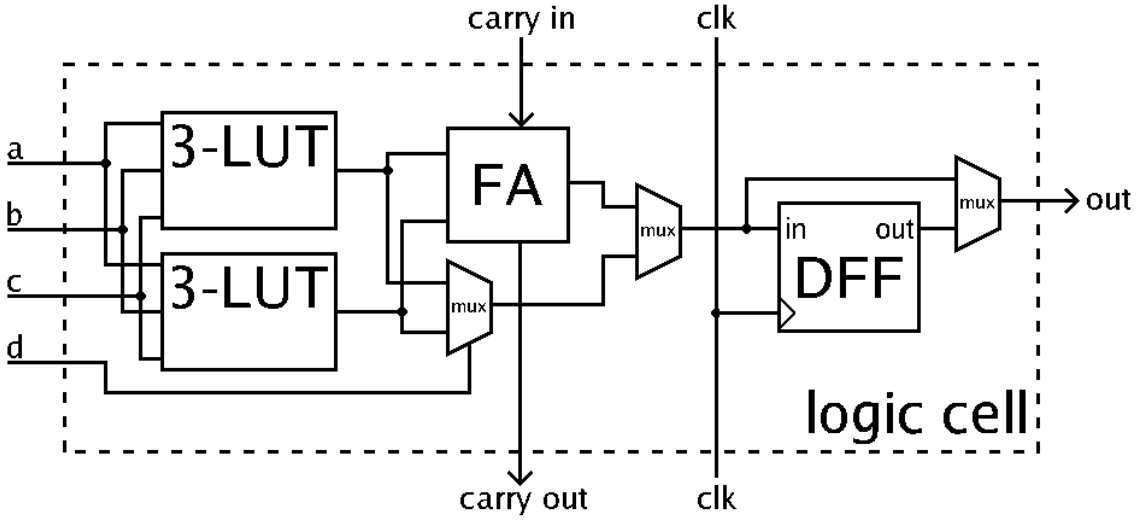
3.2.2 Field-Programmable Gate Array

Field-programmable gate arrays are integrated circuits specially designed to be configurable post-manufacturing. Typically, their configuration is defined by a hardware-description language. In the past, digital designers used circuit diagrams to specify the operation of FPGAs (and ASICs) but as designs became increasingly complex the need for higher-level algorithmic descriptions of digital circuit behavior became more pronounced. Historically, FPGAs were a progression from programmable read-only memory and programmable logic devices. In recent times, as their performance and size have increased, they have moved from being niche products to mainstream computational engines capable of surpassing the performance of CPUs and GPUs at a much higher energy-efficiency.

3.2.2.1 Logic Blocks

The fundamental building block of an FPGA is the programmable logic block. Nevertheless, these blocks can have names that vary from vendor to vendor (Xilinx calls them configurable logic blocks and Altera calls them logic array blocks). They are arranged in a matrix across the silicon wafer, hence the “array” in FPGA. While the internals of these logic blocks can vary depending on the vendor as well as the model of the chips themselves, generally they are composed of smaller logic cells called slices (Xilinx) or adaptive logic modules (Altera). These sub-units usually contain a look-up table, adder, flip-flop, and multiple multiplexers that allow the implementation of basic combinational and sequential digital logic. Below in Figure 7 is a schematic of a typical logic cell. Clearly, the presence of the far-right multiplexer indicates that the cell supports the implementation of both classes of digital systems.

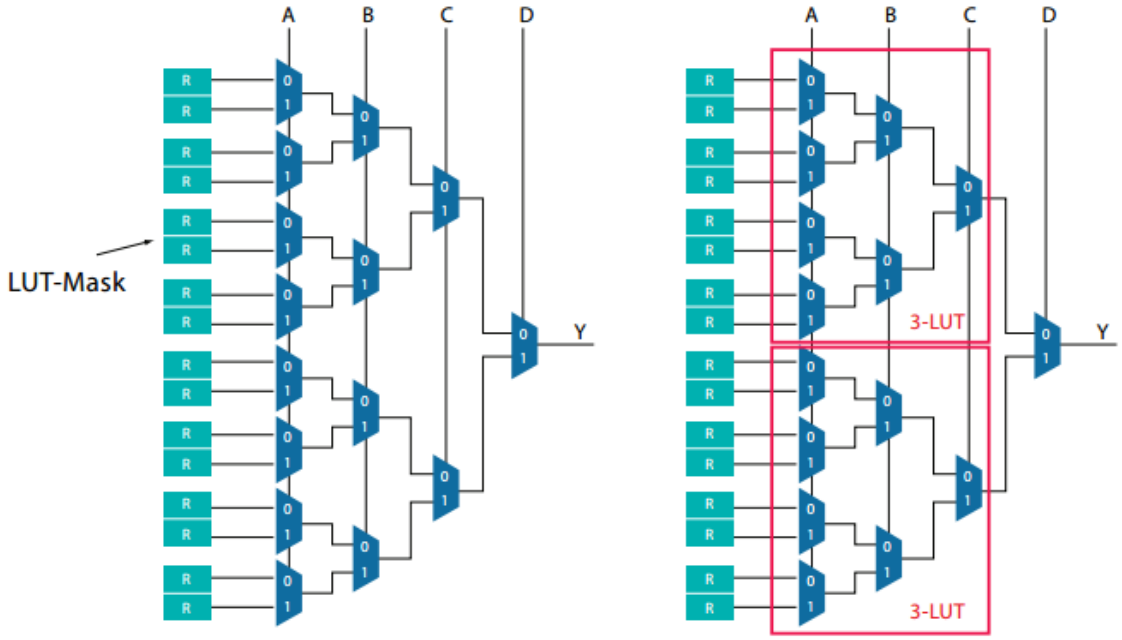
Figure 7: Typical Field-Programmable Gate-Array Logic Cell - Permission for Reprinting Obtained from Wikipedia (Image Available in the Public Domain)



Furthermore, the look-up-tables are configured while the FPGA is being programmed, thus allowing the implementation of arbitrary Boolean functions and removing the need for dedicated logic gates. In some newer devices, FPGA manufacturers have started incorporating LUTs with up to 6-bit inputs, improving performance while reducing overall utilization of the FPGA’s limited resources. Look-up-tables are composed of high-speed SRAM bits that are usually written to using an LUT-mask. This phenomenon is depicted in Figure 8. By writing user-defined data to the “Rs,” the LUT is configured and, in this case, becomes capable of implementing one function with up to 4 input bits or two separate 3-bit functions. It can be inferred then that to implement an n-bit function, the logic cell must contain 24 SRAM bits and a 2ˆ4:1 multiplexer. In practice, larger Boolean functions are computed by combining the LUTs of several neighboring logic cells. It should be noted that these LUTs allow

for the creation of latches (level-sensitive memory elements). The usage of latches is usually not recommended practice when dealing with FPGAs as it is a poor use of resources and can lead to undesirable side-effects such as mismatch in circuit behavior between simulation and run-time.

Figure 8: SRAM-based Look-Up Table Internals- Permission for Reprinting Obtained from Altera



While it would be possible to implement a full-adder using the aforementioned LUTs, it is faster to compute the addition (and subtraction) of binary numbers with dedicated transistors, improving timing and generally increasing performance while freeing the look-up tables to implement arbitrary functions. That being said, some manufacturers choose to forego including full-adders at all in their logic blocks, opting to instead implement arithmetic operations using LUT masks. In FPGAs with adders, connecting the carry-in and carry-out bits of the full-adders present in several logic cells allows an n-bit adder to be realized.

One of the most important components of a logic cell is the flip-flop, which forms the basic memory element of a digital circuit and allows for the creation of sequential logic (finite state machines, pipelines, etc.). Similar to the full-adder, placing flip-flops in parallel can generate registers with arbitrary widths. If combinational logic is desired, the FPGA programming software will configure the final multiplexer so that the output is dependent only on some combination of the full-adder output and the output(s) of the look-up table(s).

Logic blocks use their constituent logic cells to form carry chains, arithmetic chains, and register chains through the application of local interconnect routing. This local

interconnect facilitates the transfer of signals between logic cells in the same block, allowing digital designers to create complicated logic from basic digital entities.

As a side note, modern FPGAs have much more advanced control logic than that shown in Figure 7. For example, Altera's LABs are capable of routing up to 3 clocks, 3 clock enables, 2 asynchronous clears, 1 synchronous clear, and 1 synchronous load signal to their logic cells. Furthermore, the software that performs place and route is smart enough to place associated signals in the same logic block, improving the circuit's timing.

3.2.2.2 Hard Blocks

In addition to logic blocks, modern FPGAs have certain capabilities given to them during manufacturing that extend their functionality and improve the performance of common digital functions. These are termed hard blocks, as they are not programmable and lie separate on the silicon die from the array of logic blocks that are indeed, reconfigurable. Hard blocks can take on many forms. For example, an FPGA designer might include an embedded ARM processor (as is the case with Xilinx's Zynq series), DSP blocks, embedded RAM/ROM, and dedicated multipliers in the FPGA. It is possible to implement the functionality of all these hard blocks using the FPGA's programmable fabric, but the implementations may suffer performance-wise as there is an extra "layer" of LUTs that must be programmed to mimic their behavior. By realizing these commonly used functions with purpose-built transistors, ASIC-level performance is achieved and the logic block array can be used entirely for custom computing.

3.2.2.3 Routing

Routing is the process of configuring the FPGA's interconnect fabric so that the appropriate connections are made between logic blocks, hard blocks, and the FPGA's general-purpose I/O (GPIO) pins. This is essential, as realization of complex circuit behavior is not possible without heavy logic utilization.

The most common routing topology in use today is the switch-box topology, where horizontal and vertical signal channels span the length of the FPGA between the programmable logic blocks (Figure 9). At the corner of each logic block is a switch-box that routes the wires. Programmable transistors at these switch-boxes allow signals to pass to any of the other wires available at the interconnect (Figure 10). While not visible in Figure 9, transistors are present throughout the interconnect fabric that connect the wire segments to the various logic/hard blocks. Furthermore, special routing is usually employed to allow direct connections between logic blocks and to facilitate propagation of low-skew clock signals throughout the chip.

Figure 9: FPGA Routing Architecture

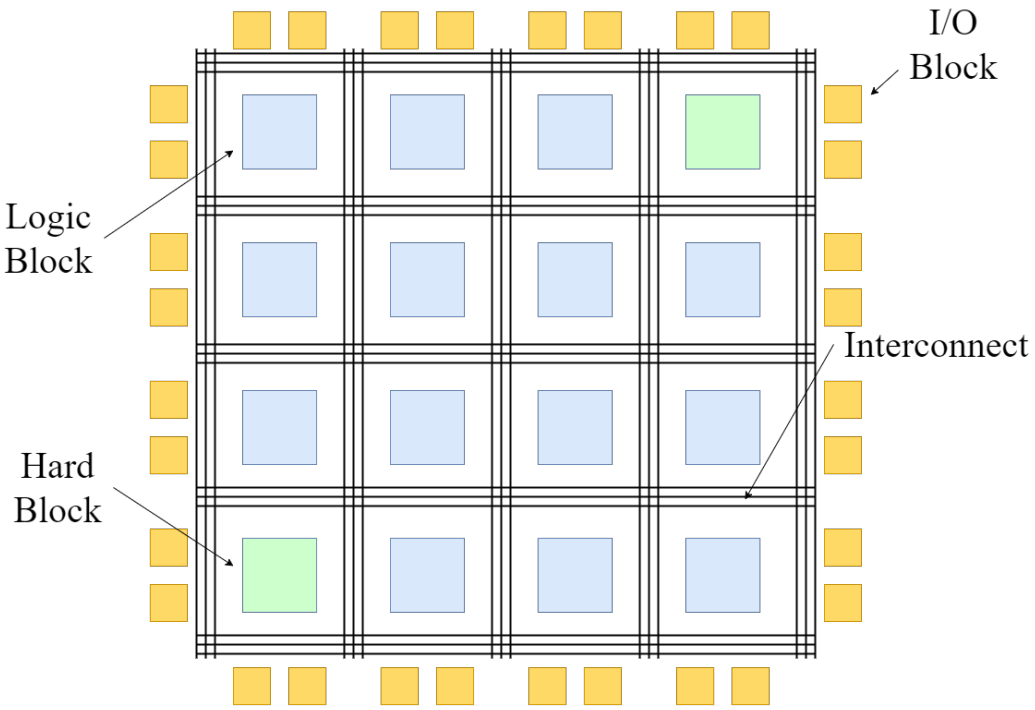
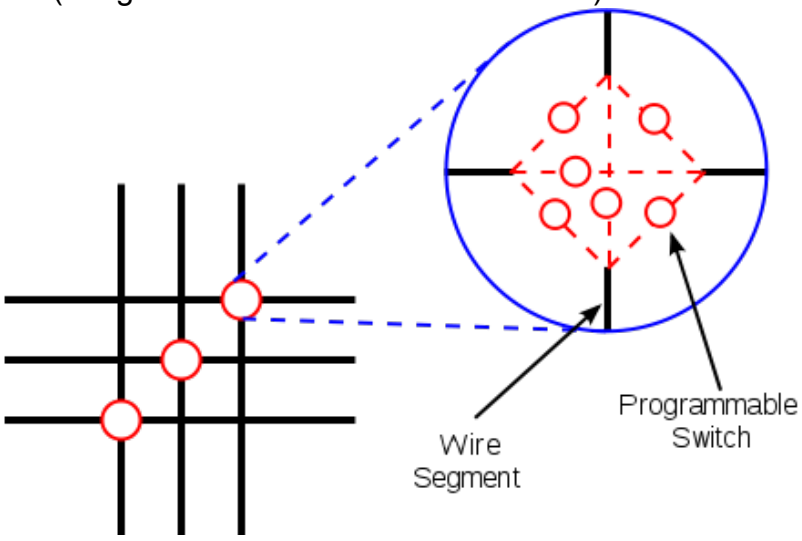


Figure 10: Routing Interconnect Switchbox - Permission for Reprinting Obtained from Wikipedia (Image Available under GFDL License)



Routing is generally considered a very difficult problem. Hundreds of thousands of logic blocks may need to be connected while meeting the designer's timing specifications. Additionally, hard blocks and GPIO pins could be tasked with driving and accepting specific signals, implying that certain logic must be constrained to specific areas of the chip to achieve timing closure. Speaking from personal experience, for very complex designs that employ around 30000 ALMs, the place and

route process can take up to an hour to complete. In some cases, the process may fail as the router either cannot fit the intended design to the FPGA or timing specifications are impossible to meet. In these cases, it is usually possible to instruct the router to optimize for speed or efficiency, affecting the timing and logic utilization of the final circuit. Our software routing options will be determined in the final stages of the design process, when the hardware-accelerated algorithm has been finalized and we can decide on whether we need a decrease in latency or or more efficient utilization of logic resources (though the two go hand-in-hand for our system).

3.2.2.4 Phase-Locked Loop

A phase-locked loop is a control system used to generate precise clock signals in digital circuits, among other applications. This is done by creating a closed-loop frequency-control system whose output is dependent on the difference in phase between an input and reference clock. By adding a divide counter in the circuit's feedback path, a PLL can be used as a frequency synthesis tool. That is, given a fixed oscillator input, a frequency synthesizer can generate a clock signal with a frequency that is an arbitrary multiple of the input clock's frequency. Furthermore, placing multiply counters in the PLL's reference path allows for division of the input clock and a more fine-tuned control of the final output clock.

Due to the re-programmable nature of the FPGA, the timing characteristics of the digital circuits it will be configured to implement are unknown during manufacturing. Therefore, a one size fits all system clock is not very feasible. To remediate this issue, FPGA vendors typically include one or more PLLs as a hard block on the silicon die, allowing the end-user to clock his design at a frequency best suited to its timing characteristics. Since these hard blocks are embedded in the silicon die, they usually require little to no additional "soft" logic resources to implement. Phase-locked loops are either driven by an included internal oscillator (usually with a frequency of 50-100 MHz) or an external oscillator that is fed into the integrated circuit through a dedicated clocking pin.

We will certainly be using a phase-locked loop to clock both our prototype and our final system. It should allow us to maximize the speed of the digital circuit (set the clock frequency at the highest it can go before experiencing timing violations). Certain factors must be taken into account when using a PLL. For example, our system must wait for the PLL to become "locked" before it begins any serious computation. A PLL lock implies the control-system has reached a steady-state and the clock will not experience any meaningful variation in frequency or duty cycle. Failure to wait for PLL lock can induce serious glitches as clock skew, clock jitter deviate, and obviously, clock frequency, deviate from the values used for a pre-configuration timing analysis. In addition to waiting for PLL lock, procedures must be put in place should the control-system lose its lock, or experience a loss of steady-state. We deal with loss of lock by resetting the entire system until the PLL sets its "locked" signal high. Additionally, the system will alert the PC-side software to stop sending data over the serial bus. Typically, loss of lock is caused by poor PCB planning,

thus we hope to avoid this issue all together.

3.2.2.5 Xilinx Spartan-6 Family

The Spartan-6 line of FPGAs from Xilinx is a family of chips focused on low-power, low-cost, and low-performance applications. Indeed, they are the lowest end chips available in the Xilinx product line. That being said, they contain several advanced features available in higher end chips that are of great use to us. These features include dedicated memory controller blocks, a clock management tile, and a moderate quantity of block RAM. Additionally, they can be programmed using a free version of Xilinx ISE, sparing us the trouble of getting a license for a paid version that can cost upwards of 1000 dollars.

A memory controller block becomes important to us when we decide to move our digital design over to our custom PCB. It relieves us of writing our own memory controller block that will undoubtedly be slower and less efficient than one designed by seasoned professionals and embedded in the silicon of the FPGA. Furthermore, it is capable of supporting DDR, DDR2, and DDR3 memory standards with data rates of up to 800 Mb/s, meaning our neural network will not be throttled by the time it takes to access an external RAM chip.

Clock management tiles (CMTs) control the frequency of the FPGA's internal oscillator while eliminating clock skew through the use of dedicated global clock networks. This means we can maximize the performance of our network by increasing the phase-locked-loop (PLL) output frequency to the utmost limit (the limit being the frequency at which flip-flop setup times are violated).

Finally, the Spartan-6 family offers chips with varying amounts of block RAM. The lowest end chip in the line has 216 Kb of block RAM while the highest end contains 4824 Kb. The amount is not that important to us considering we will be using an external RAM chip to hold our data while it is not being used, but we would preferably like to have enough to implement pipeline stage buffers and an FPGA-side weight cache, thus improving the throughput of our network.

Arguably the most important factor in choosing the Spartan-6 FPGA family is its low-cost and the availability of quad flat pack packaging for certain chips. Since no one in the group has experience designing PCBs that route BGA chips, we opted to go for a packaging type that requires less PCB layers and is less prone to error. Furthermore, using a quad flat pack package reduces the manufacturing costs of our PCB, allowing us the freedom to go through multiple iterations of our design if it fails to work properly.

3.2.2.5.1 Spartan-6 XC6SLX9-3TQG144

The specific chip from the Spartan-6 family we chose to utilize for our design was the XC6SLX9-3TQG144. Being the second-lowest tier chip in the family, the XC6SLX9 was the largest chip (in terms of available slices, logic cells, and block RAM) that came in a quad flat pack package. The TQG144 segment in the part number refers to the quad flat pack packaging and the -3 to the speed grade of the chip. We

chose the fastest speed grade of -3 because it costs only a few dollars more than the next speed grade of -2. This chip has 2 CMTs and 576 kB of block RAM but, unfortunately, due to the quad flat pack packaging does not support memory controller blocks. This means we have to implement our own, the tradeoff being we do not have to deal with ball grid arrays on our PCB.

3.2.3 Communications and Ports

In our project we will be working with multiple devices and will need to send information from one to another. In order to do this, we will need be able to communicate with different devices. There are many ways of establishing communication between unique devices and this section will cover our research of different protocols and utilities for doing so.

3.2.3.1 JTAG

The Joint Test Action Group (JTAG) was established in the 1980s as a response to the need for a better method of handling design and of printed circuit boards. This association developed standards which the Institute of Electrical and Electronics Engineers adopted. Although JTAG was designed for testing, it is primarily used as a means of debugging embedded hardware and important as a communications model.

As PCBs increase in the complexity of its components, the likelihood of a board going bad is highly probable. Even one integrated circuit can have thousands of connections which makes it near impossible to test by hand. To verify the integrity of the board, JTAG conducts an IC boundary test. This consists of taking control of the IC pins and by treating some pins as inputs and others outputs, JTAG can send and verify the reception of data.

JTAG also features firmware storage. Programmers of JTAG can more efficiently upload firmware to flash memory. Software and other information can be written often using data bus accesses, but using JTAG interfaces on memory chips makes the process faster and possibly more affordable.

A JTAG interface consists of two, four, or five pins. The latter two styles can be daisy-chained under special conditions. However, the four main logic signals are Test Data In (TDI), Test Data Out (TDO), Test Mode Select (TMS), and Test Clock (TCK). The final, Test Rest (TRST) is an optional connector pin. In the reduced version, JTAG utilizes only a clock signal and a data signal known as Test Serial Data (TMSC). While the former style can be daisy chained, the latter, in the reduced version, is connected in a star topology. Figure and Figure illustrate both examples. The advantages of this are that all parts of the system do not need to be powered as in the daisy chained interface.

Figure 11: JTAG 2 pin Interface - Permission for Reprinting from CC 2.5

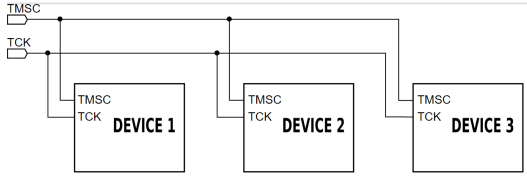
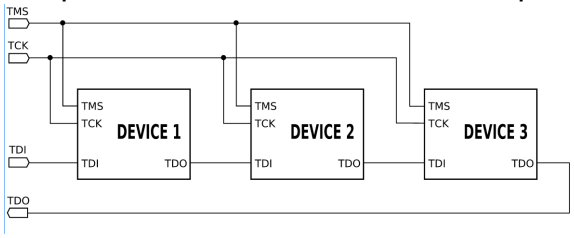


Figure 12: JTAG 4 pin Interface - Permission for Reprinting from CC 3.0



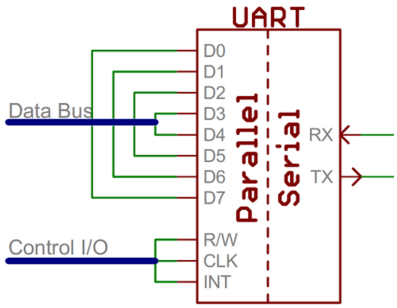
As a communications model, the aforementioned signals make up the test access port (TAP). One or more of these are associated with each device.

Learning about JTAG is important for our project as devices such as FPGAs are normally enabled with JTAG. JTAG gives us frameworks for testing without the need for additional design requirements.

3.2.3.2 UART

The Universal Asynchronous Receiver/Transmitter is important in the implementation of asynchronous serial communication. It consists of an integrated circuit and is often included in microcontrollers. It allows the computer to communicate and exchange information with other serial devices by providing the RS-232C DTE interface. It is responsible for creating the data packet with the addition of the chosen parity bit, sync bit and transmit it through the TX serial line at a set baud rate. It also has a receiving line, RX which is sampled at the chosen baud rate, and output the data. The following image simplifies how the UART is configured between a parallel and serial interface.

Figure 13: Simplified UART interface - Permission for Reprinting from Sparkfun



The UART will be very useful in our project as we will need to establish communication between different devices and send/receive serial data. The advantages are that they only use two wires and does not need a clock signal. It is also widely used and much documentation is available. The UART also features error checking by use of the parity bit. However, there are some disadvantages in that the size of the data frame cannot exceed 9 bits and the expected and actual baud must be within 10 percent of each other.

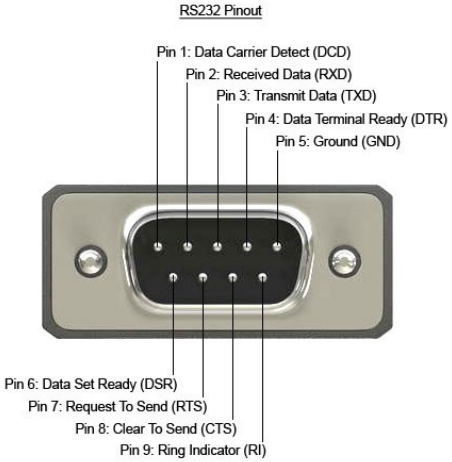
3.2.3.3 RS232

The RS232 is a serial interface standard that is often used to connect the FPGA to a computer. While it has similar applications as a UART, they have clear differences. While UART is primary concerned with the transmission and receiving of bits, RS232 is more concerned with voltage levels. A UART can help implement the RS232 serial interface. However, while USB may be faster, RS232 is more useful in instances where longer cables are needed. It is also more simple.

These devices have two types of classifications: Data Terminal Equipment (DTE) and Data Circuit-Terminating Equipment (DCE) defines the location of signal transmission and reception. DTE typically refers to the standard terminal or serial port that comes with a computer. DCE, which also stands for Data Communications Equipment refers to a device that communicates data, such as a modem. The RS232 standard delineates the DTE classification with male connectors and the DCE classification with female pins.

Implementation of the RS232 requires a transmitter and a receiver. They typically use DB-9, a connector with nine pines though older machines can have twenty-five pins. While there are many pins, the most significant pins are those that are involved with data transmission. These would be pin 2, also known as RxD, which receives the data, pin 3, or TxD which focuses on transmitting data, and finally, pin 5, GND, referring to ground. With three wires connecting to these pins, we can send bit data as a time-series.

Figure 14: RS232 Pinout



This is accomplished through asynchronous communication. Asynchronous communication differs from synchronous communication in that, the clock signal is not associated or transmitted with the data. In this case, the timing of the data bits needs to be communicated to the receiver. The RS232 standard describes the process in the following manner. The communication parameters are first agreed upon before the commencement of communication. These parameters include speed and format between the DTE and DCE. Then, the transmitter signals a certain state. These states include idle, start, and stop. While the line is idle, the transmitter sends a 1. To inform the receiver the start of a data byte, the transmitter sends start, which is 0. After the 8 bits in the data byte are sent, then the transmitter sends stop or 1 to signal the end of the byte.

There are a set of common baud values that are to be used as the speed.

The standard also outlines the specific voltage levels that refer to the data transmission line and the line of signal control. Typically, a valid control signal range is between positive or negative 5 to 15 volts depending on the common ground. Idle would be considered along the lines of -10V. For the lines involved with transmitting and receiving signals, logical one has a negative voltage and logical zero has a positive voltage. These conditions can be referred to as mark and space respectively.

3.2.3.4 I2C

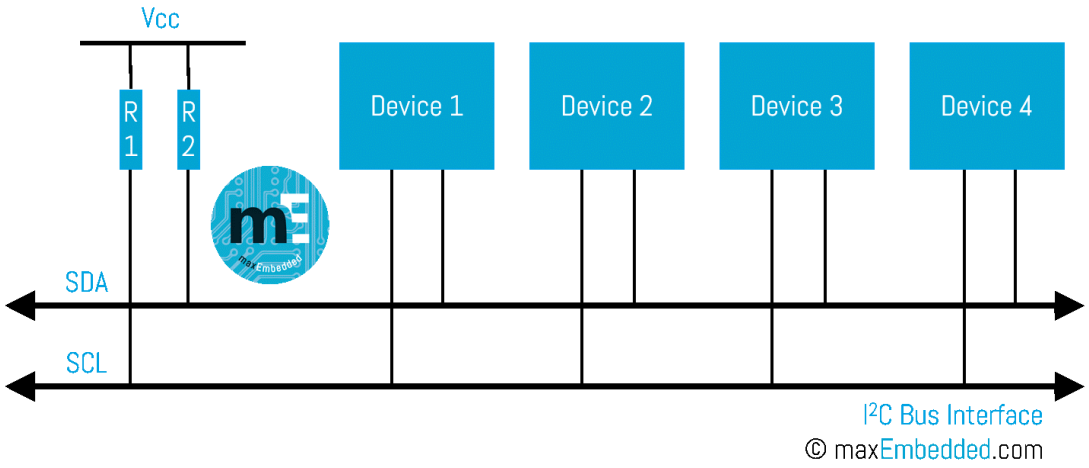
The Inter Integrated Circuit or I2C was invented by Philips Semiconductors as another protocol for serial communication. This interface utilizes two wires to connect lower speed peripheral devices such as input/output devices, integrated circuits to other devices such as processors and microcontrollers.

Most manufacturers of integrated circuits utilize this protocol. One of the advantages of using I2C is that it allows for multiple masters and multiple slaves. The simplification in its design also includes needing to only define upper bus speed. The fact that it also only needs two wires to connect I2C devices with no restriction on quantity is also very useful.

The two lines I2C has are serial clock line (SCL) and serial data line (SDA). These lines are bidirectional and are pulled up to Vdd with resistors. This is due to the fact that both the lines are designed with the open-drain classification. The I2C bus allows for an unlimited number of Master nodes and Slave nodes. The Master device generates the clock and initializes communication with the slave device.

The following figure depicts how devices may be configured on an I2C Bus.

Figure 15: I2C Bus Interface - Permission for Reprinting Granted by maxEmbedded (Available under Creative Commons 3.0



I2C communication is essentially done through the transmission of 8 bit data. Every I2C slave device requires an address provided by the recently Qualcomm-acquired NXP (previously known as Philips Semiconductors). This address can be either seven or ten bit depending on the device.

There are four states in which the I2C can be operated. They include:

1. Master Transmit
2. Master Receive
3. Slave Transmit
4. Slave Receive

I2C also involves a message protocol where each read or write begins and ends with the following conditions: START and STOP.

At the start, the master node initializes the Start condition and provides the address of the slave device it desires to communicate with. This Start condition can be 0 or 1 to signify where it wants to write or read from the slave device.

The slave will respond to the master with an acknowledgment bit to confirm whether or not it is connected to the bus. After that, the master device will continue in either Transmit or Receive mode and the slave device would follow respectively. Once all the data is read or written, then the Stop condition is generated to allow for other devices to use the bus.

3.2.3.5 SPI

Serial peripheral interface (SPI) is a serial communication protocol that is used primarily in embedded systems where distances between components are small. It makes use of a slave-master architecture, where a single master can communicate with multiple slaves using slave-select signals. A basic SPI interface for both masters and slaves has 4 signals. These are SCLK, MOSI, MISO, and SS. SCLK is short for serial clock, which is sent from the master to the slaves to synchronize transmission of data on the bus. Hence, SPI is a synchronous communication protocol. MOSI and MISO are acronyms for master output, slave input and master input, slave output, respectively. These are the data lines used to transmit data on the edges of the clock. SS is short for slave select and designates a signal used to choose from amongst multiple slaves. Each slave has its own slave select signal, thus the master can communicate with each slave individually if needed.

In our project we use SPI to establish communication between the ATmega32U4 microcontroller and the Spartan-6 on the prototype development board we are using. This is necessary for both FPGA configuration (programming) and for transferring neural network inputs, weights, and outputs to and from the board. In this configuration, the microcontroller is the master and the FPGA is the slave. We are able to achieve a data rate of only 46.1 kB/s using this method, thus for our custom PCB we will use a dedicated USB chip interface with that allows us to increase our data rate by a couple orders of magnitude.

3.2.3.6 USB

The Universal Serial Bus is another interface that allows peripheral to computer communication. It is an industry standard for devices including keyboards, disk drives, portable media players and more. Digital information can also be transferred over these ports. This connection is made when a physical port is connected with a USB cable. There are different types of layouts depending on the device.

USB 1.x is the first standard of USB and supports up to 12 Mbps of data transfer as well as 127 devices. USB 2.0 is capable of supporting data transfer rates of up to 480 Mbps. This is known as hi-speed USB. SuperSpeed USB or USB 3.0 improved upon the previous versions and can transfer up to 5.0 Gbps. Most recently USB 3.1, which emerged in 2013 is the most recent version and can transfer data up to 10 GPS. It is known as SuperSpeed+.

3.2.3.7 Communications and Ports Summary

As we are interfacing with different devices, understanding and exploring the various port types and communication protocols is extremely important.

JTAG, RS232, UART, I2C, SPI, and USB will all play roles in our project. The JTAG is particularly useful for debugging the FPGA. The USB is one of the simplest ways of transmitting information to a computer. As our information pathways will need to be bidirectional, understanding how to implement various serial communication protocols will give us added flexibility and options.

3.2.4 SDRAM

In any hardware design, there will be the need for storage and in general terms we refer to Random Access Memory or RAM. Having a sizable chunk of memory that you can temporarily use to store various bits of data and information is important. Another point to consider is being able to access the information in quick manner and at random times. However, it is important to note that this memory is not permanent.

Typically, RAM requires a few basic things in order for data accessibility to be achieved. This includes the memory address, memory bank select, control signals, data input, and data output. The way the memory is accessed largely depends on the type which will be further explored.

Dynamic Random Access Memory or DRAM is implemented using a capacitor. The dynamic characteristic is attributed to the nature of a capacitor. When dealing with capacitors, the charge that is stored on a capacitor would start experiencing a loss unless it a refresh action is taken. This is important because values or data that is not being attended would be lost.

Synchronous Dynamic Random- Access Memory (SDRAM) refers to any type of dynamic random access memory (DRAM) that is synchronized with the timing of the CPU. The advantage of this is that, it removes the need to wait between memory accesses since the clock cycle is known and having it synchronized to its connected device allows for a more predictable inputs and outputs.

Another important distinction to be aware of as we are exploring avenues of storage for hardware is SRAM. SRAM differs from DRAM in that rather than storing information on capacitors, it utilizes a couple of inverters. Due to the fact that we avoid using capacitors, you are not experiencing the same issue of leaking charges and forgotten values. Although the benefit of SRAM over DRAM includes speed and power efficiency, it is quite larger than DRAM. The disadvantage is due to the increased size; it can ramp up costs more quickly as you are not able to fit as many SRAMs on the board as you can with DRAMs. However, they are also very popular in modern processors as they can provide very fast caches. Due to this, we will largely be exploring how to implement SDRAM in our project. The following will describe the technical overview of the features of SDRAM.

SDRAM consists of 6 control signals: Clock Enable (CKE), CS (Chip Select), Data Mask (DQM), Row Address Strobe (RAS), Column Address Strobe (CAS), and Write Enable (WE). Devices that feature SDRAM are also divided into internal data banks. These typically come in 2, 4, or 8 independent divisions. The Bank Selection (Ban) is an input signal that selects the appropriate bank address for a given command. There are many generations of SDRAM with increasing transfer rates and various operating voltages.

When using an SDRAM chip it is helpful to employ a SDRAM controller to better handle interfacing needs. In addition to the 6 control signals used to delineate the different commands for the SDRAM, we will be considering more information for

the user interface. This includes the address to be read or written to, read or write signal, data for inputs and outputs, a “busy” status, and pulse signals to confirm initialization and validation of operations.

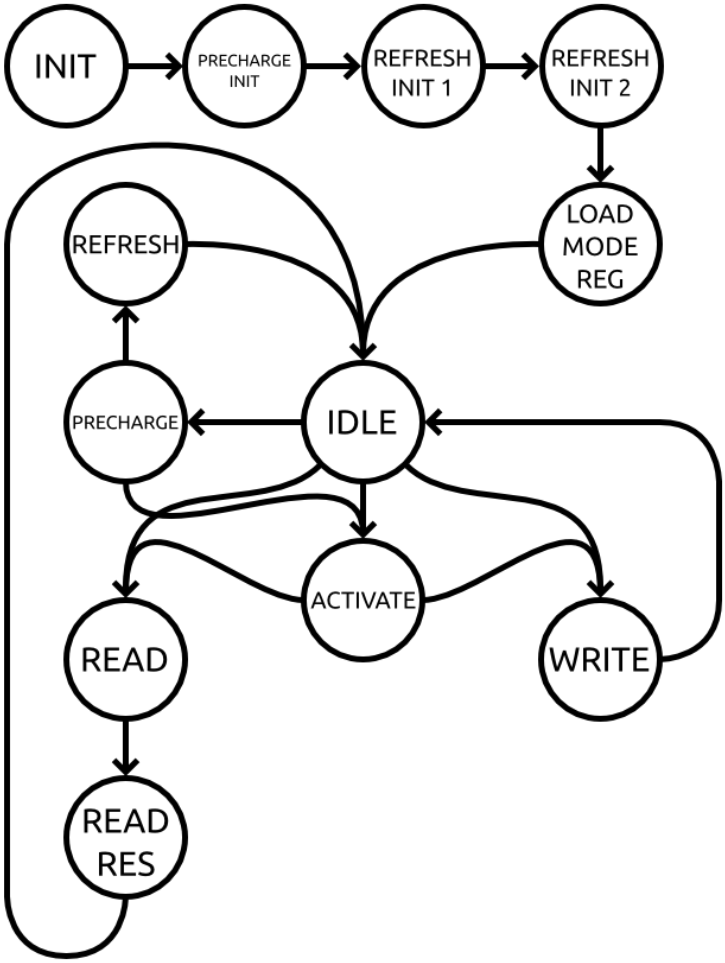
The suggested protocol for the SDRAM controller is as follows:

1. Initialize the “busy” status signal to 0
2. Set variable to your input address
3. Set variable to store data
4. Set input signal for read/written to be 1 for write
5. Set input pulse signal to high to initialize operation
6. Update “busy” status to 1 to inform that the controller is already occupied with a command
7. Update busy status to 1 when the command is taken care of
8. Set address to be read
9. Set read/write signal to 0 for read
10. Set input pulse signal to high to initialize operation
11. Receive confirmation of data read by updating output pulse signal.

The size of the SDRAM bus is 8 bits. The user may take multiple operations in order to achieve sufficient data acquisition or storage.

The following state flow diagram illustrates how the controller utilizes the Finite State Machine Model (FSM) to facilitate its operations.

Figure 16: SDRAM State Flow Diagram - Permission for Reprinting from Embedded Micro



To fully understand the SDRAM chip in conjunction with the FPGA, we also need to consider that a primitive variable of the FPGA is used to get samples of data on both the falling and rising clock edges. This allows for double data rate output. The configuration of the clock of the FPGA results in inverted clock signals. When factoring in the timing and signal propagation in the FPGA, it is advantageous to allow both devices to have time built in to stabilize their outputs.

In addition, there is a block in the programming of the FPGA that will ensure timing of the FPGA is satisfactory. Since the inversion of the clock signals does not provide a sufficient shift alone, the FPGA also has a primitive that provides delays to the clock.

Finally, the Input Output Buffer is a primitive that ensures that the registers for input and output are organized in a way to avoid further timing delays.

3.2.5 PCB

Printed circuit boards or PCBs are the products consisting of various electronic components arranged through the creation of a schematic design. These components can include integrated circuits, transistors, resistors, or capacitors that are typically soldered on to the board.

There are different type of PCBs and they are characterized by how many sides or layers they feature. They can be single-sided, with one copper layer, double-sided with two copper layers, or even multi-layer with both inner and outer layers. As with the latter, these are typically characteristic of more complex PCBs which generally would require computer aided drafting (CAD) software to design. For our purposes we will be learning to utilize Eagle CAD as our primary schematic editor.

3.2.5.1 Modern PCB

PCBs have evolved so much since their creation. From a huge room for a computer to a spinning hard drive to solid state memory, technology has gotten smaller and smaller. Circuits used to involve hand wiring and soldering each part separately. Then the part was screwed on onto a board. When Printed Circuit Boards first emerged they were revolutionary. For the first time the wires between the different components were printed inside of the board. This allowed circuits to be made without wiring mistakes. The board would be made of the designed circuit and then the parts were placed in the proper location on the surface or in their assigned holes. Over the years as PCBs evolve, the demand for them to be smaller has increased which means the boards themselves need to be smaller and that causes the constraint to transfer to the components. The need for the boards to be smaller relates to the ever so popular portable technology of the new century, smart phones, also laptops and numerous other small technology. From this push to make things smaller PCBs have evolved into surface mount and through hole mounting components. Also when accounting for components getting smaller that allows more and more components be placed in one given area. Knowing the complexities of PCB designs and the immense number of parts, layering the traces within the PCB became necessary and efficient

3.2.5.2 Through-Hole Packaging

First through-hole mounting was the standard process for mounting parts onto the bare PCB. Even though at one time though-hole technology was standard it is still used for its increased reliability over the new standard, surface-mount technology. Components that require stronger connections to the PCB, such as heavy or large components, should be mounted with through-hole technology. That way with through-hole technology the component is connected with more solder within the PCB, not just two surfaces bonded together. Another benefit is that the through-hole components leads go into and out the other side of the board which creates a stronger connection, decreased amount of corrosion disruption and allows the component to withstand more environmental stress.

There are two types of through-hole components. The first is axial and the second Is radial lead components. The axial leads have leads on two sides. Each lead exits the component on either end and this allows the component to fit flat and tightly to the board. Radial components have both leads leaving the same side of the component. Radial lead through hold components take up less surface area on the PCB. This is an advantage because it allows more space for placing more components and leading to a more densely component covered PCB.

There are many advantages and disadvantages to through-hole mounted components. This mounting technology provides stronger mechanical bonds and can handle high mechanical stress. The biggest disadvantage is that through-hole mounting requires soldering on both sides of the board.

3.2.5.3 Surface-Mount Packaging

Surface Mount Packaging is the arrangement of mounted components on the surface of printed circuit boards. It has also transformed the way printed circuit boards were designed as it widely used over through-hole packaging. IBM developed this technology in the 1960s and it rapidly grew popular by the 1980s.

Surface Mount Packaging has unique characteristics that allow it to be directly placed on the PCBs. Typically, the components are compact, with small, compact leads, if any. Much of the pins and lines used for connections would be short or flat.

There are any advantages to surface mount packaging, making it an unsurprisingly preferred method of packaging. As components are more compact in order to fit on the surface, the PCBs can hold many more components, and therefore more circuits. Also less space is wasted on the circuit board due to the through-holes. They also make the labor process more efficient and benefit from higher production than through-hole components. While through-hole packaging is one-sided, surface mounted packaging can be done on both sides of the PCB. The connections are also typically better and errors tend to fix themselves as the solder aligns components via surface tension.

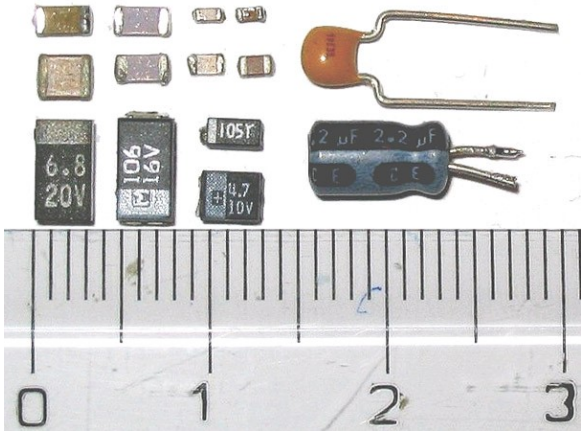
Although popular, there are also some disadvantages. It is not suitable for manual assembly as is directly incompatible with breadboards. Instead they would require a custom PCB. Solder joints are also a weak point, especially as technology moves towards the ultra-fine. Of course, surface mount packaging is not feasible with large parts or anything requiring high power or high voltage. Also the components are often connected to the board through solder joints, and very occasionally, with the additional help of adhesive. If much mechanical stress is expected, then surface mount packaging would not be appropriate.

It is common to see a combination of through-hole and surface mount packaging. For our purposes we will take advantage of this for our prototyping.

The following image depicts one of the advantages of using surface mount components versus through hole components. In this case, it is clear you can fit far more

surface mount capacitors than their through – hole counterparts.

Figure 17: Surface Mount vs. Through-Hole - Image Available in the Public Domain



3.2.5.4 Thermal Considerations

As copper is thermally conductive and is an important part in PCBs, it is no surprise that we must consider temperature in the design of our PCBs. The integrated circuit generates heat that will pass on to the copper layers of the PCB. This in effect causes the whole PCB to warm up which can have direct impacts on the operation of its components. There are various factors that can affect the temperature of operation. These include how thick the copper is, how many layers are on the PCB and how much of the copper is connected. Knowing this, we can also design for temperature optimization.

One of the easiest ways to prevent temperature spikes is to increase the layers or solid ground or other conductive layers that yields a more even distribution of heat. Making sure there is a heat sink, conducting heat and letting it dissipate into the ambient surroundings is important. Also, the thicker the board, or the higher the copper weight, the more thermal heat the board can handle.

Other best practices include configuring your surface mount components in a way that heats the PCB evenly as well. It is highly recommended to include via below components to transfer heat to the copper layers. The general rule of thumb is that for every watt of power you need to dissipate, the board needs an area of 15.3 cm2 for a 40 degrees Celsius increase in temperature.

With more even temperatures, we can use the following formula to approximate the surface temperature. This must be calculated for cooling requirements.

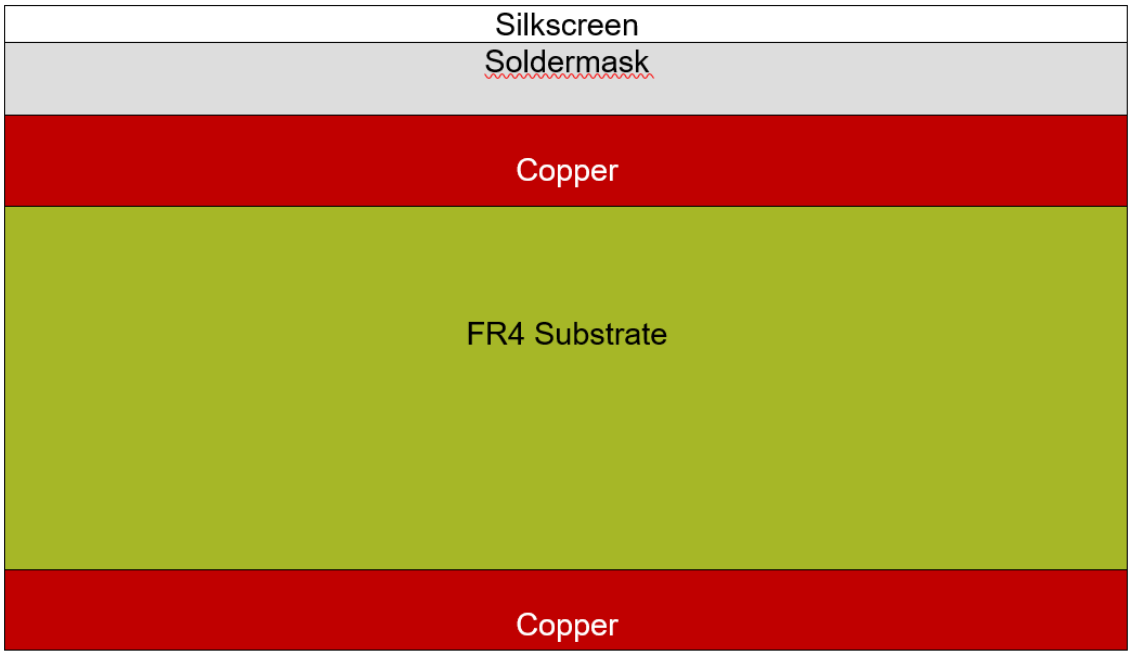
$$P = H_c \cdot A \cdot \Delta T \tag{5}$$

Where P is the power dissipated, H_c is the heat convection constant, A is the area of the board, and ΔT is the surface temperature-ambient temperature.

3.2.5.5 Layering

A PCB is composed of various layers in its base form as illustrated in Figure 18.

Figure 18: PCB Layers



The silkscreen is the lettering, symbols, and numbers visible on the surface of the PCB that helps identify the components and the board. They are most often white and almost never multicolor. Next, the soldermask is what often gives the board its signature green color, although it can be a different color. It is used to insulate the copper layer from making unintentional contact with other conductive material. The copper layer is typically a laminated thin foil and can be found on both sides of a PCB. The number of layers as mentioned earlier refers to the number of copper layers. The most recommended layering is a 4 layer PCB as the cost difference with any less is minimal and works well as a base. 6 layer boards are great for mass quantities of sizeable boards. 12 layer PCBs are best for industrial designs. Finally, the substrate is often made out of fiberglass, FR4. This is the base layer of the PCB and gives it its thickness. There are also options to use flexible materials.

When designing PCBs, it is also recommended to start with the top and bottom layers and add more inner layers if necessary. The outer layers are reserved for signals with the inner layers containing power, ground, buses, and other signals.

3.2.6 Software

3.2.6.1 Speech Data Pre-processing

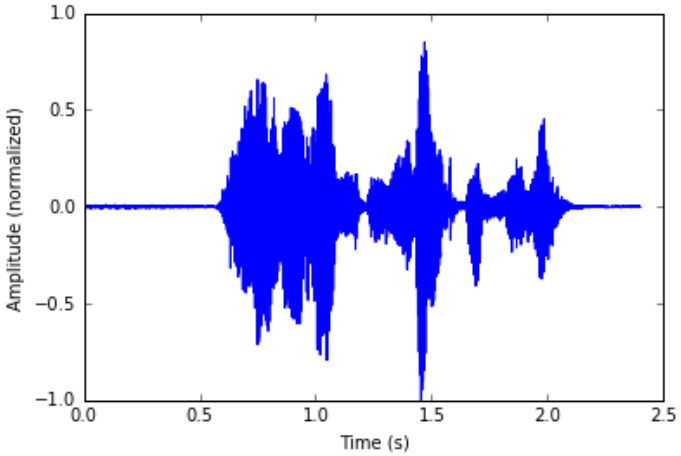
Human language is inherently noisy and vague. Not only is communicating with each other using words like ”hot” and ”cold” difficult, but making sure people can

hear us clearly in the first place is another hurdle. Consequently, we need to refine the data that we pass into our network in order for it to best ”understand” the words being spoken. In other words, we would like to pre-process our speech data to so that our network can receive the cleanest data during training and classification– otherwise, our classification algorithm could misclassify a signal just like someone could mistake one word for another. In speech recognition, it is commonplace to break up utterances into phonemes, units of sounds that distinguish one word from another in a particular language. Due to our small vocabulary, we opt for training on entire words. In the three sections that follow, we discuss several common approaches with regards to pre-processing speech data: using the raw waveform, the power spectrum, or the MFCC features.

3.2.6.1.1 Raw Waveform

As mentioned previously, working with speech signals in the time domain is challenging due to the effects of noise and the intensity of speech. In recent approaches employing CNNs, raw waveforms have been proven to be usable and effective, allowing for real-time classification in speech recognition systems. This approach also reduces the hardware required to pre-process speech signals, saving the costs and computation time associated with DSPs and filters. CNNs are used heavily within the computer vision community, excelling in various tasks, such as object recognition, due to their translation-invariance. We exploit this property in a similar approach in sections 3.2.6.1.2 and 3.2.6.1.3, highlighting a similarity between all of our pre-processing approaches: signal segmentation and linear approximations are vital in efficiently processing the non-linear nature of speech. Figure 19 demonstrates the speech signal that we will analyze using the other pre-processing steps. In particular, we will use simplify.wav for our discussion, a speech recording from the LibriSpeech ASR corpus of the utterance ”how we must simplify,” sampled at 16 kHz.

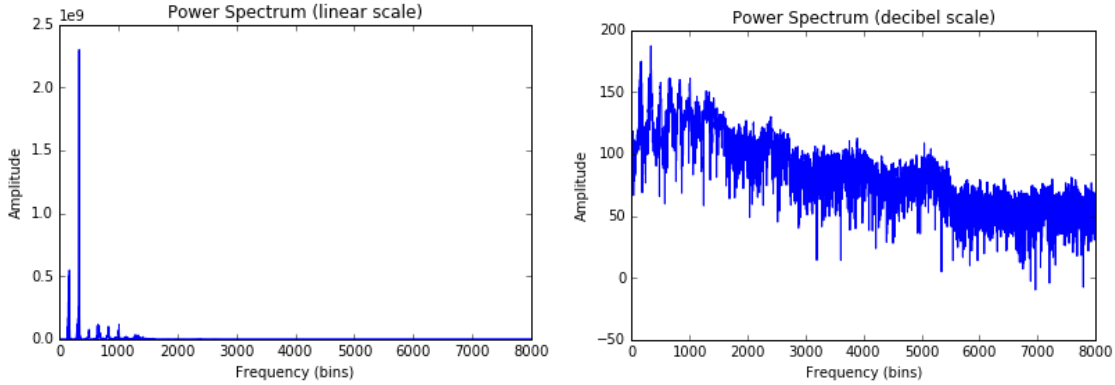
Figure 19: Speech Signal in the Time Domain



3.2.6.1.2 Power Spectrum

Figure 20 demonstrates the power spectral density of the signal in Figure 19. Accordingly, most of the power in the signal lies in the range of human speech: 300-3400 Hz.

Figure 20: Power Spectral Density of the Speech Signal



The power spectrum of a speech signal is practical due to two properties of the Fourier Transform operator: it is stable in the presence of modest translations in the signal and robust in the presence of additive noise. Being able to handle modest translations means that the frequency-domain signal is minimally affected by small shifts in the time-domain signal. For example, if the utterance ”how we must simplify” began at the 1-second mark instead of its starting point in Figure 19, it's frequency-domain representation would be identical. Its only shortcoming is its performance decrease with respect to higher frequencies. Fortunately, in the domain of human speech, this is not a problem for us. A CNN uses the power spectrum of a speech signal in a similar way, attempting to learn patterns in the data that correspond to particular words or phrases.

3.2.6.1.3 MFCC

Mel-frequency cepstral coefficients (MFCC) are widely used in speech recognition. MFCC features are features extracted from audio signals that are derived with the goal of mimicking certain parts of human speech perception. In particular, MFCC features mimic the logarithmic perception of loudness and pitch of human auditory systems through the use of the Mel scale, which relates a perceived frequency with its actual frequency. (6) below shows the conversion from perceived frequency to the Mel scale while (7) goes from the Mel scale to the frequency, though it is worth noting that there is no single Mel scale formula (i.e. the constants can differ).

$$M(f) = 1125\,\ln(1 + \frac{f}{700}) \tag{6}$$

$$M^{-1}(m) = 700(\exp(\frac{m}{1125} - 1) \tag{7}$$

Speech signals are filtered and shaped by the vocal tract, which includes the tongue, teeth, etc. Because the shape of the signal determines what sound comes out, it is of utmost importance to accurately reconstruct it. MFCC features allow us to accurately represent the shape of these sounds so that we can have a better representation of the phonemes being produced. Generally, MFCC features are obtained by the following series of steps:

1. Apply pre-emphasis to compensate for the suppression of high-frequency signals.
2. Partition the signal into short time frames.
3. For each frame, calculate the periodogram estimate of its power spectral density.
4. Apply the Mel-scale filterbank to the power spectra and sum the energy in each filter.
5. Take the logarithm of all filterbank energies.
6. Take the DCT of the log filterbank energies.
7. Keep DCT coefficients 2-13, discard the rest.
8. Find the differential and acceleration coefficients for the DCT coefficients obtained previously.

The dynamic nature of speech signals means that their statistics vary significantly over time. We would like to take advantage of *statistical stationarity*, where the statistics of a signal are constant over time, so that we can relax the computational requirements of processing a speech signal by approximating it. Since small, windowed time frames of sound vary less than longer ones, we want to select frames small enough to approximately meet this criteria of stationarity but large enough such that we can reliably reconstruct the signal. Before creating our frames, we would like to highlight the importance of pre-emphasis. Pre-emphasis is a way of compensating for the rapidly decaying spectrum of speech by applying a high-pass filter to our speech signals. This compensates for high-frequency formants and female speakers, which often produce speech signals at higher frequencies.

Once we have obtained these small, windowed time frames, we want to find the power associated with them by computing the Fourier Transform of each frame. From these results, we then derive the power spectra for the speech signal. Because the human ear cannot discern the difference between two similar frequencies, we would like to cluster the frames that are close together and compute their energies using a Mel filterbank. This effect increases with respect to frequency so filters near the 0 Hz point are narrower than filters at higher frequencies. With these filterbank energies, we then take the logarithm to mimic the human auditory system's logarithmic perception of loudness. Finally, we can take the Discrete Cosine

Transform (DCT) to decorrelate the energies of the overlapping frames, giving us coefficients that we can use as features for classifying speech. We can achieve optimal performance by only taking coefficients 2-13 of the 26 DCT coefficients.

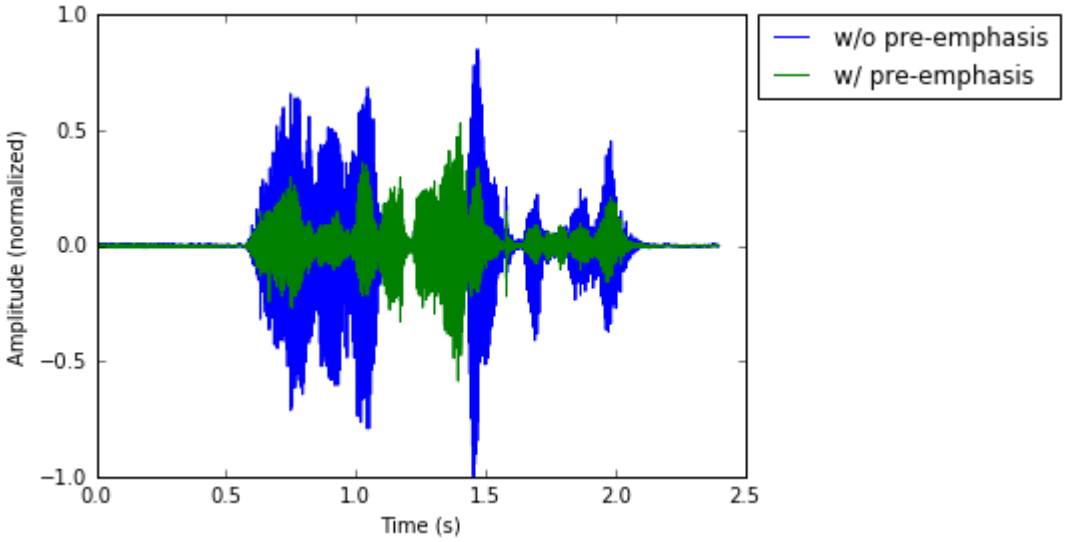
Now we will dive into these steps mathematically to understand how they truly operate. As mentioned in section 3.2.6.1.1, we will use simplify.wav for our discussion, a speech recording of the utterance "how we must simplify," sampled at 16 kHz.

1. To apply pre-emphasis to our speech signal, we will pass it through a high-pass filter with a z-transform as shown in (8), where a typically has values between 0.9 and 1.0.

$$H(z) = 1 - \frac{a}{z} \tag{8}$$

   Figure 21 displays the speech signal before and after pre-emphasis. The signal after pre-emphasis is sharper and has a smaller volume than before pre-emphasis.

Figure 21: Speech Signal With and Without Pre-emphasis



2. Window sizes typically range from 20-30 ms. We will select 25 ms as our window size. If our speech signal is sampled at 16 kHz, then every window will have 16,000 $\times$ 0.025 = 400 samples. Though overlap is optional, we will opt to use 10 ms of overlap, meaning that every frame will start at some increment of 160 samples.

3. Now, we will want to calculate the power in each frame. We do this by multiplying each frame $f_i(n)$ with a Hamming window $h(n)$ before computing the Fast Fourier Transform (FFT), a computationally-improved version of the Discrete Fourier Transform (DFT). The DFT is shown in (9), where i represents the current frame and N represents the total number of samples in the current

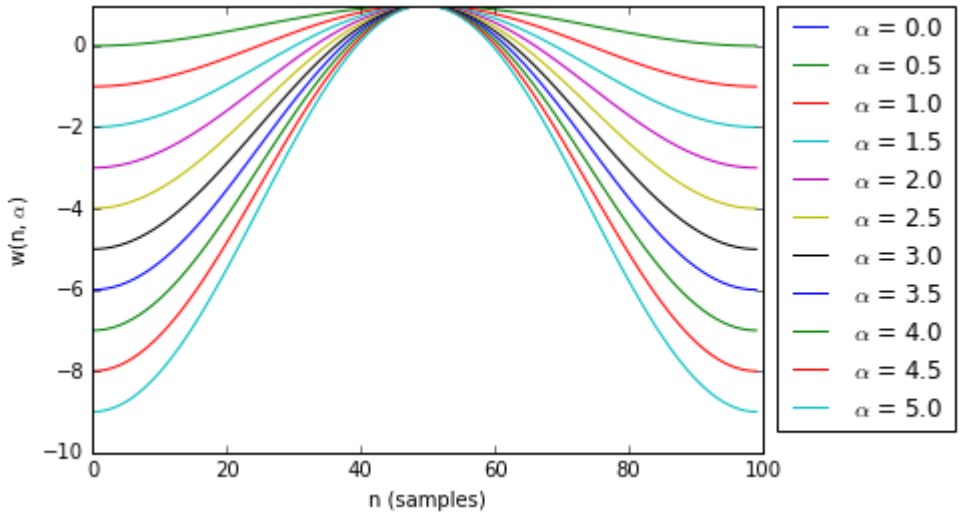
frame (i.e. 400 samples).

$$F_i(k) = \sum_{n=0}^{N-1} f_i(n)h(n)e^{\frac{-j2\pi kn}{N}}, \quad 0 \leq k \leq N-1 \tag{9}$$

The Hamming window is used in order to keep the continuity of the first and last points in the frame, allowing us to see sharper peaks in the frequency domain after taking the FFT. The Hamming window is defined by (10). Curves for varying values of α are shown in Figure 22.

$$w(n, \alpha) = (1 - \alpha) - \alpha * \cos\left(\frac{2\pi n}{N-1}\right), \quad 0 \leq n \leq N-1 \tag{10}$$

Figure 22: Generalized Hamming Window



In practice, the value of α is set to 0.46. Taking the FFT of the signal before applying a Hamming window is the same as applying a rectangular window. At first, applying a Hamming window to the signal might not seem like a valid approach. A rectangular window takes the entire signal into account while the Hamming window only emphasizes the samples close to the center of the signal or frame; however, if we incorporate overlapping frames, we are able to take most of the samples that are on the tail of the curve into account and accomplish our goal of better isolating peaks in the frequency domain.

The advantages of using a Hamming window are best illustrated in the examples in Figure 23 and Figure 24, using the 22nd frame of our speech signal from simplify.wav. On the left is the original signal and on the right is the signal after applying a Hamming window. In the time domain, the effects of the windowing accurately represent the diminishing effect that the tails of the Hamming window have on the signal. While the effect is less pronounced

in the frequency domain representation, the bottom-right plot in Figure 24 is slightly more pronounced than its counterpart.

Figure 23: Effects of Hamming Windowing in the Time Domain

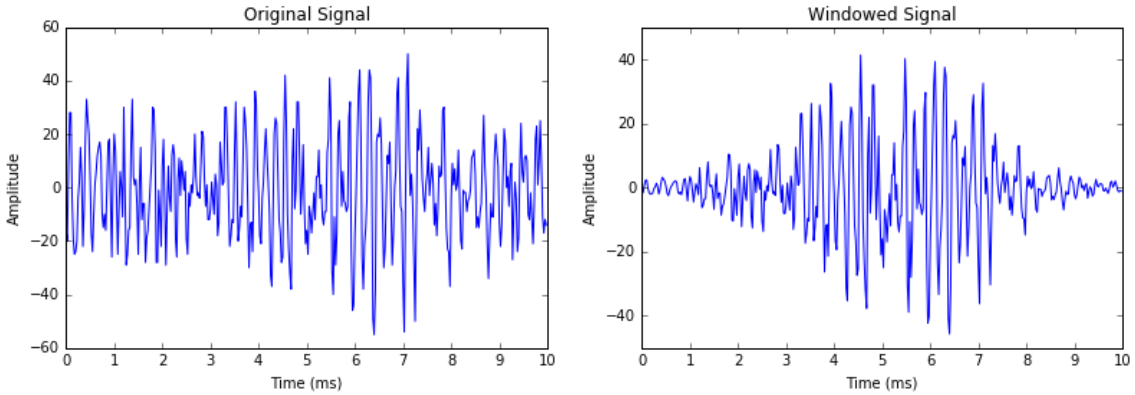
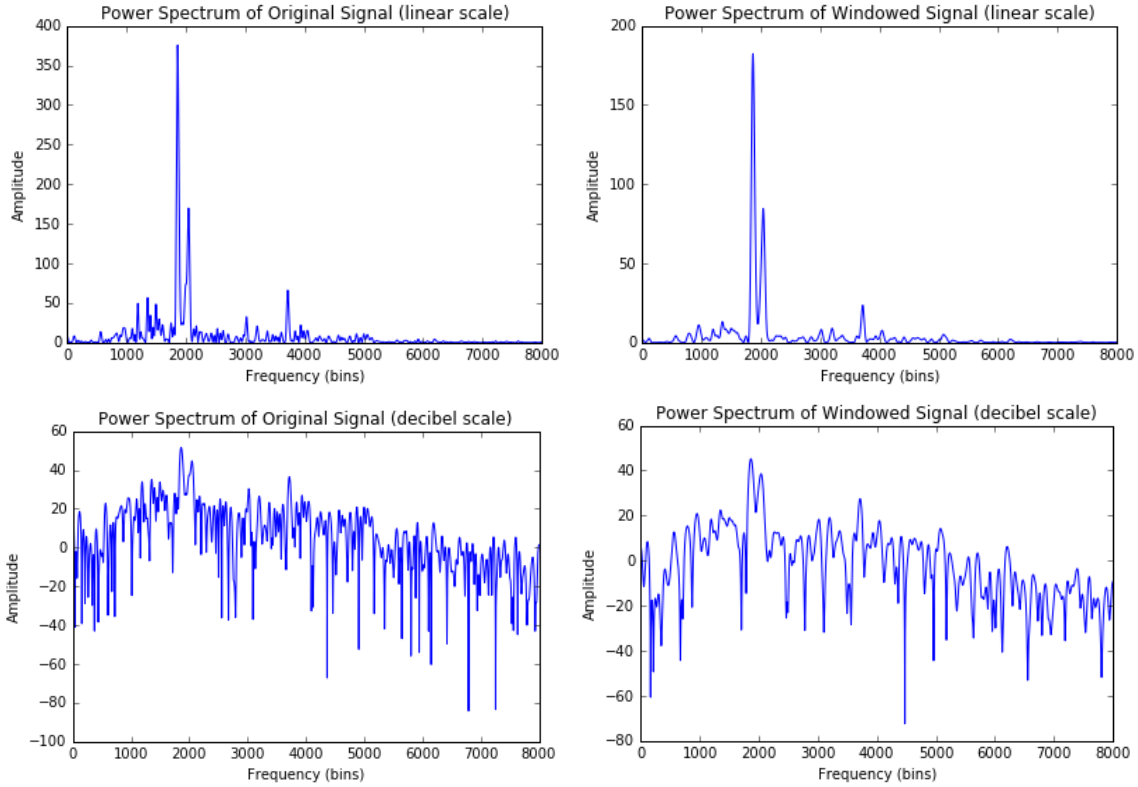


Figure 24: Effects of Hamming Windowing in the Frequency Domain



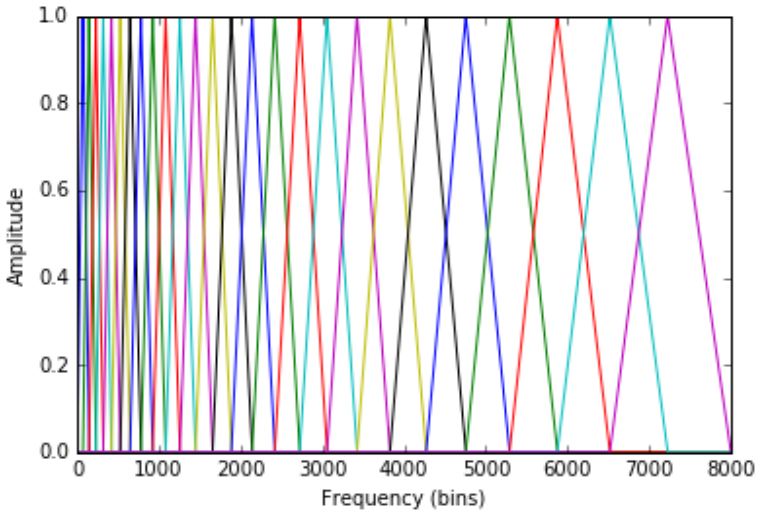
Now that we have applied a Hamming window to all of our frames, we are ready to take the FFT. Since speech and other time domain signals are real-valued, we only need to take half of the coefficients generated from the FFT in order to reconstruct the original signal–the other half correspond to complex values. In the previous example, we took the FFT size to be equal to

the sampling rate, 16,000 kHz. Taking only half of the coefficients generated would give us 8,000 bins, as shown in the plots of Figure 24. Since our frequency resolution per bin is 8000 Hz / 8000 bins = 1 Hz/bin, we can see that most of power in our frame is around 2000 Hz, which is appropriate for human speech (300-3400 Hz). After computing the FFT, we would like to find the power spectral density in each frame using (11), as is again shown by Figure 24.

$$P_i(k) = \frac{1}{N}|F_i(k)|^2 \tag{11}$$

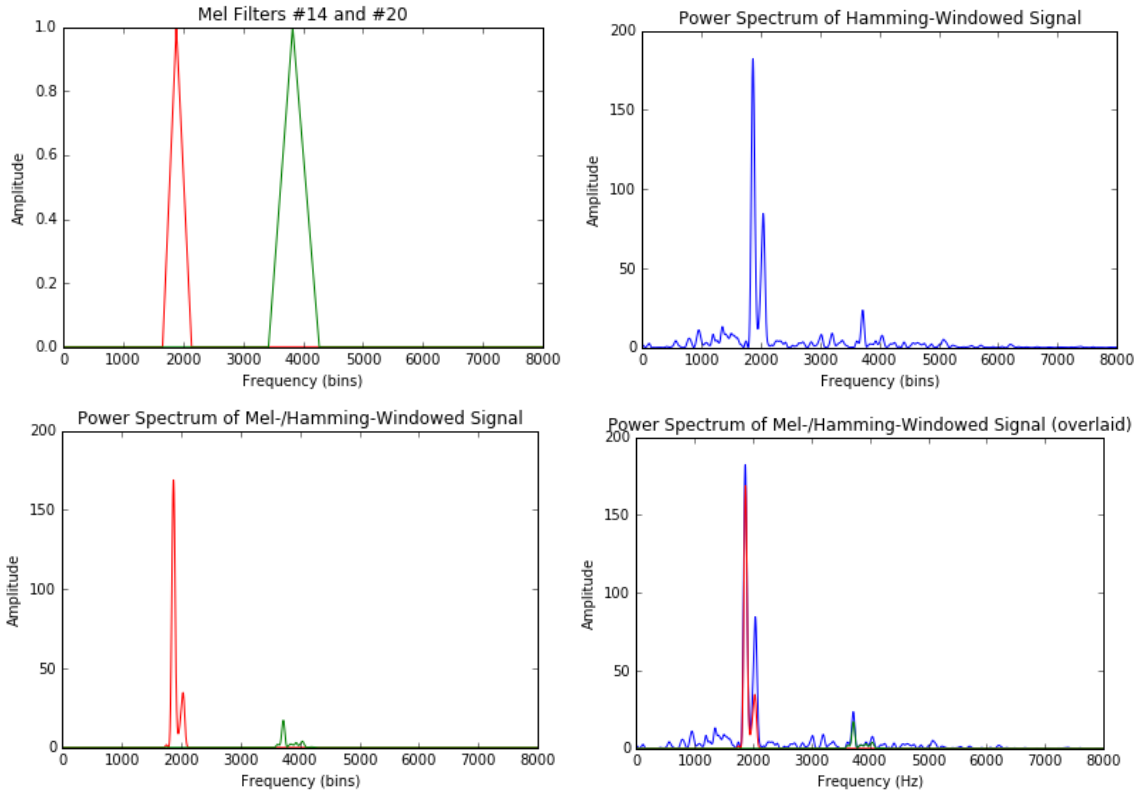
4. We will now return to (6) to cluster the frequency bins and compute their energies. The Mel filterbank is a set of 20-40 triangular filters that are applied to the power spectra of the speech signal obtained in the previous step. It is standard to use 26 filters, so that is what we will use here, as shown in Figure 25.

Figure 25: Mel-scale Filterbank with 26 Filters



The linear scale, Hamming-windowed signal in Figure 24 is used as an example of what these filters do in Figure 26. The filterbank energies are computed by performing matrix multiplication between the Mel filterbank and the signal frames. In the context of an individual frame, summing the values in the bottom-left subplot in Figure 26 produces this result for two of the filters.

Figure 26: Effects of Mel-scale Filters on Signal



5. This step is quite self-explanatory. We take the energies that we calculated in the previous step and simply compute their logarithm.

6. The DCT represents a signal as a sum of cosines of varying magnitudes and frequencies–it is similar to the real part of the DFT. While there are eight theoretically different versions of the DCT, we will be using DCT-II (as shown in (12)), commonly referred to as ”the DCT.”

$$F(k) = \sum_{n=0}^{N-1} f(n) \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)k\right], \quad 0 \leq k \leq N-1 \tag{12}$$

   Passing our 26 log filterbank energies through the DCT will give us our 26 coefficients.

7. Finally, we can extract coefficients 2-13 of the 26 computed coefficients and discard the rest.

8. The coefficients that we have computed only correspond to the static components of the power spectrum of each frame. We would like to compute additional coefficients, the derivative and acceleration coefficients, that correspond to how the MFCCs change over time. For each frame, we can use

(13) to compute these additional coefficients, where c_t corresponds to element t in the MFCC array (of the current frame) and N corresponds to the size of our window for performing a linear approximation to find the derivative coefficients. N is usually set to 2. To find the acceleration coefficients, the same equation and process is used, except the derivative coefficients are used in place of c_t, the MFCCs.

$$d_t = \frac{\sum_{n=1}^{N} n(c_{t+n} - c_{t-n})}{2\sum_{n=1}^{N} n^2} \tag{13}$$

Since c_{t+n} and c_{t-n} will go out-of-bounds at the beginning and end of the speech signal, the first and last element are replicated and extended by the appropriate amount (i.e. if the index is less than zero, set the index to zero).

3.2.6.2 Speech Recognition Algorithims

After covering the pre-processing steps we will implement on our data, we now move on to the algorithmic side of things–how can we use the features we acquired to aid us in training a model and classifying speech? Many popular algorithms have been developed over the years for speech recognition. We will stick to discussing a subset of these at a somewhat high-level, so that we are not bogged down by excessive detail but are able to understand the underlying algorithm. In particular, the algorithms we will discuss are have had the most recent success in acoustic modeling and speech recognition; namely, hidden Markov models, neural networks, and convolutional neural networks.

3.2.6.2.1 Hidden Markov Models

Hidden Markov models (HMM) are generative and statistical models in which we observe a sequence of observations, O = $\{o_1, o_2, \ldots, o_T\}$, whose probability distributions are governed by hidden states. HMMs are commonly used in speech recognition, particularly with MFCC features, because a speech signal can be modeled as a piecewise stationary process. Specifically, speech segments of about 10 ms can be approximated as short-time stationary signals. In order to understand how we can use HMMs as acoustic models to classify speech, we will start by introducing some of the theory behind Markov chains, simpler Markov models that have observable states.

A discrete-time Markov chain is a system with N states, S = $\{s_1, s_2, \ldots, s_N\}$. At each evenly-spaced, discrete-time interval, there is a transition from state s_i to state s_j, where the probability of transitioning to particular state s_j depends on a matrix of state transition probabilities. For a system with N = 3, this matrix is shown below.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

The state at time t is a random variable q_t. The probability that q_{t+1} will assume a particular state would be typically conditioned on the previous states $q_{t-1}, \ldots, q_1$.

The first-order Markov property, however, establishes that a future state, q_{t+1}, is independent of the past states, $q_{t-1}, \ldots, q_1$, given the present state, q_t. In other words, the first-order Markov property allows us to remove additional conditioning on the past by exploiting one of the consequences of independent distributions: $P(A|B) = P(A)$ if A and B are independent.

$$P(q_{t+1} = s_j | q_t = s_i, q_{t-1} = s_h, \ldots, q_1 = s_g) = P(q_{t+1} = s_j | q_t = s_i) \tag{14}$$

Where $1 \leq g, h \leq N$ and s_g and s_h in (14) are arbitrary states. Using this result, the transition probabilities take the form in (15).

$$a_{ij} = P(q_{t+1} = s_j | q_t = s_i), \quad 1 \leq i, j \leq n \tag{15}$$

For concreteness, consider the example shown in Figure 27. This Markov chain is a weather model with N = 3, S = $\{c, s, r\}$, and a state transition matrix as shown below. The states c, s, and r correspond to cloudy, sunny, and rainy, respectively.

$$A = \begin{bmatrix} 0.3 & 0.2 & 0.5 \\ 0.3 & 0.6 & 0.1 \\ 0.1 & 0.4 & 0.5 \end{bmatrix}$$

Notice that by using the Law of Total Probability, we can decompose the probability that q_t = c transitions into any other state as a summation of the horizontal entries in the first row of the state transition matrix.

$$P(q_{t+1} = c | q_t = c) + P(q_{t+1} = s | q_t = c) + P(q_{t+1} = r | q_t = c) = 0.3 + 0.2 + 0.5 = 1$$

This can be generalized for any state s_i by (16).

$$a_i = \sum_{j=1}^{N} a_{ij} = 1, \quad a_{ij} \geq 0 \tag{16}$$

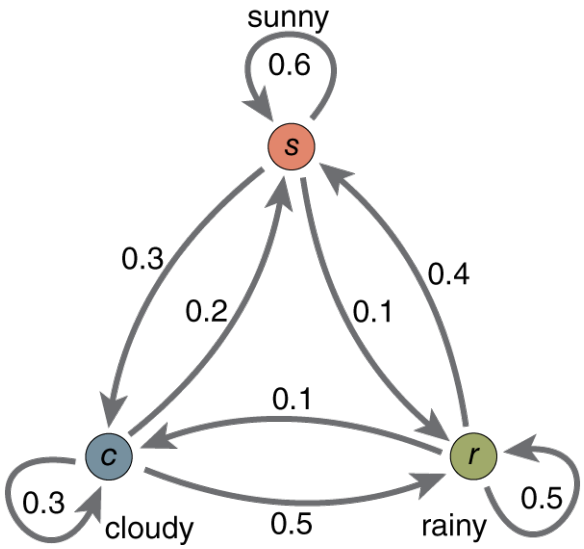
This random process is clearly an observable Markov model–at each time t, the output of the model corresponds to the random variable q_t, which directly maps to a particular state s_i. To expand on Figure 27, suppose that we have an initial state q_1 = c. In general, we define the probability of an initial state as in (17).

$$\pi_i = P(q_1 = s_i), \quad 1 \leq i \leq N \tag{17}$$

What is the probability that the sequence of weather patterns for the next five days is cloudy, cloudy, sunny, rainy, sunny? Stated more formally, what is the probability of the observation sequence O = (c, c, s, r, s) for the times t = $\{1, 2, \cdots, 5\}$ given the model? The first-order Markov property, combined with the state transition matrix, gives us the information we need to compute the solution to this question.

$$\begin{aligned} P(O|\text{Model}) &= P(c, c, s, r, s|\text{Model}) \\ &= P(c)P(c|c)P(s|c)P(r|s)P(s|r) \\ &= \pi_c \cdot a_{cc} \cdot a_{sc} \cdot a_{rs} \cdot a_{sr} \\ &= \pi_1 \cdot a_{11} \cdot a_{21} \cdot a_{32} \cdot a_{23} \\ &= 1 \cdot 0.3 \cdot 0.3 \cdot 0.4 \cdot 0.1 \\ &= 0.0036 \end{aligned}$$

Figure 27: Discrete-Time Markov Chain of the Weather - Permission for Reprinting Pending



We now move forward to HMMs. The first notable difference between Markov chains and HMMs is the sequence of observations: q_t no longer directly maps to states. Instead, it maps to a new set $V = \{v_1, v_2, \ldots, v_m\}$ which defines the set of m symbols that we can sample from when making observations. At every time t, q_t will sample a symbol v_k from V based on the probability distribution imposed by the current hidden state s_i. Specifically, (18) describes the probability distribution of a symbol v_k.

$$b_i(k) = P(v_k|q_t = s_i), \quad 1 \leq i \leq N \tag{18}$$

Sticking with the $N = 3$ example, we can create a vector B that encapsulates this information for all possible states, as shown in (19).

$$B = \begin{bmatrix} b_1(k) \\ b_2(k) \\ b_3(k) \end{bmatrix} \tag{19}$$

We should now take stock of what we can know about HMMs: the number of states, N, the number of symbols, M, the state transition probability matrix, A, and the symbol observation probability vector B. The resulting model is a doubly embedded random process where the first random process describes the transitions from state s_i to state s_j, with N being the total number of states, while the second random process describes the production of the sequence of observations, with M being the total number of symbols that can arise.

At the beginning of the section, we mentioned how the HMM is a generative model. This means that once it is fully specified, it is capable of generating an observation sequence $O = (o_1, o_2, \ldots, o_T)$, where each observation o_t is a symbol v_k and T corresponds to the total number of observations. Alternatively, we have a model for how a given observation sequence was generated by an HMM.

A complete HMM specification requires the specification of N and M, the observation symbols, and the three probability measures A, B, and π_i, the initial state probability. This is usually packaged into the notation in (20).

$$\lambda = (A, B, \pi) \tag{20}$$

Since our dictionary is only 14 words, we will represent each word with its own N-state HMM. Classification of an utterance containing multiple words from our dictionary is a matter of combining the 14 HMMs used for each word in the dictionary. For each HMM, we are tasked with encoding the speech signal of the word or its features into a set of M symbols. With our HMM model, there are three problems that we want to solve for the model to be useful for speech recognition:

1. Given an observation sequence O = $(o_1, o_2, \ldots, o_T)$, and a model λ = (A, B, π), how do we efficiently compute $P(O|\lambda)$, the probability of a sequence of observations given a specific HMM?

2. Given an observation sequence O = $(o_1, o_2, \ldots, o_T)$, and a model λ = (A, B, π), how do we determine the sequence of states Q = $(q_1, q_2, \ldots, q_T)$ that best models the observations?

3. How do we adjust the model parameters λ = (A, B, π) to maximize $P(O|\lambda)$?

Answering these questions mathematically involves a significant background in probability and statistics, as well as knowledge of dynamic programming. These questions are enough to guide our understanding.

Problem 1 corresponds to the evaluation of the model. If we have a set of competing models Λ = $\{\lambda_1, \lambda_2, \ldots, \lambda_L\}$, we would like to calculate $P(O|\lambda_l)$ for every model in Λ to evaluate which model best represents the observation sequence. In our case, L = 14. If we would like to classify the utterance ”eight,” we would have to compute $P(O|\lambda_l)$ for every model in Λ. The model with the highest value is the model that best corresponds with the utterance.

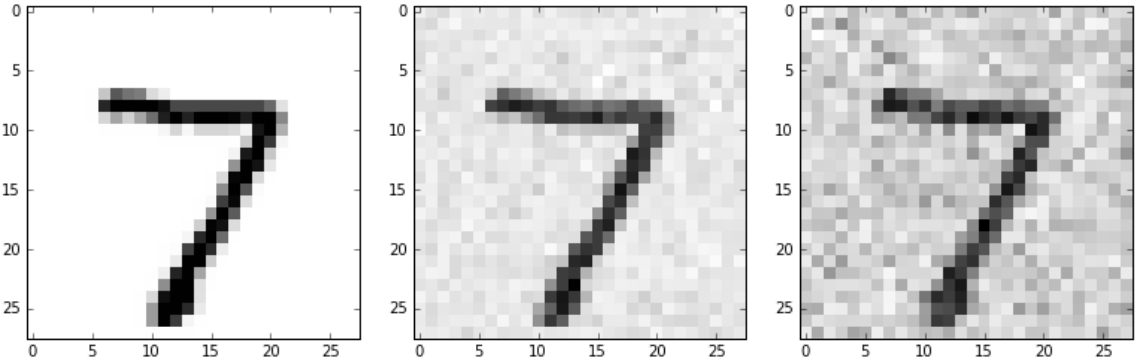
Problem 2 corresponds to attempting to find the ”correct” state sequence for the model λ. Except in the degenerate case, where the probability distributions of the symbol observations are equal to one for every state, there is no single correct state sequence. Optimality criteria are imposed in order to improve performance for speech recognition.

Problem 3 corresponds optimizing the parameters of a model λ to best describe how a given observation sequence comes about. This is where training comes in– an observation sequence, commonly referred to as the training sequence, is used to adjust the model parameters. Since we are using an HMM for each word instead of each phoneme, our training sequence is the symbol-encoded speech signal we mentioned earlier.

3.2.6.2.2 Neural Networks

While machines excel at performing massive computations in a relatively short time, they perform poorly on certain tasks that are easy for humans. In the language domain, this is evidenced by the complexity behind designing speech recognition systems. HMM approaches, as defined in the previous section, commonly used MFCC features to build a generative model of words or phonemes. Instead of building this generative model and then computing its difficult, near-intractable probability distributions, could we possibly learn a classifier instead? Neural network models offer a possible solution to the problem, forming the foundation of the CNN that we employ as our classifier. To begin our discussion, consider the images in Figure 28.

Figure 28: Image from the MNIST Dataset with Increasing Noise



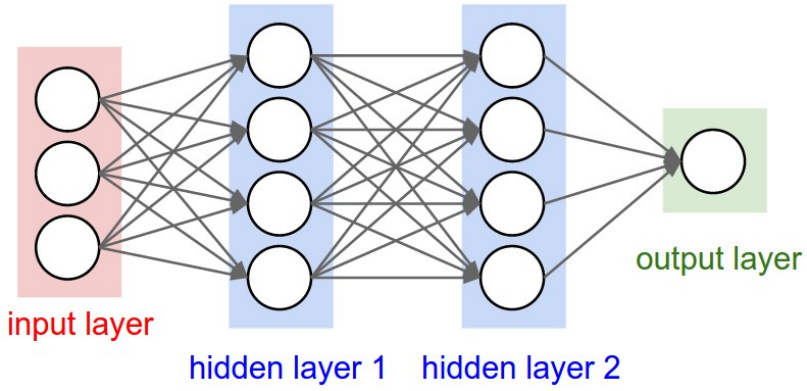
While it is quite obvious to humans that all three of these images are the number seven, machines do a poor job at realizing that these numbers are the same, especially with the third image. Until recently, they have been unable to take advantage of richly structured information, primarily due to the computational complexity of many learning algorithms on limited hardware. Inspired by the biological neural networks in our brain that allow us to make vast inferences, artificial neural networks (ANN) were created in hopes of being able to give machines the ability to process the noisy information that previous systems could not, such as the examples above.

The first computational model of an ANN was developed in 1943 by Warren McCulloch and Walter Pitts. ANNs were meant to be analogues to biological neurons, incorporating many of their features, such as layering patterns, axons and synapses, signal thresholding, and Hebbian learning. The philosophy was that this model, arranged like neurons in the brain, could approximate human learning. While there are now many different implementations of ANNs, they still share most of the aforementioned components with their biological counterparts, though these similarities are dwindling. Recent progress in machine learning has taken these algorithms further away from biological plausibility, such as in the backpropagation algorithm: no evidence has suggested that the backpropagation algorithm is what actually happens in the brain during learning. That is not to say that this trend will continue.

Convolutional neural networks, as we will see in the next section, are a biologically inspired by the animal visual cortex.

Figure 29 demonstrates an example of a 3 $\times$ 4 $\times$ 4 $\times$ 1 ANN. The first layer is known as the input layer, the last layer is known as the output layer, and any layers in the middle are known as hidden layers. Until the 1980s, ANNs were not used due to their inability to solve the exclusive-or circuit and their computational complexity. Since then, ANNs been immensely successful, solving the exclusive-or circuit and other tasks that were once thought impossible for machines. Extending the success of these networks, deep neural networks (DNN) took advantage of recent developments in hardware to extend the power of ANNs. DNNs are the same as ANNs, with one caveat: they have more layers. The amount of layers needed for an ANN to be considered a DNN is unclear but usually more than two hidden layers is a DNN. We will refer to both of them as neural networks (NN) for simplicity.

Figure 29: Example of a Neural Network (Permission Granted by Andrej Karpathy)



Across multiple domains, NNs have outperformed many of the previous techniques. One of the notable examples includes AlphaGo, a machine learning system that beat the world champion several times in Go, a game with an immense combinatorial search space. How can NNs learn from data and perform so well in various tasks despite a noisy environment?

The example images used above were from the MNIST dataset, a popular benchmark for NNs developed by Yann LeCun. The images in the MNIST dataset are 28 $\times$ 28 gray-scale images with pixel values ranging from 0-255. Before being passed through a network, these images are flattened into a vector X of pixel values. The connections between each layer make up the weights of the NN, stored in a matrix Θ. The matrices corresponding to the connections between the first two and next two layers are shown in (21). Keep in mind that the indices i and j of θ_{ij} correspond to a node in layer i connecting to a node in layer j. Consequently, the θ_{11} in Θ_{12} is not equal to the version in Θ_{23}. For a more in-depth discussion on matrix algebra,

see Appendix C.

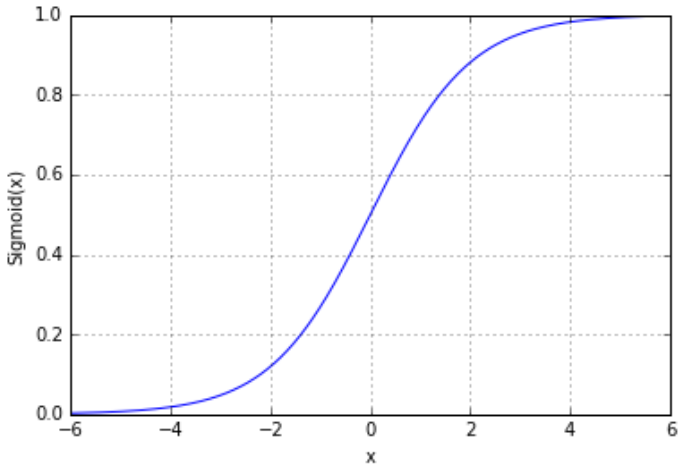
$$\Theta_{12} = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} & \theta_{14} \\ \theta_{21} & \theta_{22} & \theta_{23} & \theta_{24} \\ \theta_{31} & \theta_{32} & \theta_{33} & \theta_{34} \end{bmatrix} \quad \Theta_{23} = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} & \theta_{14} \\ \theta_{21} & \theta_{22} & \theta_{23} & \theta_{24} \\ \theta_{31} & \theta_{32} & \theta_{33} & \theta_{34} \\ \theta_{41} & \theta_{42} & \theta_{43} & \theta_{44} \end{bmatrix} \quad \Theta_{34} = \begin{bmatrix} \theta_{11} \\ \theta_{21} \\ \theta_{31} \\ \theta_{41} \end{bmatrix} \tag{21}$$

NNs take an input X at the first layer, called the input layer, and matrix multiplication is performed with the weights between this layer and the next, as shown in (22)

$$\Theta_{12}^T X = \begin{bmatrix} \theta_{11} & \theta_{21} & \theta_{31} \\ \theta_{12} & \theta_{22} & \theta_{32} \\ \theta_{13} & \theta_{23} & \theta_{33} \\ \theta_{14} & \theta_{24} & \theta_{34} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{22}$$

This output is then fed into a non-linear activation function. While various activation functions can be used, the sigmoid function in Figure 30 is the most common.

Figure 30: Sigmoid Function



This choice of activation function is commonly used in most NNs, with a $x \in (-\infty, \infty)$ and $f(x) \in (0, 1)$. Specifically, the mathematical equation for the sigmoid function is shown in (23).

$$\text{Sigmoid}(\Theta_{12}^T X) = \frac{1}{1 + \exp^{\Theta_{12}^T X}} \tag{23}$$

This process is the same for the next pair of layers, with the output of this step serving as input for the next. Ultimately, we will receive an output vector at the end. What can be interpreted by this output vector? In the case of image classification, we would like our final vector to tell us what number the network believes it ”saw”; likewise, for speech recognition, we would like this vector to tell us what utterance the network believes it ”heard.” This is where the backpropagation algorithm comes into play.

The backpropagation algorithm is an efficient way of training NNs. After the output o of the final layer is received, it is subtracted from a vector of target values t to find the error e_{jk} of the kth node in the jth. This error is used to correct the values of the weights in the NN by propagating it backwards through the network, layer by layer. Using the final layer in Figure 29, (24) shows the error calculation.

$$e_{41} = t - o_{41} \tag{24}$$

The error value corresponds to the error at that output node, which is contributed by all of the nodes that feed into it. Accordingly, this error should be sent backwards in a proportional manner, with inputs and weights that contributed more receiving more of the error. To continue our example, (25) shows that our four errors from o_{41} would be functions of the weights that connect the nodes in hidden layer 3 to the output node in layer 4:

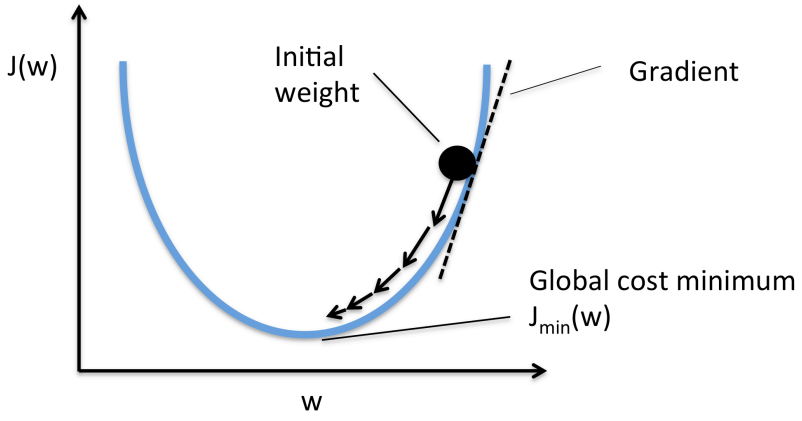
$$\begin{aligned} e_{31}(\theta_{11}) &= \frac{e_{41}\theta_{11}}{\theta_{11}+\theta_{21}+\theta_{31}+\theta_{41}} \quad e_{32}(\theta_{21}) = \frac{e_{41}\theta_{21}}{\theta_{11}+\theta_{21}+\theta_{31}+\theta_{41}} \\ e_{33}(\theta_{31}) &= \frac{e_{41}\theta_{31}}{\theta_{11}+\theta_{21}+\theta_{31}+\theta_{41}} \quad e_{34}(\theta_{41}) = \frac{e_{41}\theta_{41}}{\theta_{11}+\theta_{21}+\theta_{31}+\theta_{41}} \end{aligned} \tag{25}$$

The denominators of these errors can be omitted since they are just normalizing factors and will not affect training. Representing this using matrix multiplication:

$$\Theta_{34}E = \begin{bmatrix} \theta_{11} \\ \theta_{21} \\ \theta_{31} \\ \theta_{41} \end{bmatrix} \begin{bmatrix} e_{41} \end{bmatrix} \tag{26}$$

Another piece of the puzzle is to realize that the errors at each node are functions of the weights they feed out to, $e(\theta_{ij})$, as was shown in Figure 25. It would make sense that our next step would be to minimize the error function with respect to every weight in every layer in the network to improve the performance of our NN. Figure 31 demonstrates a simplified version of one of our error functions $e(\theta_{ij})$. In this figure, our weights correspond to w, the error function to $J(w)$, and the function minimum, local or global, to $J_{min}(w)$.

Figure 31: Function Minimization using Gradient Descent (Permission Pending)



We can solve an optimization problem to find the value of the weight w that minimizes the error $J(w)$ contributed by the associated node. Using calculus, we know that we can find the minimum of some convex function by taking the derivative of that function and setting it equal to zero. Unfortunately, this is often infeasible in practice as the error functions increase in complexity. Instead, we opt to use gradient descent to incrementally adjust our weights until they have reached a local or global minima.

Finding the minimum of this error function simply involves taking its gradient, or partial derivative, and adjusting the weight appropriately. Clearly, if the gradient is negative, then we should add to the existing weight and if the gradient is positive we should subtract from the existing weight, as indicated in Figure 31. This leads us to (27), the update equations for the weights.

$$\theta_{ij}^{new} = \theta_{ij}^{old} - \alpha \cdot \frac{de}{d\theta_{ij}} \tag{27}$$

α is known as the learning rate and it fine-tunes the magnitude of the update we make whenever we apply the update equations to the weights. The gradient is shown in (28)–we omit its derivation for the sake of brevity.

$$\frac{de}{d\theta_{ij}} = -(t_j - o_j) \cdot \text{Sigmoid}(\sum_i \theta_{i,j} o_i) \cdot (1 - \text{Sigmoid}(\sum i\theta_{ij} o_i) \cdot o_i \tag{28}$$

While our example was a simple 3 $\times$ 4 $\times$ 4 $\times$ 1 network, our network and others used end up being significantly larger. With the update equations explained, the NN can now perfectly learn a function that maps an input, such as an image or speech signal, to a target value. There remains one problem: the NN can only classify this single example. One of the essential features of human learning is the ability to generalize. Earlier, we showed the three pictures of the number seven with increasing noise. If the NN was trained to recognize the first seven, it would fail to recognize the other two, despite the fact that they were written in the exact same way. Additionally, if another seven was written and converted to a 28 $\times$ 28 pixel image, the network would again fail to generalize and classify it correctly. This is exacerbated with speech signals because of the drastic increase in data points per input. For example, the 28 $\times$ 28 image from the MNIST dataset stretches out to form an input vector of length 784 while our speech signal example from simplify.wav is a vector of length 38,400. Clearly, this massive increase in data points would mean that we would need a significantly larger neural network size to extract relevant features from the input. This added complexity calls for additional training examples to achieve the same performance.

In general, NNs are not very good at generalizing, requiring hundreds of thousands of examples to acquire performance close to the state-of-the-art. This data-dependency categorizes NNs as a type of supervised learning algorithm. Despite this data-dependency, supervised learning algorithms are still extremely useful if the data is available. For our application, there are a handful of open-source speech

datasets. Since we are focusing on such a small vocabulary, we can readily create our own speech dataset.

With the basics of NNs covered, we can proceed to our discussion of a new form of NNs, known as convolutional neural networks. Given that we have sufficient data, NNs seem promising; however, their primary shortcoming for our application is their instability when dealing with time-series data. Because they are not translation-invariant, small shifts in the time-domain signals means that the network will fail to recognize the speech signal. In the next section, we will see how convolutional neural networks are able to deal with this type of data.

3.2.6.2.3 Convolutional Neural Networks

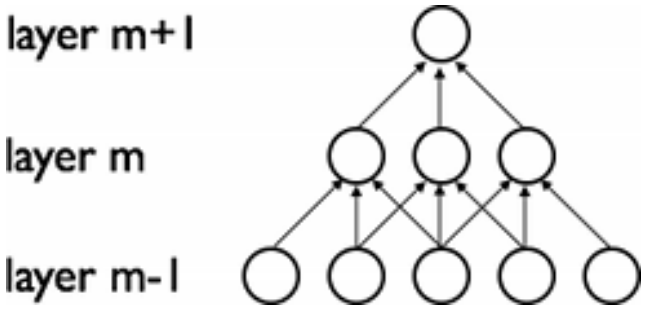
Convolutional neural networks (CNN) are quite similar to the NNs that we have discussed previously. Just like NNs, they are biologically-inspired, borrowing their properties from the neurons in the animal visual cortex. CNNs are also a type of supervised learning algorithm, requiring large training sets to improve performance. As we mentioned towards the end of the last section, CNNs are advantageous due to their ability to handle translations in a time-domain signal, the specifics of which we will cover shortly. The specific class of NNs that we were studying in the previous section are known as multi-layer perceptrons (MLP), a network of layers of fully-connected nodes. Unlike MLPs, CNNs are comprised of different layers that each perform a unique operation. The most common layers of a CNN are:

- Convolutional layer (CONV) - as noted in the name, this layer convolves the input data with a pre-specified number of filters of a pre-specified size.
- Pooling layer (POOL) - this layer performs a downsampling on the inputs, easing the computational requirements.
- ReLU layer (RELU) - this layer applies a piecewise linear function to the input layer. This function is the equivalent to applying $max(0, x)$.
- Fully-connected layer (FC) - this layer is the same as the layers used in NNs discussed previously.

The architecture of a CNN resembles the following pattern: input layer, CONV, RELU, POOL, output layer. The CONV, RELU, and POOL are packaged together in that order and are generally cascaded several times. Before we dive into the nuts and bolts of how CNNs work, it is important to realize why we should depart from MLPs. In addition to being unable to handle translations in the signal, MLPs are computationally inefficient. Let us take an image from the MNIST dataset as an example: a 28 $\times$ 28 pixel image. This leads to an input layer with 28 $\cdot$ 28 = 784 nodes. With 10 possible numbers, we would need 10 nodes at the output layer. If we can assume a 784 $\times$ 100 $\times$ 10 network, 100 nodes in the hidden layer corresponds to 784 $\cdot$ 100 = 78,400 parameters to compute. While this might be feasible, if we instead take a 200 $\times$ 200 pixel image, the number of parameters we

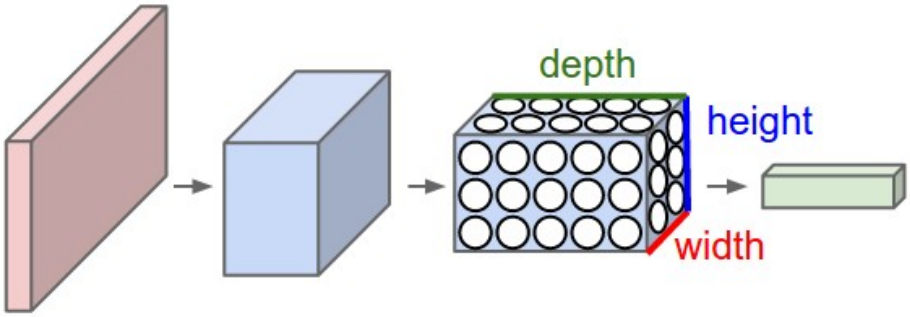
have to compute now goes up to $(200 \cdot 200) \cdot 100 = 4{,}000{,}000$ parameters. Since each layer is fully connected to the next layer, larger images and/or network sizes are too cumbersome for these networks to handle efficiently. On the other hand, the nodes in a CNN take advantage of local connectivity, as shown in Figure 32. This feature alone drastically reduces the computational requirements of forward and backward propagation.

Figure 32: Locally-Connected Neural Network (Image Available in Public Domain)



With local connectivity, we can control the number of parameters in a CNN to avoid the computational explosion that occurs with standards NNs by taking advantage of parameter sharing. Due to the fact that our speech signal might have a slight translation, parameter sharing allows us generalize the features we extract from one part of the image. To put this more concretely, suppose we have two speech signals, one that starts at time t_1 and one that starts at some later time t_2. While the standard NN would be unable to recognize utterance, the CNN takes advantage of parameter sharing to gain the translation-invariant With an understanding of why we would like to depart from MLPs, we can now shift our discussion back to CNNs. With respect to data flow, it is easiest to think of CNNs as taking an input volume, a height $\times$ width $\times$ depth signal, and transforming it into an output volume containing the class scores (e.g., 10 class scores for our MNIST image example). Figure 33 demonstrates this transformation.

Figure 33: Example of an Convolutional Neural Network (Permission Granted by Andrej Karpathy)



At a very high-level, parameter sharing allows us to take the weights and bias and share them across all of the filters. Having a better idea of how the input volume is transformed by the network's layers, we will need some more definitions to understand the output layer.

- Depth - The depth of the output volume corresponds to the number of filters we would like to use in the CONV layers. For example, if the first CONV layer takes an image from the MNIST dataset as input, then different nodes along the depth dimension may activate in presence of various oriented edges, just like receptive fields do in the animal visual cortex. We will refer to a set of node that are all looking at the same region of the input as a depth column.
- Stride - The stride specifies the amount of which we slide the filter. For an image, when the stride is one then we move the filters one pixel at a time. Similarly, when the stride is two then the filters jump two pixels at a time as we slide them around.
- Zero-padding - The zero-padding pads the input volume with zeros around the border. This specification allows us to control the spatial size of the output volumes, preserving the size of the input volume so the input and output width and height are the same.

3.2.6.3 Programming Languages

3.2.6.3.1 C/C++

C and C++ are software programming languages. They are both very useful for all around, general purpose programming with a static type design. They are also designed with the procedural programming paradigm in mind. However, C++ has the added benefit of offering multiple paradigms, including object-oriented programming.

C first appeared in the 1970s and was designed by Dennis Ritchie at Bell Labs and became one of the most popular coding languages of all time. C++ emerged in the 1980s C programming language also is standardized by the American National Standards Institute. Both C and C++ are have also become standardized by the International Organization for Standardization (ISO).

Both are low-level programming languages that are extremely useful for system and embedded programming. They also have the advantage of being very available on a multitude of systems. They also have served as the inspiration of many programming languages that developed, including Java and C#.

C++ has the advantage of the object oriented design paradigm. This is essentially the C programming language but with the ability to create classes. Object oriented programming methodology is extremely useful because it allows for modular design. When creating complex software projects, it is helpful to abstract different parts into objects and classes. Each of these objects include attributes and

methods associated with it. It also has the added feature of reusable code because through inheritance, similar entities do not need to be created completely from scratch. It also contains new features such as virtual functions and operator overloading.

Although C++ is not taught at the University of Central Florida anymore, its similarities to the C programming language makes it a lot easier to learn. The IDEs that we will be using also provide support for these languages.

3.2.6.3.2 Lucid

Lucid is a relatively new dataflow programming language. It was designed by Edward A. Ashcroft and William W. Wadge in the 1970s. Dataflow programming is a paradigm of software architecture that is also known as stream processing. It is also known as reactive programming because it is typically used in non-Von Neumann processors. Rather than follow a more procedural design, Lucid utilizes a methodology that operates more based on the flow of information.

The syntax of Lucid varies widely from other programming languages. Indeed, that is intentionally included as part of the design. It is made to encourage the idea that the data flows are the objects at hand. However, it is very similar to Verilog but influences from C++ and Java can also be found.

As a Hardware Description Language (HDL), Lucid is extremely helpful in programming embedded systems such as FPGAs. Due to the fact that it eliminates the need for creating finite state machines and various flip-flop plans, the Lucid programming language greatly simplifies digital design projects.

Lucid programming files are often implemented with a sole declaration of a module. This module declaration also includes a parameter list. The parameter list is optional and any parameters listed should have a specified value. The port list is also included in the module. It lists the signals that are sent and received by the module. These signals can be classified as one of the following types: Inputs, Outputs, and Inouts.

Inputs refer to read only data that is sent to the module. Outputs refer to information written inside the module that would be sent out. Inouts are signals that can be both read and written. The main part of the Lucid Programming File is made up of variables, instantiations of modules, and various programming blocks. These programming elements are very similar to most languages.

There are many advantages to the Lucid programming language. This includes a decrease in the amount of code that is needed to write for a project as well as providing for debugging in real time. This is possible through the Mojo IDE, which we can use to interface with our selected hardware. The challenges in using Lucid is that it is a new programming paradigm to learn, as well as that it is still new syntax to learn.

3.2.6.3.3 Verilog

Verilog is a Hardware Description Language (HDL) that came about in the 1980s and grew more commonly used as deficiencies were addressed in the 2000s. It is statically typed and is most popular for use in digital design projects. This also includes the verification of digital circuits, analog circuits, and mixed-signal circuits. As an HDL, Verilog includes the feature of delineating time propagation and signal sensitivity. Due to the fact that Verilog provides both non-blocking and blocking assignment operators, one could update finite state machines without the need for initializing and declaring temporary variables.

The advent of Verilog led to an increase in productivity for hardware designers that were already taking advantage of schematic software. The syntax of Verilog was designed with the C programming language in mind. This was due to the fact that the C programming language was already very popular for software engineering development. Much of its semantics are very similar to those in C, including its control flow keywords. Verilog can be implemented through a hierarchy of modules. These modules are designed with an internal hierarchy that allows them to communicate with other modules. These modules have inputs, bidirectional ports, and outputs that are declared within them. Although these modules include sequential programming statement blocks, they are executed concurrently. Hence, they are also classified as a dataflow language.

Although much of the Verilog programming language is not directly associated with an analog counterpart, it utilizes direct mapping instead. Each Verilog process has two special keywords used for declaration: always and initial. Verilog also has the fork/join keyword pair to build processes that will be run in parallel. The corresponding IEEE standard for the Verilog language also delineates four-valued logic. These four states are 0, 1, X, and Z.

Verilog is very useful in FPGA development and includes the advantage that there are a lot of resources and tutorials out there. In our project, Verilog would be idea for interfacing and customizing our FPGA. It can also be used to develop tools that can easily interface with other devices.

3.2.6.3.4 Python

Python is another all-purpose programming language with the high-level and dynamic type distinction. Like C++ it is also applicable to multiple paradigms, including object oriented, functional, and procedural programming design.

Python is a very new language, having been created in the early 1990s and ever growing in its popularity today. Python was designed with the beginner programmer in mind and focuses on code readability. This readability is ensured on both small and large scale programming projects.

This programming language is also available on many operating systems. It is also has influences from C/C++ and Java. It's reference, CPython, is open-source and maintained by a community of developers.

Python has of late been gaining traction in due to its broad appeal and application. In universities, Python is useful as a good beginner's language due to its strict structure. Python is also popular within the scientific community for scripting and data science purposes. The advantages of Python are many – including ease of learning, readability, and its wide applications.

The disadvantages are that as it is new, there are not as many resources out there especially with regards to embedded development. Indeed, lower level languages are far preferable for those purposes.

3.2.6.4 Graphical User Interface

3.2.6.4.1 Visual Studio

Visual Studio is a Microsoft integrated development environment (IDE) that is often used for software development on Windows OS. It is offered at a Freemium license and supports development in C, C++, VB.NET, C#, and F# in house. However, support for other languages can be installed through separate components. It also provides a quick and easy way for creating graphical user interfaces. Through a What You See is What You Get (WYSIWYG) design view, you can drag and drop components and generate the skeleton for programming the widgets.

3.2.6.4.2 QT

Qt is an open source governed application framework commonly used to develop graphical applications on many platforms including Windows, Linux, and more. Qt Creator is a C++ and QML IDE with an integrated graphical user interface design tool called QT designer. This software aims to simply application development with its own tools.It also allows for easy use with Python through PyQt libraries.

3.2.6.5 Xilinx ISE Design Suite 14.7

Xilinx ISE is a development environment used for the analysis and synthesis of hardware-description language code as well as the place and route and configuration stages of FPGA programming. The ISE GUI contains a design hierarchy window, source code editor, and a console that displays status, warning, and error messages. This configuration, while looking outdated and overly complicated, allows us to develop our design and program the Spartan-6. The FPGA present on our Mojo development board (the same one we will use for our final design) can be only be targeted using Xilinx ISE. The bit stream design that can be generated through using the ISE Design Suite software is especially important as this is what our chosen FPGA can utilize and read. It is particularly great for designing with teams. For modeling and simulation we use Mentor Graphic's ModelSim rather than Xilinx's ISim due to familiarity with its interface and a richer debugging environment.

Xilinx has partnered with third-party community to offer highly capable tools, features, software libraries, as well as design methodologies. This allows anyone to

get quickly up and running with designing tools for FPGAs and other embedded systems.

4 Related Standards and Design Constraints

4.1 PCB Standards

- IEEE 1149.7-2009 – requires any device functioning as a debug and test system to provide pull-up bias on the TMS and TDO pins.
  - The JTAG-SMT2 that we are incorporating into our PCB design has weak pull-up resistors of 100KOhms on the TMS, TDI, TDO and TCK pins in order to meet this standard.
- IEEE 1149.1 – requires the TAP and boundary-scan architecture include all mandatory elements including: the TAP, the TAP controller, the Instruction register, the instruction decoder, the boundary-scan register and the BYPASS register.
  - The Spartan-6 FPGA featured on our PCB design is fully compliant with this standard.
- 2002/95/EC (ROHS) requirements for safety don't contain more than a certain number of parts per millions of certain dangerous substances.

4.2 FPGA Constraints

Several factors were taken into account when choosing an FPGA to implement our project. These factors included the, size, cost, ease-of-use, availability, and the vendor of certain chips. Moreover, once the chip was chosen it placed constraints on the performance of our neural network. Thus, all the factors listed above place real restrictions on what is achievable in our project. In the following paragraphs a detailed analysis of how these factors affected our FPGA choice, and in turn our neural network, will be given.

The size of our neural net is directly proportional to the amount of logic we have available for use on the chip. A larger chip (in terms of CLBs/slices) allows us to increase both the size of our neural net and our throughput (number of classifications per second). By modifying the size of the neural network we can quite possibly improve the accuracy of our speech recognition system. Our hardware-accelerated algorithm must have the capability of inferencing speech in real-time, therefore the chip's throughput must be high enough to achieve this design specification.

To approximate the amount of logic needed to implement the neural network algorithm on an FPGA, a rough prototype was written in Verilog that gave us, at the least, a solid reference point to begin our search. Additionally, we used research papers referencing FPGA-based neural networks and their logic usage to deepen our understanding of their size requirements. We reached the conclusion that logic utilization is heavily dependent on the desired processing latency and the number of nodes/layers in the neural network. There is no solid figure or table to deduce this information from. It would be possible to create a design capable of classifying

kilobytes of data every dozen clock cycles, but this would require an enormous, expensive FPGA. Instead, we took the prototype design and pushed the limits of our development board, finding that it adequately held a neural network capable of satisfying our design specifications. Additionally, we ran the same prototype through Quartus's compilation process to obtain logic utilization figures for Altera chips (our development board contained a Xilinx chip).

Another important factor was the cost of our FPGA and the cost of getting that FPGA onto a custom circuit board. Most of the FPGAs we looked at were capable of implementing a neural network very efficiently, however their costs ranged from tens of dollars to thousands of dollars. Early on, we chose to look at FPGAs that came in a quad-flat-package, thus reducing the number of layers necessary for our PCB (ease-of-use), and subsequently, its costs. Most higher-end FPGAs make use of ball-grid-array packaging, a package type that requires very careful soldering and can lead to errors that are hard to diagnose without x-ray tools. With this constraint in mind, our selection was limited to low-end, low-cost FPGA solutions, which was fine given that our development board contained a low-end FPGA that could effectively implement an algorithm that met our specifications.

From the start it was decided to limit our options to those FPGAs offered by Xilinx and Altera. We could have chosen other manufacturers such as Lattice and Microsemi, but having attained most of our FPGA experience using Xilinx and Altera chips, in addition to the availability of mature ecosystems and support networks surrounding them led us to stick with what was tried and true. After searching heavily, it was determined that many candidate Altera FPGAs were not readily available, meaning we would have to wait weeks or months for the factory to produce them and leave our final PCB construction to the last couple of months of senior design. Having never designed a PCB before and uncertain of how many iterations of the design we would need to go through, we opted to choose from Xilinx's product line, which had a much higher availability on websites such as Digikey.

In the end, these factors all combined and limited the scope of what was possible in our project. Had we gone for a middle-tier FPGA in the range of $300-$500, it would have been possible to classify data using our neural network much faster, allowing us to showcase our capabilities in a more impressive manner. For example, a neural network on a Xilinx Kintex-7 FPGA would likely be capable of classifying speech data at a rate hundreds of times faster than needed for real-time speech recognition. This would mean we could blow through hundreds of hours of a validation set in a mere fraction of the time and obtain results almost instantly for our demo in Senior Design II. With the ultimate goal being that we process speech at a rate fast enough for real-time recognition, these ambitions were discarded and we instead went with the low-end Xilinx chip that will be described in detail in a later section.

4.3 Algorithmic Constraints

In addition to the constraints addressed in the previous section, the NN will have its own constraints. Among these contraints are:

- Performance loss due to hardware restrictions
- Size/availability of the training data

In running any algorithms on an FPGA, we have to be efficient when performing certain operations. This means that certain properties of the network, such as the activation function, may have to be altered to work on the FPGA. The activation function, as discussed before, is a sigmoid function:

$$\text{Sigmoid}(X) = \frac{1}{1 + e^{-x}}$$

Clearly, this continuous function would be difficult to implement on a digital system. As discussed in a previous section, we end up using the combinational approach to approximate the sigmoid on the FPGA. Fortunately, this approximation is quite efficient and only comes with some performance loss of several percent.

The other constraint that we have is the size and availability of our training data. Because of the specificity of our task, we will need to curate our own dataset and perform extra pre-processing on the speech signals to ensure that our performance does not suffer. Additionally, to offset performance losses and lack of training data, we have the ability to run multiple epochs on the same training data. The only downside to this method is the risk of overfitting: if our training data is not sufficiently diverse, overfitting will prevent the model from generalizing well in response to new speech signals.

5 Firmware, Hardware, and Software Design Details

5.1 Firmware

5.1.1 Firmware Introduction

Historically, the term firmware was used to refer to software that sat in dedicated read-only memory. It was typically executed by some embedded system and considered not nearly as flexible as software. FPGA code has no official definition, and is termed differently from industry to industry, thus for the purposes of this report, the digital circuit that is implemented by the FPGA as well as its corresponding programming files will be referred to as firmware.

Hardware description languages (HDLs) are used to generate digital circuits in a procedure that begins with register transfer-level netlist synthesis. Once the Verilog/VHDL describing the circuit is deemed syntactically correct, the chosen development environment (for this project that is Xilinx ISE 14.7) creates a netlist based on the code. This netlist is a low-level virtual digital circuit that describes at the logic gate level the connections between building blocks such as registers, multiplexers, decoders, and basic Boolean gates.

With synthesis complete, the IDE takes the final netlist and maps it to the FPGA through a process termed place and route. Here, the software attempts to “fit” the netlist onto the FPGA by routing buses between digital logic, ensuring the right logic is connected to GPIO pins, calculating the final timing of the circuit, and much more. If the timing of the circuit meets the designer’s specifications and the netlist meets the size constraints of the FPGA, place and route is said to have completed successfully. The output of place and route is a file (.hex or .ttf) that is used to generate a programming file. Once created, this programming file is sent over to the FPGA and used to actually configure the logic on the chip. The contents of programming files are indecipherable, proprietary, and vary from vendor to vendor.

In the following sections, the report will focus on the contents of our firmware and explain the architecture that implements our artificial neural network.

5.1.2 Firmware Architecture

One of our primary goals in this project is to use our hardware-accelerated multilayer perceptron neural network to achieve a speech recognition accuracy comparable to other, more common solutions (when adjusted for factors such as cost, power efficiency, and throughput). To this end, we have focused on making our digital architecture as flexible as possible. Creating our circuit with malleability in mind makes it easier to achieve our accuracy goals by fine-tuning the size, speed, or throughput of our neural network, as doing these things only requires modifying parameters in an include file before compilation. These parameters propagate throughout the design and help generate the amount of logic necessary to reach certain size and throughput specifications. By implementing our neural network in

a highly modular design and parameterizing almost all aspects of these modules, these changes become painless and do not require rewriting entire blocks of code should we decide to tweak our network.

Several terms will be used throughout the following sections that refer to the aforementioned neural network parameters. The most important of these are the NUM_TILES, SIZE, and LOOP values. The NUM_TILES parameter is self-explanatory, determining how many tiles are present in the network. This value does not count the input layer, so a neural network with 4 layers (1 input layer, 2 hidden layers, and 1 output layer) would have a NUM_TILES value of 3. The SIZE factor determines the number of processing units per tile. This number is directly proportional to the chip's logic utilization. Furthermore, increasing it while decreasing the LOOP value to maintain the same number of nodes in a tile increases the speed of the network. The LOOP value determines how many node outputs are calculated by each processing unit. For example, a SIZE value of 16 and a LOOP value of 64 instructs the synthesis software to generate 16 processing units, each of which calculate the output of 64 nodes. Changing the LOOP value does not affect the final logic usage in any meaningful way. At most, registers that are used to store counts must be wide enough to accommodate the LOOP value, so changing these registers widths by several bits affects the circuit's logic utilization almost imperceptibly. Increasing the LOOP value increases the latency of the network because many more cycles are needed to calculate one layer's outputs. The following equation can be used to determine the time it takes for the network to classify input data.

Latency (clock cycles) = (X + Y + 1) * Z + 1

Where:
X = Number of nodes in the neural network's largest tile
Y = SIZE value of the tile following the largest tile
Z = LOOP value of the tile following the largest tile

Additional cycles are added to account for overhead during processing. Only the parameters of two layers need to be taken into account because of the pipe-lined nature of the network. Using this equation, it can be determined that for a network whose largest tile contains 1024 nodes and whose tile following that tile has 16 processing units with a LOOP value of 64 (1024 total nodes), classification can be expected to occur in 66625 clock cycles. At a reasonable clock frequency of 100 MHz, this number implies that this particular configuration of the network can classify 1 kB (1024 nodes * 1 byte per node) of input data 1500 times per second. Of course, there are factors external to the network that can affect processing time, such as the bandwidth of the communication ports and the ability to distribute the network weights on a per-cycle basis.

In this way, we create a serial-parallel architecture where the speed and size of our neural net can be balanced to find a sweet spot that meets our design specifications while fitting well on our chip.

5.1.2.1 Processing Unit

The basic building block of our design is the processing unit (PU). It is used purely to implement the data path in our neural network. A processing unit takes as input an n-bit weight (n = 3,4, or 5), 8 bits of input data, a clock, an enable signal, and a reset signal. Using these signals it is capable of computing the product of the input weight and input data, adding this product to an existing 16-bit sum, and outputting the 16-bit sum to the rest of our circuit. This can occur as fast as our design is clocked, meaning with a somewhat conservative clock rate of 100 MHz this unit can perform 100 million multiply-accumulate operations per second.

Setting the reset signal high will cause the register holding the accumulated sum to return to a predefined state. For our purposes, it resets to holding the node bias value associated with our network. The enable signal (active high) instructs the processing unit to perform multiply-add operations. In this way, the unit is purely a slave of a control logic circuit.

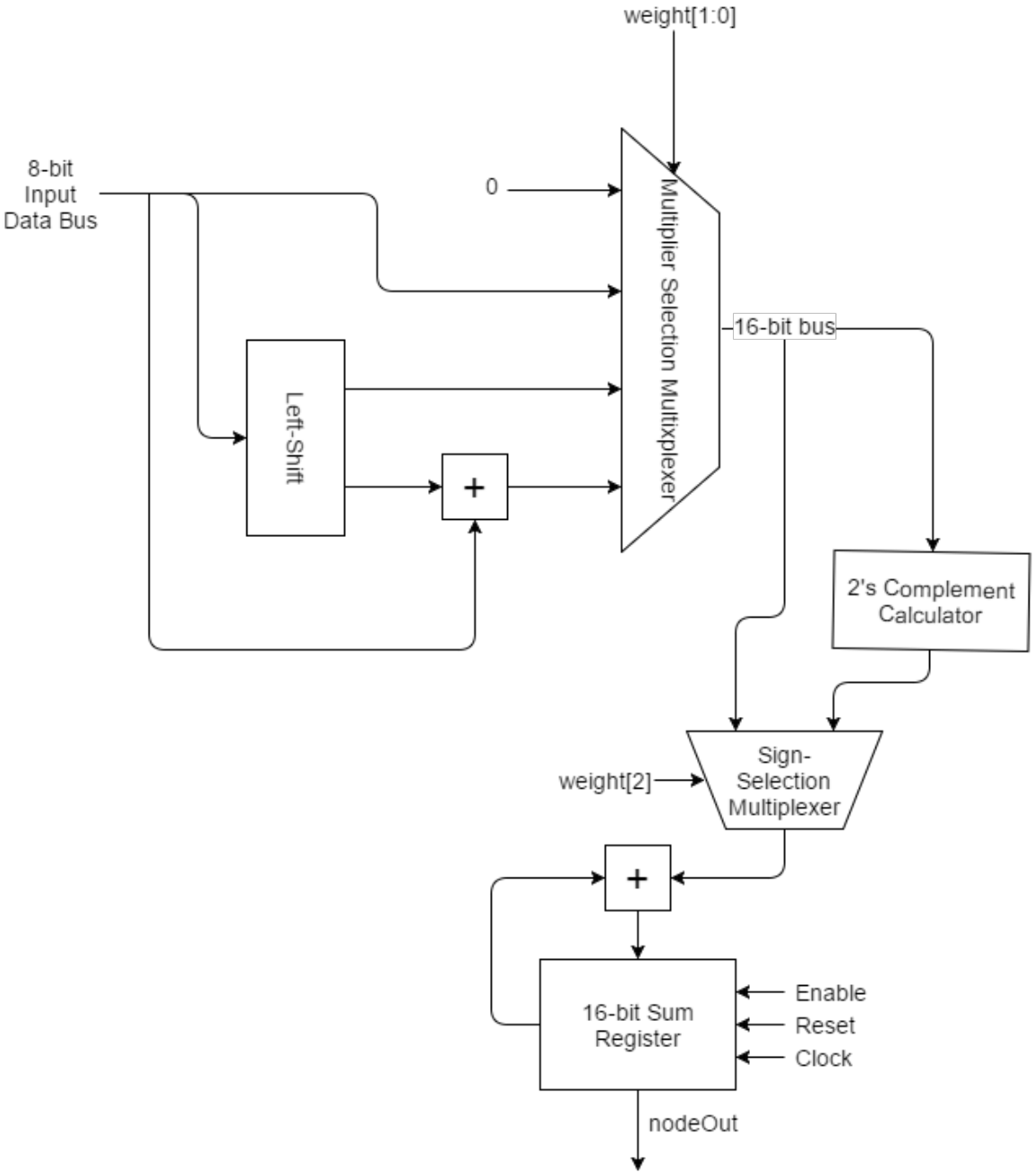
The processing unit does not use a multiplier anywhere. These are expensive in terms of logic utilization (logic is a limited resource on an FPGA) and by implementing our neural network using only n-bit weights (n = 3,4, or 5) weights we were able to remove the need for one at all. Depending on the width of the weights, the arithmetic behaviour of the processing unit can change. For example, 3-bit weights, due to their sign-magnitude representation, are capable of representing integer values in the range of -3 to 3, inclusive. Thus, we only had to take these multiplication factors into account. A weight of 0 sets the product to 0, a weight of 1 sets the product to the input data, a weight of 2 sets the product to the input data left-shifted once, and a weight of 3 sets the product to the sum of the input data and the input data left-shifted once. The process is exactly the same for negative weights, except that the 2's complement of the product is taken before being added to the existing sum. This concept is extended to 4 and 5 bit weights that represent values in pre-determined ranges. In all cases, multiplications and divisions are implemented using bit-shifts and additions. For example, in our 4-bit weight representation the binary value ”0100” instructs the PU to perform a MAC operation with a multiplication factor of 0.75. This is done by left shifting the input value once and twice and then summing the resultant bit-shifted values.

To elaborate on the example above, the weights in our system are represented symbolically, rather than literally. For example, the binary value ”0100” can denote the decimal value 4, 0.5, or 0.25 depending on the number of fractional bits. We found that most weights were distributed pretty tightly around 0 and wanted to have the majority of the weights fall in the range [-0.5,5] while still having the ability to multiply by 1 or 1.5. To accomplish this we used the symbolic notation mentioned above which allowed us to represent weights non-linearly.

To sum up the operation of a processing unit, it first computes the product of the weight and input data using the process outlined in the previous paragraph, followed by extending the resulting 8-bit product to fit a 16-bit register, . It then computes the 2's complement of this product if necessary (if the weight is negative, i.e.

the sign bit of the weight is set high) and adds it to a pre-existing sum. A register-transfer level depiction of a processing unit can be seen in Figure 34 below. Note that the figure shows a slightly simplified version of the digital circuit.

Figure 34: Processing Unit Depicted at the Register-Transfer Level



5.1.2.2 Combinational Approximation of the Sigmoid Function

Equally as important as the processing unit is the sigmoid circuit which takes the output of the processing units and applies nonlinearity in the form of the sigmoid function. This is an essential part of our neural network. We used the a sigmoid approximation based on the sig_337p proposed by M.T. Tommiska in his paper "Efficient digital implementation of the sigmoid function for reprogrammable logic." The sig_337p is a purely combinational approximation, which according to Tommiska has an average deviation from the true sigmoid function of 0.17% and a maximum deviation of 0.39%.

After extensive testing, we found that using sig_337p introduced too much error into the forward pass computation. The source of this error was the truncation of the output of the processing units to fit the input of the sigmoid approximation function. By truncating bits, a non-negligible amount of precision was lost, causing small discrepancies between input and output values. These errors would accumulate after hundreds, if not thousands, of additions and ultimately lead to an output layer value that deviated up to 20 percent from the expected value.

To solve this issue, I sought to create a similar sigmoid approximation function with a wider input and more precise output. Using a MATLAB implementation of the Quinne-McCluskey logic minimization algorithm and some C++ code that mapped fixed-point inputs to fixed-points outputs I generated the AND-OR logic necessary to create sig_368p. Sig_368 combinationally mapped 10 input bits (1 sign, 3 integer, and 6 fractional bits) to 8 output bits. The addition of the 3 bits of input precision helped lower the output layer deviation to within a negligible range (0.3-0.4 percent). In addition, the output bus has the advantage of fitting well with our existing 8-bit bus widths. One disadvantage of increasing the width of the input bus is the complexity of the logic necessary to implement a more accurate mapping. The number of required AND gates increased by a factor of 6, though the required logic was still an extremely small portion of the FPGA's finite logic resources. The more important consideration was the increase of the propagation delay in our circuit. Suddenly, sig_368p included nearly all the largest combinational delays in our datapath. To solve this issue, we simply registered the input and output of the sigmoid function, allowing us to keep clocking our logic at 100 MHz.

Sig_368 can be implemented as a sum-of-products, meaning a simple AND array feeding an OR array is enough to form the combinational circuit. Nevertheless, there are several nuances that have to be taken into consideration when using sig_368p. The letter "p" implies that only positive input values are mapped to output values, thus if the sign bit of the input is negative, its two's complement is taken before being transferred to the AND-OR planes. Similarly, the output of the AND-OR planes is subtracted from the value 1 if the sign-bit of the input is negative. In this way, logic utilization of the circuit is halved while maintaining the ability to approximate the sigmoid function for both positive and negative input values. It is important to note that input data is assumed to be in the fixed-point format s3.6,

where there is 1 sign bit, 3 integer bits, and 6 fractional bits. Data is output in the fixed-point format 0.8, implying there are 8 fractional bits. This is optimal as the sigmoid function has outputs in the range (0,1).

Below, Figure 35 contains the Boolean logic necessary to implement sig_337p as a sum-of-products. A p corresponds to the output of an AND gate and an s to the output of an OR gate. Note that s[6] is 1 because sig_337p calculates values only for positive input values, therefore the output value is always at least 0.5. A complete Verilog description of sig_368 can be found on DeepGate's GitHub repository.

Figure 35: Logic Required to Implement Sigmoid Function

sig_337p Logic Table			
AND Array		OR Array	
Gate	Gate Output	Gate	Gate Output
P_0	$X_5 \wedge X_2$	S_0	$P_0 \vee P_1 \vee P_3 \vee P_4 \vee P_6 \vee P_7 \vee P_9 \vee P_{12} \vee P_{13} \vee P_{14} \vee$
P_1	$X_5 \wedge X_4$	S_1	$P_2 \vee P_5 \vee P_6 \vee P_7 \vee P_{11} \vee P_{12} \vee P_{15} \vee P_{18} \vee P_{23} \vee P_{26}$
P_2	X_5	S_2	$P_2 \vee P_5 \vee P_6 \vee P_8 \vee P_{11} \vee P_{20} \vee P_{21} \vee P_{22} \vee P_{22} \vee P_{24}$
P_3	$X_5 \wedge X_3$	S_3	$P_2 \vee P_5 \vee P_{10} \vee P_{12} \vee P_{16} \vee P_{17} \vee P_{20} \vee P_{25} \vee P_{23}$
P_4	$X_4 \wedge \neg X_3 \wedge \neg X_2 \wedge \neg X_1 \wedge X_3$	S_4	$P_2 \vee P_4 \vee P_5 \vee P_9 \vee P_{10} \vee P_{14} \vee P_{15} \vee P_{19} \vee P_{23} \vee P_{25}$
P_5	$\neg X_4 \wedge X_3 \wedge \neg X_2 \wedge \neg X_1 \wedge X_3$	S_5	$P_2 \vee P_4 \vee P_7 \vee P_9 \vee P_{10} \vee P_{11} \vee P_{12} \vee P_{13} \vee P_{14} \vee P_{15} \vee P_{17} \vee P_{22} \vee P_{23} \vee P_{25}$
P_6	$\neg X_4 \wedge \neg X_3 \wedge X_2 \wedge X_1 \wedge X_3$	S_6	1
P_7	$X_3 \wedge \neg X_2 \wedge X_1 \wedge \neg X_0$		
P_8	$X_4 \wedge X_3 \wedge X_1 \wedge X_0$		
P_9	$X_4 \wedge \neg X_3 \wedge X_1 \wedge X_0$		
P_{10}	$X_4 \wedge X_2 \wedge X_1$		
P_{11}	$\neg X_4 \wedge X_3 \wedge X_1 \wedge X_0$		
P_{12}	$X_3 \wedge X_2 \wedge X_1$		
P_{13}	$X_3 \wedge \neg X_1 \wedge X_0$		
P_{14}	$X_4 \wedge X_2 \wedge X_0$		
P_{15}	$X_4 \wedge \neg X_3 \wedge \neg X_2 \wedge X_1$		
P_{16}	$\neg X_4 \wedge \neg X_3 \wedge \neg X_2 \wedge X_1$		
P_{17}	$\neg X_4 \wedge X_3 \wedge X_2$		
P_{18}	$X_4 \wedge X_3 \wedge X_2$		
P_{19}	$\neg X_3 \wedge X_2$		
P_{20}	$\neg X_4 \wedge X_2 \wedge X_1 \wedge X_0$		
P_{21}	$\neg X_4 \wedge X_2 \wedge \neg X_1 \wedge X_0$		
P_{22}	$X_4 \wedge \neg X_3 \wedge X_2 \wedge \neg X_1$		
P_{23}	$X_4 \wedge \neg X_2 \wedge \neg X_1 \wedge X_0$		
P_{24}	$\neg X_4 \wedge \neg X_3 \wedge \neg X_2 \wedge X_0$		
P_{25}	$X_4 \wedge X_3$		
P_{26}	$X_4 \wedge X_3 \wedge \neg X_2 \wedge \neg X_0$		

5.1.2.3 Tile Data Path Structure

Several processing units, a multiplexer, a combinational sigmoid circuit, an embedded RAM block, and some overhead logic compose the data path of what is called a tile in our design. Tiles are analogous to layers in a neural network, with the exception that the processing units they are composed of can be used to compute the output of multiple nodes. A tile takes as input that same signals a processing unit does along with an input “transfer” signal, and an output “dataWrite” signal used to write processing unit output to memory. Furthermore, the bus transferring the network weights from memory is widened to allow transfer of the weights for all the processing units comprising the tile in one clock cycle. The enable, reset, and input data bus signals are mapped directly to the processing units in the tile.

Additionally, the number of processing units in a tile is flexible, allowing us to modify the size of our network by merely changing a parameter in an include file before compilation begins (size cannot be changed when the FPGA is programmed). In any case, the upper byte of the output of each processing unit is sent to a multiplexer whose selector is wired to an internal register that increments until a control signal is set low. The output of this multiplexer forms the input of the tile’s combinational sigmoid circuit. This setup allows us to compute the sigmoid approximation of one processing unit per clock cycle. While it would be possible to compute the sigmoid for all the processing units in a tile in one cycle, it has large logic utilization costs as data buses must be widened immensely. This way, while slower, keeps the data bus at a width of 8 bits while still being fast enough for our purposes.

The transfer input signal to the tile data path commands the tile to begin writing the output of the sigmoid circuit to a RAM block. A control unit separate from the data path ensures that this signal is high only for a number of cycles equal to the number of processing units in the tile (since the output of only one PU is written at a time). While the transfer signal is high, the tile’s data path sets the “dataWrite” signal high and sends each PU output through the multiplexer to the sigmoid circuit, whose output is wired to the input of the aforementioned RAM block. The “dataWrite” signal serves as a write enable signal to this RAM.

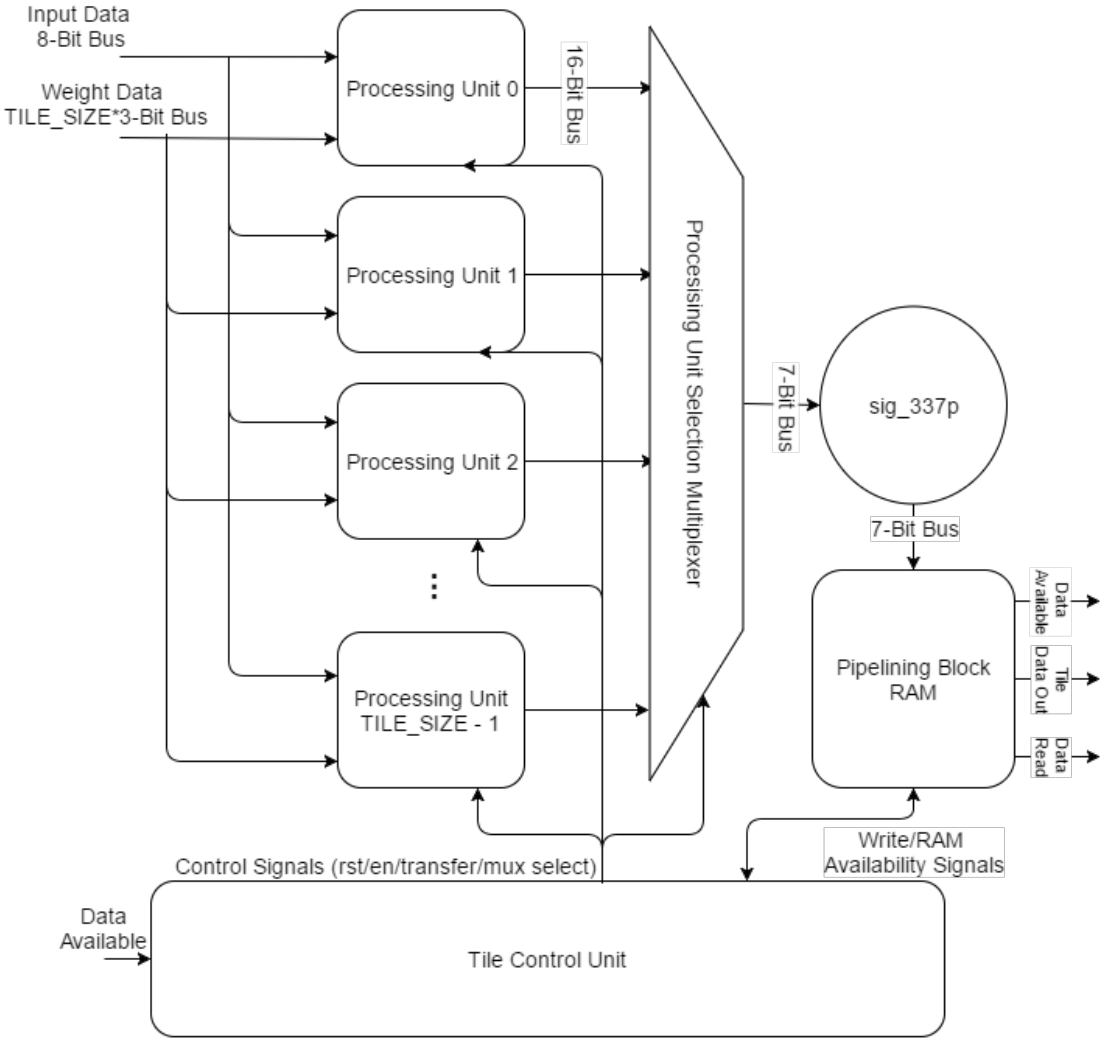
Placing RAM blocks at the output of the sigmoid circuit, and essentially the “layer,” allows data to be pipelined between tiles. To facilitate this pipelining, the RAM block is able to hold data corresponding to twice the amount of nodes in the tile. For example, if a tile has 8 processing units and each processing unit computes the output of 2 nodes, the RAM is capable of storing 32 node values at once. The purpose of this is to allow the tile to work on new data from the tile preceding it without overwriting the values it has computed previously. To achieve this, each RAM block has a small controller that tracks which sector (first or second) has been written by the tile or read by the following tile and adjusts the read and write addresses accordingly. Furthermore, the controller is aware of the number of iterations the following tile needs to compute its outputs and does not allow the tile to overwrite data unless the amount of memory read loops reaches the aforementioned number of iterations. The same controller communicates with the tile’s main controller,

which will be elaborated upon in the following section, using a signal that indicates if there is data available to be worked upon and another signal that indicates whether or not the RAM is full. Thus the rest of the data path is unaware of these signals and merely serves as a number crunching machine. Structuring our circuit this way allows us to keep the data and control logic completely separate and easy to modify should the need arise.

We had two options for storing our tile's output data, block RAM and distributed RAM. Using distributed RAM implies giving up logic resources that could be used for the other portions of our data path and control structures because of the fact that Xilinx's software maps our memory to slices and flip-flops. Quartus works similarly, mapping our RAM to ALMs that could be better used elsewhere. Taking advantage of block RAM ensures we are using dedicated memory resources on the FPGA. These are not reconfigurable in the sense that they are printed onto the silicon wafer during manufacturing. Their bus widths and read/write timings can be modified within certain constraints but they cannot be used to implement other general-purpose digital logic. Therefore, using block RAM allowed us to keep the logic utilization on the chip much lower than would be possible using distributed RAM.

The overall structure of a single tile can be seen in Figure 36. Some minor control signals are missing but the architecture and functionality is depicted accurately.

Figure 36: Simplified Tile Architecture



5.1.2.4 Tile Control Structure

Each tile has a control unit that directs the flow of data through its data path. Essentially, this circuit is a state machine whose output depends on the size of the tile it belongs to, the size of the previous tile in the network, and the signals given by the embedded RAM controllers of the current and previous tiles. To explain the operation of this control logic and paint a more concise picture of the operation of our neural network, the following paragraphs will go through each state in detail.

A tile control unit always begins in, or is reset to, the READY state. In this state, the control logic ensures that the tile “idles” in the sense that no data is being pumped through its processing units and nothing is being written to the tile’s memory. The state machine remains in this state until the “dataAvailable” signal of the RAM block in the previous tile indicates that the previous tile has successfully completed processing.

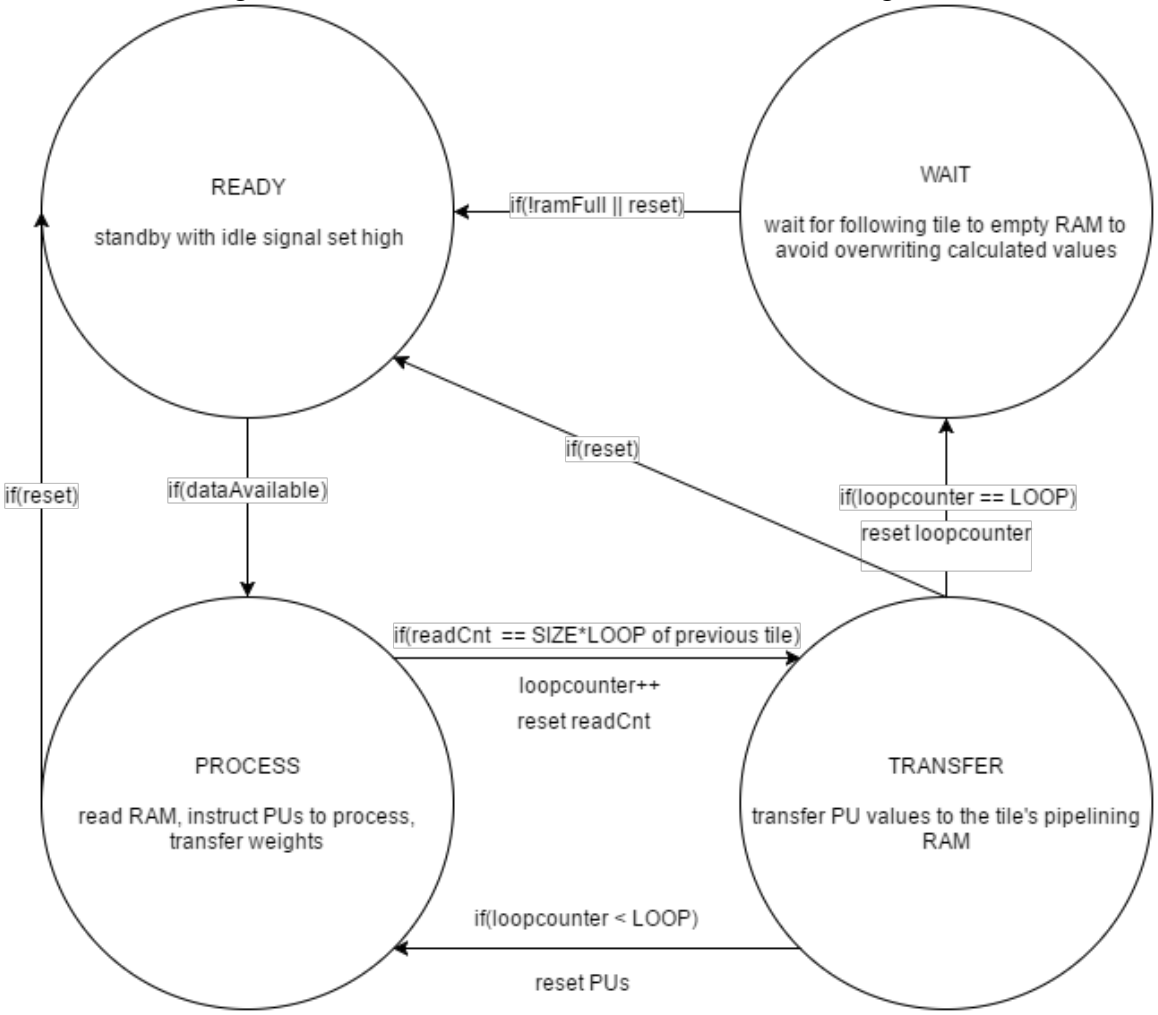
Following this, the controller immediately enters the PROCESS state, where data is read from the previous RAM block and sent through the processing units comprising the tiles in parallel. In this state, the enable signal wired to the inputs of the processing units is set high for a number of cycles equal to the number of nodes (not PUs) in the previous tile. This has the effect of sending the output of each node in the previous layer through each processing unit in the current layer. While this is occurring the PUs are taking these node values, multiplying them by their associated weights, and accumulating the sums.

Once each node value in the previous layer has been passed through the current layer, the control state machine enters the TRANSFER state, where the tile's nodes are reset to their bias values and the final sums calculated by the processing units are written to the tile's RAM. This is accomplished by setting the transfer signal high (the same one mentioned in the previous section) for a number of cycles equal to the number of processing units in the tile. Here the LOOP value of the tile comes into play. If the LOOP value is greater than 1, it implies each processing unit is to calculate the output of multiple nodes. The state machine has been tracking the number of iterations through the PROCESS-TRANSFER states and if the number is less than the LOOP value of the tile, the state machine re-enters the PROCESS state. If the number of iterations does, in fact, match the LOOP value of the tile, the state machine moves into the final state.

In the WAIT state, the circuit monitors the signal being sent from the tile's RAM block indicating whether or not is full. Should the RAM be at maximum capacity, the circuit does exactly what the state's name implies and waits for the following tile in the network to finish utilizing the data in at least one of the sectors in the RAM. As soon as this "ramFull" signal is set low once again, the state machine re-enters the READY state.

At any moment, an input reset signal controlled by a module higher up in the architecture's hierarchy can send the control unit into the READY state, resetting all internal registers and signals in the process. Figure 37 is state machine diagram which is a simplified depiction of the control circuit's functionality.

Figure 37: Tile Control Finite State Machine Diagram



5.1.2.5 Tile Pipeline

Several tiles are cascaded to form the tile pipeline. This is a separate module in our design but doesn't serve much more of a purpose than acting as simple glue logic. The number of tiles instantiated in this module is determined by a parameter in the top level include file called NUM_TILES. This pipeline module ensures signals are connected correctly between the layers while breaking out critical signals that interact with entities such as the master controller to a top level interface. Furthermore, it serves as a crucial interconnect between a tile's control, data path, and RAM buffer, providing a structural description of the connections between these modules.

This circuit connects the output data bus of a previous tile's RAM buffer to the input of the following tile. The only exception is the first tile in the chain, whose input comes from a module higher up in the architecture's hierarchy. Also unique to the first tile is the breakout of the tile's idle signal to the top level. Doing this

allows the master controller to determine when it is okay to send data through the network, preventing corruption while maximizing throughput (the controller can start pumping data through the pipeline on the clock edge it sees the idle signal is high). Additionally, each tile's dataAvailable input is wired to the dataAvailable output of the RAM in the tile preceding it. The only obvious exception is the first tile, whose dataAvailable input is sent to the top of the hierarchy to be controlled by the same controller that monitors its idle signal.

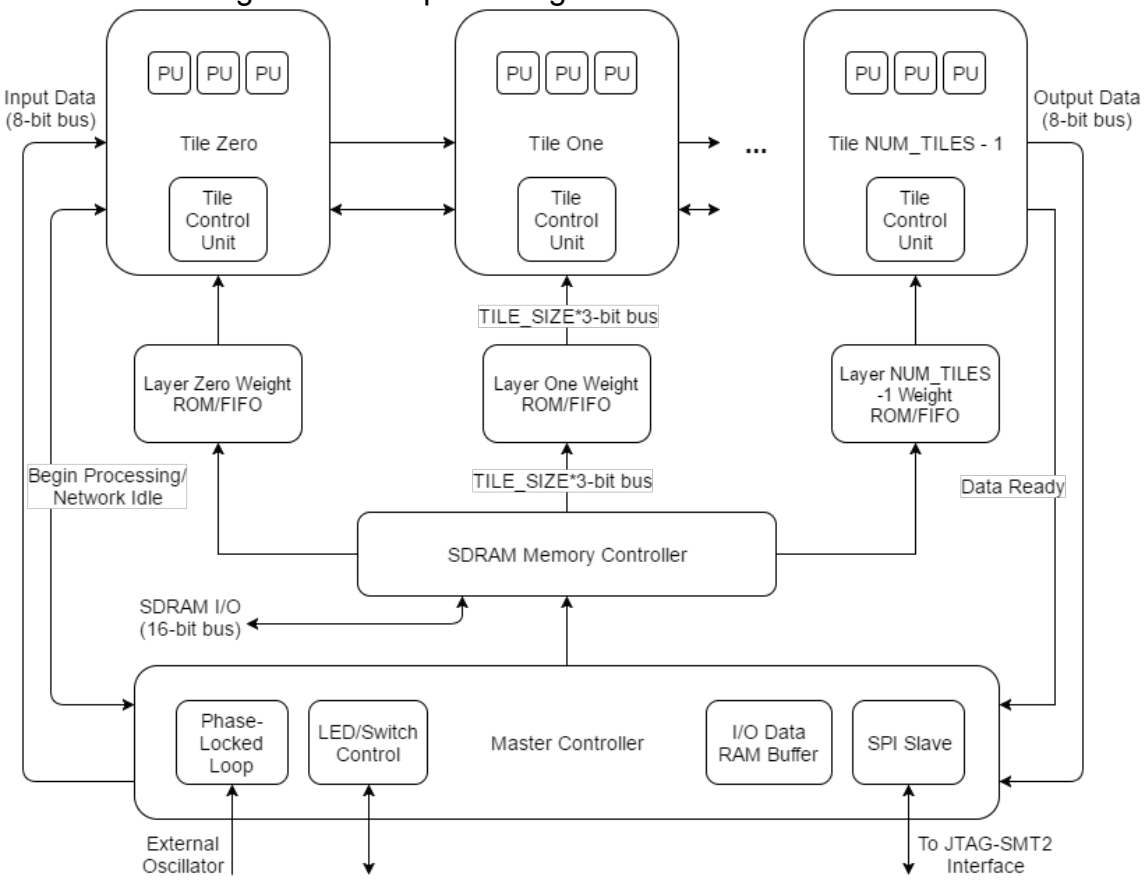
Each tile except for the last has its own pipelining RAM, whose functionality is given in the previous section. The last tile does not require this buffer because its output data is ready to be sent over the serial communication ports as soon as it becomes available.

It is in this pipelining module that the values such as LOOP and SIZE specific to a tile are used to generate the correct the number of PUs in addition to slightly tweaking the tile's control logic. This is done using module instance parameter assignment, ensuring our circuit's flexibility.

Modules higher up in the hierarchy only see the control signals such as ”tileIdle” and “networkDataReady” that instruct them when it is okay to push data through for classification and when the results of the network are ready to be extracted. In essence, this module becomes a black box that can be ported to any FPGA, so long as the correct weight distribution mechanisms are employed and a communication protocol is implemented that allows data to be transferred from and to the chip.

A high level overview of the circuit is visible in Figure 38. Some lesser control signals are not visible, but its general operation can be inferred and the structural connections between the component modules are accurate. Up until now, the master and memory controllers have not been described. In the subsequent sections their operation will be explained.

Figure 38: DeepGate Digital Architecture Overview



5.1.2.6 Weight Distribution

There exist two scenarios for the generation of our firmware. In the first case, the neural network is sufficiently small enough that the weights between the nodes can be stored using on-chip block RAM entirely. This allows us to forego using an external SDRAM chip and decreases the latency of the pipeline stages as weights do not need to be transferred from an external memory source before computation begins. Moreover, memory can be preloaded during FPGA programming in this scenario using generated .txt files. This means we wouldn't have to write extra software that transferred weights to the board post-programming. In the second case, the amount of memory necessary to contain our weights cannot be satisfied by the Spartan-6's block RAM, meaning an SDRAM chip needs to be used and additional logic added to implement a memory controller, load weights from the chip to an FPGA-side FIFO, and constrain the speed of the data path to the speed at which data can be transferred from the SDRAM. Ideally, our algorithm can be implemented within the bounds of scenario one. Since the size of the network is tentative, however, we will opt to build our design around an SDRAM chip. If it turns out that the network is too small to make use of the chip, then it can be used for storing input data and classification results (acting as a buffer between the PC and

the FPGA).

A block RAM segment is inferred for each tile that is capable of storing the weights that quantify the relationship between that tile and the layer before it. This block RAM, for scenario one above, has a size of PREV_TILE_SIZE · PREV_TILE_-LOOP · TILE_SIZE · TILE_LOOP · 3 (each weight is represented using 3 bits). It is easy to see that then that small increases in layer size can heavily influence block RAM usage. Assuming scenario one, however, block RAM usage is not a limiting factor in the circuit's performance. For both scenarios, weights need to be stored in a specific arrangement to ensure the correct flow of data through the pipeline. This arrangement is dependent on the SIZE and LOOP parameters of each tile as well as the way node outputs are computed. Essentially, because the outputs of SIZE number of nodes are calculated in parallel a LOOP number of times, weights must be stored in memory in the example arrangement depicted in Figure 39 below.

Figure 39: Example Weight Arrangement Scheme in Memory

Address Offset				
+0x03	+0x02	+0x01	+0x00	**Address**
$w_{1,4}$	$w_{1,3}$	$w_{1,2}$	$w_{1,1}$	0x00
$w_{2,4}$	$w_{2,3}$	$w_{2,2}$	$w_{2,1}$	0x04
$w_{3,4}$	$w_{3,3}$	$w_{3,2}$	$w_{3,1}$	0x08
$w_{1,8}$	$w_{1,7}$	$w_{1,6}$	$w_{1,5}$	0x0C
$w_{2,8}$	$w_{2,7}$	$w_{2,6}$	$w_{2,5}$	0x10
$w_{3,8}$	$w_{3,7}$	$w_{3,6}$	$w_{3,5}$	0x14

The figure above shows the weight memory arrangement for the second layer of a network with an input layer (first layer) size of 3 nodes and a second layer size of 8 nodes (TILE_SIZE = 4, TILE_LOOP = 2 for this layer). Subscripts are used to denote the weight between specific nodes. The weight between previous layer node i, and present layer node j is represented by wi,j. The table assumes that each memory address holds one weight (is 3 bits wide). During the PROCESS state, weights would be read from an address that is incremented by 4 every clock cycle for a number of clock cycles equal to the size of the input layer. This process repeats a TILE_LOOP number of times (when the control finite state machine moves out of the TRANSFER state). This scheme remains the same for every layer except the first. The RAM size is automatically modified by the synthesis tool to match each individual tile size; no manual editing of bus widths or array depths is needed.

In addition, this arrangement remains the same for either scenario. In the first case, the memory blocks would be read-only as they contain everything needed at the

start of the run-time. In the case that an SDRAM chip is needed, the memory would require a write port that allowed external logic to transfer weights from outside the FPGA.

The weight arrangement and distribution scheme becomes slightly more complicated when interfacing with the external RAM chip on our custom board. This is because the SDRAM's data bus width of 8 does not match up well with our 3-bit weights. In this case, the weight RAMs associated with each layer in the network must know the index of the MSB in the last weight to be loaded from external memory. For example, if a layer has 5 processing units, it needs a 15-bit weight bus and therfore 2 bytes read from external memory to load one address in the the layer's weight RAM. However, the MSB bit of the second byte is unused because the MSB of the last weight is located on bit 6, not bit 7. The weight RAM is given this information pre-synthesis and is capable of automatically discarding unused bits. Furthermore, the concept above is generalized to include weights of arbitrary widths.

Another special consideration occurs if the weight memory needs to be divided between on-chip and off-chip RAM. For example, consider a network configuration that requires 8190 B of total weight memory for one layer and can only spare 4095 B of on-chip RAM. In this scenario, there are 2 4095 B blocks of weights, one of which is loaded at a time on the FPGA. The SDRAM controller performs burst reads, meaning 4 bytes are read at once to improve read latency. In this case, since the size of the blocks falls short of a multiple of 4 bytes by 1 byte, 1 byte of padding needs to be added after each block. Different network configurations and memory divisions might require different amounts of padding bytes to be added to the weight memory.

All the scenarios above are pre-computed and dealt with by a C++ program that reads the pre-trained network weights from .csv files and transfers them to a compact binary file that can be easily uploaded to the FPGA. This program can be found on DeepGate's GitHub repository.

5.1.2.7 SPI Slave

The development board implements communication between the FPGA and on-board microcontroller using an SPI. In this configuration, the FPGA is the slave to avoid having the microcontroller constantly checking the to see if the FPGA has set the slave-select line.

The architecture of the slave is extremely basic. It utilizes an 8-bit shift register that replaces the least-significant bit every on every rising edge of the system clock (SCK), assuming that slave-select is set. The slave is hardcoded so that CPOL = 0 and CPHA = 0. That is, the clock polarity is set so that the base value of the clock is zero. A CPHA value of 0 configures the slave to read data on the rising edge of the clock and output data on the falling edge. There are many possible SPI implementations, some of which allow the user to configure the clock polarity and clock phase arbitrarily (even during run-time). For the sake of keeping our logic

utilization low we hard-code these values.

Currently we are achieving read/write speeds with the microcontroller of around 47 kB/s. By removing the microcontroller and communicating directly with the JTAG-SMT chip we plan to use, we can most likely increase the data rate by an order of magnitude. We aim to keep this slave communication setup for our final product.

5.1.2.8 Clock

Our design is largely composed of synchronous sequential logic, thus the need for a clock is non-negotiable. Since the Spartan-6 does not have a user-accessible internal oscillator, the clock signal must be sourced from a crystal oscillator external to the FPGA. This externally-generated clock arrives at an input pin and is routed to a phase-locked loop hard block that generates the system clock at a desired frequency. The generated clock is propagated throughout the die using special global clocking interconnects, limiting clock skew and increasing system performance. Xilinx provides the PLL logic and analog electronics directly on the silicon die, thus implementing Xilinx's PLL_ADV primitive in our design requires no additional logic resources.

A preliminary timing analysis shows that our Fmax falls somewhere around 112 MHz on a Spartan-6 FPGA with a speed grade of -2. Since our final design will make use of a -3 speed grade Spartan-6, we hope to increase the maximum frequency by at least half a dozen MHz.

5.1.3 Automatic Include File Generation

The size of our neural network is determined by several arrays of parameters in a Verilog include file. The values in these arrays determine the number of processing units and loop values of each tile as well as the bias value present in the nodes upon system start or reset. Furthermore, several arrays are present that are indexed to determine the size/loop values of tiles preceding and following the tile currently being synthesized. These extra arrays are used in place of the main size/loop arrays as Verilog throws errors if an array index is out of bounds (i.e. size/loop parameters don't exist for the layer before the input layer and the layer after the output layer). Finally, several parameters are needed to describe the memory architecture of the system. These indicate where in the SDRAM's memory range the weights for the different layers are located as well as the size of each division.

The include file and all the aforementioned parameters are all automatically generated by a C++ program that takes 3 arrays as input. One array describes the number of processing units in each layer, the next the loop value of each layer, and the last array includes the memory division values. The MEM_DIVIDE value determines how partitioned the weight memory for each layer is. For example, a MEM_DIVIDE value of 1 implies that all the weight values can be stored on-chip and a value of 2 splits the weight memory for a certain layer into 2 divisions, where only 1 division is on-chip at a time. MEM_DIVIDE must have a value such that it allows clean partitioning of the memory (you can't divide an odd number of bytes by

2). The C++ program takes these 3 arrays and automatically generates the include file as well as the weight memory file to be uploaded to the FPGA.

5.1.4 Physical Pinout

Signals in our design are routed to general purpose I/O pins that allow the FPGA to communicate with its environment. These pins are connected to specific components on our circuit board, therefore, ensuring that the correct internal signals are routed to the correct GPIO is a major part of the design process.

In ISE, this can be done using Xilinx's proprietary PlanAhead software that automatically generates the .ucf files used by the place and route process to determine the design's pinout. For all our signals, we use the LVTTL I/O standard and the "fast" slew rate designation to ensure that communication with the JTAG/SDRAM chips occurs at the fastest possible speed.

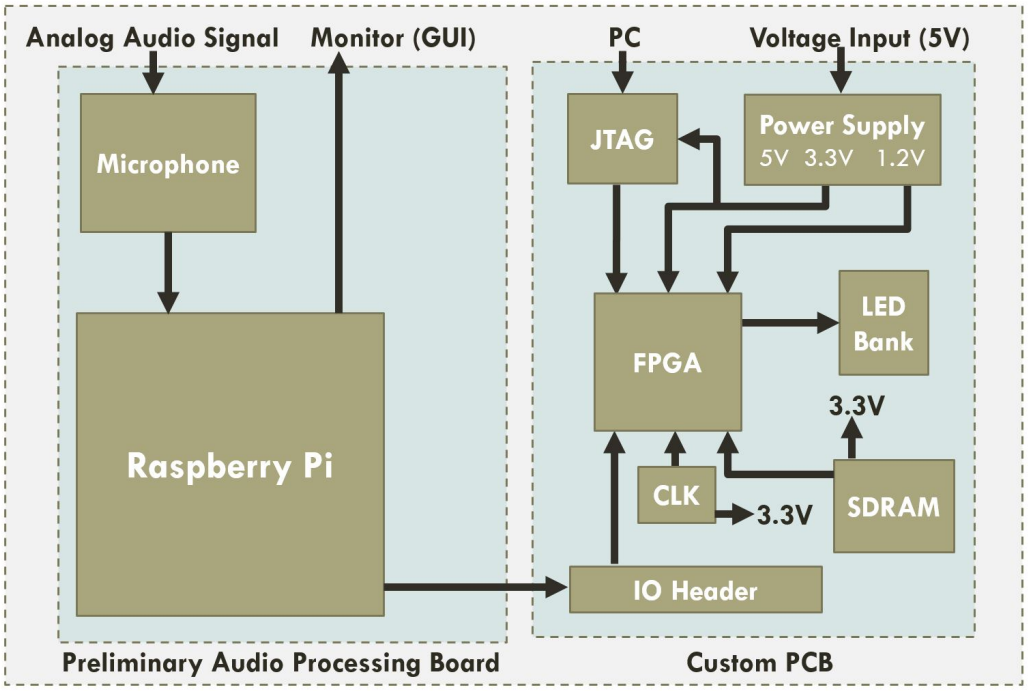
Certain internal signals must be routed to the SDRAM chips and on-board oscillator to ensure correct operation of the system. However, the large amount of LEDs, buttons, and header pins can be routed to any signal, allowing us to debug our system in multiple ways. For example, using the LEDs we can display the state of a finite state machine in our design to determine if it is getting stuck. The buttons can also serve as reset/start buttons. Header pins allow us to add additional signal debugging functionality and communication ports (SPI) to our design if for some reason the design is not working as intended.

5.2 Hardware

5.2.1 PCB Design Overview

There are always many decisions to be made when designing and laying out a PCB. In the beginning we knew we wanted to do a project that utilized FPGA technology. From there finding all the peripheral components, application components and power source components was next. After a list of required parts necessary for proper function was compiled a block diagram could be made to get an idea of the PCB components and connections. The block diagram of the PCB for our application is seen in Figure 40 below. It includes all of the necessary components, voltage levels and highlights the connections between specific components.

Figure 40: PCB Block Diagram



5.2.2 Schematics

The custom PCB schematic has been designed using EagleCAD.See Figures 41 through 44 different components of the schematic layout of the prototype PCB. The design is spread across two pages. The first page of the schematic is split up below in figures 41 and 42 on pages 80 and 81 respectively. The power connector, power LED, application LEDs and some capacitor banks are on Figure 41. The power voltage regulators and the remaining capacitor banks are in Figure 42. The second page of the schematics are below in Figures 43 and 44 on pages 82 and 83 respectively. The SDRAM and part of the FPGA are in Figure 43 of the schematics. The rest of the FPGA, JTAG and clock are on the schematics in Figure 44.

Figure 41: PCB Schematic

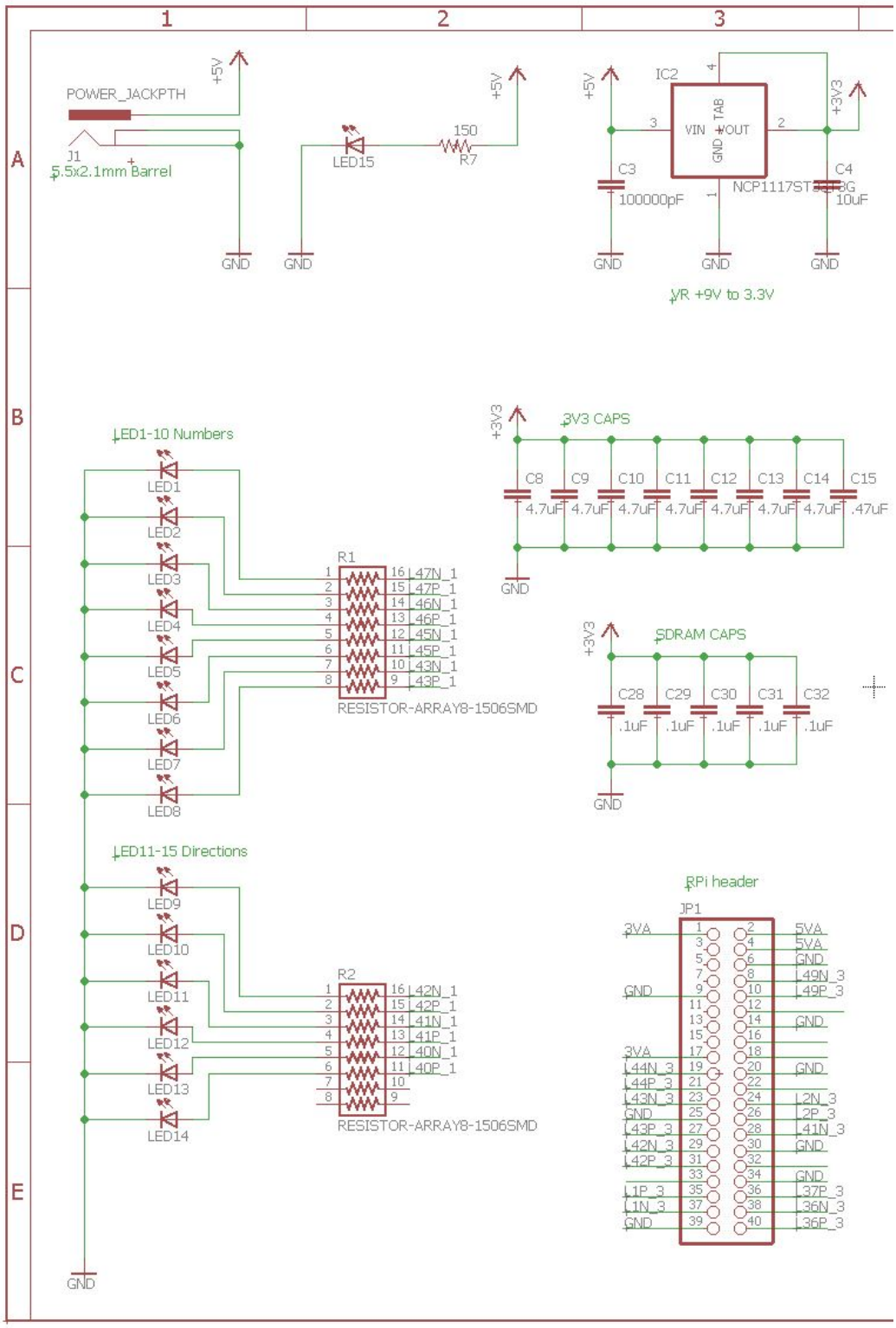


Figure 42: PCB Voltage Regulators

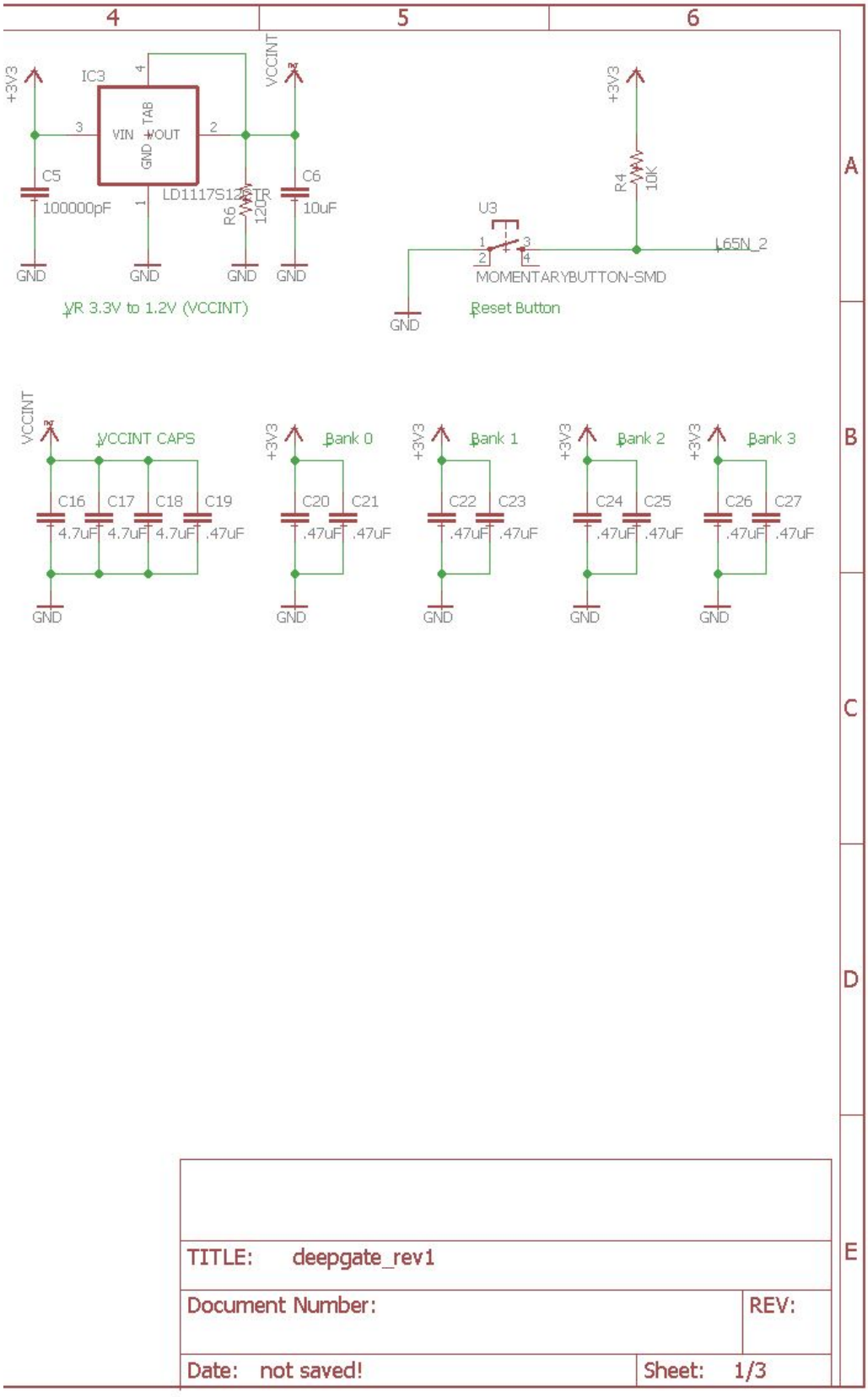


Figure 43: PCB Schematic Page 3

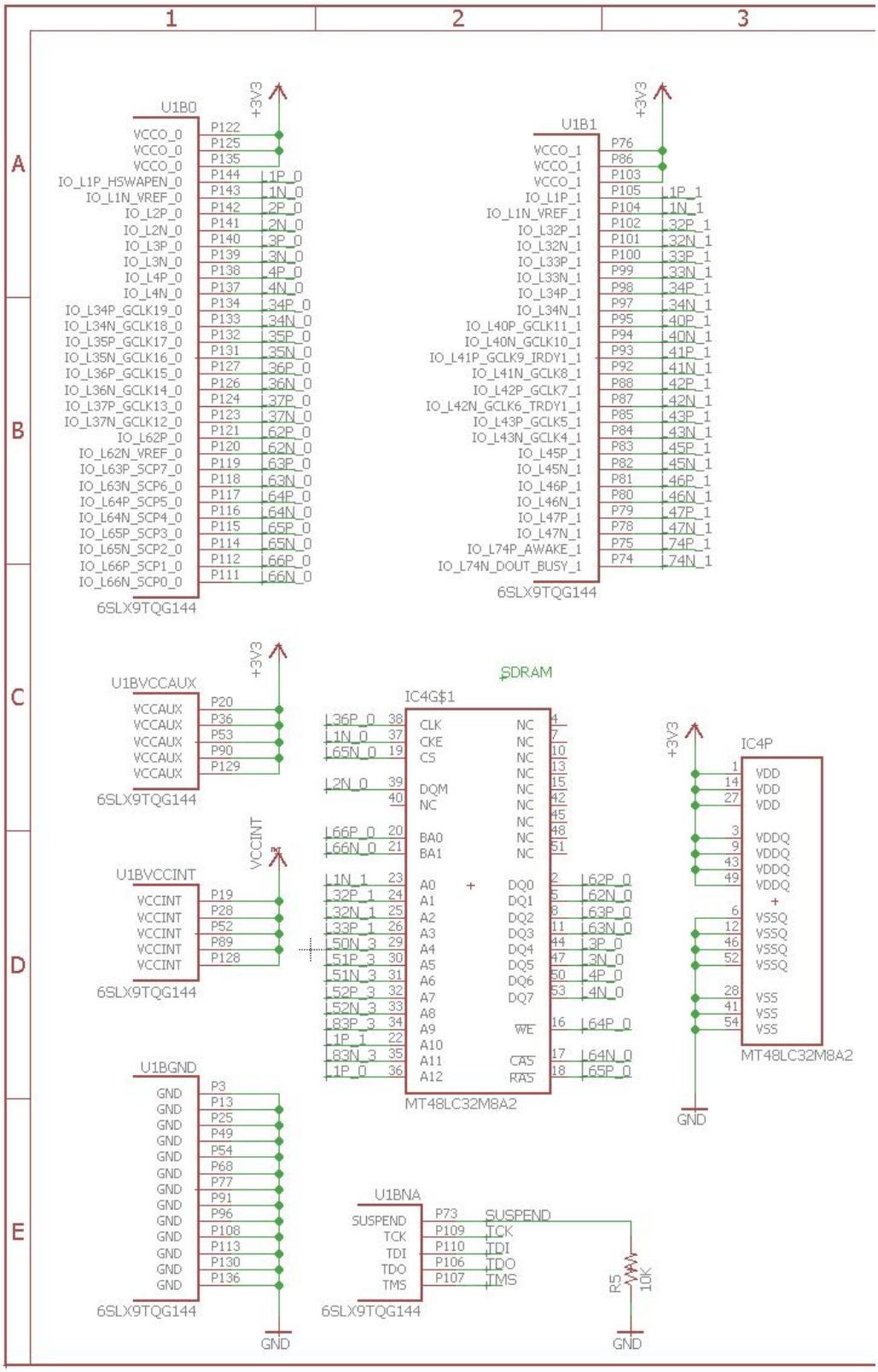
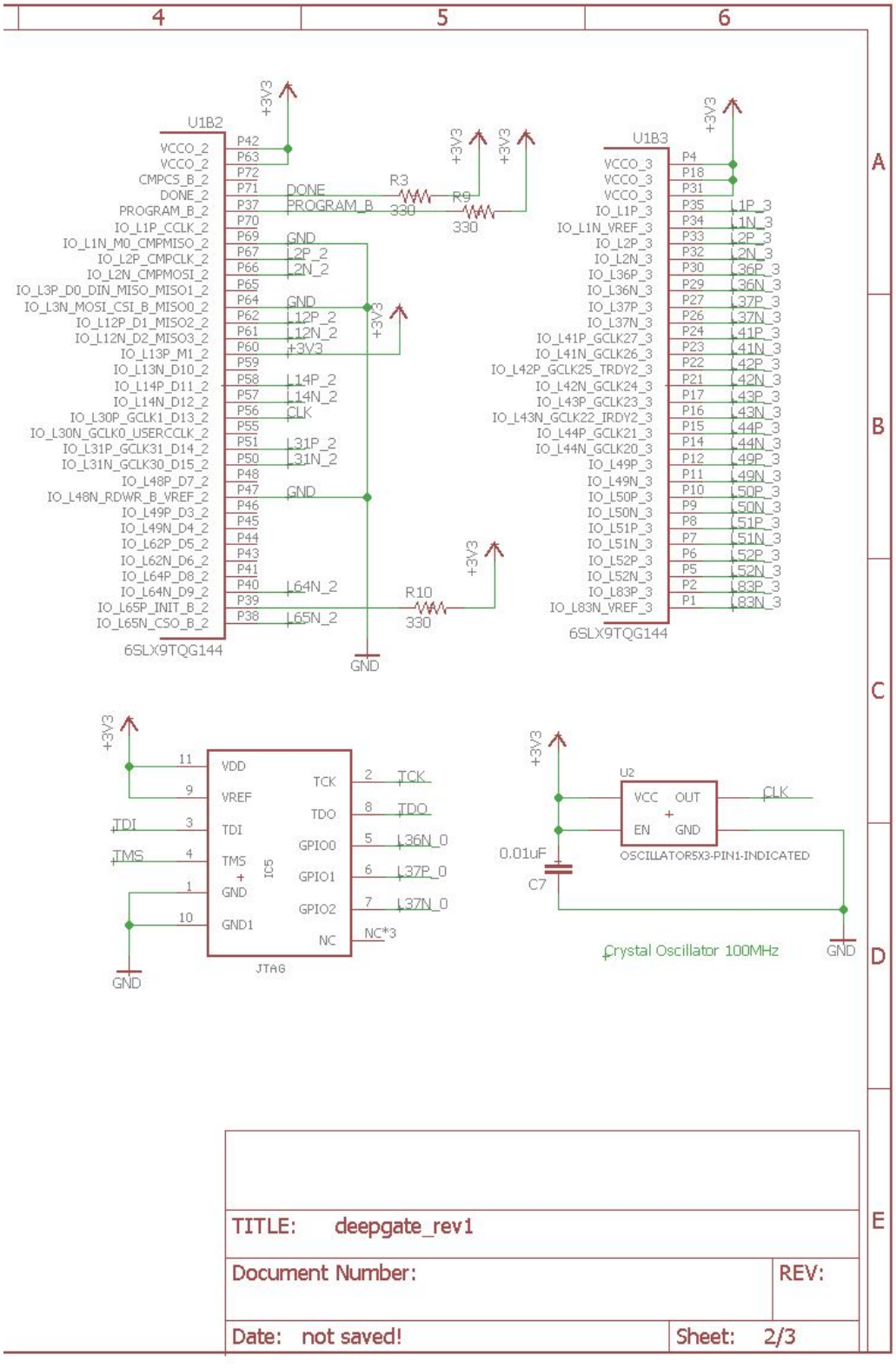


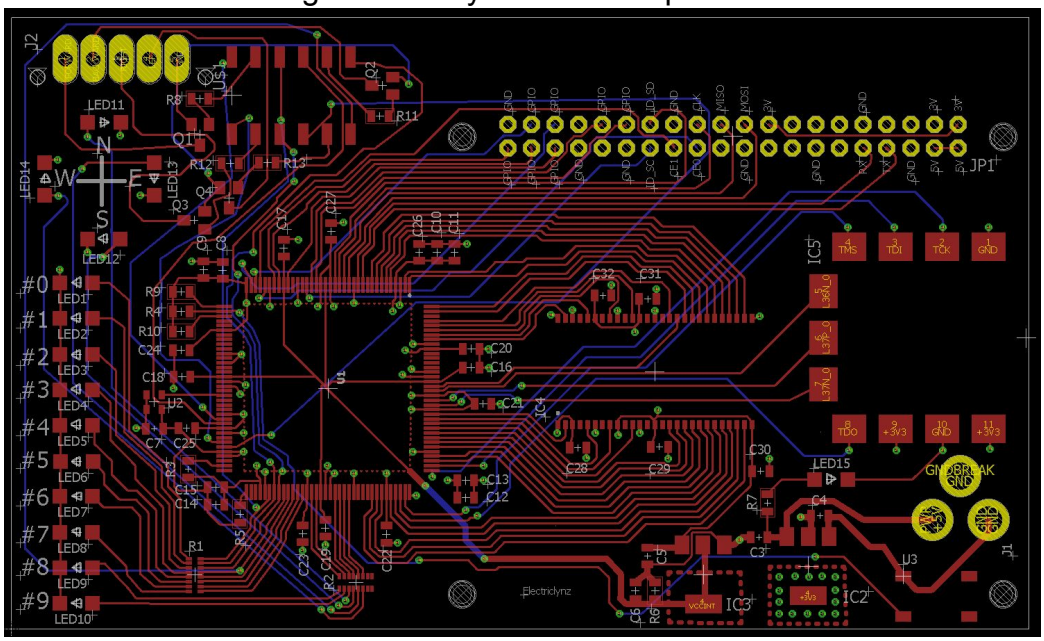
Figure 44: PCB Schematic Page 4



5.2.3 Board Layout

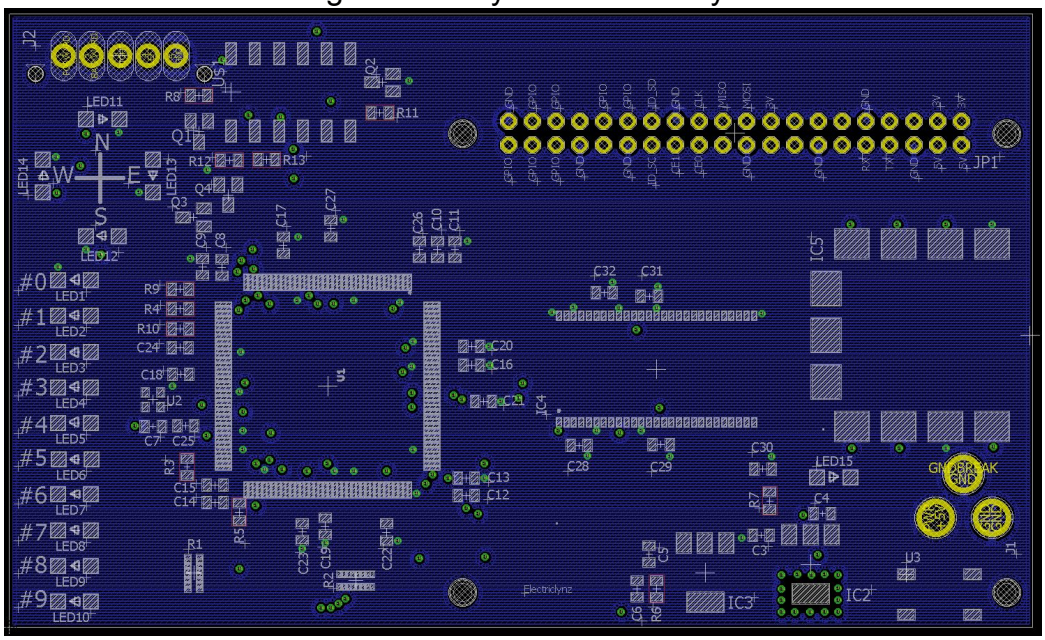
The layout of the board is just as important as the schematics. Laying out the board is essential to the components chosen working properly together. According to the data sheet for the FPGA and SDRAM a certain number, value and position of capacitors are needed connected to the supply voltages. These required capacitors are necessary for these two components so that the voltages are decoupled and not varying. The voltage regulators create heat when bringing the voltage levels down, there are heat sink pads created for the heat to dissipate across, instead of the heat affecting components nearby. Since this board has an FPGA a four layer board design was chosen. This enables signal to run across layer one and four so that two other layers can be reserved for power and ground. This allows vias to access a value on one out of 3 other layers. A via is a small hole that connects one layer to another though every layer on a board. The layout including all of the components and components names can be seen in Figure 45.

Figure 45: Layout and Components



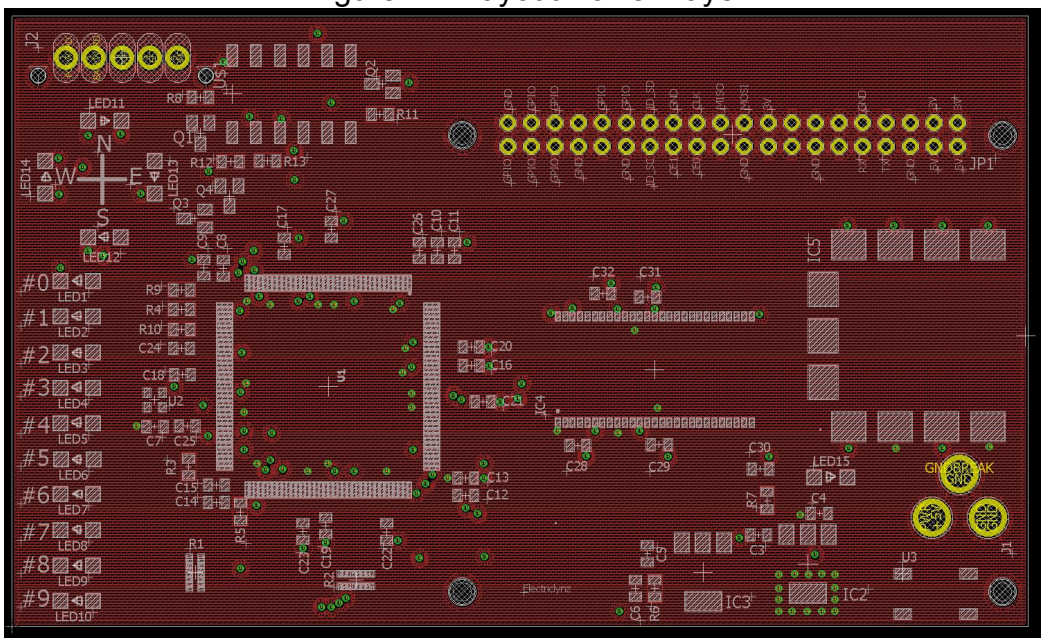
The ground layer being on layer two, seen in blue in Figure 46, allows vias to reach down a layer for ground instead of needing to cross the surface. This can be a problem with a crowded surface especially around the FPAG where the pins are really close together.

Figure 46: Layout Ground Layer



The power layer is on layer three from the top, this layer is in red on Figure 47 below. This allows the components on the top layer, layer one, to reach power easily. This allows less vias and a less crowded signal layers on the board.

Figure 47: Layout Power Layer

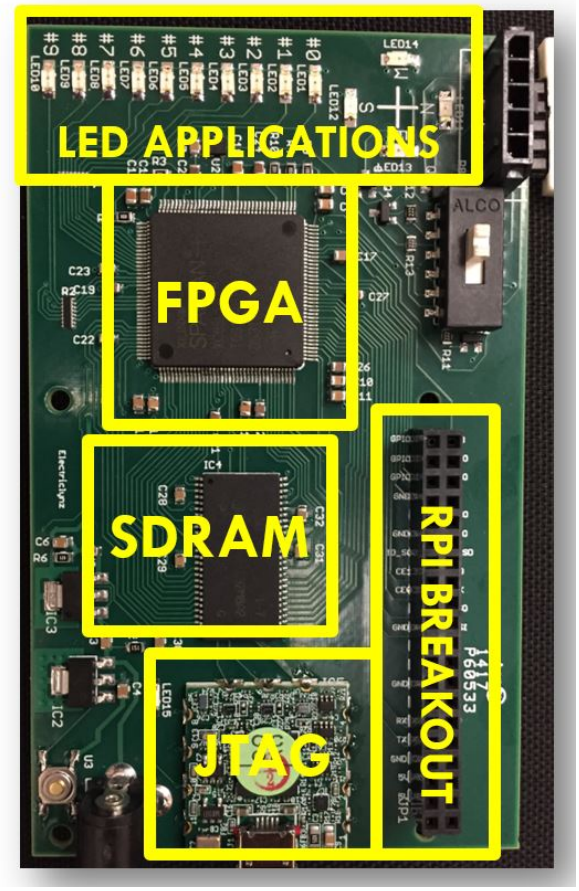


5.2.4 Components and Power

5.2.4.1 PCB Fabrication

A printed circuit board is made up of several layers. These details on layering are covered in Section 3.2.5.5 Layering on page 34. This section aims to expand further on how the board is eventually turned into a working piece of electronic equipment after all the layers are put together. As stated earlier the board is complex there are layers and ground and power planes within the board that are not on the top or bottom surface. With signal layers on the top and bottom layer of the board. Figure 48 below shows the board after the components have been assembled onto it. Most of the main components are labeled. The assembly of the board had to be done by a machine to ensure that there were no accidental connections made between the very close pins of the FPGA or SDRAM.

Figure 48: Fabricated Board Front



5.2.4.2 FPGA

An FPGA is a very complex integrated circuit. In Section 3.2.2 Field-Programmable Gate Array starting on page 15 and continuing until 21 the complex internal structure of the FPGA is explained. This includes explanations of logic blocks, hard

blocks and routing. FPGAs are faster than CPUs running a neural network. FPGA also conserves more power than a GPU.

Figure 49: FPGA Component Specification Comparisons for Part Selection

	Mounting Type	Package	Number of I/O	Price	Part Number
FPGA 1	Surface Mount	676-BBGA, FCBGA	400	$227.50	122-1869-ND
FPGA 2	Surface Mount	49-VFBGA	35	$4.19	220-1563-ND
FPGA 3	Surface Mount	144-LQFP	102	$18.13	122-1747-ND

From Table 49, above, there are similarities and differences between these FPGAs. Four our application we need a Lead Low Plastic Quad Flat Pack Package (LQFP) that way we can solder it ourselves if we need to. We have a budget of $1,000 for the entire project so we have to be consensus about the type of FPGA we get especially since we will be getting several backups. They are almost impossible to remove from the board once they are soldered on. So if anything goes wrong on any other part of the board and the board needs to be scrapped we cannot afford to spend more than $50 for a single FPGA chip The goal being to keep it under $20. We wanted to be able to get the most complex FPGA for the money. From Table 49, above, FPGA 1 has an impressive amount of I/O pins but for our application we don't need that many. This FPGA is expensive because of that, and also does not have a LQFP package for soldering. The price and package type takes FPGA 1 out of the FPGA possibilities for our project. FPGA 2 has only 35 I/O pins and it also does not have the LQFP package for mounting. Not meeting these two specification requirements take this FPGA out of contention as well. FPGA 3 is the best option out of the three for price and number of I/O pins. Even better, it has the LQPF mounting package.

In Section 3.2.2 Field-Programmable Gate Array starting on page 21 the Xilinx Spartan-6 family of FPGAs are explained in detail. We went with one of the smaller FPGAs that came in the LQFP package but we were able to select the fastest speed grade available from Xilinx FPGAs.

5.2.4.3 SDRAM

Section 3.2.4 SDRAM talks about basic explanation of RAM memory, its value to the project, how DRAM works and its difference from RAM. Also, how SDRAM works and its difference from DRAM. This section explains DRAM versus SRAM, technical features of SDRAM, and user interface information. SDRAM will be very beneficial to our project. It will hold memory storage, that is its sole purpose, which will aid in the speed of the project's execution. The FPGA will be able to access

the SDRAM for information which will be more efficient and there will be available storage,

Figure 50: SDRAM Component Specification Comparisons for Part Selection

	Memory Size	Package	Speed	Cost	Part Number
SDRAM 1	256M (16M x 16)	54-TFBGA	143MHz	$4.81	706-1464-6-ND
SDRAM 2	16M (1M x 16)	50-TSOP II	143MHz	$1.12	706-1446-ND
SDRAM 3	512M (64M x 8)	54-TSOP II	143MHz	$16.79	706-1420-ND

As seen in Table 50, above, there are three different SDRAM chips to choose from. SDRAM 1 has a fairly large memory size, which is desired, but its package type is the solder balls. We are avoiding solder ball package types because the gull wing mounting package is easier to handle. SDRAM 2 has a parallel interface package but it is too small to store a substantial amount of memory. We want the largest memory size available, which is SDRAM 3. It is also not that expensive so it is worth it to have more available memory for the FPGA to use.

5.2.4.4 JTAG

The definition of JTAG technology is found in Section 3.2.3.1 on page 22. This section also describes how the JTAG works and the four main logic signals that will be reviewed and expanded on in this section. Along with connectivity testing, the JTAG, supports boundary-scan architecture. Boundary-scan architecture allows different user defined instructions and are able to load the configuration data directly to the FPGA and connected memories. First, three different JTAG applications will be analyzed to determine the appropriate JTAG component for our application.

Figure 51: JTAG Component Specification Comparisons for Part Selection

	Maximum Speed	Support for Xilinx	Mounting Type	2-Wire JTAG	USB Connector
JTAG-SMT2	30 Mhz	Yes (Native ISE 14.1)	11 – pad SMT	Yes	Yes
JTAG-SMT2-NC	30 Mhz	Yes (Native ISE 14.1)	13-pad SMT	Yes	No
JTAG - USB	1.6 Mhz	Yes (Native ISE 13.2)	6 - pin	No	Yes

They all have a USB PC interface, SPI support, 4-wire JTAG and a Vref range of 1.8V to 5V. From Table 51, above, the comparisons show that the boards differ in the category of maximum speed, mounting type and 2-Wire JTAG application ability. JTAG-SMT2-NC is the same board as the one above it except it does not come with the USB connector hardware. This USB connector is essential to running the programming and tests of the FPGA from the PC. JTAG-USB is attached to a cable making it ineligible for our application.

With careful evaluation of the specifications, we have decided to use the Diligent JTAG-SMT2 as this surface-mount programming module best fits our current needs. This part is a fully self-contained and easily accessed through Xilinx tools. It requires a separate Vref supply for the JTAG signals and uses 3.3V power supply. It also has the advantage of using 24mA, three-state buffers that support fast bus speeds and a reasonable signal voltage range. It also utilizes a micro-AB USB connector. Permission Pending From Diligent for Reprinting.

Figure 52: JTAG-SMT2 Component Board Top

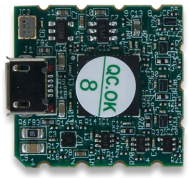
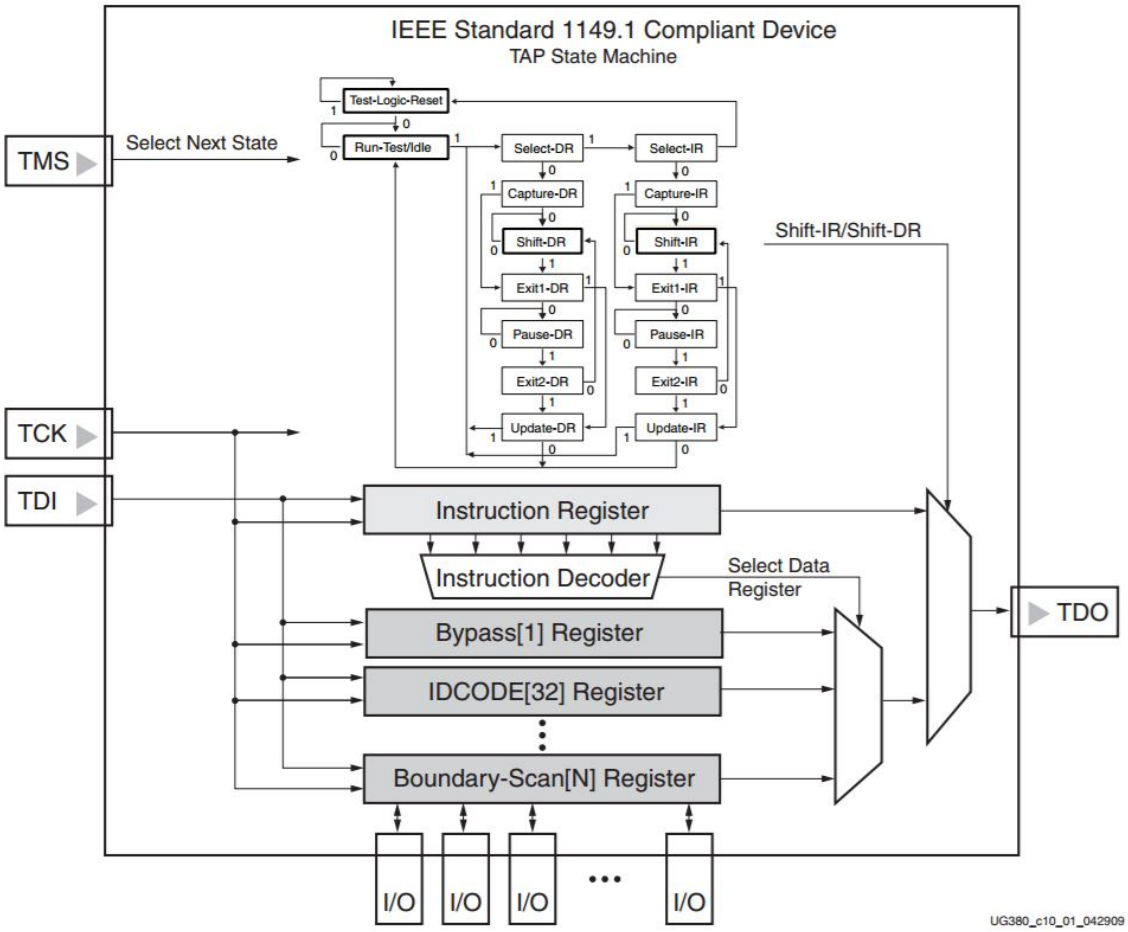


Figure 53: JTAG-SMT2 Component Board Bottom

This JTAG-SMT2, in Figure 52 and 53 above, sold by Diligent has several specifications and requirements for mounting it to a PCB. The requirements for the best results of the JTAG-SMT2 performance include mounting the JTAG near the edge of the PCB over a ground plane. Diligent recommends keeping the area beneath the JTAG clear. Even though traces can be run underneath, per the guide, for our design we will take all precautions and reroute them around the module. Another thing to keep in mind during the PCB design is to limit impedance between the JTAG-SMT2 and the FPGA below 100 Ohms to operate the JTAG at maximum speed, which is what we want. One of the reasons the JTAG was chosen over a microcontroller was because it will be able to achieve a faster data rate than the SPI serial communication that would be used between the microcontroller and FPGA. From Section 3.2.3.5SPI the data rate between the microcontroller and FPGA with SPI communication was 46.1 KB/s. The high-speed USB2 port can drive the JTAG bus between the JTAG and the FPGA to a data rate of up to 30MB/s. That is more than 650 times the data rate between the microcontroller and the FPGA.

The FPGA architecture includes all the elements required to work with the JTAG. These elements include the TAP, TAP controller, the Instruction register, the instruction decoder, the boundary-scan register and the BYPASS register. The identification register is supported for 32-Bits. All of this architecture within the FPGA complies with the IEEE Std1149.1. Before discussing the four main logic signals it is important to explain the Test Access Port (TAP) first. These four main logic signals are three input pins and one output pin and they control the boundary-scan TAP controller. The TAP controller is a state machine with 16 different states. All of the 4 input and output signals control how the data moves through the TAP state machine. In Figure 54 below, the diagram of 16-state TAP finite state machine's transitions can be seen. A transition between the states only takes place when the TCK is on its rising edge value. The two columns of seven states each represent two different types of paths. The column on the left is the Datapath and the data registers operate in the states which names end in "DR." The column on the right is the Instruction Path and their registers operate in the states whose names end in "IR."

Figure 54: TAP Controller's Finite 16-State Machine Transitions - Permission Pending From Diligent for Picture Use



As mentioned earlier the four main logic signals used in the JTAG bus are:

1. Test Mode Select (TMS)
2. Test Data In (TDI)
3. Test Data Out (TDO)
4. Test Clock (TCK)

Since only one data line for each of these signals is available the protocol for the transfer of data is serial. The TMS pin controls the operation of the test logic. The state or value of the TMS pin on the rising edge of the clock governs the sequence of state transitions. The TMS pin has an internal pull up resistor so that when there is no input the input stays high. The TDI pin receives input data in the form of serial protocol. This serial data is either sent to the test data registers or instruction

register. This is controlled by the state of the TAP controller. The TDI pin also has an internal pull-up resistor so that the input is high unless there is an outside input. The TDO pin outputs the data in serial form from one of the registers, either the test data register or the instruction register. The decision between the two is made by the state of the TAP controller. The TDO pin also has a high-impedance, which is something to take into account when adding the JTAG to the PCB.

Lastly, the TCK is not actually a clock, it is used to load the test mode data from the TMS and TMI pins. This only happens when the TCK is on the rising edge. During the falling edge of the TCK the test clock outputs the test data on the TDO pin. All of these reasons explain why it is a great choice for interfacing the programming of the FPGA and that is why we selected the JTAG-SMT2 for our PCB board.

5.2.4.5 LED

There is an abundance of Light-Emitting Diodes (LEDs) to chose from with different specifications used in various applications. There are several different types of LED packages, two of them are applicable to our project, Dual In-Line Package (DIP) and Surface Mount Device (SMD). For SMD there are different sizes to choose from. Also for each mounting type there is a Millicandela Rating.

There is an abundance of Light-Emitting Diodes (LEDs) to choose from with different specifications used in various applications. There are several different types of LED packages, two of them are applicable to our project, Dual In-Line Package (DIP) and Surface Mount Device (SMD). For SMD there are different size pad specifications to choose from. Also for each led types there is a Millicandela Rating.

The two LED mounting types that would meet our specifications are both DIP or through-hole mount and SMD or surface mount. The DIP LED has two leads coming off of the epoxy encasing. The SMD is flat and has connections on both ends that match up to contact pads on the PCB. As seen in the figures, the SMD LED is considerably smaller and more compact. The size difference along with not being a through-hole mounted part makes it a favorable choice of LED. The benefit of the SMD LED in regards to the way its mounted saves considerably more space, costs less to create drilled contact holes on a PCB and is quicker to install from a manufacturing point of view. The SMD LED can be installed with a reel and a pick and place machine. These machines can install hundreds of small surface mount parts in minutes. SMD LEDs seem to be better in every way, except for if the board may be under mechanical stress. The DIP LEDs have through-hole connections which make them more secure because they have to be soldered from both sides of the board.

The size specifications for the SMD LEDs are related to the dimensions of the part. This is applicable for all SMD parts not just the LEDs. The size of the package is indicated by a numerical code. For example, 0805, this is an Imperial code, which represents a length of .08” and a width of .05”. The imperial code used to be used the most often to specify a SMD size but in modern PCB design metric units (mm) are used. For our project we will be selecting LEDs based off of the

imperial code. There are several sizes of common SMD LEDs to use. Initially we chose 0805 which would work well. In case we decide the brightness of the LEDs are not sufficient in that size package an increase in size would allow for more luminous LED options. The increase in package size allows a larger range of LED brightness to select from, the only downfall is that they have an increased forward voltage, which requires a sufficient voltage supply to the LED.

The brightness of the LED can be controlled by the current passing through the LED. The current through the LED is controlled by the supply voltage value from the FPGA and the value of the current limiting resistor. The current limiting resistor is very important in keeping the LED from exceeding the maximum recommended current draw and to limit the current draw from the FPGA pin.The application LEDs in the design will have a supply voltage of 3.3V from FPGA I/O pins. Will be using different colored LEDs and the forward voltage of different colors changes with the millicandela rating. Only certain colors were available below 3V. The colors of LEDs to be used will be blue, green, red and white. See Resistors section below for an expanded definition of forward voltage.

The design will have four different color LEDs and two of the colors will be in different sizes. The reasoning for the different colors is to use a certain color for a certain function of the PCB. The different sizes were not necessary but was desired. The smaller indicator LEDs that are not a part of the applications of the project should not be the same size so that separation of function and application is easily visible. After these preferences were selected the next step was narrowing the LED selection even further.

There were thousands of LEDs to choose from with different qualities such as size, luminosity, forward voltage and color. Table 55 below shows three different viable LEDs for this project.

Figure 55: LED Component Specification Comparisons for Part Selection

	Color	Package Size	Millicandela Rating	Distributor	Part Number
LED 1	Green	0603	10mcd	Digikey	160-1475-1-ND
LED 2	White	2-SMD R-Ang	2100mcd	Digikey	160-1870-1-ND
LED 3	Green	0603	65mcd	Digikey	160-1834-1-ND

As seen in Table 55, all of the LEDs are very different from each other. The LED 1 was one of the colors and sizes desired but it was not bright enough to be able to see clearly from a couple of feet away. LED 2 was also a color desired but the mounting type was SMD right angle and that did not align with the mounting type selected. LED 3 was a perfect option. It was the preferred package size, luminosity and color.

For the project LED 3 was selected as an indicator LED. It will be used to establish

the board is turned on by being set to steady on when the board is on. It will also be used to indicate the reset switch has been pressed and will light up only when the button is pressed. The other LEDs selected include blue and white LEDs, with package size 1206, for the application part of the PCB. The white LEDs will light up for the numbers application and the blue LEDs will light up with the cardinal directions application. There will also be extra LEDs available for troubleshooting and other possible purposes.

5.2.4.6 RESET Button

A pull up resistor is very useful in order to assign specific voltages to pins when they aren't being utilized. This keeps the value from just being empty or floating signal which could cause the program to register it incorrectly. Pull up resistors are utilized on the JTAG-SMT2 GPIO pins to keep the signal on those pins from floating when they aren't active. A pull up resistor is also used for correct function of a reset switch. A reset switch will be included in our design in order to manually reset the FPGA. The FPGA registers the reset pin as an active high pin. This means that when the pin receives logic high input (1) = "on" and when logic low input (0) = "off". This is necessary to figure out because it is the reason for having a pull up resistor.

Figure 56: Reset Button Open with Pull-Up Resistor

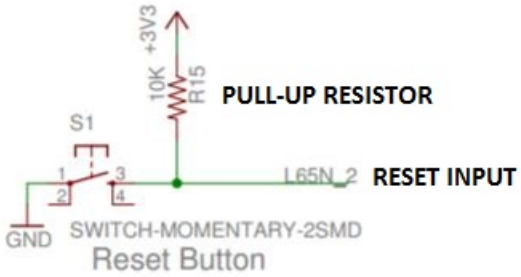


Figure 56 shows the pull-up resistor, FPGA reset pin and the momentary reset button sharing a node before the circuit goes to the ground. Below is the evaluation of both states of the button.

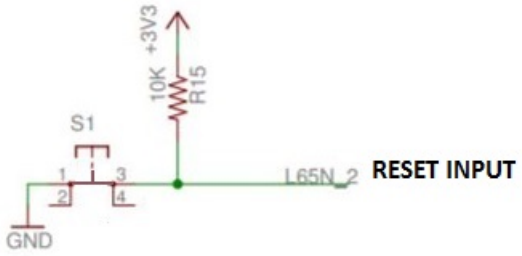
No Reset:

Figure 57: Reset Switch Open Circuit

In Figure 57 switch is open so there is no current running to ground which must mean that there is no current flowing through the resistor. Since there is no current flowing through the resistor there is no voltage drop across it. Which in turn sends the whole voltage supply of 3.3V to the reset pin on the FPGA. Since the pin is active high, mentioned earlier, the FPGA interprets this value as being “on” and to continue its processes. The high value keeps the FPGA from being reset. It is counter-intuitive to any regular switch, like a light switch, when it is not being pressed no voltage is being supplied to the circuit. This is the opposite, when the button is not pressed the FPGA pin is being supplied a voltage.

Reset:

Figure 58: Reset Switch Closed Circuit



In Figure 58 when the switch is closed, a path for the current is created to the ground. Since there is now a ground, current flows from the voltage supply through the resistor, through the switch and to ground. This is the path of least resistance. Current will always take the shortest path it can to reach ground. For this reason, no current goes into the FPGA pin. Due to the lack of current to the FPGA pin no voltage drops across the line and then there is no voltage into the FPGA reset pin. Therefore, the FPGA reset pin receives no voltage or current input and interprets this value as being “off”. After the FPGA reads an “off” from the pin it successfully resets the FPGA.

Figure 59: Reset Button Component Specification Comparisons for Part Selection

	Contact Rating @ Voltage	Mounting Type	Termination Style	Part Number
Button 1	0.05 @ 24VDC	Through hole	PC Pin	450-1650-ND
Button 2	0.02A @ 15VDC	Surface	J Lead	P14168CT-ND
Button 3	0.05A @ 12VDC	Surface	Gull Wing	EG2531CT-ND

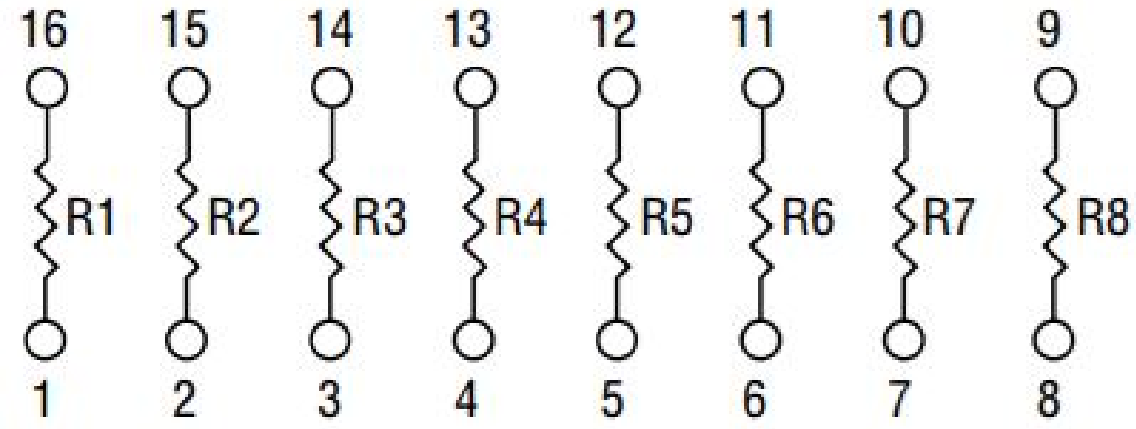
Table 59, above compares three different tactile switches, from distributor Digikey, that act like momentary pushbutton switches. Momentary means that the switch only creates a contact while it is being pressed, which is represented as Off-Mom

as the switch function type when searching Digikey. This Off-Mom shorthand name stands for, Off continuous – On momentary. Any switches that are anything other than Off-Mom connection type cannot be considered for the reset button application. All the buttons analyzed above are Off-Mon buttons which is the only thing they have in common. Button 1 is a through hole button which is unnecessary, more expensive and takes up more room so it is not the right choice. Button 2 is a surface mount type, which is desired, but it has a J Lead termination style. This means that the leads off of the SMD part turn up, and this is not the style desired. Button 3 is a surface mount part, has gull wing termination style leads and is small and compact in size. The gull wing termination style looks like little support feet off of the SMD part that will match up perfectly with the pads on the PCB.

5.2.4.7 Resistor Network

The LEDs in the PCB need current limiting resistors. Some of the LEDs are for the applications of the PCB, twenty, and then some are extra for assisting in troubleshooting any issues. Since there are so many LEDs it would save space and be more efficient to select a resistor network. It would also save time during assembly when inserting a resistor network instead of one resistor at a time. A resistor network is single component that is made up of a combination of several resistors. There are three main different types of resistor networks, isolated resistors, bussed resistors and dual terminator. For our application, as current limiting resistors, isolated resistor networks are chosen. This means the resistors are independent of one another, as seen in Figure 61 below, and are not connected at any point within the resistor network component.

Figure 60: Resistor Network Internal Schematic



There are many different characteristics of resistor networks besides the arrangement of the resistors within the network. The different characteristics include number of elements, resistance, tolerance, maximum working voltage, power dissipation, and size code. All of these characteristics will be specified before selecting

a resistor network, which will narrow down the possible options to a reasonable number to choose from.

The number of elements within a resistor network can range from 2 to more than 30. For this project a resistor network with 8 elements or resistors was chosen. This choice was made because there are 24 LEDs within the application section of the PCB that are all related in conveying the success of the project. These LEDs will all be connected to three 8 element resistor networks. In case one of the resistor networks breaks, there are still two other working resistor networks that can be used for the application. For the application, 14 LEDs are required to work, so if one of the resistor networks fail, there are still two others working with 16 LEDs. We decided having one failsafe was enough and therefor selected an eight element resistor network. Also, eight element resistor networks are more common than seven element resistor networks which would allow for more options to choose from and 10 element resistor networks was more than what was needed.

The resistance of the resistor networks depends on the maximum supply current into the LEDs and the required forward voltage for the LEDs to turn on. The formula for finding the required resistance is Ohm's Law:

$$V = IR$$

To use the formula for the application LEDs selected the forward voltage needs to be known. In Figure 72 on page 130 the Bill of Materials, the blue and white LEDs description lists the maximum current and forward voltage specifications. The blue and white LEDs are the chosen application LEDs and they both require a maximum current of 20mA for maximum brightness. The maximum current level is the same for both making one aspect of choosing the required resistance simpler. The blue LEDs require a forward voltage of 3.3V. The white LEDs require a forward voltage of 3.2V. Since forward voltage required for the LEDs are similar but not exactly the same some decisions need to be made before calculating the resistance required to limit the current. The forward voltage is similar which means the same resistor value can be used for both. In order to select a resistor value to work for both a single forward voltage value must be chosen for the calculations. The value of 3.2V will be used for the calculations. The KVL equations were explained in previous sections. The selected voltage supply is 5V and 3.2V drops across the LED, then 1.8V needs to drop across the resistor. Since the maximum current is set at 20mA, ohms law, seen above, can be used to find the value of the resistor.

$$\begin{aligned} R &= V/I \\ &= 1.800V/0.020A \\ &= 90\Omega \end{aligned}$$

The resistor value is found to be 90 Ohm, which is not an available value. From the available choices 100 Ohms was selected. The detailed reasoning for round-

ing up to a higher resistor value is found in the previously mentioned section. To summarize, increasing the resistor value simply decreases the current value due to resistance and current being inversely proportional. This is preferred because the maximum current allowable through the LEDs is 20mA, and a larger resistor value would decrease the current passing through the LEDs.

$$\begin{aligned} I &= V/R \\ &= 1.800V/100\Omega \\ &= 0.018A \end{aligned}$$

From equations above the current through the white LED is 18mA which is an appropriate value. The current limiting reasoning is why 3.2V was selected for the calculations of the resistor value for both LED colors. When 5V is supplied to the blue LED with a forward voltage of 3.3V, 3.3V will drop across the LED, which leaves 1.7V to be dropped across the resistor.

Using Ohm's Law, equation the resulting resistance value is 85 ohms in order to limit the current passing thought the resistor and blue LEDs to 20mA. The 100 Ohm resistor value is a good choice for the 3.3V forward voltage LED for the same reason it was for the 3.2V forward voltage LED calculations. The only difference is the current will be limited more because the available voltage drop across the resistor is lower. Using Ohm's Law, the current through the blue LED can be found.

$$I = V/R$$

$$I = 1.7V/100\Omega$$

$$I = 17mA$$

According to the equations above, the current through the resistor and blue LED is limited to 17mA. This decrease in current would only effect the brightness of the blue LED by a slight amount and is acceptable for use.

The tolerance level is the percent error of the resistor value. For example, a resistor of 100 ohms with a tolerance of +- 5% means the resistor could actually have a value in the range of 95 to 105 ohms. The tolerance of the resistor network does not need to be hyper specific. A tolerance of +- 5% or lower is adequate for the chosen resistor value of 100 ohms. Since the resistor value could range from 95 or 105 ohm the current would be limited to a range of 18.94 mA to 17.14 mA for the 3.2V white LEDs and a current range of 17.89mA to 16.19mA for the 3.3V blue LEDs. The ranges of current values are adequate to light the LEDs sufficiently.

The power dissipation per element is another specification of a resistor network. To select the minimum power dissipation allowable, the power dissipation must first be

found using the equation below. Since the power dissipation is a maximum value, the maximum power dissipation must be found with our selected resistor value and corresponding current ranges. To find the maximum power dissipation, Pmax, maximum values of voltage and current through the resistor will be used. Vmax is the maximum voltage drop across one of the resistor elements. Found above, the maximum voltage drop, Vmax, is 1.8V when the resistor element is paired with the 3.2V white LED. The maximum current limit, Imax, from this 1.8V voltage drop across the resistor element is 18.94 mA, which was calculated in

$$\begin{aligned} P_{max} &= V_{max} \cdot I_{max} \\ &= 1.800V \cdot 18.940mA \\ &= 34.090mW \end{aligned}$$

The maximum power dissipation was found to be 34.09mW. Therefore, the power dissipation value specification can be any value above 60 mW. The reason for choosing 60mW even though the calculated maximum was 34.09mW is to give an adequate buffer, double, between the calculated maximum and the specified maximum. This is precautionary just to absolutely prevent any damage from surpassing the power dissipation specification. The size code specification is not something that is needed to be determined before looking at available parts. The reason being is that there will be more available options without being specific. Although one thing can be specified about the part itself that would go hand in hand with the size, which is the mounting type. Our goal is for our designed PCB to be as small as possible and as cheap as possible. Choosing a surface mount resistor array allows the part to be smaller and cheaper than creating through holes on the board. Having specified a surface mount requirement is another step in finding the desired resistor network. After all these requirements desired have been specified choosing the resistor network was the next step. In Table 61 below, three different resistors found on distributor, Digikey's website, are compared and one of them was ultimately selected.

Figure 61: Resistor Network Component Specification Comparisons for Part Selection

	Resistor Connection	Number of Elements	Resistance Value (Ohm)	Tolerance	Power Dissipation (mW)	Mounting Type	Part Number
R1	Isolated	8	1,000	+-2%	250	Through Hole	4116R-1-102LF-ND
R2	Bussed	8	10,000	+-2%	200	Through Hole	4609X-101-103LF-ND
R3	Isolated	8	100	+-2%	160	SMD	4816P-T01-101LF-ND

These three resistor networks, Figure 61, all have eight elements within them and the same tolerance but they are also very different from each other. Even though R1 has a power dissipation per element level higher than what we need, it is a through hole mounted part and has a resistance value of 1,000 ohms which are not within the specifications desired. The R2 option is completely different from the specifications, most importantly it is a bussed resistor network. Which means the resistors are connected to one another, unlike the isolated resistor network. Like previously mentioned, in an isolated resistor network the resistors within are not wired to each other in any way and this is what we specified. This is why R3 is the perfect choice. It meets all of the specifications made for a resistor network that would function as desired within the circuit.

5.2.4.7.1 Current Limiting Resistors

Forward voltage is the voltage required across the diode to turn it on. For example, the first set of LEDs we will order have a forward voltage of 3.3V. The chosen supply voltage is set to 5V. To ensure the brightness of the LED the current available for the LED to draw needs to be a maximum of 20mA. This maximum current draw is found on the LED datasheet. Now all that is left is to determine the current limiting resistor's value. If we want the LED to be as bright as possible then it is simple. Ohm's Law and Kirchhoff voltage law (KVL) will be used to determine and compare the value. First use KVL starting with the voltage source and ending at the ground. given values: Supply Voltage = 5V, V_f = 3.3V and I_Dmax = 20mA.

$$-V_s + V_f + (R \times I_D max) = 0 \tag{29}$$

$$-5V + 3.3V + (R \times 20mA) = 0$$

$$R = 5V – 3.3V \div 20mA$$

$$R = 85\Omega$$

Since this is not a real world manufactured resistor value I will round up to a larger resistor. The resistor selected has a value of 100 ohms. The reason I am rounding up to a larger resistor is because that will limit the current even more, the larger the resistor the smaller the current will become. This is because the current limiting resistor is causing the circuit to draw less current. This can be seen in the equations below, using Ohm's Law, the first with a small resistance and the second with a large resistance in place.

Ohm's Law V = voltage (Volts, V) I = current (Ampere, A) R = resistance (Ohm, Ω)

$$V = IR \tag{30}$$

V= IR rearrange the terms I=V/R

Example 1: R = 100 Ω

$$I = (5V – 3.3V) \div R$$
$$I = 1.7V \div 100\Omega$$
$$I = 17mA$$

Example 2: R = 250 Ω

$$I = (5V – 3.3V) \div R$$
$$I = 1.7V \div 200\Omega$$
$$I = 8.5mA$$

As seen from the examples the current in Example 1 is 17mA, and in Example 2 the current is 8.5mA which is half of the current from Example 1. The resistor in example 2 was twice as large and the resulting current from example 2 was half as large as the values from example 1. So we have proved with Ohm's law that current and resistance are inversely proportional to each other. Which means if one is increased the other decreases.

5.2.4.8 Power Supply

To supply the correct power supply to the PCB each component's maximum voltage needs to be noted. Whatever the highest level of voltage required is the minimum amount that will need to be supplied to the PCB. From Table 62 the maximum required voltage was found to be 3.3 V.

Figure 62: Component Supply Voltage and Maximum Current Specifications

Component	**Voltage Requirements (V)**	**Max Current (V)**
FPGA	V_{CCINT} = 1.2, V_{CCAUX} = 3.3, V_{CCO} = 3.3	I_{FS}=40mA, I_{INmax}=10mA I/O
JTAG	V_{DD}=3.3, V_{REF} = 3.3	I=20mA
SDRAM	V_{DD} = 3.3	I=135mA
Total	V_{max}=3.3V	I=205mA

There are a couple of different ways to supply voltage to the PCB, namely, AC to DC power from a wall outlet and batteries. We decided to choose AC/DC power from the outlet for a couple of reasons. First, having a steady stream of power is extremely reliable, we won't have to worry about replacing batteries or recharging batteries. Also, we don't want to deal with all the debugging and prototyping issues that could arise when we don't realize the batteries are dead. There are many options for connecting the power to the board. There are cylindrical connectors, snap and lock DC power connectors, Molex connector, Tamiya connectors, JST RCY connector, and much more. A cylindrical connector or barrel jack was chosen for our design due to its small size, high voltage and current limits. Even though the board doesn't need more supply voltage than that the maximum voltage the lowest supply voltage available from an AC/DC barrel jack power supply is 12VDC. One requirement would be the minimum output voltage requirement of the power supply, which is 5V. Another requirement of the power supply is to have a maximum

current draw of at least twice as large as the calculated maximum current drawn from the PCB components. This requires the power supply to have at least 500mA maximum current draw. The power barrel connector jack, which will be soldered onto the PCB and receive the barrel connector from the power supply will need to meet the same requirements above.

Figure 63: Power Supply Adapter Specification Comparisons for Part Selection

	Memory Size	Package	Speed	Cost	Part Number
SDRAM 1	256M (16M x 16)	54-TFBGA	143MHz	$4.81	706-1464-6-ND
SDRAM 2	16M (1M x 16)	50-TSOP II	143MHz	$1.12	706-1446-ND
SDRAM 3	512M (64M x 8)	54-TSOP II	143MHz	$16.79	706-1420-ND

In Table 63, there are three different power supplies compared from the distributor Digikey. Power supply 1 meets all of the current and voltage requirements, but it has a negative center power supply barrel jack output connector. This is incompatible with the through hole mounted power barrel connector jack since its center is positive. Power supply 2 meets the voltage and current output requirements but the output connector is not the barrel jack connector. This is an important requirement and therefor this power supply cannot be considered. Power supply 3 is the best choice. It meets all of the previously stated requirements and has a positive center polarity. The power supply output barrel jack connector has dimensions 2.1mm inner diameter (ID) and 5.50mm outer diameter (OD) which means the connector on the PCB must match dimensions for the connection to work. The through hole mounting type was specified for the PCB jack connector because it can withstand more mechanical stress that could be applied when repeatedly removing and connecting the power supply connector.

Figure 64: Power Barrel Connector Specification Comparisons for Part Selection

	Voltage rating (VDC)	Current rating (A)	Mating diameter	Mounting type	Part number
Jack 1	12	2	1.35mm ID, 3.50mm OD	Solder	CP-2519-ND
Jack 2	12	5	2.10mm ID, 5.50mm OD	Solder eyelets	EJ508A-ND
Jack 3	24	5	2.00mm ID, 5.50mm OD	Solder	CP-102A-ND

In Table 64, the power barrel connector jack, there are three different options compared from the distributor, Digikey. Jack 1 meets the current and voltage requirements to match the power supply but it has different mating diameters which is not compatible with the power supply output connector already selected. Jack 2 meets the current and voltage requirements but has solder eyelets which is an undesired through hole pin shape and difficult to replace. Jack 3 is the desired and chosen through hole connector because it meets the current and voltage requirements as well as having flat through hole pins for easy installation and removal.

5.2.4.9 Voltage Levels and Regulators

From the list, as seen below, of voltage supply levels of components requiring a voltage source the voltage values can be determined. The voltage levels that are required to be supplied to components are 3.3V and 1.2V. This information helps to determine what kind of voltage regulator is needed based off of the desired voltage levels themselves. Also the number of different values of voltage equals the number of different voltage regulators The next section will explain in more detail the voltage regulators that were chosen to meet these voltage levels.

- FPGA:
  - VCCINT= 1.2V
  - VCCAUX = 3.3V
  - VCCO_0,1,2,3 = 3.3V
  - VREF = 3.3V
- JTAG:
  - VDD = 3.3V
  - VREF = 3.3V
- SDRAM:
  - V_{DD} = 3.3 V
  - VVDDQ = 3.3V

To decide on which voltage regulators to use requires the same type of research from the initial power supply voltage amount decision and more information about the current draw of each component. There is a voltage regulator for each level of voltage required by each component. The current of all of the components added together will be the maximum current draw from the power source through the voltage regulators. So a voltage regulator that will allow that much current to pass through without overheating is the one we will be selecting.

Figure 65: Voltage Regulator Component Specification Comparisons for Part Selection

	Regulator Topology	Mounting Type	Voltage In/Out	Current Out	Part Number
VR 1	Positive Fixed	Through Hole	<6V, 3.3V	250mA	MCP1700-3302E/TO-ND
VR 2	Positive Adjustable	Surface Mount	<40V, 1.2-37V	100mA	LM317LDR2GOSCT-ND
VR 3	Positive Fixed	Surface Mount	<20V, 3.3V	1A	NCP1117ST33T3GOSCT-ND

Table 65 shows three very different voltage regulators, there are so many to choose from and such different characteristics to specify. From the list of voltage characteristics on page 5.2.4.9 there are two different voltage values which means two different voltage regulators are required. As per the total current draw amount established in Section 5.2.4.8 Power Supply on page 101, the total current output minimum of 500mA. Above, Table 65 shows VR 1 and VR 3 with positive fixed regulator topology. This topology is the typical type for applications with simple supply voltage requirements. Positive fixed topology means that the output voltage is positive voltage with respect to ground. Negative topology means the output voltage is negative with respect to ground which cannot be attained by simply switching the connections on the positive topology voltage regulator. VR 2 is a positive adjustable regulator topology which is useful but not necessary for our project since we only need two set voltage supply levels. VR 1 has a current output that is 250mA which does not meet the 500mA minimum current output requirement. That narrows it down to VR 3 which has twice as much maximum current output as the minimum requirement at 1A. This voltage regulator was selected for converting the 12V power supply to 3.3V. A second voltage regulator was selected to convert 3.3V to 1.2V.

5.3 Software

The aspects of software design in the project encompass acquiring a dataset, preprocessing, and classification, all within an intuitive GUI. In the sections below, we outline how we curated our dataset, the preprocessing that we do, the classification, and the GUI. In general, the storage, preprocessing, and GUI are implemented on the Raspberry Pi and the classification is implemented on the FPGA.

5.3.1 Speech Recognition

5.3.1.1 Algorithm Choice

Among the classification algorithms that are commonly used in speech recognition, we chose to go with a feedforward deep neural network (DNN). The primary reason that we chose this algorithm over the others was its computational complexity. HMMs are complex, statistical models that take a significant amount of time to understand and implement. CNNs, on the other hand, are simply too costly to implement on low-cost hardware. DNNs are simple to implement on low-level hardware and the network can be designed to be small enough to run effectively on our low-cost FPGA. Overall, our choice is optimal for maximizing performance and ease of implementation.

5.3.1.2 Dataset & Preprocessing

The preprocessing stage is essential in order to get our inputs in the ideal format for our classifier. Just as the semantics of natural language can be finicky, so can comparing speech signals: even from the same individual. Speech signals will be initially sampled via a microphone at 16,000 Hz and stored in a WAV file. We chose our sampling rate based on the fact that 16,000 Hz captures all of the details we might want about a particular utterance, while avoiding the extra computations that would come with oversampling (i.e., having more samples per signal). The WAV format was a natural choice for us as it is an uncompressed file format for audio signals.

Our dataset currently consists of six people's speech sets. Each speech set contain five more subsets, where each subset represents that individual saying every word in our vocabulary once. The data is collected using the sampling rate and file format mentioned above. We are always expanding the dataset to increase the generalizability of our classifier.

To account for any noise that occurs during initialization, we scan through the speech signal and look for the first sample that is greater than or equal to approximately 40% of the maximum value of the absolute value of the signal. We set this point as the starting point and take the next 6999 samples per signal. This heuristic is necessary in order to achieve a simple, automatic preprocessing routine that can grant us good classification results during testing.

Once we have trimmed our raw signals to 7000 samples, we partition each signal into a collection of frames and compute the MFCCs for each frame. For compatibility with our classifier, we scaled the MFCCs so that they fit between -1 and 1, divided by 2, and then shifted by half so that our feature vector lies between 0 and 1. This reduces the chances of the DNN having saturated inputs into any of the hidden or output nodes, which are detrimental for learning. Finally, this values within this feature vector are approximated to their closest 8-bit fixed point representation.

5.3.1.3 Classifier

As mentioned previously, our speech classifier is a feedforward DNN tasked with classifying a vocabulary of 14 words. The vocabulary consists of the numbers 0-9 and the cardinal directions: east, north, south, and west. There are several other powerful choices that are often employed in speech recognition systems, such as hidden Markov models (HMM) and recurrent neural networks (RNN). These algorithms tend to be quite sophisticated in terms of implementation and require computational resources that we do not have with our hardware. Consequently, we chose to go with a feedforward DNN for our application.

The network has 516 nodes at the input layer (to resemble the size of our feature vector), 100 nodes at the first hidden layer, 50 nodes at the second hidden layer, and 14 nodes at the output layer (to resemble the size of our vocabulary). For training, the network uses the sigmoid function as the activation function and gradient descent with weights initialized from sampling a normal distribution with parameters dependent on the number of hidden layers in the network. The network does not currently have a regularization term.

In desktop and laptop processors, floating-point precision is rarely a concern. Because we plan to implement this network in a low-cost FPGA, we are limited by the amount of block memory and, hence, are restricted in the amount of precision we can utilize. Our nodes will be unsigned and encoded using 8 bits and our weights will be signed and encoded using four bits. The sigmoid function will also have to be approximated by using a combinational approximation. Figure 2 demonstrates the tight fit that the combinational approximation has with the standard sigmoid function. Empirical tests show that the error induced by the approximation is minimal: significantly less than 1%. Our learning algorithm (i.e., gradient descent) is unaffected since the training will be off-loaded to the Raspberry Pi.

With an understanding of the preprocessing steps and the speech classifier, we can now take a bird's eye view of the classification process. We will split up our discussion into a training phase and a testing phase, according to where each phase is processed.

During the training phase, we perform all of our computations on the Raspberry Pi. We first preprocess all of the raw signals in our dataset. This includes trimming, framing, and computing the MFCC feature vector for every signal. We then instantiate an instance of the DNN to initialize the weights, set the learning rate, and set the number of epochs to iterate through. Iterating through the training set multiple time via epoch iteration is useful for "generating" additional training data for the network to learn from. Though this can cause the network to overfit the training data, we have tested that 100 epochs are generally sufficient to obtain the classification results we want (greater than 80%) without overfitting.

During the testing phase, we move our computations over to the FPGA. First, we approximate the weights generated on the Raspberry Pi to values that fall within the precision available with four bits. These weights are then exported to a text

file, converted to an appropriate format for the FPGA to read, and then sent to the FPGA via serial peripheral interface (SPI). At this point, the system switches to testing mode. Any speech signals that are recorded from the mic are converted to an appropriate format for the FPGA on the Raspberry Pi and then sent over for processing. The FPGA will then communicate back to the Raspberry Pi with the classification results.

5.3.2 Graphical User Interface Design

The user will be able to interact with the speech processing system through a graphical user interface that can run on a computer.

5.3.2.1 Functional Requirements

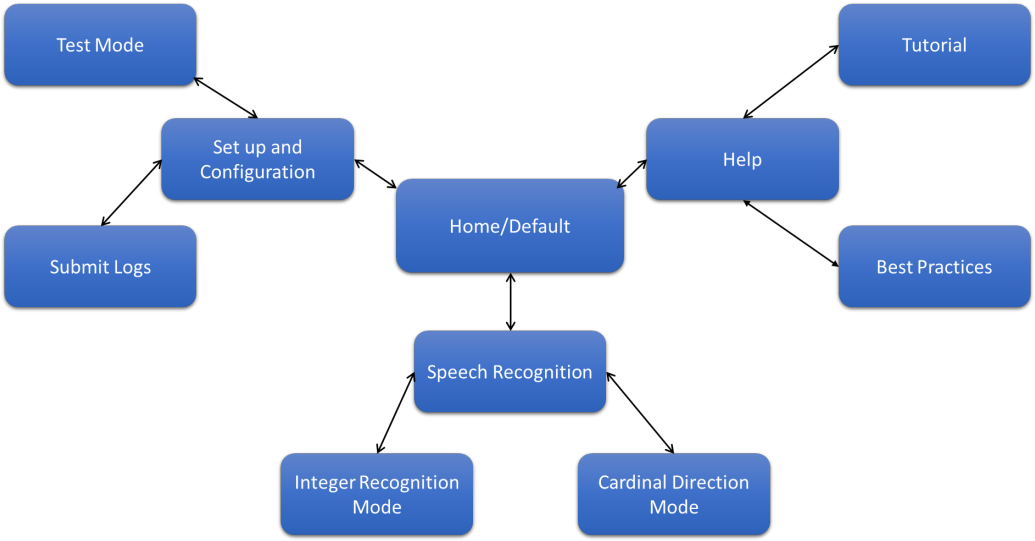
- Counting Mode
  - Allows user to set this mode to on/off
  - When this mode is on, it allows the user to verify if spoken numbers are recognized correctly or not
- Directions Mode
  - Allows user to set this mode to on/off
  - When this mode is on, it allows the user to verify if spoken cardinal directions are recognized correctly or not
- Logging
  - Allows user to keep data logs for future review

5.3.2.2 Block Diagram/State Machine

The following block diagram serves as a general guide for state machine of the GUI. This will allow us to better understand the behaviors we expect the user to take when interacting with our application.

We can use the state machine to ensure that the actual application successfully allows for the event and state transitions as depicted here:

Figure 66: GUI State Machine Block Diagram



6 System Design Summaries

6.1 Hardware Design Summary

1. Preliminary schematic
2. Breadboard with Development board (test I/O pin selections for LEDs)
3. Final Schematic Rev0
4. Layout PCB Rev0
5. Order PCB Rev0
6. Place Parts PCB Rev0
7. Test PCB Rev0
8. Route 1: Rev0 works
    (a) Order spare parts
9. Route 2: Rev0 needs modifications
    (a) Preliminary Schematic Rev1
    (b) Breadboard with PCB Rev0
    (c) Final Schematic Rev2
    (d) Layout PCB Rev1
    (e) Order PCB Rev1
    (f) Place Parts PCB Rev0
    (g) Test PCB Rev1
    (h) Route 1: Rev1 works
        i. Order spare parts
    (i) Route 3: Rev2 needs modifications
        i. Repeat Route 2 for Rev2 until Route 1 is chosen

6.2 Software Design Summary

DeepGate will utilize one software entity that will encompass most of the user interactions with the hardware. Although there are pre-existing tools that are compatible with FPGAs, our goal is to be able to have a custom graphical user application that would better suit our needs. This GUI would allow us to better test and maximize our efficiency by allowing for multiple configurations. Our hope is that instead of having to require continuous hardware compilation, the finished GUI would feature settings that the user can easily and quickly implement with just the click of a button.

While this is clearly something that is possible given the availability of software tools offered by embedded hardware providers such as Xilinx and Diligent, there are a lot of factors that need to be considered when designing an application that interfaces with complex hardware. This section will go into further detail on how this can be implemented.

6.2.1 Establishing Serial Communication

The FPGA we are looking to utilize is configured with a microcontroller. This microcontroller allows for the sending and receiving of serial data using the USB port as well as analog to digital conversion of information such as voltages. The first step is to ensure confirm the settings are appropriate for establishing serial port communication over the USB and sampling analog information.

In addition, as covered in our research regarding serial port communication, we can interface with the microcontroller by utilizing the SPI module. In this case, the microcontroller is used as a means to detect whether the FPGA has new information which can be read or written. One of the best methods of handling this would be through using the Memory Mapping Technique as described below.

6.2.2 Memory Mapping

The reading and writing to the SDRAM can be achieved through a memory mapping technique. This is often utilized in the programming of microcontrollers and other processors, particularly in instances where one needs to interface with inputs and outputs from another device. It is also known as memory-mapped input/output. Another related method for dealing with inputs and outputs between processors and peripheral devices is port-mapped I/O. The advantages and disadvantages of each of these methods will be further explored in the following.

The memory mapping technique is extremely common and useful. The memory and input/output devices are addressed from the same address space. Each of the registers and memory of the peripheral devices is associated with address values. The advantage of this is that it allows the processor to access peripheral devices in the same way it accesses its memory. Port-mapped input/output has the distinction that the address space it references is isolated from the main memory.

Memory mapped I/O is well known for its utility in embedded systems and it follows the general principles of reduced instruction set architecture. Since the port-mapped I/O includes the need for an isolated address space, it sacrifices some simplicity. However, having that specialized bus offers the advantage of speedier operations. Typically, peripheral devices are usually slower than main memory. This is only worsened when the buses for the address and information are shared. Memory mapped I/O, due to its lessened complexity, is inherently more efficient, more compact, and speedier in other aspects.

In order to implement the memory mapping technique on a device that does not permit direct mapping to the memory space, such as a microcontroller, we have to

consider other avenues of access. A viable option of doing this would be over a serial port interface bus. A register interface over the SPI bus is a suitable alternative to utilizing the ADC ports on the FPGA. In the software, we can then specify the addresses that we need to use as well as configure the appropriate pins as inputs or outputs of the FPGA.

By exploring the different types of methods in dealing with inputs and outputs with peripheral devices, we learned that memory mapped I/O and port-mapped I/O methods are both viable and effective techniques. With the ever increasing size of processors in regards to computer architecture, the memory mapped I/O technique is a very popular choice. Any concerns with the range of memory address space have been virtually eliminated. There is more than sufficient space for the memory and peripheral devices. However, both methods are comparable in their effectiveness in present times.

6.2.3 Interfacing via JTAG

Although the microcontroller is a strong option, we are also looking into interfacing with the hardware through the JTAG and SDRAM chip as an alternative to the microcontroller. This is due to the fact that working with both of the devices adds complexity to debugging. By utilizing the JTAG as a port for serial communication, the JTAG can record any input/output operation between the microcontroller and the FPGA. The data, which is stored on the FPGA's own memory resources can be transferred easily to a personal computer.

Using a communication circuit that the vendor supplies with the JTAG, we can add access to the data to the user. Using specific application programming interfaces (API's), the information can be interacted with by the user through a GUI.

Often, manufacturers of FPGAs do also provide APIs in TCL/TK or Tool Command Language Tool Kit to allow interfacing between the JTAG and the personal computer. These APIs can provide a basis for user logic and interactivity.

6.2.4 Receiving Algorithm Feedback

The neural network will output the particular word or phrase that it has the most confidence that the user is saying. This information will be passed through the graphical user interface logic and ultimately output as a status. There will be a set of functions that will control this interfacing.

6.2.5 Speech Validation and Recognition

As we expect to have immediate output after speech input is achieved, we can verify whether or not the output was accurate. We can also close this feedback loop by providing user input that would confirm whether or not what they said turned out to be the actual text. This can be simply implemented as two buttons marking "Yes" if the output was as expected or "No" if it was not.

6.2.6 Status/Log Tracking

Keeping logs regarding the user interaction is extremely useful in our application. However, it is not as important as the immediate output that is displayed on the graphical user interface.

The purpose of the log tracking is to be able to save a downloadable file for our records to review the output and feedback of previous sessions. We will strive to achieve this as a text file that is either generated by a command or automatically generated.

The possible difficulties in achieving this feature is dependent on how much we would like to format this file. As the amount of processing for the log file increases, so will the difficulty. For mostly our part we will work to ensure that a simple text file of the session can be saved.

6.2.7 User Interface Design

User Interface Design is all about making the application as user-friendly as possible. The user is the center at which the design is focused on. It is important to make sure that the user can navigate the application as efficiently as possible with as little confusion.

To do this, it is important to use a consistent aesthetic across all the elements. Keeping the user interface design simple also allows the design to not distract the user from performing the intended functions.

Qualities like color and texture must be used with a purpose and these can influence what will the user direct their immediate attention to. Typography can also be used to emphasize in the same way.

It is also important for the user interface to communicate feedback. This includes any changes regarding the user interface as well as in state and errors.

In user interface design there are a variety of elements that can be included. The input controls include include:

- Input Controls
  - Text fields
  - Checkboxes
  - Buttons
  - Radio Buttons
  - Dropdown lists
  - Toggle

These input controls are useful for anything requiring user input. Due to the fact that we want to make the interaction as simple as possible. We will likely utilize mostly buttons and perhaps a text field to take in a user name or add in a note.

- Navigational Components
  - Slider
  - Search
  - Tabs
  - Tags

As our graphical user interface will require the use of different modes, a tabbed or grouping can be useful in directing the user's attention.

- Informational Components
  - Notifications
  - Messages
  - Modal Windows
  - Icons
  - Progress bars

Finally, we can utilize modal (also known as pop-up) windows to help the user realize we are in a different session. This can be very useful for the user to start a session in any of the two recognition modes and easily navigate back to the default by exiting the modal window. The graphical user interface needs to be designed with the user in mind. Understanding the user's goals, what they like, their preferences, and observing how they tend to react to different scenarios is extremely important.

The difficulty in user interface design is that as the base of users increase, there are an increasing amount of preferences and tastes to be taken into account. Therefore it is important to focus on the most general commonalities of users and also prioritize their tastes. In our project, we will first to strive to ensure all group members can easily utilize our tool and then proceed to verify random users can also reasonably navigate our graphical user interface.

We will strive to design our defaults to reduce frustration on the user. We will also conduct multiple usability tests to ensure that this will be the case.

7 Project Prototype Construction

7.1 Part Selection and Acquisition

The parts selected for the custom PCB design for our system are listed in the Bill of Materials Figure on page 130. The decision making behind each part selected is detailed in Section 5.2 Hardware on page 78. All of the parts selected are in stock and available for shipment immediately from Digikey which is an electronic parts distributor. We will be ordering from them several times for initial parts and extras if we find the need. Electronic parts shipments from Digikey follow all electronic packaging standards to prevent static charge from damaging the parts.

7.2 PCB Vendor and Assembly

There are hundreds of custom PCB fabrication vendors to choose from. These vendors are able to produce small numbers of custom PCB boards because they are able to place more than one different design on the same board sheet for fabrication. This manufacturing method has allowed custom PCB designs to be made very inexpensively.

Depending on the number of layers and the size of the PCB the cost changes. To keep expenses down we will limit the size of the PCB layout as much as we can.

After looking at several fabrication companies, still leaving our options open to better companies or deals, we have selected a preliminary vendor. Silver Circuits is a PCB manufacturer that specializes in prototype and low volume PCB production which is exactly what we need. The PCB prototypes start at $12.50 each and can be up to 4 layers thick and a maximum board size of 8.4” x 5”. Their lead time is 8 working days for 4 layer boards so we will make sure we send our order out as soon as possible to avoid delays in the prototype plan.

We will try to self-solder all components initially. If this proves to difficult or doesn’t work, we will outsource our surface mount assembly parts to a pick and place vendor. These vendors will use machines to place the surface mount parts onto the board. We have initially selected the vendor Small Batch Assembly as our assembly service. They will only place the surface mount parts, and then we can solder the through hold parts ourselves when the boards arrive.

7.3 PCB Prototype Construction

The board has been designed using EagleCAD software. The design will be sent to a company, once we decide between a couple, to create the PCB with only traces and holes. Then we will either solder all of the components on ourselves or select a pick and place company to do it for us. We will probably try to do it ourselves first since the parts are relatively cheap and we can get multiple PCBs fabricated. If that doesn’t work, then we will send the board and our parts to the pick and place company we decide on. Once we finish or receive the board with all the parts on it then we will be able to test the functionality of the PCB and verify that it works

properly according to the description in proper operations. Then proceed with the PCB according to the PCB Prototype Testing Plan.

7.4 Facilities and Equipment

As students of the College of Engineering and Computer science we have access to numerous labs on campus. There are different labs that can be helpful for different aspects of our project. The facilities these labs are within are very student focused and create a learning environment. The student focus helps us realize that other people are there doing the same things we are doing and therefore we feel very comfortable working near our peers. The learning environment places emphasis on the student and their success. The professors and lab managers contribute greatly to the support and help students receive within the labs. We will be spending a great deal of our time within the Innovation lab. There are also other labs available to us such as the machining lab and lots of computer labs. The computer labs are extremely useful for research and designing with the team. There are printers available and that is convenient when needing to print the design to bring to the other lab as a reference.

The innovation lab has three large tables that are perfect for building projects on a large scale. Above the table are outlets hanging which is so accessible and suited for modern day engineering students. There are computer stations along one wall which makes testing with programs and the project within reach inside the lab. Above the computers are power supplies and various testing equipment for use while running initial tests on a breadboard set up. Along the back wall are large tool chests with a large inventory of helpful tools that come in handy when building. On the last wall there are three soldering stations. These soldering stations are amazing with high quality soldering irons stations, direct movable lighting and solder fume fans. These fans work to keep users from inhaling the fumes from soldering by pulling the air away from the user. In the back part of the innovation lab there is even more equipment that can be used on a first come first serve basis.

The machining lab will be valuable to us as we near the end of the project and have finished testing the PCB. When the PCB works correctly we can put it inside of a housing. We will be able to make our own design for the housing for the PCB based foremost on the size of the PCB itself. Once we have the design for the housing we can take it to the machining lab and they will be able to create it for us. When it comes to that point in the project we can decide what material we want it to be made out of which will force us to build it a certain way. We can choose to 3D print the enclosure for the board which would be one of the cheapest options available. It would be simple to design, cheap to make and simple to assemble. On the other hand, using flat materials that need to be cut and fastened together would be time consuming, expensive and difficult to assemble. Whatever we end up deciding for our enclosure fabrication material we are comfortable knowing the labs will be able to help create our desire.

During our breadboading and prototyping stages of the project we will be utilizing

the Innovation lab as much as possible. It is thoroughly equipped for us to successfully build and test our PCB. We will be able to utilize the soldering stations, tools and assistance from anyone in the lab whenever we may need it. We have several labs available to us conveniently located within one of our college's buildings. They are there just for us and we plan to take full advantage to all that they have to offer.

After the design phase, the company 4PCB was chosen to manufacture the custom prototype design. They were able to make a single board for $66 for students. Then after a week, the board was received along with the parts that were ordered from Digikey, ARROW and Sparkfun. The parts where labeled with their location on the board. The company QMS assembled all of the components onto the board in two days. This was an incredible time and money saver for our team as it would have taken two weeks and $250 with an outsourced company in the USA. It would have taken four weeks if we had it assembled in China.

8 Project Development Board Testing

8.1 Hardware

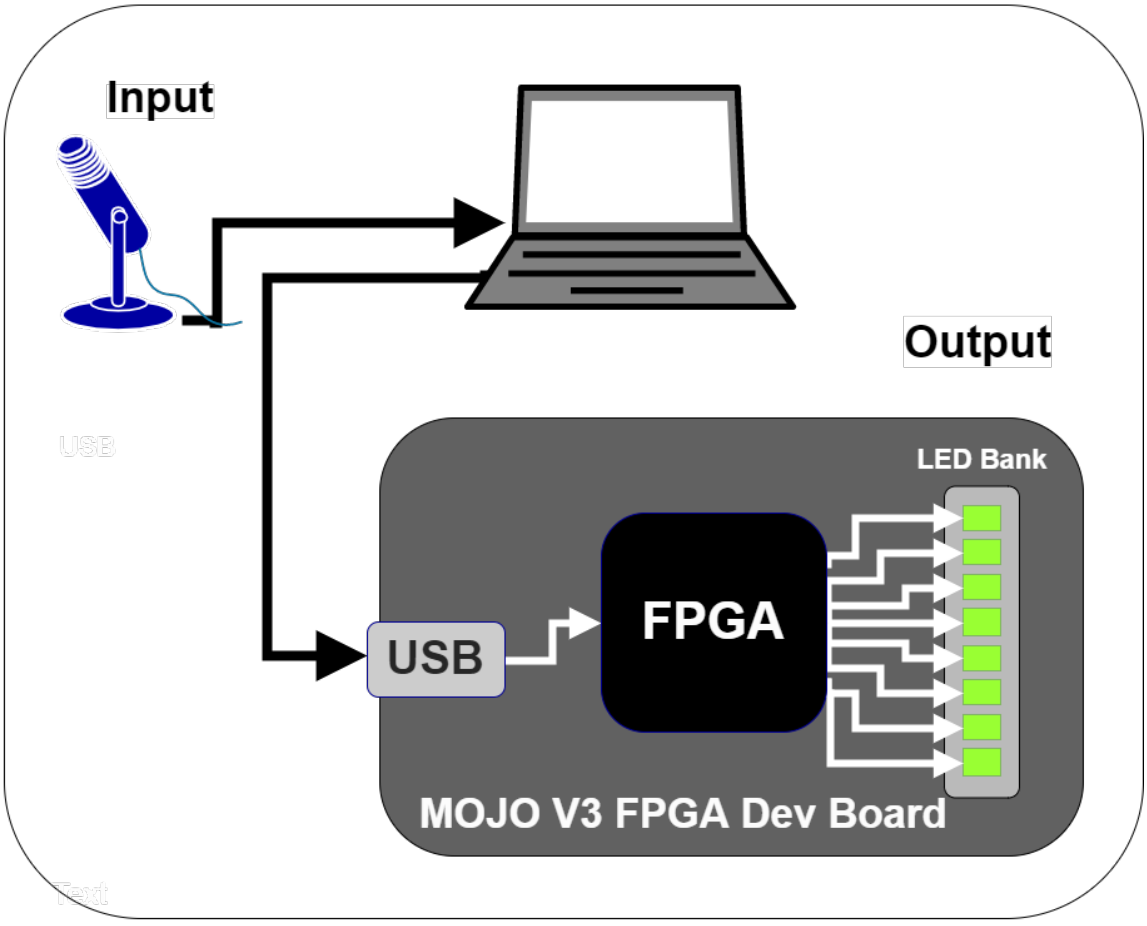
8.1.1 Design

8.2 Breadboard Prototype

8.2.1 Breadboard Design

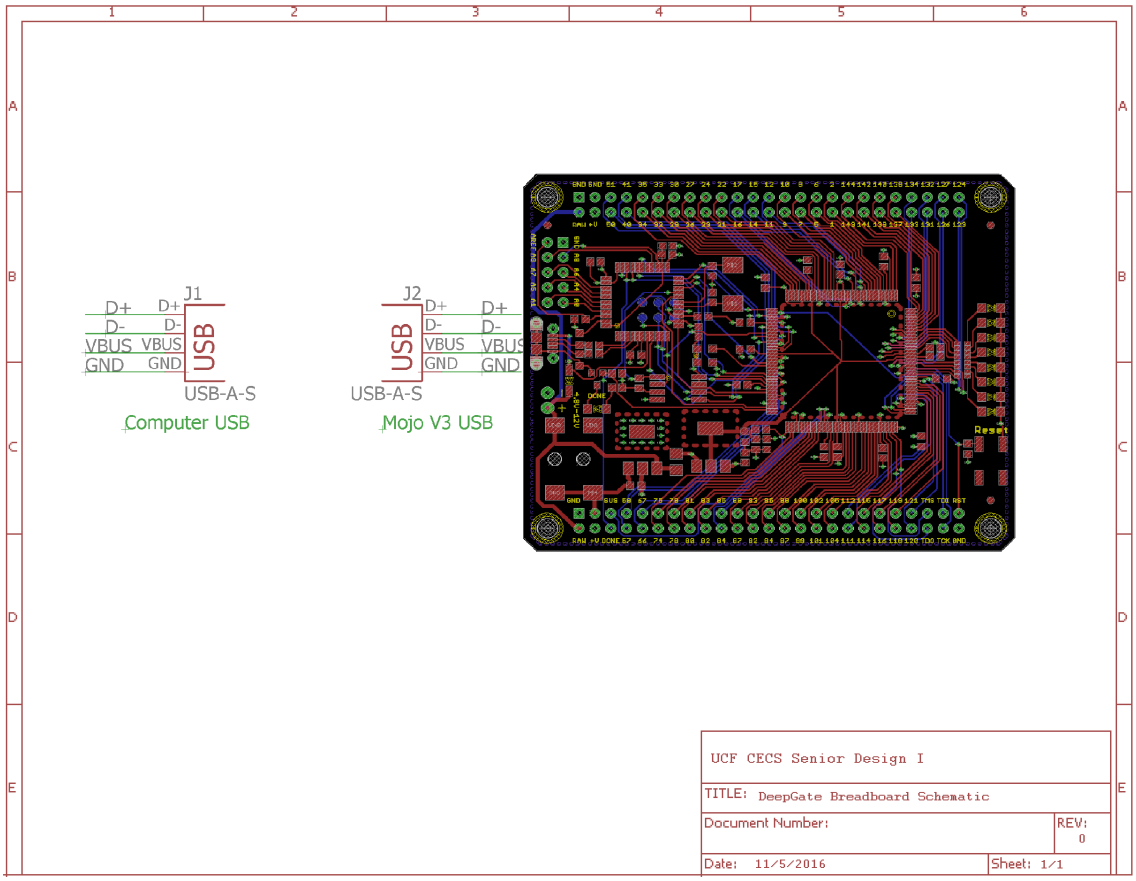
For preliminary testing of the hardware we used a development board with an FPGA and LED components. For more information on the development board see Section 8.3.2 Prototype on page 123. A breadboard block diagram was drawn up as a preliminary view to what the test set up would look like before the schematic was made, seen below Figure 67. It is always a good idea to draw a block diagram before the schematic to get a simplistic view of all the components involved, what kind of power or data is transferred between them and an overall picture of the design together.

Figure 67: Breadboard Block Diagram



The breadboard schematic is very simple, Figure 68 below. This is due to the fact that it includes a preassembled development board with all of the necessary peripherals to test the FPGA connections with the LEDs. The development board used in testing is an Embedded Micro Mojo V3, which is explained in further detail in Section 8.3.2 on page 123. The main goal of this project is for the FPGA to implement a deep learning framework to identify speech patterns. Since the FPGA is the main hardware focused component that was the priority in testing the FPGA development board and not including some of the other peripheral components in the breadboard test. The FPGA, JTAG and SDRAM do not come in through hole mounted parts so it was impossible to breadboard all those components together. The purpose of the breadboard test was to establish a proper function between the FPGA and the LEDs, which is the most important part of the project. Since the development board already included LEDs connected to the FPGA we used those in the experiment and didn't not see the need to add LEDs to the header pins to test functionality.

Figure 68: Breadboard Schematic



8.2.2 Breadboard Testing

For preliminary breadboard testing of the design the Embedded Micro Mojo 3 FPGA development board was used. This development board is explained in depth in Section 8.3.2 on page 123. A simple program was written in order to test the connections between the computer, FPGA and LEDs. The program was written to be used with RealTerm, also explained in **??**.

Connections to be tested:

- PC to Development Board USB
- USB to Microcontroller
- Microcontroller to FPGA
- FPGA to LEDs

8.2.3 Experimental Setup

Breadboard Setup:

- PC
- USB cable
- Embedded Micro Mojo V3 Development Board Containing:
  - USB connector
  - Microcontroller
  - FPGA
  - 8 LEDs
  - Resistor Bank
  - Reset Button
  - Clock

Figure 69: RealTerm Development Board Testing Program

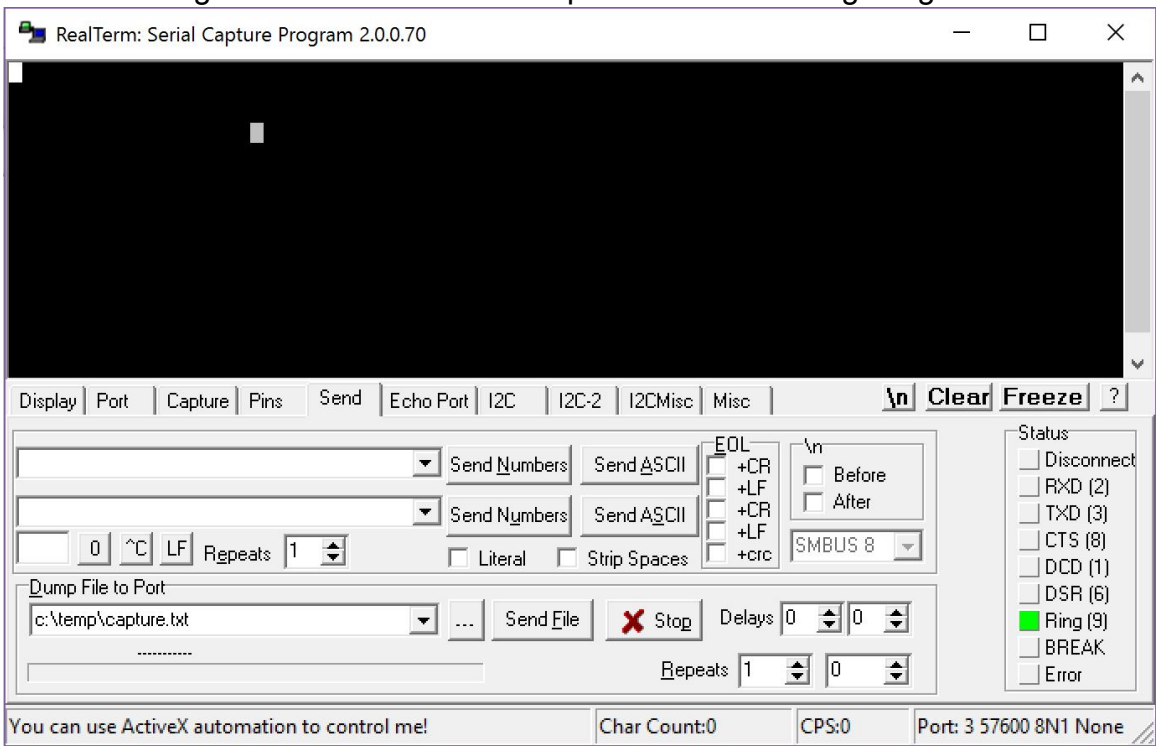
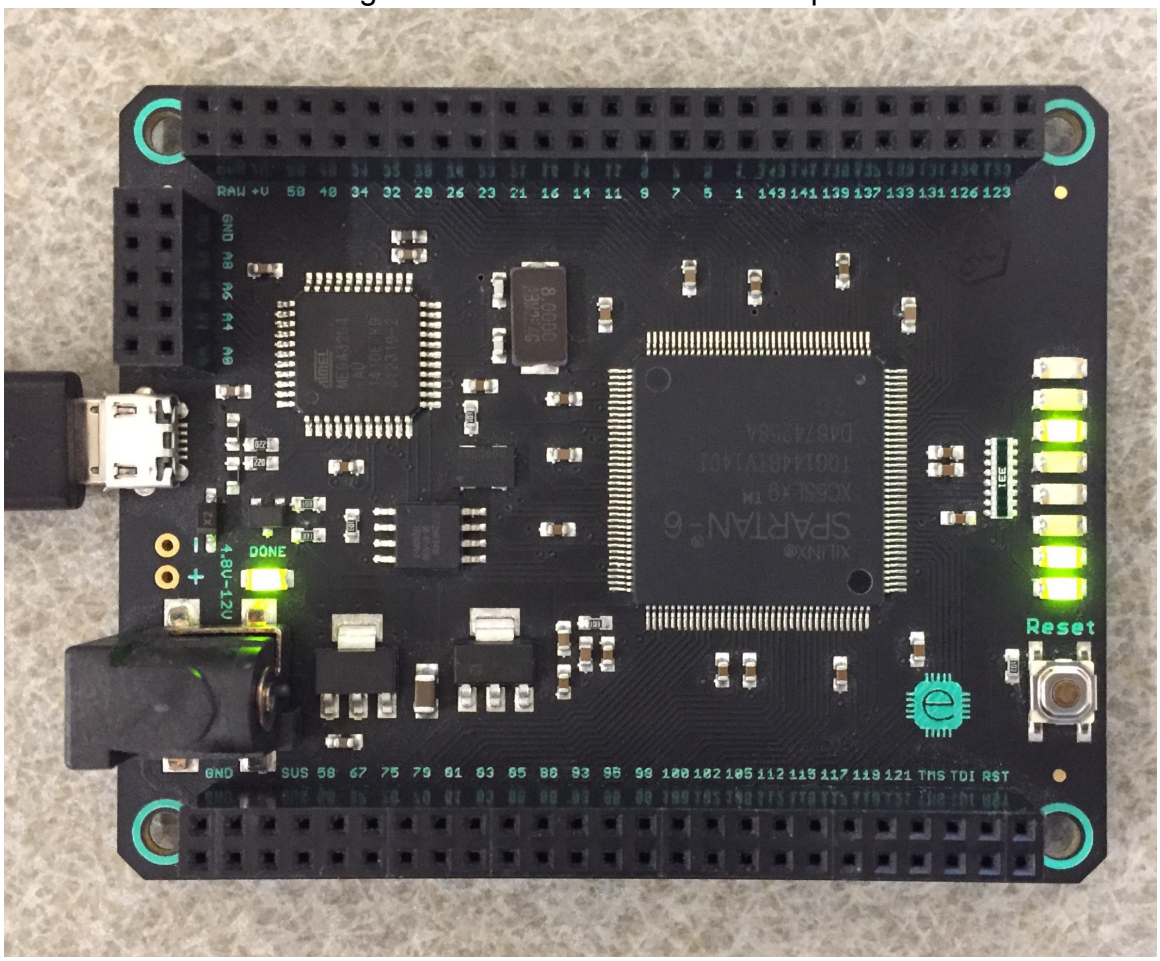


Figure 69 shows the interface of the testing program that communicates with the development board from the PC. The status column on the bottom right shows which LED pins are lit up at any given time during the test.

Figure 70: Breadboard Test Set Up



Seen in Figure 70 the USB from the computer is connected to the USB connector on the Mojo V3 development board. The board uses a micro controller (smaller quad-flat package chip on the left side of the board) to interface the USB signals with the rest of the board's functions, mainly the FPGA (the large chip in the middle). The FPGA is connected to the microcontroller using specific pins established for that use and the FPGA is programmed through those pins. The microcontroller has a large amount of pins, many of them being general purpose I/O pins. Eight of these I/O pins are connected to an isolated resistor network which is then connected to eight LEDs. Each LED represents a different I/O pin connection.

Experimental Steps:

1. Write program to test LEDs
2. Download and Install RealTerm
3. Upload Program to JTAG

4. Connect Mojo V3 to PC with USB cable
5. Verify Board Startup, LED indicator light is steady on
6. Use RealTerm to run program on JTAG
7. Press PC number keys to test application LED function
8. Verify that with a number selection the proper application LED lights up
9. Repeat step 8 for adequate data

Results: The LEDs successfully lit up. This means that the main hardware piece, the FPGA, can successfully send and receive information with all the other peripherals. This includes the computer via the microcontroller to USB connector. The LEDs via general I/O pins. Since we are not using the microcontroller utilized in the development board for the experiment there will be a slightly different PCB design but the real test was to make sure the application side of the project could be implemented with the FPGA and LEDs. The connection between the FPGA and LEDs should be similar. One of the JTAG's applications is testing an FPGA so that connection should work fine. We could not test that due to the surface mount nature of the FPGA and the JTAG.

8.3 Firmware Testing Environment

8.3.1 ModelSim

ModelSim is an HDL simulator developed by Mentor Graphics. It is used to simulate the behavior of our circuit before it is synthesized, routed, and placed on the FPGA. Additionally, it has a full Verilog debug suite capable of decoding simulation directives and displaying visually the value of every signal in our design at every time step in a defined interval. This is more commonly known as a waveform diagram. ModelSim is an essential part of the testing environment, allowing us to inspect our neural network and take note of discrepancies between actual and intended circuit behavior before the FPGA is programmed. Debugging run-time errors is extremely tedious and can sometimes be impossible without the use of dedicated simulation tools. We take advantage of ModelSim by writing Verilog test benches that it compiles and runs. These test benches place our design under test by emulating the signals that would be sent into it post FPGA configuration. ModelSim verifies that the output behavior of the unit matches the intended behavior outlined in the testbench. If not, these test benches instruct ModelSim to output detailed error messages to a console, aiding us in our efforts to ensure the correct functionality of our circuit.

Furthermore, it has the advantage of being compatible with both Altera Quartus and Xilinx ISE as it is completely vendor independent (it merely compiles Verilog code that is compliant with the accepted standard). In this project, we use ModelSim-Altera version 10.3d.

8.3.1.1 Main Testbench

As the design hierarchy became increasingly complex, the main test bench used in simulations of our circuit grew in terms of design coverage and test vectors used. It started out as a small piece of code intended to test the functionality of a single processing unit and is now capable of driving and reading the ports of the high-level tile pipeline architecture. This test bench simulates the input of data to the pipeline and verifies that the classification results match pre-determined values. Furthermore, it is capable of triggering certain events such as a pipeline reset, pipeline stop, and PLL loss of lock. Behavior in every test case is automatically verified and error/status messages are output to the console for further review. Anytime a change is made to the under lying unit-under-test (the pipeline), the test bench is compiled and run in ModelSim to ensure functionality is maintained. If a discrepancy in behavior is logged, the test bench will note at what time-step it occurred. This time-step can then be located in the ModelSim waveform viewer and the source of the error can be determined by tracking data and control signals. Thus, using this test bench in conjunction with ModelSim, we perform a comprehensive review of the circuit before the Spartan-6 is configured to realize it.

8.3.2 Development Boards

FPGA development boards are used for prototyping both ASIC and FPGA based designs before they are put into production. Moreover, they serve an educational purpose, allowing those without the capital required to manufacture ASICs or their own custom boards the ability to practice and learn digital hardware design.

8.3.2.1 Embedded Micro Mojo V3

For this project, we will prototype our digital circuit on a Mojo V3 FPGA Development Board created by Embedded Micro. The Mojo has 84 GPIO pins, several LEDs, and an ATmega32U4 microcontroller that is used for programming the FPGA, amongst other things. Most importantly, it sports a Xilinx Spartan-6 XC6SLX9 FPGA, the same chip we will be implementing on our own printed circuit board. Due to this, and the availability of extensions to the board such as the SDRAM shield we will go into in the following section, the Mojo is an ideal board for prototyping our neural network.

The presence of a Xilinx chip signifies that the compilation process and its corresponding programming files can be generated using the Xilinx ISE software ecosystem. Most development boards can be programmed directly using the vendor's software, however, because Embedded Micro used a microcontroller with an SPI interface for configuration (avoiding the JTAG pins), we must program the board using Embedded Micro's proprietary Mojo Loader program. This does not affect the development process in any major way. The only subtle difference is that ISE must be instructed to generate .bin programming files.

A slight advantage of this development board is the presence of a flash memory chip that allows us to store our .bin programming files in non-volatile memory. When the

board is powered on, the microcontroller will automatically configure the Spartan-6 with the contents of the flash memory, bypassing the need for an external interface at all. This could prove valuable during any demonstrations by not having us waste time on logic configuration.

There are several differences between this development board and our final purpose-built board. These are the speed grade of the FPGA, and the presence of application-specific components on the PCB, not including a microcontroller and including dedicated SDRAM.

For our own design we will be using a Spartan-6 XC6SLX9 with a speed grade of -3. This is faster than the -2 speed grade chip available on this board, allowing us to more easily meet timing closure and pump data through the pipeline at a faster rate. Otherwise, the entire compilation process is the same and the same .bin files can be used to configure both chips. However, if ISE is notified of the speed grade it can provide more accurate timing information. Additionally, we will not have a microcontroller on our board, opting to use the space for a memory chip instead.

8.3.2.2 Embedded Micro SDRAM Shield

Embedded Micro offers several extensions to their main development board. These include, but are not limited to, an LED visualizer shield, a camera shield, and a servo controller shield. We will be using an SDRAM shield during our prototyping phase. A shield is a separate PCB that connects to the main development board using the breakout headers located on the periphery of the main PCB. These are similar to the shields available to Arduino enthusiasts.

The SDRAM shield contains a 256 Mbit SDRAM chip (part number MT48LC32M8A2P-7E: G) that helps us extend the size of our neural network considerably. The FPGA only has a limited amount of block RAM, meaning neural network weights must be stored off-chip when they are not being used. Extending our usable memory size with this SDRAM means we can create a network almost an order of magnitude larger.

Embedded Micro supplies the Verilog code for an FPGA-side memory controller as well as the locations of the GPIO pins connected to the external memory chip, allowing us to focus our work flow solely on the implementation of the neural network. Furthermore, the Mojo provides almost the exact same environment our final firmware will be operating in, thus checking for compatibility issues and analyzing logic utilization metrics for our final production board can be done using the development board.

8.3.3 RealTerm

RealTerm is an open-source serial communication program that allows us to communicate with the microcontroller on the Mojo development board. We use this application while our GUI and custom serial library are being developed to facilitate data transfer and storage. Using RealTerm, we can send both ASCII and binary

values over our computer's serial port, giving us maximum flexibility in terms of command and data instructions. The program can be instructed to output data stored in text files, meaning data intended for the input layer of our neural network has to be stored in plain-text. This is much simpler than the alternative, which would entail manually sending individual values over the bus using a program like HyperTerminal. Furthermore, RealTerm has the ability to capture FPGA output data and store it in text files. This functionality is necessary to the verification of our network. On a side note, RealTerm offers a wide range of serial baud rates which allow us to maximize the speed of data transfer. This rate is currently limited by the AVR on the development board, but this bottleneck should be removed with the introduction of our custom board.

8.4 Software Testing Environment

In addition to the firmware, we will need to conduct tests on the high-level software; specifically, on our CNN and GUI. Fortunately, we can perform much of high-level testing without the FPGA, enabling us to work on the hardware, firmware, and software in parallel. For the CNN, we will use Anaconda and Keras for prototype development and testing–all in Python. Due to the limitations that the FPGA constrains on the CNN, most of our development will take place using Anaconda. On the other hand, our GUI will be developed and tested in Python using PyQt and Qt Creator.

8.4.1 Anaconda

Anaconda is a freemium open source distribution of the Python and R programming languages for large-scale data processing, predictive analytics, and scientific computing. We chose to use the Anaconda distribution because it comes pre-loaded with the Spyder IDE and many of the libraries that will be useful in developing DeepGate at every step of the way, including NumPy and SciPy.

8.4.2 Qt Creator

Qt Creator is an IDE that is a WYSIWYG interface for developing GUIs. While Qt supports multiple programming languages, we will primarily be using Python and the PyQt Libraries to develop our GUI.

8.5 Software-Specific Testing

In order to ensure that our software works correctly, we have devised several testing protocols that we will employ during development. In particular, these protocols test the speech recognition capability of our CNN, as well as the functionality of the GUI that we have developed. In general, we would like to classify two categories of speech–the numbers 0-9 and the cardinal directions: north, south, east, and west. The next several sections list and describe the steps in these protocols with greater detail.

8.5.1 Integer Recognition

First, we will look into the capability of our CNN to recognize the integers 0-9. We will test to see if the system successfully recognizes these integers by the following test plan:

1. Configure the speech preprocessing settings on the GUI to either raw, power, or MFCC.
2. Verify that the FPGA is properly communicating with the computer.
3. Verify that the microphone is actively listening.
4. Say a number between 0-9.
5. Check confirmation of input by the illumination of the LED on the PCB and the GUI.
6. Verify that the number is as stated.

The system should also output a percentage of confidence in the recognized number.

8.5.2 Cardinal Direction Recognition

The next aspect of DeepGate that we will test is its ability to classify the cardinal directions: north, south, east, and west. We will test to see if the system successfully recognizes cardinal directions by the following test plan:

1. Configure the appropriate settings.
2. Verify that the microphone is listening
3. Say North, south, east or west
4. Check confirmation of input
5. Verify if the output is as expected.

The system should also output a percentage of confidence in the recognized direction.

8.5.3 Status Logging

We will test to see if the system successfully records information from the session:

1. Configure the appropriate settings.
2. Verify that the microphone is listening
3. Operate Cardinal Direction Recognition Mode
4. Verify outputs from the GUI and check if a local text file is saved.
5. Repeat for Integer Recognition Mode

8.5.4 User-Friendliness

Our targeted user would be the team and a person that is able to use a tablet or personal computer with relative ease. In this case, the user would be at least 18 years of age with the basic understanding of the software from reading a given tutorial.

If time permits, the graphical user interface would be easily understandable even without a tutorial and designed with the principles of human centered design.

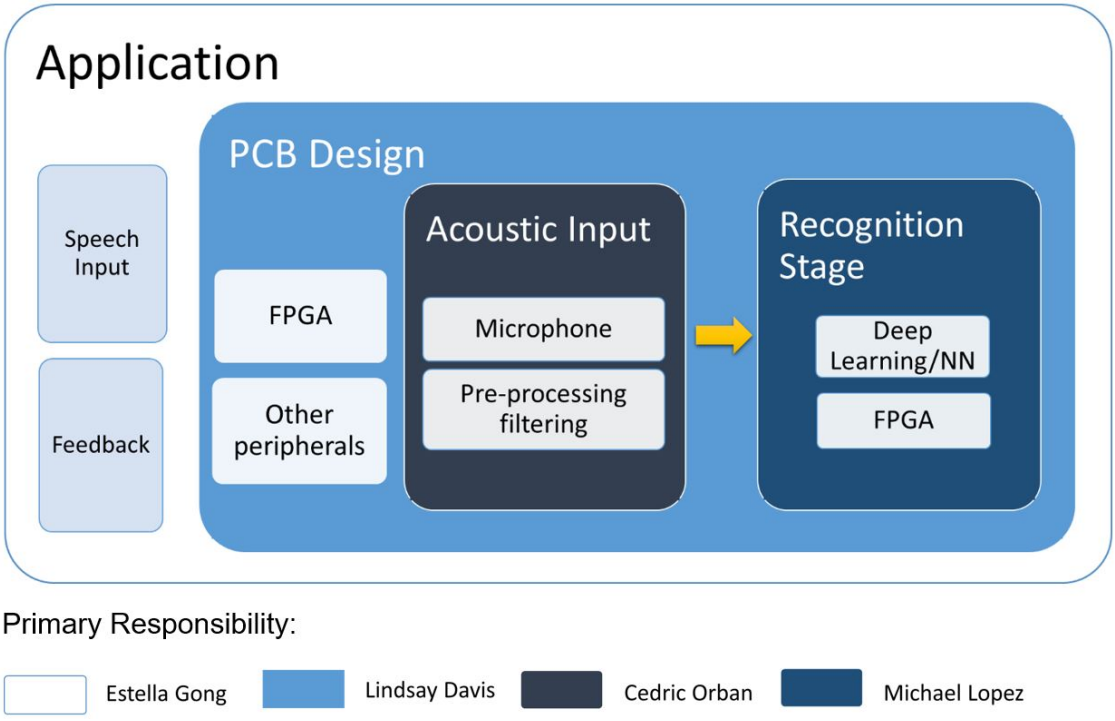
1. Verify that the hardware is connected to the PC
2. Navigate to the executable for the GUI
3. Run the GUI
4. Ask the user to navigate to the tutorial or any help pages
    (a) Verify that they can complete this task
5. Prompt the user to test if the microphone is listening
    (a) Verify they can complete this task
6. Prompt the user to run the Integer Recognition mode
    (a) Note if the user can easily navigate to the Modes group
    (b) Note if they select the Integer Recognition Mode
    (c) Note if they start speaking integers
    (d) Note the responsiveness of the application
7. Prompt the user to exit the Integer Recognition Mode
    (a) Verify if this task can be easily completed
8. Prompt the user to run the Cardinal Direction Recognition mode
    (a) Note if the user can easily navigate to the Modes group
    (b) Note if they select the Cardinal Direction Mode
    (c) Note if they start speaking cardinal directions
    (d) Note the responsiveness of the application
9. Prompt the user to exit the Cardinal Direction Recognition mode
    (a) Verify they return to the original screen
10. Perform these user experience tests for at least five unique users.
    (a) Note any failures and successes.
    (b) Note any reactions or questions they have.

9 Administrative Content

9.1 Time Management

The largest challenge to this project is the hard deadline for completion. We only have 7 months to complete this project. We will need to work a lot on our own and keep up with our own share of the workload in order to finish the project with a promising deliverable. Efficiently scheduling our time and carefully planning project steps will help us succeed. The focus of the different components of this project will be split up between the four team members. Each having a set number of responsibilities. This also requires that everyone has their own part to work on which fosters learning and team work to get the project done.

Figure 71: Application Responsibility Breakdown Diagram



9.2 Finances

9.2.1 Overview

We will need to purchase two major components at the least. Each of these components can be very expensive. We plan on applying for multiple project sponsors in order to reach our fund-raising goal and be able to purchase all the necessary parts.

9.2.2 Budget

Budget Breakdown	
Part	**Price**
Manufacturing Service	$150
Data	$200
Document Printing	$80

9.2.2.1 PCB Bill of Materials

The list of parts determined for the custom PCB are listed in Figure 72 below. Each item has information listed about it such as description and quantity per board. The multiplier is an estimation of how many extra boards or parts we will want to order initially for PCB design Rev0. The Bill of materials includes the prices of each item so that budgeting decisions can be made. It also has the part numbers of each component so that they can be found on the distributor's website, Digikey, for ease of ordering.

Figure 72: Bill of Materials for PCB Prototype Rev0

Assembly Name :	DeepGate
Assembly Number :	
Assembly Revision :	Rev0
Approval Date :	
Part Count :	50
Total Cost :	$696.75

Picture of Assembly

Distrubut	Part Name	Description	Qty	Multiplier	Unit Cost	Cost
Digikey	FPGA chip	XC6SLX9-3TQG144C	1	5	$ 18.13	$ 90.65
Diligent	JTAG board	Programming Module	1	3	$ 54.00	$ 162.00
Digikey	SDRAM Chip	512M (64Mx8), 143MHz, 54-Pin TSOP II	1	3	$ 16.79	$ 50.37
Digikey	AC/DC wall adapter	12V, 18W, 1.5A	1	5	$ 11.45	$ 57.25
Digikey	Power Barrel connector	12VDC, 2A, through hole	2	5	$ 0.98	$ 9.80
Digikey	LED 1206 Blue	3.3V, 104mcd, 20mA, direction	4	5	$ 0.39	$ 7.80
Digikey	LED 1206 White	3.2V, 260mcd, 20mA, number application	10	5	$ 0.51	$ 25.50
Digikey	LED 0603 Red	2V, 54mcd, 20mA	1	5	$ 0.29	$ 1.45
Digikey	LED 0603 Green	2.1 V, 65mcd, 20mA	2	5	$ 0.41	$ 4.10
Digikey	Resistor network	100ohm, 8 element resistor bank	3	5	$ 1.09	$ 16.35
Digikey	Reset Momentary Button	SPST, .05A @ 12VDC	1	5	$ 0.63	$ 3.15
Digikey	Voltage Regulator 5Vo	Fixed 1 Output 5V 1A SOT-223	1	5	$ 0.46	$ 2.30
Digikey	Voltage Regulator 3.3Vo	Fixed 1 Output 3.3V 1A SOT-223	1	5	$ 0.46	$ 2.30
Digikey	Voltage Regulator 1.2Vo	Fixed 1 Output 1.2V 800mA SOT-223	1	5	$ 0.51	$ 2.55
Digikey	DB9 Serial Connector	Female, 9 pin, right angle PCB mount	1	5	$ 1.37	$ 6.85
Digikey	DB9 Serial Cable	Male-Male, 9 pin, 5ft, shielded	1	1	$ 8.33	$ 8.33
Digikey	Max232	TSSOP, Converts RS232 to Serial	1	5	$1.27	$ 6.35
Digikey	Crystal Oscillator	50MHz, SMD-5X3.2	1	5	$ 0.23	$ 1.15
Digikey	Capacitor	4.7uF, 10uF, .1uF	10	5	$ 0.21	$ 10.50
Digikey	Resistor	100ohm	2	5	$ 0.15	$ 1.50
Digikey	Resistor	10kohm	2	5	$ 0.15	$ 1.50
Silver Ci	Board manufacture	8.4" x5" max board, 4 layer max	1	5	$ 20.00	$ 100.00
Small Ba	Board part place		1	5	$ 25.00	$ 125.00
	Total		50			$ 696.75

9.2.2.2 Total Budget Breakdown

Figure 73: Costs Summary

Xilinx Inc. XC7K160T-2FFG676C (FPGA)	$373.10
JTAG-SMT2 Programming Module	$54.00
Surface-Mount DDR3 SDRAM	$2-$5
EEPROM	$2
External Power Supply (AC/DC) 5 V	$10
Miscellaneous Electronic Components (linear regular, resistors, capacitors)	$10-$30
PCB Manufacturing Service	$300 - $600

9.2.3 Sponsor

SoarTech has graciously chosen our team to sponsor and generously provided us with 1000 dollars. Their financial backing expanded our potential range of FPGA options and gave us some leeway when it came to our PCB size. Despite this, we decided to go with a low-cost, medium range chip to avoid going “bankrupt” should our PCB not work as planned after the first manufacturing run.

We haven’t developed a PCB schematic yet, which is why some of our parts still have ambiguous pricing. For example, the manufacturing cost greatly depends on the number of layers in our design. We will have to wrestle with using smaller, less expensive components while trying to maintain accurate speech recognition.

9.3 Team Composition

In order to maximize our success with our project. We have assigned each team member with the responsibilities that would be most appropriate. This was done with careful consideration of skill set, interests, experience, and also factored in the intensity of the learning curve.

The responsibility of Project Team Lead was assigned to Cedric Orban who had the skills and time necessary to take on the position. However, the whole team was involved in communications with the sponsor, faculty, as well as checking in with each other.

We breakdown the team’s primary responsibilities as follows

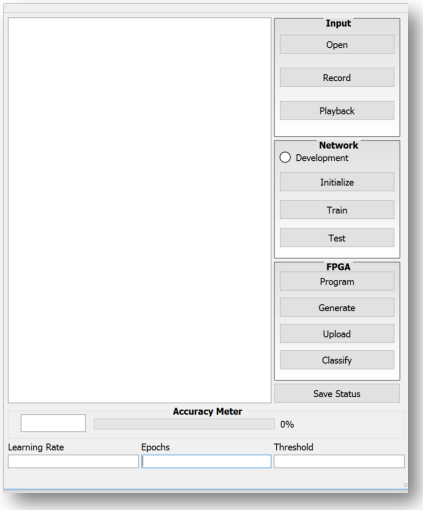
- Lindsay Davis
  - PCB Design
  - Hardware Interface
- Estella Gong
  - User Interfaces
  - Digital Communication
  - Systems Integration and Testing
- Michael Lopez-Brau
  - Pre-processing Algorithms
  - Deep Learning
- Cedric Orban
  - Team Project Lead
  - FPGA Firmware Programming

9.4 Project Operation

To ensure that the correct firmware is loaded on to the FPGA, first connect a USB 2.0 Micro-B type cable to the JTAG-SMT chip on the board. Then, using either Xilinx ISE or Diligent's Adept software package, upload the DeepGate programming file to the board. After a short time (less than 30 seconds) the status LED on the board will blink five times with a half-second period to notify the user that the FPGA is operational.

9.4.1 GUI Operation

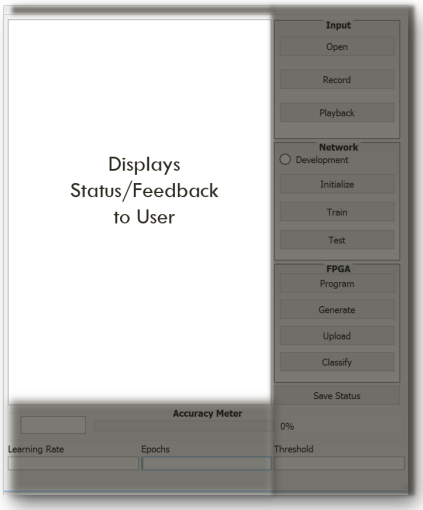
Figure 74: Graphical User Interface



The graphical user interface can be operated in the following manner:

- User

Figure 75: View Status

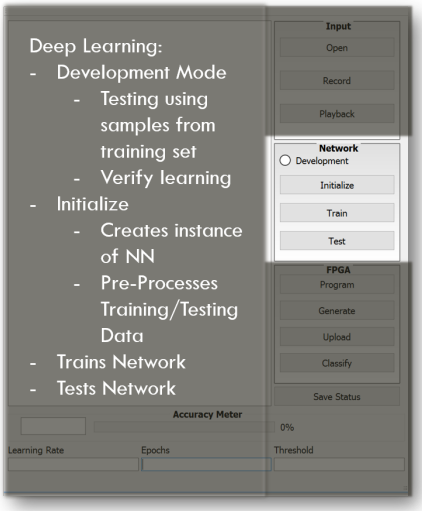


- User views status updates and can make notes via the edit box.

Figure 76: Input Controls

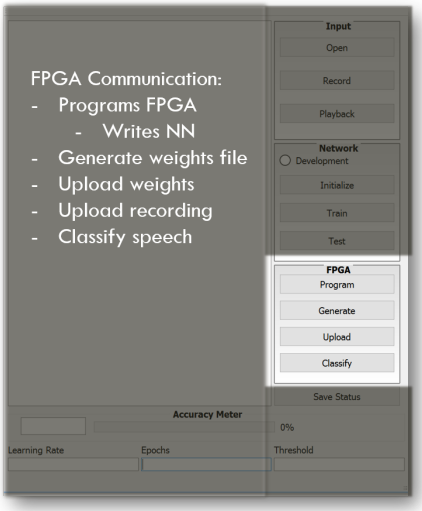
- User presses record. After a few seconds, the user will clearly state a number from 1-9 or north, south, east, west.
- The resulting wav file can be played back using the playback button and selecting the respective file.

Figure 77: Neural Network Controls



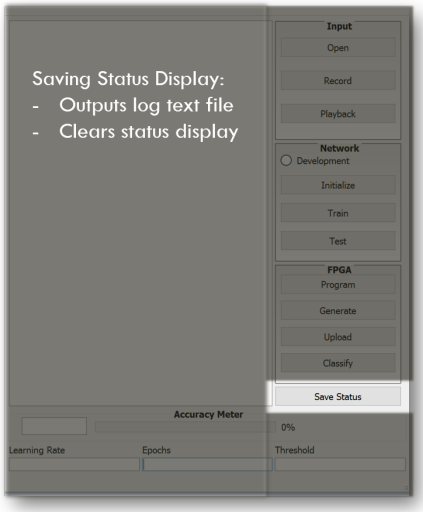
– User may verify the deep learning of the neural network by selecting development mode. This set is not necessary during the demo.
– User selects initialize to create instance of neural network and complete preprocessing
– The user may select train or test to either train or test the neural network but in the demo this is no longer necessary.

Figure 78: FPGA Controls



– Once the user has recorded speech and selected initialize, the user clicks on the Program FPGA button. This only needs to be done once per session. A session is defined as the powering up of the hardware.

Figure 79: Save Status/Logging

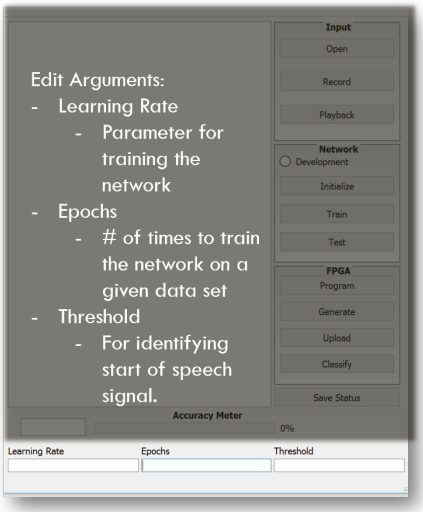


– User presses ”Save Status” to save all current status updates to a text file and clears the edit box.

Figure 80: Accuracy Meter

– User can view the recognized word and the performance percentage.

Figure 81: Edit Parameters



  - User can edit various parameters for the neural network.

- GUI Controller
  - Send speech data or other user input to the Raspberry Pi which communicates with the FPGA chip over SPI
  - Waits for the processing to finish
  - Receives feedback from the FPGA
  - Displays result to the user

9.5 Milestones

The milestones were developed for our project as we decided the range of requirements we were going to have. Considering that at least half of our team will be completely new to key components of our project – FPGA and Neural Networks, we have allotted more time for the extensive research that is needed for our team members to succeed.

The Milestone Chart will be routinely updated as we start creating our prototypes.

Figure 82: Milestones

Milestones	Sept.	Oct.	Nov.	Dec.	Jan.	Feb.	Mar.	Apr.
Research Phase								
Finalize Application Idea								
FPGA								
Machine Learning Methods								
PCB Design								
Embedded Programming								
Speech Recognition								
Design and Documentation Phase								
Application Interface								
PCB interfacing with FPGA								
PCB interfacing with Microphone								
PCB interfacing with other peripherals								
Acoustic Input								
Recognition Stage								
Implementation Phase								
Acoustic Hardware								
Acoustic Processing								
Application Software								
Recognition Stage Software								
Recognition Stage Hardware (PCB's etc.)								
Integration and Testing Phase								
Acoustic Integration								
Acoustic Testing								
PCB Integration								
PCB Testing								
FPGA Integration								
FPGA Testing								
Recognition/Machine Learning Testing								
Recognition Stage Integration								
Entire System Integration								
Entire System Testing								

10 Project Summary

To date, the team has been working diligently on each thrust of the project. On the hardware side, the components and our distributors have been selected and the PCB design has begun. On the software side, we have been testing various ANN and CNN configurations

10.1 Project Design Discussion

A Post-Mortem Analysis is conducted to reflect upon the work of this past semester. In this section we will review the roadblocks and challenges that we encountered. While these challenges also include the technical, we will also reflect on the administrative as well.

10.2 Technical and Administrative Challenges

The challenges we faced were largely due to inexperience. Therefore, it was very important for us to ensure that we plan for more research time. Two of our members had prior experience with FPGAs and neural networks. In order for the other members to ensure that we were all on the same page, understanding and empathy is key. We strive to be very open in sharing our knowledge and resources. It is also understood that it is the individual's responsibility to gain the information they needed.

10.3 Project Scheduling Update

We attempted to stick with our proposed milestone scheduling. However, as challenges arose during the semester it is clear that expected adjustments needed to be made. The table below delineates our milestone dates for our current semester's objectives.

Figure 83: Senior Design 1 Final Schedule

Scheduling	Days total	Start	End	Actual Start	Actual End
SENIOR DESIGN 1					
Research Phase					
Finalize Application Idea	30	08/01	10/01	08/01	09/09
FPGA	30	08/01	10/01	09/09	10/01
Machine Learning Methods	30	08/01	10/01	09/09	10/01
PCB Design	30	08/01	10/01	09/09	11/01
Embedded Programming	30	08/01	10/01	09/09	10/01
Speech Recognition	30	08/01	10/01	09/09	10/01
Design and Documentation Phase					
Application Interface	30	10/01	11/01	11/01	12/01
PCB interfacing with FPGA	60	10/01	12/01	11/01	12/01
PCB interfacing with other peripherals	60	10/01	12/01	11/01	12/01
Acoustic Input	30	10/01	11/01	10/01	11/01
Recognition Stage	60	10/01	12/01	11/01	12/01

The Table: Senior Design II Proposed Schedule shows our adjusted proposed timeline for our Implementation Phase, Integration and Testing Phase, and Refinement Phase.

Figure 84: Senior Design 2 Proposed Schedule

SENIOR DESIGN 2	Days total	Start	End	Actual Start	Actual End
Implementation Phase					
Acoustic Hardware/PC	1	12/20	12/21		
Acoustic Processing	20	01/09	01/29		
Application Software	24	01/23	02/06		
Recognition Stage Software	30	02/06	03/06		
Recognition Stage Hardware (PCB's etc.)	30	02/06	03/06		
Integration and Testing Phase					
Acoustic Integration	10	03/07	03/17		
Acoustic Testing	3	03/17	03/20		
PCB Integration	10	03/07	03/17		
PCB Testing	3	03/17	03/20		
FPGA Integration	7	03/10	03/17		
FPGA Testing	3	03/17	03/20		
Recognition/Machine Learning Testing	3	03/17	03/20		
Recognition Stage Integration	7	03/20	03/27		
Entire System Integration	7	03/27	04/03		
Entire System Testing	7	04/03	04/10		
Refinement Phase					
Software Refactoring (If necessary)	3	04/10	04/13		
Hardware Refinement	3	04/10	04/13		
Application Innovation	20	04/13	04/23		
Final Deployment Phase					
Verification of Working Parts	4	04/23	04/27		
Verification of Working Backup Parts	3	04/27	04/20		

10.4 Best Practices

Ideally for next semester, we want to ensure that meetings are more regular, including one-one-one check ins. We will also continuously update Dr. Lei Wei and sponsors on a weekly to bi-weekly basis to ensure that all anticipated roadblocks are resolved as quickly and efficiently as possible.

Currently our main method of contact is through Slack. Slack is a messaging app for teams that was very effective for our needs. We also communicate with each other by phone and email if necessary. To organize all our documents, we share a Google Drive folder. This folder contains our ideas, and different files, images, and resources that we can share.

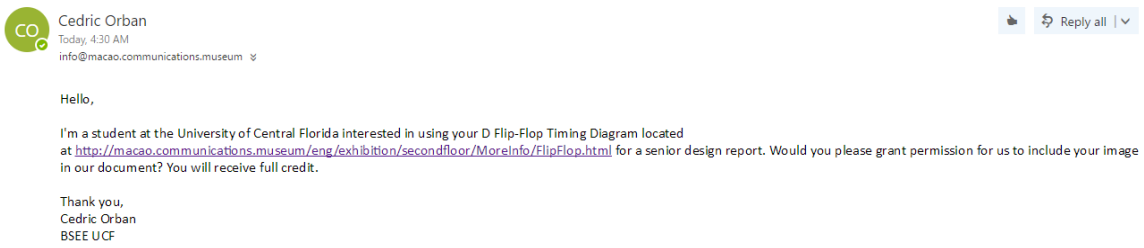
Our team has chosen to write our report and related through Overleaf. Overleaf is a real-time collaborative writing and publishing platform. It is primarily known as a platform for scientific writing. Overleaf utilizes the LaTeX typesetting system. It is primarily used for scientific communication and publication. These qualities make it ideal for our purposes.

A Copyright Permissions

Material in the Public Domain		
Figure	**Page**	**Citation**
3	11	Wikipedia. Retrieved November 10, 2016, from https://en.wikipedia.org/wiki/Flip-flop_(electronics)
7	16	Wikipedia. Retrieved November 10, 2016, from https://en.wikipedia.org/wiki/Logic_Block
18	34	Wikipedia. Retrieved November 12, 2016, from https://en.wikipedia.org/
32	53	DeepLearning. Retrieved December 6, 2016, from http://deeplearning.net/tutorial/lenet.html

Material Available Under Miscellaneous Licenses			
Figure	**Page**	**License**	**Citation**
10	19	GFDL	Wikipedia. Retrieved November 10, 2016, from https://en.wikipedia.org/wiki/Logic_block
14	24	CC	Stratus Engineering Retrieved November 15, 2016, from https://www.stratusengineering.com/rs232-9-pin-pinout/
15	26	CC 3.0	maxEmbedded Retrieved November 15, 2016, from http://maxembedded.com/

Figure 4 on page 12 has permission for inclusion in this document per the following email exchange.



Cedric Orban
Today, 4:30 AM
info@macao.communications.museum

Reply all

Hello,

I'm a student at the University of Central Florida interested in using your D Flip-Flop Timing Diagram located at http://macao.communications.museum/eng/exhibition/secondfloor/MoreInfo/FlipFlop.html for a senior design report. Would you please grant permission for us to include your image in our document? You will receive full credit.

Thank you,
Cedric Orban
BSEE UCF

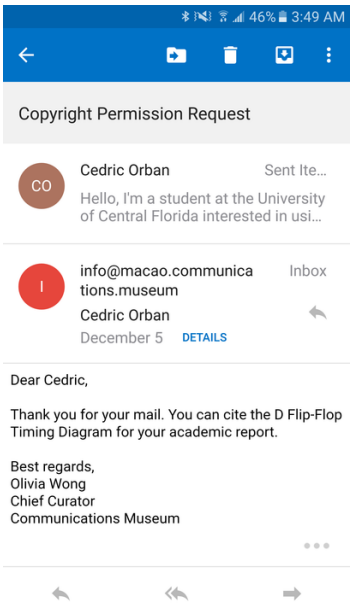


Figure 13 on page 23 is granted permission for inclusion in this document per the following exchange.

DEC 06, 2016 | 09:10AM MST

Nick M replied:

Hello-

You can use any images or text from the site that you'd like, as long as SparkFun is properly credited for all of the items you use. Otherwise, best of luck with your project!

Nick Miranda
SparkFun Electronics
Distributor and Customer Service
303-945-2984 x 607

Figure 8 on page 17 is permitted in this document per the following legal notice found at https://www.altera.com/about/legal.html.

Figures 5 on page 13 and 6 on page 14 have permission pending for inclusion in this document. Proof of a permission request is found below.

Type of Publication: Electronic product

E-prod Book Title:
E-prod Book ISBN:
E-prod Book Author:
E-prod Book Year:
E-prod Book Pages: to
E-prod Book Chapter number:
E-prod Book Chapter Title:
E-prod Video/DVD Title: Digital Design and Computer Architecture - Chapter 3 :: Sequential Logic Design Slides
E-prod Video/DVD ISBN:
E-prod Video/DVD Author: David Money Harris and Sarah L. Harris
E-prod Video/DVD Year: 2007?
E-prod Website Title:
E-prod Website URL:

I would like to use: Figure(s)
Quantity of material: 2 figures. The setup and hold time figures on slides 21 and 24 of the chapter slides, respectively.
Excerpts:
Are you the author of the Elsevier material? No
If not, is the Elsevier author involved? No
If yes, please provide details of how the Elsevier author is involved:
In what format will you use the material? Print and Electronic
Will you be translating the material? No
If yes, specify language:
Information about proposed use: Other
Proposed use text: Senior Design Project Report at the University of Central Florida
Additional Comments / Information: Ideally, we would like permission to use these images by December 6th, 2016. Thank you.

Kind regards,

Elsevier Permissions

Figure 16 on page 30 is granted permission for inclusion in this document per the following exchange. It is sourced from: https://embeddedmicro.com/tutorials/mojo/sdram

Re: Contact Form - Estella Gong

JR

Justin Rajewski <justin@embeddedmicro.com>
Tue 12/6/2016, 11:08 AM
Estella Gong

Reply all

Flag for follow up. Start by Tuesday, December 06, 2016. Due by Tuesday, December 06, 2016.

Hello Estella,

You can definitely use the diagrams in your report as long as you credit the page you got it from. Some kind of caption with a link or a pointer to your reference section with a link is sufficient.

Justin Rajewski

On Mon, Dec 5, 2016 at 6:34 AM Contact Form <contact@embeddedmicro.com> wrote:

> Name: Estella Gong
> Email: eg0131@knights.ucf.edu
> Telephone:
>
> Comment: Good morning,
>
> I am writing to inquire if it is at all possible to include some of your tutorial diagrams in our Senior Design Project Report at the University of Central Florida. This report will be in electronic and print forms. We will be more than glad to credit the image in the manner you prefer, including listing in our references, linking back to the blog, etc. Please let me know if this is at all possible at your earliest convenience. Thanks so much.

Figure 52, 53, and 54 on page 89 and 91, is pending permission for inclusion in this document per the following exchange.

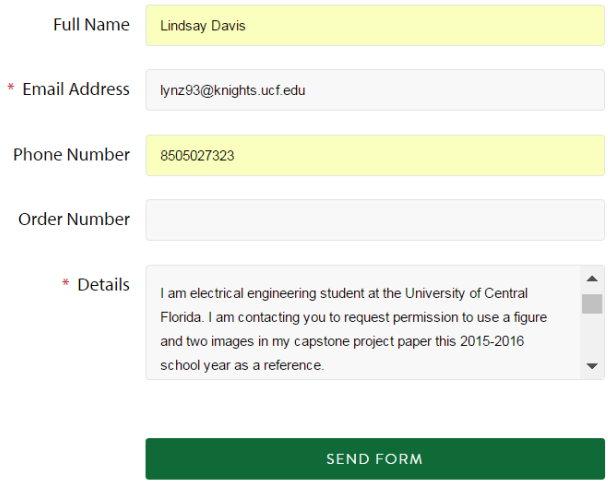


Figure 27 on page 45 is pending permission for inclusion in this document per the following exchange.

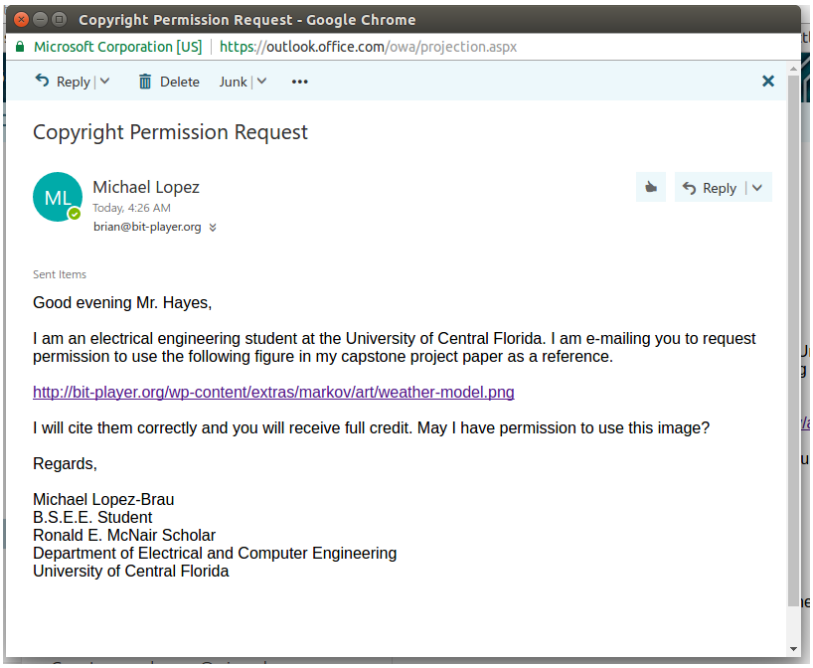


Figure 31 on page 50 is pending permission for inclusion in this document per the following exchange.

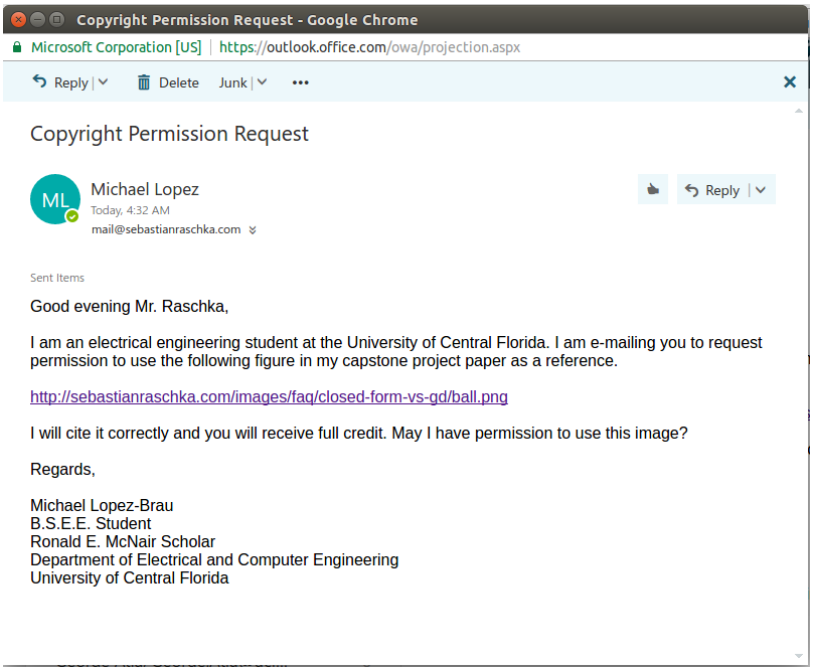
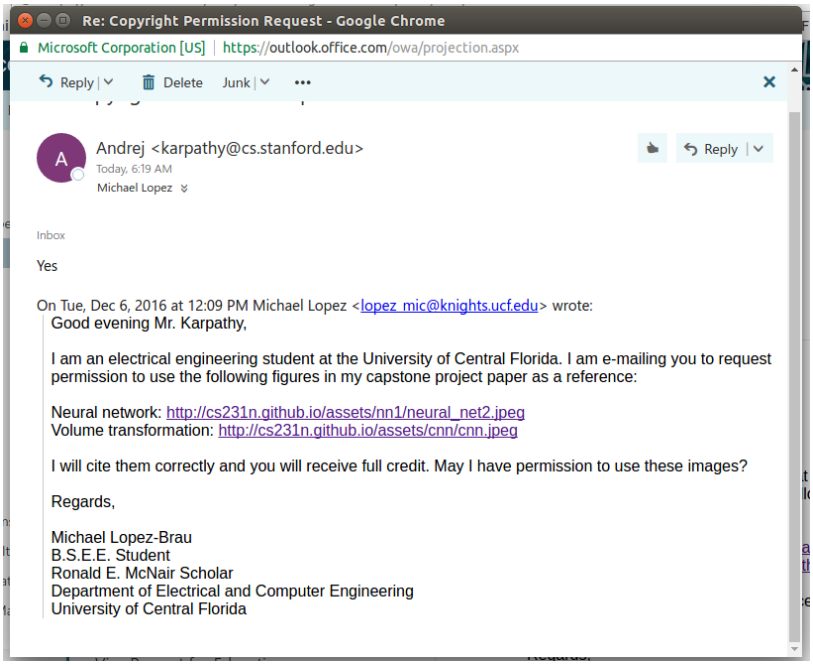


Figure 29 and Figure 33 on page 48 and page 53, respectively, have been approved for inclusion in this document per the following exchange.


Re: Copyright Permission Request - Google Chrome
Microsoft Corporation [US] | https://outlook.office.com/owa/projection.aspx
Reply | Delete Junk | •••
A
Andrej <karpathy@cs.stanford.edu>
Today, 6:19 AM
Michael Lopez
Reply |
Inbox
Yes
On Tue, Dec 6, 2016 at 12:09 PM Michael Lopez <lopez_mic@knights.ucf.edu> wrote:
Good evening Mr. Karpathy,
I am an electrical engineering student at the University of Central Florida. I am e-mailing you to request permission to use the following figures in my capstone project paper as a reference:
Neural network: http://cs231n.github.io/assets/nn1/neural_net2.jpeg
Volume transformation: http://cs231n.github.io/assets/cnn/cnn.jpeg
I will cite them correctly and you will receive full credit. May I have permission to use these images?
Regards,
Michael Lopez-Brau
B.S.E.E. Student
Ronald E. McNair Scholar
Department of Electrical and Computer Engineering
University of Central Florida

B References

[1] Axelson, J. (2007). Serial port complete: COM ports, USB virtual COM ports, and ports for embedded systems. Madison, WI: Lakeview Research.

[2] Beginning Electronics. (n.d.). Retrieved November 15, 2016, from https://embeddedmicro.com/tutorials/beginning-electronics

[3] Rabiner, L., & Juang, B. (1986). An introduction to hidden Markov models. ieee assp magazine, 3(1), 4-16.

[4] Caulfield, A. M., Chung., E. S., & Putnam, A. (2016, October). A Cloud-Scale Acceleration Architecture. Retrieved November 26, 2016, from https://www.microsoft.com/en-us/research/wp-content/uploads/2016/10/Cloud-Scale-Acceleration-Architecture.pdf.

[5] I2C Info – I2C Bus, Interface and Protocol. (n.d.). Retrieved November 24, 2016, from http://i2c.info/

[6] Linn, A. (2016). The moonshot that succeeded: How Bing and Azure are using an AI supercomputer in the cloud - Next at Microsoft. Retrieved November 06, 2016, from http://blogs.microsoft.com/next/2016/10/17/the_moonshot_-that_succeeded/

[7] Project Catapult - Microsoft Research. (n.d.). Retrieved November 10, 2016, from https://www.microsoft.com/en-us/research/project/project-catapult/

[8] Where FPGAs are fun. (n.d.). Retrieved November 05, 2016, from http://fpga-4fun.com/

[9] Park, J., & Sung, W. (2016). FPGA based implementation of deep neural networks using on-chip memory only. 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). doi:10.1109/-icassp.2016.7471828

[10] Collobert, R., Puhrsch, C., & Synnaeve, G. (2016). Wav2Letter: an End-to-End ConvNet-based Speech Recognition System. arXiv preprint arXiv:1609.-03193.

[11] Tommiska, M. (2003). Efficient digital implementation of the sigmoid function for reprogrammable logic. IEE Proceedings - Computers and Digital Techniques, 150(6), 403. doi:10.1049/ip-cdt:20030965

[12] Moerland, P. D., & Fiesler, E. (n.d.). Neural network adaptations to hardware implementations. Handbook of Neural Computation. doi:10.1887-/07503031-23/b365c78

[13] Anwar, S., Hwang, K., & Sung, W. (2015). Fixed point optimization of deep convolutional neural networks for object recognition. 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). doi:10.1-109/icassp.2015.7178146

[14] Harris, D. M., & Harris, S. L. (2007). Digital Design and Computer Architecture. Amsterdam: Morgan Kaufmann.

[15] Spartan-6 Family Overview DS160 (v2.0) [Pdf]. (2011, October 25). Xilinx Inc.

[16] Spartan-6 FPGA Packaging and Pinouts User Guide UG385 (v2.3). (2014, May 12). Xilinx Inc.

[17] Spartan-6 FPGA Block RAM Resources User Guide UG383 (v1.5) [Pdf]. (2011, July 8). Xilinx Inc.

[18] Spartan-6 FPGA Clocking Resources User Guide UG382 (v1.10) [Pdf]. (2015, June 19). Xilinx Inc.

[19] ”How to Connect Speakers and Microphones to a Computer.” How to Connect Speakers and Microphones to a Computer. N.p., n.d. Web. 06 Dec. 2016.

[20] Court, 1300 Henley, Wa 99163 Pullman, 509.334.6306, and www.digilentinc.c-om. JTAG-SMTTM Programming Module for Xilinx ® FPGAs (n.d.): n. pag. Web.

[21] D-sub 9 Connector Pinout. (n.d.). Retrieved December 06, 2016, from http://w-ww.db9-pinout.com/

[22] ”Xilinx Spartan-3 Specific Memory.” FPGA Prototyping by VHDL Examples (n.d.): 243-56. Web.

[23] Xilinx, Inc. Xilinx UG393 Spartan-6 FPGA PCB Design Guide (n.d.): n. pag. Web.

[24] Xilinx, Inc. Spartan-6 FPGA SelectIO Resources User Guide (UG381) (n.d.): n. pag. Web.