The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

# ECE532 Group Report: Speech Recognition via FPGA Network

Atharva Datar

Dan Nicolau

Defne Dilbaz

14 April 2022

**Table of Contents**


**1. Overview** **3**
1.1 Motivation 3
1.2 Goals 3
1.3 Block Diagram 5
1.4 Code Sources 6

**2. Outcome** **8**
2.1. Feature Status 8
2.2. Changes to Project and Method 8
2.2.1 Changes to Speech Recognition Algorithm 8
2.2.2. Changes to Hardware Implementation of Current Design 9
2.3. Next Steps 10

**3. Project Schedule** **11**
3.1 Milestone 1 11
3.2 Milestone 2 12
3.3 Milestone 3 12
3.4 Milestone 4 13
3.5 Milestone 5 14
3.6 Milestone 6 15
3.7 Final Demonstration 15

**4. Description of Blocks** **16**
4.1 Desktop Client 16
4.1.1 Python TCP Client 16
4.2 Custom IPs - Load Manager FPGA 17
4.2.1 Load Manager 17
4.3 Custom IPs - Compute FPGA 18
4.3.1 Word Clipper 19
4.3.2 Fast Fourier Transform 20
4.3.3 Euclidean Distance 22
4.4 Xilinx IPs 23
4.4.1 Clocking Wizard 23
4.4.2 Processor System Reset 23
4.4.3 AXI Interconnect 23
4.4.4 Concat 23
4.4.5 AXI SmartConnect 23
4.4.6 AXI Timer 24
4.4.7 AXI Interrupt Controller 24

4.4.8 Microblaze Debug Module 24
4.4.9 Memory Interface Generator 24
4.4.10 AXI Uartlite 24
4.4.11 AXI Ethernetlite 24
4.4.12 Ethernet PHY MII to Reduced MII 24
4.4.13 AXI TFT Controller 25
4.4.14 Slice 25
4.4.15 Integrated Logic Analyzer 25
4.4.16 Microblaze 25

**5. Description of Design Tree 26**

**6. Tips and Tricks 28**

**7. Video 29**

**8. References 30**

**9. Appendices 33**

# 1. Overview

## 1.1 Motivation

Speech recognition is a popular technology that is used in digital personal devices, devices catered towards hearing-impaired individuals, hands-free technology, etc [1]. In the context of this project, speech recognition refers to receiving an array of English words and outputting matched words in the sentence. Therefore, it involves audio processing and computation. The basics of speech recognition computation can be done using Direct Fourier Transform (DFT) or Fast Fourier Transform (FFT) and computing Euclidean distances[2][3].

Speech recognition using DFT/ FFT and Euclidean distances is a relatively simple software project[4]. The speech recognition widgets implemented on our phones, computers, and any personal technological devices are also implemented in software. However, translating speech recognition technology into hardware is a creative challenge the team wants to take on.

There are existing speech recognition projects implemented on FPGA[5][6] that we have looked into. Some of these examples had several advantages. Our primary research has shown that Hidden Markov Models are much better at word recognition than DFT/FFT[7][8]. Given that DFT[9]/FFT[10][11] IP already exists in Vivado, and Hidden Markov Models is a very complicated algorithm; we wanted to focus on improving the disadvantages of speech recognition projects which relied on DFT/ FFT. Most of these projects used a single FPGA. However, this does not translate to a real life use case. We believe an FPGA speech recognition project must take into consideration that there cannot be a single FPGA for each instance where speech recognition might be needed. Instead, we decided that our project would be transcribing for multiple speech recognition requests at the same time. Furthermore, the highest number of words used in these examples was 2, which we wanted to improve upon.

## 1.2 Goals

We want to build an architecture where a single FPGA receives requests from multiple desktops, distributes these requests to several FPGAs which match audio to words, and returns the matched words to the central FPGA, to be output to a VGA display. While doing so, we want the FPGAs tasked with speech recognition to be able to recognize a good selection of words. Therefore, our project goal is to implement a speech recognition system that is able to cater multiple requests at once and has a larger selection of words it can identify than existing speech recognition implementations on FPGA.

We have provided Table 1 listing our project goals in table format and their status at the end of the project.

Table 1: Project Goals and Status

| **Project Goal** | **Status** |
|---|---|
| Implement new audio processing ip's in hardware, or utilize existing ones (i.e. FFT) | FULLY ACHIEVED: We made a new ip block which uses absolute_value, moving_average, and word_clipper. We used FFT ip, and we made our own euclidean distance ip. |
| Handle multiple speech-to-text identification requests at the same time | FULLY ACHIEVED: We have 3 separate desktops sending requests at the same time, and 3 compute FPGA's identifying words from these 3 separate text files |
| Improve upon the dictionary size | PARTIALLY ACHIEVED: We were able to try out with a dictionary size of 3 words, which is better than hardware examples online (2 words) but still relatively small improvement. |
| Decrease reliance on software | ACHIEVED: Unlike other hardware examples, we do preprocessing, FFT, and Euclidean Distance in hardware. |
| Use FFT instead of more complicated algorithms (Markov Models, neural nets) for better logic/ space usage | ACHIEVED: We used Vivado FFT IP. |

## 1.3 Block Diagram

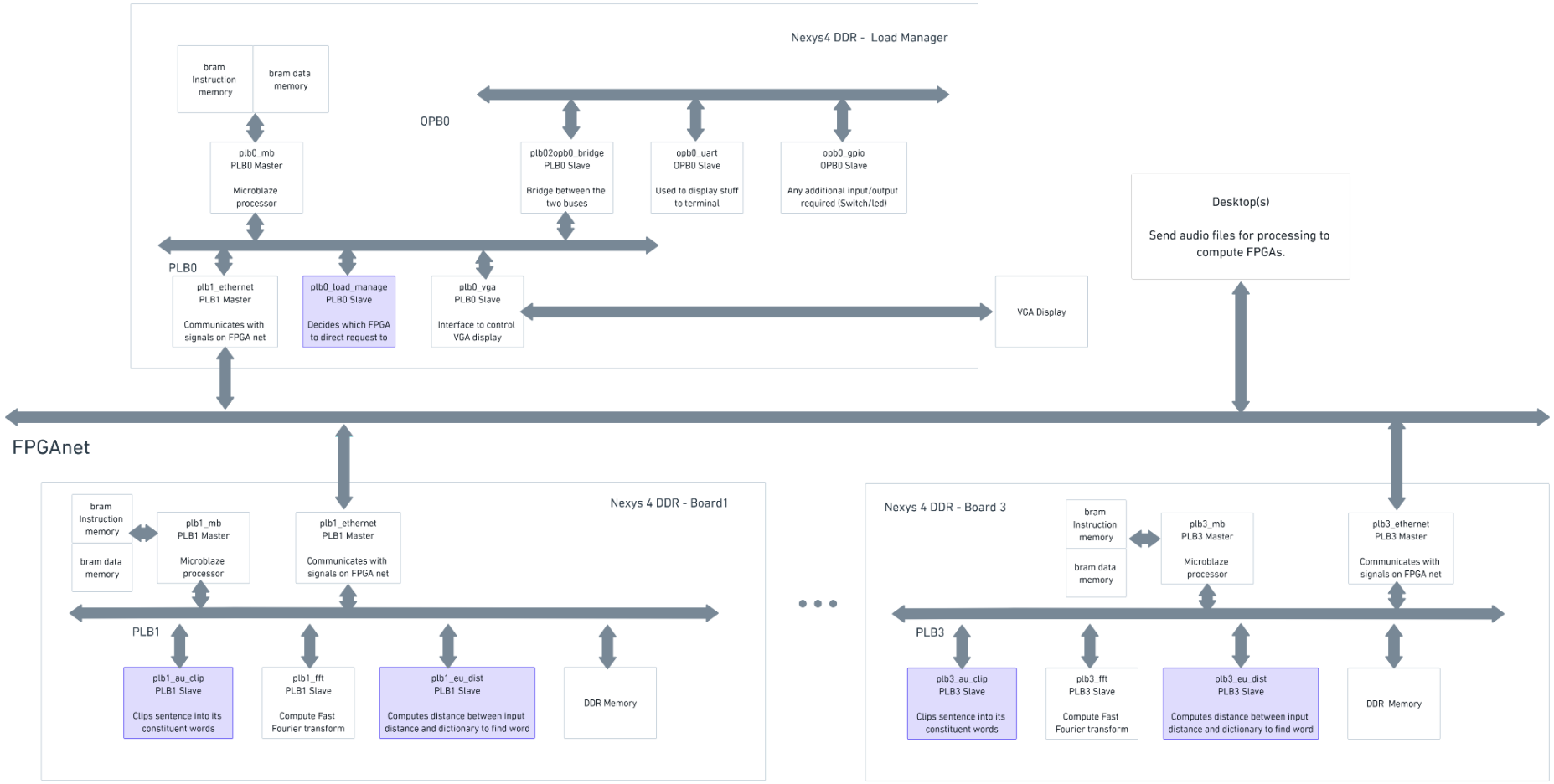



Figure 1: Block Diagram of Proposed Design

### 1.4 Code Sources

Throughout the project, we have written software and hardware code, adapted existing work, or reused IP's. In table 2, we are listing the components of our project, a brief description about them, and their sources.

Table 2: List of components, their descriptions and sources

| **Component** | **Description** | **Source** |
|---|---|---|
| Ethernet (client & server) | Ethernet communication between desktops and load manager / compute FPGAs | Adapted from ECE532 Tutorial 3 |
| FFT | Fast fourier transform IP | Vivado IP [10] |
| Absolute value, moving average, word clipper | An IP block for audio processing which finds the start and end indices of a word in an audio file | Custom IP made by team |
| Load manager | An IP block for managing multiple requests from desktops and assigning them to compute FPGAs in round-robin order | Custom IP made by team |
| Framing | After the boundaries of a word are found within a sentence (i.e. we have a time window for the word), we must split the window into several overlapping frames, with each frame passed to the FFT module for processing. | Custom software made by team |
| Euclidean Distance | A module for calculating the square of the difference between two signals | Custom hardware made by team |
| VGA | An IP block to display on VGA | Adapted from Misc. Tutorial on TFT Controller |
| Speech recognition Python Implementation | Speech recognition implementation in Python | Custom software made by team |

| | | |
|---|---|---|
| Dictionary generation | Software to generate audio files for dictionary words | Custom software made by team |

# 2. Outcome

## 2.1. Feature Status

Table 3: Feature status

| Feature | Completed | Comments |
|---|---|---|
| Word Boundaries | Yes | Word boundaries are detected, but thresholds are not robust enough to survive ambient noise and still detect words accurately. |
| Word Recognition | Yes | |
| SD Card Dictionary | No | |
| VGA Output | Yes | There is an issue where the VGA output will jump to the right by a few dozen pixels randomly. |
| SDK Terminal Output | Yes | |
| FPGA-FPGA Communication | Yes | |
| Desktop-FPGA Communication | Yes | |
| Load Management | Yes | |

## 2.2. Changes to Project and Method

### 2.2.1 Changes to Speech Recognition Algorithm

The speech recognition algorithm that we selected based on its ease of implementation in hardware is great for identifying speakers but underperforms when it comes to detecting words. We propose some improvements that are likely to improve the accuracy of the speech recognition algorithm, which would also enable use of a larger dictionary. This section would also apply to the software model.

Table 4: Suggested Improvements to Speech Recognition Algorithm

| Improvement | Method |
|---|---|
| Normalize word length | Map word to a 1-2 second, standard length, recording. |
| Normalize word frequency (speakers who communicate at a | Store relative deviation from base talking frequency from FFT instead of absolute frequency amplitudes. |

| | |
|---|---|
| higher or lower pitch than the dataset can be misinterpreted) | |
| Adaptive word detection threshold generation | Use a rolling average to detect a baseline envelope amplitude, then identify words based on a minimum length of activity. |
| Moving Average Alternative | Moving Average uses considerably more hardware resources than other envelope generation strategies. A "leaky integrator" may be more resource efficient while still providing similar results. |
| Account for lag of envelope generation method. | Moving averages or leaky integrators both have a lag associated with them. The word boundaries are affected by this lag. Adjusting the word boundaries to account for this lag will add more useful word data to the overall stored data. |
| Use finer framing and windowing using Hamming windows (if computationally feasible) | Using Hamming windows for frames and windows may remove some of the error from further discretizing our audio signal into a matrix of features. |
| Maximum Cost Threshold | If the minimum cost between a test word and all test words is still above a threshold it is likely the word is just noise, or not in the dictionary and can either be not reported or reported as not relevant. |
| Use variable width windows | Frequencies are not perceived on a linear scale. The window width should increase accordingly (however this may not be reasonable in hardware). |

### 2.2.2. Changes to Hardware Implementation of Current Design

The above table lists many computational changes that would create significant changes in the current design. This section will focus on changes that we would make to the current design with the current speech recognition algorithm.

Table 5: Suggested Improvements to Hardware Implementation

| **Change** | **Comments** |
|---|---|
| Fully Pipelined Hardware Implementation of Algorithm | The current design uses a lot of communication on a single AXI bus. This was largely due to the course emphasis on using the AXI bus, the ease of manipulating data with the microblaze processor, and the security of using software as a fallback (in case the hardware did not work). |

| | |
|---|---|
| | In theory, the computation of a single matrix element can be pipelined to increase performance and remove the bottleneck on the AXI bus. |
| Use of Video Board | The video board has additional resources which may have been used to increase the efficiency or accuracy of the algorithm, especially with the proposed changes to the algorithm. |
| Use run-time programmable registers for some algorithm parameters. | Some algorithm parameters use a constant such as the upper and lower thresholds. Mapping these registers to memory and editing these parameters at run-time using software would have saved a few hours of development time. |

### 2.3. Next Steps

The next steps for this project would be implementing the above changes, implementing SD card support, and including a microphone on the FPGA for real-time data input – which may also help with getting an intuitive understanding of which words get mixed up, and how pronunciation affects accuracy.

## 3. Project Schedule

In this section, we highlight the differences between the proposed milestones in our progress report with the realized progress. There are some important aspects we should highlight.

As a risk mitigation strategy, we left the last two milestones to broadly cover "integration" in case we would not accomplish prior milestones. We ended up resorting to this strategy, as our milestones until then were not fully completed. Furthermore, it is important to note that we made considerable work after the last milestone and between final demonstration, as integration was not complete by the end of Milestone 6.

Reasons why our progress stalled at times include smaller level integration problems we encountered. For example, while testing two modules which work consecutively, we ran into issues we did not face with modules individually. This was mostly noticed when we used audio files as inputs, instead of data from testbenches. In addition, certain weeks had higher course workload and affected our performance.

In the following sections, we give an overview of our progress on a milestone basis. Then, we explain the differences between our original plan and our actual progress through a table format.

### 3.1 Milestone 1

Our approach with milestone 1 was to keep it fairly simple and focus on research. Therefore, we were able to accomplish our tasks.

Table 6: Milestone 1 Plan vs Status

| | **Tasks** | **Status** |
|---|---|---|
| Tasks originally planned & accomplished | Research python implementation of FFT/DFT & Euclidean distance based word recognition | Completed |
| | Research sending audio files over ethernet. | Completed |
| | Research available FFT/DFT IPs on Vivado, audio filtering algorithms | Completed |
| | Research into the dictionary of words to have | Completed |

### 3.2 Milestone 2

In milestone 2, we started working on the software model which set us up to changes we needed to do in the hardware model. We also realized there were some secondary tasks originally overlooked, such as IBM Watson API, and completed them.

Table 7: Milestone 2 Plan vs Status

| | **Tasks** | **Status** |
|---|---|---|
| Tasks originally planned & accomplished | Test audio clipping on software | Completed |
| | Research load management & Desktop-to-FPGA communication | Completed |
| Tasks originally planned but not completed | Finalize python implementation of FFT/DFT & Euclidean distance based word recognition | Mostly completed, needed some additional work |
| Tasks originally not planned but progress made | Research python implementation of FFT/DFT & Euclidean distance based word recognition | Completed |
| | IBM Watson API | Completed |
| | Further research & implementation on FFT IP | Completed |

### 3.3 Milestone 3

Our software implementation in Milestone 2 made us reprioritize certain tasks. Therefore, certain tasks were pushed to milestone 4. To compensate for the lack of work, we added new tasks that we mostly completed.

Table 8: Milestone 3 Plan vs Status

| | **Tasks** | **Status** |
|---|---|---|
| Tasks originally planned & accomplished | Start Euclidean distance based word recognition on FPGA | Completed but we divided it into word clipping, convolution, and framing |
| | Start FFT/DFT implementation on FPGA | Was completed in milestone 1 |
| | Start Desktop-to-FPGA communication | Completed but we changed it to “Multiple Desktop-to-FPGA communication” |

| | | |
|---|---|---|
| Tasks originally planned but postponed to later milestone | Start a dictionary of words | Pushed to milestone 4 |
| | Start VGA output | Pushed to milestone 4 |
| Tasks originally not planned but progress made | Speech recognition architecture modularization & RTL write-up - Complete Word Clipping IP | Completed |
| | Research into memory management | Completed |
| | Multiple Desktop-to-FPGA communication | Completed |
| | Speech recognition architecture modularization & RTL write-up - convolution | Started, not completed |
| | Complete Framing module RTL | Started, not completed |

### 3.4 Milestone 4

In this milestone, we started facing some issues with integration, which resulted in us pushing VGA display to Milestone 5. This was a particularly busy week coursework-wise, therefore progress was not at a level that we expected.

Table 9: Milestone 4 Plan vs Status

| | Tasks | Status |
|---|---|---|
| Tasks originally planned & accomplished | Start FPGA-to-FPGA communication | Completed for DDR boards |
| | Start load management | Completed |
| Tasks originally planned but postponed to later milestone | Identify a word from its FFT/DFT | Incomplete - word clipping, absolute value, moving average, and framing modules were individually done but not integrated with FFT |
| | Test VGA display output | Postponed to Milestone 5 |

| | | |
|---|---|---|
| Tasks originally not planned but progress made | Start a dictionary of words | Completed |
| | Word Clipping + Absolute Value Modules and Testbenches | Completed |
| | Integrator module (Similar to FIR) | Completed |
| | Update Software Model to use 16 bit integers and adjust parameters accordingly | Incomplete |
| | Integration of completed submodules | Completed |
| | Start memory management | Incomplete |
| | Complete implementation testing and integration for the Framing module | Completed |

### 3.5 Milestone 5

In milestone 5, we expected we would have issues with integration. Though certain tasks were incomplete, we had originally planned to leave Milestone 6 for further integration as a safety strategy.

Table 10: Milestone 5 Plan vs Status

| | Tasks | Status |
|---|---|---|
| Tasks originally planned & accomplished | Test load management | Completed |
| Tasks originally planned but postponed to later milestone | Integrate Desktop-to-FPGA, FPGA-to-FPGA, DFT/FTT/ Euclidean distance to identify one word | Postponed to final demonstration |
| Tasks originally not planned but progress made | SD Card | Incomplete |
| | Update Software Model to use 16 bit ints | Incomplete |
| | VGA output | Incomplete |
| | Euclidean Distance Module | Research done |
| | Print on the terminal something more meaningful | Completed |
| | Word framing and FFT module usage | Incomplete |

### 3.6 Milestone 6

In milestone 6, we did not complete all integration. Certain important components, SD card and VGA, were still not working. Therefore, we had to work for another week until the final demonstration to complete our project.

Table 11: Milestone 6 Plan vs Status

| | Tasks | Status |
|---|---|---|
| Tasks originally planned but postponed to later milestone | Integrate all components together | Postponed to final demonstration |
| Tasks originally not planned but progress made | Finish Framing | Complete |
| | Finish FFT | Complete |
| | FPGA-FPGA communication | Incomplete |
| | VGA Output | Incomplete |
| | FFT and window integral | Complete |
| | SD Card Wrapper/ FSM | Incomplete |
| | Euclidean distance | Complete |
| | Euclidean Distance Comparator | Complete |

### 3.7 Final Demonstration

After Milestone 6, we decided to scrap the SD card as we had Video board in hand and we had many things to debug already. This week can be characterized as merging all components together and finalizing the VGA display.

## 4. Description of Blocks

As seen in Figure 1, the project consists of 3 main components: desktops, which send speech recognition requests; load manager, which receives requests from desktops and assigns them to compute FPGAs; and compute FPGAs, which receive audio files and output the words identified. In the following sections, we give detailed information about these 3 components.

### 4.1 Desktop Client

The desktop acts as the client in our system and represents the entity that initiates communication and begins the process of speech recognition. For our project we store the raw speech that needs to be processed (or "recognized"), in the form of audio files on the desktop. These audio files are stored in WAV file format and are transferred to the compute FPGAs via ethernet. For this purpose we set up a TCP client on the desktop using a Python script. A stepwise description of this is provided below.

#### 4.1.1 Python TCP Client

1. The script begins by sending a "compute request" (encoded by ASCII character '1') to the load manage FPGA. The desktop client also sends its own IP address to the load manage FPGA for its own record keeping. To this request the FPGA replies with the IP address of a compute FPGA to which the data must be sent. The desktop client now uses this new IP address for all further communication (Appendix A).
2. Once the desktop client receives the IP address of the compute FPGA, it begins pre-processing the WAV file to prepare the data for transfer. Pre-processing involves computing the file size and removing the file header before beginning data transfer (Appendix B).
3. Next, we open an ethernet connection between the desktop client and the compute FPGA (Appendix C).
4. After the connection has been established, we send a "new data" request (encoded by ASCII character '2') to the compute FPGA. Once the compute FPGA acknowledges this request we know it is ready for data transfer (Appendix D).
5. Next we begin transferring the WAV file to the compute FPGA in ethernet packets of size 1460 bytes which is the maximum size allowed for a single packet (Appendix E).

6. After we have transferred the entire file, we send an “end data” request (encoded by ASCII character ‘3’) to the compute FPGA indicating end of data transfer (Appendix F).
7. Once we receive an acknowledgement from the compute FPGA for the end data request, we send a “start compute” request (encoded by ASCII character ‘4’) to indicate to the FPGA to start computing (Appendix G).
8. Once the compute FPGA finishes processing the audio file, it sends an acknowledge signal to the desktop client. At this point, we have finished all communication, and so we proceed with closing the ethernet connection between the desktop client and compute FPGA (Appendix H).

### 4.2 Custom IPs - Load Manager FPGA

The load manager FPGA acts as a central server in our system and represents the entity that handles all incoming requests from clients of the system. The primary function of the load manager is to ensure that incoming requests are equally distributed amongst all available compute FPGA so as to prevent any one FPGA from being overwhelmed with requests. For this purpose, we implemented a custom load manage IP that is described below.

#### 4.2.1 Load Manager

Description

- The load manager IP is used to assign a particular compute FPGA to incoming requests from desktop clients.
- The main motivation behind this IP is the need to be able to effectively manage incoming requests such that load is equally distributed amongst the available compute FPGAs
- For our project we used a round robin algorithm. Incoming requests are assigned to the compute nodes one after the other, irrespective of the size of the request (i.e. size of WAV file to be processed).
- Once every available compute FPGA has been assigned a request, the next request that comes in will be assigned back to the first compute FPGA and so on.
- Currently the system handles up to 3 compute nodes. RTL as well as software changes will have to be implemented to handle a larger number of compute nodes.

- The IP address of the 3 compute FPGAs are stored in the load manage IP at system startup and initialization (Appendix I)

Inputs

1. **in_1**: IP Address (i.e. Lab Station no.) of first compute FPGA (stored in slv_reg0)
2. **in_2**: IP Address (i.e. Lab Station no.) of second compute FPGA (stored in slv_reg1)
3. **in_3**: IP Address (i.e. Lab Station no.) of third compute FPGA (stored in slv_reg2)

Outputs

1. **out**: IP Address (i.e. Lab Station no.) of compute FPGA to be sent to client (stored in slv_reg7)

It is important to note that these inputs and outputs are not visible at the top level but are rather accessed through slave registers compatible with the AXI-Lite protocol. These slave registers are written to and read from using the software drivers automatically generated by the Xilinx SDK.

Parameters

Load manage IP is not parameterizable.

Compatibility

AXI- Lite Compatible

### 4.3 Custom IPs - Compute FPGA

The compute FPGA acts as a processing node in our system. Its main role is to process the data it receives from the desktop client in order to identify the words present in the audio file and then to send the results of its computation to the load manager FPGA to be displayed as the system's output. The processing of the audio data is done in 3 main steps. First step is identifying the word boundaries in a sentence. The next step is computing the fast fourier transform (FFT) of each word. The last step is finding the Euclidean distance between every word in our dictionary and the provided word to perform speech recognition. Each of these steps was completed primarily using custom IPs the team designed, the description of which is given below.

### 4.3.1 Word Clipper

Description

- This IP consists of 3 independent modules connected to each other via custom interfaces.
- The first module is an absolute value module. Given a 2's complement input number it produces the absolute value of the number as output
- The output of the absolute value module is fed into the moving average module. This module calculates the moving average of N inputs. The number N can be controlled by the user.
- The output of the moving average is fed into a word clipper module. The word clipper module will determine the boundaries of the word.
- The output of the word clipper is the output of the entire IP.

Inputs

1. **in_data**: a 16-bit value that represents a single sample from the WAV file, read from DDR memory (stored in slv_reg0)
2. **in_addr**: a 32-bit value representing the address of the location in DDR memory where in_data was read from (stored in slv_reg1)
3. **in_valid**: a 1-bit value that is asserted when input to the module is valid. Stays high just for one clock cycle (stored in slv_reg2)
4. **in_last**: a 1-bit value that is asserted when the last data sample of the WAV file has been read from memory (stored in slv_reg3)
5. **in_ack**: a 1-bit value that is asserted when the system acknowledges that the module's output has been received. Stays high just for one clock cycle (stored in slv_reg4)

Outputs

1. **out_word_start**: a 32-bit value that represents an address in DDR memory where the current word starts (stored in slv_reg14)
2. **out_word_end**: a 32-bit value that represents an address in DDR memory where the current word ends (stored in slv_reg15)
3. **out_valid**: a 1-bit value that is asserted when word_start and word_end are valid. Stays high till the in_ack signal is received from the system (stored in slv_reg12)

4. **out_done**: a 1-bit value that is asserted when the module is finished processing the entire WAV file (stored in slv_reg13)

Similar to the load manage custom IP these inputs and outputs are not visible at the top level but are rather accessed through slave registers compatible with the AXI-Lite protocol.

Parameters

1. **Sample Width**: Controls the bit width of data samples. Possible values: 4, 8, 16, 32, 64
2. **Sample Num**: Controls the number of 16-bit samples to be included in the moving average computation. Possible values are multiple of 2 and can range from 2 to 8192
3. **Word Clip Lower Threshold**: Controls the lower threshold value to be used in determining word boundaries. Possible value is any 16-bit integer
4. **Word Clip Upper Threshold**: Controls the upper threshold value to be used in determining word boundaries. Possible value is any 16-bit integer

Compatibility

AXI-Lite Compatible

### 4.3.2 Fast Fourier Transform

Description

- To compute fast fourier transform we made use of the Xilinx FFT IP [10]. However, since the Xilinx IP was only AXI Stream compatible we had to create a top level module that maps AXI Lite signals to AXI Stream, so that we could make it compatible with our system.
- Version 9.1 of the Xilinx FFT IP was used
- Input to the FFT module was frames of the WAV file that were determined through software. Each word was split into 64 overlapping frames, and FFT was computed for each frame
- Output of the FFT module was also divided into 64 frames, although this time with non-overlapping frames. The data in each frame was summed together and this sum became one element in the matrix representation of the word.

- Thus each word was represented by a 64x64 matrix with 32-bit elements making the size of 1 word 16 kilo-bytes

Inputs

1. **xin**: a 16-bit value that represents a single sample from the WAV file, read from DDR memory (stored in slv_reg0)
2. **xvalid**: a 1-bit value that is asserted when input to the module is valid. Stays high just for one clock cycle (stored in slv_reg1)
3. **xlast**: a 1-bit value that is asserted when the last data sample of the WAV file has been read from memory (stored in slv_reg2)
4. **fft_ready**: a 1-bit value that is asserted when we are ready to read the output of the FFT module. Stays high for just one clock cycle (stored in slv_reg15)

Outputs

1. **xready**: a 1-bit value, when asserted, indicates to us that the FFT module is ready to accept new data (stored in slv_reg3)
2. **fft_out**: a 16-bit value generated by the FFT module as output (stored in slv_reg14)
3. **fft_valid**: a 1-bit value indicating that the current value of fft_out is a valid output (stored in slv_reg13)
4. **fft_last**: a 1-bit value that is asserted when the fft module produces its last valid output (stored in slv_reg12)

Similar to the previous custom IP these inputs and outputs are not visible at the top level but are rather accessed through slave registers compatible with the AXI-Lite protocol.

Parameters

Fast Fourier Transform IP is not parameterizable.

Compatibility

AXI-Lite Compatible

### 4.3.3 Euclidean Distance

Description

- The Euclidean Distance is used to compute the square between the difference of two 32-bit input values.
- This module does not sum and then squareroot the square of differences. Summation is handled in software.
- As input values are 32-bits, this module uses many logical elements. Initially, the module was failing timing therefore, it was pipelined and summation was removed.

Inputs

- **idata_0:** a 32-bit value representing one sample from processed audio file (stored in slv_reg0)
- **idata_1:** a 32-bit value representing one sample from processed audio file (stored in slv_reg1)
- **ivalid:** a 1-bit value that is asserted when input to the module is valid. Stays high just for one clock cycle (stored in slv_reg2)
- **iack:** a 1-bit value that is asserted when the system acknowledges that the module's output has been received. Stays high just for one clock cycle (stored in slv_reg3)

Outputs

- **ovalid:** a 1-bit value that is asserted when word_start and word_end are valid. Stays high till the in_ack signal is received from the system (stored in slv_reg13)
- **odata:** a 64-bit value which is the square of the difference between idata_0 and idata_1 (stored in slv_reg14 and slv_reg15)

Similar to the previous custom IP these inputs and outputs are not visible at the top level but are rather accessed through slave registers compatible with the AXI-Lite protocol.

Parameters

Euclidean Distance IP is not parameterizable.

Compatibility

AXI-Lite Compatible

### 4.4 Xilinx IPs

In the completion of our project, we made use of several IPs from the Xilinx IP library. Below we provide the versions of each IP used along with a short description of its purpose in our system.

#### 4.4.1 Clocking Wizard

- Version: 6.0 [12]
- Used to generate 3 clocks:
    - 50 MHz - used to clock the Ethernet PHY MII to Reduced MII IP
    - 200 MHz - used to clock the Memory Interface Generator (MIG) IP
    - 100 MHz - used to clock all other peripherals as well as the Microblaze

#### 4.4.2 Processor System Reset

- Version: 5.0 [13]
- This IP is instantiated twice:
    - rst_clk_wiz_1_100M: provides reset for all peripherals
    - rst_mig_7series_0_81M: provides reset for the memory interface generator IP

#### 4.4.3 AXI Interconnect

- Version: 2.1 [14]
- This module is used to connect the Microblaze to all other peripherals except MIG
- It receives input from AXI Master of Microblaze and redirects this to the required AXI Slave peripheral

#### 4.4.4 Concat

- Version: 2.1 [15]
- Used to concatenate multiple signals into one signal. In our case concatenates interrupt signals

#### 4.4.5 AXI SmartConnect

- Version: 1.0 [16]

- Similar to the AXI Interconnect module but used specifically for the MIG

#### 4.4.6 AXI Timer

- Version: 2.0 [17]
- Used to generate a timed interrupt signal

#### 4.4.7 AXI Interrupt Controller

- Version: 4.1 [18]
- Receives interrupts from AXI Slave peripherals and sends these to the microblaze

#### 4.4.8 Microblaze Debug Module

- Version: 3.2 [19]
- Allows us to perform JTAg based debugging of the microblaze

#### 4.4.9 Memory Interface Generator

- Version: 4.2 [20]
- The MIG is used to interface with the off chip DDR memory.
- We use the DDR memory on the Nexys DDR board to store the WAV file as well as the dictionary of words.

#### 4.4.10 AXI Uartlite

- Version: 2.0 [21]
- Allows for asynchronous serial data transfer

#### 4.4.11 AXI Ethernetlite

- Version: 3.0 [22]
- Used to transfer handshaking messages as well as data packets between the load manage FPGA, compute FPGA and desktop clients

#### 4.4.12 Ethernet PHY MII to Reduced MII

- Version: 2.0 [23]

- Used to transfer handshaking messages as well as data packets between the load manage FPGA, compute FPGA and desktop clients

#### 4.4.13 AXI TFT Controller

- Version: 2.0 [24]
- Writes values from a programmable base address in DDR memory to the screen by driving VGA colour pins, HSYNC and VSYNC signals.

#### 4.4.14 Slice

- Version: 1.0 [25]
- Removes bits from a vector. Instantiated 3 times - one for red, green and blue 6 bit signals from the TFT controller to interface with 4 bit vga pins.

#### 4.4.15 Integrated Logic Analyzer

- Version: 6.2 [26]
- We used ILA to debug our SD Card implementation FSM.
- Also used ILA to debug AXI Ethernetlite signals to track packets

#### 4.4.16 Microblaze

- Version: 11.0 [27]
- The microblaze core was used as the orchestrator in the design with all the peripherals designed around it.
- Communication between the custom IPs as well as communication with DDR memory is done via the microblaze. main.c and echo.c are the two C source files where all software code run on the microblaze is present.
- The load manage FPGA and compute FPGA each have their own versions of main.c and echo.c

Block diagrams for load manage FPGA and compute FPGA can be found in Appendix J and Appendix K respectively.

## 5. Description of Design Tree

The design tree can be accessed at: https://github.com/DanNicolau/fpga-speech-recognition

**Description of Design Tree**

**-> compute_node**

This contains the relevant compute_node files we thought were worth including since it should be possible to recreate the block diagram and includes the constraint files. If you are skimming this project, this probably isn't what you want to look at.

Large parts of the autogenerated Vivado project such as the ip/ have been removed.

The compute_node/demo_group.sdk folder contains some relevant code in echo.c and main.c .

compute_node/ip_repo/ is the ip repository for our custom IPs.

**-> load_manager**

Same as above but for the load manager fpga.

**-> software_model**

This contains the code for the software model, minus any audio files you might use to generate the dictionary.

**-> src**

Contains most of the user-generated code.

**-> -> mem**

echo.c contains the load manager echo.c code. letters.h and .c contain some helped functions for writing text to the VGA assuming the video buffer is mapped to the right memory. Contains some scratchpad files for font bitmap as well.

**-> -> rtl**

This contains our rtl files:

- absolute_value.v
- euclidean_comparator.v
- frame.v
- load_manage.v
- moving_average.v
- pre_fft_wrapper.v
- vga_controller.v
- word_clipper.v
- word_clipper_end.v

**-> -> scripts**

Contains some python scripts:

- ibm_watson_test_to_speech: Some code to access ibm watson api
- tcp_client(DESLA): Detailed description of this file in section 4.1.1

**-> -> tb_local**

This contains testbenches for the rtl files. These are not as rigorous as we would have liked them to be but were still very useful in our development.

**-> text-to-speech**

This contains a python script to generate tts word samples.

## 6. Tips and Tricks

- Creating a software model is a valuable tool for better understanding what your hardware will need to do and more importantly for debugging. This is especially true for accelerating deterministic algorithms (hashing, ML, etc.); If you are not an expert on your algorithm, writing a software model will be beneficial to your team. There is also a tradeoff to consider between having an exact match of your software model to your hardware model and spending more time debugging a discrepancy. A close fit between HW and SW may be simple if you only use custom IP but it can be difficult to model Xilinx IPs that you cannot understand how they work under the hood (FFT for instance). In such a case, for some algorithms, breaking up your software model in pieces and comparing I/O at critical points will be beneficial.
- Write testbenches and simulations for your hardware. It is unlikely you will implement even simple modules without logical errors. Spending 15 minutes to write some menial SystemVerilog will save you many hours of hardware debugging time plus the time it would take to generate a bitstream.
- You could try out a different interface besides AXI. Especially inside an accelerator that can be pipelined, you can probably save debugging time by using only a valid signal or ready-to-receive/ready-to-send signals. Use AXI when you need to connect your accelerator to a bus, use vivado IP or want a standard interface.
- While trying to setup ethernet servers on the Nexys Video board make sure you leave the “temac_adapter_options” as autodetect for lwip202.
- Get features working individually in their own project before integration.
- Talk to your team and ask questions.

## 7. Video

We have made a video for our project which gives an overview of our implementation, describes the logic behind speech recognition, and then showcases a demo of our project. Please visit https://www.youtube.com/watch?v=peI0MENgLRk

## 8. References

[1] “What Are the Benefits of Speech Recognition Technology?,” *IEEE Signal Processing Society*, Nov. 02, 2018. https://signalprocessingsociety.org/publications-resources/blog/what-are-benefits-speech-recognition-technology (accessed Apr. 14, 2022).

[2] “An FFT-based speech recognition system,” *Journal of the Franklin Institute*, vol. 329, no. 3, pp. 555–562, doi: 10.1016/0016-0032(92)90054-K.

[3] L. 285, “Lecture 20: Fourier Transform and Speech Recognition - University of Southern California”.

[4] K. Chaudhary, “Understanding Audio data, Fourier Transform, FFT and Spectrogram features for a Speech Recognition System,” *Towards Data Science*, Jun. 04, 2021. Accessed: Apr. 14, 2022. [Online]. Available: https://towardsdatascience.com/understanding-audio-data-fourier-transform-fft-spectrogram-and-speech-recognition-a4072d228520

[5] M. rashad, “FPGA Speech Recognition (MATLAB + Altera DE0),” *YouTube*. Jun. 09, 2017. Accessed: Apr. 14, 2022. [Online]. Available: https://www.youtube.com/watch?v=hxbMyYtls48

[6] MohammedRashad, “GitHub - MohammedRashad/FPGA-Speech-Recognition: Expiremental Speech Recognition System using VHDL & MATLAB.,” *GitHub*. https://github.com/MohammedRashad/FPGA-Speech-Recognition (accessed Apr. 14, 2022).

[7] ncqui, “Speech Recognition Using Hidden Markov Model And FPGA.avi,” *YouTube*. Mar. 25, 2011. Accessed: Apr. 14, 2022. [Online]. Available: https://www.youtube.com/watch?v=KfzWQhKrTLM

[8] J. Hui, “Speech Recognition — GMM, HMM - Jonathan Hui,” *Medium*, Sep. 15, 2019. Accessed: Apr. 14, 2022. [Online]. Available: https://jonathan-hui.medium.com/speech-recognition-gmm-hmm-8bb5eff8b196

[9] “Discrete Fourier Transform (DFT),” *Xilinx*. https://www.xilinx.com/products/intellectual-property/dft.html (accessed Apr. 14, 2022).

[10] “Fast Fourier Transform (FFT),” *Xilinx*. https://www.xilinx.com/products/intellectual-property/fft.html (accessed Apr. 14, 2022).

[11]

https://www.allaboutcircuits.com/technical-articles/xilinx-fft-ip-core-fast-fourier-transform-software-walkthrough/ (accessed Apr. 14, 2022).

[12] “Documentation Portal.” https://docs.xilinx.com/v/u/en-US/pg065-clk-wiz (accessed Apr. 14, 2022).

[13] “Documentation Portal.” https://docs.xilinx.com/v/u/en-US/pg164-proc-sys-reset (accessed Apr. 14, 2022).

[14] “Documentation Portal.” https://docs.xilinx.com/r/en-US/ug1273-versal-acap-design/AXI-Interconnect (accessed Apr. 14, 2022).

[15] “Documentation Portal.” https://docs.xilinx.com/r/2020.2-English/ug1483-model-composer-sys-gen-user-guide/Concat (accessed Apr. 14, 2022).

[16] “Documentation Portal.” https://docs.xilinx.com/r/en-US/pg300-v-dp-rxss1/AXI-SmartConnect-IP-Core (accessed Apr. 14, 2022).

[17] “Documentation Portal.” https://docs.xilinx.com/v/u/en-US/pg079-axi-timer (accessed Apr. 14, 2022).

[18] “Documentation Portal.” https://docs.xilinx.com/v/u/en-US/pg099-axi-intc (accessed Apr. 14, 2022).

[19] “Documentation Portal.” https://docs.xilinx.com/r/en-US/pg115-mdm (accessed Apr. 14, 2022).

[20] “Documentation Portal.” https://docs.xilinx.com/v/u/en-US/wp260 (accessed Apr. 14, 2022).

[21] “Documentation Portal.” https://docs.xilinx.com/v/u/en-US/pg142-axi-uartlite (accessed Apr. 14, 2022).

[22] “Documentation Portal.” https://docs.xilinx.com/v/u/3.00a-English/ds759_axi_ethernet (accessed Apr. 14, 2022).

[23] “Documentation Portal.” https://docs.xilinx.com/v/u/en-US/pg146-mii-to-rmii (accessed Apr. 14, 2022).

[24] “Documentation Portal.” https://docs.xilinx.com/v/u/en-US/pg095-axi-tft (accessed Apr. 14, 2022).

[25] “Documentation Portal.” https://docs.xilinx.com/r/2020.2-English/ug1483-model-composer-sys-gen-user-guide/Slice (accessed Apr. 14, 2022).

[26] “Documentation Portal.” https://docs.xilinx.com/r/en-US/ug908-vivado-programming-debugging/ILA (accessed Apr. 14, 2022).

[27] “Documentation Portal.” https://docs.xilinx.com/v/u/en-US/wp501-microblaze (accessed Apr. 14, 2022).

# 9. Appendices

```
########### GETTING THE IP ADDRESS OF THE COMPUTE FPGA FROM THE LOAD MANAGE FPGA ########


in_data = b""       # this variable stores the reply after sending compute request
in_data_host = b""
in_data_request_id = b""

print("\nSending a compute request to IP: " + HOST)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:

    s.connect((HOST, PORT))

    byte_data = b"\x31";                        # using this code to indicate a compute request
    byte_data = byte_data + STATION_NO

    print("Sending: ", repr(byte_data))
    s.sendall(byte_data)
    in_data = s.recv(1024)

    in_data_host = in_data[0:1]
    in_data_request_id = in_data[1:2]

    print("Received", repr(in_data))

    s.shutdown(1)
    s.close()

in_data_host_str = int.from_bytes(in_data_host, 'little')

HOST = "1.1." + str(in_data_host_str) + ".2"
print("Compute request reply received, will send data to: " + HOST)

################################################################################################
```

Appendix A: Desktop Client getting IP Address of Compute FPGA from Load Manage FPGA

```
################ Opening WAV file for PREPROCESSING ###################################

print("\nPre-processing WAV file")

wav_f = open(r"W:\ECE532\Project\Audio Files\WAV\data_set_2.wav", "rb")
byte_data = wav_f.read(2)    # start reading the header
head_count = 1

while (byte_data != b"da"):           # keep reading the header until we get to the data section
    head_count = head_count + 1
    byte_data = wav_f.read(2)

byte_data = wav_f.read(2)
head_count = head_count + 1

if (byte_data != b"ta"):         # assert that read data is "ta"
    exit()

byte_data = wav_f.read(4)              # next 4 bytes following the data section is the size of the file
byte_data = int.from_bytes(byte_data, 'little')

print("Total number of bytes skipped (header): ", (head_count+1)*2)
print("Size of the file (in bytes) is: ", byte_data)

iterations = byte_data // PACKET_SIZE
remainder = byte_data % PACKET_SIZE

################################################################################################
```

Appendix B: Desktop Client Preprocessing the WAV file before sending to Compute FPGA

```
######### Opening a new CONNECTION ###################

# Open .wav file for reading in binary mode
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(30)
s.connect((HOST, PORT))

#####################################################
```

Appendix C: Desktop Client Establishing Connection with the Compute FPGA

```
######### Sending a NEW_DATA REQUEST #################

new_data_request = b"\x32"
new_data_request = new_data_request + in_data_request_id

print("\nNow sending a new_data request to " + HOST)
s.sendall(new_data_request)

server_reply = s.recv(1024)
server_reply = int.from_bytes(server_reply, 'little')

if (server_reply):
    print("Server is ready to receive data")
else:
    print("Server is not ready to receive data")

#####################################################
```

Appendix D: Handshaking Between Desktop Client and Compute FPGA before transferring data

```
#################### Send DATA and close WAV file ########

print("\nStarting data transfer!!")

packet_count = 1
for x in range(iterations):
    byte_data = wav_f.read(PACKET_SIZE)
    print("Sending packet: ", packet_count)
    #print("Sending: ", repr(byte_data))
    s.sendall(byte_data)
    in_data = s.recv(PACKET_SIZE)
    #print("Received", repr(in_data))
    packet_count = packet_count + 1
    x = x+1


byte_data = wav_f.read(remainder)
print("Sending packet: ", packet_count)
s.sendall(byte_data)
in_data = s.recv(1024)

wav_f.close();
print("Ending data transfer!!")

#####################################################
```

Appendix E: Transferring the WAV file to compute FPGA

```
######### Sending an END_DATA REQUEST #################

print("\nNow sending a end_data request to " + HOST)
s.sendall(b"\x33")

server_reply = s.recv(1024)
server_reply = int.from_bytes(server_reply, 'little')

#print("Server reply ", server_reply, "\n")

if (server_reply):
    print("Server acknowledges end_data request")
else:
    print("Server DOES NOT acknowledge end_data request")


#########################################################
```

Appendix F: Handshaking between Desktop Client and Compute FPGA after data transfer

```
######### Sending an START_COMPUTE REQUEST #################

s.settimeout(120)


print("\nNow sending a start_compute request to " + HOST)
s.sendall(b"\x34")

server_reply = s.recv(1024)
server_reply = int.from_bytes(server_reply, 'little')

if (server_reply):
    print("Server acknowledges start_compute request")
else:
    print("Server DOES NOT acknowledge start_compute request")


#########################################################
```

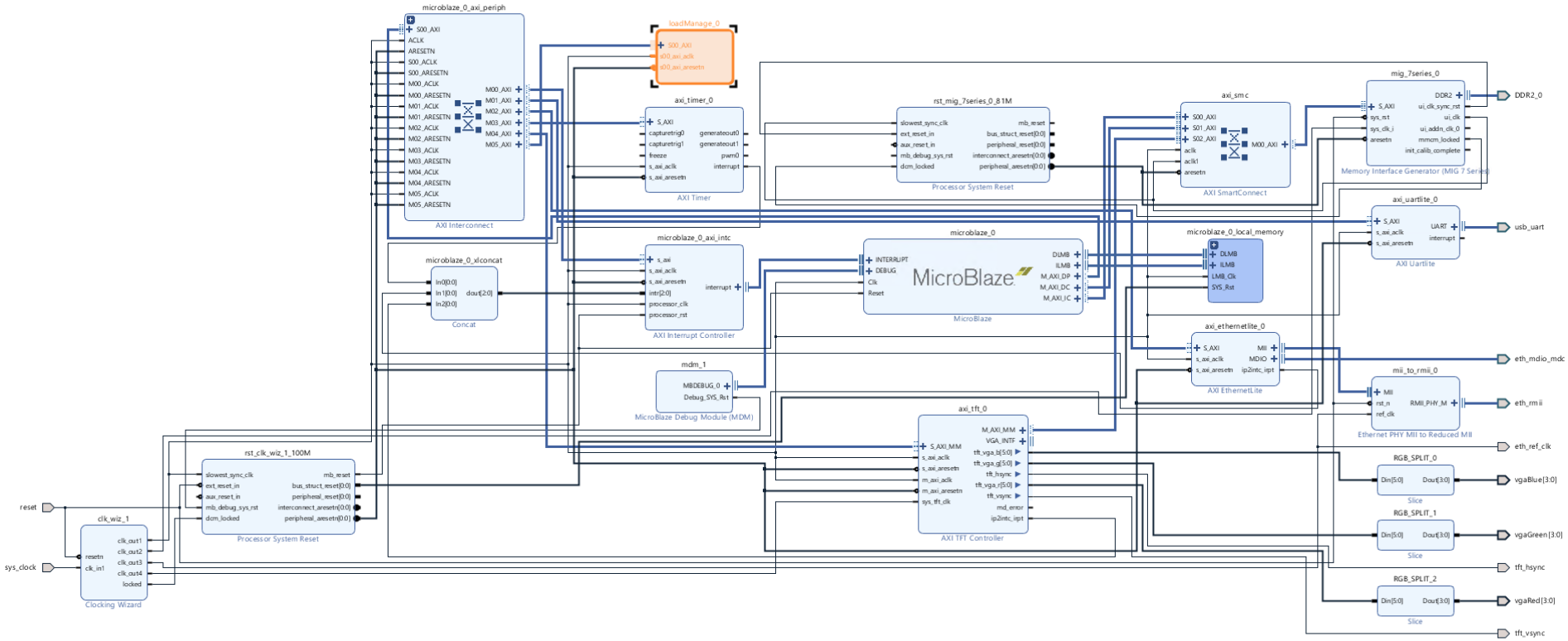
Appendix G: Desktop Client requesting Compute FPGA to begin speech recognition

```
################## CLOSING CONNECTION ###############

s.shutdown(1)
s.close()

#####################################################
```

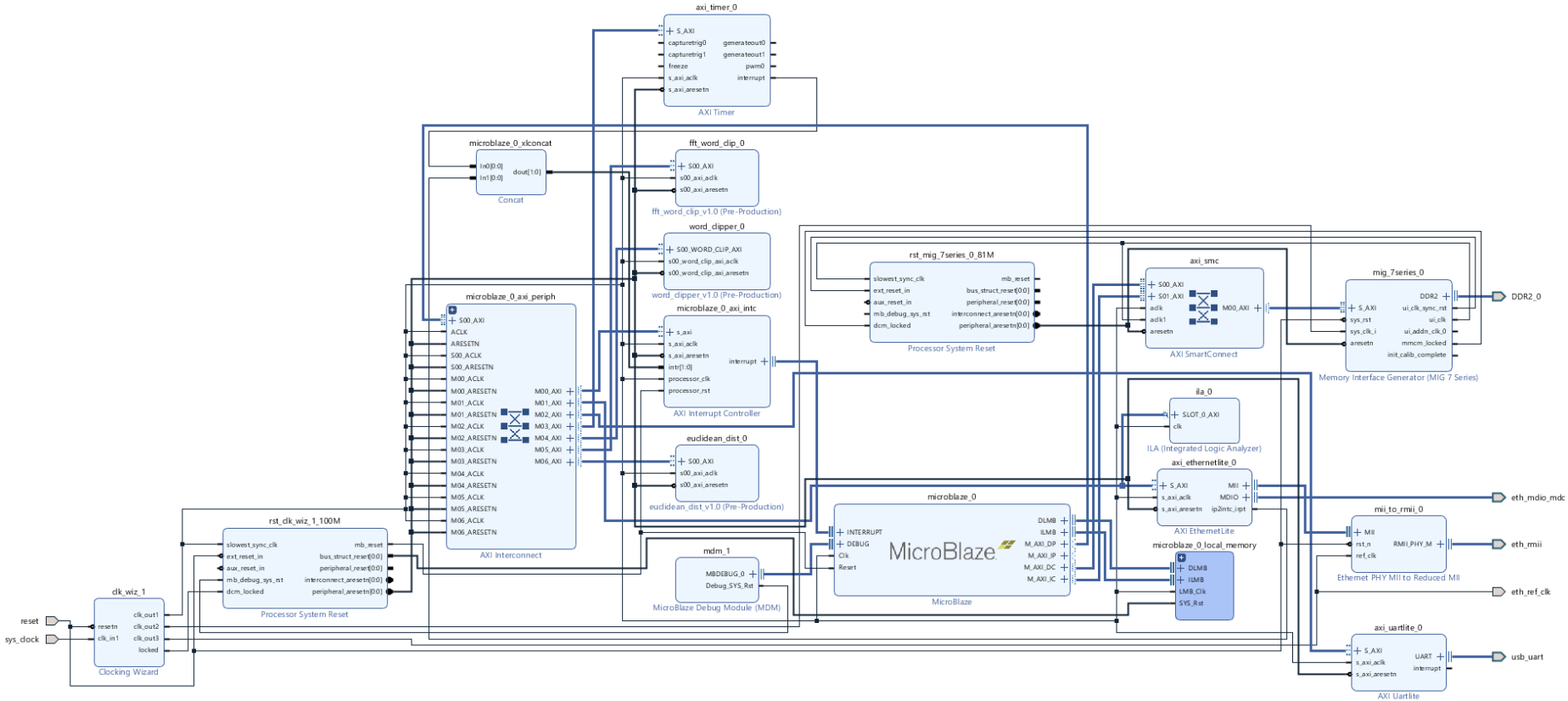
Appendix H: Close connection between Desktop Client and Compute FPGA

```
void init_load_manager() {

    LOADMANAGE_mWriteReg(XPAR_LOADMANAGE_0_S00_AXI_BASEADDR, LOADMANAGE_S00_AXI_SLV_REG0_OFFSET, 5);
    LOADMANAGE_mWriteReg(XPAR_LOADMANAGE_0_S00_AXI_BASEADDR, LOADMANAGE_S00_AXI_SLV_REG1_OFFSET, 8);
    LOADMANAGE_mWriteReg(XPAR_LOADMANAGE_0_S00_AXI_BASEADDR, LOADMANAGE_S00_AXI_SLV_REG2_OFFSET, 9);
}
```

Appendix I: Function to be called in main.c to initialize the load manager with the compute FPGA station numbers




Appendix J: Block Diagram for Load manage FPGA




Appendix K: Block Diagram for Compute FPGA