ECE532: Digital Systems Design

April 6, 2025

Keyword Speech Recognition Accelerator

Group 33:

Xinyue Chen

Xinyu Chen

Zhenze Zhao

Sherman Lin

Contents

1 Overview

This project implements a hardware-accelerated keyword speech recognition system using the Nexys4 DDR FPGA platform. A machine learning model is deployed directly onto the FPGA to perform inference with improved performance. The platform incorporates several components, including a Bluetooth BT2 module, DDR DRAM, Ethernet communication with an external PC, and a soft-core processor (MicroBlaze). During inference, the system captures input speech data from the user, preprocesses the data, and transmits it from the PC to the FPGA via Ethernet, and the recognized keyword is sent back via Bluetooth. Our machine learning model is trained to recognize 8 keywords: *down*, *go*, *left*, *no*, *right*, *stop*, *yes* and *up*. Additionally, the model's confidence score is shown on the seven-segment display. The system highlights how custom hardware can be used to accelerate speech recognition tasks in a resource-constrained environment.

1.1 Background

Speech recognition technology is becoming increasingly prevalent in today's applications, from virtual assistants to voice-controlled devices. Traditional implementations often rely on high-performance CPUs or GPUs, which consume significant power and are not ideal for embedded systems. FPGAs offer a viable alternative due to their reconfigurability and capacity for parallel computation, making them ideal for accelerating compute-heavy tasks like machine learning inference. Our project builds on this potential by targeting the keyword recognition task and implementing the model directly in hardware.

1.2 Motivation

Our group chose to explore keyword speech recognition acceleration on FPGAs to bridge our interests in machine learning and digital hardware design. We noticed that existing approaches often offload core computations to software, which introduces latency and inefficiencies. We wanted to take a different route by deploying the speech recognition model directly in hardware, minimizing software dependencies. This project allows us to apply machine learning concepts to a real-time embedded system while providing an opportunity to gain hands-on experience with peripheral integration, data communication, and hardware/software co-design.

1.3 Goals

The primary goal of this project is to develop a keyword speech recognition system that runs efficiently on an FPGA. This involves implementing a custom accelerator for a machine learning model capable of identifying specific keywords, while optimizing resource usage and performance. Furthermore, the system is also designed to offer a streamlined, user-friendly experience, requiring only a button press to initiate speech input. Behind the scenes, the system integrates data flow between a host PC and the FPGA using Ethernet, model weights are transferred during setup, and input speech data is streamed during inference. The FPGA processes this data in real time, and outputs the prediction result via Bluetooth and on-board seven segment displays. Supporting components like MicroBlaze handle tasks such as control flow and memory access. Overall, the system aims to demonstrate a cohesive and practical approach to hardware-accelerated keyword speech recognition.

1.4 Block Diagram

The following block diagram (Figure 1) illustrates the overall system architecture of our FPGA-based speech recognition accelerator. It highlights the key components and data flow between the PC, FPGA, Bluetooth, and GUI interfaces.

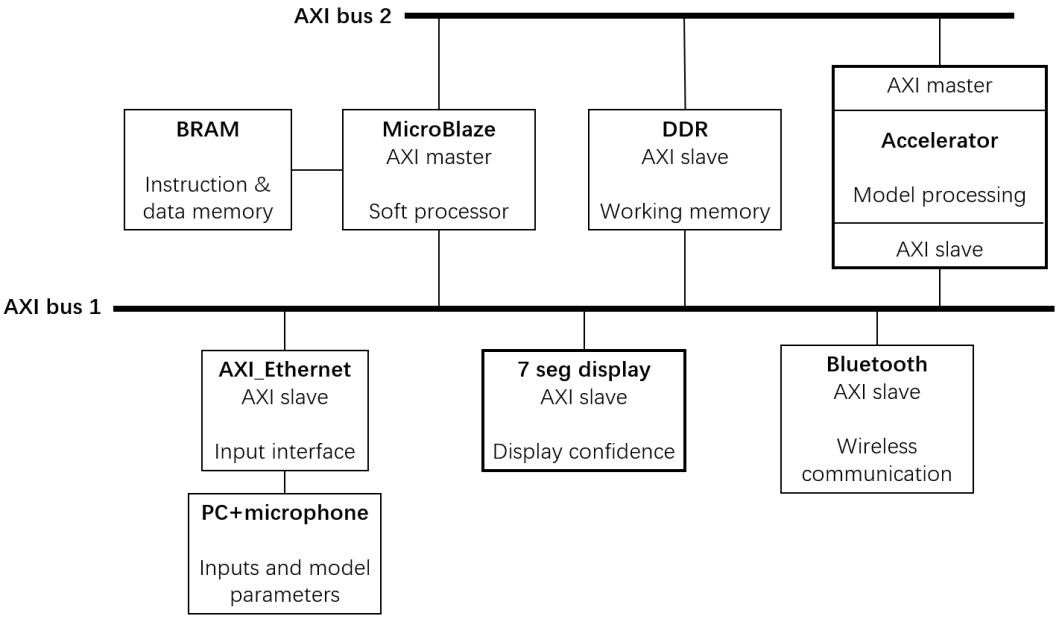

Figure 1: System block diagram showing the interaction between PC, FPGA, and Bluetooth GUI

2 Outcome

In this project, we successfully designed and implemented a complete audio keyword recognition system accelerated by FPGA.

2.1 Result

Our project successfully demonstrates a real-time FPGA-based keyword recognition system with reliable Ethernet data transmission and hardware-accelerated inference. We achieved stable and efficient communication between the PC and FPGA for both model parameters and audio input data. The weights and biases are transmitted through IP/MAC with a handshake protocol, while audio spectrograms are sent upon push-button detection on the FPGA. The fragmentation, acknowledgment, and retransmission mechanism ensures data integrity throughout the process.

Inference is performed primarily using a custom-designed PE (Processing Element) array, which handles both convolutional and fully connected layers. Maxpooling is executed by the MicroBlaze soft processor. This hardware-accelerated design significantly reduces inference time. Compared to a pure C-based software implementation that takes over five hours to complete inference, our FPGA-based system completes the same task in approximately one hour — a substantial performance improvement demonstrating the advantage of parallel processing in hardware.

The output of the inference is transmitted via Bluetooth to a PC, where a GUI displays a stickman animation corresponding to the recognized keyword, providing an intuitive visual feedback loop. While our original quantized model achieved an accuracy of 66%, the final FPGA implementation using fixed-point arithmetic saw a drop in accuracy to approximately 30%. This reduction is attributed to the precision limitations of fixed-point representation and the lack of floating-point support on the FPGA.

Nonetheless, the project achieved a functional, accelerated inference pipeline with interactive visualization, validating the feasibility of deploying AI workloads on resource-constrained devices.

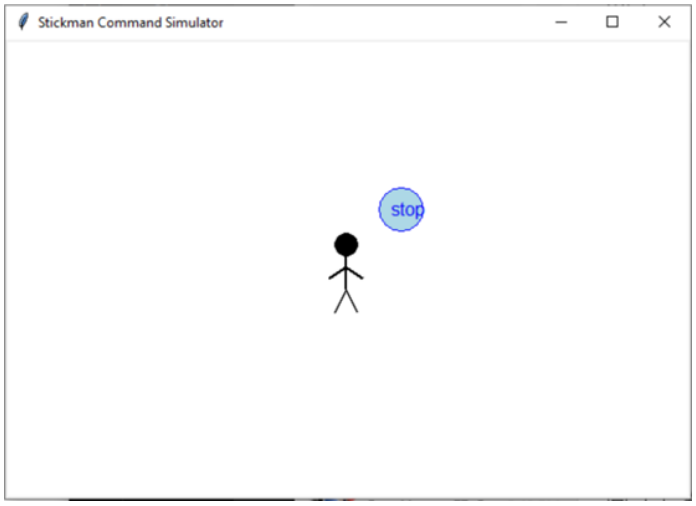

Figure 2: System overview showing FPGA-based keyword recognition and Bluetooth output.

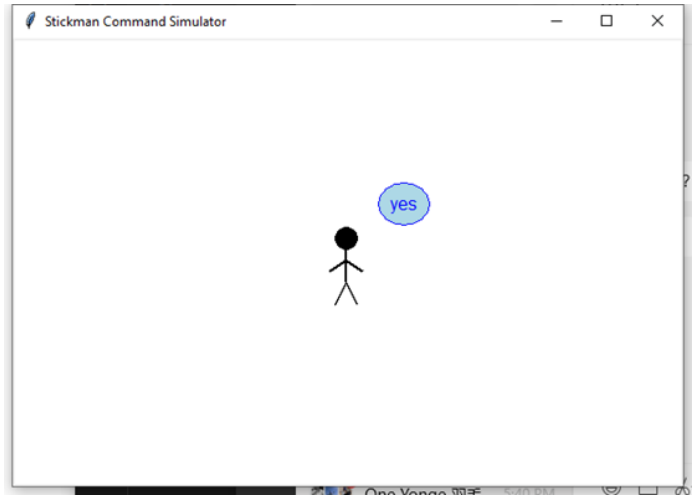

Figure 3: System overview showing FPGA-based keyword recognition and Bluetooth output.

2.2 Limitations

One of the main limitations we encountered in this project was the constrained hardware resources on the FPGA, particularly limited on-chip memory and restricted bandwidth for data movement. These constraints made it challenging to deploy large or high-accuracy neural network models directly, especially after quantization, which already introduces a trade-off between model size and performance.

Additionally, the communication between the PC and FPGA for loading weights and transmitting audio input was initially slow and error-prone due to fragmentation and lack of handshaking. This limitation often led to corrupted or incomplete data during model loading, impacting inference reliability.

To address these challenges, we implemented a custom Ethernet packet fragmentation protocol with ACK/NACK handshaking to ensure reliable transmission of model weights and audio inputs from PC to FPGA. This significantly improved the robustness of data transfer. We also restructured the model and quantized it to fit into the resource budget of the FPGA, sacrificing some accuracy in exchange for deployability.

Furthermore, we modularized the workload by assigning convolution and fully connected layers to the PE array for acceleration, while keeping max pooling under MicroBlaze control to balance complexity and utilization. These improvements enabled us to achieve a functional end-to-end system, with reasonable inference speed and reliable dataflow between host and hardware.

2.3 Future Work and Optimizations

Software

From the software side, one key area for improvement is optimizing the model architecture to balance accuracy and size more effectively. Our current model was aggressively quantized to fit within the FPGA's resource constraints, which contributed to a noticeable accuracy drop. In future iterations, techniques such as structured pruning, mixed-precision quantization, or knowledge distillation could be explored to reduce model complexity while preserving recognition performance. Additionally, experimenting with different layer configurations and activation functions tailored for fixed-point computation may yield better compatibility with hardware accelerators and improve overall efficiency.

Another software optimization involves the transmission of large model weights and biases. Currently, these tensors are sent over Ethernet using a reliable but time-consuming fragmentation and handshake protocol. Introducing a lightweight compression algorithm, such as run-length encoding or simple Huffman coding, could significantly reduce transmission time without adding substantial decompression overhead on the FPGA side. This would improve system responsiveness during initialization and make the model loading process more practical for dynamic or reconfigurable models in the future.

Hardware

On the hardware side, incorporating a non-volatile memory module such as QSPI flash or SD card storage would enable persistent storage of model weights and biases on the FPGA board. This addition would eliminate the need to reload the model data from the PC each

time the system powers up, significantly reducing initialization time and improving overall usability. Moreover, by offloading the weight storage from limited on-chip memory, we can support larger and more accurate models that were previously constrained by BRAM or DRAM capacity. This enhancement would allow the system to scale up in complexity and performance while maintaining fast startup and flexible deployment.

To further improve inference speed, it is critical to fully utilize the processing capability of the PE array. Our current PE array consists of 8 PEs, each capable of performing a matrix multiplication in just 9 clock cycles. However, due to bottlenecks in data movement—specifically the weight and activation fetching handled by the MicroBlaze processor—only two PEs are used concurrently when their 48-byte input buffer is filled. This introduces underutilization and idle cycles that slow down overall inference. To address this, implementing multithreading or pipelined data loading strategies on the MicroBlaze could allow data fetching and computation to occur in parallel. This would reduce latency between data availability and PE execution, improving PE utilization.

Another critical limitation is the data throughput between memory and the PE array. Currently, the AXI4 protocol cannot efficiently deliver the required 48×4 bytes of data at high speed. Replacing or complementing AXI with a higher-throughput interconnect, such as AXI-Stream with burst DMA or even custom high-bandwidth bus protocols, could allow larger chunks of data to be fed into the PE array per cycle. Moreover, with the addition of non-volatile memory or onboard SSD, weights and biases could be assigned fixed memory addresses, allowing data fetching to be entirely offloaded to dedicated hardware using DMA engines. This eliminates processor involvement during runtime, significantly reducing latency and improving determinism of data transfer.

Looking forward, a deeper integration between memory hierarchy and compute blocks can be considered. Implementing double-buffering or local cache registers near PEs could hide memory latency and further increase throughput. Additionally, introducing scheduling logic in programmable logic (PL) to orchestrate the loading of weights and activations into PEs—based on inference flow—would reduce reliance on software and allow finer control of parallelism. By combining multithreaded software pipelines, efficient bus architecture, and hardware-managed memory access, the system can approach the theoretical peak performance of the PE array, enabling near-real-time inference on more complex models in future iterations.

3 Milestone Summary

3.1 Weekly Milestones

Table 1: Comparison of Planned and Final Milestones

Milestone	**Original Milestone**	**Final Milestone**
Week 1	Integrate microphone functionality and access SD card.	Designed and tested PC audio interface with microphone and SD card. Implemented UART module to prepare for Bluetooth communication.
Week 2	Implement and verify a single PE.	Developed the audio streaming and loopback test pipeline, and began testing FFT-based analysis.
Week 3	Implement and verify PE array.	Finalized PE array RTL design and tested. Implemented IP packet audio transmission and model quantization.
Week 4	Implement the full accelerator on FPGA.	Connected accelerator to BRAM and DRAM, completed DRAM-BRAM interface testing. Bluetooth PMOD integrated with MicroBlaze UART.
Week 5	Achieve full system functionality.	Stickman GUI and continuous control over Bluetooth. Converted audio and model file transfer from Ethernet.
Week 6	Add debug info and verify Bluetooth.	Completed UART-based stickman control via Bluetooth. Switched to UART for both audio input and weight transfer; used Bluetooth UART solely for control output.

3.2 Discussion and Changes

Over the course of the semester, our project underwent several key architectural changes driven by resource limitations and software compatibility issues. These changes, while devi-

ating from the original milestone plan, enabled us to deliver a robust and functional FPGA-accelerated keyword recognition system.

Initially, we proposed using the onboard microphone for direct audio input to the FPGA. However, we discovered that pre-processing the audio was necessary before feeding data into the inference pipeline. Performing this pre-processing step on the FPGA proved infeasible due to hardware limitations and complexity. As a result, we shifted the audio input and pre-processing to the PC, using Ethernet to transmit the processed data to the FPGA. This change significantly simplified the system while maintaining accuracy.

For model weight storage, we originally intended to utilize an SD card. Unfortunately, we encountered persistent issues integrating the XilFFS library within Vivado 2018.3, preventing successful file system access on the MicroBlaze. To overcome this limitation, we opted to load model weights directly into DDR memory from the PC, bypassing the need for SD-based storage entirely.

Initially, we planned to use UART (via USB) to transfer both the model weights and audio input to the FPGA. However, with the removal of the SD card, we needed an additional component to maintain project complexity. We decided to incorporate a Bluetooth PMOD (BT2) module for output visualization. Since the FPGA development board offers only one UARTLite peripheral, a conflict arose between UART usage for data transfer and Bluetooth communication. To resolve this, we dedicated the single UARTLite interface to the Bluetooth PMOD and transitioned all model and audio data transfers to Ethernet.

The original plan assumed quick integration of a single processing element (PE), but implementation delays and complexity led us to focus on a robust MAC unit architecture and its surrounding control logic first. By taking the time to validate correct memory mapping and hardware simulation outputs, we laid a solid foundation for scaling to a full PE array.

In later weeks, we encountered bandwidth and memory bottlenecks when deploying the full model. As a solution, we quantized the model using QAT (Quantization-Aware Training) and developed a robust fragmentation protocol with acknowledgment to transfer weights and audio features reliably over Ethernet. This improved the robustness and reliability of the data pipeline.

Overall, these iterative changes allowed us to meet our functional goals, even if the technical path diverged from the original proposal. The system's current architecture balances hardware constraints with practical usability, resulting in a modular, efficient, and responsive keyword recognition accelerator.

4 Hardware Architecture

4.1 Overview of FPGA Design

This project employs the Xilinx Nexys 4 DDR board (Fig. 4) to implement a speech recognition machine learning accelerator. The system architecture begins with audio sampling and preprocessing on a host PC, after which the processed data is stored in the DDR memory on the FPGA board. A custom-designed intellectual property (IP) core is responsible for executing the computationally intensive convolution operations integral to the machine learning model. Oversight of the overall data flow is managed by a MicroBlaze soft processor, which coordinates the interaction between subsystems. The final output is transmitted via Bluetooth, and a seven-segment display provides a real-time indication of the confidence level associated with the recognition results.

The architecture exhibits a robust interplay between dedicated hardware and programmable logic, enabling enhanced parallel processing and efficient data management. The integration of a specialized convolution accelerator with the MicroBlaze soft processor supports optimized resource allocation and dynamic control over computational tasks. This co-design strategy not only minimizes latency and bolsters real-time performance but also establishes a scalable platform adaptable to increasingly complex machine learning models. In essence, the approach highlights the transformative potential of reconfigurable computing in advancing embedded system capabilities while addressing evolving application demands.

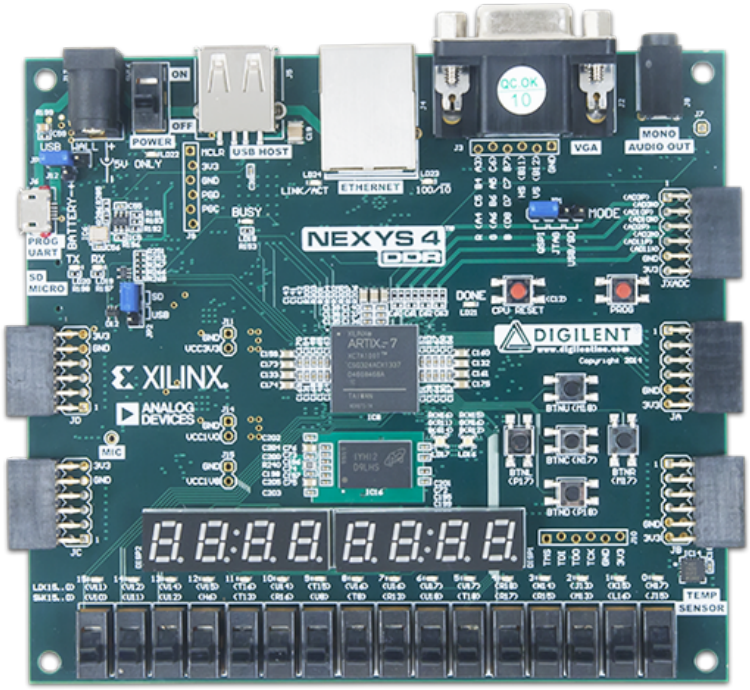

Figure 4: Nexys 4 DDR FPGA board.

4.2 Neural Network Inference Core

The accelerator IP (Fig. 5) is a fundamental component of the system, engineered to accelerate the convolution operations central to the speech recognition model. At its core, the architecture employs eight Processing Elements (PEs) that execute multiply-and-accumulate operations in parallel, thereby reducing computation time and boosting overall system throughput.

Each of the eight PEs is meticulously designed to perform signed 8-bit multiplication and 32-bit accumulation, leveraging the advanced capabilities of DSP48 blocks to achieve efficient and high-speed arithmetic processing. In each PE, the multiplication operation is executed via a dedicated multiplier IP core from Vivado, implemented with a three-stage pipeline that maximizes throughput while maintaining computational accuracy by exploiting the inherent parallelism of DSP48 blocks. Similarly, the accumulation function is realized through an adder IP core, featuring a two-stage pipeline that ensures rapid and precise addition of intermediate results. To further optimize performance, each PE incorporates a local register file designed to store computed results for subsequent use as dictated by the instructions, effectively reducing the overhead associated with data transfers between the processing element and external memory. This comprehensive design not only accelerates

the core arithmetic operations essential for convolution computations but also enhances overall system efficiency by minimizing data movement and latency.

A key architectural feature of the accelerator is the integration of a Ping-Pong buffer mechanism. This dual-buffer strategy minimizes data loading latency by allowing one buffer to continuously feed data to the PEs while the other is simultaneously replenished. Such an arrangement is critical for maintaining a constant stream of data, essential for the real-time processing demands of the system.

Furthermore, the accelerator is equipped with both AXI slave and AXI master interfaces. The AXI slave interface facilitates the reception of data and instructions from the MicroBlaze soft processor, ensuring smooth integration with system control logic. Meanwhile, the AXI master interface empowers the accelerator to independently access DDR memory, enabling autonomous data fetch and storage operations. This dual-interface configuration significantly enhances system flexibility and scalability by streamlining both data and instruction flows.

In summary, the integrated design of the accelerator IP combines optimized parallel processing, efficient data buffering, and versatile interfacing, establishing a robust, scalable subsystem ideally suited for real-time speech recognition applications.

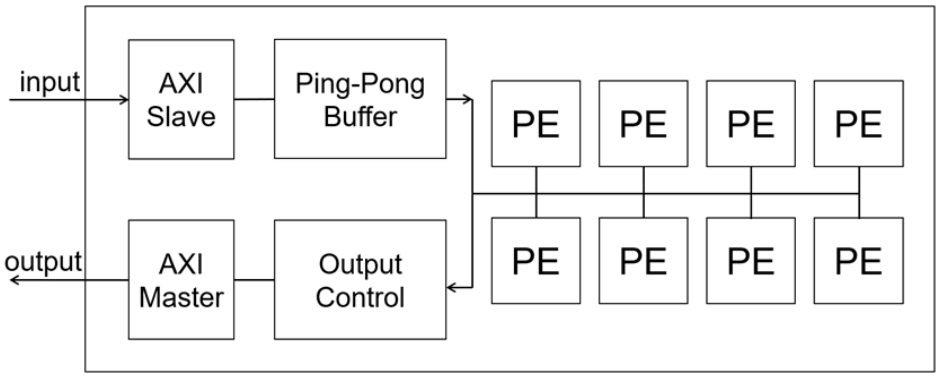

Figure 5: Accelerator Architecture.

4.3 AXI Bus and Memory Arrangement

The system architecture incorporates a dual 32-bit AXI bus configuration to effectively manage communication among the MicroBlaze processor, the accelerator IP, and various peripheral devices. One of the AXI buses is dedicated to facilitating high-bandwidth, low-latency access to the DDR memory by both the MicroBlaze processor and the accelerator. This design choice ensures that data-intensive operations, such as fetching audio inputs and processing intermediate results, are handled efficiently. In contrast, the second AXI bus is reserved for peripheral control, enabling the MicroBlaze to manage devices such as the

7-segment display, Bluetooth module, Ethernet interface, and other I/O components. This segregation of data and control pathways minimizes interference between memory transactions and peripheral management, ultimately enhancing overall system responsiveness.

Within the DDR memory, a deliberate partitioning scheme has been implemented to optimize resource allocation and support the system's varied operational requirements. Specifically, one memory segment is allocated for storing audio inputs and model parameters, ensuring that both raw data and pre-configured parameters are readily available for processing. A second partition is dedicated to holding the intermediate results generated by the accelerator during convolution operations, thereby reducing the need for redundant data transfers and enabling iterative computation with reduced latency. Additionally, a separate segment is reserved for the storage of instructions and operands for the MicroBlaze processor, facilitating streamlined execution of control and computational tasks.

Overall, this dual AXI bus configuration combined with strategic DDR memory partitioning enhances data flow management and system scalability. The arrangement not only supports efficient high-speed data exchanges and peripheral control but also ensures that computational resources are optimally allocated to meet the real-time performance requirements of the speech recognition accelerator.

4.4 Output Communication over UART/Bluetooth

The output communication in our hardware system is implemented via UART over Bluetooth using Digilent's BT2 PMOD module equipped with the RN42 Bluetooth transceiver. The RN42 is a class 2 Bluetooth v2.1 + EDR module that supports the Serial Port Profile (SPP), making it ideal for low-latency, wireless UART communication. This module is connected to the Nexys 4 DDR board through a PMOD interface, allowing the MicroBlaze soft-core processor to transmit inference results wirelessly to a PC in real time.

The Nexys 4 DDR FPGA board connects to the RN42 module using two FPGA I/O pins mapped to the TX and RX lines of the UART core. The UART peripheral is instantiated using Xilinx IP in the Vivado design and mapped to the MicroBlaze's memory space. It is configured with 8 data bits, no parity, 1 stop bit (8-N-1 format), and a baud rate of 115200 bps — a speed well supported by both the MicroBlaze UART and the RN42 module. Output messages consist of simple ASCII sequences representing the classification result (e.g., an index from 0 to 7 corresponding to a keyword). These messages are written to the UART transmit FIFO, which automatically handles the byte framing and start/stop bit generation required by the UART protocol.

The RN42 transmits this data over Bluetooth using SPP. On the PC side, once the module is paired via Bluetooth, it appears as a virtual COM port (e.g., COM5) to the operating system. A Python script uses the `pyserial` library to continuously monitor this COM port. Upon receiving a complete message, it decodes the output and triggers a response in a graphical interface. Specifically, the script controls a stickman character on the screen that reacts to the recognized keyword — visually representing commands such as "left," "right," or "jump."

This UART-Bluetooth communication pipeline provides a lightweight and reliable mechanism to wirelessly transmit inference results without the need for an Ethernet or USB connection. It separates the inference engine from the display interface, reducing system complexity and increasing portability. The use of RN42's robust SPP implementation ensures minimal transmission delay and high compatibility across different host platforms.

4.5 FPGA Resource Utilization

The tables below summarize the FPGA resource utilization for key components of the design.

Table 2: Slice Logic

Site Type	**Used**	**Fixed**	**Available**	**Util%**
Slice LUTs	23578	0	63400	37.19
LUT as Logic	19483	0	63400	30.73
LUT as Memory	4095	0	19000	21.55
LUT as Distributed RAM	1808	0		
LUT as Shift Register	2287	0		
Slice Registers	27698	13	126800	21.84
Register as Flip Flop	27682	13	126800	21.83
Register as Latch	0	0	126800	0.00
Register as AND/OR	16	0	126800	0.01
F7 Muxes	480	0	31700	1.51
F8 Muxes	36	0	15850	0.23

Table 3: Memory

Site Type	**Used**	**Fixed**	**Available**	**Util%**
Block RAM Tile	64.5	0	135	47.78
RAMB36/FIFO*	60	0	135	44.44
RAMB36E1 only	60			
RAMB18	9	0	270	3.33
RAMB18E1 only	9			

Table 4: DSP

Site Type	**Used**	**Fixed**	**Available**	**Util%**
DSPs	16	0	240	6.67
DSP48E1 only	16			

Table 5: IO and GT Specific

Site Type	**Used**	**Fixed**	**Available**	**Util%**
Bonded IOB	77	77	210	36.67
IOB Master Pads	38			
IOB Slave Pads	36			
Bonded IPADs	0	0	2	0.00
PHY_CONTROL	1	1	6	16.67
PHASER_REF	1	1	6	16.67
OUT_FIFO	4	4	24	16.67
IN_FIFO	2	2	24	8.33
IDELAYCTRL	1	0	6	16.67
IBUFDS	0	0	202	0.00
PHASER_OUT/PHASER_OUT_PHY	4	4	24	16.67
_PHASER_OUT_PHY only	4			
PHASER_IN/PHASER_IN_PHY	2	2	24	8.33
_PHASER_IN_PHY only	2			
IDELAY2/IDELAY2E_FINEDelay	16	16	300	5.33
_IDELAY2 only	16			
ILOGIC	16	16	210	7.62
ISERDES	16			
OLOGIC	45	45	210	21.43
OUTFF_ODDR_Register	3	3		
TFF_ODDR_Register	2	2		
OSERDES	42	42		

5 Software Design

The main software components in this project can be broken up into five parts:

- Model Pre-Training & Quantization, Encoding of Model Parameters
- Audio Input & Preprocessing

- Data Transfer using Ethernet Communication / Parsing of Received Data
- Main MicroBlaze Program
- Python GUI & Bluetooth Integration

In this section, we will explain each component in more detail.

5.1 Model Pretraining & Quantization

The machine learning model used in this project is adapted from the TensorFlow tutorial *"Simple Audio Recognition: Recognizing Keywords"* [1]. Following this tutorial as a baseline, we trained a convolutional neural network (CNN) to classify eight specific keywords: *down*, *go*, *left*, *no*, *right*, *stop*, *yes* and *up*. We used the same mini Speech Commands dataset provided in the tutorial but made slight modifications to the model architecture. Specifically, we removed the resizing and normalization layers, as these layers are not compatible with TensorFlow's Quantization-Aware Training (QAT) framework. As a result of this change, our model achieves approximately 80% accuracy prior to quantization, in contrast to the 90% accuracy reported in the original tutorial.

Quantization-aware training was applied to enable conversion of the model to fixed-point data types, which is necessary for FPGA deployment. We followed the TensorFlow *"Quantization-Aware Training in Keras example"* [2] to perform QAT and convert the model to TensorFlow Lite (TFLite) format. In the quantized model, all weights are stored as 8-bit integers (int8), and biases are stored as 32-bit integers (int32). This conversion significantly reduces the model size from 120MB to approximately 30MB, with a corresponding drop in accuracy to around 60%.

After extensive hyperparameter tuning and training iterations, we obtained a final quantized model suitable for deployment on the FPGA. However, TFLite models are not directly usable on the FPGA, as they are stored in a structured flatbuffer format that is not optimized for parsing on hardware. Therefore, we developed a custom toolchain to convert the `.tflite` model into a raw binary format suitable for Ethernet transfer and hardware loading.

This conversion process involved parsing the `.tflite` file to extract necessary tensor information, including: tensor ID, shape, dimensionality, data type, quantization scales and zero points, and the raw tensor data (e.g., model weights and biases). These were stored in a Python dictionary structure before being encoded into a binary file using a custom Python script. The binary encoding follows a specific structure:

- 4 bytes for the tensor ID
- 4 bytes for the total number of shape dimensions
- 4 bytes for each dimension size
- 4 bytes for the data type
- 4 bytes for the number of quantization scales
- 4 bytes for each scale
- 4 bytes for the number of zero points
- 4 bytes for each zero point
- 4 bytes for the length of the raw tensor data
- Followed by the variable-length raw data

This format ensures that the FPGA can correctly parse and load the model data during inference.

5.2 Audio Input & Preprocessing

The audio input and preprocessing pipeline is implemented entirely on the PC using Python, leveraging standard libraries such as `pyaudio`, `numpy`, `scipy`, and `opencv`. This software module is responsible for capturing raw audio signals from the PC's microphone, transforming them into spectrograms, and sending them to the FPGA for inference. The audio capture is triggered by a request signal sent from the FPGA over Ethernet, ensuring the PC only records input when the FPGA is ready to receive and process data.

Upon receiving a request, the Python script uses the `pyaudio` library to initiate real-time audio sampling at a sampling rate of 16 kHz, single channel, 16-bit depth. The recording duration is fixed at 2 seconds, producing a waveform with 32,000 samples. This waveform is then segmented using short-time framing with a frame length of 255 and a frame step of 128. Each frame is windowed using a Hann window, followed by a real-valued FFT (Fast Fourier Transform) of size 256 to extract frequency-domain features. This process results in a spectrogram, which is then log-scaled and normalized using dataset-specific mean and standard deviation values.

To match the input format of the neural network on the FPGA, the normalized spectrogram is resized to a fixed shape of 124×129 using cubic interpolation via OpenCV. The floating-point spectrogram is quantized into `uint8` format using the equation:

$$\text{quantized\_value} = \text{clip}\left(\frac{\text{value}}{\text{scale}} + \text{zero\_point}, 0, 255\right) \tag{1}$$

with `scale = 1/128` and `zero_point = 128`, ensuring compatibility with the fixed-point inference logic on the FPGA. The quantized spectrogram is stored temporarily and fragmented into multiple Ethernet packets, each embedded with custom headers for fragmentation and reassembly. A handshake protocol is used with ACK/NACK packets to guarantee data integrity and reliability during transmission to the FPGA.

This preprocessing stage plays a critical role in transforming real-world audio into an efficient and hardware-friendly format, enabling rapid and robust keyword spotting inference on the FPGA.

5.3 Ethernet-Based Data Transfer and FPGA-Side Parsing

This project utilizes raw Ethernet communication to transfer all necessary data from an external PC to the FPGA. This includes both the pre-trained model in binary format and the quantized input audio data after preprocessing. A custom Python script implements a real-time audio processing and transmission pipeline that communicates directly with the FPGA at the data link layer.

The script performs two core functions: (1) sending pre-trained model tensors extracted from a binary file to the FPGA, and (2) capturing live audio input upon FPGA request, converting it into a quantized spectrogram, and transmitting it for inference. Although model parameters and audio data are sent at different stages, both use the same transmission protocol. Data is sent using raw Ethernet frames with a custom `EtherType` field to distinguish between payload types which are either model tensors, audio input, or response messages (ACK/NACK). By inspecting the `EtherType` field, the FPGA can interpret the type of each incoming packet accordingly.

Due to the 1500-byte limit of standard Ethernet frames, both model and input data are divided into fragments prior to transmission. Each fragment is prepended with a custom header that encodes metadata such as the tensor ID, fragment index, total number of fragments, and the payload length. The first fragment of a tensor also includes the total tensor size, which the FPGA parses and uses to allocate space in DDR DRAM. To ensure

reliability, the script implements a stop-and-wait protocol. After each fragment is sent using the `send_frame()` function, the sender invokes `wait_for_ack()` to listen for an acknowledgment from the FPGA. ACK/NACK packets from the FPGA are filtered by `EtherType` and unpacked to retrieve the tensor ID, fragment index, and status byte. If an ACK is received, the sender proceeds to the next fragment. In the case of a NACK or timeout, the sender retransmits the fragment indicated by the packet's fragment index.

The data transmission sequence begins with sending model parameter tensors. Each tensor is serialized into a byte stream, segmented into fragments, and transmitted over Ethernet using the reliable transport protocol described above. Once the model has been fully transferred, the script enters a reactive mode and waits for an inference request from the FPGA. A special control frame with a reserved `EtherType` is used as a trigger signal, initiated by a button press on the FPGA board. Upon receiving this request, the script captures a short audio segment using the PyAudio library, processes it into a log-scaled spectrogram, quantizes it to 8-bit integers, and transmits it to the FPGA using the same fragmentation and acknowledgment mechanism.

On the FPGA side, Ethernet communication is managed by the Xilinx `XEmacLite` peripheral, which continuously monitors incoming Ethernet frames. Packets are filtered by tensor ID to identify whether they carry model tensor data or audio input. Audio input is handled using a reserved tensor ID, so the FPGA knows what type of data the fragment is.

Upon receiving each fragment, the FPGA verifies the fragment index against the expected fragment index. Valid fragments are either written into contiguous regions of the DRAM for model tensors or a dedicated buffer for audio input (`AudioInputBuffer`). For model tensors, a global memory offset is tracked, and each tensor's metadata (start address, size) is recorded in lookup tables for later use during inference. These lookup tables allow MicroBlaze to easily locate where each tensor is located in the DRAM and load them when needed for inference. If a fragment is received out of order, the FPGA responds with a NACK packet containing the expected index, prompting retransmission. Otherwise, an ACK is sent to confirm successful receipt. Once all fragments of the audio input are received and reassembled, a flag (`audio_ready`) is set to indicate that the input is ready for inference.

This tightly integrated communication protocol ensures reliable delivery, correct sequencing, and efficient synchronization between the PC and FPGA while minimizing overhead and maximizing throughput.

5.4 MicroBlaze Software Workflow

The main function of the FPGA program controls the complete workflow for receiving model parameters, triggering live audio capture, and performing inference using a custom hardware accelerator. It begins by initializing the hardware platform and configuring the Ethernet Lite (EmacLite) peripheral with the FPGA's MAC address. Once initialization is complete, the system enters a reception phase where it continuously listens for incoming Ethernet packets containing pre-trained model tensors. These tensors are transmitted from the host PC and stored sequentially into DRAM. Each tensor is identified by its `tensor_id`, and the system tracks the number of tensors received. This phase continues until all expected tensors are received, after which the reception loop exits.

After model reception is complete, the FPGA transitions into an inference-ready state, entering an infinite loop that continuously monitors for a physical button press. When a rising edge on the button input is detected, the FPGA sends a request packet to the PC, prompting the PC to start recording, preprocess the audio data, and send the audio spectrogram data back to the FPGA. The FPGA then re-enters the packet reception routine and waits until the entire spectrogram has been received. Once the `audio_ready` flag is set, the system launches the inference pipeline.

The inference pipeline, at each layer of the model, essentially just loads weights and biases from previously received model tensors stored in DRAM and applies them to the input tensor of that layer. Quantization parameters, including scales and zero-points, are also retrieved from DRAM, ensuring proper interpretation of the 8-bit quantized data throughout the inference steps.

The first stage of the pipeline is a convolutional layer (Conv1) that applies 32 3×3 filters to the input. For each spatial position, a 3×3 patch is extracted and processed using a custom MAC (multiply-accumulate) hardware accelerator. The accelerator supports double buffering, allowing two sets of filters and input activations to be inputted to the accelerator by alternating between two register buffers. The resulting values are added to the bias, quantized using output scale and zero-point parameters, clamped to the 8-bit integer range, and stored in an output buffer with dimensions 122×127×32.

The second convolutional layer (Conv2) operates on the output of Conv1. It applies 64 filters, each of shape 3×3×32, meaning the convolution spans 32 input channels per output filter. Each filter-channel pair is processed through the MAC accelerator, and the outputs are accumulated across channels and requantized. This operation produces a 120×125×64 output tensor, which is subsequently downsampled using a 2×2 max pooling layer with a

stride of 2. The resulting pooled tensor, now of size 60×62×64, is flattened into a 1D array and used as the input to the first fully connected (FC1) layer.

FC1 projects the flattened vector into a 128-dimensional space by computing dot products between the input vector and 128 weight vectors. These operations are also offloaded to the MAC accelerator by processing the input and weights in 9-element blocks to align with the accelerator and optimize throughput, with special handling for any remaining elements. The outputs are quantized and passed to the final fully connected layer (FC2), which computes logits for eight keyword classes. The FC2 layer reduces the 128-dimensional input vector into an 8-dimensional output using the same accelerator and quantization pipeline.

The FC2 output is then dequantized to produce floating-point logits, which are passed through a softmax function to calculate class probabilities. The predicted class corresponds to the index of the highest probability value. This result is sent to an external device via Bluetooth and the confidence value is displayed on the FPGA's seven-segment display.

Throughout the main function, careful memory management and synchronization with the host PC ensures a seamless real-time keyword recognition flow, with model initialization, audio input, and inference executed in a tightly coupled embedded environment.

5.5 Quantization of Output Activations during Inference

Neural networks are typically trained and evaluated using 32-bit floating-point representations. However, floating-point arithmetic is computationally expensive and memory-intensive, especially on FPGA platforms, which generally lack efficient native support for floating-point operations. To address this, quantization is applied to the model, converting all weights, activations, and biases into lower-precision fixed-point representations while preserving the range and distribution of the original data.

In this project, TensorFlow's quantization-aware training (QAT) framework is used to quantize all model weights to 8-bit integers (`int8`) and biases to 32-bit integers (`int32`). Each quantized tensor is associated with a scale factor and a zero-point. The scale is a floating-point value that defines the real-world value represented by a one-unit step in the quantized domain, while the zero-point is the integer value that corresponds to a real value of zero.

During inference on the FPGA, convolutional or dense (fully connected) layers utilize the hardware accelerator to compute dot products using 8-bit inputs and weights. The results are summed with 32-bit biases, producing 32-bit integer accumulations. These accumulated outputs are not directly usable by the next layer, which expects quantized 8-bit inputs. Thus,

each accumulation must be *requantized* using the appropriate output quantization scale and zero-point.

This is done by applying the following transformation to each accumulated result:

$$\text{output}_q = \text{clamp}_0^{127}\left(\text{round}\left(\frac{S_x \cdot S_w}{S_y} \cdot \text{acc}\right) + Z_y\right) \tag{2}$$

Where:

- S_x and S_w are the quantization scales of the layer's input tensor and weights, respectively.
- S_y and Z_y are the quantization scale and zero-point for the output tensor.
- `clamp` ensures the result remains within the valid 8-bit range (0–255), effectively applying a ReLU activation.

The resulting value `output_q` is stored as an `int8_t` and passed to the next layer. This requantization step is performed after each accumulation and bias addition. The scale values and zero-points are extracted from tensors received from the PC during model initialization.

By applying these transformations, the inference pipeline preserves the semantics of the original floating-point model while remaining entirely in the integer domain, enabling low-latency and resource-efficient execution on the FPGA.

5.6 Python GUI and Bluetooth Integration

To provide intuitive and interactive visualization of classification results generated by the FPGA-based speech recognition accelerator, we developed a Python-based Graphical User Interface (GUI). The GUI acts as a bridge between the hardware and the end user, receiving prediction results via Bluetooth and visualizing the recognized command through an animated stickman figure.

System Overview

The GUI is implemented using Python's `tkinter` framework for graphics rendering and `pyserial` for Bluetooth communication. It is designed to run on a PC or laptop connected to the FPGA via a Bluetooth UART module (PMOD BT2)[3]. The FPGA sends newline-terminated strings corresponding to recognized speech commands. The GUI interprets these commands and triggers appropriate animations or display responses in real time.

Stickman Controller

The central feature of the GUI is a stylized stickman drawn on a canvas. It can move in four cardinal directions (left, right, up, down) and display command-related expressions (go, stop, yes, no) in a speech bubble. This gives the project a visual and interactive interface that enhances its appeal and user understanding.

- **Directional Commands:** On receiving commands like `left`, the stickman shifts its position on the canvas by a fixed number of pixels. Movement updates the internal position state and applies transformations to each body part.
- **Speech Commands:** For non-directional outputs like `yes`, `stop`, etc., a bubble appears near the stickman's head showing the recognized word. The bubble resizes based on text length and is styled to resemble a comic-style speech bubble.
- **Bubble Behavior:** The bubble automatically disappears after a set timeout (e.g., 2 seconds), managed via `tkinter.after()` to keep the display clean and reduce clutter.

Bluetooth Communication Handling

The communication pipeline is established over a serial port (typically `COM5` or `COM6`) via Bluetooth. The FPGA sends a simple ASCII string (e.g., `"left\n"`) using the `XUartLite_Send` function in the Xilinx SDK.

On the PC side, the GUI continuously listens to the port in a background thread, using a non-blocking read mechanism. It accumulates characters until a newline is detected, indicating the end of a command. The command is then passed to the GUI controller logic for parsing and rendering.

6 Advice for Future Students

6.1 Bluetooth UART Communication with PMOD BT2

Students looking to interface FPGA systems with external controllers (like a GUI or mobile device) over Bluetooth should take advantage of the PMOD BT2 module for UART-based wireless communication. The PMOD BT2 is relatively straightforward to integrate and configure using Xilinx's UARTLite IP. One recommendation is to test communication

early in the project using a simple loopback or string echo design to ensure correct TX/RX connections and baud rate matching.

One critical insight we gained is that the PMOD BT2 module can only operate as a slave device, meaning the external host (typically a Python GUI or mobile device) must initiate the pairing and data transmission[3]. In our project, we used a Python GUI as the master, receiving predicted command strings from the FPGA through a virtual COM port created by the Bluetooth link. This design enables real-time control of GUI animations (e.g., a stickman responding to commands like "left", "right", etc.).

We advise future teams to avoid using polling for UART reception on the GUI side—non-blocking, event-based serial reading (e.g., using `pyserial` with threading or timeouts) ensures smoother integration. Also, be mindful of string encoding issues and newline characters (\n) when sending commands, as inconsistencies can break command parsing on the GUI side.

6.2 Debugging and Communication Protocols

Another key learning was the importance of having a robust communication protocol when sending model parameters and data between the PC and FPGA. In our case, we used raw Ethernet packets to transmit model tensors to the MicroBlaze soft processor, implementing a custom acknowledgment protocol to guarantee data integrity. While this low-level approach gave us fine-grained control, it also introduced complexity in managing synchronization, packet fragmentation, and reassembly.

We recommend future students clearly define and document their communication packet format early on, including headers, payload length fields, and CRC or checksum bits. This avoids late-stage debugging due to mismatches in parsing or unexpected data alignment.

Finally, having terminal output logs and debugging tools like ILA (Integrated Logic Analyzer) in Vivado proved invaluable. We suggest reserving time for setting up these tools rather than treating them as last-resort debugging options. They allow you to verify that data is correctly received, decoded, and routed in real-time across the system—critical when coordinating between hardware accelerator outputs and MicroBlaze processing.

6.3 System Integration: Start Simple, Build Up

A valuable tip from our experience is to begin system integration from the smallest working blocks—like verifying UART, DMA, or Ethernet transfers separately—before combining

them. Attempting to integrate everything at once, especially with tight project timelines, can lead to hard-to-trace errors and debugging bottlenecks.

In summary, communication reliability, modular integration, and early testing of I/O peripherals (like Bluetooth and Ethernet) are cornerstones for a successful system-level project. These insights should help future groups better structure their development and debugging process in similar FPGA-based machine learning designs.

7 Design Tree

The complete project repository is hosted on GitHub and can be accessed at the following URL:

`https://github.com/ZhenzezZ/FPGA-Speak-recognition-ACC`

The repository is structured into four main directories, each representing a major component of the system:

1. `doc/`

This directory contains the final documentation deliverables for the project. It includes the PDF version of the final report, the slides used in our final demonstration presentation, and a video recording of the full demo. These artifacts provide a comprehensive overview of the project's development, implementation, and testing process.

2. `PC_code/`

The `PC_code` folder houses all Python scripts that are executed on the host PC. This includes:

- Jupyter notebook for machine learning model pre-training, quantization-aware training (QAT), and encoding of model parameters to a binary file
- The audio capture and preprocessing pipeline using PyAudio and FFT
- Scripts to transmit model parameters and quantized audio data via raw Ethernet frames
- The interactive GUI program using `tkinter`, which visualizes prediction results in real-time by animating a stickman avatar in response to recognized commands

- Bluetooth integration code leveraging the `pyserial` library

3. `Microblaze/`

This directory contains the main C code that runs on the Xilinx MicroBlaze soft-core processor. The program is responsible for:

- Receiving model parameters and audio input packets over Ethernet
- Managing memory-mapped data structures and fragmentation reassembly
- Triggering inference using the hardware accelerator
- Sending classification results via UART to the Bluetooth PMOD module

4. `Hardware_Design/`

This folder includes all Verilog HDL source files used to implement the hardware accelerator on the FPGA. It contains:

- RTL code for the convolutional and fully connected layers
- The Verilog description of the custom MAC (multiply-accumulate) PEs
- The interface logic for AXI memory access and control signals
- Xilinx constraint files (.xdc) specifying pin mappings and timing constraints

The well-organized structure of the repository is intended to help future developers quickly navigate between software, hardware, and documentation components of the project, and to facilitate reproducibility and collaboration.

References

[1] TensorFlow, "Simple audio recognition," TensorFlow Tutorial, 2024, https://www.tensorflow.org/tutorials/audio/simple_audio.

[2] T. M. Optimization, "Quantization aware training in keras," TensorFlow Guide, 2024, https://www.tensorflow.org/model_optimization/guide/quantization/training_example.

[3] Microchip Technology Inc., *RN42 Bluetooth Module User Guide*, 2012, version 1.0r. [Online]. Available: https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/bluetooth_cr_UG-v1.0r.pdf