ECE532 Final Design Report

~~~

Black Duck Down

~~~

Wen Bo Li
Jianwei Sun
Wenyi Yin

~~~

11 April 2017
~~~

Table Of Contents

Overview

The improvement of audio processing methods in recent years has given rise to a variety of applications, most notably in the software space. Some of these applications fall into the ‘voice-training’ category, where the user produces attempts to produce sounds that fulfill some criteria, and the application provides feedback to the user on how well they were able to fulfill that criteria.

The goal of this project was to implement a similar application on a Xilinx FPGA, and to wrap that functionality into an engaging game to prolong the player’s interest and present more of a challenge. In this project, we have taken inspiration from the gameplay dynamics of both ‘Black Duck Down’ and ‘Guitar Hero’.

Our project, called ‘Black Duck Down’, is a real-time voice-training game where the player must sing specific notes in order to score points. This is game-ified by representing the notes as ducks and representing wins as killing the corresponding ducks.

Similar to existing applications, this game can be used to train the player’s pitch recognition and voice stability. During gameplay, ducks appear at random on the screen, each associated with a unique note in the range C2-C5. Players who are able to find the correct pitches faster, score higher in the game.

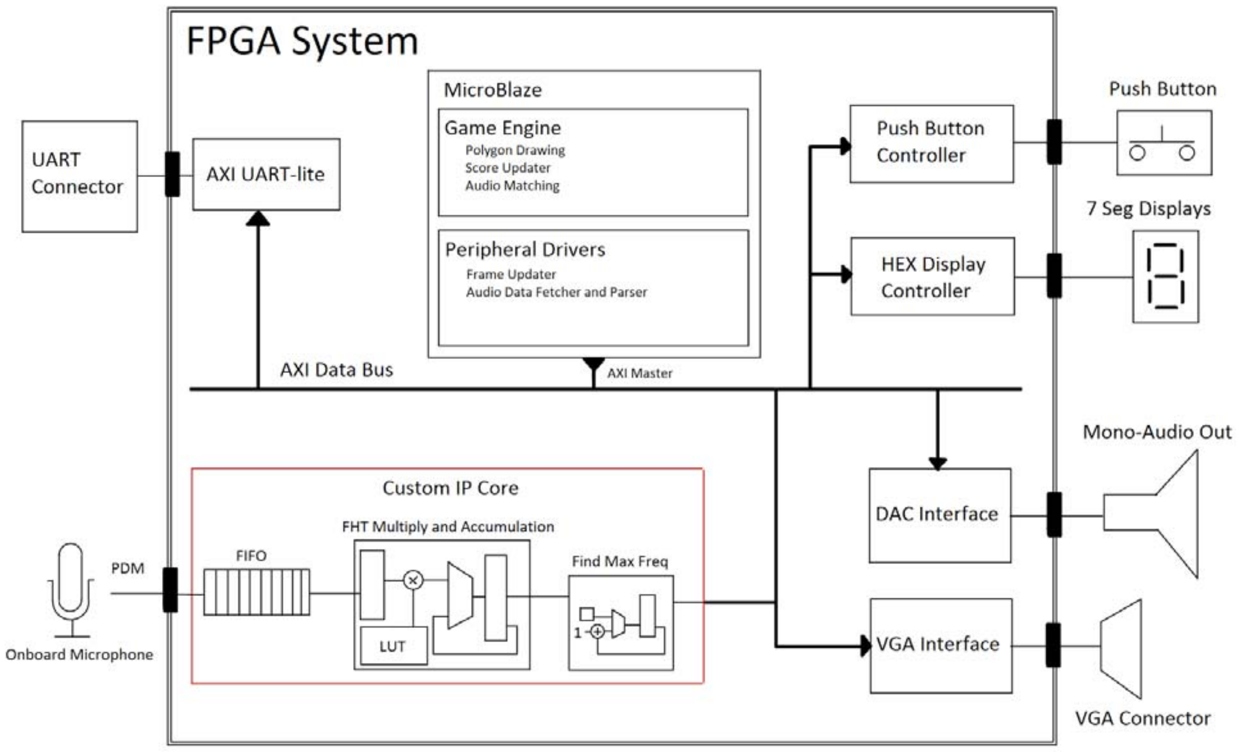
Motivation

The primary advantage to implementing this application on an FPGA, rather than pure software, is the acceleration provided by a custom hardware core. The improvement in video and audio processing time allows for extremely low-latency feedback, leading to a better gameplay experience.

The primary motivation for this project, however, was to gain experience with implementing an Fast Fourier Transform (FFT) in FPGAs. FFTs are commonly implemented in FPGAs due to superior speed, so this project was a convenient opportunity for learning more about how this is done. From a high-level point of view, this project is fun, has some useful applications, and presented a good learning experience.

Block Diagram

Figure 1. Block diagram of the final design.



IPs Used

All IP used in this project was written by the team members, excluding Vivado's auto-generated AXI peripheral Verilog code and the pre-existing UART module.

1. *Custom IP core*

   The custom IP core, freqcalccore, is an AXI slave device which does the following:
   - Interfaces with the onboard PDM microphone by sampling at a frequency of 1.024 kHz
   - Converts PDM data to PCM data by integrating over the signal, and stores the results in a FIFO buffer
   - Does a Hartley Transform (real-valued equivalent of FFT) on the PCM data, extracting amplitude and frequency information from the current audio data window
   - Makes amplitude and frequency values available via registers accessible to the MicroBlaze

2. *VGA Interface*

   The VGA interface is contained within a custom IP called display_v3.0, which is an AXI slave device. This IP produces the sync signals necessary to drive the VGA display at a 1280x1024 pixel resolution with a 60Hz refresh rate. It also handles the displaying of the game background, ducks, score, and note labels via configurable registers accessible to the MicroBlaze, which is the master.

3. *MicroBlaze + AXI Peripheral*

   The game logic is loaded onto the MicroBlaze, which communicates with all other IP via the AXI interface. The AXI Peripheral address and their offsets are used for the AXI interface communications.

4. *PWM Speaker*

   The PWM speaker, quacker, is an AXI slave device. It stores the audio data for a 'quacking' sound, and plays the sound on request.

5. *Pushbutton*

   The pushbutton module is an interface between physical pushbuttons on the board to the processor via GPIO, and is used to signal the MicroBlaze to initialize gameplay.

6. *Hex Display*

   The hex display, hexprint_1.0, is an AXI slave device. During gameplay, it displays the current score, and note-name of the last detected audio sample.

7. *UART*

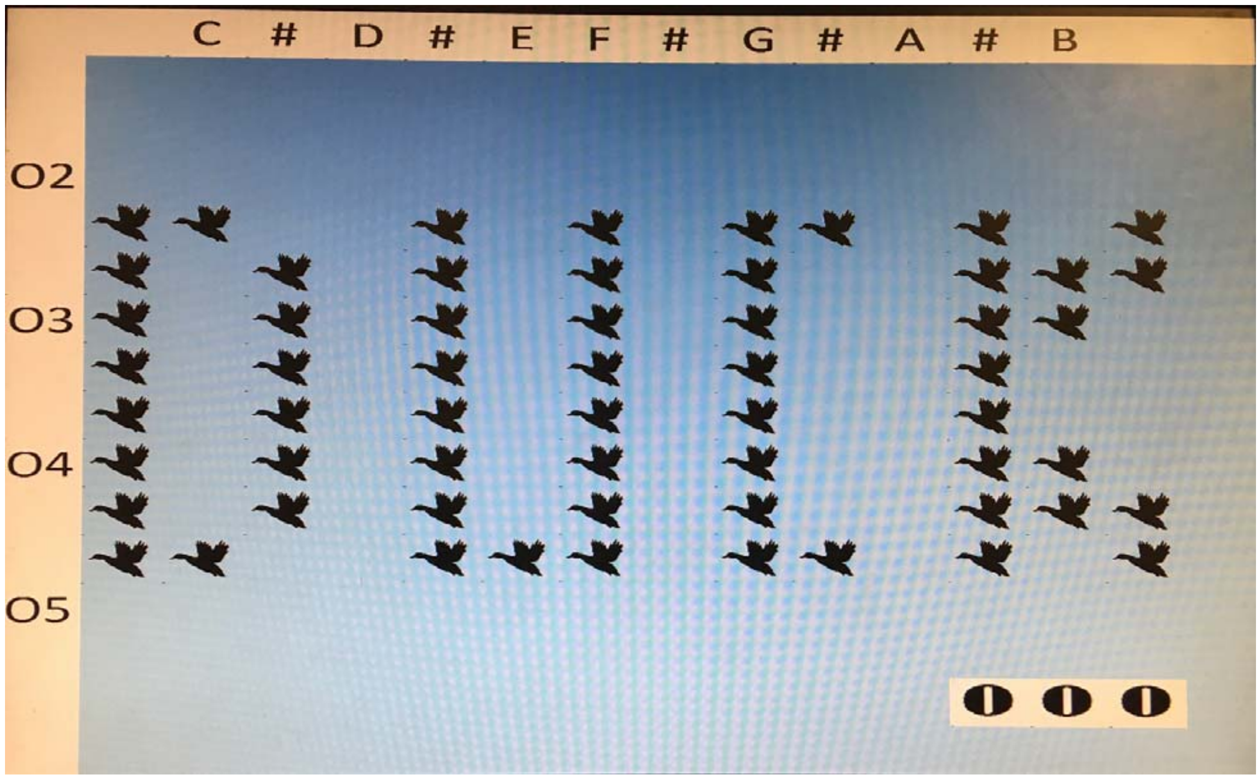
   The pre-existing UART module available in Vivado's default IP library was used [1].

Outcome

Resulting Gameplay

Black Duck Down begins with a splash screen displaying an abbreviated name of the game using duck symbols. The game begins when the player presses the BTNC push button, otherwise the splash screen is continuously displayed.

Figure 2. The start screen displayed prior to gameplay.



Upon starting a new game, the display is changed to the in-game screen which features a few different components. The main game area is a giant matrix with columns being notes with the same letter, and rows representing notes belonging to the same octave. Indicator bars appear on the top and left sides of the screen so players can see which notes must be sung. Black ducks appearing then randomly appear at a gradually increasing rate. The horizontal component of the duck's position is matched with the bar across the top to determine the note letter. Likewise, the vertical component of the duck on the screen determines which octave the required note belongs in. When the player sings a note that matches one in which a duck occupies, the duck is removed from play and a brief quacking sound is played. The score is incremented as indicated on the score counter on the bottom right of the display as well as on the HEX display on the Nexys 4 DDR board [1].

The black bar on the bottom of the screen moves according to the latest dominant pitch picked up by the microphone. This bar provides visual feedback to the player so that the player knows how to adjust her pitch appropriately to kill the target ducks.

The game continues until the number of ducks on the screen exceeds a certain threshold, after which the game transitions to the end game screen. The end game screen displays the word "OVER" to indicate the end of the game as well as show the final score in the bottom right corner. The game can be reset for another playthrough by pressing the reset button on the board [1].

Changes from Initial Design

1. Our original design included the use of an external PMOD microphone, which was unfortunately not available in time for this project. Instead, we used the on-board PDM microphone and converted its input to PCM data [1]. A major drawback of using this microphone was that the player had to sing closely to the board, as the microphone did not have a very good pickup range.

2. The original vision for the display was to have ducks that fly back and forth on the screen. Due to the lack of prior knowledge of the VGA, how many notes were to be implemented, and how exactly we wanted the ducks to appear, we decided to solve these problems by partitioning the screen into 16x16 squares of static images - this meant that no ducks moved on the screen. This modification also allowed for an increase in the speed of implementation due to the design's simplicity.

3. We were unable to implement the actual FFT algorithm, as it required operations on complex numbers. Instead, we used the Fast Hartley Transform (FHT), whose output can then be transformed into the Fourier domain to obtain frequency and amplitude information. This core was implemented in a very pipelined fashion, as it had problems meeting timing requirements due to many operations. Additionally, the window size of the PCM data used had to be decreased in order to meet timing requirements and utilization limits on DSP48 slices [1][2].

Evaluation of Results

There was one major problem against the initial goals: sometimes, an unstable voice would kill many ducks of different frequencies instead of the frequency that would have been heard by human ears. The details of the problem are described below. The other parts of the game works as originally envisioned, apart from the lack of fancy visual effects.

The Smoothing Algorithm

The smoothing algorithm is implemented as an average of the last 32 FHT readings which were above the required amplitude threshold. This simple scheme has some apparent problems:

1. When there is a pause in the human voice, frequency readings in the previous continuous segment of the voice will be counted in the first 31 readings of the current segment, even when that could have been an arbitrary amount of time in the past.
2. When there is a change in the frequency of the detected voice, all frequencies in transition between the first and second frequencies will be detected as well.

As a result, ducks with frequencies much farther away from the sung frequency are sometimes detected as hit, resulting in too many ducks being killed. In fact, if the user were to blow air very hard at the microphone, approximating white noise, all frequencies are detected by the FHT. A better algorithm would not only address the two problems aforementioned, but also detect sounds over the magnitude that are not intended as voice. One way of implementing this is through enforcing a rule whereby the player must hold a note for a specific amount of time.

Quality of Graphics

Currently, the graphics consists of static images on squares that partition the screen. Although simple and effective, it is not in the spirit of Black Duck Down for ducks to not fly around. An improved implementation would allow ducks to move around the screen and update the notes to which they correspond. This adds another level of difficulty to the game - the user must deal with the changing positions of ducks.

Hindsight and Next Steps

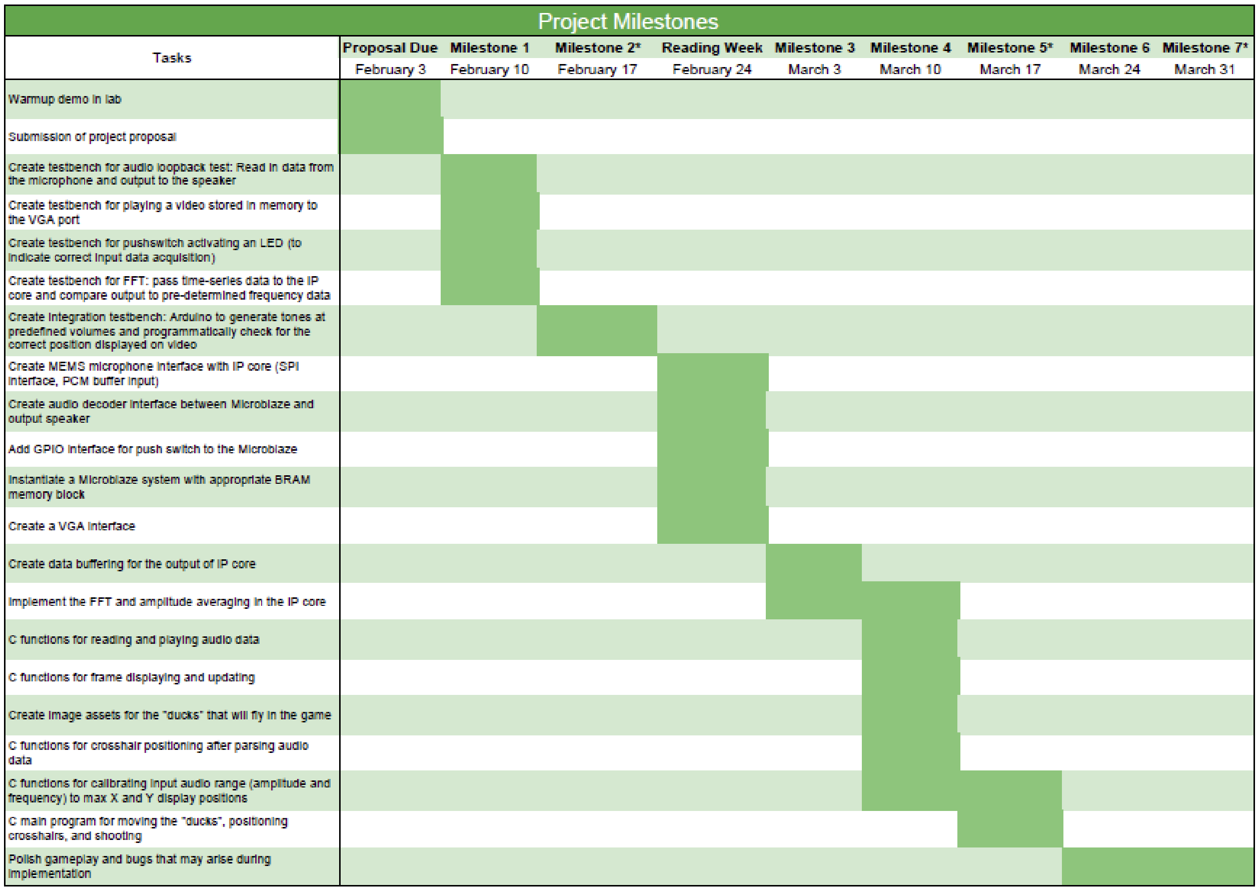
If we could start over, one thing we would do is to anticipate the problem of smoothing the frequency, and finding a robust way to handle the problem before using the current naive version to handle the smoothing. By doing this, we would have solved the smoothing challenge earlier. However, it is arguably difficult to understand the details of the problem as well as now before seeing the problem and thinking about how it relates to the current implementation. Thus, a more realistic hindsight insight would have been to integrate earlier - whenever a new block’s implementation and testing had been completed. By running integration tests earlier, integration problems such as this one would have been discovered earlier.

As next steps for someone who is taking over the project, the first step would be to find and implement a robust algorithm to solve the smoothing challenge as described in the improvements section. The other improvements as described in the improvements would also add value to the project towards it being able to be used reliably as a game.

Project Schedule

Several difficulties were encountered during the design of Black Duck Down. These problems resulted in certain tasks being delayed and caused us to deviate from the original milestone timeline. Issues in designing the FHT core, especially due to satisfying timing constraints, caused the most amount of setback, but was eventually overcome by Milestone 4. Other difficulties included working with the VGA controller, which delayed the originally projected completion date by two weeks. The other smaller subsystems were created with little difficulty and followed the proposed timeline.

Figure 3. The original schedule, which can also be found in the proposal.



	Project Milestones								
Tasks	Proposal Due	Milestone 1	Milestone 2*	Reading Week	Milestone 3	Milestone 4	Milestone 5*	Milestone 6	Milestone 7*
	February 3	February 10	February 17	February 24	March 3	March 10	March 17	March 24	March 31
Warmup demo in lab	🟩								
Submission of project proposal	🟩								
Create testbench for audio loopback test: Read in data from the microphone and output to the speaker		🟩							
Create testbench for playing a video stored in memory to the VGA port		🟩							
Create testbench for pushswitch activating an LED (to indicate correct input data acquisition)		🟩							
Create testbench for FFT: pass time-series data to the IP core and compare output to pre-determined frequency data		🟩							
Create integration testbench: Arduino to generate tones at predefined volumes and programmatically check for the correct position displayed on video			🟩						
Create MEMS microphone interface with IP core (SPI interface, PCM buffer input)				🟩					
Create audio decoder interface between Microblaze and output speaker				🟩					
Add GPIO interface for push switch to the Microblaze				🟩					
Instantiate a Microblaze system with appropriate BRAM memory block				🟩					
Create a VGA interface				🟩					
Create data buffering for the output of IP core					🟩				
Implement the FFT and amplitude averaging in the IP core					🟩	🟩			
C functions for reading and playing audio data						🟩			
C functions for frame displaying and updating						🟩			
Create image assets for the "ducks" that will fly in the game						🟩			
C functions for crosshair positioning after parsing audio data						🟩			
C functions for calibrating input audio range (amplitude and frequency) to max X and Y display positions						🟩	🟩		
C main program for moving the "ducks", positioning crosshairs, and shooting							🟩		
Polish gameplay and bugs that may arise during implementation								🟩	🟩

Figure 4. The actual schedule followed by the team.

Project Milestones									
Tasks	**Proposal Due**	**Milestone 1**	**Milestone 2**	**Reading Week**	**Milestone 3**	**Milestone 4**	**Milestone 5**	**Milestone 6**	**Milestone 7**
	February 3	February 10	February 17	February 24	March 3	March 10	March 17	March 24	March 31
Warmup demo in lab	🟩								
Submission of project proposal	🟩								
Gathered frequency data from human sources, created microphone and speaker interface		🟩	🟩						
Decided on FHT core and created algorithm, completed microphone and speaker interface			🟩	🟩					
Created inital FHT core, but failed timing constraints, created HEX display controller			🟩	🟩	🟩				
Modified FHT core to meet timing constraints, began VGA display interface				🟩	🟩				
Interface FHT with PDM microphone					🟩				
Created visual assets for displaying through VGA, began work on speaker core						🟩	🟩		
Completed speaker core, began software drivers and designing game logic						🟩	🟩		
Integrated successfully with FHT core and game logic							🟩	🟩	
Completed integration, successfully debugged, and finalized entire project								🟩	🟩

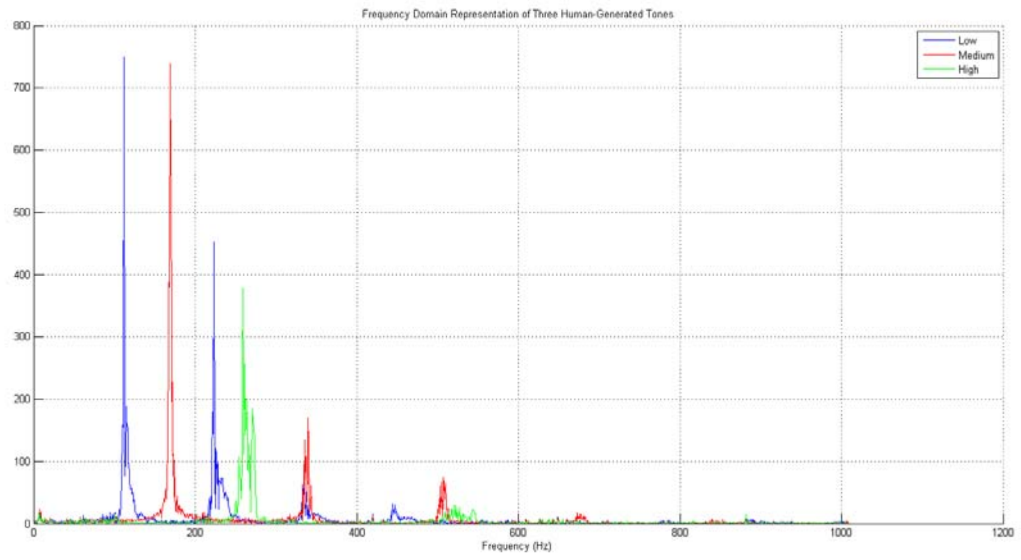
Description of Components

FHT Core

The requirements for the FHT core were defined early in the initial design of Black Duck Down. The core is responsible for collecting audio data, calculating the maximum frequency component within the audio sample, and recording the result in some registers that can be read by a microcontroller device. The FHT was designed as an AXI peripheral slave device to increase modularity and versatility, allowing it to be used in this project and others in the future.

Before designing the FHT core, we determined it was necessary to understand human audio capturing and processing through experimentation. Since the objective of the core was to determine dominant pitch from a person singing, we first recorded audio samples of various musical notes sung by individuals. These notes were then processed using the FFT command in MATLAB to understand what sort of frequency ranges were ultimately required by the FHT core. The following plot shows the magnitude plot of the gathered data against frequency.

Figure 5. MATLAB-generated frequency plot of three audio samples.



The plot reveals the various harmonics for three human generated tones. The second and third harmonics also appear in the plot, but it was decided that only the fundamental would be of interest. The plot reveals that by finding the frequency corresponding to the largest amplitude, it would suffice in calculating dominant pitch. This plot also reveals that the highest note generated was below 500 Hz, which allowed us to determine the microphone sampling rate and various windows lengths in the FHT algorithm. Furthermore, this experiment showed that frequencies below 50 Hz were very difficult to be generated comfortably by a human. As a result, those frequencies can be discarded in the FHT core, which also allowed us to remove the DC bias in the processed audio data.

A Fast Fourier Transform AXI slave core created by Xilinx does already exist in the Xilinx repository [1][3]. The motivation behind creating our own pitch detection core rather than using Xilinx's was three-fold. A major purpose of this project was to learn about building complex digital systems; we felt creating our custom pitch detection core would most strongly support this learning objective. A custom core allowed for a deeper understanding of the mathematical theory in frequency detection, as well as in-depth practice in digital design. Also, our requirement was only to determine the dominant frequency, which was much simpler than what Xilinx's FFT core could provide. Incorporating all the complexity of Xilinx's FFT was unnecessary for our application, so in the interest of FPGA resource utilization, it was not implemented [1]. Furthermore, we believed that due to the simplicity of our requirement for the pitch detection core, we could achieve faster performance than using the comprehensive FFT core from Xilinx. For these three reasons, the pitch detection core was fully custom implemented.

The FFT algorithm works on time series complex data and returns complex frequencies. Determining the dominant frequency requires normalizing the complex frequencies first, which throws out phase information. Since the phase information was not required for our purpose, we explored alternatives that could simplify the calculations. Furthermore, we did not need to work with complex valued data since all the microphone audio data would be real valued, and the output frequencies would also be real valued. Given these two additional simplifications, we looked for better performing alternatives. We found that the best candidate is the Fast Hartley Transform, which works on real valued data and provides real valued output that can be mapped directly to the equivalent output from the FFT. The FHT algorithm works on a window of time series audio amplitude data and requires a nonlinear transformation to get useful frequency data. The algorithm for determining dominant frequency from audio data is as follows:

1. Perform the matrix multiplication y = Mx on the input data
2. Perform the nonlinear transformation Y = y.^2 + flip(y).^2, element wise
3. Find the dominant frequency f = argmax(Y)

The FHT was designed as an 8-staged pipelined architecture, with a Finite State Machine generating all the necessary control signals. These stages broke down arithmetic operations so that the entire core could be run at the 100 MHz default clock signal. All the floating point numbers required in the FHT were normalized so that they could be represented by signed integers. The loss in precision is not noticeable, but results in significantly easier implementation. The FHT core is responsible for interface with the PDM microphone as well as provide an AXI interface to connect to other blocks. The AXI interface code automatically generated by Vivado was used to expose four slave registers to AXI master devices. These slave registers correspond to the following functions within the FHT core.

Table 1. Register value definitions for this IP core.

Register	Name	Purpose	R/W
slv_reg0	Busy Indicator	The register is 0 when the FHT core is idle and 1 when an operation is happening.	Read only
slv_reg1	Frequency Result	When the busy indicator is off, this register stores the frequency result of the computation.	Read only
slv_reg2	Amplitude Result	This register stores the corresponding amplitude to the frequency in the previous register.	Read only
slv_reg3	Start Register	This control register starts a new FHT computation when a 1 is written to it.	Write only

Given FPGA resource utilization requirements, the window size was determined to be 128, corresponding to audio data sampled at 1.024 kHz, the Nyquist rate for capturing our frequencies of interest. 12-bit audio was used because it provided enough resolution for our application without noticeable loss of data. A precomputed matrix, M, full of constants was stored into a RAMB18 block using the readmemh() command [3][4]. The size of M is 128 x 128, so the synchronous RAMB18 was used due to its high data storage capabilities [5]. Fast data retrieval from the memory block was not necessary due to the pipelined design of the core. The result y = Mx was computed in several steps, with the result being accumulated in another register bank. This accumulation step accounted for the majority of the computation time, but later tests revealed that its speed of operation was appropriate for our application.

Following the accumulation stage, the next stage is responsible for computing the nonlinear transformation of the data. This result involves two sets of multiplication and an addition, so it was broken into three smaller pipelined stages in order to meet timing requirements. The DSP48 blocks were used in the implementation to make used of existing efficient architecture. After this stage, the maximum frequency detection occurs. This operation reads the result of the previous stage, Y, in a linear fashion in order to determine the frequency corresponding to the largest amplitude. Since the data, Y, is symmetric and human generated tones are greater than 50 Hz, this search is reduced to only part of Y. After the results have been determined, they are stored into a result slave register, ready to be read by AXI master devices. The FHT state machine resets to the idle state, ready for another computation.

The front end of the FHT core is responsible for interfacing with an external microphone to gather input data. The on-board PDM microphone [1] was used due to its convenience. The FHT requires time series amplitude data, which would be the output of PCM audio data, but the microphone only provided a PDM interface. As a result, custom conversion logic had to be implemented. An internal FIFO of size 128 created the time domain window over which the FHT computed. On each start condition generated by writing a 1 to slv_reg3, the contents of the FIFO were copied to a register bank of 128 that was referenced when computing the FHT result. This additional step was required because new data could be pushed into the FIFO during an FHT computation stage. New data was shifted into the FIFO at a rate of 1.024 kHz, corresponding to the desired audio sampling rate. Each new datum point was generated from the summation of PDM output over 1/1024 seconds. Since the PDM microphone was driven at a clock rate of 3.125 MHz, this meant that a maximum data value of 3125000/1024 would be expected, which resulted in the design of the FIFO being 12 bits wide. The shifting of the FIFO and the summation of PDM data to generate time series audio data were designed to always be running for simplicity.

The PDM microphone interface, internal FIFO, FHT computation logic, lookup-table of constants, and AXI peripheral interface comprise the entire FHT core. The entire core is packaged together into a single AXI peripheral device, which allowed for ease of integration into the entire Black Duck Down system.

Speaker Core

The purpose of the speaker core was to play a quacking sound whenever the command was issued. The sound was required by the game in order to create a more entertaining environment. The speaker core, called quacker, interfaced with the PWM 3.5 mm mono audio jack [1] as well as through the AXI interface. The design of this core required audio assets of the quacking sound to be played on command. The quacking noise was obtained through free music samples online and then downsampled to 8-bit 8 kHz audio for ease of use. The amplitude data then was loaded into RAMB18 block RAM using the readmemh() command [3] [4].

An internal Finite State Machine was used to handle the logical progression of events: transitioning from idle state, loading data from the RAMB18, outputting the data to PWM, and resetting. The PWM frequency was set to be 80 kHz, which was high enough so that PWM noise would be filtered out by the on-board analog 4-th order Butterworth low pass filter. Generating the PWM signal was accomplished with two counters that handled when the next 8-bit audio datum would be read from memory and how long each PWM pulse would stay high. Together, these counters allowed for the transformation of the time series audio data into PWM that could directly be output on to 3.5 mm audio jack.

Generic third party speakers with a 3.5 mm audio jack interface were used as part of the development, testing, and final demonstration of Black Duck Down.

Pushbutton Core

The purpose of the pushbutton core was to interface with the physical push buttons on the Nexys 4 DDR board [1]. The push buttons allowed for a user-friendly physical interface, and controlled the start and resetting of a game instance. In a run of the game, the starting splash screen is displayed until the user presses the push button BTNC, after which the game commences.

The push button interface was built as an AXI peripheral, which allowed for easy implementation with the Microblaze system. The peripheral continuously samples the push button state and latches in the slave registers when the push button is pressed. The AXI master is responsible for polling the slave registers for a push button event, and then clearing the register so that subsequent button presses could be registered.

Hex Display Core

The on-board HEX displays [1] provide a useful visualization of data during development or playing the game. The HEX display format is custom so that during development, useful values could be displayed; and during game play, scores and parsed notes could be shown. The HEX controller was designed as an AXI peripheral to facilitate ease of use and reuse in future applications.

Eight slave registers store the desired values of each digit. The HEX display controller multiplexes between the eight available digits at a rate of 2 ms per digit. One counter was used to repeatedly count to 200000, which corresponds to 2ms on a 100 MHz clock. This counter reset every time it hit 200000, and incremented an anode select register. This register determined which of the eight HEX digits are active, and is also used to index from the eight slave registers to get the corresponding value to display on the selected digit. The value from the slave registers passes through a HEX encoding look-up table, with the selected encoding appearing on the cathodes of the HEX displays. The controller continually multiplexes as long as the controller is enabled.

VGA Display Core

To simplify the VGA output implementation, which can only write to the screen in a pixel-by-pixel, left-to-right, top-to-bottom fashion (in that order), we chose a simple way to decide what a pixel should display with little combinatorial logic.

We divide the screen into 16x16 blocks, where each block is 80x64 pixels. Next, we assign one register to one block on the screen - reg 0 corresponds to the top left block on the screen, and reg 255 corresponds to the bottom right block on the screen. Each block’s register value is used to determine what should be displayed on that block. There are a few possibilities for what can be displayed, which are:

- A picture of a duck silhouette.
- The background, which is a light blue colour. The colour code is 12'h9cf.
- A digit, used to display the score.
- A note name.
- An octave number.

All of these pictures are defined as 80x64 arrays of 12-bit colour values, which are transformed into ROMs during implementation. This approach is less memory-intensive than storing entire frames of colour values for every possible configuration of the screen. All blocks in the topmost row and leftmost column of the screen are used to display the note names and octave labels. Essentially, all other blocks on the screen are assigned to a specific note in a specific octave, and if a duck appears at a block, the player must sing the note associated with the position of the block.

Figure 6. Screen during gameplay.

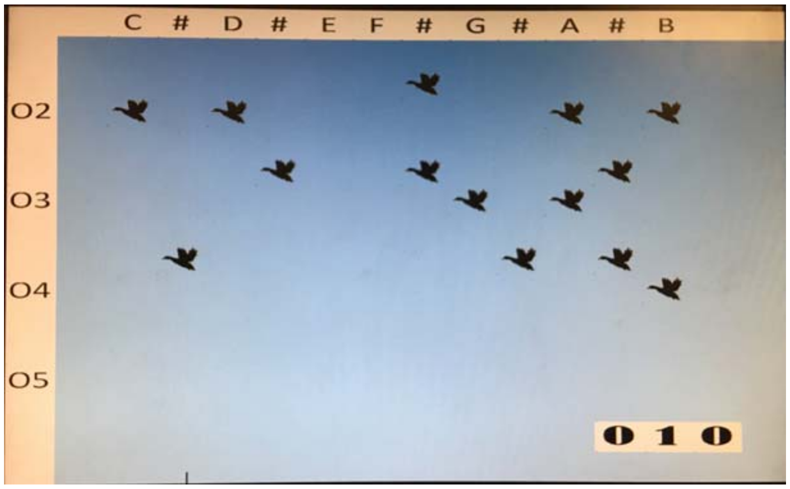


Table 2 - AXI slave register meanings.

Register Value	Meaning	Specific Meaning
0	If this is a square which could be a duck, indicates whether there is an alive duck here.	No duck (blue background)
1		Alive

All game elements can be seen in figure 6, which is composed of static elements. Clearly, there is only a finite number of different blocks which can be controlled by keeping track of state in the processor. Table 2 shows how the AXI slave register is controlled by the processor to set the dead/alive state of the ducks in the game screen.

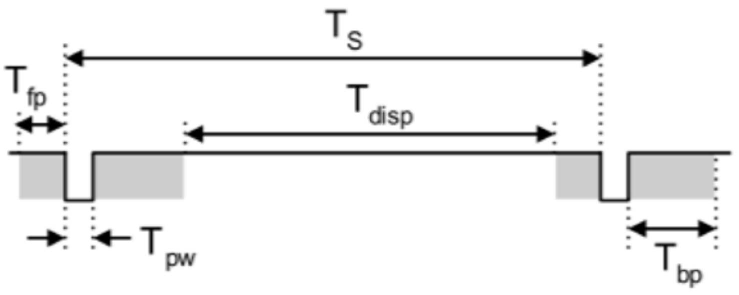
VGA Sync Signal Generator

A submodule, vga_controller, produces the necessary signals to drive a 1280x1024 pixel resolution at a 60Hz refresh rate, which is a supported resolution in most monitors. A 100 MHz clock is used to control the VGA display [1]. The signals are:

- HS - HSYNC, a pulse signal. When high, indicates that a new line on the screen is now being written.
- VS - VSYNC, a pulse signal. When high, indicates that the entire screen has been written.

A set of values for the period of all display and porch were used. The below figure shows how a sync signal is expected to look like.

Figure 7. Pulse and porch windows for a sync signal [1]



Using the HS and VS values, it is possible to determine the row and column position of the pixel that is currently being written [1]. At each VS assert, we reset the row counter to 0. At each HS assert, we increment the row counter by 1. At each pixel clock edge, we increment the column counter by 1 (wraparound at 1024). These values are then used to index into the stored colour value arrays in order to extract the correct 12-bit colour for that pixel. From this information, we can also determine the row and column position of the block we are currently writing to, which can take on values from [0, 0] to [15, 15].

Additional Blocks for the Display of Score and Note Names

Some blocks on the screen were assigned to location-specific types of pictures.
The blocks in the topmost row and leftmost column were assigned to display note names and octave names at all times. Therefore, a separate hardware module, disp_labels, was created to handle the assets for note and octave names - essentially, it defined colour value arrays for each possible digit and letter, and took as input the row and column values of the current pixel being written. Its output was a 12-bit colour value for the current pixel. Then, in the main display module, if the current block being written was in the topmost row or leftmost column, this 12-bit colour value was chosen as the colour to be displayed.

In a similar fashion, a module disp_score was written to handle the colour values for the player's current score, which was displayed over 3 blocks in the bottom right of the screen.

Modularity

The VGA display is segregated from the game logic, and does not attempt to retain the state of the game. At all times when the program is running on the board, the VGA display module expects values of AXI slave registers to be constantly updated according to the state of the game, and outputs the correct display onto the screen faithfully.

Game Engine inside MicroBlaze

The game is separated into 3 stages:

1. The start screen
2. The game
3. The end screen

The start screen and the end screen are trivial screens in which the game play is ended and waiting to bet reset and restarted upon command. The core gameplay operates as an infinite loop of interactions between the random generation and the targeted killing of ducks. These opposing forces determine the gameplay until the user fails to keep too many ducks from being on the screen at the same time - this again transitions the game into the end screen.

The gameplay sequence is as follows:

- A start screen is displayed to the player, with the letters 'DUCK' written to the screen
- The MicroBlaze waits for a button press from the player, signaling that the player is ready to begin the game
- The main game screen is displayed to the player, which is initially blank. The borders of the screen consist of note and octave labels, and the initial score in the bottom right of the screen is 0.
- The core gameplay begins looping:
    - Every N loops, a new duck is randomly generated and displayed on the screen according to its note name and octave label
    - On each loop, frequency and amplitude data is read from the FHT core
    - If the frequency reading corresponds to a duck currently displayed on the screen, 'kill' the duck by removing it from the screen, play a 'quacking' sound on the speaker, and increment the score on the screen.
    - Every time a new duck is generated, decrease the amount of time until the next duck is generated (up to a lower limit).
- The player loses when there are more than 30 ducks on the screen at one time.
- An end screen, with 'GAME OVER' displayed, is shown to the player.
- To restart the game, the player can press the CPU reset button on the Nexys 4 DDR board [1].

Start and End Screens

The start and end screens are implemented as predefined arrays, containing register values to be written to the VGA core. These arrays are defined in the header file duck_list.h. It is trivially easy to display these screens - one simply loops through the array and writes its values to all 256 VGA interface registers.

Core Gameplay

When the game begins, a custom data structure called DuckList is initialized to keep track of how many ducks are on the screen, and which note each duck is associated with. The DuckList has the following attributes:

- regs, an array of unsigned shorts, size 256. Stores the status of each square on the screen.
- emptyVec, an array of unsigned shorts, size 256. Stores in any order the indexes of the squares on the screen which are currently blank AND can display a duck.
- emptySize, an unsigned int. Keeps track of the number of ready-to-spawn duck squares stored in emptyVec.

This data structure has the ability to both spawn ducks randomly and kill specific ducks on the screen in O(1) time. The reason is because regs stores all ducks in order, including the alive ducks, and emptyVec stores all ready-to-spawn ducks, and can remove any duck in O(1) by swapping with the last suck duck inside. See complete implementation in TODO: refer to github.

With the duck records contained within DuckList initialized, the game runs by continuously polling the FHT for its frequency and amplitude readings. At the core of the logic, the loop checks whether the reading's amplitude passes a threshold (which effectively ignores readings of background noise), and whether the frequency is within range of a particular note. If so, it registers the kill with the DuckList and writes to the AXI slave register for the corresponding display score to update the display. However, in order to avoid the many frequencies which can be detected in the noisy human voice, a smoothing mechanism was implemented. It is implemented as an average of the last 32 FHT readings which were above the required amplitude threshold.

Description of Design Tree

BlackDuckDown - Root directory of project

- **Peripherals** - Directory containing all custom peripheral cores
  - **Display_3.0** - Custom VGA controller IP core
  - **Frequency_calculation_core** - Custom Fast Hartley Transform core
    - **FHT_LUT_VALUES.txt** - Lookup table constants
  - **Hexprint** - Custom core for printing to on-board HEX displays
  - **Pushbutton** - Simple core for latching push button states
  - **Quacker** - Custom core for generating quack sounds

- **Game** - Directory containing the integrated design
  - **Constraints.xdc** - Constraint file specific to the XC7A100TCSG324-1 FPGA
  - **Game.v** - Definition of the generated block design
  - **Game_wrapper.v** - Instantiation of the integrated design

- **Software** - Directory containing game logic running on the Microblaze soft processor
  - **Definitions.h** - Global game constants
  - **Duck_list.h** - Data structure definitions for spawning and killing ducks
  - **Main.c** - Integrated game code

Tips and Tricks

1. Integrate and do integration tests (functionality test) early: this can help you find integration problems earlier so you have more time to solve them.
2. Agree on the interface definitions down to the very specifics between all main modules before separately implementing them.
3. Xilinx discussion boards can be helpful when debugging very specific issues.
4. The SDK can be buggy at times. Always delete your SDK folders anytime a major change occurs in your project block diagram, and rebuild your project from scratch.
5. Timing requirements can be a nightmare if your custom IP core needs to do complex arithmetic operations. Always pipeline your operations so that you don't have timing issues.

References

1. Nexys 4 DDR Manual. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual
2. UG479 - 7 Series DSP48E1 Slice. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf
3. UG473 - 7 Series FPGAs Memory Resources. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf
4. WP231 - HDL Coding Practices to Accelerate Design Performance. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp231.pdf
5. WP335 - Creative Uses of Block RAM. [Online]. Available: https://www.fpgacentral.com/fpga-whitepaper/xilinx/wp335-creative-uses-block-ram