“Move-O-Phone” Movement Controlled Musical Instrument

ECE 532 Project Group Report

James Durst ([redacted]
Stuart Byma ([redacted]
Cyu Yeol (Brian) Rhee ([redacted]
April 4th, 2011

Table of Contents

1 – Overview

1.1 – Project Motivation

Due to the recent trend in creating devices that allow the playing of games using movement rather than a traditional joystick, controller, or keyboard, we felt that a project that followed this idea would be interesting. This led us to the idea of using movement to control a musical instrument, while removing a physical instrument from the equation.

1.2 – Goals

The goal of this project was to create a system which would allow the user to control musical tones using body movements. This was to be done by having the user wear brightly coloured armbands so that video processing could be used to track movements. The angle formed between the 'line' with the two armbands as endpoints would then be used to decide on a tone to play, which would be created with a tone generator.

1.3 – System Block Diagram

See Appendix A for the System Block Diagram.

1.4 – IP and Hardware Descriptions

Hardware / IP	Core Function	Creator(s)
video_to_ram	This core interfaces with the FPGA's video decoder chip, capturing video data from the camera, and then stores this data frame by frame into the DDR_SDRAM	Jeffrey Goeders, Durwyn D'Silva, and Chirag Ravishankar (Group responsible for the ECE532 2010 project "Virtual Pong")
paddle_detector	This core was modified from an existing IP to scan the video frame data in RAM to find the first and last instances (pixels) of two distinct colours, which are then written to a location in RAM to be read by the microblaze processor	Original verilog code by the ECE532 "Virtual Pong" group. Edited by our group to detect first and last points of two colours, rather than the last point of two colours in each half of the screen.
audio_0 AC97 interface	This is an interface to the onboard audio chip, which outputs sounds to the line out port on the board.	Embedded Computing [1]

audio_0 custom hardware logic block	This core is now primarily composed of our own custom logic block, only using the original interface described above. The custom logic allows control over the frequency and volume of a generated tone, shifting of the tone down one octave, and adding vibrato.	Group
Microblaze	Soft processor core for initializing the video interface, GPIOs, as well as translation of coordinates from the paddle_detector core into angles or distances for use in simulating instruments, control of the audio core	Xilinx. The program running on the processor was implemented by the group.
video_out	System which reads the video data from the RAM and writes it to the VGA port for output to the screen.	Xilinx
dlmb and dlmb_ctrl	Data memory and controller for data memory.	Xilinx
ilmb and ilmb_ctrl	Instruction memory and controller for instruction memory.	Xilinx
plb_v46	Several PLBs which interface the Microblaze to other IP cores in the system, including memory and GPIO, the paddle_detector to the DDR_SRAM, the video_out core to the DDR_SRAM, and the video_to_ram core to the DDR_SRAM	Xilinx
DDR_SDRAM	A multiport memory controller module that allows for multiple cores to interface with the RAM over PLB busses	Xilinx
xps_gpio	Used for reading and writing of signals between the audio core and Microblaze	Xilinx
uart_uB	A uart connection allowing the Microblaze to communicate with a computer terminal via RS-232 serial connection	Xilinx
Software	Program running on the Microblaze. The main portion of the code is a loop that translates coordinate data into signals for the audio core in various ways to simulate multiple instruments	Group
Video Daughter Board	Allows for a video decoder interface	Digilent
Video Camera	Streams video data to the video daughter board decoder via composite cable	
Speakers	Allow for audio output from the system from the FPGA board's 'line out' or 'amp out' ports	

2 – Outcome

2.1 – Results and Successes

The team considers the project a success. A fully functional system was developed which translates user's body motions into musical notes, enabling the user themselves to become an "instrument".

The system works by tracking LED lights attached to the hands or limbs of the users. Since it works by picking up certain colors in the video stream, the LEDs were necessary to provide a contrasting point of color that the system could detect accurately. Tracking of the points was quite successful, with fairly smooth translations between musical notes.

At first, the system could only accommodate one movement, a waving motion, tracking points on the user's wrist and shoulder. The angle from the horizontal formed by the line between the points was translated into one of eight notes on the Major scale. After the success of the newly dubbed "Armophone", two other mainstream instruments were emulated. A trombone, tracking points on both hands of the user, created notes based on the distance between the points. A guitar-like concept used a similar method, but only playing a note if one point crossed a horizontal based on the other point (simulating a strumming motion). Additionally, covering up one point will cause any note to stop playing.

All three instruments were successfully tested. The team proceeded to add a form of concurrency to the system, allowing two instruments to be on the screen and playing simultaneously. Each instrument uses different coloured LEDs so the system can differentiate between them.

Several other features were added to the custom hardware audio core. Initially, the audio core used different frequency PWM signals to generate the different notes to be played. This was useful for testing purposes, but begins to hurt ones ears after a while. In order to do away with the incredibly annoying PWM sound, a sine wave was generated in hardware using 32 data

points. These points are fed into the AC97 audio core in a loop at the required frequency, producing an approximate sine wave that is much more pleasing to the ear.

Additional features in the custom audio core include vibrato and octave switching. Switches on the VirtexII board allow the user to switch between a high and low octave, allowing one to play a large range of notes, and opening options of one instrument playing a melody and another playing bass. Another switch allows users to play with a vibrato, achieved by slightly varying the frequency of the sine wave going into the AC97 core. These features allow for a more diverse musical experience.

2.2 – Future Work Possibilities

There are a few suggestions we would like to add that could be used to further improve upon this project. One of the issues at the moment is that, due to the fact that the tones we are producing are discreet digital signals with different sampling rates, their waveforms cannot simply be added and output to a single channel. If the signals could be created with identical sampling rates, or converted to analog signals and then combined, this issue could be solved. Because the stereo audio core only has two channels, not being able to mix audio signals means that only two instruments can be played simultaneously. Additional video processing cores could be added to the system in order to detect more instruments, or the current core could be expanded to detect more colours. One last area of improvement we could suggest would be the use of pre-recorded audio clips for instruments so that theirs sounds are distinct from one another.

3 – Description of Design Blocks

3.1 – Video to RAM Block (video_to_ram)

This block was provided by the group responsible for last year's "Virtual Pong" project. It was used unedited. Full documentation is available in that group's final report [2].

3.2 – Point Detection Block (paddle_detector)

The original version of this block was provided by the group responsible for last year's "Virtual Pong" project. Documentation of the original code is available in their report [2]. Although the interface – inputs and outputs – of this hardware block were not changed, the internal workings of it were. In its original form, this block scanned each pixel in the video data and compared its RGB values to a set of values written into RAM by the MicroBlaze. This comparison was used to detect the last instance of two distinct colours present in the each of the left and right halves of the screen.

Our edit of this core causes it now to use those comparisons to instead detect the first and last instances of each colour throughout the entire video frame. It also now requires that it sees a sequential series of pixels of the same colour in order for the pixel to register, helping to rule out single pixels of noise that may be present in the data. As in the original code, the four detected coordinate points are then written to memory to be read and used by the MicroBlaze processor at its discretion.

3.3 – Audio Core (audio)

The custom hardware audio core is used by the system to connect to the on-board AC97 audio chip, in order to play the musical notes. The core provides two audio channels, left and right, for a maximum of two instruments play at any given time. Functions of the core include:

1. Generating a 32 point discrete sine wave
2. Outputting the sine wave to the AC97 chip at the correct frequency as dictated by software control registers
3. Allow the volume to be adjusted on either channel
4. Allow the user to enable vibrato
5. Allow the user to shift between high and low octaves

The entire core is described in detail below.

3.3.1 – Audio Core Software Control and Hardware Description

The audio core is controlled by software running on a Microblaze processor through software control registers. A GPIO block connected to the PLB bus provides the control registers. The audio core relies on two control registers, one for the left channel (Lcontrol) and one for the right channel (Rcontrol). Each register setup is identical.

Bit 31: Activate	Bit 30..24: Unused	Bit 23.. 4: Volume	Bit 3..0: Note

Figure 1. Audio core software control register.

The activate signal turns the channel on or off. The note signal is a four-bit signal fed to a frequency decoder module, which decodes the note into one of 16 different notes as a frequency value. Octave shifting and vibrato are accomplished in separate modules taking the frequency value as input. The Volume is a 20-bit signal that is attenuation rather than amplification. Signals about to be output the AC97 chip are shifted (divided) by the amount specified in Volume. Bits 30 to 24 are unused. A diagram of the audio core is provided in Figure 2 below.

The top-level file is audio.v. Module processing.v houses all of the audio processing steps. The frequency_decoder module takes as input the Notes signal from the control registers and translates this into one of 16 distinct notes, as a period in clock cycles. The module outputs two signals, one for the left channel and one for the right channel. These are then fed into octave_shift.v, which, if LOctave or ROctave is high, will divide the periods so that the frequencies are shifted into the next octave range.

In vibrato_generator.v, if Lvibrato or Rvibrato is high, the corresponding right or left channel signal will be varied slightly in frequency to simulate a vibrato. This is accomplished by using a generating a PWM signal of a period of 20 million clock cycles and amplitude 3500. This is added to the incoming frequency value, essentially compressing and stretching the sine wave periodically to simulate a vibrato.

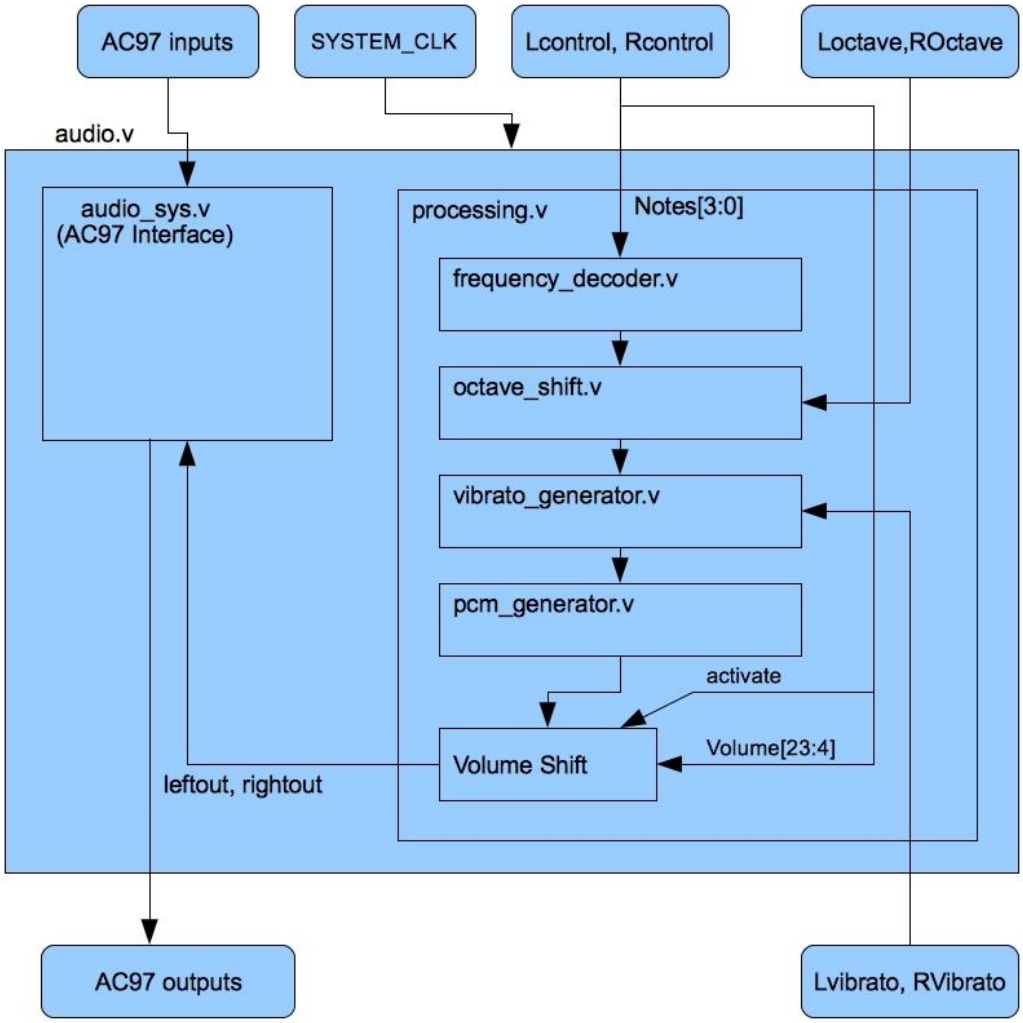


Figure 2. A diagram of the custom audio core.

The final frequency value is fed into pcm_generator.v, which generates the actual sine wave. It is here that the activate signals from the control registers decide whether or not the sound in either channel is to be played. If it is, the signal is attenuated by dividing (shifting) by the amount specified in the Volume portion of the control registers. So, a volume input of zero means full volume.

The audio_sys.v module provides a low level interface to the AC97 chip. The team acquired this module from Embedded Computing [1]. It takes several inputs and outputs that are board level connections, shown in the constraints file (system.ucf, shown in Appendix B). The other two

inputs are the two channels, which are fed the 20-bit leftout and rightout signals generated by processing.v.

3.3.2 – Sine Wave Generator

The pcm_generator module warrants some additional explanation. A discrete sine wave is generated to do away with a simple PWM signal for sound output. A PWM is sharp and rather annoying, while a sine wave is smoother and more pleasing to the ear. In order to generate a continuous signal, 32 data points were calculated and hard coded into the system, with a maximum value of 524287. This value is half of the maximum value accepted by the AC97 core (a 20 bit signal). Since all values need to be positive for the AC97 chip, the sine wave is given a DC offset of 524287 before being fed into the AC97 chip after volume control.

3.4 – Microblaze

An instantiation of the MicroBlaze v7.10d core from the Xilinx IP library.

3.5 – Video Out

An instantiation of the XPS TFT v1.00a core from the Xilinx IP library.

3.6 – Local Memory Bus and LMB Control Blocks

Instantiations of the Local Memory Bus (LMB) 1.0 v1.00a core and the LMB BRAM Controller v2.10a from the Xilinx IP library.

3.7 – Processor Local Bus Cores

Instantiations of the Processor Local Bus (PLB) 4.6 v1.03a core from the Xilinx IP library.

3.8 – DDR SDRAM Multi-Ported Memory Controller

An instantiation of the Multi-Port Memory Controller (DDR/DDR2/SDRAM) v4.03a core from the Xilinx IP library.

3.9 – General Purpose IO

Instantiations of the XPS General Purpose IO v1.00a core from the Xilinx IP library. Since the audio core is not itself a PLB peripheral, it is connected to the MicroBlaze through a GPIO.

3.10 – UART

An instantiation of the XPS UART (Lite) v1.00a from the Xilinx IP library.

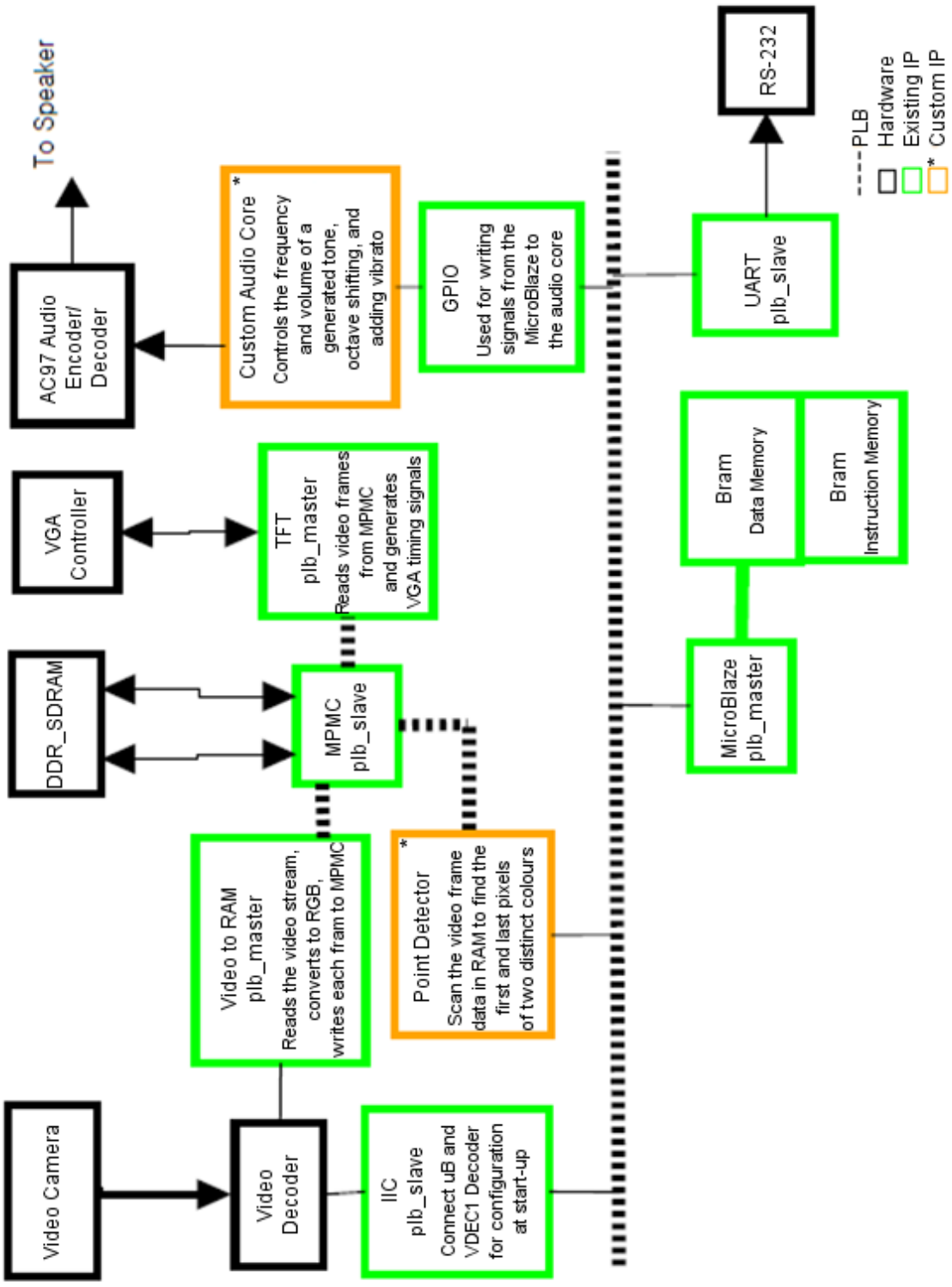
3.11 – Software

The software which runs the system is composed of an initialization section at the beginning for the video decoder and GPIO. Once initialization is complete, the program enters a loop where it acquires the coordinates of the tracked points from memory. One of these sets of points is passed to a function which calculates an angle formed by them, and writes to the GPIO to control the audio core to produce a musical note corresponding to the angle. If the two points are detected to be in close proximity to each other, then it is likely that in fact only point of the LEDs is being detected, and so the audio is turned off.

The other set of points is used in one of two different ways. To act as a trombone like instrument, the distance between the points is calculated, and a tone is generated based on this distance. In the other case, the points act as a guitar – in this case, the tone is determined by the horizontal distance between the points, but a tone is only generated if one of the points is detected to have moved above or below the other, simulating the strumming action of playing a guitar. This is done by toggling a variable which keeps track of whether the last 'strum' was up or down. The duration of the tone is implemented using a decrementing counter, which is reset to a default value each time a strum is detected.

Primarily for debugging purposes, the software writes over the video frame data in memory in order to add small coloured crosshairs at the points which it is detecting.

Appendix A – System Block Diagram



Appendix B – Audio Core Constrains from system.ucf

```
##audiocore outputs to ac97 chip
NET "audio_0_ac97_sdata_out_pin" LOC = "E8";
NET "audio_0_ac97_sdata_in_pin" LOC = "E9";
NET "audio_0_ac97_synch_pin" LOC = "F7";
NET "audio_0_ac97_bit_clock_pin" LOC = "F8";
NET "audio_0_audio_reset_b_pin" LOC = "E6";
NET "audio_0_ac97_sdata_out_pin" IOSTANDARD = LVTTL;
NET "audio_0_ac97_sdata_in_pin" IOSTANDARD = LVTTL;
NET "audio_0_ac97_synch_pin" IOSTANDARD = LVTTL;
NET "audio_0_ac97_bit_clock_pin" IOSTANDARD = LVTTL;
NET "audio_0_audio_reset_b_pin" IOSTANDARD = LVTTL;
NET "audio_0_ac97_sdata_out_pin" DRIVE = 8;
NET "audio_0_ac97_synch_pin" DRIVE = 8;
NET "audio_0_audio_reset_b_pin" DRIVE = 8;
NET "audio_0_ac97_sdata_out_pin" SLEW = SLOW;
NET "audio_0_ac97_synch_pin" SLEW = SLOW;
NET "audio_0_audio_reset_b_pin" SLEW = SLOW;

##audiocore connections to DIP switches for vibrato and octave control
NET audio_0_LVibAct_pin LOC=AC11;
NET audio_0_LVibAct_pin IOSTANDARD = LVCMOS25;
NET audio_0_LOctave_pin LOC=AD11;
NET audio_0_LOctave_pin IOSTANDARD = LVCMOS25;
NET audio_0_RVibAct_pin LOC=AF8;
NET audio_0_RVibAct_pin IOSTANDARD = LVCMOS25;
NET audio_0_ROctave_pin LOC=AF9;
NET audio_0_ROctave_pin IOSTANDARD = LVCMOS25;
```

Resources

1. Embedded Computing – Audio Core:

   http://embedded.olin.edu/xilinx_docs/projects/audio-v2p.php

2. “Virtual Pong” Documentation:

   http://www.eecg.toronto.edu/~pc/courses/432/2010/projects/virtualpong.pdf