Gesture-Recognition System for Music Playback Control

6.111 Final Project Proposal

Team: Jenny Li, Shana Mathew

1. Overview

We intend to develop a gesture recognition system that allows users to control their music playing experience solely through hand movements. Users will control play/pause state, volume level, and music selection through our handheld/wearable system that track user's hand acceleration and rotation in 3D space. For example, an upward lifting motion should be interpreted as increasing the volume of the song while a wave of a hand towards the user's right tells our system to fast forward 10 seconds of the current song.

An external board fitted with a Teensy/ESP328, IMU, and gyroscope will track user's hand movements (acceleration and rotation) and interpret the user's movements as a specific command to modify music playback. Song data bits will initially be stored in the SD card memory as a minimum viable product and later altered to stream through a Raspberry Pi and laptop. Song information such as song name and artist will be displayed on a monitor display along with the play pause signs, volume level, and real-time updates on the progress of the song.

Reach goals for this project include allowing users to tune bass and treble frequencies to enhance their music listening experience. In addition to that, we will attempt to implement stereo L/R pairing with lab speakers to best simulate a surround sound or multi-directional listening experience.

Below is a breakdown of the physical components to be used, the visualization of tasks and modules, and elaboration on task delegation and possible challenges for our project.

2. Hardware Components

- **a.** Nexsys 4 Development board
- **b.** Teensy (or ESP328) - This will power the IMU and gyroscope components.
- **c.** IMU - An IMU with a built-in 3-axis gyroscope will be used to capture user's x,y,z acceleration and rotation which will be sent serially to the Nexsys 4 Development Board in 2's complement form.
- **d.** Display Monitor - A display monitor will be used to display song information along with playback status and volume information. Monitor should be updated once every 65 MHz clock cycles to ensure a proper song progress update of once per second.
- **e.** Raspberry Pi - A Raspberry Pi will be used to stream song data information from a laptop to the Nexsys 4 Development Board. This is the next step once song information can be reliably streamed via its storage in the SD card.
- **f.** SD Card - An SD card will be used to store song data (actual song bits, artist, title, meta data) and be read-accessible by the Nexsys board to look up songs.

3. Modules

- **a. Sample Clock** (Jenny)
  Sampling clock that oversamples incoming IMU data by a parameterized value to be set by developers. This sampling clock runs off of the 65 MHz system clock

input and outputs an enabled signal that other modules can operate off of. This sampling clock will be specific to the IMU and gyroscopic data collection. Individual testing of this module will be done via a test bench to verify that the sampling clock asserts at correct intervals relative to the 65 MHz clock.

b. **Data Collection** (Jenny)
The data collection module is a receiver that receives the IMU and Gyro data-bit streams one after the other and stores it in a 120 bit shift register with start and stop bits delineating the beginning and end of each byte. Acceleration and gyro data for one axis is 16 bits with least significant bit of the least significant byte denoting the start of the x-direction acceleration followed by the y-direction acceleration, and finally the most significant bit of the most significant byte denoting the end of the z-direction acceleration. This modules takes in the system clock, the sampling clock's enabled signal, and the data stream in order to output two 60-bit IMU_data and gyro_data registers and a 1-bit done signal. We will be streaming in the acceleration data first from the sensor data registers before streaming in the gyro data since both data streams cannot be sampled simultaneously. After concatenating all 120 bits of data and asserting done, we will then split most significant 60 bits of data into the gyro_data and the lowest order 60 bits of data into the IMU_data. Testing method for this module includes visualizing the transmission of bits and recovery period on an oscilloscope.

c. **Parser** (Jenny)
The Parser module pulls the x, y, and z values from the 60-bit inputs IMU_data and gyro_data and stores the x, y, z values for each in 16-bit output registers. The module also takes clock_65mhz as an input. To test this module, feed in the given data arrays and use the waveform viewer to make sure x, y, and z are the correct values.

d. **Gesture Recognition** (Jenny)
This module takes in clock_65mhz and the parsed 16-bit x,y,z registers as inputs and cross correlates this incoming data with registers containing hard-coded x, y, and z values for each gesture in order to determine what action was performed. It outputs the parameter value corresponding to that action at the end of a 16-bit action_queue (the output) of all the actions that need to be processed. Testing this would involve performing a gesture, outputting the calculated action on the Nexsys 7-segment display, and comparing this to the expected action.

e. **Action FSM** (Jenny)
This module acts as a finite state machine in order to determine which action(s) in the queue to actually perform. It takes the first element (parameter value) in the action_queue output by the Gesture Recognition module and transitions to the corresponding state to output two pieces of information: a 4-bit state that represents the sequence of assert signals controlling each action module and the actual parameter value itself. Testing this would involve going through all the possible combinations of the queue and comparing it to the expected sequence of assert signals in the output state.

f. **Music Stream** (Shana)
We want our system to be sending song bits at a faster rate than they are output by the speaker to ensure no lagging audio, therefore this module will output a song bit on every rising edge of clock_65mhz to be read by the main module. This module takes a one bit song_bit input which pulls a bit of song data directly from the Nexsys 4 Development Board SD card memory. Testing of this module uses a test bench to visualize the song bits being sent and comparing that to the song bit information stored in SD memory.

g. **Main Module** (Jenny + Shana)
This module controls actual music playback and holds instantiations of all other modules. Gets inputs from clock_65mhz, song bit stream, vsync, hsync, vcount, and hcount for monitor display. The main module will output VGA_R, VGA_G, VGA_B, and LED[15:0]. This module will be tested using the logic analyzer displaying all outputs.

h. **Volume Control** (Shana)
This action module will have a gesture parameter value of 4'd0 or 4d'1 (corresponding to raising or lowering the volume) and operate on clock_65mhz. The module will run when the [3:0] value outputted from the Gesture Recognition module to this module is 4'd0 or 4'd1. It will output an 8-bit register containing the new volume level. To test this module, we will create a test bench to make sure the volume level outputted is increased or decreased by the appropriate amount depending on the gesture inputted.

i. **Play/Pause** (Shana)
The Play/Pause action module controls playing or pausing song depending on the current play/paused state which will be stored in a 1-bit output register. It will flip the bit of this register when [3:0] value outputted from the Gesture Recognition module to this module is 4'd2, corresponding to a play/pause gesture and will operate on clock_65mhz. Testing this module involves using a test bench to visualize the current play/paused state when a play/pause gesture is inputted.

j. **Song Scroll** (Shana)
This action module allows user to scroll through songs and select a new one. If the 4-bit gesture input corresponds to a value of 4'd5 (scroll right) or 4'd6 (scroll left), then using the Song Lookup Table input, the module will output a 15-bit register that acts as a pointer to the section in the SD card storing the next/previous song (depending on what direction the user scrolled). The module operates on clock_65mhz. To test this module, we will create a test bench that will make sure the output to a scroll up or scroll down gesture is the expected song in the list of songs in memory as well as test cases where there are no previous/next songs.

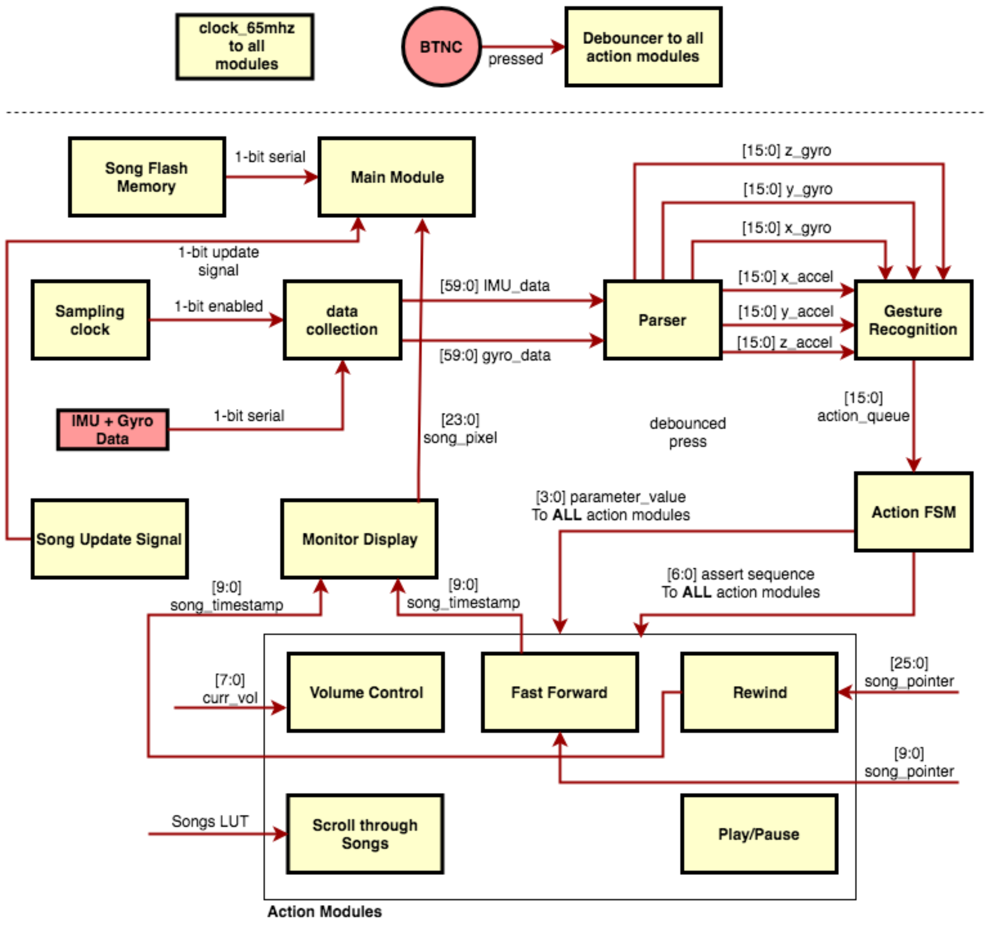
l. **Song Fast Forward/Rewind** (Shana)
These action modules allows users to fast forward or rewind the current song. If the 4-bit gesture input corresponds to a value of 4'd3 (fast forward 10 seconds) or 4'd4 (rewind 10 seconds), then the module will shift two 25-bit pointer

registers to the correct data bit in the progression of song bits to rewind/fast forward a song. The pointer registers will be updated every clock_65mhz clock cycle during normal playing. One pointer register can be shifted back (rewind) or forward to the new starting point and another (faster) pointer register will fill in the song bits that come after to ensure correct song sequence. This module will output a 10-bit register holding the new time that the song is at. Testing this module includes viewing module behavior on an oscilloscope in addition to manual testing with song data.

m. **One Second Signal** (Shana)
This module will output an enabled signal every second to control display monitor update rate. An internal [25:0] counter register that counts to 65,000,000 and asserts when counter = 65 million to simulate a one second clock. Testing for this module includes a test bench ensuring the clock is only enabled every second.

n. **Monitor Display** (Shana)
The monitor display module takes in clock_65mhz, an 10-bit input holding the time that the song is at, an 15-bit pointer to the memory location containing information about the current song (such as title, artist and length). It displays this information as a 24-bit RGB value rgb to the monitor as well as outputs the 1-bit vsync and hsync signals, and the 10-bit vcount and 11-bit hcount registers.

4. Block Diagram Visual



5. Reach Goals

a. **<u>Bass and Treble tuning</u>**
Allow users to tune bass and treble frequencies to their liking. To achieve this goal, we will first need to analyze frequencies as high/low and scale the frequencies up or down respectively. A combination of base boost low pass filtering and gain scaling will be used to achieve frequency tuning.

b. **<u>Stereo L/R pairing with lab speakers</u>**
Implement stereo speaker capabilities with two lab speakers. This goal will simulate a surround sound or multi-directional listening experience in addition to gesture controls. A basic form of this goal is to stream the song bits to both speakers rather than one, and handle streaming lag between either speakers empirically. More advanced versions of this goal will be discussed in the final report.

6. Possible Obstacles/Issues

Timing challenges when a gesture is popped from the action_queue to be interpreted and any problems that may arise with multiple gestures being interpreted before a physical behavior is exhibited.

7. Gesture Breakdown

a. **Volume control (parameter values: 4'd0 & 4'd1, assert sequence: 5'b00_001)**:
  - Hand starts in **default position** (palm towards ground, fingers straight)
  - Press BTNC for system to start listening for gesture
  - Rotate hand 180 degrees (palm now facing towards ceiling)
  - Raise hand slowly upwards for volume up, keeping palm facing towards ceiling and as level as possible (gesture **4'd0**)
  - Lower hand slowly from **default position** for volume down. (gesture **4'd1**)
  - The moment action is done (hand still raised/lowered), let go of BTNC.

b. **Play/Pause (parameter value: 4'd2, assert sequence: 5'b00_010)***:
  - Hand starts in **default position**
  - Press BTNC for system to start listening for gesture
  - Rotate hand so that palm faces forwards in the direction of your sight (high-five formation)
  - Move hand forwards in a pushing motion.
  - Moment action is done, let go of BTNC

c. **Fast Forward 10 sec (parameter value: 4'd3, assert sequence: 5'b00_100)**:
  - Hand starts in **default position**
  - Press BTNC for system to start listening for gesture
  - Rotate hand so palm faces to the right with fingers straight out
  - Move hand rightwards in a sweeping motion
  - Moment action is done, let go of BTNC

d. **Rewind 10 sec (parameter value: 4’d4, assert sequence: 5’b01_000)** :
   - Hand starts in **default position**
   - Press BTNC for system to start listening for gesture
   - Rotate hand so palm faces to the left
   - Move hand leftwards in a sweeping motion
   - Moment action is done, let go of BTNC

e. **Scroll through songs (parameter values: 4’d5 & 4d’6, assert sequence: 5’b10_000)**:
   - Hand starts in **default position**
   - Press BTNC for system to start listening for gesture
   - Rotate hand so palm is facing leftwards & rotate around the wrist in a rightward sweeping motion for scroll previous (gesture **4’d5**)
   - Rotate hand so palm is facing rightwards & rotate around the wrist in a leftward sweeping motion for scroll next (gesture **4’d6**)
   - Moment action is done, let go of BTNC

*****Play/pause action will also control playing a selected song based on a currently_playing flag to keep track of whether a song is being played.