MAMBO

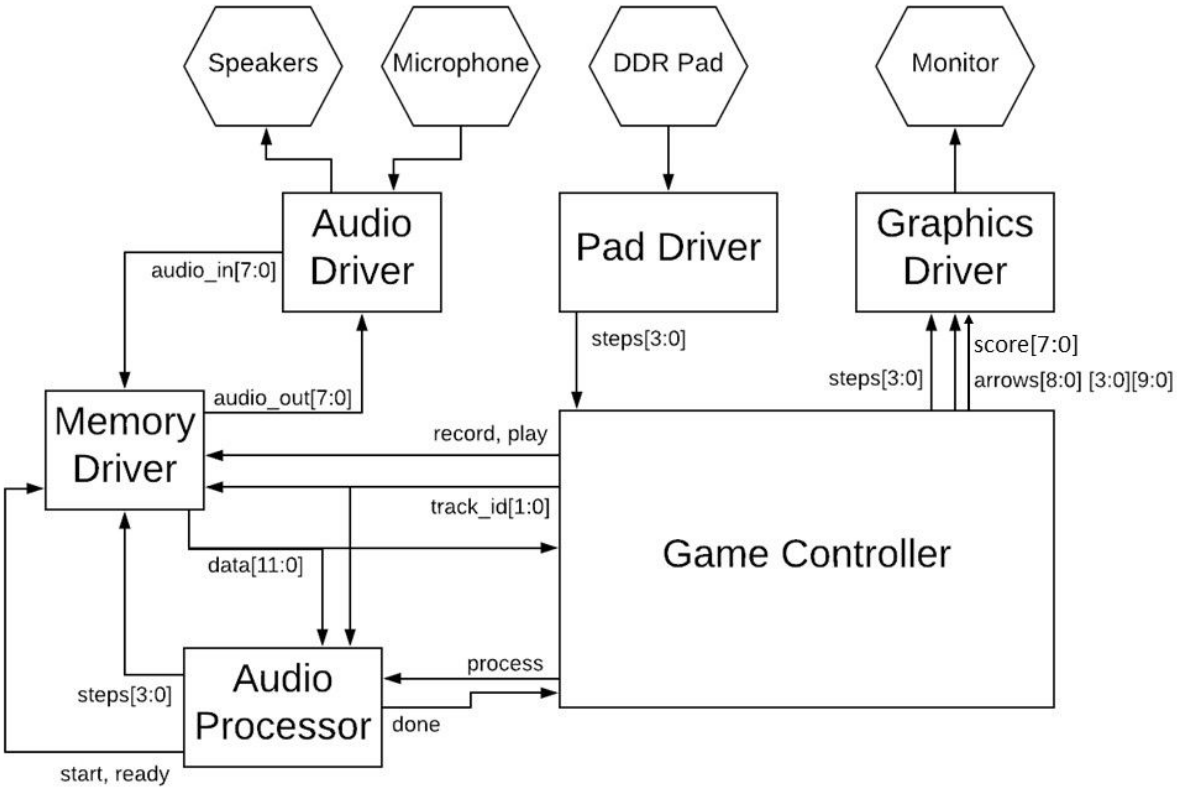
Jamie Bloxham
Matthew Tung

Overview

We aim to create a system that will allow users to play a Dance Dance Revolution-style game. As in DDR, players will interact with a system via a dance mat with four pressure-sensitive arrows on it. On a monitor (connected to the FPGA via VGA, perhaps), players will be indicated of a sequence of arrows; their job is to stomp on the arrows on the pad in a corresponding pattern. We will attempt to create such a pad ourselves, and suspect this will not be as difficult as it sounds. Finally, as a stretch goal, we hope to implement automatic chart generation (where a "chart" is the sequence of arrows that players must match). Specifically, we will allow some sort of interface via which players can provide their own songs (perhaps via a USB thumb drive), which are then read by the FPGA. It will then apply several signal processing techniques on the audio file (e.g. beat detection and instrumentation analysis), to generate a sequence of arrows that adequately complements the music.

Module Description

A block diagram of the system can be found below. We will be using the labkit.



Game Controller

Responsible for keeping track of current game state, and handling associated logic. The FSM has 4 “core” states: idle, playing, recording, and processing.

When idle, the system is not doing anything besides waiting for player input. In this state, using buttons on the labkit, the player can select a “track”, numbered from 0 to 3. The player can either “play” this track, or “record” a new track under this ID, resulting in a switch to the corresponding mode, and the corresponding output is also sent to the memory driver.

When playing, the controller receives a steady stream of 6000 Hz 12-bit data from the memory driver, encoding both the audio data (which isn’t used by this module), and the arrow pattern. Specifically, 4 bits are used for this purpose, one for each arrow. A high bit indicates that the corresponding arrow appears in the next 2.5s. This “preamble” is necessary so that arrows can be displayed before the player will need to step on them. The game controller is responsible for interpreting this data to keep track of the position of each arrow. While each arrow is associated with a single instant in time when the player should step on it, the controller allows a 100ms window surrounding during which the arrow can be hit. Depending on the time, we will create thresholds in which, the closer the matching to the precise time, the higher the score given. The controller is thus also responsible for interpreting information from the pad driver to determine whether a hit is valid, and to update an internal score accordingly, which will be displayed on the labkit’s hex display. In this state, the controller is also responsible for sending information to the graphics driver. This consists of two parts, a 4-bit value “steps”, which indicates which arrows the player has stepped on, and “arrows”, which indicates the coming arrow pattern. Specifically, “arrows” is a 2-dimensional array of 8-bit values, each of which represent the y-coordinate of an arrow as it falls down. This array is 4x10, with the 4 being for each direction, and the 10 being the size of the buffer per direction. In other words, we can display up to 10 arrows at once, for each direction. Once the song is over, the controller returns to idle mode. Any stretch goals for this game controller would be more a comprehensive score calculation (streaks, power moves, etc.) and varying difficulties or tracks.

When recording, the controller waits for a repeated “record” input, which indicates that recording is over, at which point the controller enters “process” mode, and sends a corresponding input to the audio processor.

When processing, the controller waits for a “done” input from the audio processor, and then enters “idle” mode.

Audio Processor

The audio processor is responsible for interpreting raw audio data from the audio driver, and producing an appropriate arrow pattern from that data, on receiving a “process” input from the controller. We anticipate that this will be a somewhat slow procedure, and thus this module will

not attempt to keep up to speed with the system clock. We also expect that this module will not be able to hold entire songs in its own memory at a time. Thus, this module will hold only a window of the audio data at a time, process it, then request more data from the memory module using the “ready” output.

To actually process the data, we need to implement some sort of onset detection algorithm. If we are able to do FFT, we can calculate the energy as the square of the FFT, and look for peaks in the “derivative” of this graph, outputting a random arrow where we find a peak. However, simpler algorithms may achieve acceptable performance, and we anticipate needing to experiment with a few different ones such a low pass filter and just variation checking The end goal is to break down a song into a set of comprehensive yet concise beats to then convert into moves.

Memory Driver

The memory driver is responsible for interfacing with both flash memory and ROM (not shown in the block diagram). Flash memory will be used for record functionality, and ROM will be used so that the player does not need to record a song in order to use the system (which will be useful for testing). The memory driver selects an appropriate storage medium for its tasks based off of the track id, with track id 0 indicating flash, and otherwise ROM. This means that songs can only be recorded into track 0. In a stretch goal we would access external storage and add the feature of having multiple tracks, but for the MVP, we would use the built in flash. Essentially, we need to the memory to store the music itself for playback as well as the corresponding step patterns from the audio processor so it can then be sent to the main game module.

Audio Driver

This module is necessary for producing an audio signal from a microphone, and playing an audio signal on external speakers. This will likely not be doing anything more complicated than what was seen in Lab 5.

Pad Driver

This module will interpret and debounce the input from the physical DDR-style pad that we will build. It should essentially be 4 switches that will determine if it is on (being stepped on) or not and send it to the game module in order to check if they player did the correct move at the correct time.

Graphics Driver

This module is responsible for interpreting signals from the game controller (signal format detailed in that section), to produce valid VGA output, as we will mainly be displaying on the computer screen for our project. It should show the current score, the ‘scoring’ region with outlines of the 4 arrows where we want to there to be a match, and the upcoming dance moves themselves as arrow patterns as they drop down the screen at a consistent speed. If the moves are executed correctly, the falling arrows should disappear within the region, and if not fall off

the bottom of the screen. Stretch goal would be to add possible background images, streaks, and animations for failing or matching. As well as a start and end screen.

Hardware

We will be building a physical DDR-style pad, consisting essentially of 4 pressure plates. This will basically act as 4 switches to work with the pad driver. We will definitely need to experiment with both our design as well as the way it interfaces with the labkit, but we essentially plan to use metal sheets or plating (possibly of copper or aluminum), wire, and a frame material of some sort, whether it be cardboard or wood or whatnot. The design will be wire being connected to 2 sheets of the metal that are not touching each other when idle (some sort of intermediate layer) but are close enough and facing each other in a way that when enough pressure is applied on top, the plates will bend and the metal sheets will make sufficient contact, completing/connecting the circuit which will, depending on the design, set the state from HIGH to LOW or LOW to HIGH. Once this has been achieved, we will most likely make a sleeve to hold the plates/switches in place for maximum dance ability.

As of the now, the only other hardware we will need will be the audio in and out, in which we will send the audio data into the labkit from a device (laptop, phone, etc.) via the microphone jack, most likely in the form of a .wav file. This will be the data that's sent to the audio driver for the memory. We will then want to play the song back during game mode which will most likely also be from the audio driver and be sent out the headphone jack to a speaker or, if playing quietly, earphones.

Limitations

Some potential performance limitations that I can see happening with this project is the delay due to the audio process. We definitely do not plan on doing real time audio processing to play mode as we expect the audio processing to take a decent amount of time. We will have to make sure our pre-processing is efficient and works with the clock restrictions of the labkit. In lieu of that, we must also be smart with our memory whereas we only have 128 Megabits of flash memory to use, so in order to be able to hold both the audio itself and the corresponding moves, we will have to sample and store efficiently. The game itself will also be fairly speed intensive, as it will need to do a lot of calculation and communication with the memory and screen module, so it was important that we made separate, but we still need to make sure all of it is feasible within the appropriate clock cycle as we want to make sure it's all synced, especially since we want real-time audio playing with the game component. Like with any project, speed and efficiently is something we must take heed to.

As for hardware, we will first have to make sure our design works and actually behaves like a switch, then we will have to make sure there is not significant delay in order to make sure it actually works and is playable in real time. The switches' delay and effective may also be different, causing there to be performance problems and frustrations. We need to make sure our

switches are quick and responsive, or at the very least, consistent with each other so that we can properly offset. Since the functionality of the audio hardware should be similar to Lab 5, we believe our main concern will be making sure are pressure plate work properly.