F.R.E.D.D.Y

Jamie Bloxham
Matthew Tung

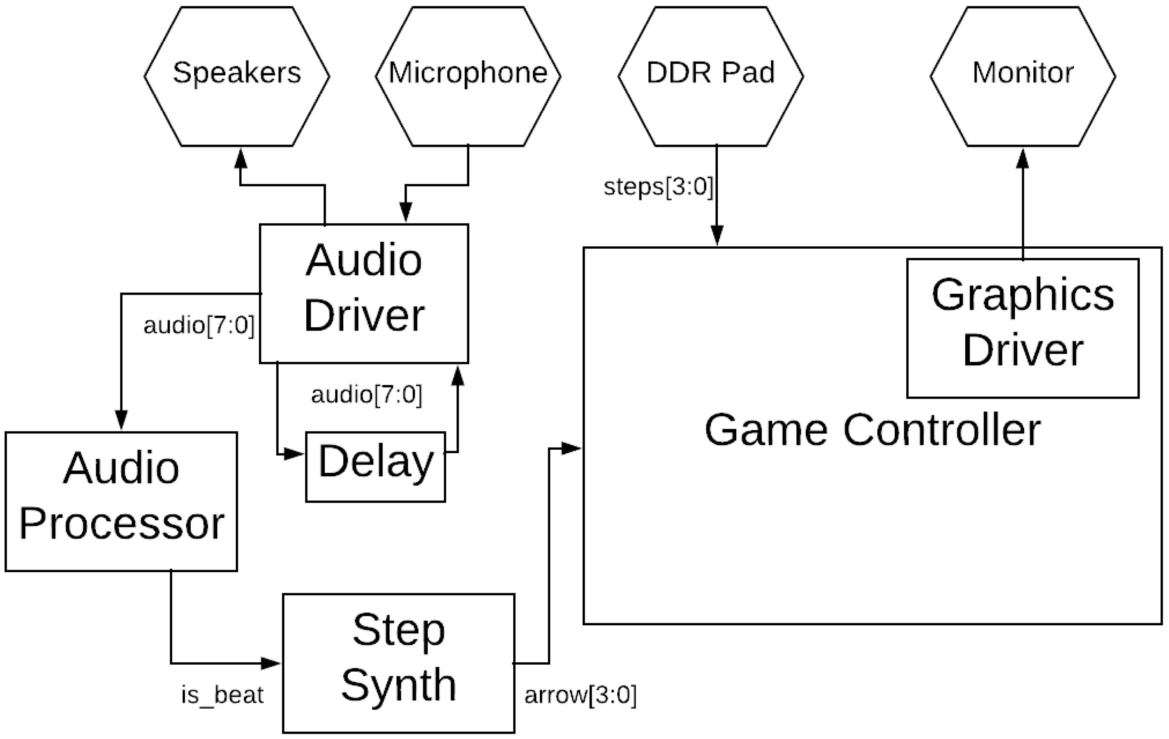
Overview

We created 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), players are indicated of a sequence of arrows; and their job is to stomp on the arrows on the pad in a corresponding pattern. Using a cardboard, wire, aluminum foil, copper tape, and resistor, we created step pads that the player would use. We also implemented a way to increment and keep score, as well as streaks and multipliers and displayed it on the labkit's hex display. On the other side, we wanted to be able process audio and use it to produce the sequence of arrows sent to the game side. We essentially received the audio through the microphone jack of the labkit, put it through a low pass filter, then detected 'energy peaks' by comparing different windows of the audio in order to get the strong beats of the song to make moves out of. We also implemented a 'difficulty' setting where can flip a switch to decrease the energy threshold and detect more beats, and as a result, moves. The audio is also put through a delay via memory in order to synchronize the move when it reached the scoring region. This audio is output via the headphone jack.

Our motivation for creating our project was to develop something fun with the labkit and our knowledge from 6.111. We saw what the labkit could do from labs 3 and 5 and we wanted to expound on it in a creative way. We thought that DDR would be an attainable project while also having the potential to provide many challenges and push ourselves.We were able to make a fun project that is both interesting, dynamic, and can build up a sweat. I think we were able to achieve quite a bit with our project, and there is still plenty to be done. We are both fans of the beat and rhythm games such as real DDR and Rock Band and wanted to see if could create a version using an FPGA.

Module Description

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



By Module

The overarching module is **lab3.** Aforementioned, a lot of capabilities of the labkit and its ability to work with the VGA was demonstrated in lab 3, so we decided to use it as a baseline for the labkit's basic setup.

Game Controller

Inputs: clock, [10:0] hcount, [9:0] vcount, hsync, vsync, blank, [3:0] steps, [3:0] pads, steps_clock, steps_ready, new_song
Outputs: phsync, pvsync, pblank, [15:0] score, [2:0] test_multiplier, [23:0] pixel

As the meat and potatoes of the game logic, this logic takes input from the audio processing component and the hardware pad. When the steps input asserts a new value/arrow for the game, which gets detected using the 27 Mhz steps_clock, it gets written to next available "arrow_blob" module (to be explained later) via the use of multiple registers and buffers. This newly assigned arrow_blob then travels down the VGA at 8 pixels a vsync cycle. Then if the pads input matches the arrow pattern when it is in the vicinity of the scoring region at the bottom of the screen, as displayed by another "blob" module, it is 'hit', causing it to disappear and increment the streak as well as the pixel's score based on it's pixel-based accuracy and the multiplier, which is determined by the current streak. Otherwise, if the arrows is missed as it passes the scoring region to the bottom of the screen or the pads get prematurely hit, the streak resets and score is not improved. In order to have enough arrows available for incoming step inputs, 7 of the arrow_blobs are generated in addition to the scoring region blob. All of the pixels are OR'ed with each for the ultimate displayed pixel for the VGA

Arrow_blob (Game controller submodule)

This is a scanning pixel region that shows arrows, or nothing based on the input, as defined by the 4 bit input for steps, then if the corresponding bit is asserted, it assigns the arrow's pixels to the defined color using geometry and algebra. Two triangles are made to act as the arrow's whereas the y pixel is a function of the x pixel, and then a small rectangle is created to be the head. The region is 192 pixel high and 1028 pixels long, so it has the capability to display all for arrows as it travels down the screen based on the y input (the 8 pixel travel variable defined in the above module).

blob (Game controller submodule)

This is the scoring region at the bottom of the screen where the player should match the arrows. It is essentially 2 white bars 192 pixels apart.

Graphic Driver (Game controller submodule)

This is using the xvga components and modules given in lab 3 to display all of the aforementioned modules onto the VGA screen, using all of the hsync, vsync and other various variables and components to properly display on the VGA

Meanwhile the pixels' score are added up to an overall score, which, in addition to the streak is displayed on the labkit's hex display using several modules.

bcd_counter (Game controller submodule)

A counter that counts to ten by a clock signal.

four_digit_bcd_counter (Game controller submodule)

Makes 4 bcd counters for the score value to be displayed on the hex.

ascii2dots (Game controller submodule)

Converts the input from the bcd counter as well as the display modules to ascii characters.

display_string (Game controller submodule)

Creates a 500kHz clock and then takes the ascii data from the above module and displays in on the hex display in the format: Score: [0000] Streak (1x)

Audio Processor

Inputs: clock, reset, ready, [7:0] audio_input, hard
Outputs: phsync, pvsync, pblank, [15:0] score, [2:0] test_multiplier, [23:0] pixel

The audio processor is responsible for interpreting raw audio data from the audio driver, and producing an appropriate arrow pattern from that data. First thing it does is put the raw audio through a low pass filter module identical to that used in lab 5. This then goes through an energy window comparator to then find peaks. If these peaks are strong, they are labeled as beats and step values are created for the game component. In addition, the audio is delayed by 1.6 seconds in order to match when its corresponding arrow hits the scoring region by storing it in RAM similar to lab 5 and then playing it through a register instead the straight connection.

coeffs31 (Audio processor submodule)

Similar to lab 5, except scaled by 2**10 now that we're a 48kHz sample rate, this module contains a bunch of coefficients for the low pass filter.

fir31 (Audio processor submodule)

The low pass filter module that takes the 8 bit raw audio data, scales it by the coefficients defined in the coeffs31 module for that young 24 bit new low passed audio.

enf (Audio processor submodule)

Takes the first 8 bits of the filtered audio and then creates energy windows by taking the square of the input and adding it to two different 32 bit accumulator values. These values are offset and then subtracted from each other in order to represent the change in the energy or volume of the audio , creating a peak value which is then sent to the other beat module.

peaks (Audio processor submodule)

This module takes the difference values of the peaks and if it's over a certain threshold as defined by the hard input from the overarching module, it will label it as a beat and asserts that a step command be made for it by sending the last 2 bits of the audio input to the step_synth module.

Step_synth (Audio processor submodule)

When those slick 2 bits are received and labeled as a beat, it converts it into a 4 bit step value for the game by left-shifting 4'b1 by the decimal value of the 2 bits. If not labeled as a beat, it just asserts a 0 value arrow/step.

Audio Driver

This module what takes the audio from the microphone jack of the labkit as well as output the audio through the headphone jack. It is pretty much identical to what implemented in lab 5.

ac97 (Audio driver submodule)

Assembles or disassembles the input audio date to AC97 serial frames for the labkit's audio capabilities (identical to lab 5).

ac97commands (Audio driver submodule)

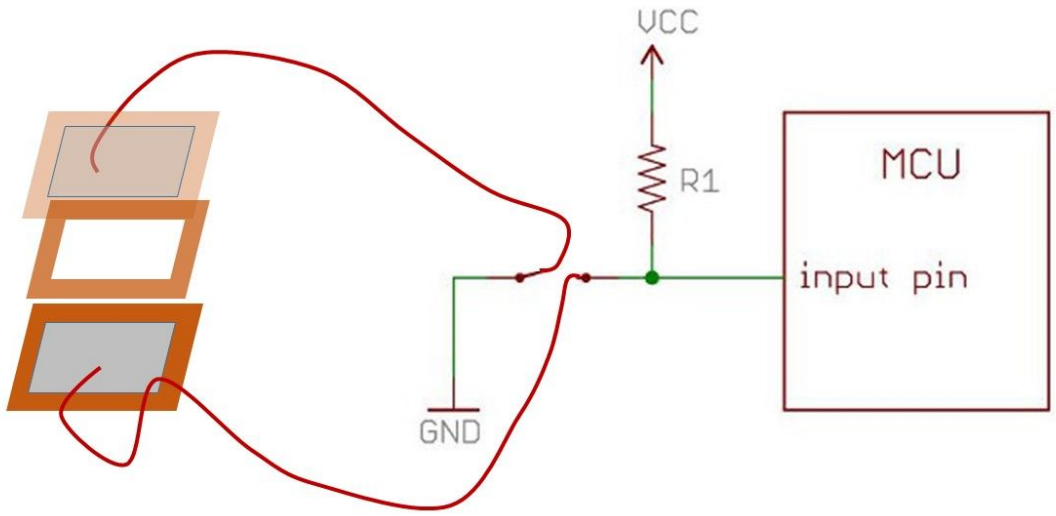
Takes volume, address, source data and sends to the labkit's AC97 for initialization (identical to lab 5).

lab5audio (Audio driver submodule)

Takes clock, reset, a 4-bit value for volume, and 8-bit audio data and sends it to the above modules for that young sound action (identical to lab 5)

Hardware

In addition to speakers for the audio output and aux for the audio input, our main hardware component was the 4 step pads. Made of cardboard, aluminum foil, copper tape, and wire, they act as pressure pads which activate when stepped. We combine these switches with resistors in order to pull-up resistor switches. Basically, when stepped on, the metal pads connect and ground the circuit. We can then invert this signal and make it so it works with the steps and game logic.



Limitations

While we were able to achieve most of our goals and have a fairly well-functioning project by the end, we definitely hit many road bumps along the way. When it came to the hardware, we had to go through several iterations of design before finally being able to make pads that would properly decompress. We were able to end up just using cardboard as the frame, but it had to be heavily supported. The pull-up resistor was also necessary as there was definitely too much noise with just an antenna connection. Also, our original intent was to store the audio in some sort of memory, such as flash, but that proved to be too inconsistent and difficult to implement, so we decided to do all of the audio processing in real-time, and it possibly would have been difficult based on the other problems we encountered. With the preamble, the multiple modules present on the VGA as well as all of the other modules, it seems as if we hit the the hardware limitations of the labkit. Our original plans was to implement COEs to display score, streaks, multipliers, more animated arrows, etc. on the VGA, but whenever we implemented them, they would never process correctly. Even without extra images and using pixel geometry over COE, our project would take over 25 minutes to fully process and would still sometimes glitch (the arrows would become warped and uncontrolled). While there are most likely ways to optimize, our project definitely pushed the hardware capabilities of the labkit, which was definite challenge for us in both what we could add and long wait time in between.