Portable DDR

Gabriel Ha ('11)

Daniel Kim ('11)

6.111 Introductory Digital Systems Lab Final Project

Foreword

To be completely upfront about the nature of this report in lieu of our hopes for the final project, we were not able to completely implement the project to our satisfaction. It follows then that much of the contents of this report fall under the category of “theory.” However, we firmly stand by our convictions that our not being able to complete the project was a matter of figuring out how to interact with the labkit system (especially issues such as memory), as we devoted significant time to the project in thought, design, and working inside and outside lab to implement everything. Though we may not have been able to finish the project ourselves, we believe that this report will satisfy its purpose: “to describe the work you did in enough detail so that a reader can understand how your implementation works and enable them to replicate it themselves if they wanted to.”

Table of Contents

Introduction	**4**
About DDR	**5**
Appeal	**6**
Overview and Block Diagram	**7**
Modules	**9**
Player Motion Processing	9
mp3 Encoder	11
Music to Arrows Algorithm	12
Game Logic and Synchronizer	13
Play Module	14
Scoring	15
Video	15
Final Thoughts	17
Acknowledgements	18

Appendix uploaded to website

Introduction

This project is based off of Konami's popular video game Dance Dance Revolution. We address two limitations of the original game: the fact that the game requires a game pad, and the limited song selection to which DDR players can play. We use one or more cameras to track the player's movements as a substitute for the game pad. To address the issue of song choice variety, we implement a music-to-arrows (m2a) algorithm that analyzes the music waveform that is produced via the 6.111 labkit's ac97 module which can decode and playback an mp3.

For gameplay purposes, we attempt to emulate a significant portion of the original DDR's functionality as possible, including arrow scrolling, timing accuracy, and scoring.

About DDR

DDR is a musical video game created by Konami and released about ten years ago in 1999. The game is marketed in both video arcades and home systems. The objective of the game is to physically step in real-time to a preset series of arrows that generally corresponds in rhythm to a song. The arrows scroll from the bottom of the screen to the top at a constant rate, though the default rate for each song may be faster or slower depending on the tempo of the piece. A screenshot of the gameplay is seen below:

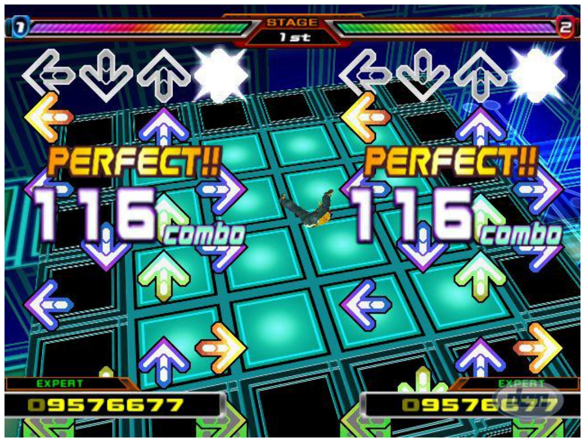

Figure 1 – A screenshot of DDR gameplay

This particular screenshot is that of two players playing the game who are scored individually, though it is possible to play solo as well. For our project, we emulate the one-person mode only, featuring a single set of arrows onscreen. When the arrows reach the top of the screen, the players must step in the corresponding direction on the game pad, seen here:

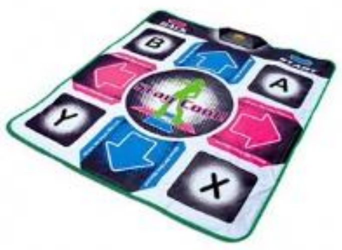

Figure 2 – An example of a DDR gamepad

Appeal

There are two limitations of DDR that are recognizable to anyone who is familiar with the game. First of all, the game requires a game pad. If the player is playing in a video arcade, there is a metal game pad that is provided for them, as seen below:

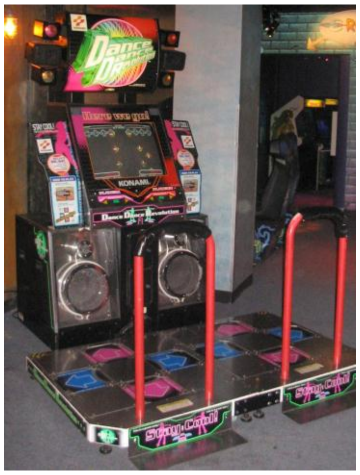

Figure 3 – Game pad seen in arcade

However, if a player wishes to play at home, he must purchase a game pad, which are generally very flimsy because they are designed to be folded up to preserve storage space. What usually ends up happening is that the game pad generally slides and shifts around on the floor as the player plays, especially as gameplay gets more intense, and this has the unfortunate effect of disrupting the player's gameplay. Some people even resort to duct-taping their pads on the floor to prevent it from shifting around, and as one can imagine, this turns out to be rather messy in terms of both setup and cleanup. Replacing the pad with cameras tracking is one solution to this problem, and we explore this possibility in this project.

The second limitation of DDR is the song choice selection. Though each DDR game comes with about 40-50 songs, that is all a player gets. What if he wanted to dance to other music, or possibly his own music? To compensate for this issue, we process a song through a music-to-arrows algorithm that generates a series of arrows based on the rhythm of the music.

Overview and Block Diagram

The project is comprised of three major modules: the module that processes the player's movements from the *camera (not implemented due to time constraints, but our ideas on it will still be discussed)*, the module that converts any music into arrows and stores them into memory, and the module that synchronizes these signals and performs the game logic. A basic block diagram is seen below:

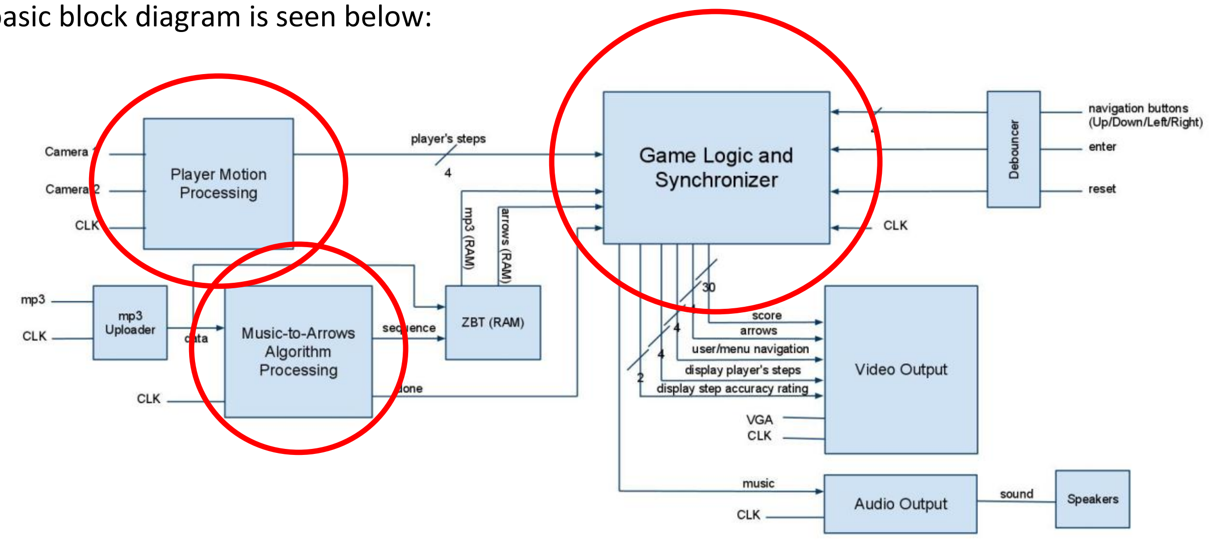

Figure 4 – Block diagram with the three main modules circled

A player would go through the following steps to setup for gameplay:

1) **Calibrate the cameras** – Depending on the players physical position, the pixels that he ends up comprising in the camera(s)' field of view will change, and must be set accordingly. As mentioned above, we did not have time to fully implement this module, but our ideas conceived in pre-production will still be discussed.
2) **Upload the music** – Music is uploaded via a male-to-male cable that is connected into the microphone jack of the labkit from any music source, generally some sort of mp3 player like an iPod. In our design, the player depresses the enter button of the labkit to record and write whatever is going into the microphone jack to memory. It stops when the enter button is released. Music can be re-recorded in this manner to the satisfaction of the player. When the player is satisfied, he continues to the next step.
3) **Select difficulty** – The music to arrows algorithm was originally designed to produce two sets of arrows: a normal set, and a harder set. Though the harder set was not implemented, it follows along a very similar concept to that of how the normal arrows are generated. A difficulty level is selected by the player and he proceeds from that point on to gameplay.

4) **Gameplay** – The player plays the game. In the original DDR game, if you did poorly enough the game would stop and it would be game over, but we just allow the player to continue playing until the music is done.
5) **End game** – At the end of the game, when the music is done, the player has the option of playing the same song again or uploading a new one.

Other modules in the system include the mp3 uploader, which was reused (but still significantly modified) from this year's lab 4A, the module to display the video, and, of course, the memory included in the labkit (ZBT RAM) (not a module in of itself, but a very important component of the system nonetheless). The ZBT stores both the music waveform as well as the arrows that go along with the music, and is retrieved by the game logic module during gameplay.

Modules

The modules are listed here in the approximate order that the player accesses them (whether directly or indirectly).

Player Motion Processing (Verilog: not implemented) (Gabriel Ha)

As noted above, due to time constraints, this module was not able to be implemented. However, we will still share our ideas on what we believe would have worked.

The most important thing in terms of facilitating this process would have been to reserve about a week devoted to figuring out how the provided Verilog modules from years past decode the output of the NTSC cameras used in the lab, and how to use the image data. There is not a lot more insight that we can provide on that, other than that the provided code only generates black and white images to start with, so figuring out how to get color is a significant part of that task. We recommend using brightly colored tape or paper to facilitate the camera's ability to track movement.

Once the camera signal has been converted into some useable form, there are two ways to approach the tracking issue, depending on the number of cameras you wish to use.

One Camera method:

This method utilizes an overhead camera, which can be set up in the lab room simply by stacking two columns of some sort of lab furniture (like cabinets), placing a long and wide wooden board to bridge the columns, and tape the NTSC camera to the board's underside, looking down at the player. The brightly colored tape would be placed on top of a cap that player would wear.

During calibration, the player starts out standing in what would be the DDR pad's neutral position (the center square of the game pad, see **Fig. 2** – it is the area in the middle with the outline of a green girl, because obviously that makes so much sense). The module should ideally run autonomously during the calibration process based on a sequential FSM (i.e. a deterministically sequenced Finite State Machine that progresses only forward and never returns to a previous state unless the process is restarted), and the player should be aided by the instructions on the monitor. Even nicer would be to have the image from the camera looking at the player onscreen during calibration, but this is not necessary. Upon starting, the camera allows the player five seconds to position himself in the neutral position (directly below the camera) and then records the current center of the colored item using a center of mass algorithm. The player is then instructed to take a step to the left. After another five seconds, the camera takes the new center of mass and sets the “left arrow” boundary threshold as the average of the x-coordinates of the neutral position and the new position. Now, whenever the object’s center of mass moves past that threshold, it is interpreted as a left arrow. The same procedure follows for taking a step forward (uses a y-coordinate threshold), to the right (x-coordinate), and backward (y-coordinate). This creates four points to create a rectangle to delineate the neutral position. If the player should take a step diagonally, for example, this should not confuse the code, as it should interpret the movement as whichever arrow direction was programmed first in the if-else if conditions.

Two Camera method:

This method utilizes two cameras placed diagonally in front of the player and on the ground, shown in the following fashion:

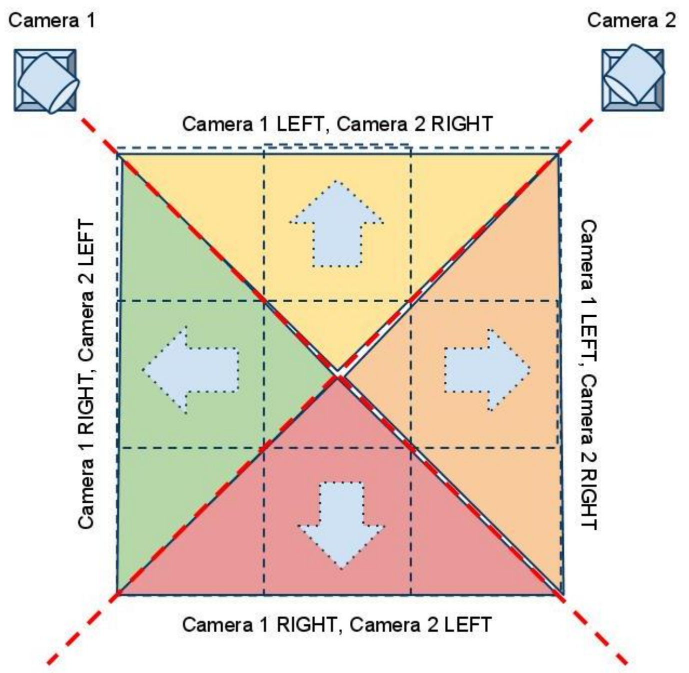
Camera 1
Camera 2
Camera 1 LEFT, Camera 2 RIGHT
Camera 1 RIGHT, Camera 2 LEFT
Camera 1 LEFT, Camera 2 RIGHT
Camera 1 RIGHT, Camera 2 LEFT

Figure 5 – Two camera design

As one can see, there is a unique combination of left and right from the two cameras that represent each arrow direction. Bear in mind that there is no actual pad, so these colors and arrows are simply virtual representations. The calibration process is similar, but both cameras need to take measurements now. Since there are two measurements, for example, for Camera 1 in determining what is to the right of the camera (the left and down arrows) the smaller of the two distances should be taken as the minimum threshold to register a step.

Due to the 2-dimensionalness of the images, neither camera algorithm should be able to detect the difference between a person actually stepping and simply waving their foot (or shifting their head) to the respective direction. This is okay for the purposes of this project; more advanced algorithms can be utilized in a future version that involves more than just distance thresholds.

mp3 Encoder (Verilog: mp3_encoder_v2.v) (Dan Kim)

The mp3 encoder is intended to take in an mp3 file from the microphone input as audio. This encoder is analogous to the recorder created in lab 4A, except that changes have been made in the recorder module. The filter has been decoupled, and the end address is propagated out of the module, as it is needed in other modules to determine when to end play mode. Additionally, the module has been set up to utilize the ZBT instead of the BRAM, which was originally the RAM destination of lab 4A.

The ac97 samples music and creates a waveform that is composed of 48,000 frames per second. We chose to sample every 8th frame instead (resulting in 6000 frames per second) and compensate for the playback accordingly. Each frame is stored sequentially in memory starting at the ZBT's "0" address. The ZBT is 512,000-bits long by X 36-bits wide, so there's a significant amount of music that can be stored, though we do not use all 36 bits at each address. A music waveform is akin to a sine wave, so one can imagine simply sampling the amplitude at a certain time resolution and storing that value. The waveform generated by the ac97 is composed of 8 bits, so this is stored in bits 7 down to 0 in the ZBT. The 8 bits are signed, meaning that a frame holds a value inclusively between -128 and 127.

The module is programmed to begin recording when the enter button of the labkit is depressed, and to stop recording when released. In this phase, the music will continue to loop around until a signal is sent to the m2a module to begin processing. Additionally, it sends the m2a module the end address of the recording so the algorithm knows when to stop (to compensate for issues like when you record over previous music that was longer than your current sample)

m2a Algorithm (Verilog: m2a_v2.v) (Gabriel Ha)

The music to arrows algorithm appends arrows (in the form of bits, of course – it is stored as 6 bits: one bit for each arrow, and the next two bits for the arrow type, of which there are 4) to the points of music stored in the ZBT that it deems an arrow should be placed. These addresses have an additional 6 bits appended to the 8 bits of music already stored.

The algorithm makes the assumption that the music provided has very strong down beats, and is hence solely amplitude-based in its analysis. It is also a sequential FSM.

The algorithm looks at the first 5-10 seconds of the recording (since we record 6000 frames per second, this is equivalent to 30,000-60,000 frames; we look for 5 seconds in the algorithm) and stores the highest amplitude magnitude it finds (whether positive or negative, so it just looks at 7 bits and ignores the MSB). It then starts over at the beginning of the music and looks for the first 3 consecutive peaks that have this amplitude or something within 10 units (bits) of it. It assumes that these peaks are down beats of the song and takes the difference between the first and third peaks, right shifts the difference by 1 (equivalent of dividing it by 2), and this number is the tempo of the song, in units of frames per beat (fpb).

Note that each RAM address stores a frame of music, so we are subtracting the RAM addresses of the first and third peaks.

From this point, the algorithm begins assigning random arrows to the beats. To generate a random bit, we use the code

```
reg [18:0] lfsr=0;

always @ (posedge clock)
   lfsr <= {lfsr, ~lfsr[18]^lfsr[5]^lfsr[1]^lfsr[0]};
```

where the zeroeth bit (lfsr[0]) is the random bit. To generate two random bits (to map to one of four arrows), you can either use a second clock and a second variable, which would be the more accurate thing to do, or save the delayed value of the variable and concatenate them together, as we did.

For the normal mode, the algorithm assigns a random arrow to every downbeat of the song that has significant amplitude there. Significant is defined arbitrarily as ¼ of the maximum amplitude found (right-shifted by 2). The algorithm assigns a random arrow to the first peak it found and then jumps forward a number of RAM addresses equal to the fpb. This is because that is the next point where it is expecting a downbeat. If it doesn't see what it's looking for, it check four frames to the left and four frames to the right for significant amplitude. If none, it moves on, adding another *fpb* to the current RAM address.

All these values are stored to a massive 2-D array of registers to be written to the ZBT later. The array is XXXXX – by 55 bits wide, where XXXXX is some arbitrarily large number that is supposed to represent the number of arrows you expect a song to have (for most songs, somewhere in the range of 500-1000, for this project's demonstration purposes, less than 100). The first 19 bits stores the RAM address where the arrow will go, and the next 36 bits stores the data to be written. Since we only have 14-bits of data, the first 22 bits are assigned to 0, followed by the active arrows (4 bits) and then the two-bit code for an on-beat arrow. The last 8 bits are the original music data (essentially what was in the ZBT before)

When the algorithm has reached the end of the music address (supplied by the mp3 encoder) it then proceeds to write all the data into ZBT by parsing the massive register that the info was saved to.

The more difficult set of arrows is generated very similarly. We did not have time to implement this as we spent countless hours (and days) with no success in debugging memory writing issues from the module to the ZBT. However, the concept is as follows: the harder algorithm does the same thing but introduces the concept of off-beats and tuplets (another two arrow types). It similarly looks at the areas of the stored music waveform where it expects to put an off-beat or a tuplet, and if it sees a significant waveform, it assigns a random arrow there, and if not, it checks around and then passes on to the next arrow type. These arrows are stored in the next 6-bits in the ZBT, so the 55 bits stored are now RAM address (19 bits), 16 zeroes (16 bits), the hard arrows data (6 bits), and the remaining data, which consists of either the normal level arrows or just zeros (6 bits), and then the original music (8 bits).

Once the algorithm is done, it passes a done signal to the game logic module, as well as the frames per beat of the song.

Game Logic and Synchronizer (Verilog: game_control.v) (Dan Kim)

This module is the top level state machine for controlling the DDR game. This module steps through five different states.

The module is initialized in the first state, SETUP_CAMERA. When the module is in this state, it sends a signal to what would be the Player Motion Processing module to start calibrating the cameras (calibrate_camera). When the camera processing module is done, it sends a signal back to this module (calibrate_camera_done), which then shifts the module into the next state: setup of the mp3 file and encoding of the arrows (SETUP_MP3).

When the module is in this state, it sends a signal to the mp3 uploader to start uploading the file (load_mp3). However, the file does not actually start uploading until the enter button is pressed. After the music is uploaded, a signal is sent to the m2a algorithm to start processing the data file.

After the data is processed, a signal (load_mp3_done) is sent back to this module to let it know that music uploading and processing has been completed, as well as the frames per beat. This then moves the module to the next state, which is the game ready state (SETUP_GAME_READY). This allows the user to start the game whenever the user feels ready.

To start the game, the user turns on switch 2 to start game play. This then turns on the control bit "play" so that the play module can run and begin game play. The play module continues until the end game signal is sent back to this module (end_game).

When this signal is returned, the control bit "play" is set to zero and this module is moved into the next state, END. At this point the game is complete. At this point there are two options. The first option is to upload an entirely new song. This can be done by turning on switch 3. This takes you back to the SETUP_CAMERA state at the very beginning to recalibrate the cameras and upload a new song. The second option is to replay the song. This is done by turning on switch 4. This takes you back to SETUP_GAME_READY, which allows you to start the song again. Play_module and scoring are instantiated within this module, as they both directly affect game play.

Play Module (Verilog: play_module.v) (Dan Kim)

This module controls actual game play. In this module, a rotating BRAM is initialized to delay the music 5 beats behind the arrows (this is why it is important that the game logic module receives the frames per beat [fpb] of the song from the m2a module), since arrows take 5 beats to reach the top of the screen. The address of the ZBT and BRAM are initialized to zero. From there, at every ready cycle (the ready signal being pulled from the ac97 to ensure that the music remains properly clocked to the ac97 chip), music is first pulled from the BRAM and sent to the speakers. A cycle later (implemented by a delayed copy of the ready signal), a new music sample pulled from the ZBT is put into that address slot, and the arrows from that same ZBT address are sent directly to the video. As such, music remains delayed while arrows are sent to the video module right away. Currently the end_game signal is only sent if the music has ended (the final address of the ZBT as created by the mp3 encoder module has been reached), as opposed to ending the song if the player does poorly, which is what the original DDR game does.

Scoring (Verilog: scoring.v) (Dan Kim)

Scoring takes care of matching the arrows with the steps processed by the camera module. Since the play module delays the music and sends the arrows straight through, the arrows must be delayed in this module to wait for the steps coming in from the camera processing module. As such, a small array of registers is instantiated to hold the arrows. At every beat length (encoded for in number of ready cycles), an arrow or nothing is saved into this buffer. Since everything is on beat (because we didn't get to the harder arrows algorithm), it isn't necessary to do anything off beat, but you would implement it similarly.

While this process is occurring every beat, halfway through the beat a separate process pulls the arrows five beats ahead onto a series of registers designated to be the current arrows (arrows_current). It's pulled half a beat early so that it can track whether a step has been made early or late; if it had been pulled at the normal time, you could not track an accidental early step. Thus, this set of registers is updated every beat with bits indicating the presence of arrows or no bits indicating no arrows.

At this point, a register which tracks whether a score has been marked is set to 0 (score_marked). As steps come in, they are checked to see what cycle periods they fall into. Depending on when they land, they will be matched into decent, good, or perfect with different scores depending on how well the steps match the arrows. These scores are compiled and displayed on the hex reader.

Video (Verilog: video.v) (Dan Kim)

The video module displays arrows on the screen and marches them up the screen according to the beat length. Most of the implementation was taken from lab 5, in which a game of Pong was displayed on the computer monitor. As such, all modules for video display are the same except for the original pong_game module, which is now DDR_video. Within this module, 6 arrows of each type are instantiated, as 5 beats are on the screen at any given time (meaning that at most 6 arrows will be on screen at one time if the screen is divided into 5 beats).

For each arrow, only the y movement can be controlled. The y movement of each arrow is controlled by a mux. If the arrow is inactive, a constant 0 is sent to the y variable. If the arrow is active, as determined by need, the arrow is controlled by another set of registers that continuously decrements by 1 in accordance with the number of cycles needed to get the arrows to the top of the screen in five beats. The module keeps track of which arrow was last activated, and then sets the next arrow to be activated. When the arrow reaches the top of the screen it is inactivated. As such, the arrows are rotated through and reused in order.

Final Thoughts

One of the things that we wasted a lot of time on was attempting to figure out how to decode an mp3 file. Remember that you DO NOT need to do this and will likely be unable to found out how this is done because the process is patented. The ac97 module decodes the mp3 into a waveform, so you only need to worry about analyzing the waveform, and not decode the mp3.

The second thing we would recommend is devote a day to just developing a good way to debug what is being stored to the ZBT. I'd recommend a combination of using the buttons and switches to be able to represent a RAM address and fetch that address' contents from memory and display it on the hex reader (merely a suggestion).

And of course, start early. We were actually really excited about the ideas we came up with, and I'm sure it was disappointing both to 6.111 staff as much as it was to ourselves that we got bogged down by labkit logistics.

Acknowledgements

We would like thank Valdimir, Gim, Jacky, and Hossein for their massive time helping out all the students in lab, keeping the lab open throughout the night on some days, and always doing their best to be helpful and be understanding both for the issues that students had as well as their projects so as to be able to provide insight into how things could be done.