Stepmania (AKA HA-DI-DI-R)

Dash Kosaka and Ian Klatzco
ECE385, Tues 2PM
TAs: Hadi and Dan Petrinsko

[Idea and Overview]

[List of Features]

[Components and Challenges]

[Block Diagram]

[Simulations and Waveforms]

[Module Enumeration]

[Conclusions, Mistakes, and Lessons Learned]

Idea and Overview

We built a Stepmania clone. It's a vertical scrolling rhythm game involving sound, moving sprites, static image or video background, a lifebar, and score system. Every time an "arrow" passed into a target zone, the player needs to be able to press the corresponding button to that arrow in order to score points. Arrows keep appearing until the end of the song.

We used a considerable amount of existing code:

- An Altera example for SD Card Audio playing, from the Altera demonstrations CD.
    - We referenced code from UW (that you linked us in an email), Koushik Roy's VHDL module from the course wiki, and an AlteraForums post.
- A PS2 keyboard module from the course wiki

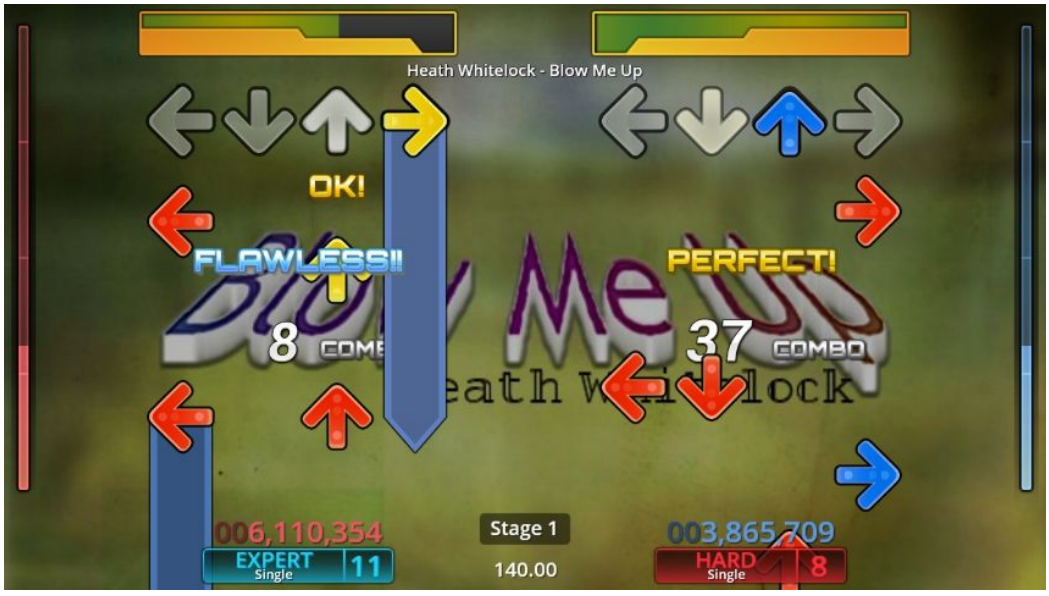


Image courtesy Wikipedia

List of Features

- PS2 keyboard input.
- Drew playfield via VGA, in the same style as Lab 8.
- Made playfield respond to PS2 keypresses (receptors)
- Audio playback, read from SD card. Volume control.
- Reading simfile (step arrow information) from BRAM
- Scorebar / progress bar on side of screen.
- A simfile custom-written by Ian's roommate!

Components and Challenges

We'll discuss how things work.

- Keyboard input

Easy-peasy: load the code from the final project page on the wiki and go. However, our PS2 keyboard eventually broke, which was surprising to us. It stopped working with other devices, as well. As far as we could tell, the keyboard would still send keycodes which we could read on our FPGA, but they would then cycle between different codes.

- Audio

The easiest way to write the audio driver was to store an uncompressed WAV file and then put it on the audio interface's output every divided clock cycle.

Writing a WAV was nontrivial. We found that the provider sd_card_programmer.sof did not work past 512 bytes - it would order blocks incorrectly in memory.

We had to convert our original WAV files to 44.1khz, with 16bit samples from the original OGG format.

We tried Koushik Roy's VHDL code, but couldn't coax it to work.

We tried using some code from a post on Altera's forums, but had issues trying to hook up the VHDL ports and gave up after a short while.

We tried using code from University of Washington that was written for the DE1 and porting it to the DE2. We successfully produced a tone when we fed in a square wave, but could only get static data-reading noises when we read from SRAM. We knew that we were successfully reading SRAM because we could hear the difference between the uninitialized section and audio-file loaded section.

We finally settled on modifying some Altera example code that read a WAV from an SD card. We found that only certain kinds of WAV worked, and inspecting them further, realized that the WAV reader implemented expected a certain kind of WAV header, whereas ffmpeg would add some more to the header. We manually cut out that part of the header and successfully loaded the WAV, and rejoiced when it played.

The code instantiates a NIOS and runs through a basic event loop to read songs from SD card and play them out a standard 3.5mm audio jack.

NIOS runs an entire software stack that interfaces with the PIO.

Here is what main.c does**:**

- Displays messages on the LCD and LEDs

- Sets default volume
- Allows for adjustment in audio (Volume, stop, play)
- Loads SD card, checks for .wav files by reading header (which is where we ran into our problem)
- Starts playing the song out the audio interface.

Here are other important c files:

- WaveLib.c - opening WAVs and getting data from them
- I2C.c I2C protocol but in NIOS.
- AUDIO.c - sending control words to the WM8731

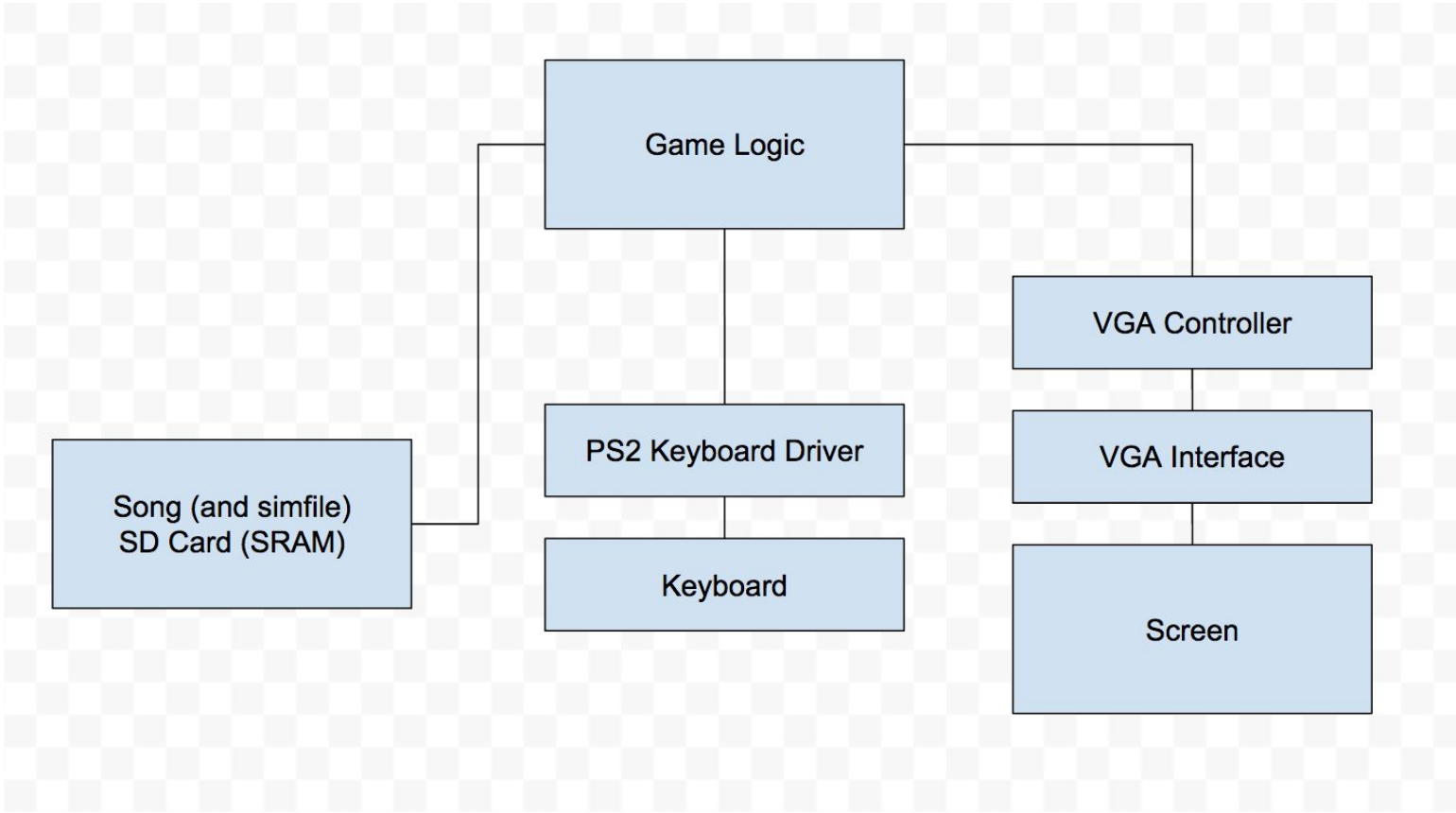
The audio chip is a Wolfson WM8731. It must be initialized with a series of control words via a protocol called I2C. Volume changes are also sent via I2C.

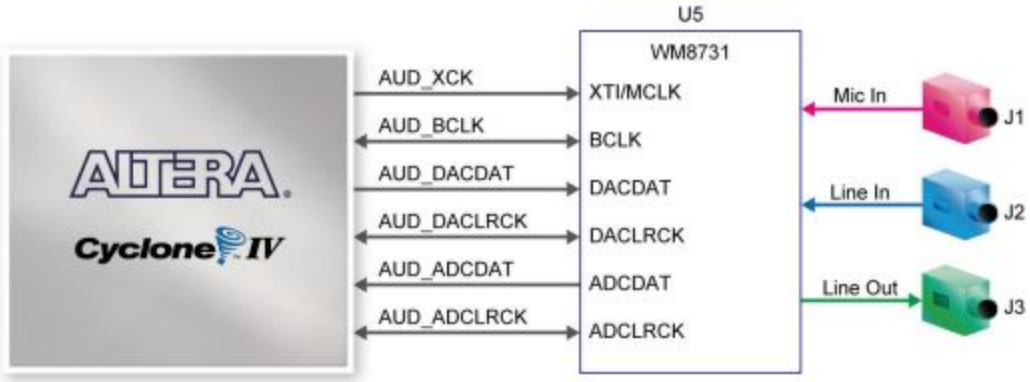
    - Video

Tricky to set up initially: This actually turned out to be a hard thing to do since the jump from going from one ball to many arrows is pretty sizable. It is easy to implement an arrow system that can have one arrow at a time that just floated up the screen. However, our implementation let up to 480 (pixel height) arrows to be drawn at once in a single row. This was accomplished by having a 480 bit register that stored whether or not an arrow was located at certain height and the register basically shifted its values down to zero to simulate the arrows moving. Our implementation was flexible to many different inputs and also allowed for modifications like changing different arrow speeds easy.

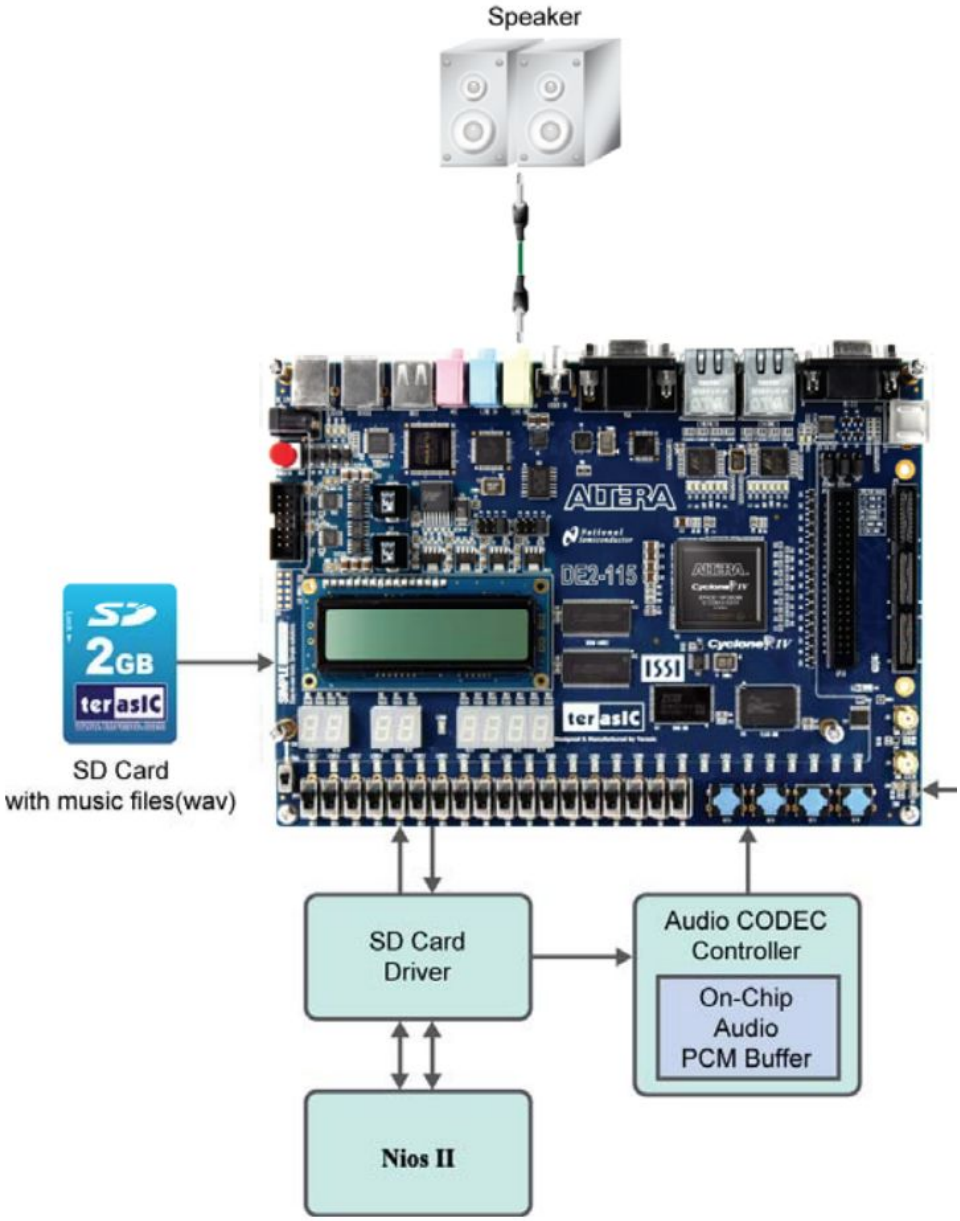
Easier than last time: Since lab 6, it seems like we became experts in SRAM since we knew how to access memory locations in order to write and read before the final project started. We set up a very simple finite state machine in order to accomplish reads from the SRAM to get the current arrow patterns.

Block Diagram





Audio Driver Block Diagram (from Altera Documentation)



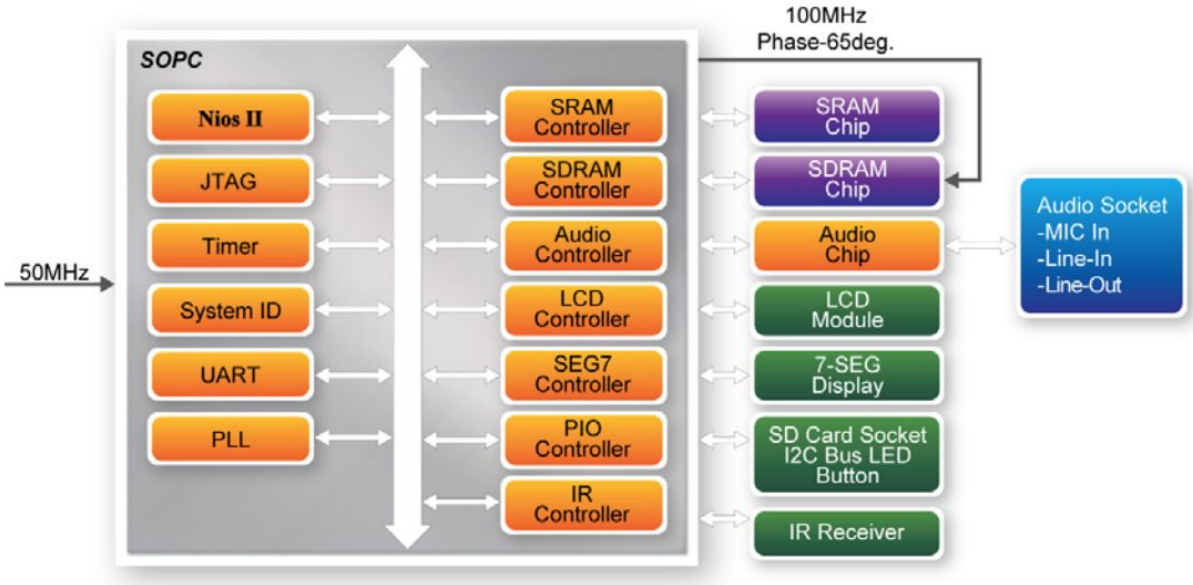


Figure 6-13 Block diagram of the SD music player demonstration

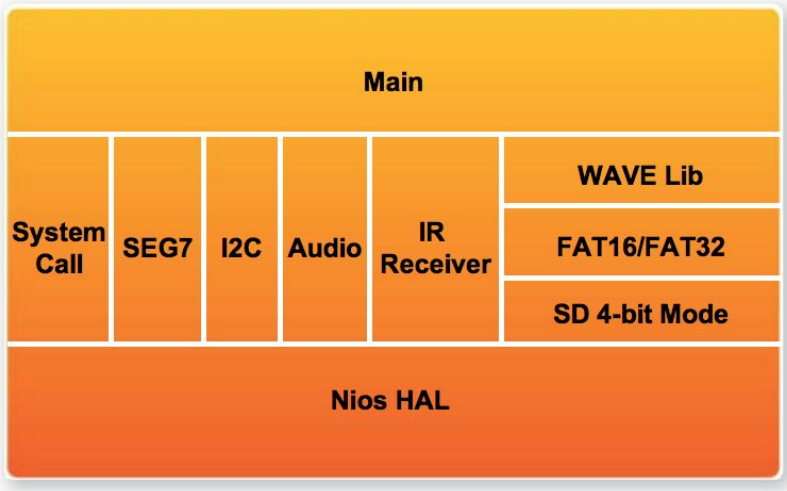
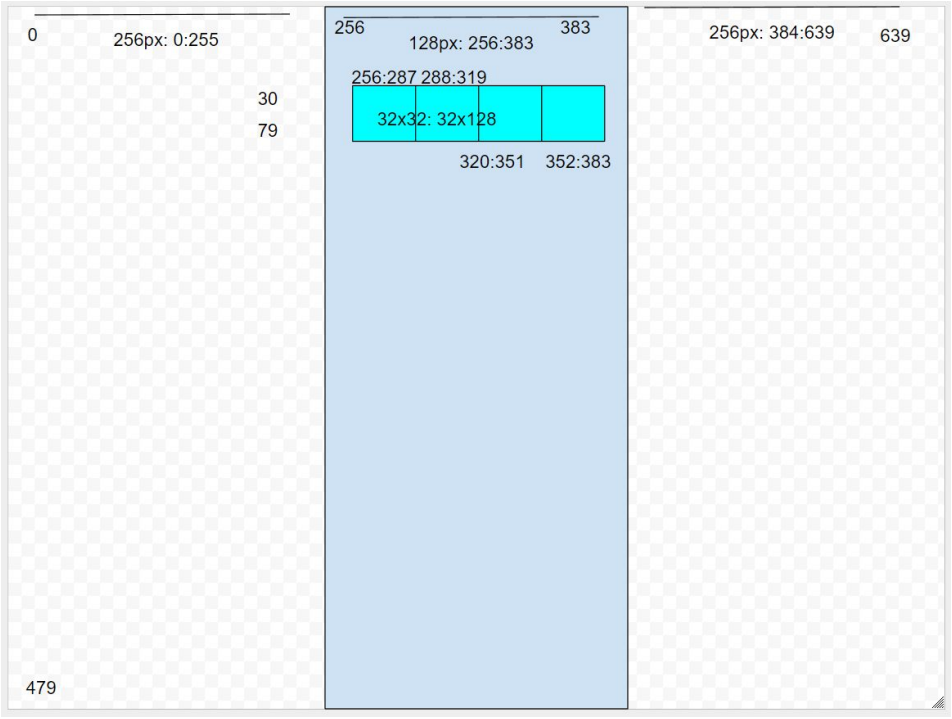


Figure 6-14 Software Stack of the SD music player demonstration

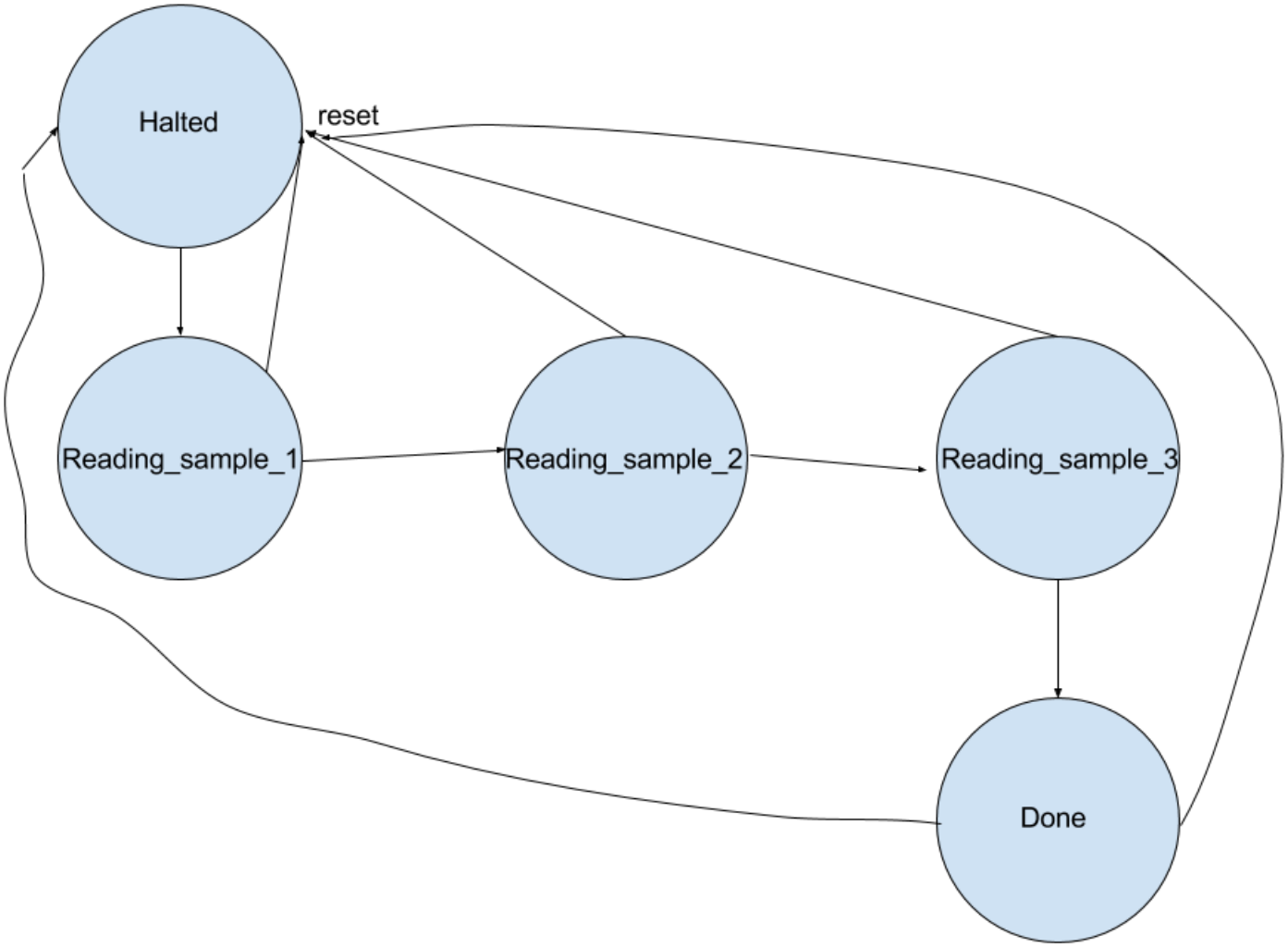


Designing the playfield layout (not to scale, but addresses are correct)

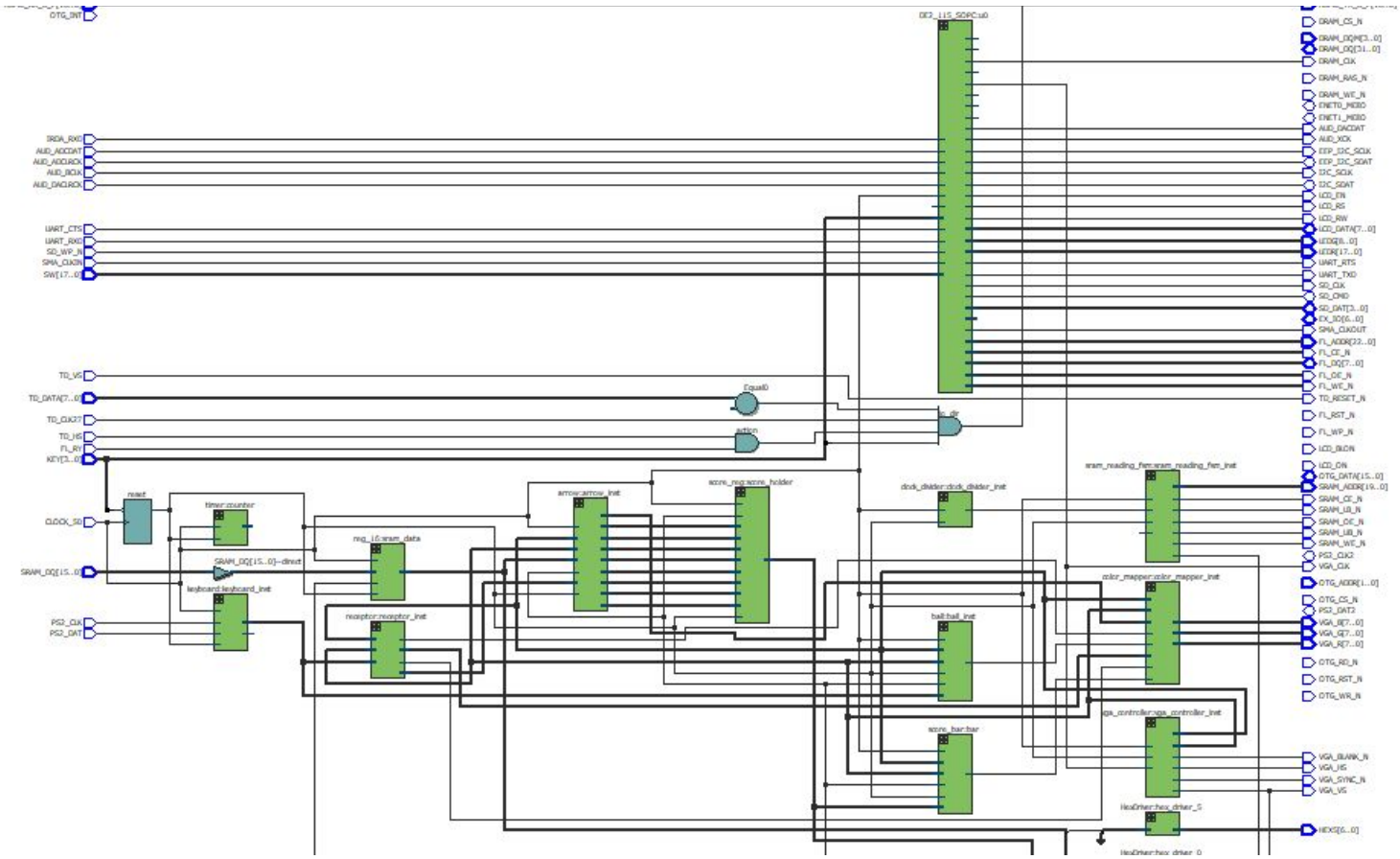
Simulations and Waveforms

We never needed to make any; we asked you if you wanted us to and you said there was no need.

State Diagram (SRAM reading FSM for Step Charts)



RTL Diagram



Design Resource Stats

LUT	11,133 Registers
DSP	15
Memory (BRAM)	2,534,300
Flip-Flop	11,133 Registers
Frequency	69.03 Mhz
Static Power	109.8 mW
Dynamic Power	512.49 mW
Total Power	1102.69 mW

Module Enumeration

Outputs syncing signals and current X and Y position of the “electron gun”.
vga_controller vga_controller_inst(

```
// inputs
    .Clk (Clk),    .Reset (reset),
    .VGA_HS   (VGA_HS),
    .VGA_CLK(VGA_CLK),
// outputs
    .VGA_VS   (VGA_VS),
    .VGA_BLANK_N(VGA_BLANK_N),
    .VGA_SYNC_N (VGA_SYNC_N),
    .DrawX      (DrawX),
    .DrawY      (DrawY)
);
```

For any given x,y location, decides what color to output to the VGA.

```
color_mapper color_mapper_inst(
// inputs
    .is_ball(ball),
    .is_receptor(receptor),
    .is_background(background),
    .is_receptor_background(receptor_background),
    .display_arrow(display_arrow),
    .score_bar(is_score)
    .DrawX  (DrawX),    .DrawY  (DrawY),
// outputs
    .VGA_R  (VGA_R),    .VGA_G  (VGA_G),    .VGA_B  (VGA_B),
);
```

Outputs signals that tell the color mapper whether to draw the receptor, background, or keypress.

```
receptor receptor_inst(
// inputs
    .keycode            (keycode[7:0]),
    .DrawX(DrawX), .DrawY(DrawY),
// outputs
    .is_receptor        (receptor),
    .is_receptor_background(receptor_background),
    .is_background      (background),
    .key_press(key_press)
);
```

Handles arrow hit detection and outputs drawing signals, for color mapper.

```
arrow arrow_inst(
// inputs
    .Clk        (Clk),    .reset          (reset),
    .frame_clk  (VGA_VS),
    .display_signal(sram_data_out),
    .display_arrow(display_arrow),
    .DrawX (DrawX), .DrawY (DrawY),
// outputs
```

```
    .hit0, .hit1, .hit2, .hit3,
    .miss0, .miss1, .miss2, .miss3,
    .key_press(key_press)
);
// endisplay
```

Drivers to display the progress/score bar.

```
HexDriver hex_driver_0 ( .In0 (), .Out0() );
```

Holds current data loaded from SRAM, ie, the next steps to display on the screen.

```
reg_16 sram_data(
// inputs
    .clk(Clk), .reset, .start
    .load (sram_data_load), .din(SRAM_DQ),
// output
    .dout (sram_data_out)
);
```

FSM to send control signals. Implements only reading.

```
sram_reading_fsm sram_reading_fsm_inst(
// inputs
    .Clk, .frame_clk(SIMFILE_CLK), .reset,
// outputs
    .SRAM_CE_N, .SRAM_UB_N, .SRAM_LB_N, .SRAM_OE_N, .SRAM_WE_N,
    .SRAM_ADDR,
    .sram_data_load,
);
```

Register to hold current score/progress value.

```
score_reg score_holder(
// inputs
    .Clk, .reset,
    .frame_clk(VGA_VS),
    .hit0, .hit1, .hit2, .hit3,
    .miss0, .miss1, .miss2, .miss3,
// ouput
    .score
);
```

Outputs signal to tell the color mapper to draw the score bar.

```
score_bar bar(
// inputs
    .Clk, .reset, .frame_clk(VGA_VS),
          .DrawX, .DrawY,
          .score,
// output
          .is_score
```

```
);

_____
Handle PS2 keyboard input
keyboard keyboard_inst(
    .Clk (Clk), .reset  (reset),
    .psClk  (PS2_CLK), .psData(PS2_DAT),
    .keyCode(keycode),
);

_____
Divide the clock (for reading simfiles from SRAM)
clock_divider clock_divider_inst (
        .clk(Clk),
        .rst(reset),
        .clk_div(SIMFILE_CLK)
);
```

Conclusions, Mistakes, and Lessons Learned

This project was rough since we thought we were so close to getting the original sound hook-ups using the SRAM correct, but decided to restart using a completely different approach. The original plan was to use the Koushik Roy's pre-made audio driver however, this was really hard to do with what we needed to accomplish. After hours of struggling with this code and getting a popping noise and static out our speakers (which we confirmed to be from reading data from the SRAM at least), we gave up on the approach since it just did not turn into music. The code we used was pulled from an existing example which used a SD card to read a song instead of the SRAM and we had to do a few hook-ups for that in order to make it work with our project. This code had used NIOS to initialize the audio chip, and read data from an SD card.

On the other side of things, setting up and using the SRAM was pretty easy but was also beneficial to the project. We initially planned to read a song off of the SRAM, but reading an uncompressed In our case, we planned to make a step chart to be used in place of a kind of "level" where the arrows on the screen would correspond to how the step chart was made. This allowed flexibility and creativity of our game since not only could the music (and beat) be changed, but every "level" could be adjusted to the beat also. A really funny effect of using the SRAM to hold this data is that if we had not programmed the SRAM with data beforehand, we would get something that looks like a hard mode since the leftover bits in the SRAM would be loaded as a step chart and fill the screen with arrows.

There was a lot of features that we ended up not implementing, however we think we got the base game working well. If we had (emphasis on had) to add more features to the project, We would update things like graphics to use sprites and nicer backgrounds (even though we also almost finished sprites). Moreover, most of the features in the game depended on other features working correctly also. For example, in order to test the arrows and make sure they work correctly, we needed things like a step chart, hit detection (keyboard and key registration), and even the color mapper to also work correctly. We are positive that if we started to add more features to the project, it would lead to adding more features that were originally intended.

This last point does show things that happen while working on virtually any engineering project. Had we only had a week to do this whole project like any other lab, we may have not had anything like we had during the demo. The

full month we got to work on this allowed the problems that we faced settle in our minds so that we could eventually think of viable solutions over time.

Thank you to Ben Tung for writing us a simfile explicitly for this project. We love you, Ben <3