Group 2 Final Project Design Documentation

Introduction

This document details team twos final project created for ECE 571 Spring 2025. We chose to make an audio and video player using the Nexy's A7 FPGA board which we chose because both team members wanted to continue their studies from previous classes. For Waylon this meant making an audio and visual player and for Dilanthi this meant gaining more experience using the Nexy's A7 board. Our starting point came from *The FPGA Programming Handbook* which walked us through using Vivado and the FPGA board.

Design

We broke our project into a few major parts: the fast forward and pause buttons, the seven segment display, the audio and the SD card. We implemented these parts as follows:

Seven Segment Display Files

To get the seven segment display on the FPGA to show the characters we wanted we first created a display_converter.sv file capable of taking a five bit digit and converting it to its seven bit seven segment equivalent. We chose to implement this as a lookup table because previously Dilanthi had done an uglier version of this code as a case statement in ECE 351. Using a lookup table instead meant that the code became shorter and easier to read which indexes were unused and left blank in case they were needed for additional characters.

We mainly used the FPGA display to show the seconds, minutes and hours into the audio we were playing which we implemented in a set of files called sixty_display.sv and display_anode_driver.sv. sixty_display.sv takes in a binary number from zero to sixty and generates the ones and tens place for the seven segment display and then the display_anode_driver module generates a one-hot encoded output of which digit to light up and the corresponding binary index cycling so fast it looks like all 8 digits are lit at the same time.

Pause and Fast Forward Buttons

The fast forward and pause buttons make use of a debouncer and playback_controller module to function. The debouncer module turns a button press on the FPGA into a single pulse. The pulses generated by these presses are then used by the playback_controller module as inputs to a moore finite state

machine to output playback speed. The state machine we built for this part can be seen at the end of this document in appendix one.

Audio Files

There are two files that we used to make the audio on the FPGA work: audio_buffer.sv. and pwn.sv. The audio_buffer module stores audio samples and then feeds them to the pwm module which changes the duty cycle on the pwm audio output to generate an audio signal that can be heard by humans using a counter.

SD Card Files

The SD card has components for writing commands and reading their responses, a component for reading data off the data line and feeding it into the audio buffer, a component for generating 7-bit CRCs, and a control module to tie it all together and to use the previous modules to initialize and read data off a SD card. The SD card uses 4-bit wide SPIO to get extra bandwidth, as we were unable to clock the SD card past 200khz. There is also a module in here for reading from a ROM and sending that data to the buffer.

Verification

Bugs Caught By Verification

A good number of our bugs were caught by verification, using Vivado's built-in simulation tooling:

Audio Buffer

A counter in the audio buffer was never getting incremented, caught using our testbench

Low Freq Clock Gen

This module was using a modulo operator in a spot that was running at 100MHz and an error was detected by Vivado related to timing

SD Rom

The BROM in this module was getting optimized out. This manifested as reads from the ROM always returning 0. This was found with the testbench, along with a couple off-by-one errors as the ROM was one cycle slower than expected.

Read Command

The response_type was not getting saved in a register, quickly throwing off the state machine driving the logic for this module. This was detected in a testbench.

Read Data

The Read Data module was always writing to the same address in the audio buffer, never changing. This was found in a testbench.

Bugs Not Caught By Verification

A few of our bugs were not caught by verification and instead had to be witnessed in real life while handling the FPGA board.

Button Debounce

We initially had an issue where our button presses were inconsistently being registered depending on when someone hit the button and for how long which we found via physically pressing the buttons. The module, which was supposed to remember that it was pressed and should have only output for one cycle, did not remember and would cycle on and off. To resolve this we rewrote the module and added a bonus register to keep track of this, and added some comprehensive tests to make sure no new errors occurred.

Sound Distortion

We also encountered multiple issues with sound distortion which needed to be heard in person to be found. These were caused by 100 MHz not being fast enough for a 16 bit audio which caused a 1.5 KHz whine, the audio buffer not getting fed fast enough which made the sound gravely and incomplete and a buggy clock that caused the audio buffer to run at three times the speed making it sound like chipmunk music. We currently still have a little audio distortion caused by having to truncate the 16 bit audio which makes the audio samples sound robotic and off tune.

Modular Clock

Next issue we had was that the clock had an issue during reset where it would stop ticking and reset to its default value. This looked correct in verification but in real life it meant nothing using the modular clock saw resets because we were using synchronous resets.

Off By One Errors

There were a large number of off-by-one errors caught here. Some were because of bad testbench design, with the testbench half of a transaction accepting

something subtly wrong from the DUT half. Some seemed to be legitimately different in HW versus in testing, with one counter offset having to be changed to work in HW.

Issues Requiring Redesign

We were humbled by our initial proposal and ran into a few issues requiring redesign:

Bandwidth

It was a struggle working with the SD bus. For some reason the FPGA refused to drive the SD_CMD line at a rate faster than 200khz, and even then sometimes the SD controller state machine would wildly transition states. There is some weird undefined behaviour going on, maybe some weird x propagation or something? Maybe potential timing issues? Whatever the case, we were able to get barely enough bandwidth for 16-bit audio as long as we were using all 4 sd data lines instead of just 1.

Time

There were a lot of unknowns coming into this project. Getting set up with FPGA development had many different slowdowns. Learning how to use Vivado to get code on the board, learning how to interface with the hardware, and learning how to use Vivado to effectively debug the FPGA all took significant amounts of time. Finding FPGA bugs is significantly slower than finding bugs in simulation because of the time it takes to compile and stick code on the FPGA. This project had many different parts that all came together in a kind of clunky way, but in the end we learned a lot about tools, hardware, and audio.

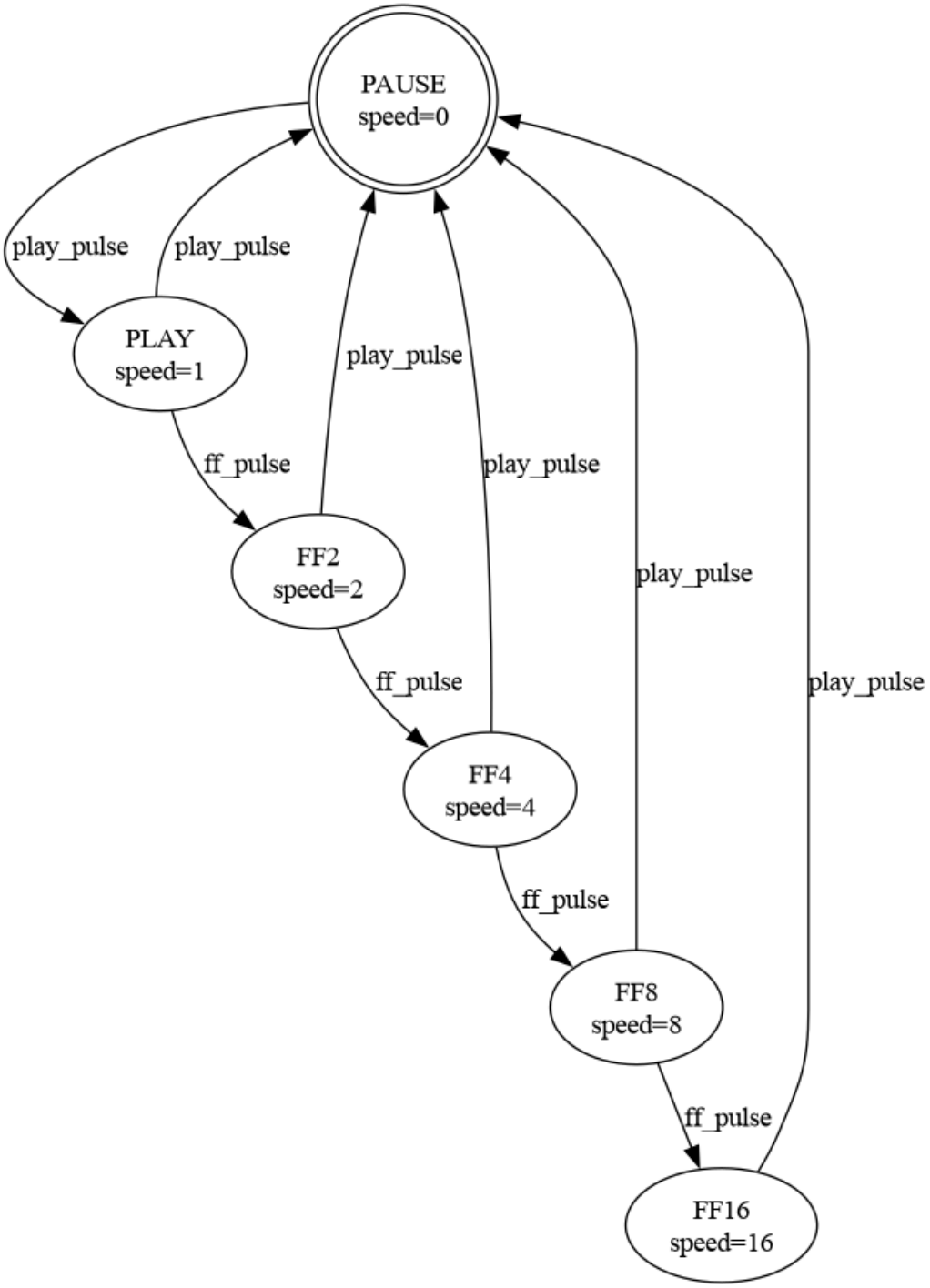
References

- https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual

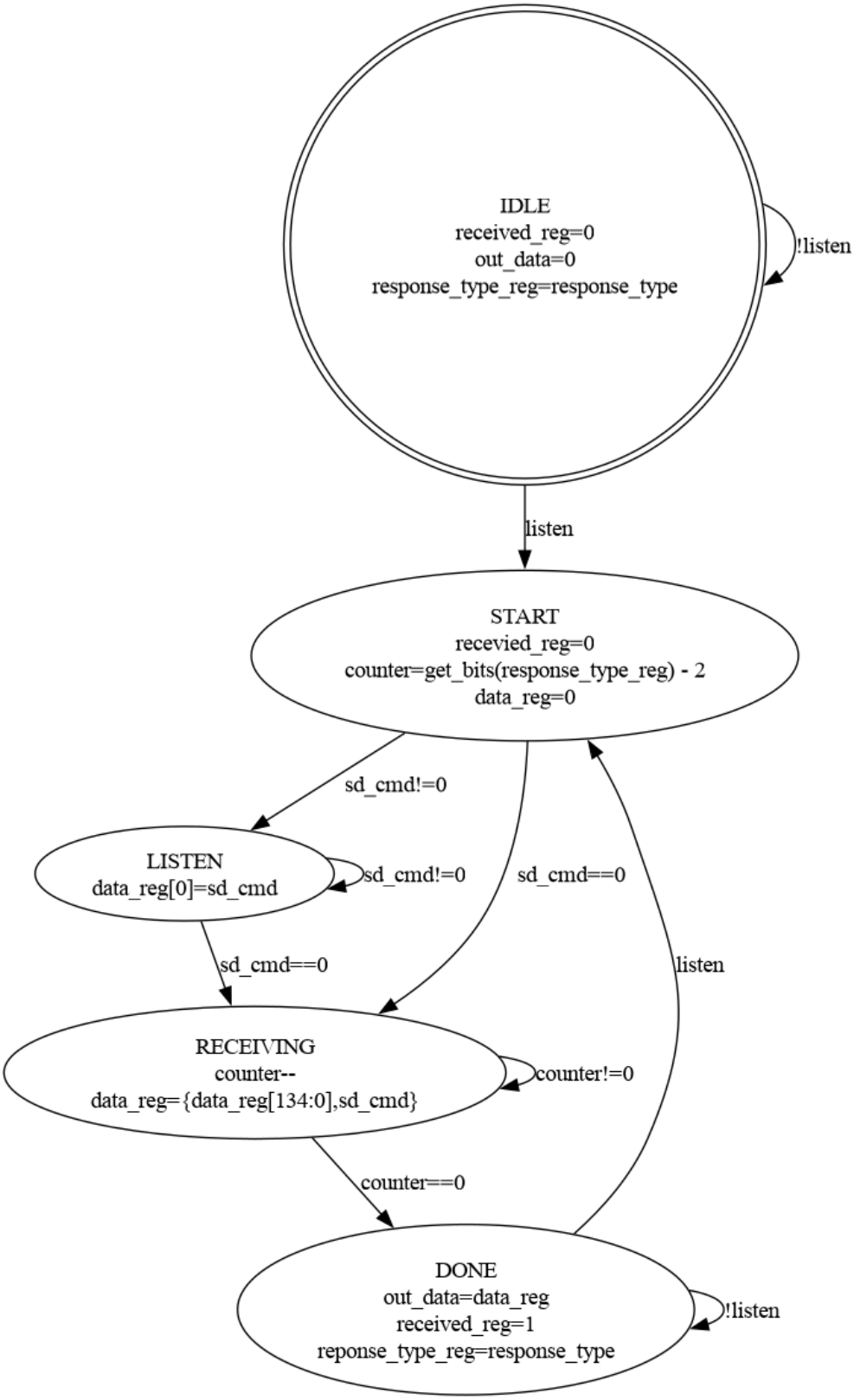
Appendix One: Finite State Machine Diagrams

NOTE: These are all also in the doc/ folder as pngs

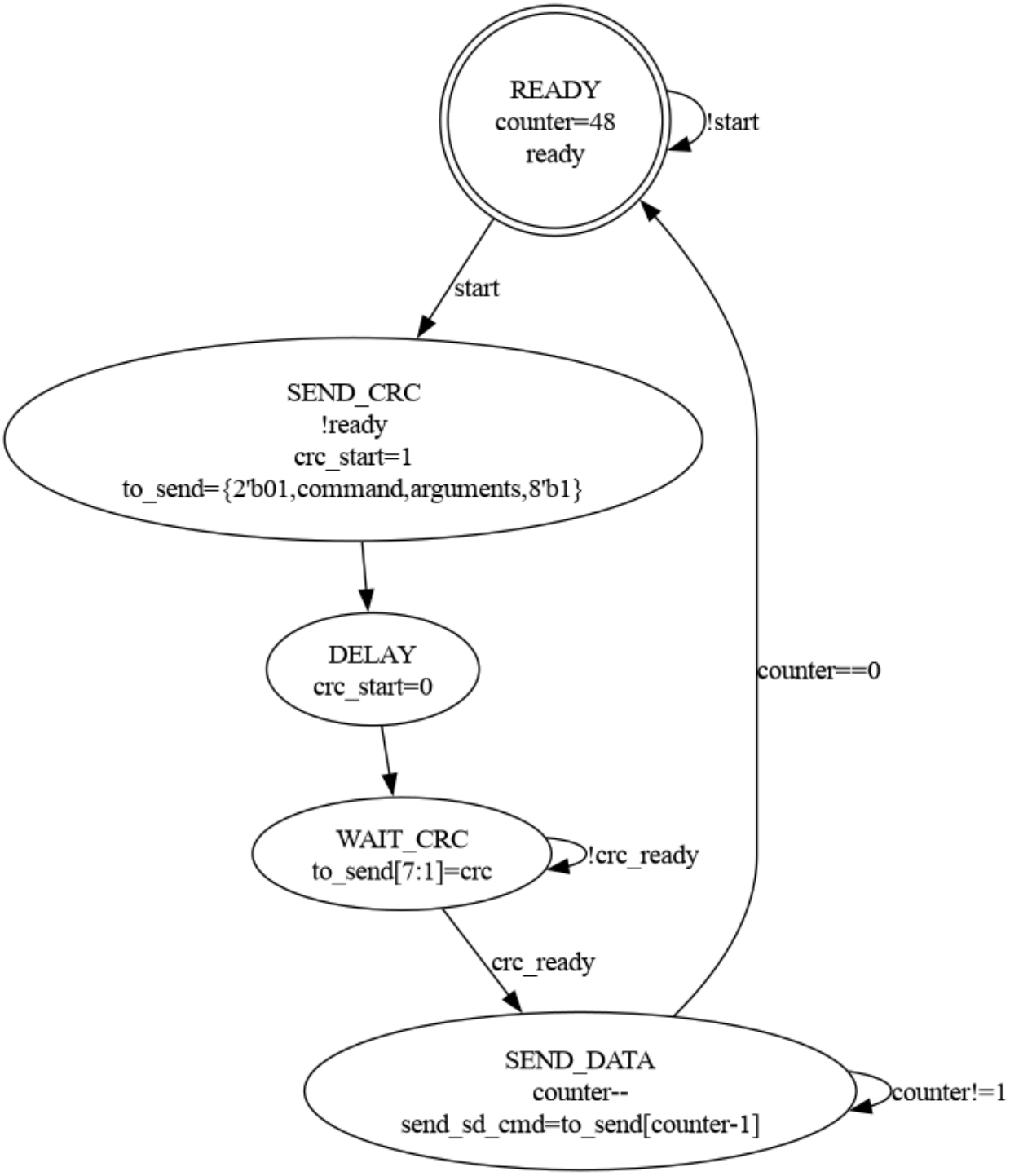
Playback_Controller FSM Diagram



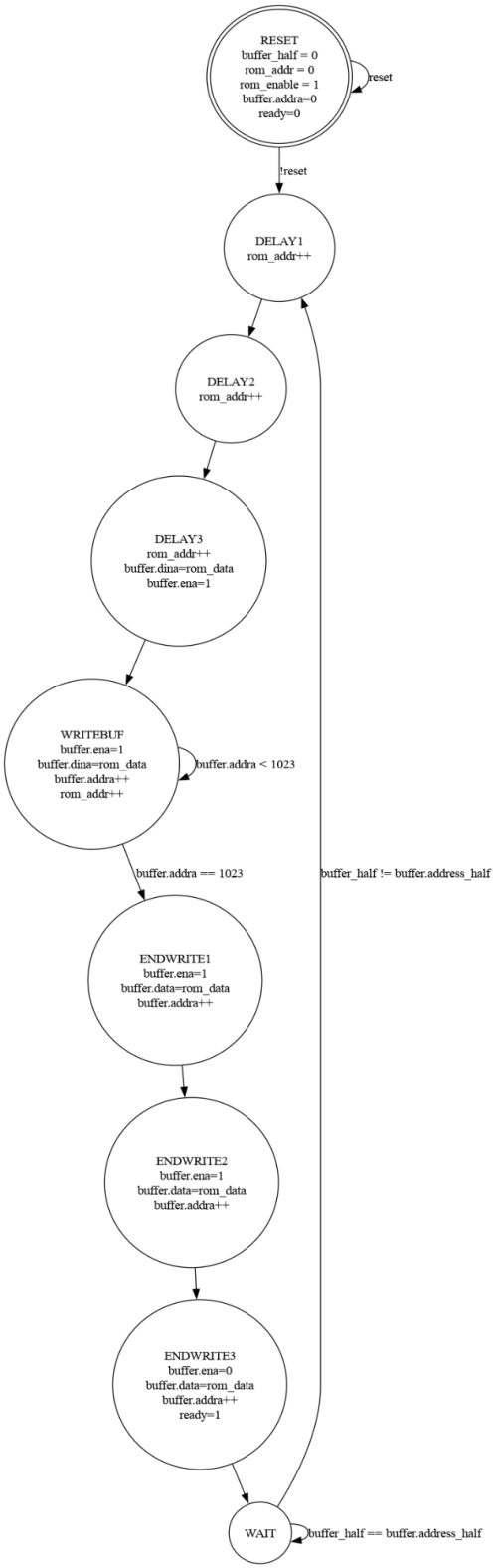
Read Command State Machine



Send Command State Machine



Rom Sd State Machine



Sd Controller State Machine

NOTE: For a readable version look in doc/, there is also a copy at https://imgur.com/a/MEw8YYY