Voice Recorder

Ben Wolsieffer and Afia Semin
ENGS 31 - 18 X

Abstract

This project is a voice recorder that captures audio from an onboard microphone and plays it back through a speaker. Audio data is stored on an SD card, allowing for long recording times and long-term storage. It is also possible to write an audio file to the SD card from a computer and play it back using the voice recorder. The recorder was implemented on a Xilinx Artix-7 FPGA using the Digilent Basys 3 development board, as a well as a variety of Digilent Pmods.

Table of Contents

[1] To see enlarged versions of the diagrams in this appendix, click on the images.

1. Introduction

The primary objective of this project was to create a digital voice recorder that was capable of recording audio from a microphone, storing it and then playing it back through a speaker. An additional goal after accomplishing the primary goal was to store the audio on an SD card, allowing much longer recording times.

2. Design solution

2.1. Specifications

This system is a voice recorder that records audio from an onboard microphone and stores it onto an SD card. It is also capable of playing back audio from the SD card. The voice recorder uses the onboard LEDs and pushbuttons of the Basys 3 for its user interface. It uses a Pmod MIC3 connected to port JA for audio input, and a Pmod DA2 connected to port JXADC to generate an analog audio signal. The analog signal is passed through an adapter board to a Pmod AMP2 audio amplifier, which drives a speaker. Finally, an SD card socket (Pmod SD) is attached to port JB. A photograph of this setup is shown in Appendix A. The connections and internal pin mappings between the FPGA and Basys 3 ports are shown in Figure B1.

2.2. Operating Instructions

To operate the system, the Pmods must be connected as described in the specifications section and shown in the front panel diagram (Appendix A). An SD card must be inserted in the

socket before the system is powered on. If the SD card is removed while the system is on, the card must be reinserted and the system must be restarted before it can be used. To begin a recording, press the record button. To stop the recording, press the record button again. The recording can be played back at a chosen speed using any of the play buttons. The recording will play until it is finished, or the same button is pressed again. The row of lights along the bottom of the board allows the user to determine the current state of the system. The SD card driver respects the write protect switch on the SD card. If the record LED does not turn on after the record button is pressed, make sure the write protect switch is in the unlocked position.

2.3. Theory of Operation

The main components of the system are the UI controller, the audio controller, and the SD card driver, as well hardware interface components for the A/D and D/A convertors (their connections are shown in Figure B2).

Audio is recorded with a Pmod MIC3, which uses a Texas Instruments ADCS7476 A/D convertor, and audio is played using a Pmod DA2 with a Texas Instruments DAC121S101 D/A convertor. The driver for the Pmod MIC3 is identical to the one used in the last two labs, except for the addition of a done signal that is asserted as the last bit is shifted in. The Pmod DA2 driver is similar to the A/D driver, except it is a parallel to serial shift register instead of a serial to parallel shift register (see Figure B3 and C1).

The A/D and D/A drivers are connected by the audio controller. The audio controller is a finite state machine (described in Figure C2) that outputs a take sample pulse at the audio sample rate (specified by the UI controller). This signal is continuously passed to the A/D driver, even during playback, but it is gated by an enable signal before being passed to the D/A driver to

prevent unwanted noises from being played by the D/A while recording. The audio controller is also responsible for writing the A/D and D/A data to and from the RAM buffer. It receives the current position in the RAM buffer from the SD card driver and uses it to decide whether there is enough data in the circular buffer to continue playback, and also whether the buffer is full while recording (see Figure B4 for details of the bounds checking). The circular buffer and bounds checking are implemented using a monotonically increasing counter that represents the number of samples that have been played or recorded. The SD driver has its own version of this counter, which keeps track of how many samples it has read or written. The actual position in the RAM is the sample index modulo the RAM size. The RAM size is configured to be a power of two (in this case, 131,072 or 2^{17}), which turns the modulus operation into an efficient bit mask. The sample index is used in the comparison operations, which makes it unnecessary to explicitly determine when the circular buffer wraps.

Data in the circular buffer is transferred to an SD card for permanent storage. SD cards have two modes of communication, a native high-performance protocol using four data lines, and a simpler SPI mode. Communication is symmetrical, with both the master and slave (SD card) shifting out data on the falling edge of the clock and latching data on the rising edge. The master initiates all communication.

In this design, the SD card driver has the minimum possible interaction with the audio components of the system. The SD driver consists of three levels of components, starting at the lowest level with a pair of components that constantly send and receive from the SPI bus in 8-bit increments. The SD protocol operates on the byte level, so the smallest unit the higher-level components of the driver will have to work with is one byte (8 bits). The sending and receiving

components operate as parallel to serial and serial to parallel shift register convertors, respectively. These components are always shifting, and they assert a signal as the last bit is being shifted out or in. In addition, the sending component has a signal that causes an input to be stored into the parallel register. When there is no data to send, the bus is held high (ones are shifted out), which indicates that it is idle. The SD card also holds its master-out-slave-in (MOSI) line high when idle, making it possible to detect traffic on the bus by waiting for a received byte that is not equal to 0xFF (the first byte of any transfer is guaranteed to have at least one zero). The sending component is also responsible for asserting chip select (CS) when the first data is sent. This is done at such a low level because CS serves as the framing signal for the data, and must occur as the first bit is sent. The higher-level components of the driver do not operate at the bit level, making it difficult for them to accomplish this precise timing. In this implementation, the CS signal is never deasserted, because the SD cards tested were able to maintain their framing without it. There is anecdotal evidence (web comments) that some cards require CS to be deasserted and asserted between commands. The detailed design of these components is shown in Figures B6 and B7.

Above the sending and receiving components is the command driver, which executes commands and returns their responses. It is implemented as a finite state machine (FSM), fully described in Figure C5. When a start signal is asserted, the state machines registers a command index and argument, sends it to the card, waits for a response and then asserts a done signal. Most commands have a short response ranging from 1 byte to 5 bytes. The first byte always has the same format, and indicates whether an error occurred. Certain commands, in particular those that read from the card, send a second response that contains the data that was read. While the

driver returns the first response as a 32 bit signal, the second response is returned as a single byte and an index that increments as the response is read. This makes it easy to directly map this data into RAM. Data writes are performed using a similar mechanism in the opposite direction, with the master sending a data block after receiving the first response from the card.

The highest level of the SD card driver manages the sequencing of commands during initialization, as well as the transfer of data between the audio RAM and the SD command driver. The SD card driver is also an FSM, with many states named after SD command mnemonics (described in Figure C5, the state diagram for the SD card driver, and B8, which is an abbreviated version of the block diagram for the command controller and the card driver; a more complete block diagram for this part of the project is not available due to complexity and time constraints). The SD card standard has gone through several revisions, and therefore a fully compatible driver must perform different commands and checks depending on which version of the standard a particular card implements. To make implementation simpler, only SD version 2 and greater cards were supported. In addition, only SDHC and SDXC cards (which use sector addressing) were available to test, so support for standard capacity SD cards (which use byte addressing) is only theoretical. Data transfers are performed using single block read and write commands. While using multiblock data transfer commands would have been more efficient, there was not sufficient time to implement them. This is particularly true for writing, as a naive single block writing algorithm suffers from the FLASH write amplification issue. FLASH memory must be erased before it can be written, and the minimum size that can be erased is much larger than the minimum size that can be written. Therefore, to change a single byte, it is necessary to buffer the entire erase block, erase it and then write the data back with the single

byte changed. This reduces performance and can cause premature failure of the FLASH device because FLASH can only withstand a limited number of erase cycles. Multiblock writes work around this problem by making it possible for the SD card firmware to buffer many writes before erasing a block. A possibly simpler solution is also available, because the SD protocol allows sectors to be explicitly erased. In the case of the voice recorder, a new recording overwrites an old recording, making it possible to preemptively erase a large block ahead of the recording. The SD driver implements these commands, erasing 4 MiB blocks ahead of the current recording position. It is possible to determine the exact erase block size of a particular card, but this was not done due to time constraints.

The driver implements minimal error checking and recovery. Error codes or unexpected values in command responses are detected, but no attempt is made to recover. The driver simply stops in the case of errors. Timeouts are not implemented, meaning that if a card is disconnected in the middle of a command, the driver will hang. The SD protocol optionally uses a cyclic redundancy check (CRC) to verify the integrity of commands and data, but this is not implemented by this driver beyond the first few initialization commands that require it. Rather than calculating a CRC, hardcoded values are used because the contents of the commands are fixed.

The SD driver interacts with the audio components through the block RAM as well as an index that indicates which sample of the audio file is currently being recorded or played. The SD driver only writes during recording when there are at least 512 bytes (one sector) buffered in the RAM, and only reads during playback when there are at least 512 bytes free in the circular buffer.

The high-level state of the system is controlled by the user interface (UI) controller. It takes the debounced monopulsed buttons as inputs and produces record and play signals as outputs. In addition, it has two inputs from the audio controller and SD driver that indicate that they are done (either after the recording or after playback was stopped manually, or the end of the file was reached). If the UI controller is in the idle state and one of the buttons is pressed, it transitions to the corresponding state, and asserts either the play or record output. It remains in that state until the same button is pressed again, or either the audio controller or SD controller indicates that it is done. In either case, it waits for both done signals to be asserted before returning to the idle state. The audio controller and SD driver also wait for their counterparts' done signals, which synchronizes the three major components of the system. To implement the fast and slow play capabilities, the UI controller outputs a signal that controls the speed of the take sample pulse. The full state diagram and block diagram can be seen in Figures C3 and B5, respectively.

2.4. Construction and Debugging

The first components that were designed and implemented were the drivers for the A/D and D/A convertors. The A/D convertor driver was already written for the last few labs, so it was simply modified to include a done signal that is asserted as the last bit is shifted in. Once these components were written and tested in simulation (see Appendix I for simulation waveforms), a simple demo system was created that simply forwarded data from the A/D to the D/A. This worked on the first try, except for an issue with a different pin mapping between the Pmod AD1 and the Pmod MIC3. The next step was to implement the system that would accomplish the first design goal. This was the design presented at the design review. This system worked with few

issues, other than a strange problem where the Pmod ports of the Basys 3 seemingly stopped working, although the buttons and LEDs still worked. This problem disappeared the following day and never occurred again. Once this part of the project was working, design was started on the SD card driver. This began with the SD sending component, which was tested in simulation (Appendix I) and then with a simple test harness that sent a single command. Debugging this system took a significant amount of time because it was not known that data was supposed to be clocked out on the falling edge. Once this was solved, the component was turned into the SD command controller, which allowed a higher level driver to execute a series of commands to initialize the card. In this first phase, testing was done without any connection to the audio system. Once it was possible to fully initialize the SD card, work was begun to integrate it into the audio system. First, reading was implemented. Reading was deemed slightly easier than writing because there was no need to worry about erasing. In order to have something to read, a Python program was written to allow audio files to be written to an SD card in the format expected by the voice recorder (see Appendix J). The most difficult part of getting this to work was making sure the audio controller and SD driver stayed within the correct bounds of the buffer. Lastly, writing and erasing were implemented, which had similar issues as reading.

3. Evaluation

Although the design goals were satisfied, there are a number of features that would almost certainly be necessary for this system to have any real applications or practicality. It is only capable of recording a single audio file, and new recording overwrites the previous recordings. A practical voice recorder needs to be able to store multiple recordings, ideally in a file system that is interoperable with other systems. The ability to store multiple recordings

would also likely require a better user interface that provides more feedback to the user than just a few LEDs. It is not possible to swap SD cards without restarting the system, which is inconvenient. Many of the more complicated features are unsuited to an FPGA implementation, and would be much simpler to implement using a microprocessor.

4. Conclusions

The final design accomplished the initial goal of being able to record and play audio from the onboard RAM, as well as the secondary goal of writing that data to an SD card. The primary goal was easily achievable with the skills and resources obtained in ENGS 31. The SD card portion of the project was more difficult and required significant research outside of what was taught in class (How to Use MMC/SDC, 2018; SD Specifications, 2017). This meant that the TAs had limited ability to help with this part of the design. Students who want to implement an SD card driver in the future should be prepared to perform independent research and read technical specifications.

5. Acknowledgments

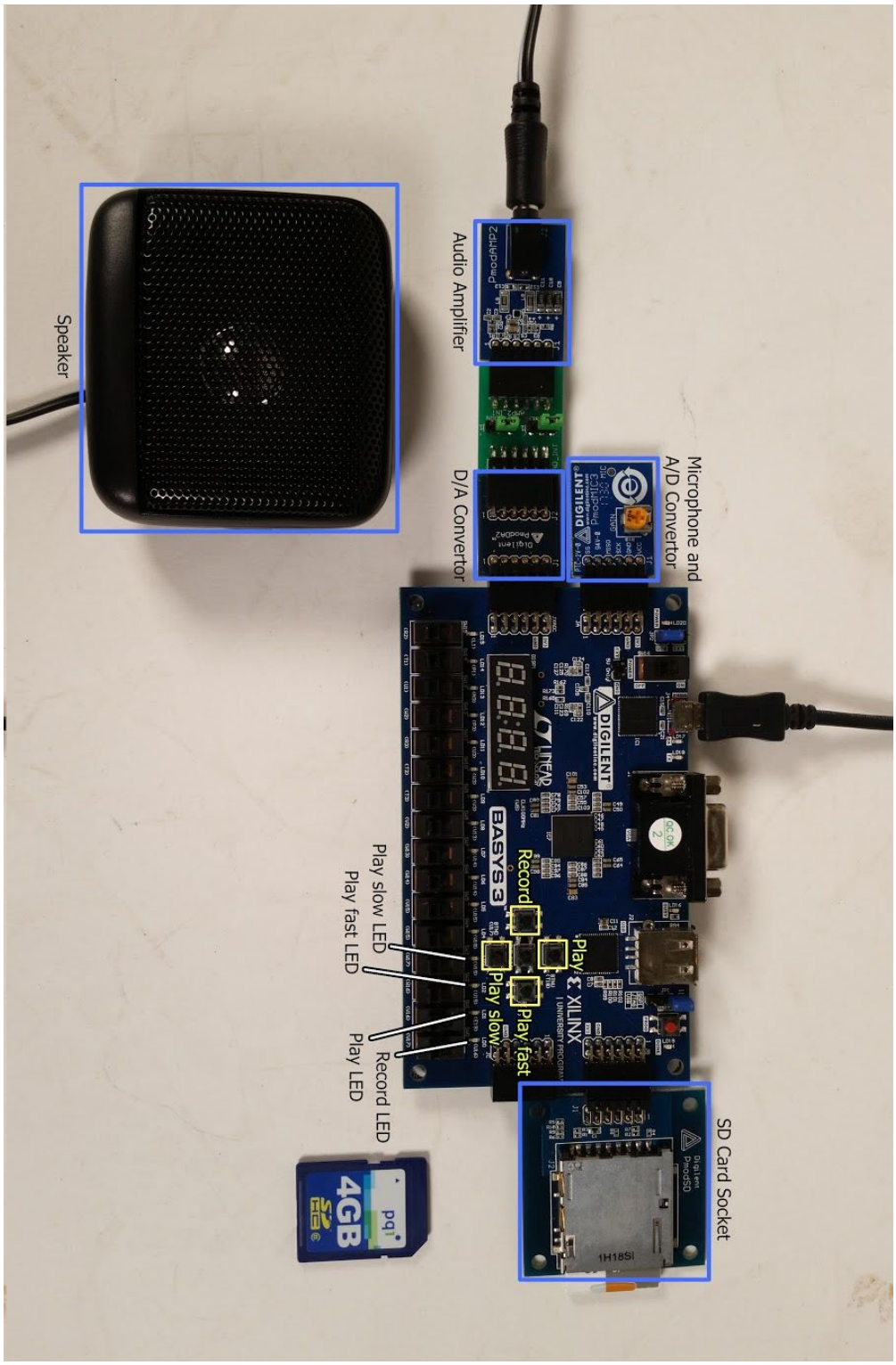
We gained the skills to implement this project from Professor Hansen's lectures and the laboratory exercises. In addition, our design process was facilitated in consultation with our learning fellow, Ella Ryan. The audio controller and the modifications to the A/D driver were done by Ben Wolsieffer. The D/A driver and the UI controller were written by Afia Semin. The SD card driver design and implementation were done by Ben Wolsieffer. The SD card portion of the project was outside the specifications as initially proposed at the design review.

6. References

"How to Use MMC/SDC." March 13, 2018. Accessed August 16, 2018. http://elm-chan.org/docs/mmc/mmc_e.html.

“SD Specifications Part 1: Physical Layer Simplified Specification." April 10, 2017. SD Association. Accessed August 12, 2018. https://www.sdcard.org/downloads/pls/.

Appendix A - Annotated front panel



Appendix B - Block diagrams

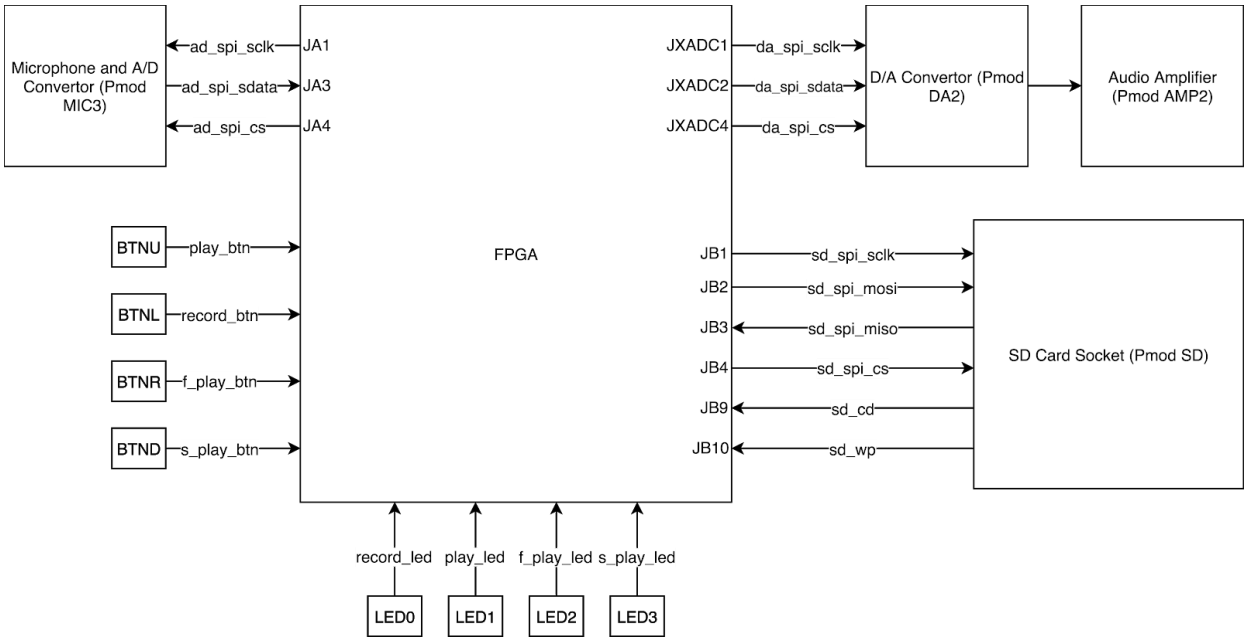


Figure B1. Hardware Block Diagram

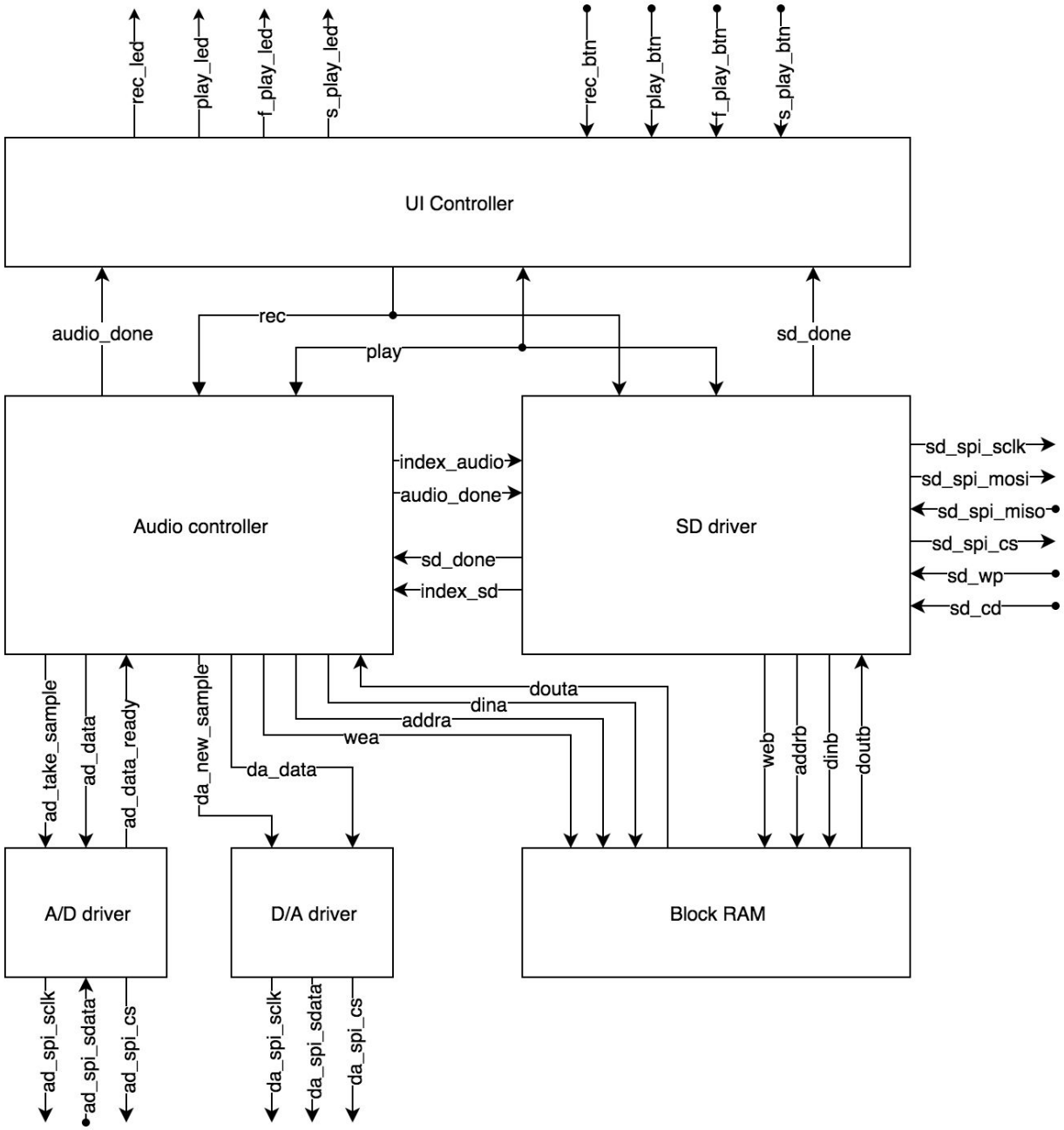


Figure B2. Top level block diagram

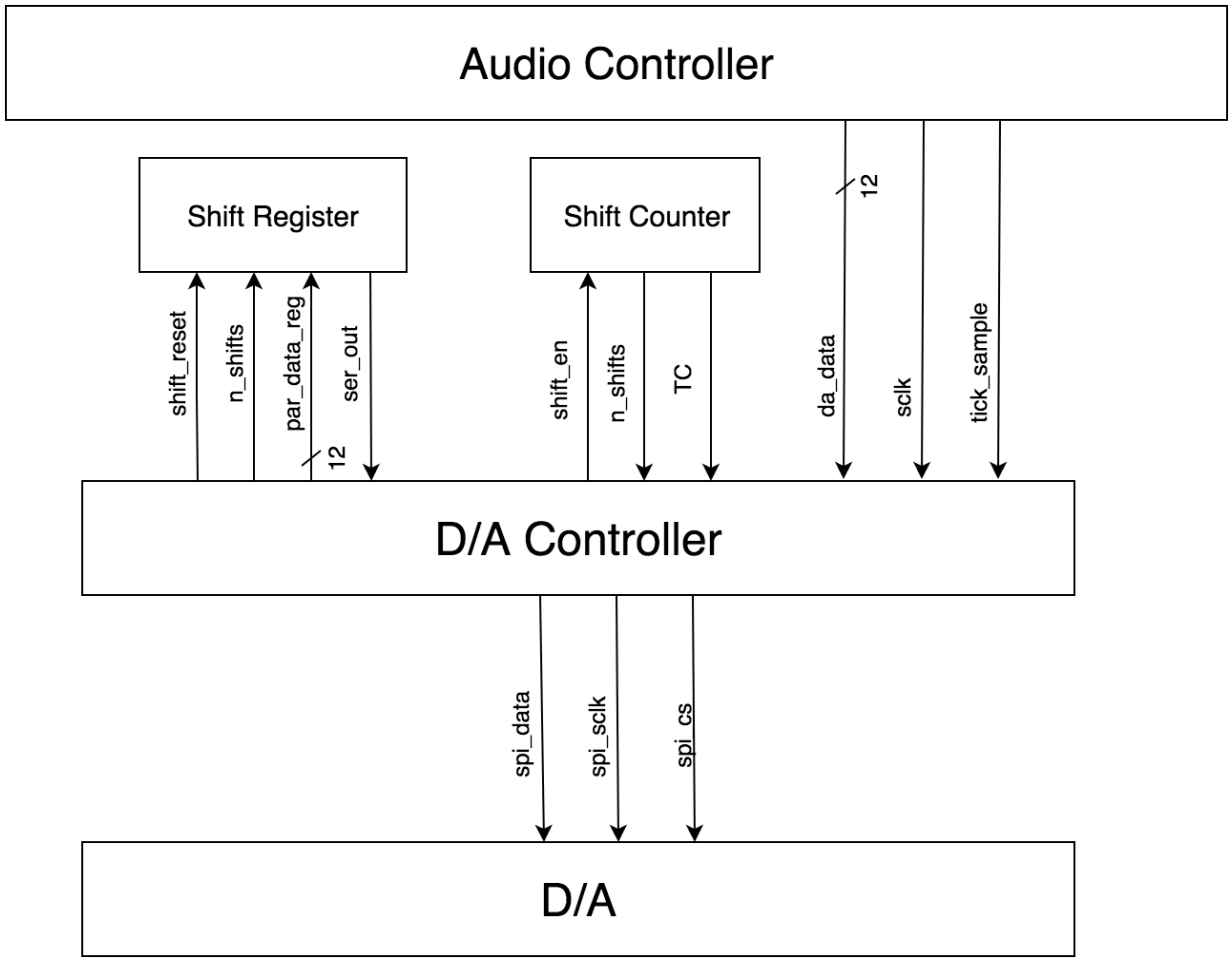


Figure B3. D/A driver block diagram

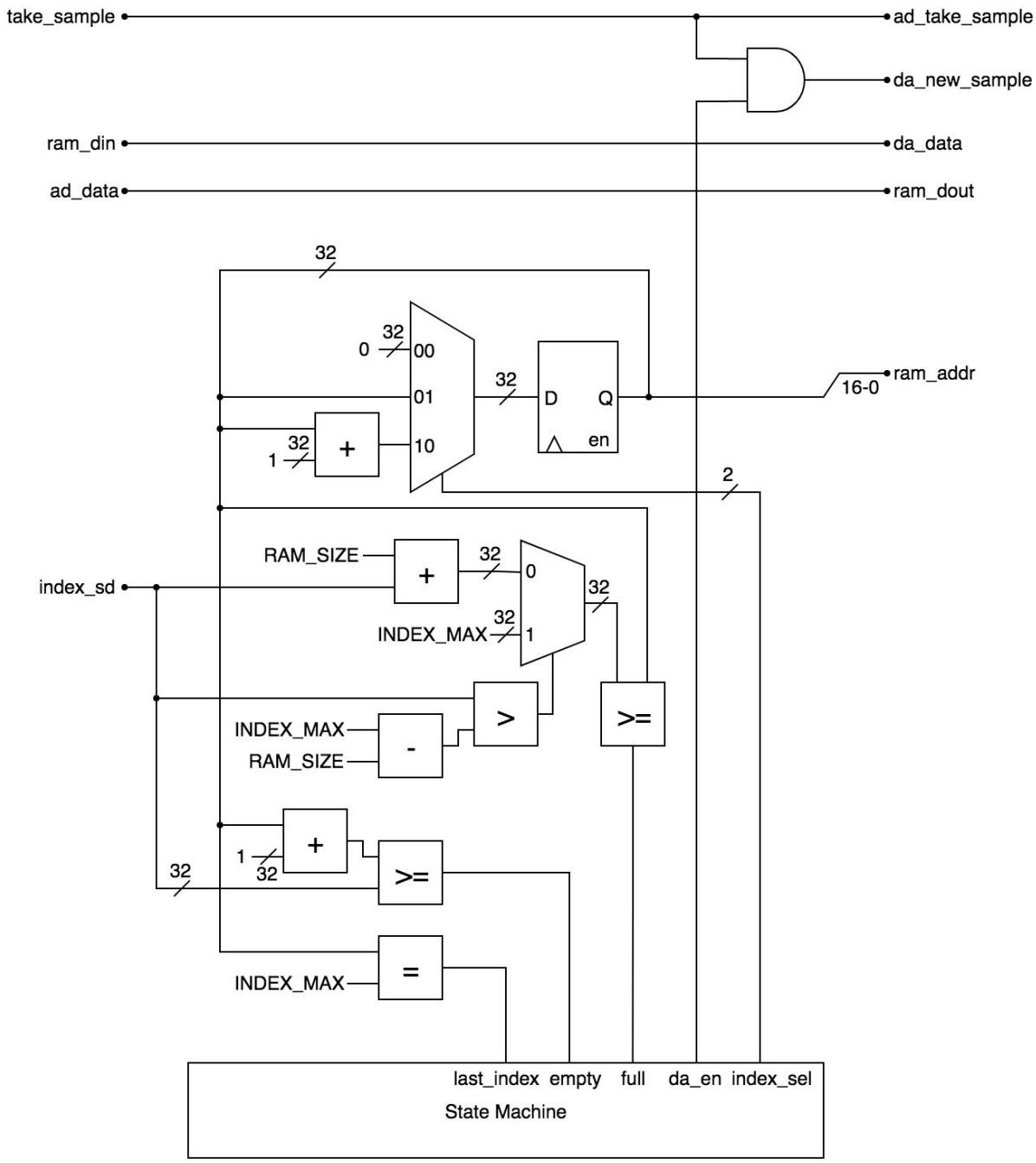


Figure B4. Audio controller block diagram

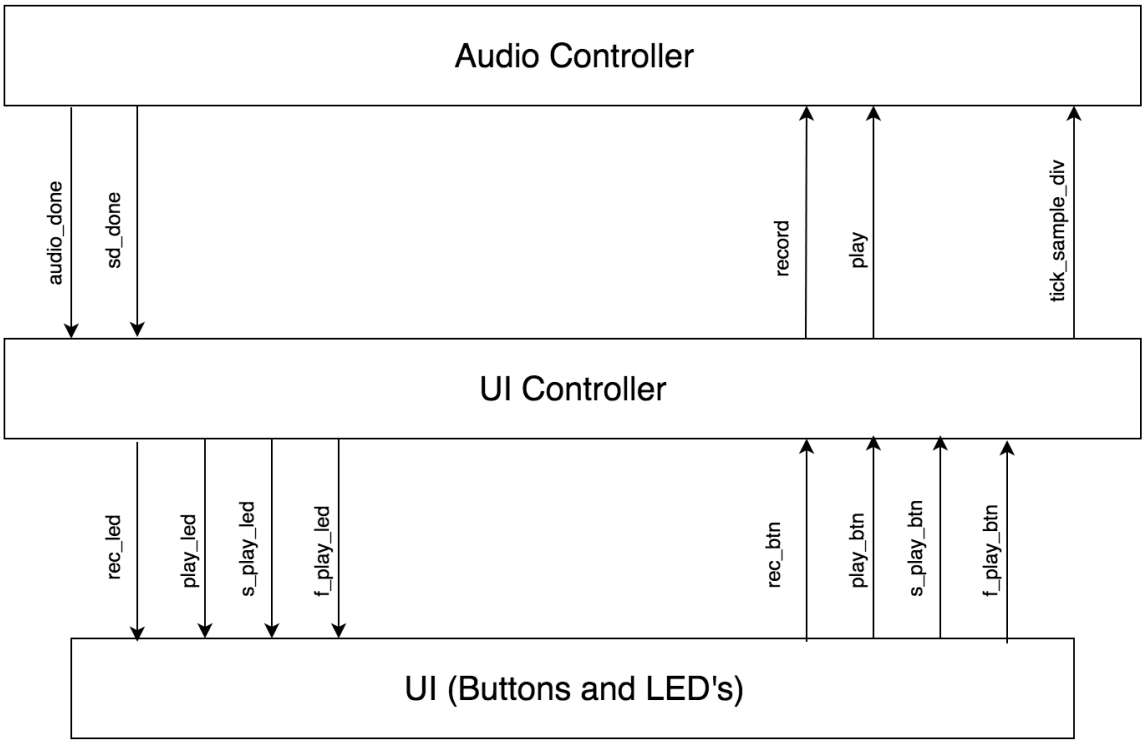


Figure B5. UI controller block diagram

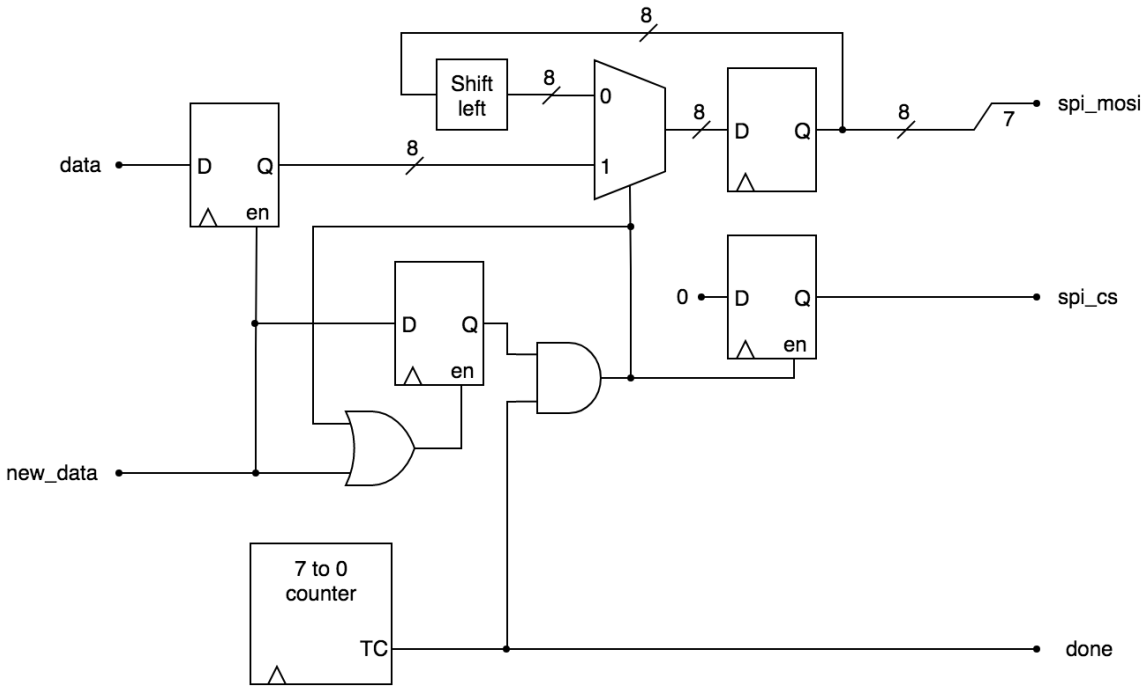


Figure B6. SD send component logic diagram

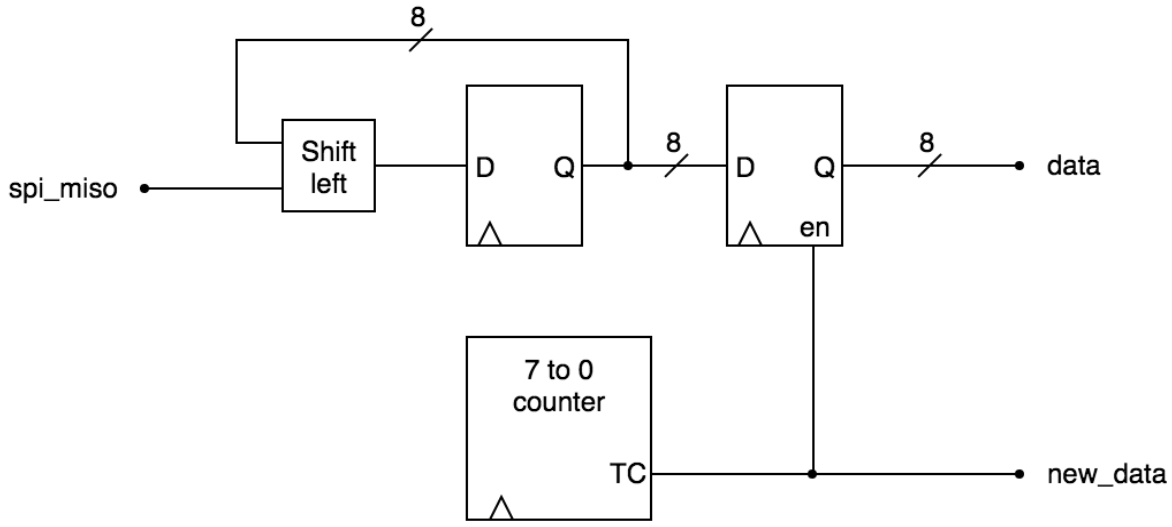


Figure B7. SD receive component logic diagram

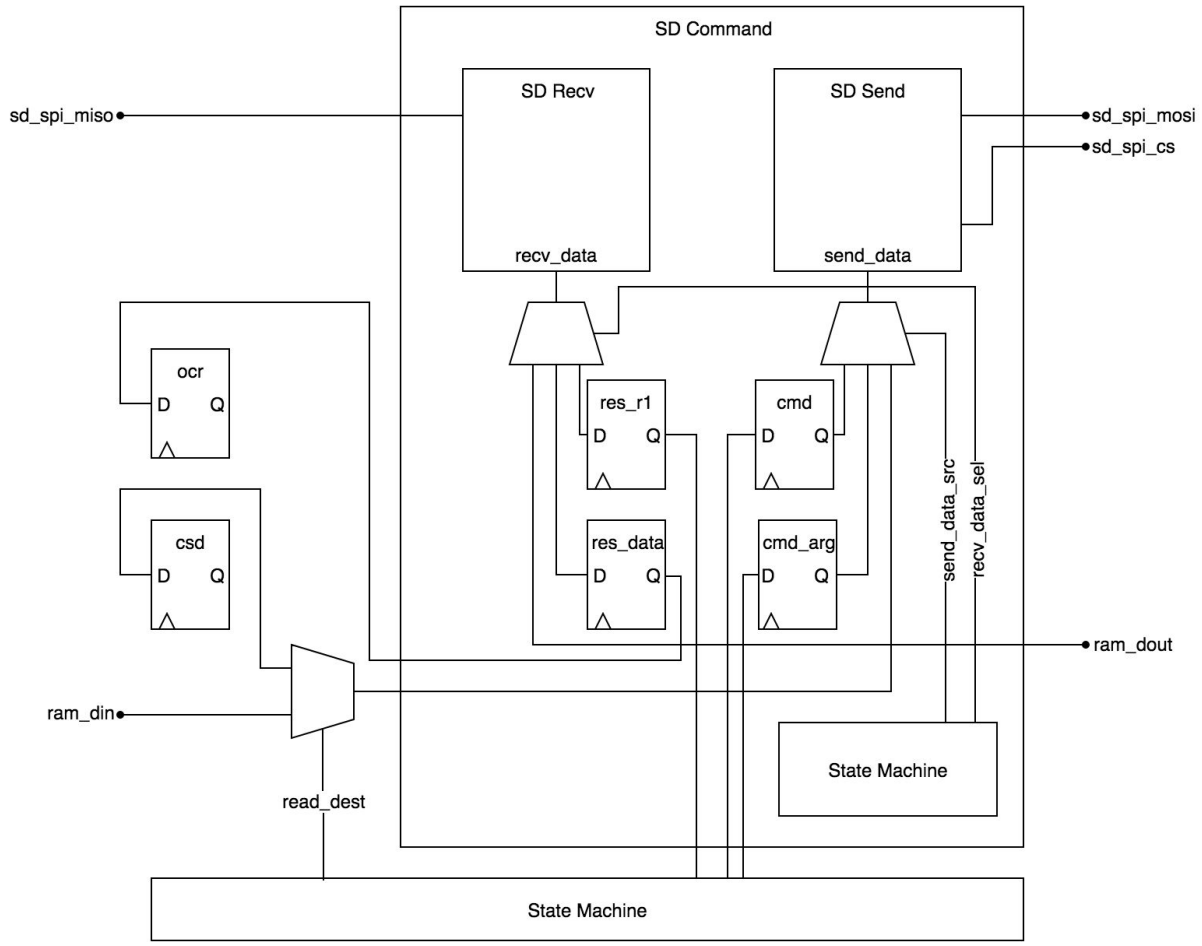


Figure B8. Simplified block diagram for SD driver and command controller. This diagram shows basic data flows, but is missing many of the complicated implementation details.

Appendix C - State diagrams[2]

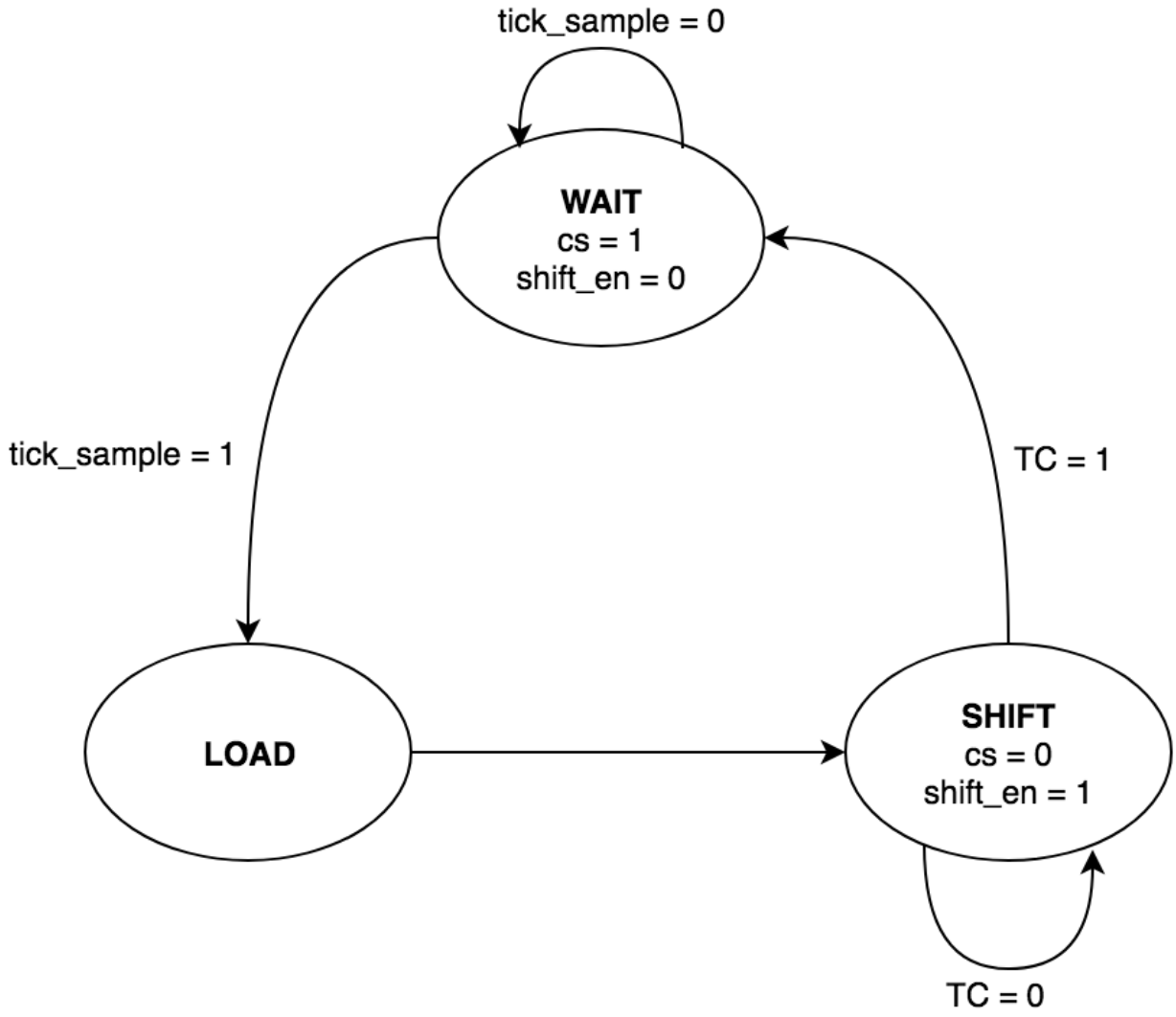


Figure C1. D/A driver state diagram

[2] To see enlarged versions of these diagrams, click on the image.

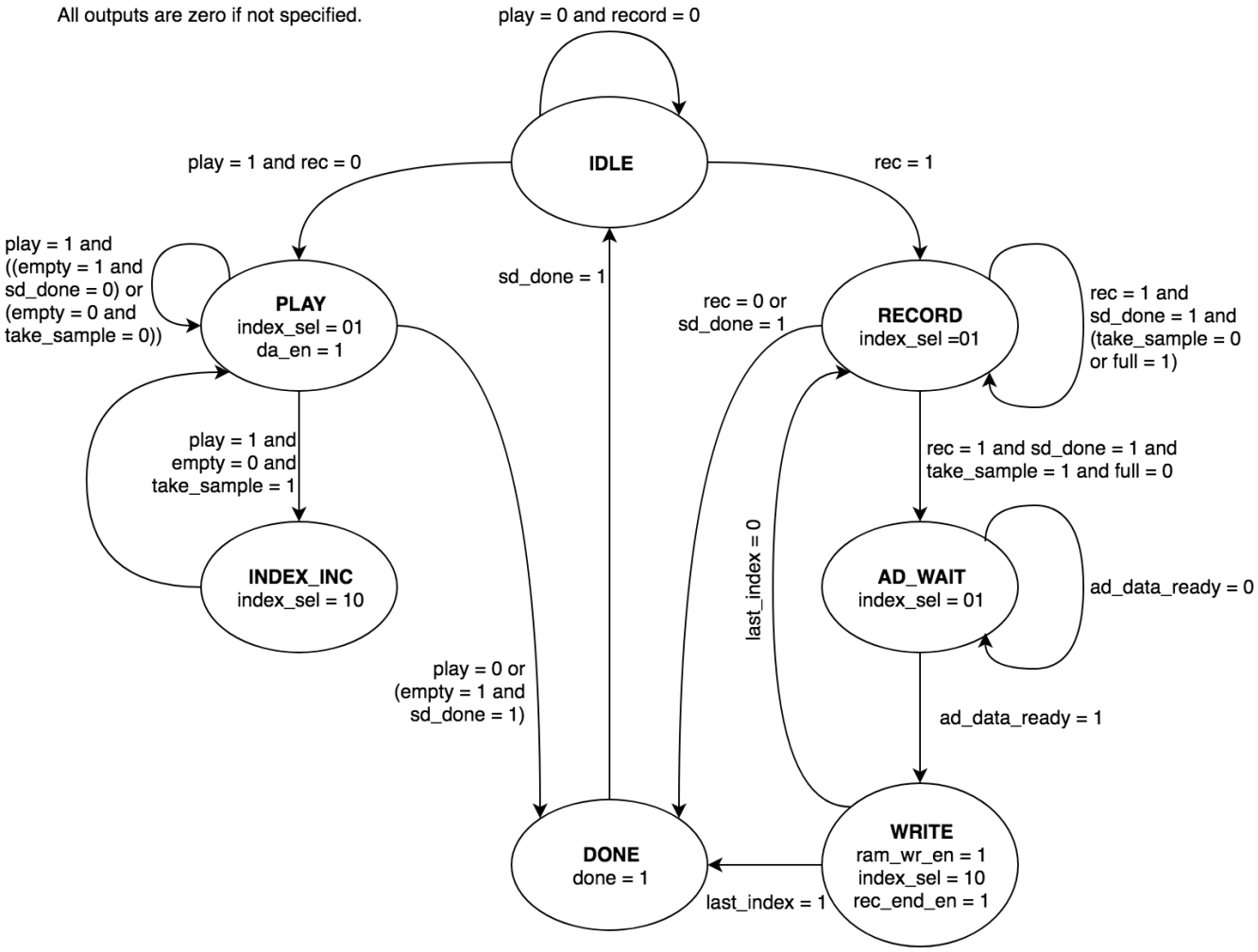


Figure C2. Audio controller state diagram

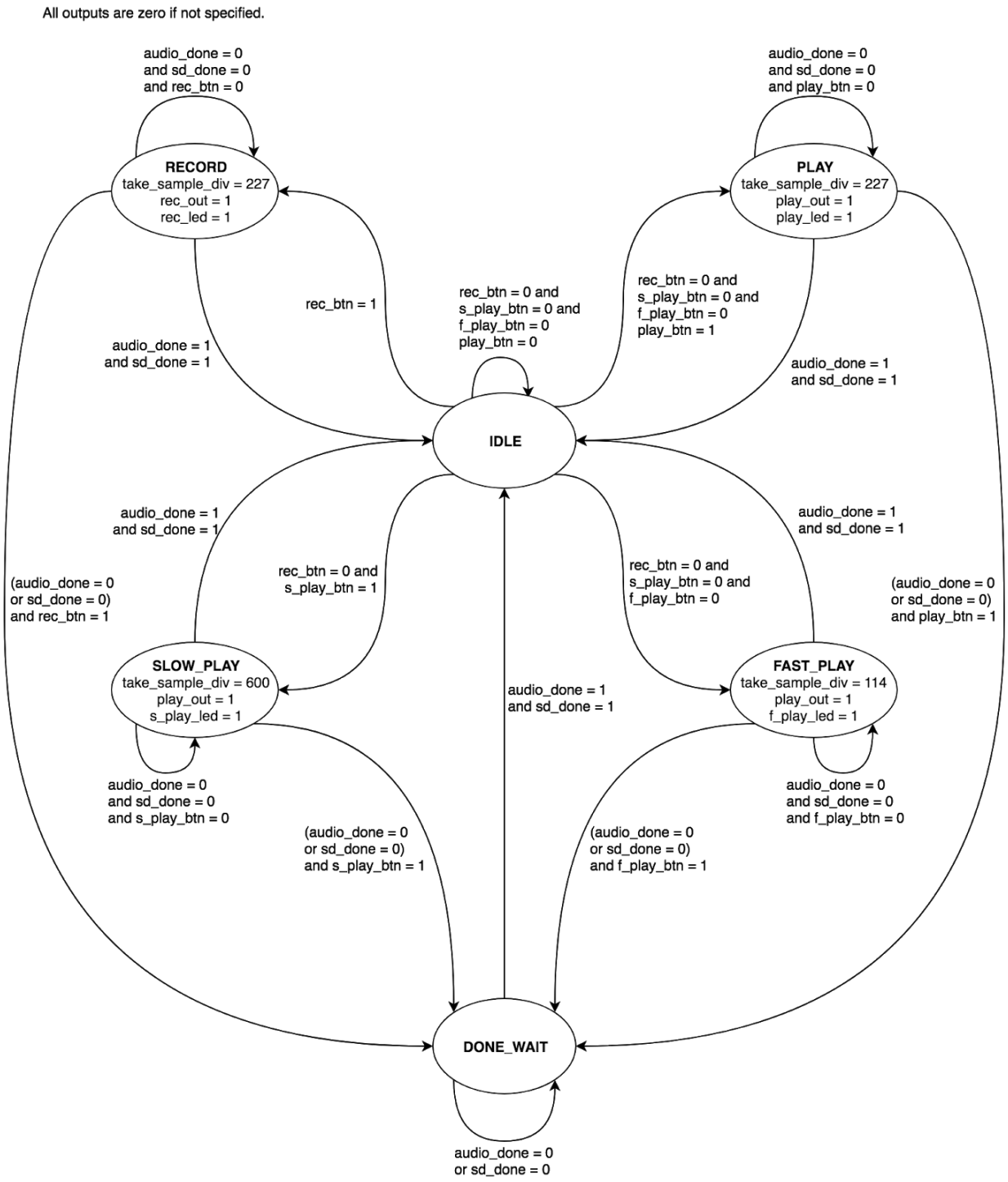


Figure C3. UI controller state diagram

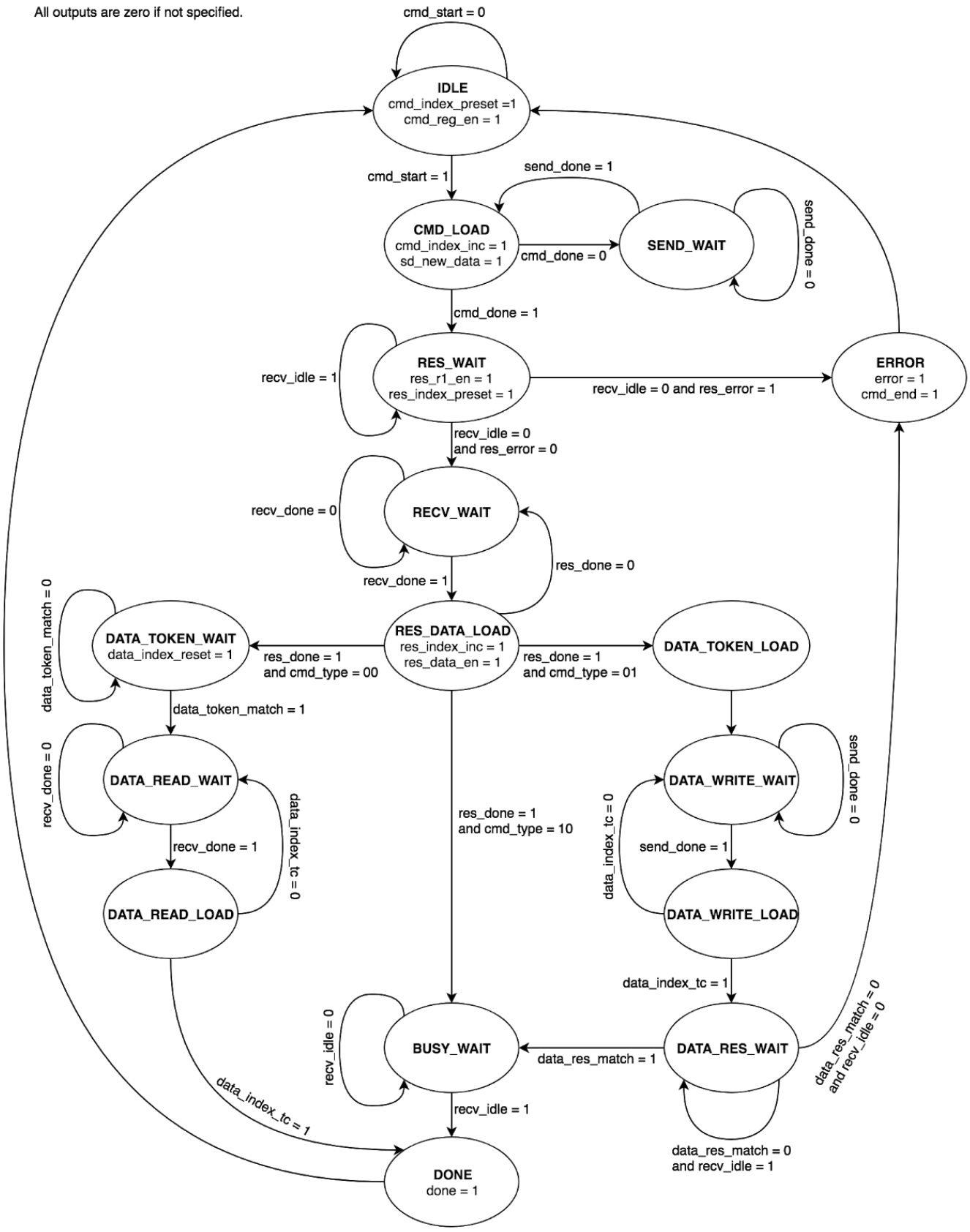


Figure C4. SD command controller state diagram

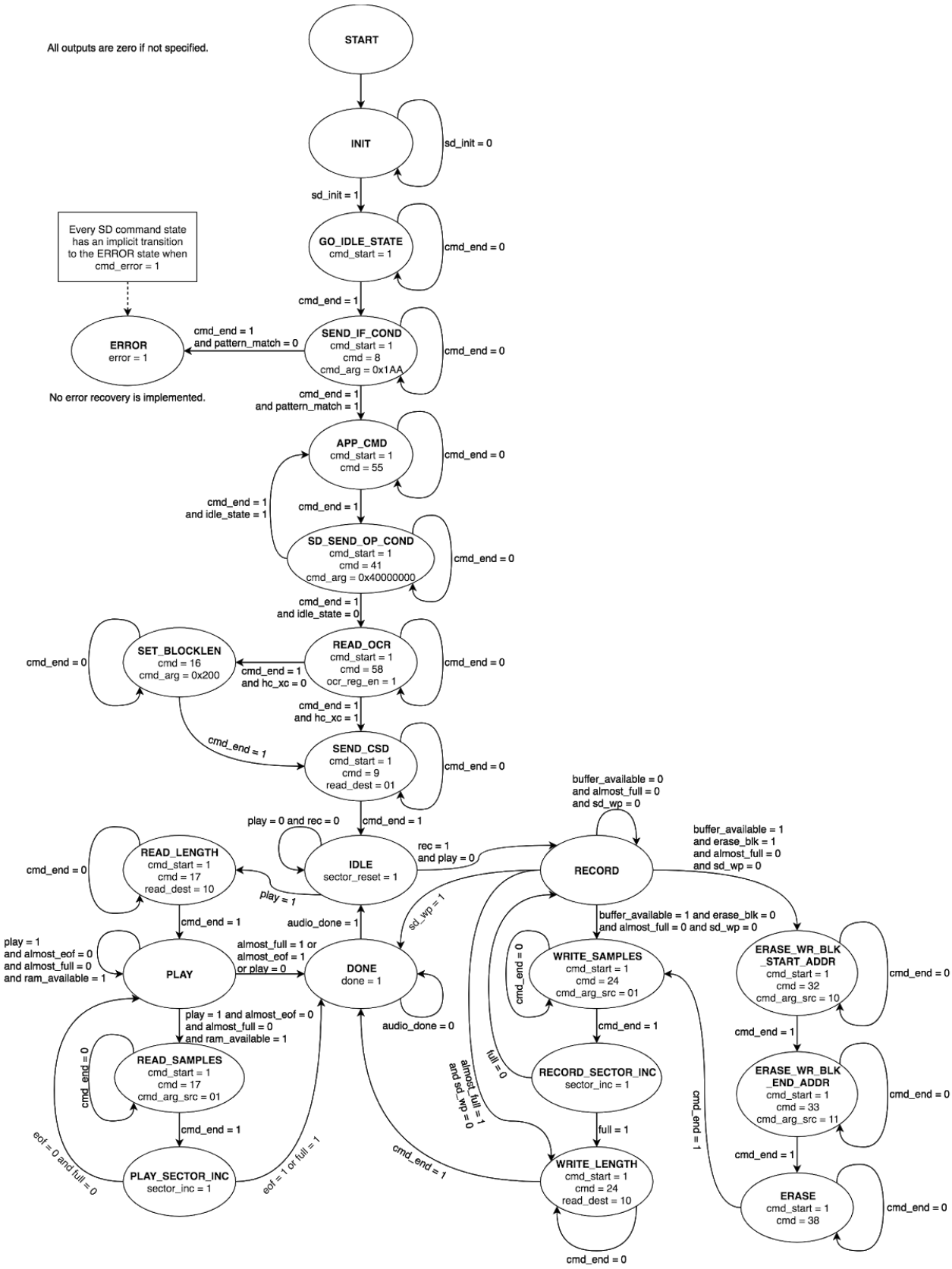


Figure C5. SD driver state diagram

Appendix D - Parts list

Quantity	Name	Description
1	Basys 3	Xilinx Artix-7 FPGA development board
1	Pmod MIC3	MEMS Microphone and 12-bit Analog-to-Digital convertor
1	Pmod DA2	2 channel 12-bit Digital-to-Analog converter
1	Pmod AMP2	Low power audio amplifier
1	Pmod SD	Full-sized SD Card Slot
1	Speaker	Speaker with male 3.5 mm jack

Appendix E - VHDL code and constraints

voice_recorder.vhd

```
----------------------------------------------------------------------------
-------------
-- Company: ENGS 31, 18X
-- Engineer: Ben Wolsieffer, Afia Semin
--
-- Create Date: 08/12/2018 08:28:03 PM
-- Design Name:
-- Module Name: voice_recorder - behavior
-- Project Name: VoiceRecorder
-- Target Devices: Basys 3
-- Tool Versions:
-- Description: Top level file for the voice recorder
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
-------------


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.math_real.all;

entity voice_recorder is
    port(mclk: in std_logic;

         f_play_btn: in std_logic;
         s_play_btn: in std_logic;
         play_btn: in std_logic;
```

```
        record_btn: in std_logic;
        f_play_led: out std_logic;
        s_play_led: out std_logic;
        play_led: out std_logic;
        record_led: out std_logic;

        -- Data visualization
        data_leds: out std_logic_vector(11 downto 0);

        ad_spi_sclk: out std_logic;
        ad_spi_sdata: in std_logic;
        ad_spi_cs: out std_logic;

        da_spi_sclk: out std_logic;
        da_spi_sdata: out std_logic;
        da_spi_cs: out std_logic;

        sd_spi_sclk: out std_logic;
        sd_spi_mosi: out std_logic;
        sd_spi_miso: in std_logic;
        sd_spi_cs: out std_logic;
        sd_wp: in std_logic;
        sd_cd: in std_logic);
end voice_recorder;

architecture behavior of voice_recorder is
    constant SCLK_DIVIDER: positive := 10; -- 10 MHz

    constant SAMPLE_BITS: positive := 12;
    -- number of bits in virtual address space used for counting
samples
    constant INDEX_BITS: positive := 32;
    -- number of RAM address bits (RAM must be configured so entire
address space is addressable)
    constant ADDR_BITS: positive := 17;

    component clock_divider is
        generic(divider: integer);
```

```
        port(mclk: in std_logic;
             dclk: out std_logic);
    end component;

    component down_counter is
        generic(bits: positive := 4);
        port(clk: in std_logic;
             k: in std_logic_vector(bits - 1 downto 0); -- preset
value
             CE: in std_logic := '1'; -- count enable
             preset: in std_logic := '0'; -- assert to set the
counter to k
             y: out std_logic_vector(bits - 1 downto 0); -- counter
output
             TC: out std_logic); -- terminal count
    end component;

    component pmod_ad1 is
        port(sclk: in std_logic;
             take_sample: in std_logic;
             ad_data: out std_logic_vector(11 downto 0) := (others =>
'0');
             ad_data_ready: out std_logic;

             spi_sclk: out std_logic;
             spi_cs: out std_logic;
             spi_sdata: in std_logic);
    end component;

    component pmod_da2
        port(da_data: in std_logic_vector (11 downto 0);
             tick_sample: in std_logic;
             sclk: in std_logic;
             spi_data: out std_logic;
             spi_sclk: out std_logic;
             spi_cs: out std_logic);
    end component;
```

```
    component UI_controller
        port(sclk: in std_logic;
             play_btn: in std_logic;
             f_play_btn: in std_logic;
             s_play_btn: in std_logic;
             audio_done: in std_logic;
             sd_done: in std_logic;
             rec_btn: in std_logic;
             play_out: out std_logic;
             play_led: out std_logic;
             f_play_led: out std_logic;
             s_play_led: out std_logic;
             rec_out: out std_logic;
             rec_led: out std_logic;
             tick_sample_div: out std_logic_vector(9 downto 0));
    end component;

    component sync is
        port(clk: in std_logic;
             input: in std_logic;
             output: out std_logic);
    end component;

    component audio_controller is
        generic(sample_bits: positive := 12; -- sample depth in RAM
                index_bits: positive := 32;
                addr_bits: positive := 17); -- width of RAM address
bus (must be fully addressable)
        port(clk: in std_logic;

             -- UI signals
             rec: in std_logic;
             play: in std_logic;
             sd_done: in std_logic;
             done: out std_logic;

             take_sample: in std_logic;
```

```
            -- A/D
            ad_data: in std_logic_vector(11 downto 0);
            ad_data_ready: in std_logic;
            ad_take_sample: out std_logic;

            -- D/A
            da_data: out std_logic_vector(11 downto 0);
            da_new_sample: out std_logic;

            -- Address mapping information
            index_audio: out std_logic_vector(index_bits - 1 downto
0); -- index of end of buffer/free space if recording or playing,
respectively (exclusive)
            index_sd: in std_logic_vector(index_bits - 1 downto 0);

            -- RAM
            ram_wr_en: out std_logic;
            ram_addr: out std_logic_vector(addr_bits - 1 downto 0);
            ram_din: in std_logic_vector(sample_bits - 1 downto 0);
            ram_dout: out std_logic_vector(sample_bits - 1 downto
0));
    end component;

    component sd_driver is
        generic(sample_bits: positive := 12; -- sample depth in RAM
                index_bits: positive := 40;
                addr_bits: positive := 17); -- width of RAM address
bus (must be fully addressable)
        port(sclk: in std_logic;
             rec: in std_logic;
             play: in std_logic;
             audio_done: in std_logic;
             done: out std_logic;
             error: out std_logic;

             index_audio: in std_logic_vector(index_bits - 1 downto
0); -- location of audio controller in recordin
             index_sd: out std_logic_vector(index_bits - 1 downto 0);
```

```
-- location of SD driver in recording

            -- SD card
            sd_spi_sclk: out std_logic;
            sd_spi_mosi: out std_logic;
            sd_spi_miso: in std_logic;
            sd_spi_cs: out std_logic;
            sd_wp: in std_logic;
            sd_cd: in std_logic;

            -- RAM
            ram_wr_en: out std_logic;
            ram_addr: out std_logic_vector(addr_bits - 1 downto 0);
            ram_din: in std_logic_vector(sample_bits - 1 downto 0);
            ram_dout: out std_logic_vector(sample_bits - 1 downto
0));
    end component;

    component audio_buffer is
      port(clka: in std_logic;
           wea: in std_logic_vector(0 downto 0);
           addra: in std_logic_vector(16 downto 0);
           dina: in std_logic_vector(11 downto 0);
           douta: out std_logic_vector(11 downto 0);
           clkb: in std_logic;
           web: in std_logic_vector(0 downto 0);
           addrb: in std_logic_vector(16 downto 0);
           dinb: in std_logic_vector(11 downto 0);
           doutb: out std_logic_vector(11 downto 0));
    end component;

    component button is
        generic(count: positive := 1000);
        port(clk: in std_logic;
             input: in std_logic;
             output: out std_logic);
    end component;
```

```
    -- clock used for SPI and all logic
    signal sclk: std_logic;

    -- monopulsed buttons
    signal s_play_btn_mp, f_play_btn_mp, play_btn_mp, record_btn_mp:
std_logic := '0';

    -- UI signals
    signal rec, play, audio_done, sd_done, eof, sd_error: std_logic
:= '0';

    signal take_sample: std_logic := '0';

    signal tick_sample_div: std_logic_vector(9 downto 0) := (others
=> '0');

    -- A/D signals
    signal ad_data: std_logic_vector(11 downto 0) := (others => '0');
    signal ad_data_ready, ad_take_sample: std_logic := '0';

    -- D/A signals
    signal da_data: std_logic_vector(11 downto 0);
    signal da_new_sample: std_logic;

    -- SD signals
    signal sd_wp_sync: std_logic;

    signal index_audio, index_sd: std_logic_vector(INDEX_BITS - 1
downto 0);

    -- Audio RAM signals
    signal audio_ram_wr_en: std_logic;
    signal audio_ram_wr_en_vec: std_logic_vector(0 downto 0);
    signal audio_ram_addr: std_logic_vector(ADDR_BITS - 1 downto 0)
:= (others => '0');
    signal audio_ram_din, audio_ram_dout:
std_logic_vector(SAMPLE_BITS - 1 downto 0) := (others => '0');
```

```
    -- SD RAM signals
    signal sd_ram_wr_en: std_logic;
    signal sd_ram_wr_en_vec: std_logic_vector(0 downto 0);
    signal sd_ram_addr: std_logic_vector(ADDR_BITS - 1 downto 0) :=
(others => '0');
    signal sd_ram_din, sd_ram_dout: std_logic_vector(SAMPLE_BITS - 1
downto 0) := (others => '0');

begin

    data_leds <= da_data;

    -- sclk generator
    sclk_generator: clock_divider
        generic map(divider => SCLK_DIVIDER)
        port map(mclk => mclk,
                 dclk => sclk);

    -- take_sample generator
    take_sample_counter: down_counter
        generic map(bits => 10)
        port map(clk => sclk,
                 k => tick_sample_div,
                 TC => take_sample);

    -- A/D
    ad: pmod_ad1
        port map(sclk => sclk,
                 take_sample => ad_take_sample,
                 ad_data => ad_data,
                 ad_data_ready => ad_data_ready,
                 spi_sclk => ad_spi_sclk,
                 spi_sdata => ad_spi_sdata,
                 spi_cs => ad_spi_cs);

    -- D/A
    da: pmod_da2
        port map(sclk => sclk,
```

```
                tick_sample => da_new_sample,
                da_data => da_data,
                spi_sclk => da_spi_sclk,
                spi_data => da_spi_sdata,
                spi_cs => da_spi_cs);

    -- play button debouncer
    play_btn_debounce: button
        port map(clk => sclk,
                input => play_btn,
                output => play_btn_mp);

    --fast play button debouncer
    f_play_btn_debounce: button
            port map(clk => sclk,
                    input => f_play_btn,
                    output => f_play_btn_mp);

    -- slow play button debouncer
   s_play_btn_debounce: button
       port map(clk => sclk,
                input => s_play_btn,
                output => s_play_btn_mp);

    -- record button debounce
    record_btn_debounce: button
        port map(clk => sclk,
                input => record_btn,
                output => record_btn_mp);

    -- UI controller
    ui_controller_map: UI_controller
        port map(sclk => sclk,
                play_btn => play_btn_mp,
                f_play_btn => f_play_btn_mp,
                s_play_btn => s_play_btn_mp,
                rec_btn => record_btn_mp,
                play_led => play_led,
```

```
            f_play_led => f_play_led,
            s_play_led => s_play_led,
            rec_led => record_led,
            audio_done => audio_done,
            sd_done => sd_done,
            play_out => play,
            rec_out => rec,
            tick_sample_div => tick_sample_div);

-- port map main audio controller
audio_controller_map: audio_controller
    generic map(sample_bits => SAMPLE_BITS,
                index_bits => INDEX_BITS,
                addr_bits => ADDR_BITS)
    port map(clk => sclk,
             rec => rec,
             play => play,
             sd_done => sd_done,
             done => audio_done,
             take_sample => take_sample,
             ad_data => ad_data,
             ad_data_ready => ad_data_ready,
             ad_take_sample => ad_take_sample,
             da_data => da_data,
             da_new_sample => da_new_sample,
             index_audio => index_audio,
             index_sd => index_sd,
             ram_wr_en => audio_ram_wr_en,
             ram_addr => audio_ram_addr,
             ram_din => audio_ram_din,
             ram_dout => audio_ram_dout);

sd_wp_sync_map: sync
    port map(clk => sclk,
             input => sd_wp,
             output => sd_wp_sync);

sd_driver_map: sd_driver
```

```
        generic map(sample_bits => SAMPLE_BITS,
                    index_bits => INDEX_BITS,
                    addr_bits => ADDR_BITS)
        port map(sclk => sclk,
                 rec => rec,
                 play => play,
                 audio_done => audio_done,
                 done => sd_done,
                 error => sd_error,
                 index_audio => index_audio,
                 index_sd => index_sd,
                 sd_spi_sclk => sd_spi_sclk,
                 sd_spi_mosi => sd_spi_mosi,
                 sd_spi_miso => sd_spi_miso,
                 sd_spi_cs => sd_spi_cs,
                 sd_wp => sd_wp_sync,
                 sd_cd => '1',
                 ram_wr_en => sd_ram_wr_en,
                 ram_addr => sd_ram_addr,
                 ram_din => sd_ram_din,
                 ram_dout => sd_ram_dout);

    -- block RAM audio buffer
    audio_ram_wr_en_vec(0) <= audio_ram_wr_en;
    sd_ram_wr_en_vec(0) <= sd_ram_wr_en;
    ram: audio_buffer
        port map(clka => sclk,
                 wea => audio_ram_wr_en_vec,
                 addra => audio_ram_addr,
                 dina => audio_ram_dout,
                 douta => audio_ram_din,
                 clkb => sclk,
                 web => sd_ram_wr_en_vec,
                 addrb => sd_ram_addr,
                 dinb => sd_ram_dout,
                 doutb => sd_ram_din);

end behavior;
```

pmod_ad1.vhd

```
----------------------------------------------------------------------------
-------------
-- Company: ENGS 31, 18X
-- Engineer: Ben Wolsieffer
--
-- Create Date: 07/20/2018 09:27:40 AM
-- Design Name:
-- Module Name: pmod_ad1 - behavior
-- Project Name: pmod_ad1
-- Target Devices: Artix 7 - Basys 3
-- Tool Versions:
-- Description: Driver for the Diligent Pmod AD1 (Analog Devices
AD7476A).
--
-- Dependencies: down_counter.vhd
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
-------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity pmod_ad1 is
    port(sclk: in std_logic;
         take_sample: in std_logic;
         ad_data: out std_logic_vector(11 downto 0) := (others =>
'0');
         ad_data_ready: out std_logic;

         spi_sclk: out std_logic;
         spi_cs: out std_logic;
```

```
         spi_sdata: in std_logic);
end pmod_ad1;

architecture behavior of pmod_ad1 is
    component down_counter is
        generic(bits: positive := 4);
        port (clk: in std_logic;
              k: in std_logic_vector(bits - 1 downto 0); -- preset
value
              CE: in std_logic := '1'; -- count enable
              preset: in std_logic := '0'; -- assert to set the
counter to k
              y: out std_logic_vector(bits - 1 downto 0); -- counter
output
              TC: out std_logic); -- terminal count
    end component;

    -- Controller
    type state_type is (st_wait, st_shift, st_load);
    signal state: state_type := st_wait;
    signal next_state: state_type := st_wait;

    -- Datapath
    signal shift_en: std_logic := '0';
    signal load_en: std_logic := '0';
    signal ser_data_reg: std_logic_vector(11 downto 0) := (others =>
'0');
    signal shift_tc: std_logic := '0';
    signal shift_preset: std_logic := '0';
begin
    -- Controller
    shift_counter: down_counter port map(
        clk => sclk,
        k => std_logic_vector(to_unsigned(14, 4)),
        preset => shift_preset,
        TC => shift_tc);

    output_proc: process(state) begin
```

```
    shift_en <= '0';
    load_en <= '0';
    shift_preset <= '0';
    spi_cs <= '1';
    ad_data_ready <= '0';

    case state is
        when st_wait => shift_preset <= '1';
        when st_shift =>
            shift_en <= '1';
            spi_cs <= '0';
        when st_load =>
            load_en <= '1';
            ad_data_ready <= '1';
    end case;
end process;

next_state_proc: process(state, take_sample, shift_tc) begin
    next_state <= state;
    case state is
        when st_wait =>
            if take_sample = '1' then
                next_state <= st_shift;
            end if;
        when st_shift =>
            if shift_tc = '1' then
                next_state <= st_load;
            end if;
        when st_load => next_state <= st_wait;
    end case;
end process;

state_update_proc: process(sclk) begin
    if rising_edge(sclk) then
        state <= next_state;
    end if;
end process;
```

```
    -- Datapath

    -- pass clock input to output
    spi_sclk <= sclk;

    shift_proc: process(sclk) begin
        if rising_edge(sclk) then
            if shift_en = '1' then
                -- shift SPI data into the LSB
                ser_data_reg <= ser_data_reg(ser_data_reg'high - 1
downto 0) & spi_sdata;
            end if;
        end if;
    end process;

    load_proc: process(sclk) begin
        if rising_edge(sclk) then
            if load_en = '1' then
                -- copy 12 least significant bits from serial
register to A/D
                -- data register
                ad_data <= ser_data_reg(ad_data'range);
            end if;
        end if;
    end process;
end behavior;
```

pmod_da2.vhd

```
----------------------------------------------------------------------
-------------
-- Company: ENGS 31, 18X
-- Engineer: Afia Semin
--
-- Create Date: 08/11/2018 07:54:21 PM
-- Design Name:
-- Module Name: pmod_da2 - behavior
-- Project Name: VoiceRecorder
```

```
-- Target Devices: Artix 7 - Basys 3
-- Tool Versions:
-- Description: Driver for the Diligent Pmod DA2 (Texas Instruments
DAC121S101).
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------
------------


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity pmod_da2 is
    port(da_data: in std_logic_vector (11 downto 0);
         tick_sample: in std_logic;
         sclk: in std_logic;
         spi_data: out std_logic;
         spi_sclk: out std_logic;
         spi_cs: out std_logic);
end pmod_da2;

architecture behavior of pmod_da2 is

    signal n_shifts: unsigned(3 downto 0):="1111";
    signal shift_en: std_logic;
    signal par_data_reg: std_logic_vector( 15 downto 0) := (others =>
'0');
    signal TC: std_logic;
    signal iCS: std_logic;
    type state is (waits, load, shift);
    signal curr_state, next_state: state;
```

```
begin
    spi_sclk <= sclk;
    spi_cs <= iCS;


    shift_count: process(sclk, n_shifts, iCS) begin
        if rising_edge(sclk) then
            if (n_shifts > 0) and (iCS = '0') then
                n_shifts <= n_shifts - 1;
            elsif n_shifts = "0000" then
                n_shifts <= "1111";
            end if;
        end if;

        if n_shifts = "0000" then
            TC <= '1';
        else
            TC <= '0';
        end if;
    end process shift_count;

    input_reg: process(sclk, tick_sample) begin
        if rising_edge(sclk) then
            if tick_sample = '1' then
                par_data_reg <= std_logic_vector(resize(
unsigned(da_data), 16));
            elsif shift_en <= '1' then
               spi_data <= par_data_reg(15);
               par_data_reg <= par_data_reg(14 downto 0) & "0";
            end if;
        end if;
    end process input_reg;

    state_update: process(sclk) begin
        if rising_edge(sclk) then
            curr_state <= next_state;
        end if;
```

```
    end process state_update;

    controller: process(curr_state, tick_sample, TC) begin
        iCS <= '1';
        shift_en <= '0';
        next_state <= curr_state;

        case curr_state is
            when waits =>
                iCS <= '1';
                if tick_sample = '1' then
                    next_state <= load;
                end if;

            when load =>
                next_state <= shift;

           when shift =>
                iCS <= '0';
                shift_en <= '1';
                if TC = '1' then
                    next_state <= waits;
                end if;
        end case;
    end process controller;

end behavior;
```

audio_controller.vhd

```
----------------------------------------------------------------------------
-------------
-- Company: ENGS 31, 18X
-- Engineer: Ben Wolsieffer
--
-- Create Date: 08/11/2018 07:12:17 PM
-- Design Name:
-- Module Name: audio_controller - behavior
```

```
-- Project Name: VoiceREcorder
-- Target Devices: Artix 7 - Basys 3
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.math_real.all;

entity audio_controller is
    generic(sample_bits: positive := 12; -- sample depth in RAM
            index_bits: positive := 40;
            addr_bits: positive := 17); -- width of RAM address bus
(must be fully addressable)
    port(clk: in std_logic;

         -- UI signals
         rec: in std_logic;
         play: in std_logic;
         sd_done: in std_logic;
         done: out std_logic;

         take_sample: in std_logic;

         -- A/D
         ad_data: in std_logic_vector(11 downto 0);
         ad_data_ready: in std_logic;
         ad_take_sample: out std_logic;
```

```
        -- D/A
        da_data: out std_logic_vector(11 downto 0);
        da_new_sample: out std_logic;

        -- Address mapping information
        index_audio: out std_logic_vector(index_bits - 1 downto 0);
-- index of end of buffer/free space if recording or playing,
respectively (exclusive)
        index_sd: in std_logic_vector(index_bits - 1 downto 0);

        -- RAM
        ram_wr_en: out std_logic;
        ram_addr: out std_logic_vector(addr_bits - 1 downto 0);
        ram_din: in std_logic_vector(sample_bits - 1 downto 0);
        ram_dout: out std_logic_vector(sample_bits - 1 downto 0));
end audio_controller;

architecture behavior of audio_controller is
    constant TAKE_SAMPLE_DIVIDER: integer := 20;
    constant TAKE_SAMPLE_BITS: integer :=
integer(ceil(log2(real(TAKE_SAMPLE_DIVIDER))));

    constant RAM_SIZE: positive := 2 ** addr_bits;

    -- Maximum possible index
    constant INDEX_MAX: unsigned(index_bits - 1 downto 0) := (others
=> '1');

    component down_counter is
        generic(bits: positive := 4);
        port (clk: in std_logic;
              k: in std_logic_vector(bits - 1 downto 0); -- preset
value
              CE: in std_logic := '1'; -- count enable
              preset: in std_logic := '0'; -- assert to set the
counter to k
              y: out std_logic_vector(bits - 1 downto 0); -- counter
```

```
output
              TC: out std_logic); -- terminal count
    end component;

    type state_type is (st_idle, st_record, st_ad_wait, st_write,
st_play, st_index_inc, st_done);
    type index_sel_type is (hold, increment, reset);

    signal state: state_type := st_idle;
    signal next_state: state_type;

    signal index_sel: index_sel_type;
    signal rec_end_en: std_logic;
    signal da_en: std_logic;

    signal index_reg: unsigned(INDEX_MAX'range) := (others => '0');
begin

    ad_take_sample <= take_sample;
    da_new_sample <= take_sample when da_en = '1' else '0';

    -- Take index modulo the RAM size, creating a circular buffer
    ram_addr <= std_logic_vector(index_reg(ram_addr'range));
    ram_dout <= ad_data;
    da_data <= ram_din;

    -- Update address register
    index_reg_proc: process(clk) begin
        if rising_edge(clk) then
            case index_sel is
                when hold => null;
                when increment => index_reg <= index_reg + 1;
                when reset => index_reg <= (others => '0');
            end case;
        end if;
    end process;

    index_audio <= std_logic_vector(index_reg);
```

```
    next_state_proc: process(state, index_reg, play, rec,
take_sample, ad_data_ready, sd_done, index_sd)
        variable index_sd_end: unsigned(index_sd'range); -- end of
free RAM space while recording
    begin
        -- Calculate end of free space (only used when recording)
        if unsigned(index_sd) > INDEX_MAX - RAM_SIZE then
            index_sd_end := INDEX_MAX;
        else
            index_sd_end := unsigned(index_sd) + RAM_SIZE;
        end if;
        next_state <= state;

        case state is
            when st_idle =>
                if rec = '1' then
                    next_state <= st_record;
                elsif play = '1' then
                    next_state <= st_play;
                end if;
            -- Record
            when st_record =>
                if rec = '0' or sd_done = '1' then
                    next_state <= st_done;
                elsif take_sample = '1' and index_reg < index_sd_end
then
                    -- only record when there is free space
                    next_state <= st_ad_wait;
                end if;
            when st_ad_wait =>
                if ad_data_ready = '1' then
                    next_state <= st_write;
                end if;
            when st_write =>
                if index_reg = INDEX_MAX then
                    -- If we run out of indices, stop.
                    -- Must be checked after sample has been taken,
```

```
but before
                    -- an overflow can occur
                    next_state <= st_done;
                else
                    next_state <= st_record;
                end if;
            -- Play
            when st_play =>
                if play = '0' then
                    next_state <= st_done;
                elsif index_reg + 1 < unsigned(index_sd) then
                    -- only play when there are samples in the buffer
                    if take_sample = '1' then
                        next_state <= st_index_inc;
                    end if;
                elsif sd_done = '1' then
                    -- if the sd card has finished buffering and we
reach
                    -- the end of the buffer, we are done
                    next_state <= st_done;
                end if;
            when st_index_inc => next_state <= st_play;
            when st_done =>
                if sd_done = '1' then
                    next_state <= st_idle;
                end if;
        end case;
    end process;

    output_proc: process(state) begin
        index_sel <= reset;
        rec_end_en <= '0';
        da_en <= '0';

        ram_wr_en <= '0';
        done <= '0';

        case state is
```

```
            when st_idle => null;
            when st_record =>
                index_sel <= hold;
            when st_ad_wait =>
                index_sel <= hold;
            when st_write =>
                ram_wr_en <= '1';
                index_sel <= increment;
                rec_end_en <= '1';
            when st_play =>
                index_sel <= hold;
                da_en <= '1';
            when st_index_inc =>
                index_sel <= increment;
            when st_done =>
                done <= '1';
        end case;
    end process;

    state_update_proc: process(clk) begin
        if rising_edge(clk) then
            state <= next_state;
        end if;
    end process;

end behavior;
```

UI_controller.vhd

```
----------------------------------------------------------------------------
-------------
-- Company: ENGS 31, 18X
-- Engineer: Afia Semin
--
-- Create Date: 08/12/2018 07:35:23 PM
-- Design Name:
-- Module Name: UI_controller - Behavioral
-- Project Name: VoiceRecorder
```

```
-- Target Devices: Artix 7 - Basys 3
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
-------------


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.math_real.all;

entity UI_controller is
    port(sclk: in std_logic;
         play_btn: in std_logic;
         f_play_btn: in std_logic;
         s_play_btn: in std_logic;
         audio_done: in std_logic;
         sd_done: in std_logic;
         rec_btn: in std_logic;
         play_out: out std_logic;
         play_led: out std_logic;
         f_play_led: out std_logic;
         s_play_led: out std_logic;
         rec_out: out std_logic;
         rec_led: out std_logic;
         tick_sample_div: out std_logic_vector(9 downto 0));
end UI_controller;

architecture Behavioral of UI_controller is
```

```
    constant TAKE_SAMPLE_BITS: integer := 10;

    constant TAKE_SAMPLE_DIVIDER: integer :=  227; -- 44.052 kHz
    constant F_TAKE_SAMPLE_DIVIDER: integer := 114; -- 87.719 kHz
    constant S_TAKE_SAMPLE_DIVIDER: integer := 600; -- 16.667 kHz

    type state is (idle, play, fast_play, slow_play, rec, done_wait);
    signal curr_state, next_state: state;

begin

    controller: process(curr_state, f_play_btn, s_play_btn, play_btn,
rec_btn, audio_done, sd_done) begin
        play_out <= '0';
        play_led <= '0';
        f_play_led <= '0';
        s_play_led <= '0';
        rec_out <= '0';
        rec_led <= '0';
        tick_sample_div <=
std_logic_vector(to_unsigned(TAKE_SAMPLE_DIVIDER, TAKE_SAMPLE_BITS));
        next_state <= curr_state;

        case curr_state is
            when idle =>
                if play_btn = '1' then
                    next_state <= play;
                end if;

                if f_play_btn = '1' then
                    next_state <= fast_play;
                end if;

                if s_play_btn = '1' then
                    next_state <= slow_play;
                end if;

                if rec_btn = '1' then
```

```
                next_state <= rec;
            end if;

        when play =>
            play_out <= '1';
            play_led <= '1';
            if audio_done = '1' and sd_done = '1' then
                next_state <= idle;
            elsif play_btn = '1' then
                next_state <= done_wait;
            end if;

        when fast_play =>
            play_out <= '1';
            f_play_led <= '1';
            tick_sample_div <=
std_logic_vector(to_unsigned(F_TAKE_SAMPLE_DIVIDER,
TAKE_SAMPLE_BITS));
            if audio_done = '1' and sd_done = '1' then
                next_state <= idle;
            elsif f_play_btn = '1' then
                next_state <= done_wait;
            end if;

        when slow_play =>
            play_out <= '1';
            s_play_led <= '1';
            tick_sample_div <=
std_logic_vector(to_unsigned(S_TAKE_SAMPLE_DIVIDER,
TAKE_SAMPLE_BITS));
            if audio_done = '1' and sd_done = '1' then
                next_state <= idle;
            elsif s_play_btn = '1' then
                next_state <= done_wait;
            end if;

        when rec =>
            rec_out <= '1';
```

```
                rec_led <= '1';
                if audio_done = '1' and sd_done = '1' then
                    next_state <= idle;
                elsif rec_btn = '1' then
                    next_state <= done_wait;
                end if;

            when done_wait =>
                if audio_done = '1' and sd_done = '1' then
                    next_state <= idle;
                end if;
        end case;
    end process;

    state_update: process(sclk) begin
        if rising_edge(sclk) then
            curr_state <= next_state;
        end if;
    end process state_update;

end Behavioral;
```

sd_driver.vhd

```
----------------------------------------------------------------------------
-------------
-- Company: ENGS 31, 18X
-- Engineer: Ben Wolsieffer
--
-- Create Date: 08/14/2018 04:27:27 PM
-- Design Name:
-- Module Name: sd_driver - behavior
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies: sd_cmd.vhd
```

```
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.math_real.all;

entity sd_driver is
    generic(sample_bits: positive := 12; -- sample depth in RAM
            index_bits: positive := 40;
            addr_bits: positive := 17); -- width of RAM address bus
(must be fully addressable)
    port(sclk: in std_logic;
         rec: in std_logic;
         play: in std_logic;
         audio_done: in std_logic;
         done: out std_logic;
         error: out std_logic;

         index_audio: in std_logic_vector(index_bits - 1 downto 0);
-- location of audio controller in recordin
         index_sd: out std_logic_vector(index_bits - 1 downto 0); --
location of SD driver in recording

         -- SD card
         sd_spi_sclk: out std_logic;
         sd_spi_mosi: out std_logic;
         sd_spi_miso: in std_logic;
         sd_spi_cs: out std_logic;
         sd_wp: in std_logic;
         sd_cd: in std_logic;
```

```
        -- RAM
        ram_wr_en: out std_logic;
        ram_addr: out std_logic_vector(addr_bits - 1 downto 0);
        ram_din: in std_logic_vector(sample_bits - 1 downto 0);
        ram_dout: out std_logic_vector(sample_bits - 1 downto 0));
end sd_driver;

architecture behavior of sd_driver is

    constant SECTOR_SIZE: positive := 512;

    -- don't start at beginning of device
    constant SECTOR_OFFSET: natural := 8192;
    constant ADDR_OFFSET: natural := SECTOR_OFFSET * SECTOR_SIZE;

    -- hardcoded erase block size
    constant ERASE_SECTORS: positive := 8192;

    constant BYTES_PER_INDEX: positive := 2;
    constant INDICES_PER_SECTOR: positive := SECTOR_SIZE /
BYTES_PER_INDEX;

    constant RAM_SIZE: positive := 2 ** addr_bits;

    -- Maximum possible index
    constant INDEX_MAX: unsigned(index_bits - 1 downto 0) := (others
=> '1');

    component down_counter is
        generic(bits: positive := 4);
        port(clk: in std_logic;
             k: in std_logic_vector(bits - 1 downto 0); -- preset
value
             CE: in std_logic := '1'; -- count enable
             preset: in std_logic := '0'; -- assert to set the
counter to k
             y: out std_logic_vector(bits - 1 downto 0); -- counter
```

```
output
            TC: out std_logic); -- terminal count
    end component;

    component sd_cmd is
        port(sclk: in std_logic;
             cmd: in std_logic_vector(5 downto 0);
             cmd_arg: in std_logic_vector(31 downto 0);
             cmd_start: in std_logic; -- assert to start command
             cmd_end: out std_logic;
             res_r1: out std_logic_vector(7 downto 0) := x"00"; -- r1
response (first byte of response)
             res_data: out std_logic_vector(31 downto 0) :=
x"00000000"; -- response data/card status
             data_index: out std_logic_vector(8 downto 0);
             data_in: in std_logic_vector(7 downto 0);
             data_out: out std_logic_vector(7 downto 0);
             data_out_en: out std_logic;
             error: out std_logic;
             sd_spi_sclk: out std_logic;
             sd_spi_mosi: out std_logic;
             sd_spi_miso: in std_logic;
             sd_spi_cs: out std_logic);
    end component;

    type state_type is (start, init, go_idle_state, send_if_cond,
app_cmd, sd_send_op_cond, read_ocr, set_blocklen, send_csd,
                        idle, st_done, st_error,
                        st_play, read_length, read_samples,
play_sector_inc,
                        st_record, write_samples, record_sector_inc,
erase_wr_blk_start_addr, erase_wr_blk_end_addr, erase, write_length);
    signal state: state_type := start;
    signal next_state: state_type;

    signal sd_init: std_logic;

    -- Command type and argument registers
```

```
    signal cmd: std_logic_vector(5 downto 0);
    signal cmd_arg: std_logic_vector(31 downto 0);
    signal cmd_start, cmd_end: std_logic;

    -- Response data
    signal res_r1: std_logic_vector(7 downto 0);
    signal res_data: std_logic_vector(31 downto 0);

    -- Data block
    signal data_index: std_logic_vector(8 downto 0);
    signal data_in, data_out: std_logic_vector(7 downto 0);
    signal data_out_en: std_logic;

    type read_dest_type is (dest_length, dest_ram, dest_csd);
    signal read_dest: read_dest_type;

    signal cmd_error: std_logic;

    -- card information


    -- card specific data (CSD)
    subtype CSD_STRUCTURE is natural range 127 downto 126;
    subtype CSD_V1_READ_BL_LEN is natural range 83 downto 80;
    subtype CSD_V1_C_SIZE is natural range 73 downto 62;
    subtype CSD_V1_C_SIZE_MULT is natural range 49 downto 47;
    subtype CSD_V2_C_SIZE is natural range 69 downto 48;
    signal csd_reg: std_logic_vector(127 downto 0) := (others =>
'0');

    -- maximum sector index (from CSD)
    signal card_sector_max: unsigned(31 downto 0);

    -- operation conditions register (indicates that card is SDHC or
SDXC)
    constant OCR_CCS: natural := 30;
    signal ocr_reg: std_logic_vector(31 downto 0);
    signal ocr_reg_en: std_logic;
```

```
    -- Audio information

    -- current sector
    signal sector: unsigned(31 downto 0) := (others => '0');
    signal addr: unsigned(40 downto 0);
    signal sector_reset, sector_inc: std_logic;

    signal sector_multiplier: unsigned(9 downto 0);

    -- index: logical index in the audio file
    signal index: unsigned(index_bits - 1 downto 0);
    -- maximum logical index of recorded data
    -- stored in first bytes of card (little endian)
    signal index_end: unsigned(index_bits - 1 downto 0) := (others =>
'0');

begin

    init_counter: down_counter
        generic map(bits => 8)
        port map(clk => sclk,
                 k => x"64",
                 TC => sd_init);

    sd_cmd_map: sd_cmd
        port map(sclk => sclk,
                 cmd => cmd,
                 cmd_arg => cmd_arg,
                 cmd_start => cmd_start,
                 cmd_end => cmd_end,
                 res_r1 => res_r1,
                 res_data => res_data,
                 data_index => data_index,
                 data_in => data_in,
                 data_out => data_out,
                 data_out_en => data_out_en,
                 error => cmd_error,
```

```
                sd_spi_sclk => sd_spi_sclk,
                sd_spi_mosi => sd_spi_mosi,
                sd_spi_miso => sd_spi_miso,
                sd_spi_cs => sd_spi_cs);

    ocr_proc: process(sclk) begin
        if rising_edge(sclk) then
            if ocr_reg_en = '1' then
                ocr_reg <= res_data;
            end if;
        end if;
    end process;

    read_dest_proc: process(sclk, index, index_end, data_index,
read_dest, ram_din, data_out, data_out_en)
        variable bit_low: natural;
        variable unwrapped_index: unsigned(index'range);
    begin
        bit_low := to_integer(unsigned(data_index)) * 8;

        ram_wr_en <= '0';
        ram_dout <= (others => '0');
        data_in <= (others => '0');

        -- Every 12 bit sample is stored in two bytes on the SD card
        unwrapped_index := index + unsigned(data_index) /
BYTES_PER_INDEX;
        ram_addr <=
std_logic_vector(unwrapped_index(ram_addr'range));

        case read_dest is
            when dest_ram =>
                ram_wr_en <= data_out_en;
                if data_index(0) = '0' then -- even
                    data_in <= ram_din(3 downto 0) & "0000";
                    ram_dout <= "00000000" & data_out(7 downto 4);
                else -- odd
                    data_in <= ram_din(11 downto 4);
```

```
                    ram_dout <= data_out & ram_din(3 downto 0);
                end if;
            when dest_length =>
                if unsigned(data_index) < 4 then
                    data_in <= std_logic_vector(index(bit_low + 7
downto bit_low));
                end if;
            when others => null;
        end case;

        if rising_edge(sclk) then
            if data_out_en = '1' then
                case read_dest is
                    when dest_length =>
                        if unsigned(data_index) < 4 then
                            index_end(bit_low + 7 downto bit_low) <=
unsigned(data_out);
                        end if;
                    when dest_csd => csd_reg(bit_low + 7 downto
bit_low) <= data_out;
                    when others => null;
                end case;
            end if;
        end if;
    end process;

    card_size_proc: process(csd_reg) begin
        if csd_reg(CSD_STRUCTURE) = "00" then
            -- SD v1.xx or MMC
            card_sector_max <=
(resize(unsigned(csd_reg(CSD_V1_C_SIZE)), card_sector_max'length) +
1) sll
                (to_integer(unsigned(csd_reg(CSD_V1_C_SIZE_MULT))) +
2 + to_integer(unsigned(csd_reg(CSD_V1_READ_BL_LEN))) / SECTOR_SIZE);
        else
            -- SD >=v2.00
            card_sector_max <=
resize(unsigned(csd_reg(CSD_V2_C_SIZE)) * 1024 + 1023,
```

```
card_sector_max'length);
        end if;
    end process;

    sector_proc: process(sclk) begin
        if rising_edge(sclk) then
            if sector_reset = '1' then
                sector <= to_unsigned(SECTOR_OFFSET, sector'length);
            elsif sector_inc = '1' then
                sector <= sector + 1;
            end if;
        end if;
    end process;

    sector_multiplier <= "0000000001" when ocr_reg(OCR_CCS) = '1'
else "1000000000";

    index <= resize((sector - SECTOR_OFFSET) * INDICES_PER_SECTOR,
index'length);
    addr <= resize(sector * SECTOR_SIZE, addr'length);

    index_sd <= std_logic_vector(index);

    next_state_proc: process(state, sd_init, play, rec, cmd_end,
cmd_error, res_r1, res_data, index_audio, index, index_end, sector,
card_sector_max, audio_done, sd_wp)
        variable index_audio_end: unsigned(index_sd'range); -- end of
free RAM space during playback
    begin
        -- Calculate end of free space (only used when playing)
        if unsigned(index_audio) > INDEX_MAX - RAM_SIZE then
            index_audio_end := INDEX_MAX;
        else
            index_audio_end := unsigned(index_audio) + RAM_SIZE;
        end if;
        next_state <= state;

        case state is
```

```
            when start => next_state <= init;
            when init =>
                if sd_init = '1' then
                    next_state <= go_idle_state;
                end if;
            when idle =>
                if play = '1' then
                    next_state <= read_length;
                elsif rec = '1' then
                    next_state <= st_record;
                end if;
            when st_play =>
                -- the index conditions are also checked in
play_sector_inc.
                -- They are checked here in the unlikely case that
they are
                -- violated before the recording even begins (very
short
                -- recording, card smaller than the offset), but they
must also
                -- be checked before the increment occurs to prevent
wrapping
                -- with the largest cards and recordings.
                if play = '0' or index + (INDICES_PER_SECTOR - 1) >
index_end or sector > card_sector_max then
                    next_state <= st_done;
                elsif index_audio_end > index + (INDICES_PER_SECTOR -
1) then
                    next_state <= read_samples;
                end if;
            when play_sector_inc =>
                -- handle reaching the end of the recording, running
out of SD
                -- card space, or reaching the maximum index. These
must be
                -- handled before they occur to avoid a possible
overflow, but
                -- after the samples for the sector have been read.
```

```
                if index + (INDICES_PER_SECTOR - 1) >= index_end or
sector >= card_sector_max then
                    next_state <= st_done;
                else
                    next_state <= st_play;
                end if;
            when st_record =>
                -- Like playback, we should handle the boundary
conditions both
                -- here and in record_sector_inc
                if sd_wp = '1' then
                    next_state <= st_done;
                elsif sector > card_sector_max then
                    next_state <= write_length;
                elsif unsigned(index_audio) > index +
(INDICES_PER_SECTOR - 1) then
                    if (sector mod ERASE_SECTORS) = 0 then
                        next_state <= erase_wr_blk_start_addr;
                    else
                        next_state <= write_samples;
                    end if;
                elsif audio_done = '1' then
                    next_state <= write_length;
                end if;
            when record_sector_inc =>
                if sector >= card_sector_max then
                    next_state <= write_length;
                else
                    next_state <= st_record;
                end if;
            when st_done =>
                if audio_done = '1' then
                    next_state <= idle;
                end if;
            when st_error => null;
            when others =>
                if cmd_end = '1' then
                    case state is
```

```
                        when go_idle_state =>
                            if res_r1 = x"01" then
                                next_state <= send_if_cond;
                            end if;
                        when send_if_cond =>
                            if res_data(11 downto 0) = x"1AA" then
                                next_state <= app_cmd;
                            else
                                next_state <= st_error;
                            end if;
                        when app_cmd =>
                            next_state <= sd_send_op_cond;
                        when sd_send_op_cond =>
                            if res_r1 = x"01" then
                                -- retry ACMD41
                                next_state <= app_cmd;
                            else
                                next_state <= read_ocr;
                            end if;
                        when read_ocr =>
                            if res_data(OCR_CCS) = '1' then
                                next_state <= send_csd;
                            else
                                next_state <= set_blocklen;
                            end if;
                        when set_blocklen => next_state <= send_csd;
                        when send_csd => next_state <= idle;
                        when read_length => next_state <= st_play;
                        when read_samples => next_state <=
play_sector_inc;
                        when write_samples => next_state <=
record_sector_inc;
                        when erase_wr_blk_start_addr => next_state <=
erase_wr_blk_end_addr;
                        when erase_wr_blk_end_addr => next_state <=
erase;
                        when erase => next_state <= write_samples;
                        when write_length => next_state <= st_done;
```

```
                    when others => null;
                end case;
            end if;
    end case;

    -- error catching
    if cmd_error = '1' then
        next_state <= st_error;
    end if;
end process;

output_proc: process(state, sector, sector_multiplier) begin
    done <= '0';
    error <= '0';
    sector_reset <= '0';
    sector_inc <= '0';
    ocr_reg_en <= '0';
    cmd_start <= '0';
    cmd <= "000000";
    cmd_arg <= x"00000000";
    read_dest <= dest_ram;

    case state is
        when start => null;
        when init => null;
        when idle => sector_reset <= '1';
        when st_play => null;
        when play_sector_inc => sector_inc <= '1';
        when st_record => null;
        when record_sector_inc => sector_inc <= '1';
        when st_done => done <= '1';
        when st_error => error <= '1';
        when others =>
            cmd_start <= '1';
            case state is
                when go_idle_state => null;
                when send_if_cond =>
                    cmd <= "001000";
```

```
                    cmd_arg <= x"000001AA";
                when app_cmd =>
                    cmd <= "110111";
                when sd_send_op_cond =>
                    cmd <= "101001";
                    cmd_arg <= x"40000000";
                when read_ocr =>
                    cmd <= "111010";
                    ocr_reg_en <= '1';
                when set_blocklen =>
                    cmd <= "010000";
                    cmd_arg <= x"00000200";
                when send_csd =>
                    cmd <= "001001";
                    read_dest <= dest_csd;
                when read_length =>
                    cmd <= "010001";
                    read_dest <= dest_length;
                when read_samples =>
                    cmd <= "010001";
                    cmd_arg <= std_logic_vector(resize(sector *
sector_multiplier, cmd_arg'length));
                    read_dest <= dest_ram;
                when write_samples =>
                    cmd <= "011000";
                    cmd_arg <= std_logic_vector(resize(sector *
sector_multiplier, cmd_arg'length));
                    read_dest <= dest_ram;
                when erase_wr_blk_start_addr =>
                    cmd <= "100000";
                    cmd_arg <= std_logic_vector(resize((sector +
ERASE_SECTORS) * sector_multiplier + ERASE_SECTORS, cmd_arg'length));
                when erase_wr_blk_end_addr =>
                    cmd <= "100001";
                    cmd_arg <= std_logic_vector(resize((sector +
ERASE_SECTORS - 1) * sector_multiplier, cmd_arg'length));
                when erase =>
                    cmd <= "100110";
```

```
                    when write_length =>
                        cmd <= "011000";
                        read_dest <= dest_length;
                    when others => null;
                end case;
        end case;
    end process;

    state_update_proc: process(sclk) begin
        if rising_edge(sclk) then
            state <= next_state;
        end if;
    end process;

end behavior;
```

sd_cmd.vhd

```
----------------------------------------------------------------------------
-------------
-- Company: ENGS 31, 18X
-- Engineer: Ben Wolsieffer
--
-- Create Date: 08/14/2018 04:27:27 PM
-- Design Name:
-- Module Name: sd_cmd - behavior
-- Project Name: VoiceRecordr
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies: sd_send.vhd, sd_recv.vhd, down_counter.vhd
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
```

```
-------------


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.math_real.all;

entity sd_cmd is
    port(sclk: in std_logic;
         cmd: in std_logic_vector(5 downto 0);
         cmd_arg: in std_logic_vector(31 downto 0);
         cmd_start: in std_logic; -- assert to start command
         cmd_end: out std_logic;
         res_r1: out std_logic_vector(7 downto 0) := x"00"; -- r1
response (first byte of response)
         res_data: out std_logic_vector(31 downto 0) := x"00000000";
-- response data/card status
         data_index: out std_logic_vector(8 downto 0);
         data_in: in std_logic_vector(7 downto 0);
         data_out: out std_logic_vector(7 downto 0);
         data_out_en: out std_logic;
         error: out std_logic;
         sd_spi_sclk: out std_logic;
         sd_spi_mosi: out std_logic;
         sd_spi_miso: in std_logic;
         sd_spi_cs: out std_logic);
end sd_cmd;

architecture behavior of sd_cmd is
    constant CMD_LENGTH: integer := 7;
    constant CMD_INDEX_BITS: integer :=
integer(ceil(log2(real(CMD_LENGTH))));

    component down_counter is
        generic(bits: positive := 4);
        port(clk: in std_logic;
             k: in std_logic_vector(bits - 1 downto 0); -- preset
```

```
value
            CE: in std_logic := '1'; -- count enable
            preset: in std_logic := '0'; -- assert to set the
counter to k
            y: out std_logic_vector(bits - 1 downto 0); -- counter
output
            TC: out std_logic); -- terminal count
    end component;

    component sd_send is
        generic(bits: positive := 8);
        port(clk: in std_logic;
             data: in std_logic_vector(bits - 1 downto 0); -- data to
send to card
             new_data: in std_logic; -- data is registered on rising
edge when asserted
             done: out std_logic; -- when asserted, the previously
registered data has been processed
             spi_mosi: out std_logic := '0';
             spi_cs: out std_logic := '1');
    end component;

    component sd_recv is
         generic(bits: positive := 8);
         port(clk: in std_logic;
              data: out std_logic_vector(bits - 1 downto 0); -- data
to send to card
              new_data: out std_logic; -- asserted when new data
available on next rising edge, cleared after get_data asserted
              spi_miso: in std_logic := '0');
     end component;

    type state_type is (idle,
                        cmd_load, send_wait, -- send command
                        res_wait, recv_wait, res_data_load, --
receive response
                        data_token_wait, data_read_wait,
data_read_load, -- read data block
```

```
                        data_token_load, data_write_load,
data_write_wait, data_response_wait, -- write data block
                        busy_wait, done, sd_error);
    signal state: state_type := idle;
    signal next_state: state_type;

    type send_data_src_type is (src_cmd, src_data_in,
src_data_token);
    signal send_data_src: send_data_src_type;

    signal send_data: std_logic_vector(7 downto 0);
    signal send_done, send_new_data: std_logic;

    signal recv_data: std_logic_vector(7 downto 0);
    signal recv_data_block, recv_done: std_logic;

    -- Command type and argument registers
    signal cmd_reg: unsigned(cmd'range);
    signal cmd_arg_reg: std_logic_vector(cmd_arg'range);
    signal cmd_reg_en: std_logic;

    signal cmd_index_vec: std_logic_vector(CMD_INDEX_BITS - 1 downto
0);
    signal cmd_index: unsigned(cmd_index_vec'range);
    signal cmd_index_preset, cmd_index_inc, cmd_done: std_logic;
    signal cmd_data: std_logic_vector(55 downto 0);

    -- Response data
    signal res_max_index: std_logic_vector(3 downto 0);
    signal res_index_vec: std_logic_vector(res_max_index'range);
    signal res_index: unsigned(res_index_vec'range);
    signal res_index_preset, res_index_inc, res_r1_en, res_data_en,
res_done: std_logic;

    -- Data block
    signal data_index_reset, data_index_inc: std_logic;
    signal data_index_reg, data_index_max: unsigned(9 downto 0) :=
(others => '0');
```

```
begin

    sd_spi_sclk <= sclk;

    send: sd_send
        port map(clk => sclk,
                 data => send_data,
                 new_data => send_new_data,
                 done => send_done,
                 spi_mosi => sd_spi_mosi,
                 spi_cs => sd_spi_cs);

    recv: sd_recv
        port map(clk => sclk,
                 data => recv_data,
                 new_data => recv_done,
                 spi_miso => sd_spi_miso);

    cmd_index <= unsigned(cmd_index_vec);
    cmd_index_counter: down_counter
        generic map(bits => CMD_INDEX_BITS)
        port map(clk => sclk,
                 k => std_logic_vector(to_unsigned(CMD_LENGTH - 1,
CMD_INDEX_BITS)),
                 CE => cmd_index_inc,
                 preset => cmd_index_preset,
                 y => cmd_index_vec,
                 TC => cmd_done);

    cmd_reg_update: process(sclk) begin
        if rising_edge(sclk) then
            if cmd_reg_en = '1' then
                cmd_reg <= unsigned(cmd);
                cmd_arg_reg <= cmd_arg;
            end if;
        end if;
    end process;
```

```
    cmd_data_proc: process(cmd_reg, cmd_arg_reg)
        variable crc: std_logic_vector(6 downto 0);
    begin
        -- Only CMD0 and CMD8 actually require CRC unless it is
explicitly
        -- enabled. Rather than calculate it, we use hardcoded values
assuming
        -- the argument will always be the same.
        if cmd_reg = 0 then
            -- arg is 0x00000000
            crc := "1001010";
        elsif cmd_reg = 8 then
            -- arg is 0x000001AA
            crc := "1000011";
        else
            crc := "0000000";
        end if;

        -- some cards apparently like having a 0xFF byte ahead of the
command (after CS is asserted)
        cmd_data <= x"FF" & "01" & std_logic_vector(cmd_reg) &
cmd_arg_reg & crc & "1";
    end process;

    send_data_proc: process(send_data_src, cmd_index, cmd_data,
data_index_reg, data_index_max, data_in)
        variable addr_low: integer;
    begin
        addr_low := to_integer(cmd_index) * 8;
        case send_data_src is
            when src_cmd => send_data <= cmd_data(addr_low + 7 downto
addr_low);
            when src_data_in =>
                if data_index_reg < data_index_max - 1 then
                    send_data <= data_in;
                else
                    -- last two bytes are a dummy CRC
```

```
                send_data <= x"00";
            end if;
        when src_data_token => send_data <= x"FE";
    end case;
end process;

res_max_index_proc: process(cmd_reg) begin
    -- calculate response length
    case to_integer(cmd_reg) is
        when 8 | 41 | 58 =>
            res_max_index <= x"3";
        when others =>
            res_max_index <= x"0";
    end case;
end process;

res_index <= unsigned(res_index_vec);
res_index_counter: down_counter
        generic map(bits => res_index'length)
        port map(clk => sclk,
                 k => res_max_index,
                 CE => res_index_inc,
                 preset => res_index_preset,
                 y => res_index_vec,
                 TC => res_done);

res_r1_proc: process(sclk) begin
    if rising_edge(sclk) then
        if res_r1_en = '1' then
            res_r1 <= recv_data;
        end if;
    end if;
end process;

res_data_proc: process(sclk, res_index)
    variable addr_low: integer;
begin
    addr_low := to_integer(res_index) * 8;
```

```
        if rising_edge(sclk) then
            if res_data_en = '1' then
                res_data(addr_low + 7 downto addr_low) <= recv_data;
            end if;
        end if;
    end process;

    data_index_max_proc: process(cmd_reg) begin
        case to_integer(cmd_reg) is
            when 17 | 18 | 24 | 25 => data_index_max <= "1000000001";
-- 513 (512 bytes + CRC)
            when 9 | 10 => data_index_max <= "0000010001"; -- 17 (16
bytes + CRC)
            when others => data_index_max <= (others => '0');
        end case;
    end process;

    data_index_counter: process(sclk, data_index_reset,
data_index_inc) begin
        if rising_edge(sclk) then
            if data_index_reset = '1' then
                data_index_reg <= (others => '0');
            elsif data_index_inc = '1' then
                data_index_reg <= data_index_reg + 1;
            end if;
        end if;
    end process;

    -- Truncate index register and send it to output. This will wrap
around when
    -- it reaches the CRC (which the user doesn't care about), but
data_out_en
    -- will not be asserted, so users should ignore it.
    data_index <= std_logic_vector(data_index_reg(data_index'range));
    -- send received data directly to data_out
    -- should only be assumed to be valid when data_out_en is
asserted
    data_out <= recv_data;
```

```
    next_state_proc: process(state, cmd_reg, cmd_start, cmd_done,
send_done, recv_data, recv_done, res_done, data_index_reg,
data_index_max) begin
        next_state <= state;
        case state is
            when idle =>
                if cmd_start = '1' then
                    next_state <= cmd_load;
                end if;
            when cmd_load =>
                if cmd_done = '1' then
                    next_state <= res_wait;
                else
                    next_state <= send_wait;
                end if;
            when send_wait =>
                if send_done = '1' then
                    next_state <= cmd_load;
                end if;
            when res_wait =>
                if recv_data /= x"FF" then
                    -- ignore idle state bit in determining if there
was an error
                    if (recv_data and x"FE") = x"00" then
                        next_state <= recv_wait;
                    else
                        next_state <= sd_error;
                    end if;
                end if;
            when recv_wait =>
                if recv_done = '1' then
                    next_state <= res_data_load;
                end if;
            when res_data_load =>
                if res_done = '1' then
                    case to_integer(cmd_reg) is
                        -- commands that have data block
```

```
                        when 9 | 10 | 17 => next_state <=
data_token_wait;
                        when 24 => next_state <= data_token_load;
                        -- responses possibly have a busy signal
after them
                        when others => next_state <= busy_wait;
                    end case;
                else
                    next_state <= recv_wait;
                end if;
            when data_token_wait =>
                if recv_data = x"FE" then
                    next_state <= data_read_wait;
                end if;
            when data_read_wait =>
                if recv_done = '1' then
                    next_state <= data_read_load;
                end if;
            when data_read_load =>
                if data_index_reg = data_index_max then
                    next_state <= done;
                else
                    next_state <= data_read_wait;
                end if;
            when data_token_load => next_state <= data_write_wait;
            when data_write_wait =>
                if send_done = '1' then
                    next_state <= data_write_load;
                end if;
            when data_write_load =>
                if data_index_reg = data_index_max then
                    next_state <= data_response_wait;
                else
                    next_state <= data_write_wait;
                end if;
            when data_response_wait =>
                if recv_data(4 downto 0) = "00101" then
                    -- correct response, wait for busy signal to end
```

```
                next_state <= busy_wait;
            elsif recv_data /= x"FF"  then
                -- got response, but it was not what was expected
                next_state <= sd_error;
            end if;
        when busy_wait =>
            if recv_data = x"FF" then
                next_state <= done;
            end if;
        when done => next_state <= idle;
        when sd_error => next_state <= idle;
    end case;
end process;

output_proc: process(state, data_index_reg, data_index_max) begin
    cmd_reg_en <= '0';
    cmd_index_preset <= '0';
    cmd_index_inc <= '0';
    send_new_data <= '0';
    res_index_preset <= '0';
    res_index_inc <= '0';
    res_r1_en <= '0';
    res_data_en <= '0';
    data_index_reset <= '0';
    data_index_inc <= '0';
    data_out_en <= '0';
    send_data_src <= src_cmd;
    cmd_end <= '0';
    error <= '0';

    case state is
        when idle =>
            cmd_reg_en <= '1';
            cmd_index_preset <= '1';
        when cmd_load =>
            cmd_index_inc <= '1';
            send_new_data <= '1';
        when send_wait => null;
```

```
            when res_wait =>
                res_index_preset <= '1';
                res_r1_en <= '1';
            when recv_wait => null;
            when res_data_load =>
                res_index_inc <= '1';
                res_data_en <= '1';
            when data_token_wait =>
                data_index_reset <= '1';
            when data_read_wait => null;
            when data_read_load =>
                data_index_inc <= '1';
                -- only enable output when we have not yet reached
the CRC
                if data_index_reg <= data_index_max - 2 then
                    data_out_en <= '1';
                end if;
            when data_token_load =>
                data_index_reset <= '1';
                send_data_src <= src_data_token;
                send_new_data <= '1';
            when data_write_wait => null;
            when data_write_load =>
                data_index_inc <= '1';
                send_new_data <= '1';
                send_data_src <= src_data_in;
            when data_response_wait => null;
            when busy_wait => null;
            when done => cmd_end <= '1';
            when sd_error =>
                error <= '1';
                cmd_end <= '1';
        end case;
    end process;

    state_update_proc: process(sclk) begin
        if rising_edge(sclk) then
            state <= next_state;
```

```
        end if;
    end process;

end behavior;
```

sd_send.vhd

```
----------------------------------------------------------------------------
-------------
-- Company: ENGS 31, 18X
-- Engineer: Ben Wolsieffer
--
-- Create Date: 08/13/2018 08:04:27 PM
-- Design Name:
-- Module Name: sd_send - behavior
-- Project Name: VoiceRecorder
-- Target Devices: Artix 7 - Basys 3
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
-------------


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.math_real.all;

entity sd_send is
    generic(bits: positive := 8);
    port(clk: in std_logic;
```

```
        data: in std_logic_vector(bits - 1 downto 0); -- data to
send to card
        new_data: in std_logic; -- data is registered on rising edge
when asserted
        done: out std_logic; -- when asserted, the previously
registered data has been processed
        spi_mosi: out std_logic;
        spi_cs: out std_logic := '1');
end sd_send;

architecture behavior of sd_send is
    -- number of bits required to represent the bits parameter
    constant BITS_BITS: positive := integer(ceil(log2(real(bits))));

    component down_counter is
        generic(bits: positive := 4);
        port(clk: in std_logic;
             k: in std_logic_vector(bits - 1 downto 0); -- preset
value
             CE: in std_logic := '1'; -- count enable
             preset: in std_logic := '0'; -- assert to set the
counter to k
             y: out std_logic_vector(bits - 1 downto 0); -- counter
output
             TC: out std_logic); -- terminal count
    end component;

    signal data_reg, shift_reg: std_logic_vector(bits - 1 downto 0)
:= (others => '1');

    -- asserted when the last bit is shifted
    signal last_bit: std_logic;

    -- asserted when data has arrived but has not been moved to shift
register
    signal new_data_reg: std_logic := '0';
begin
```

```
    spi_mosi <= shift_reg(bits - 1);
    done <= last_bit;

    process(clk) begin
        if falling_edge(clk) then
            -- shift or move data from data_reg
            if new_data_reg = '1' and last_bit = '1' then
                shift_reg <= data_reg;
                new_data_reg <= '0';
                -- assert CS when starting to transfer
                -- CS is never deasserted, because all the SD cards I
have
                -- tested do not require it, although some cards
supposedly do
                -- require it before each command. The CS signal is
used to
                -- frame the start of a byte, and the framing is
maintained as
                -- long as the clock does not glitch
                spi_cs <= '0';
            else
                shift_reg <= shift_reg(bits - 2 downto 0) & '1';
            end if;

            -- copy data input into register
            if new_data = '1' then
                data_reg <= data;
                new_data_reg <= '1';
            end if;
        end if;
    end process;

    shift_counter: down_counter
        generic map(bits => BITS_BITS)
        port map(clk => clk,
                 k => std_logic_vector(to_unsigned(bits - 1,
BITS_BITS)),
                 TC => last_bit);
```

```
end behavior;
```

sd_recv.vhd

```
----------------------------------------------------------------------------
-------------
-- Company: ENGS 31, 18X
-- Engineer: Ben Wolsieffer
--
-- Create Date: 08/13/2018 08:04:27 PM
-- Design Name:
-- Module Name: sd_send - behavior
-- Project Name: VoiceRecorder
-- Target Devices: Artix 7 - Basys 3
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
-------------


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.math_real.all;

entity sd_recv is
    generic(bits: positive := 8);
    port(clk: in std_logic;
         data: out std_logic_vector(bits - 1 downto 0); -- data to
send to card
```

```
         new_data: out std_logic; -- asserted when new data available
on next rising edge, cleared after get_data asserted
         spi_miso: in std_logic := '0');
end sd_recv;

architecture behavior of sd_recv is
    -- number of bits required to represent bits - 1
    constant BITS_BITS: positive := integer(ceil(log2(real(bits))));

    component down_counter is
        generic(bits: positive := 4);
        port(clk: in std_logic;
             k: in std_logic_vector(bits - 1 downto 0); -- preset
value
             CE: in std_logic := '1'; -- count enable
             preset: in std_logic := '0'; -- assert to set the
counter to k
             y: out std_logic_vector(bits - 1 downto 0); -- counter
output
             TC: out std_logic); -- terminal count
    end component;

    signal shift_reg: std_logic_vector(bits - 1 downto 0) := (others
=> '1');

    -- asserted when the last bit is shifted
    signal last_bit: std_logic;
begin

    new_data <= last_bit;

    process(clk) begin
        if rising_edge(clk) then
            -- start shifting
            if last_bit = '1' then
                data <= shift_reg;
            end if;
```

```
            shift_reg <= shift_reg(bits - 2 downto 0) & spi_miso;
        end if;
    end process;

    shift_counter: down_counter
        generic map(bits => BITS_BITS)
        port map(clk => clk,
                 k => std_logic_vector(to_unsigned(bits - 1,
BITS_BITS)),
                 TC => last_bit);

end behavior;
```

down_counter.vhd

```
----------------------------------------------------------------------
-------------
-- Company: ENGS 31, 18X
-- Engineer: Ben Wolsieffer
--
-- Create Date: 07/23/2018 10:04:40 PM
-- Design Name:
-- Module Name: down_counter - behavior
-- Project Name: down_counter
-- Target Devices: Artix 7 - Basys 3
-- Tool Versions:
-- Description: A generic counter implementation that only counts
down.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------
-------------
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity down_counter is
    generic(bits: positive := 4);
    port (clk: in std_logic;
          k: in std_logic_vector(bits - 1 downto 0); -- preset value
          CE: in std_logic := '1'; -- count enable
          preset: in std_logic := '0'; -- assert to set the counter
to k
          y: out std_logic_vector(bits - 1 downto 0); -- counter
output
          TC: out std_logic); -- terminal count
end down_counter;

architecture behavior of down_counter is
    signal uy: unsigned(y'range) := (others => '0');
begin
    y <= std_logic_vector(uy);

    TC <= '1' when uy = 0 else '0';

    process(clk) begin
        if rising_edge(clk) then
            if preset = '1' then
                uy <= unsigned(k);
            elsif CE = '1' then
                -- wrap around to k
                if uy = 0 then
                    uy <= unsigned(k);
                else
                    uy <= uy - 1;
                end if;
            end if;
        end if;
    end process;
end behavior;
```

clock_divider.vhd

```
----------------------------------------------------------------------------
-------------
-- Company: ENGS 31, 18X
-- Engineer: Ben Wolsieffer
--
-- Create Date: 08/11/2018 08:38:05 PM
-- Design Name:
-- Module Name: clock_divider - behavior
-- Project Name:
-- Target Devices: Artix 7 - Basys 3
-- Tool Versions:
-- Description:
--
-- Dependencies: down_counter.vhd
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------
-------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.math_real.all;

library UNISIM;
use UNISIM.VComponents.all;

entity clock_divider is
    generic(divider: integer);
    port(mclk: in std_logic;
         dclk: out std_logic);
end clock_divider;
```

```
architecture behavior of clock_divider is

    constant COUNTER_VALUE: integer := divider / 2 - 1;
    constant COUNTER_BITS: integer :=
integer(ceil(log2(real(COUNTER_VALUE + 1))));
    signal dclk_unbuf: std_logic := '0'; -- unbuffered clock
    signal dclk_toggle: std_logic;

    component down_counter is
        generic(bits: positive := 4);
        port (clk: in std_logic;
              k: in std_logic_vector(bits - 1 downto 0); -- preset
value
              CE: in std_logic := '1'; -- count enable
              preset: in std_logic := '0'; -- assert to set the
counter to k
              y: out std_logic_vector(bits - 1 downto 0); -- counter
output
              TC: out std_logic); -- terminal count
    end component;
begin

    assert (COUNTER_VALUE + 1) * 2 = divider report "Divider must be
a multiple of two";

    counter: down_counter
        generic map(COUNTER_BITS)
        port map(clk => mclk,
                 k => std_logic_vector(to_unsigned(COUNTER_VALUE,
COUNTER_BITS)),
                 TC => dclk_toggle);

    process(mclk) begin
        if rising_edge(mclk) then
            if dclk_toggle = '1' then
                dclk_unbuf <= not(dclk_unbuf);
            end if;
        end if;
```

```
    end process;

    -- The BUFG component puts the signal onto the FPGA clocking
network
    dclk_buffer: BUFG
        port map(I => dclk_unbuf,
                 O => dclk);
end behavior;
```

sync.vhd

```
----------------------------------------------------------------------
-------------
-- Company: ENGS 31, 18X
-- Engineer: Ben Wolsieffer
--
-- Create Date: 08/12/2018 10:21:29 PM
-- Design Name:
-- Module Name: sync - behavior
-- Project Name: VoiceRecorder
-- Target Devices:
-- Tool Versions:
-- Description: Dual flop synchronizer
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------
-------------


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.math_real.all;
use IEEE.numeric_std.all;
```

```
entity sync is
    port(clk: in std_logic;
         input: in std_logic;
         output: out std_logic);
end sync;

architecture behavior of sync is
    signal sync_1: std_logic := '0';
    signal sync_2: std_logic := '0';
begin

   -- synchronization
    sync_proc: process(clk) begin
        if rising_edge(clk) then
            sync_1 <= input;
            sync_2 <= sync_1;
        end if;
    end process;

    output <= sync_2;

end behavior;
```

button.vhd

```
----------------------------------------------------------------------------
-------------
-- Company: ENGS 31, 18X
-- Engineer: Ben Wolsieffer
--
-- Create Date: 08/12/2018 10:21:29 PM
-- Design Name:
-- Module Name: button - behavior
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description: Button synchronizer, debouncer and monopulser
```

```
--
-- Dependencies: down_counter.vhd
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.math_real.all;
use IEEE.numeric_std.all;

entity button is
    generic(count: positive := 10000);
    port(clk: in std_logic;
         input: in std_logic;
         output: out std_logic);
end button;

architecture behavior of button is
    constant DEBOUNCE_BITS: positive := integer(ceil(log2(real(count
+ 1))));

    component sync is
        port(clk: in std_logic;
             input: in std_logic;
             output: out std_logic);
    end component;

    component down_counter is
        generic(bits: positive := 4);
        port (clk: in std_logic;
              k: in std_logic_vector(bits - 1 downto 0); -- preset
value
```

```
            CE: in std_logic := '1'; -- count enable
            preset: in std_logic := '0'; -- assert to set the
counter to k
            y: out std_logic_vector(bits - 1 downto 0); -- counter
output
            TC: out std_logic); -- terminal count
    end component;

    -- synchronization
    signal sync_out: std_logic;

    -- debouncing
    signal debounce_output: std_logic := '0';
    signal not_changing: std_logic;
    signal change_trigger: std_logic;

    -- monopulsing
    signal mp_reg: std_logic_vector(1 downto 0) := "00";
begin

   -- synchronization
    sync_map: sync
        port map(clk => clk,
                 input => input,
                 output => sync_out);

    -- debouncing
    not_changing <= '1' when sync_out = debounce_output else '0';
    debounce_counter: down_counter
        generic map(bits => DEBOUNCE_BITS)
        port map(clk => clk,
                 k => std_logic_vector(to_unsigned(count,
DEBOUNCE_BITS)),
                 preset => not_changing,
                 TC => change_trigger);

    process(clk) begin
        if rising_edge(clk) then
```

```
            if change_trigger = '1' then
                debounce_output <= sync_out;
            end if;
        end if;
    end process;

    -- monopulsing
    monopulser: process(clk, mp_reg, debounce_output)
    begin
        if rising_edge(clk) then
            mp_reg <= debounce_output & mp_reg(1);
        end if;

        output <= mp_reg(1) and not(mp_reg(0));
    end process monopulser;

end behavior;
```

VoiceRecorder.xdc

```
# Constraint file for the voice recorder

# Clock signal
#Bank = 34, Pin name = CLK,                          Sch name = CLK100MHZ
set_property PACKAGE_PIN W5 [get_ports mclk]
set_property IOSTANDARD LVCMOS33 [get_ports mclk]
create_clock -period 20.000 -name sys_clk_pin -waveform {0.000
10.000} -add [get_ports mclk]

# LEDs
set_property PACKAGE_PIN U16 [get_ports {record_led}]
set_property IOSTANDARD LVCMOS33 [get_ports {record_led}]
set_property PACKAGE_PIN E19 [get_ports {play_led}]
set_property IOSTANDARD LVCMOS33 [get_ports {play_led}]
set_property PACKAGE_PIN U19 [get_ports {f_play_led}]
set_property IOSTANDARD LVCMOS33 [get_ports {f_play_led}]
set_property PACKAGE_PIN V19 [get_ports {s_play_led}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {s_play_led}]
set_property PACKAGE_PIN W18 [get_ports {data_leds[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {data_leds[0]}]
set_property PACKAGE_PIN U15 [get_ports {data_leds[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {data_leds[1]}]
set_property PACKAGE_PIN U14 [get_ports {data_leds[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {data_leds[2]}]
set_property PACKAGE_PIN V14 [get_ports {data_leds[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {data_leds[3]}]
set_property PACKAGE_PIN V13 [get_ports {data_leds[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {data_leds[4]}]
set_property PACKAGE_PIN V3 [get_ports {data_leds[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {data_leds[5]}]
set_property PACKAGE_PIN W3 [get_ports {data_leds[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {data_leds[6]}]
set_property PACKAGE_PIN U3 [get_ports {data_leds[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {data_leds[7]}]
set_property PACKAGE_PIN P3 [get_ports {data_leds[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {data_leds[8]}]
set_property PACKAGE_PIN N3 [get_ports {data_leds[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {data_leds[9]}]
set_property PACKAGE_PIN P1 [get_ports {data_leds[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {data_leds[10]}]
set_property PACKAGE_PIN L1 [get_ports {data_leds[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {data_leds[11]}]

#Buttons
##Bank = 14, Pin name = ,                                          Sch
name = BTNU
set_property PACKAGE_PIN T18 [get_ports {play_btn}]
set_property IOSTANDARD LVCMOS33 [get_ports {play_btn}]
#Bank = 14, Pin name = ,    Sch name = BTNL
set_property PACKAGE_PIN W19 [get_ports {record_btn}]
set_property IOSTANDARD LVCMOS33 [get_ports {record_btn}]
##Bank = 14, Pin name = ,
Sch name = BTNR
set_property PACKAGE_PIN T17 [get_ports {f_play_btn}]
set_property IOSTANDARD LVCMOS33 [get_ports {f_play_btn}]
```

```
##Bank = 14, Pin name = ,                                   Sch
name = BTND
set_property PACKAGE_PIN U17 [get_ports {s_play_btn}]
set_property IOSTANDARD LVCMOS33 [get_ports {s_play_btn}]



##Pmod Header JA
##Bank = 15, Pin name = IO_L1N_T0_AD0N_15,                  Sch
name = JA1
set_property PACKAGE_PIN J1 [get_ports {ad_spi_cs}]
set_property IOSTANDARD LVCMOS33 [get_ports {ad_spi_cs}]
##Bank = 15, Pin name = IO_L16N_T2_A27_15,                  Sch
name = JA3
set_property PACKAGE_PIN J2 [get_ports {ad_spi_sdata}]
set_property IOSTANDARD LVCMOS33 [get_ports {ad_spi_sdata}]
##Bank = 15, Pin name = IO_L16P_T2_A28_15,                  Sch
name = JA4
set_property PACKAGE_PIN G2 [get_ports {ad_spi_sclk}]
set_property IOSTANDARD LVCMOS33 [get_ports {ad_spi_sclk}]

##Pmod Header JB
#Bank = 15, Pin name = IO_L15N_T2_DQS_ADV_B_15,            Sch
name = JB1
set_property PACKAGE_PIN A14 [get_ports {sd_spi_cs}]
set_property IOSTANDARD LVCMOS33 [get_ports {sd_spi_cs}]
###Bank = 14, Pin name = IO_L13P_T2_MRCC_14,                Sch
name = JB2
set_property PACKAGE_PIN A16 [get_ports {sd_spi_mosi}]
set_property IOSTANDARD LVCMOS33 [get_ports {sd_spi_mosi}]
###Bank = 14, Pin name = IO_L21N_T3_DQS_A06_D22_14,         Sch
name = JB3
set_property PACKAGE_PIN B15 [get_ports {sd_spi_miso}]
set_property IOSTANDARD LVCMOS33 [get_ports {sd_spi_miso}]
###Bank = CONFIG, Pin name = IO_L16P_T2_CSI_B_14,      Sch name
= JB4
set_property PACKAGE_PIN B16 [get_ports {sd_spi_sclk}]
set_property IOSTANDARD LVCMOS33 [get_ports {sd_spi_sclk}]
```

```
##Bank = 14, Pin name = IO_L24P_T3_A01_D17_14,                    Sch
name = JB9
set_property PACKAGE_PIN C15 [get_ports {sd_cd}]
set_property IOSTANDARD LVCMOS33 [get_ports {sd_cd}]
##Bank = 14, Pin name = IO_L19N_T3_A09_D25_VREF_14,               Sch
name = JB10
set_property PACKAGE_PIN C16 [get_ports {sd_wp}]
set_property IOSTANDARD LVCMOS33 [get_ports {sd_wp}]

##Pmod Header JXADC
##Bank = 15, Pin name = IO_L9P_T1_DQS_AD3P_15,                    Sch
name = XADC1_P -> XA1_P
set_property PACKAGE_PIN J3 [get_ports {da_spi_cs}]
set_property IOSTANDARD LVCMOS33 [get_ports {da_spi_cs}]
##Bank = 15, Pin name = IO_L8P_T1_AD10P_15,                       Sch
name = XADC2_P -> XA2_P
set_property PACKAGE_PIN L3 [get_ports {da_spi_sdata}]
set_property IOSTANDARD LVCMOS33 [get_ports {da_spi_sdata}]
##Bank = 15, Pin name = IO_L10P_T1_AD11P_15,
     Sch name = XADC4_P -> XA4_P
set_property PACKAGE_PIN N2 [get_ports {da_spi_sclk}]
set_property IOSTANDARD LVCMOS33 [get_ports {da_spi_sclk}]

set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
set_property BITSTREAM.CONFIG.SPI_BUSWIDTH 4 [current_design]
set_property CONFIG_MODE SPIx4 [current_design]

set_property BITSTREAM.CONFIG.CONFIGRATE 33 [current_design]

set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]
```

Appendix F - Resource utilization

Table F1. Slice Logic

Site Type	**Used**	**Fixed**	**Available**	**Util%**
Slice LUTs	935	0	20800	4.50
LUT as Logic	935	0	20800	4.50
LUT as Memory	0	0	9600	0.00
Slice Registers	425	0	41600	1.02
Register as Flip Flop	425	0	41600	1.02
Register as Latch	0	0	41600	0.00
F7 Muxes	75	0	16300	0.46
F8 Muxes	0	0	8150	0.00

Table F2. Memory

Site Type	**Used**	**Fixed**	**Available**	**Util%**
Block RAM Tile	44	0	50	88.00
RAMB36/FIFO*	44	0	50	88.00
RAMB36E1 only	44			
RAMB18	0	0	100	0.00

Table F3. Primitives

Ref Name	**Used**	**Functional Category**
FDRE	412	Flop & Latch
LUT6	405	LUT
LUT4	284	LUT
LUT5	219	LUT
LUT3	138	LUT
CARRY4	82	CarryLogic
MUXF7	75	MuxFx
LUT2	61	LUT
RAMB36E1	44	Block Memory
LUT1	40	LUT
OBUF	24	IO
FDSE	9	Flop & Latch
IBUF	8	IO
FDCE	4	Flop & Latch
BUFG	2	Clock

Appendix G - Residual warnings

```
[Synth 8-3331] design sd_driver has unconnected port sd_cd
[Synth 8-3331] design voice_recorder has unconnected port sd_cd
```

These warnings occur because the SD card detect pin is mapped in the constraint file and the top level file, but was never used due to time constraints.

```
[Synth 8-3936] Found unconnected internal register 'ocr_reg_reg' and
it is trimmed from '32' to '31' bits. ["sd/sd_driver.vhd":195]
[Synth 8-3332] Sequential element (sd_cmd_map/res_data_reg[31]) is
unused and will be removed from module sd_driver.
[Synth 8-3332] Sequential element (sd_cmd_map/res_data_reg[29]) is
unused and will be removed from module sd_driver.
... elided ...
[Synth 8-3332] Sequential element (sd_cmd_map/res_data_reg[13]) is
unused and will be removed from module sd_driver.
[Synth 8-3332] Sequential element (sd_cmd_map/res_data_reg[12]) is
unused and will be removed from module sd_driver.
```

These warnings occur because the entire SD Operating Conditions Register is saved to a register, but the code only uses one bit from it.

```
[Synth 8-3332] Sequential element (csd_reg_reg[125]) is unused and
will be removed from module sd_driver.
[Synth 8-3332] Sequential element (csd_reg_reg[124]) is unused and
will be removed from module sd_driver.
... elided ...
[Synth 8-3332] Sequential element (csd_reg_reg[19]) is unused and will
be removed from module sd_driver.
[Synth 8-3332] Sequential element (csd_reg_reg[18]) is unused and will
be removed from module sd_driver.
```

These warnings occur because the entire SD Card Specific Data register is saved, but only a portion of it is used.

Appendix H - Memory map

The block RAM on the FPGA was used as a circular buffer that stored audio samples as they were transferred between the audio controller and SD driver. Each element was 12 bits wide, the size of a sample, and the size of the RAM was 131,072 (2^{17}), the largest power of two that fits in the available block RAM.

Appendix I - Simulation waveforms[3]

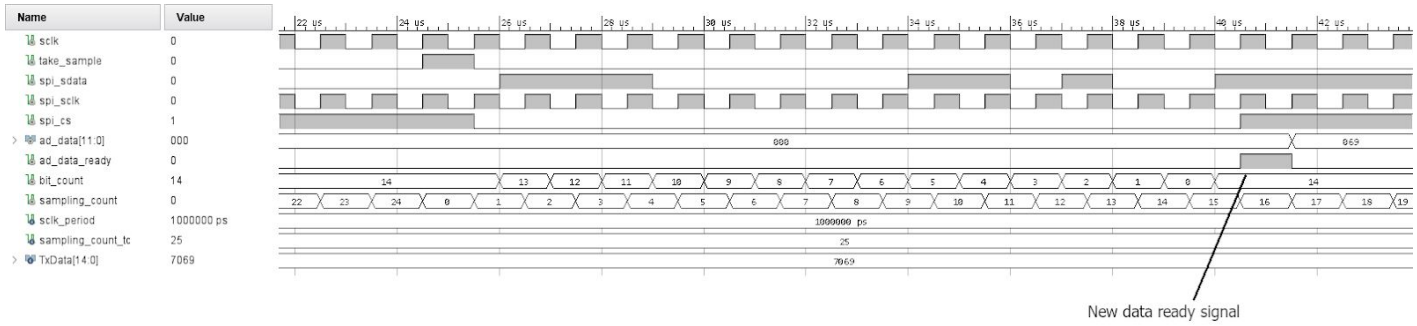


Figure I1. Pmod AD1 simulation waveform

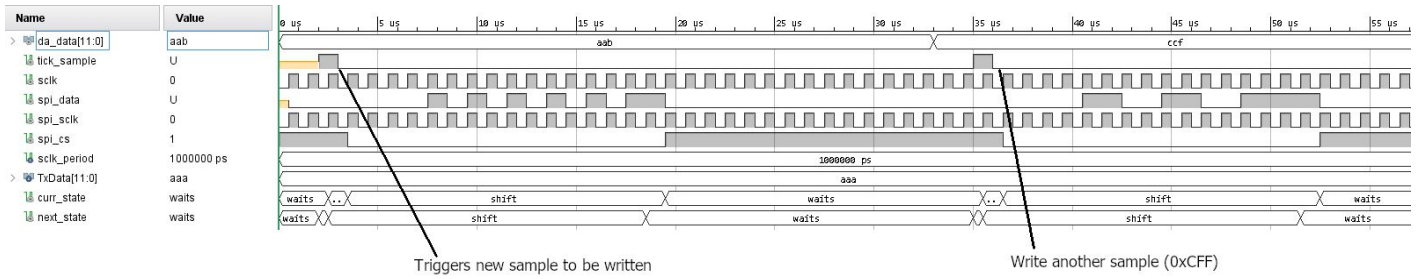


Figure I2. Pmod DA2 simulation waveform

[3] To see enlarged versions of these diagrams, click on the image.

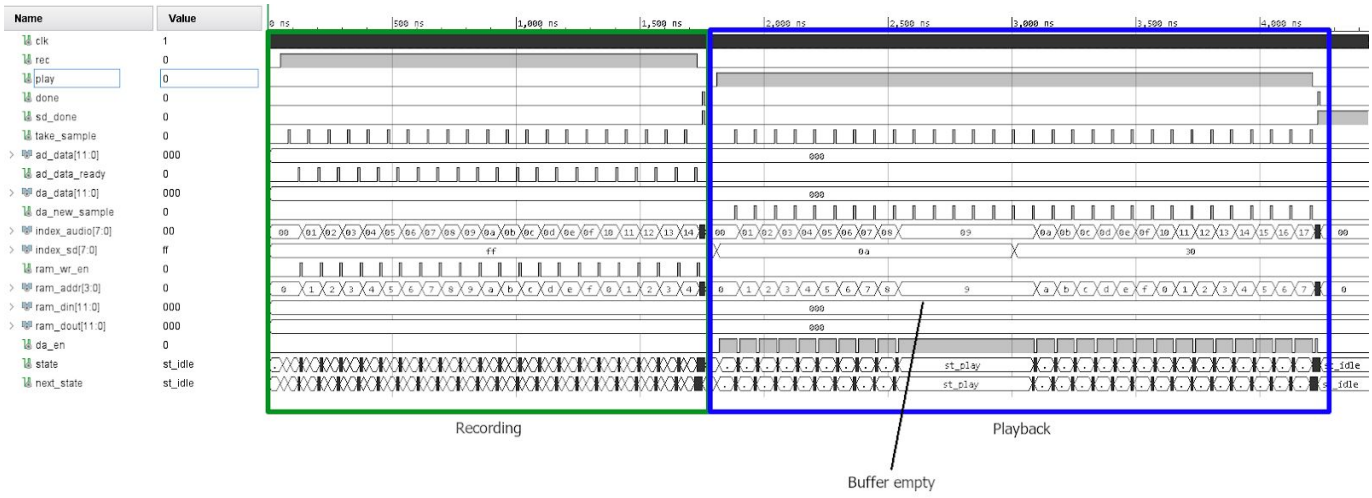


Figure I3. Audio controller simulation waveform

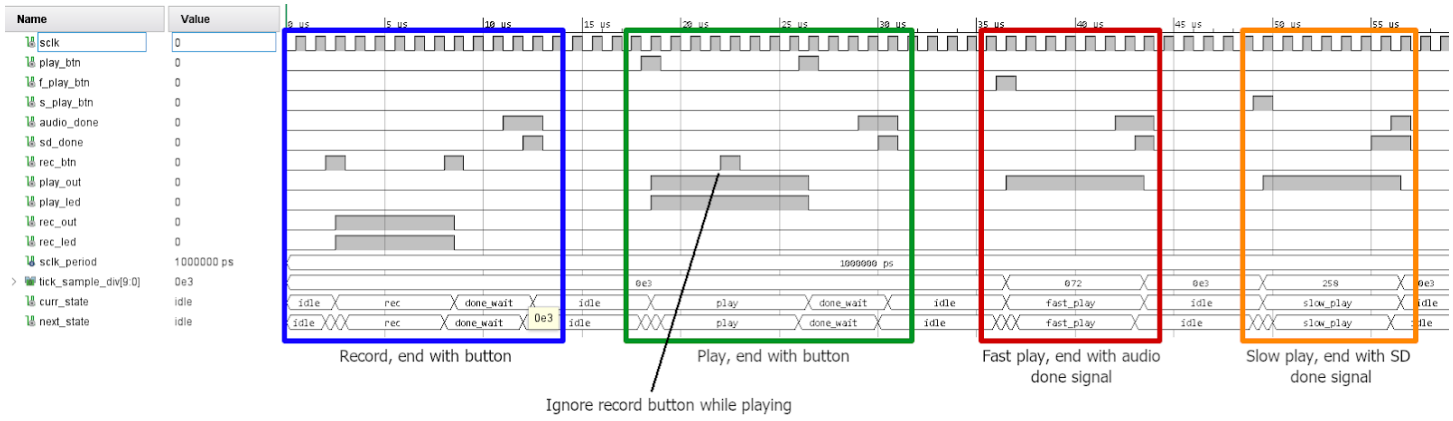


Figure I4. UI controller simulation waveform

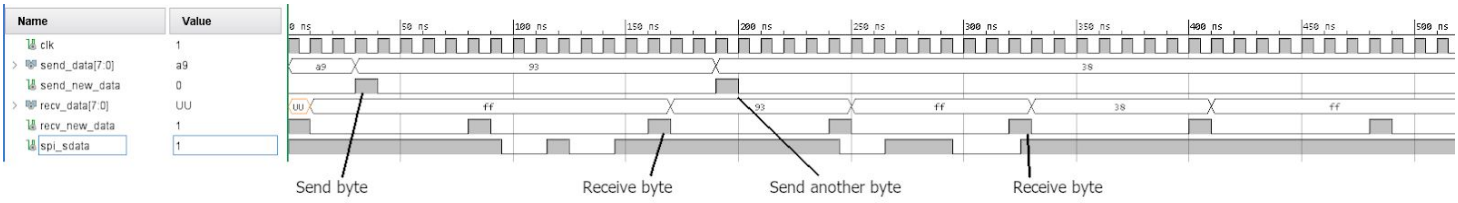


Figure I5. Simulation waveform for communication between sd_send and sd_recv components.

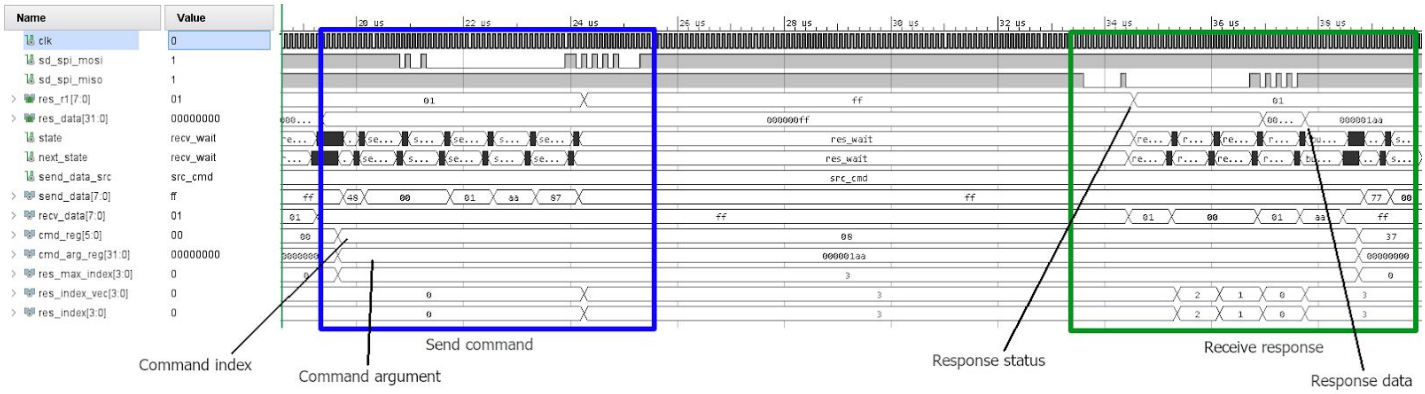


Figure I6. SD command controller simulation waveform

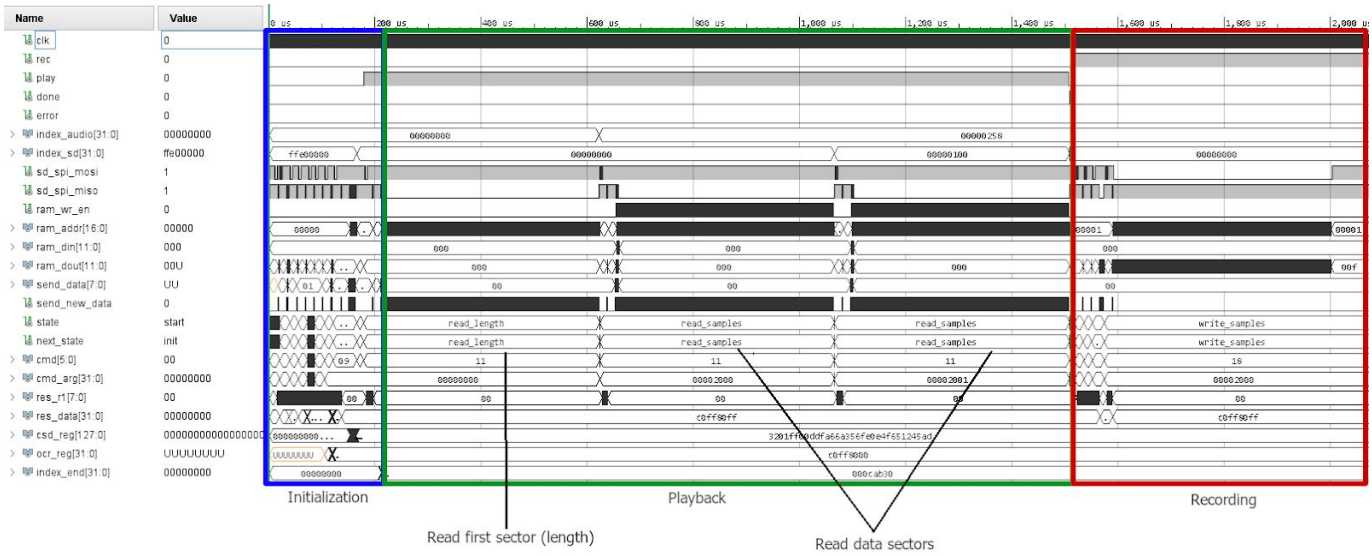


Figure I7. SD driver simulation waveform

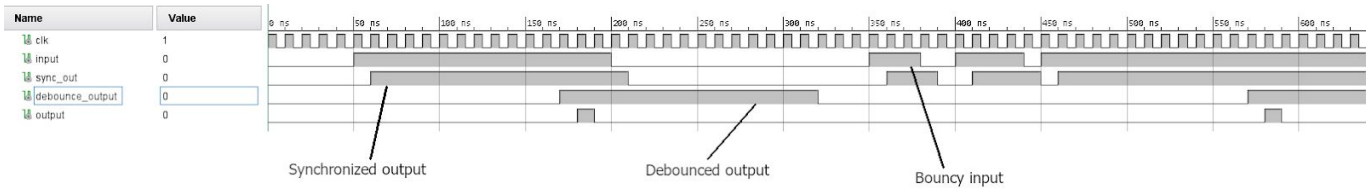


Figure I8. Button synchronizer, debouncer and monopulser simulation waveform

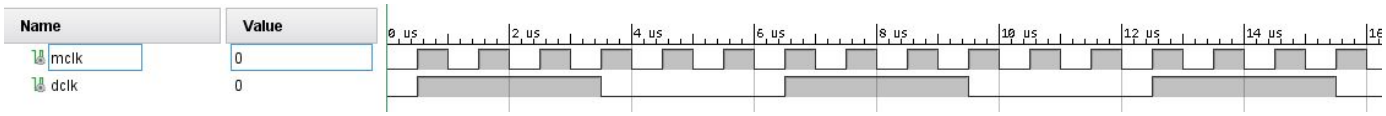


Figure I9. Clock divider simulation waveform

Appendix J - Computer program

Python script for writing an audio file to an SD card in the format expected by the voice recorder.

```
#!/usr/bin/env python3

import sys
import soundfile as sf
import struct
import numpy as np

SECTOR_SIZE = 512
DATA_OFFSET = 8192 * SECTOR_SIZE


def main():
    if len(sys.argv) < 3:
        print("usage: {} audio_file sd_dev".format(sys.argv[0]),
file=sys.stderr)
        sys.exit(1)

    audio_file_name = sys.argv[1]
    sd_name = sys.argv[2]

    with sf.SoundFile(audio_file_name, 'r') as f:
        print(f.format_info)
        print(f.extra_info)
        print(f.subtype_info)

        assert f.samplerate == 44100
        assert f.subtype == 'PCM_16'

        with open(sd_name, 'rb+') as sd:
            sd.seek(0)
            # Index of end of data (relative to offset)
            sd.write(struct.pack('<I', len(f) - 1))
            sd.seek(DATA_OFFSET)
```

```
            while f.tell() < len(f):
                data = f.read(frames=4096, dtype='int16')
                mono_data = np.mean(data, axis=1).astype(np.int32)
                unsigned_data = (mono_data + 32768).astype(np.uint16)
                for sample in unsigned_data:
                    sd.write(struct.pack('<H', sample))


if __name__ == "__main__":
    main()
```