Digital Music Player
ELEX7660 Digital System Design

Nathaniel Rohrick
Preston Thompson
## Contents

1 Introduction .......................................................... 4

2 Background .................................................................... 5
   2.1 Digital Audio ......................................................... 5

3 System Architecture ..................................................... 5
   3.1 Digital-to-Analog Conversion .................................... 6
      3.1.1 dac CPU Peripheral ......................................... 7
      3.1.2 Analog Front-End ............................................. 7
      3.1.3 Class-D Amplifier ............................................. 7
   3.2 SD Card Data Management ....................................... 12
      3.2.1 spi CPU Peripheral ........................................... 13
   3.3 sdc Driver for SD Card Using the spi Peripheral .......... 14
   3.4 keypad CPU Peripheral ............................................ 16
   3.5 keypad Driver ....................................................... 16
   3.6 ledc CPU Peripheral .............................................. 17
      3.7 Firmware ........................................................ 17
         3.7.1 SD Card Format ............................................. 18
         3.7.2 FAT32 Driver ................................................. 19

4 Challenges .................................................................... 20
   4.1 dac Ring Buffer ...................................................... 20
   4.2 Keypad ................................................................ 20
   4.3 SD Card ................................................................ 20

5 Conclusion .................................................................... 21

6 Appendice ..................................................................... 23
   6.1 QSYS Circuit Schematic ......................................... 23
   6.2 Class-D Amplifier Circuit Schematic ......................... 24
   6.3 Software Listings ................................................... 31
   6.4 Bill of Materials .................................................... 61
      6.4.1 BOM billed to BCIT ........................................... 61
      6.4.2 Class-D Amplifier BOM ..................................... 62
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>System block diagram</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Breadboarded DAC circuit</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>DAC circuit schematic</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>Stereo PWM schematic</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Single channel PWM output</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>Schematic of complementary PWM generator with deadtime</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>Single channel H-Bridge schematic</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>SD card circuit schematic</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>State-machine for spi peripheral</td>
<td>13</td>
</tr>
<tr>
<td>10</td>
<td>State-machine for the keypad peripheral</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>Flowchart of firmware’s main loop</td>
<td>18</td>
</tr>
<tr>
<td>12</td>
<td>Class-D amplifier schematic pg. 1/6</td>
<td>25</td>
</tr>
<tr>
<td>13</td>
<td>Class-D amplifier schematic pg. 5/6</td>
<td>26</td>
</tr>
<tr>
<td>14</td>
<td>Class-D amplifier schematic pg. 3/6</td>
<td>27</td>
</tr>
<tr>
<td>15</td>
<td>Class-D amplifier schematic pg. 4/6</td>
<td>28</td>
</tr>
<tr>
<td>16</td>
<td>Class-D amplifier schematic pg. 5/6</td>
<td>29</td>
</tr>
<tr>
<td>17</td>
<td>Class-D amplifier schematic pg. 6/6</td>
<td>30</td>
</tr>
<tr>
<td>18</td>
<td>Components used in final design</td>
<td>61</td>
</tr>
<tr>
<td>19</td>
<td>Components needed for class-D amplifier</td>
<td>62</td>
</tr>
</tbody>
</table>

## Listings

<table>
<thead>
<tr>
<th>Listing</th>
<th>File</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>musicplayer_top.sv</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>ledc.sv</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>dac.sv</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>dac_tb.sv</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>spi.sv</td>
<td>36</td>
</tr>
<tr>
<td>6</td>
<td>spi_tb.sv</td>
<td>39</td>
</tr>
<tr>
<td>7</td>
<td>sdc.c</td>
<td>41</td>
</tr>
<tr>
<td>8</td>
<td>sdc.h</td>
<td>46</td>
</tr>
<tr>
<td>9</td>
<td>sdc_cmd.h</td>
<td>46</td>
</tr>
<tr>
<td>10</td>
<td>keypad.sv</td>
<td>47</td>
</tr>
<tr>
<td>11</td>
<td>keypad.c</td>
<td>50</td>
</tr>
<tr>
<td>12</td>
<td>keypad.h</td>
<td>51</td>
</tr>
<tr>
<td>13</td>
<td>main.c</td>
<td>51</td>
</tr>
<tr>
<td>14</td>
<td>fat.c</td>
<td>53</td>
</tr>
<tr>
<td>15</td>
<td>fat.h</td>
<td>55</td>
</tr>
<tr>
<td>16</td>
<td>deadtime.m</td>
<td>56</td>
</tr>
<tr>
<td>17</td>
<td>V12switcher.m</td>
<td>56</td>
</tr>
<tr>
<td>18</td>
<td>GATE_DRV_DESIGN.m</td>
<td>59</td>
</tr>
</tbody>
</table>
1 Introduction

For our Digital System Design course at BCIT we decided to build a digital music player. The system reads digital audio off of an SD card and interfaces to a digital-to-analog converter. While this certainly has been done before (every cell phone sold today is capable of this!) we wanted to design the digital hardware from scratch and learn about SD cards and digital audio.

In addition, we started work on a class-D audio amplifier to go along with the digital music player. The inspiration originally was to have the FPGA drive the H-bridge of the audio amplifier directly however we found out that the speed requirements to do this digitally were not possible. In light of this, the amplifier became a side project of the music player. We have included it in this report for posterity.

Work was also started on a FAT32 firmware driver and but due to time constraints it was not integrated with the rest of the system.
2 Background

2.1 Digital Audio

The audio format used in this project follows the Compact Disc Digital Audio (CDDA) standard for audio stored on CDs. The audio encoding used is 2-channel signed 16-bit PCM with a sample rate of 44.1 kHz. For simplicity, we opted to use little endian byte ordering to store the samples as the Nios II processor is little endian.

The music used to demonstrate this project came originally from MPEG-3 encoded audio files (.mp3). The MPEG-3 audio first needed to be decoded for use with the music player. To do this, a Linux utility (mpg123) was used to decode the audio and save it in the Waveform Audio File Format (.wav). Afterwards, another Linux utility (sox) was used to convert the .wav files to the raw audio format described above.

3 System Architecture

The system consisted of a Nios II soft-core processor and three custom memory-mapped peripherals for it: the dac, the spi, the keypad and the ledc modules.

![Figure 1: System block diagram](image-url)
3.1 Digital-to-Analog Conversion

The DAC used is a Burr-Brown PCM1725 Stereo Audio 16 bit 96 kHz DAC. The DAC only came from Digikey in a SOIC-14 package, so a SOIC-14 breakout board from Adafruit was used to connect it to the breadboard.

![Breadboarded DAC circuit](image1)

Figure 2: Breadboarded DAC circuit

![DAC circuit schematic](image2)

Figure 3: DAC circuit schematic
3.1.1 dac CPU Peripheral

The PCM1725 DAC supports I²S as well as having its own digital interface described in its datasheet. Fortunately this digital interface is simple enough to be able to interface to using nothing but the timing diagram. The only detail that was not mentioned in the datasheet is that the DAC uses signed integers.

A complication we faced was that the sample rate of the music was not an integer divisor of the FPGA’s system clock. Even using one of the available PLLs on the FPGA would not get an accurate enough clock because of the large multiplier and divisor. To solve this problem, we devised a method where the DAC chip’s clock signal’s period was constantly modified to average out to make exactly 44.1 kHz playback rate.

The dac peripheral has a FIFO in the form of a ring buffer for storing the samples. From the CPU’s perspective there is only one register to write to, dac_sample. When the CPU writes to dac_sample, the dac peripheral puts the sample onto the ring buffer. The format of the data written to dac_sample is the 16-bit left channel sample in bits \([31:16]\) and the 16-bit right channel sample in bits \([15:0]\).

3.1.2 Analog Front-End

The output of the DAC needed to be filtered, amplified, and AC-coupled before going out to a pair of headphones or speakers. To do this we used an audio-grade op-amp, the OPA2134PA. It is a dual op-amp circuit in a DIP-8 package. The op-amp was configured to be a 3rd order low-pass filter. The output of the op-amp is AC-coupled before connecting to a 3.5mm TRRS stereo connector.

3.1.3 Class-D Amplifier

The class-D amplifier is a power amplifier topology that boasts the highest power efficiency of all the amplifier classes. It accomplishes this by modulating the input signal to a high frequency pulse-width modulated waveform (PWM). Since PWM is digital in nature, they do not require FETs to be in the linear I-V region. FETs dissipate little power in the saturation conduction region and efficiency is mostly limited by switching losses [1].

To modulate the incoming signal, a simple comparison is made with a triangle wave which acts as the carrier. Below gives a practical implementation of this modulation scheme [2].
Figure 4: Stereo PWM schematic
Now we have a digital signal that represents the baseband signal’s amplitude in its duty cycle. From this signal, we can drive an inverter topology of our choosing. In this case we’ve selected the H-bridge topology for higher power amplification for a given supply voltage. Before we can do this, we need complementary PWM signals with deadtime to drive the diagonal legs of the H-bridge. This is accomplished with a simple inverter and a FET based asymmetric RCD delay network fed into a comparator with hysteresis. Below shows such a circuit, its complement has an inverter at the input.

Lastly, the power stage of the amplifier. The H-bridge is constructed of 4 FETs controlled by PWM. For stereo audio we would require a total of 8 FETs! Some unique challenges are faced by the designer; particularly, driving the relatively large gate capacitances of the FETs.
at high frequencies and driving the floating gate of the high side FET. Information can be found at [2]. This is left to the reader to analyze and is out of the scope of this report. A single channel of our design can be seen on the next page.

Demodulation is accomplished by simply filtering off the carrier frequency. In our case it was recommended by [2] to use a second order Butterworth filter with only reactive components to minimize losses.

Additional electronics was needed for the operation of this all-in-one class-D amplifier, particularly the power supply designs, input protection and signal conditioning, jack detect, and DC blocking. Appended to the end of this report is the complete schematic. MATLAB scripts developed for the analysis of the design are also included.
3.2 SD Card Data Management

For the implementation of the music player we require a music source to play from. A significant amount of data storage is necessary for this project because we won’t have the means to decompress MPEG-3 encoded audio. Therefore we will be using raw 16-bit signed audio with a sample rate of 44.1 kHz. This will also put considerable strain on the communications interface and CPU when managing the immense amount of data.

The SD card interface will be broken down as follows:

- Hardware communications interface `spi.sv`
- Firmware driver for communications interface `sd_card.c`
- The main loop in `main.c`

The SD card used is the SDHC variant card which allows for capacities up to 32GB. We will be utilizing a micro SD card breakout board by Adafruit which will be powered and controlled by the DE0 Nano.

![Figure 8: SD card circuit schematic](image)

Below we will detail the requirements and design steps needed to realize the above components of the SD card interface.
3.2.1 spi CPU Peripheral

SD cards can be communicated with using two protocols: a serial peripheral interface (SPI) and a proprietary 5 wire synchronous communications interface. We will be using SPI since the latter requires licensing from the SD Association. From [3], the SD card requires the following hardware parameters:

- Variable SCK control
- 8-bit data packets
- Type 0 SPI mode
- Send and receive capability
- Busy status flag
- CPU integratable

Because we have incorporated the Nios II softcore processor in our design, it allows us to have direct control over our spi peripheral by connecting control registers to the Avalon MM bus. Given the CPU’s ability to read/write to data registers, we can set status flags for the CPU to check and data/control registers to operate the spi peripheral.

The entire spi peripheral can be broken down into 3 states represented below as a state diagram:

![State-machine for spi peripheral](image)

Figure 9: State-machine for spi peripheral
At start-up the module enters the init state which initializes all the design registers and settings to a default value. This involves setting a clk divider to 2 (max sck speed) and pre-loading the corresponding counters to drive sck.

Additionally, any state (including init) will transition to init when the reset signal is asserted. After a single clk cycle the state machine transitions to idle where the peripheral is continuously loading valid count and I/O pin states, as well as handling CPU requests to read/write data to its registers.

When the CPU requests to write data to the rcv_buf the next state becomes snd_rcv. This state pulls the busy flag high and sends/receives data through the mosi and miso signals. This is accomplished using a shift register designed using pointer arithmetic in hardware. When exactly 8 falling edges of sck have been sent over mosi, the next state becomes idle and the busy flag is deasserted.

3.3 sdc Driver for SD Card Using the spi Peripheral

To simplify the configuration of the spi peripheral and initialization process of the SD card, we opted for a firmware solution for this task. We've developed an API for the SD card constituting the following functions:

- bool sdc.init();
- bool baud_set(uint8_t div);
- bool sdc.read(uint32_t addr, uint8_t len, uint8_t* const buf);
- bool sdc_send_cmd(const uint8_t cmd, const uint32_t payload, const uint8_t resp, uint8_t* const resp_buf);

The baud_set() function takes an argument div and simply checks whether the spi interface is busy by parsing the busy flag, if not busy the CPU writes div to the divider register in the spi peripheral.

The sdc_send_cmd() follows a command and response sequence specified by the SD card specification found in [3]. It starts by pulling n_ss line low and sends a command byte, a 4 bytes payload, and a CRC byte over mosi. The CRC byte is optional in SPI mode according to [3]. This function then commands the spi peripheral to send dummy bytes (0xFF) to shift data in from the SD card over miso. The beginning of a response is indicated by a start bit. Once a response is received, it checks if the command was accepted and if the SD card is functioning properly.

The SD card always responds to a command with a defined response which can vary between different commands. The exact expected response has to be specified by the caller but could conceivably be decoded with a simple lookup table.
The initialization function `sdc_init()` initializes the sd card following the procedure outlined by [3]. It utilizes the `sdc_send_cmd()` and `baud_set()` functions to operate.

Finally the `sdc_read()` function is used to read a set amount of blocks from the SD card. It takes an address, expected length of response in blocks and a pointer to a buffer to put the data into. This function utilizes `sdc_send_cmd()` to initialize the data read then iterates a loop that loads the response into `buf`. We will be transferring the contents of `buf` into the dac ring buffer.
3.4 keypad CPU Peripheral

We required a means for the user to control audio playback. We designed a keypad module that detects a keypress and decodes the value and stores it into a register that can be read by the CPU. The internal circuitry of the keypad is a 4x4 matrix of normally open switches with each row and column connected to a pin header. We set up the DE0 to output to the rows and have the columns connected as inputs with pull-up resistors enabled. To decode a keypress we simply pull a row low and check the states of the columns. If any of the columns are low, continuity to ground has been made and the exact button pressed can be determined from the low row/column combination.

To ensure reliability a hardware debouncer was setup to store the decoded value on the positive edge of clk immediately after a keypress is detected then wait 100 ms before checking again. The state machine is illustrated below:

![State-machine for the keypad peripheral](image)

Figure 10: State-machine for the keypad peripheral

3.5 keypad Driver

A simple API was created for the keypad module, the keypad_get_button() function, which parses the register that stores the latest keypress and returns an integer corresponding to the key pressed. An enumeration is defined in a header file that details the return value. This module is designed such that the main program should constantly be polling with this function to detect any changes in keypress states. The physical hardware configuration is as simple as connecting the 8 pin headers to the keypad board.
3.6 ledc CPU Peripheral

The ledc is the LED controller peripheral. Initially it was designed to control the LEDs on the DE0 Nano board for debugging use, however later the ledc was extended to include GPIO functionality. The GPIO functionality made debugging much easier by allowing us to trigger an oscilloscope on events originating in firmware, and allowing us to determine the execution time of firmware subroutines.

3.7 Firmware

The firmware for the music player consisted of an initialization routine and a main program loop that checks the keypad for events, transfers audio data to the dac if there is room and handles switching to the next track if the currently playing track has reached its end.
3.7.1 SD Card Format

After the music has been prepared as raw audio files, a Python script running on a Linux PC is used to prepare the SD card image. At the beginning of the image, in the first block (512
bytes), a table exists that contains information for each of the music tracks. For each track two pieces of information is recorded in the table: the file size in bytes, and which 512-byte block the track starts in. Finally, the Unix utility \texttt{dd} is used to write the image to the SD card.

Each track starts on its own block. The space in between tracks is padded with zeros until the next track starts on a 512-byte block boundary. Below is an example of the SD card layout for two tracks.

<table>
<thead>
<tr>
<th>Byte address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>Track 0 start block number</td>
</tr>
<tr>
<td>0x0004</td>
<td>Track 0 size</td>
</tr>
<tr>
<td>0x0008</td>
<td>Track 1 start block number</td>
</tr>
<tr>
<td>0x000c</td>
<td>Track 1 size</td>
</tr>
<tr>
<td>0x0010</td>
<td>Zeros</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x0200</td>
<td>Beginning of track 1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x29f4</td>
<td>Last byte of track 1</td>
</tr>
<tr>
<td>0x29f8</td>
<td>Zeros</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x2a00</td>
<td>Beginning of track 2</td>
</tr>
</tbody>
</table>

### 3.7.2 FAT32 Driver

A basic read-only FAT32 implementation was created but due to time constraints it was not integrated into the project. It is capable of getting a directory listing of the root directory and reading files in the root directory. It is a primitive driver and is not optimized for performance. The source code listing is attached in the appendice section.

An important performance improvement that could be made for the FAT32 driver is to store the FAT in RAM instead of constantly re-reading the FAT to find out where the next sector of a file is.
4 Challenges

4.1 dac Ring Buffer

During initial debugging, we experimented with changing the dac’s FIFO buffer size. To do this, two things must be changed: the size of the dac buffer, and the limits of the read and write pointers for managing the ring buffer. In increasing the size of the dac buffer, we had to increase the limits of the read and write pointers. Once the initial issues had been resolved, we reduced the dac buffer size but forgot to change the limits of the read and write pointers.

The symptoms were that the audio would repeatedly click and stutter. This is because the write and read pointers that are supposed to index samples in the buffer were pointing beyond the limit of the buffer. Eventually the pointers would overflow and point within the buffer again and audio would resume properly.

To debug this issue, we created a GPIO module that allowed the CPU to toggle pins from firmware. By correlating the toggling of the pins with the audio glitches we were able to determine where the CPU was spending most of its time. This allowed us to rule out the SD card as the source of the glitches and revealed that the bug was in the dac module.

4.2 Keypad

During the initial bring-up phase of the keypad module we had much difficulty in getting all the rows to work on the keypad. After exhausting all other options over the course of several hours we finally decided to do a continuity test on the ribbon cable that connects the keypad to the FPGA. It turned out that one of the wires was broken. Continuity testing is important when using cheap ribbon cables.

4.3 SD Card

After all of our modules were designed and debugged, we encountered performance issues when reading information from the SD card. The maximum sck rate that we could reliably operate the SD card at was 12.5 MHz. Without a dedicated DMA module we optimized the firmware to improve efficiency on the firmware side. The minimum required baud rate needed to run 16-bit stereo audio is 1,411,200 bits per second. While this is significantly slower than SCK, the SD card latency coupled with ferrying uncompressed audio data with less than optimal code was introducing too much overhead. This was giving us a baud rate of 1,120,000 bits per second.

The solution was to simplify low level loops and pre-calculate constants used for the duration of the loop. Additionally, we increased the compiler optimization settings to maximum. With these changes the data rate exceeded 3,840,000 bits per second!
5 Conclusion

Overall, we saw many fine examples of digital system design. We learned more than expected. The sdc implementation proved to be the most complex module of the project. Overall, the project was a success and concluded with a working demonstration to the rest of the class.

Initially we were concerned about signal integrity issues surrounding the connection between the FPGA and the DAC and SD card. While error rates at 25 MHz were unacceptable, it not an issue at all at 12.5 MHz. Poor signal integrity would have limited our bitrate and potentially caused the whole project to fail if it was not fast enough. In our testing we did not detect any bit errors at this speed, although they could have been possible.

Recommendations for future work include finishing and validating the FAT32 filesystem implementation, implementing a real-time LCD-based frequency spectrum display, implementing an audio equalizer for boosting and attenuating frequency bands, and construct the class-D amplifier.
References


6 Appendice

6.1 QSYS Circuit Schematic
6.2 Class-D Amplifier Circuit Schematic
Figure 12: Class-D amplifier schematic pg. 1/6
Figure 13: Class-D amplifier schematic pg. 5/6
Figure 14: Class-D amplifier schematic pg. 3/6
Figure 15: Class-D amplifier schematic pg. 4/6
Figure 16: Class-D amplifier schematic pg. 5/6
Figure 17: Class-D amplifier schematic pg. 6/6
6.3 Software Listings

Listing 1: musicplayer_top.sv

```verilog
// Author: Preston Thompson & Nathaniel Rohrick
// Date: April 7, 2017

module musicplayer_top (
  input logic CLOCK_50,
  output logic [15:0] LED,
  input logic [1:0] KEY,
  input logic SDC_MISO,
  output logic SDC_SCK,
  output logic SDC_MOSI,
  output logic SDC_N_SS,
  output logic dac_dout,
  output logic dac_lrcout,
  output logic dac_sckout,
  output logic dac_bckout,
  (* altera_attribute = "-name WEAK_PULL_UP_RESISTOR ON" *)
  input logic [3:0] keypad_kpr,
  output logic [3:0] keypad_kpc
);

musicplayer u0 (  
  .clk_clk(CLOCK_50),
  .reset_reset_n(KEY[0]),
  .ledc_conn_leds(LED),
  .sdcs_conn_miso(SDC_MISO),
  .sdcs_conn_mosi(SDC_MOSI),
  .sdcs_conn_sck(SDC_SCK),
  .sdcs_conn_n_ss(SDC_N_SS),
  .dac_conn_dout(dac_dout),
  .dac_conn_lrcout(dac_lrcout),
  .dac_conn_sckout(dac_sckout),
  .dac_conn_bckout(dac_bckout),
  .keypad_conn_kpc(keypad_kpc),
  .keypad_conn_kpr(keypad_kpr)
);
endmodule
```

Listing 2: ledc.sv

```verilog
// Author: Preston Thompson
// Date: April 7, 2017

module ledc (  
  input logic reset,
  input logic clk,
  input logic avs_read,
  input logic avs_write,
  input logic [15:0] avs_writedata,
```
output logic [15:0] avs_readdata,

output logic [15:0] leds
);
always_ff @(posedge clk) begin
  if (reset) begin
    leds <= '0;
  end else begin
    if (avs_write) begin
      leds <= avs_writedata;
    end else if (avs_read) begin
      avs_readdata <= leds;
    end
  end
endmodule

Listing 3: dac.sv

// Author: Preston Thompson
// Date: April 7, 2017
module dac #(
  // 50 MHz / (15625/7056) = 22.5792 MHz
  parameter SCK_NUMERATOR = 16'd15625,
  parameter SCK_DENOMINATOR = 16'd7056,
  parameter BUFFER_SIZE = 2048
) (
  input logic reset,
  input logic clk,
  output logic dout,
  output logic lrcout,
  output logic sckout,
  output logic bckout,
  input logic avs_write,
  input logic avs_read,
  input logic [31:0] avs_writedata,
  output logic [31:0] avs_readdata,
  input logic [3:0] avs_address
);

logic [31:0] sample_buf[BUFFER_SIZE-1:0];
logic [10:0] sample_read_ptr;
logic [10:0] sample_write_ptr;
logic [15:0] sck_count;
logic [15:0] sck_count_next;
logic [2:0] bck_count;
logic [4:0] lrcout_count;
logic [31:0] sample;
logic [4:0] dout_ptr);
enum logic {increment, overflow} op;

always_ff @(posedge clk) begin

if (reset) begin
  sck_count <= '0;
bck_count <= '0;
lrcout_count <= '0;
lrcout <= '1;
sckout <= '0;
bckout <= '0;
dout_ptr <= '0;
sample_write_ptr <= '0;
sample_read_ptr <= '0;
end else begin

  // if the ring buffer is empty, output zeros and nop
if (sample_read_ptr != sample_write_ptr) begin
  dout <= sample[5'hf - dout_ptr];
sck_count <= sck_count_next;
end else begin
  dout <= '0;
end

  // handle any reads from the CPU
if (avs_read) begin
  unique case (avs_address)
  4'h0: begin
    avs_readdata[31:16] <= '0;
    avs_readdata[15:0] <= sample_read_ptr;
  end
  4'h1: begin
    avs_readdata[31:16] <= '0;
    avs_readdata[15:0] <= sample_write_ptr;
  end
  4'h2: begin
    avs_readdata <= sample_buf[sample_write_ptr];
  end
endcase

  // handle any writes from the CPU
else if (avs_write) begin
  unique case (avs_address)
  4'h0: begin
    sample_read_ptr <= avs_writedata[15:0];
  end
  4'h1: begin
    sample_write_ptr <= avs_writedata[15:0];
  end
  4'h2: begin

sample_buf[sample_write_ptr] <= avs_writedata;
sample_write_ptr <= sample_write_ptr + 16'd1;

end
endcase

// only interact with the DAC if there’s samples in the ring buffer
if (op == overflow && sample_read_ptr != sample_write_ptr) begin
  sckout <= ~sckout;
  bck_count <= bck_count + 3'b1;
  if (bck_count == '1) begin
    lrcout_count <= lrcout_count + 5'b1;
    bckout <= ~bckout;
    if (bckout) begin
      dout_ptr <= dout_ptr + 5'b1;
    end
    if (lrcout_count == '1) begin
      lrcout <= ~lrcout;
    end
  end

  if (dout_ptr == '1 &&
      bck_count == '1 &&
      op == overflow &&
      lrcout_count == '1
    ) begin
    sample_read_ptr <= sample_read_ptr + 16'd1;
    sample <= sample_buf[sample_read_ptr + 16'd1];
  end

end

always_comb begin
  if (sck_count >= (SCK_NUMERATOR - SCK_DENOMINATOR)) begin
    sck_count_next = sck_count - (SCK_NUMERATOR - SCK_DENOMINATOR);
    op = overflow;
  end else begin
    sck_count_next = sck_count + SCK_DENOMINATOR;
    op = increment;
  end
end
logic reset;
logic clk;

// HALF SCALE SINE
logic [15:0] lsamples [16] = '{
  16'd0, 16'd6122, 16'd11313, 16'd14782,
  16'd16000, 16'd14782, 16'd11313, 16'd6122,
  16'd0, -16'd6122, -16'd11313, -16'd14782,
  -16'd16000, -16'd14782, -16'd11313, -16'd6122
};
logic [15:0] rsamples [16] = '{
  16'd0, 16'd6122, 16'd11313, 16'd14782,
  16'd16000, 16'd14782, 16'd11313, 16'd6122,
  16'd0, -16'd6122, -16'd11313, -16'd14782,
  -16'd16000, -16'd14782, -16'd11313, -16'd6122
};

logic avs_write;
logic avs_read;
logic [31:0] avs_writedata;
logic [3:0] avs_address;

wire dout;
wire lrcout;
wire sckout;
wire bckout;
wire [31:0] avs_readdata;

dac dac_0;(*);

initial begin

  // initial state
  reset = 1;
  clk = 0;
  avs_write = 0;
  avs_read = 0;

  repeat (2) begin
    #10ns; clk = ~clk;
  end

  reset = 0;

  repeat (10) begin
    #10ns; clk = ~clk;
  end

  avs_address = 4'h8;
  avs_write = 1;
  for (int i = 0; i < 16; i++) begin
    avs_writedata = {lsamples[i], rsamples[i]};
    repeat (2) begin

module spi (
  input logic clk, // master clk
  input logic reset, // master reset
  input logic miso, // rx data from sd card
  input logic avs_write, // write flag from cpu
  input logic avs_read, // read flag from cpu
  input logic [2:0] avs_address, // cpu address bus
  input logic [31:0] avs_writedata, // cpu data output bus
  output logic [31:0] avs_readdata, // cpu data input bus
  output logic sck, // sd card clock
  output logic mosi, // tx data to sd card
  output logic n_ss // chip select
);

// Below shows the memory allocation of the registers needed by spi.sv
*/
ADDRESS BEGIN | ADDRESS END | REG NAME | DESCRIPTION
--------------------------------------------------------------------
0x0000 | 0x0000 | data | MOSI data buffer to be send out
0x0001 | 0x0001 | rcv_buf | MISO buffer to receive data
0x0002 | 0x0002 | div | sck frequency divider register
0x0003 | 0x0003 | busy | Tells spi caller if peripheral is busy
0x0004 | 0x0004 | n_ss | peripheral enable
--------------------------------------------------------------------
*|
localparam DATA_START = 8’h0;
localparam RCV_START = 8’h1;
localparam DIV_START = 8’h2;
localparam BUSY_START = 8’h3;
localparam SS_START = 8’h4;

// CPU visible registers
logic [7:0] data;     // data to send to slave
logic [7:0] rcv_buf;  // rcv buffer that holds data from send
logic [7:0] div;      // frequency divider register
logic busy;          // busy flag (1 when busy, 0 otherwise)

// End memory register description

localparam BYTE = 16'd8;    // size of a byte

enum {init, idle, snd_rcv} state, state_next;

logic [15:0] sck_freq_cnt;  // loaded with number of clk cycles
logic [15:0] sck_cnt;       // loaded with number of sck
logic [7:0] mosi_ptr;       // pointer to select cmd_buf bit
logic [7:0] miso_ptr;       // pointer to select cmd_buf byte
logic [7:0] divider;       // variable to make div the actual clock divider rate

assign divider = div >> 1'd1; // divide by 2 for usability on the firmware side

// state exit conditions
// allow data to latch on falling edge
assign SND_RCV_EXIT = (sck_cnt == 16'd0) && (sck == 0);
assign IDLE_EXIT = (avs_write && (avs_address == DATA_START));

always_ff @(posedge clk) begin
    if (avs_read && (avs_address == BUSY_START)) begin
        avs_readdata[0] <= busy;
        avs_readdata[31:1] <= '0;
    end

    state <= state_next;

    if (state_next == init)
        busy <= 1;
    else if (state_next == idle)
        busy <= 0;
    else if (state_next == snd_rcv)
        busy <= 1;

    unique case (state)
      init: begin
        div <= 8'd1;        // default clk divide is (2/2)*2 = 2
        sck_cnt <= 16'd8;   // load with number of sck cycles
        sck_freq_cnt <= divider; // load with number of clk cycles
        mosi_ptr <= 8'h7;   // get mosi_ptr looking at MSb of data
        miso_ptr <= 8'h7;   // get miso_ptr looking at MSb of rcv_buf
        mosi <= 1'd1;
        sck <= 1'd0;
      end
endcase
mosi <= 1'd1; //default MOSI state is high
sck <= 1'd0; //default SCK state is high
n_ss <= 1'd1; //de-select chip
end

idle: begin
  //if CPU requests to read/write, allow to do so in idle
  if (avs_write && (avs_address == DATA_START))
    data <= avs_writedata[7:0];
  if (avs_read && (avs_address == RCV_START)) begin
    avs_readdata[31:8] <= 24'd0;
    avs_readdata[7:0] <= rcv_buf;
  end else if (avs_read && (avs_address == DIV_START)) begin
    avs_readdata[31:8] <= 24'd0;
    avs_readdata[7:0] <= div;
  end
  if (avs_write && (avs_address == DIV_START))
    div <= avs_writedata[7:0];
  if (avs_write && (avs_address == SS_START))
    n_ss <= avs_writedata[0];
end

snd_rcv: begin
  if (sck_freq_cnt == 16'd0) begin
    sck_freq_cnt <= divider;
    sck <= ~sck;
  end else begin
    sck_freq_cnt <= sck_freq_cnt - 16'd1;
  end
  if ((sck == 1'd0) && (sck_freq_cnt == 16'd0)) begin // latch on rising edge
    sck_cnt <= sck_cnt - 16'd1;
    rcv_buf[miso_ptr] <= miso;
    miso_ptr <= miso_ptr - 8'd1;
    mosi_ptr <= mosi_ptr - 8'd1;
  end
  if (sck && (sck_freq_cnt == 16'd0)) begin // shift on falling edge
    mosi <= data[mosi_ptr];
    // put valid data on MOSI (SPI mode 0)
  end else if ((sck_cnt == 16'd0) && (sck_freq_cnt == divider)) begin
    mosi <= data[mosi_ptr];
  end
endcase
end

//bring system to next state
always_comb begin
    case (state)
        init: begin
            if (reset)
                state_next <= init;
            else
                state_next <= idle;
        end
        idle: begin
            if (reset)
                state_next <= init;
            else if (IDLE_EXIT)
                state_next <= snd_rcv;
            else
                state_next <= idle;
        end
        snd_rcv: begin
            if (reset)
                state_next <= init;
            else if (SND_RCV_EXIT)
                state_next = idle;
            else
                state_next = snd_rcv;
        end
    endcase
end
endmodule

---

Listing 6: spi_tb.sv

```vhdl
// Author: Preston Thompson
// Date: April 7, 2017
module spi_tb();

logic clk;
logic reset;

logic avs_write;
logic avs_read;
logic [2:0] avs_address;
logic [31:0] avs_writedata;
wire [31:0] avs_readdata;
wire sck;
wire mosi;
wire n_ss;
spi DUT(.*);
assign miso = mosi;
```
initial begin

    // initial state
    reset = 1;
    clk = 0;
    avs_write = 0;
    avs_read = 0;

    // perform a reset
    repeat(2) begin
        #10ns; clk = ~clk;
    end

    reset = 0;

    // idle for a bit
    repeat (10) begin
        #10ns; clk = ~clk;
    end

    // write to baud div register
    avs_address = 3'h2;
    avs_write = 1;
    avs_writedata = 32'd4;
    repeat (2) begin
        #10ns; clk = ~clk;
    end
    avs_write = 0;

    // idle for a bit
    repeat (10) begin
        #10ns; clk = ~clk;
    end

    // deassert ss
    avs_address = 3'h4;
    avs_write = 1;
    avs_writedata = 32'd1;
    repeat (2) begin
        #10ns; clk = ~clk;
    end
    avs_write = 0;

    // idle for a bit
    repeat (10) begin
        #10ns; clk = ~clk;
    end

    // assert ss
    avs_address = 3'h4;
    avs_write = 1;
    avs_writedata = 32'd0;
    repeat (2) begin

#10ns; clk = ~clk;
end
avs_write = 0;

// idle for a bit
repeat (10) begin
    #10ns; clk = ~clk;
end

// write a byte
avs_address = 3’h0;
avs_write = 1;
avs_writedata = 32’ha5;
repeat (2) begin
    #10ns; clk = ~clk;
end
avs_write = 0;

// many cycles
repeat (100) begin
    #10ns; clk = ~clk;
end

// read the busy bit
avs_address = 3’h3;
avs_read = 1;
repeat (2) begin
    #10ns; clk = ~clk;
end
avs_read = 0;

// idle for a bit
repeat (10) begin
    #10ns; clk = ~clk;
end

endmodule

Listing 7: sdc.c

//Author: Nathaniel Rohrick
//Date: April 7, 2017

#include "../musicplayer_bsp/system.h"
#include <stdbool.h>
#include <stdint.h>
#include "sdc_cmd.h"
#include "sdc.h"

static volatile const uint8_t* const spi_sdc =
(volatile uint8_t*)SDC_SPI_0_BASE;
static volatile uint8_t* const spi_data =
  (volatile uint8_t*)(SDC_SPI_0_BASE + 0x00);
static volatile uint8_t* const spi_rcvBuf =
  (volatile uint8_t*)(SDC_SPI_0_BASE + 0x04);
static volatile uint8_t* const spi_div =
  (volatile uint8_t*)(SDC_SPI_0_BASE + 0x08);
static volatile uint8_t* const spi_n_ready =
  (volatile uint8_t*)(SDC_SPI_0_BASE + 0x0C);
static volatile uint8_t* const spi_n_ss =
  (volatile uint8_t*)(SDC_SPI_0_BASE + 0x10);
static volatile uint8_t* const ledc =
  (volatile uint8_t*)LEDC_0_BASE;

static const uint32_t WORD_LEN = 32;
static const uint8_t STARTb = 0x40;  // dummy byte, all 1's
static const uint8_t idleb = 0x01;  // idle bit
static const uint8_t respb = 0xFE;  // response good byte
static const uint32_t CLK = 50000000; // clk rate of fpga

// Function: sdInit
// Arguments: None
// Return: true when error
// Description: Initializes an spi communications interface with an external
// sd card. Puts sd card into spi mode and sets clock divider to default 2.
bool sdc_init ()
{
  uint8_t resp_buf[5];

  for (int i = 0; i < 50000; i++); // delay 1ms
    if (baud_set(126)) // set sck to 50M/126~400kHz
      return true;

  while (*spi_n_ready);
  *spi_n_ss = 1; // deassert SS
  for (int i = 0; i < 50000; i++); // delay 1ms
  for (int i = 0; i < 200; i++)
  {
    while (*spi_n_ready);
    *spi_data = dByte;
  }

  for (int i = 0; i < 50000; i++); // delay 1ms
  if (sdc_send_cmd(CMD0, 0x0, R1, resp_buf)) // send CMDO
    return true;

  if (sdc_send_cmd(CMD8, 0x1AA, R7, resp_buf))
    return true;

  while (*spi_n_ready);
if (*resp_buf > 1)
  return true;

for (int i = 0; i <= TIMEOUT; i++)
{
  if (sdc_send_cmd(CMD55, 0x0, R1, resp_buf))
    return true;

while (*spi_n_ready);
  if (*resp_buf > 1)
    continue;
  if (sdc_send_cmd(CMD41, 0x40000000, R1, resp_buf))
    return true;
  if (*resp_buf > 0)
    continue;
  else
    break;
}
sdc_send_cmd(CMD16, BLOCK_SIZE, R1, resp_buf);
while (*spi_n_ready);
if (*resp_buf > 0)
  return true;
return (baud_set(2));

// Function: baud_set
// Arguments: uint8 div divides clk by a value between 2 and 255
// Return: True with error
// Description: Sets clk divider to generate sck, the spi clock
bool baud_set (uint8_t div)
{
  while (*spi_n_ready);
  *spi_div = div;
  return false;
}

// Function: sdc_send_cmd
// Arguments: takes a command number and a corresponding payload
// and an expected response type in resp. It loads the response
// into resp_buf.
// Return: True when error
// Description: Sends a command to SD card and loads a buffer with
// the response.
bool sdc_send_cmd(
  const uint8_t cmd,
  const uint32_t payload,
  const uint8_t resp,
  uint8_t* resp_buf
)
{
  uint16_t spi_timeout;
  while (*spi_n_ready);
*spi_n_ss = 0; // assert SS
while (*spi_n_ready);
*spi_data = dByte;
//send cmd index
while (*spi_n_ready);
*spi_data = cmd | STARTb;
//send payload MSB first
for (int i = 3; i >= 0; i--)
{
    while (*spi_n_ready);
    *spi_data = (payload >> (i*8));
}
//send CRC
while (*spi_n_ready);
if (resp == R1 || resp == R1b)
    *spi_data = DEF_CRC;
else if (resp == R7)
    *spi_data = CMD8_CRC;
while (*spi_n_ready);
*spi_data = dByte;
for (spi_timeout = 0; spi_timeout < TIMEOUT; spi_timeout++)
{
    while (*spi_n_ready);
    if (*spi_rcvBuf & (1 << 7))
        *spi_data = dByte; // send another byte
    else if (resp == R1)
    {
        *resp_buf = *spi_rcvBuf;
        break;
    }
    else if (resp == R7)
    {
        *resp_buf = *spi_rcvBuf;
        for (int i = 1; i < 5; i++)
        {
            while (*spi_n_ready);
            *spi_data = dByte;
            *(resp_buf + i) = *spi_rcvBuf;
        }
        break;
    }
    else if (resp == R1b)
    {
        do
        {
            while (*spi_n_ready);
            *spi_data = dByte;
            while (*spi_n_ready);
        }
while(*spi_rcvBuf != 0xFF);
break;
}

if (((cmd == CMD17 )||(cmd == CMD18))&&(spi_timeout != TIMEOUT))
    return false;

while (*spi_n_ready);
*spi_n_ss = 1;//deassert SS

//if (spi_timeout == TIMEOUT)
//return true;
while (*spi_n_ready);
*spi_data = dByte;
while (*spi_n_ready);
*spi_data = dByte;
return false;
}

bool sdc_read(
  uint32_t addr,
  uint8_t len,
  uint8_t* buf
)
{
  uint8_t resp_buf[8];
  if (sdc_send_cmd (CMD18 , addr , R1 , resp_buf ))
      return true;

  //if (resp_buf[0] != 0)
  //return true;
  while (*spi_n_ready);
  *spi_data = dByte;
  for (int iblock = 0; iblock < len; iblock++)
  {
    for (int i = 0; i <= TIMEOUT ; i++)
    {
      while (*spi_n_ready);
      if (*spi_rcvBuf == 0xFE)
          break;
      else
      {
        while (*spi_n_ready);
        *spi_data = dByte;
      }
      //if (i == TIMEOUT)
      //return true;
    }
  }
//unsigned block_offs = iblock * BLOCK_SIZE;
uint8_t *end = buf + BLOCK_SIZE;
for (; buf < end; buf++)//work starts here
{
    while (*spi_n_ready);
    *spi_data = dByte;
    while (*spi_n_ready);
    *buf = *spi_rcvBuf;
    //buf[i + block_offs] = *spi_rcvBuf;
    //*buf++ = *spi_rcvBuf;
}
while (*spi_n_ready);
*spi_data = dByte;
while (*spi_n_ready);
*spi_data = dByte;

if (sdc_send_cmd(CMD12, 0x0, R1b, resp_buf)) //stop transmission
    return true;

    //if (resp_buf[0] != 0)
    //return true;

    //while (*spi_n_ready);
    //*spi_n_ss = 1;//deassert SS
return false;

Listing 8: sdc.h

Listing 9: sdc_cmd.h

// Author: Nathaniel Rohrick
// Date: April 7, 2017

#ifndef _SDC_H
#define _SDC_H

#include <stdbool.h>
#include "system.h"

bool baud_set (uint8_t div);
bool sdc_init ();
bool sdc_read(uint32_t addr, uint8_t len, uint8_t* buf);
bool sdc_send_cmd(const uint8_t cmd, const uint32_t payload,
    const uint8_t resp, uint8_t* const resp_buf);
#define BLOCK_SIZE 512

#endif

Listing 8: sdc.h

Listing 9: sdc_cmd.h
/*
Command breakdown
--------------------------------
B1 | B1 - B5 | B6
--------------------------------
index | payload | CRC
--------------------------------
*/
// name and payload for commands, use 0x95 for CRC (not used)

#define CMD0 0
#define CMD1 1
#define CMD8 8
#define CMD9 9
#define CMD10 10
#define CMD12 12
#define CMD16 16
#define CMD17 17
#define CMD18 18
#define CMD23 23
#define CMD24 24
#define CMD25 25
#define CMD26 26
#define CMD41 41
#define CMD55 55
#define CMD58 58
#define CMD8_CRC 0x87
#define DEF_CRC 0x95
#define R1 1
#define R7 2
#define R1b 3
#define TIMEOUT 100000

Listing 10: keypad.sv

/*
Command breakdown
--------------------------------
B1 | B1 - B5 | B6
--------------------------------
index | payload | CRC
--------------------------------
*/
// name and payload for commands, use 0x95 for CRC (not used)

#define CMD0 0
#define CMD1 1
#define CMD8 8
#define CMD9 9
#define CMD10 10
#define CMD12 12
#define CMD16 16
#define CMD17 17
#define CMD18 18
#define CMD23 23
#define CMD24 24
#define CMD25 25
#define CMD26 26
#define CMD41 41
#define CMD55 55
#define CMD58 58
#define CMD8_CRC 0x87
#define DEF_CRC 0x95
#define R1 1
#define R7 2
#define R1b 3
#define TIMEOUT 100000

Listing 10: keypad.sv

module keypad(
    input logic [3:0] kpr,       // row vector on keypad
    input logic reset,           // master reset
    input logic clk,             // master clk
    input logic avs_write,       // write flag from cpu
    input logic avs_read,        // read flag from cpu
    input logic [1:0] avs_address,  // cpu address bus
    input logic [31:0] avs_writedata, // cpu data output bus
    output logic [31:0] avs_readdata, // cpu data input bus
    output logic [3:0] kpc       // column vector on keypad
);
// localparam KPHIT_START = 3'd0; // start address for keypad hit flag
localparam BUTTON_START = 1'd0; // start address for decoded keypress register

localparam WAIT = 22'hFFFFF; // ~100ms delay for debouncing
localparam DIV = 16'd62500; // clk divider for polling CLK/(4*DIV)

logic kphit; // high when keypad is pressed
logic [3:0] kpr_reg;
logic [21:0] dbnc_cnt; // debounce counter
logic [15:0] clk_cnt; // clk frequency counter

denum {blank, next, v_up, v_down, mute, prev} button; // button state
denum {idle, active, rst} state, state_next; // state machine variables

// exit conditions
assign IDLE_EXIT = kphit;
assign ACTIVE_EXIT = !kphit && (dbnc_cnt == '0);

// datapath
always_ff@(posedge clk) begin
    state <= state_next;
    kpr_reg <= kpr;
    if (clk_cnt == '0)
        clk_cnt <= DIV;
    else
        clk_cnt <= clk_cnt - 'd1;
end

// always handle requests for keypress info
if (avs_read && (avs_address == BUTTON_START)) begin
    avs_readdata[31:3] <= '0;
    avs_readdata[2:0] <= button;
end

case (state)
    rst: begin // load default states
        kpc <= 4'b0111;
        dbnc_cnt <= WAIT;
    end
    idle: begin
        dbnc_cnt <= WAIT; // prep debounce counter
        if (state_next == idle) begin // if key is pressed, only 1 to decode
            if ((kpr_reg == 4'b1111) && (clk_cnt == 0)) begin
                unique case (kpc)
                    4'b0111: kpc <= 4'b1011;
                    4'b1011: kpc <= 4'b1101;
                    4'b1101: kpc <= 4'b1110;
                    4'b1110: kpc <= 4'b0111;
                endcase
            end
            if ((kpr_reg == 4'b1110) && (kpc == 4'b1110)) begin
                button <= blank;
            end
    end
endcase
kphit <= '1;
else if ((kpr_reg == 4'b1110) && (kpc == 4'b1101)) begin
    button <= blank;
    kphit <= '1;
end else if ((kpr_reg == 4'b1110) && (kpc == 4'b1011)) begin
    button <= blank;
    kphit <= '1;
end else if ((kpr_reg == 4'b1110) && (kpc == 4'b0111)) begin
    button <= blank;
    kphit <= '1;
end else if ((kpr_reg == 4'b1011) && (kpc == 4'b1110)) begin
    button <= blank;
    kphit <= '1;
end else if ((kpr_reg == 4'b1011) && (kpc == 4'b1101)) begin
    button <= blank;
    kphit <= '1;
end else if ((kpr_reg == 4'b1011) && (kpc == 4'b0111)) begin
    button <= blank;
    kphit <= '1;
end else if ((kpr_reg == 4'b1011) && (kpc == 4'b1011)) begin
    button <= blank;
    kphit <= '1;
end else if ((kpr_reg == 4'b1011) && (kpc == 4'b0111)) begin
    button <= blank;
    kphit <= '1;
end else if ((kpr_reg == 4'b1011) && (kpc == 4'b0111)) begin
    button <= blank;
    kphit <= '1;
end else if ((kpr_reg == 4'b0111) && (kpc == 4'b1110)) begin
    button <= blank;
    kphit <= '1;
end else if ((kpr_reg == 4'b0111) && (kpc == 4'b1101)) begin
    button <= blank;
    kphit <= '1;
end else if ((kpr_reg == 4'b0111) && (kpc == 4'b0111)) begin
    button <= blank;
    kphit <= '1;
end else if ((kpr_reg == 4'b0111) && (kpc == 4'b0111)) begin
    button <= blank;
    kphit <= '1;
end else if ((kpr_reg == 4'b0111) && (kpc == 4'b1011)) begin
    button <= blank;
    kphit <= '1;
end else if ((kpr_reg == 4'b0111) && (kpc == 4'b1011)) begin
    button <= blank;
    kphit <= '1;
end else if ((kpr_reg == 4'b0111) && (kpc == 4'b0111)) begin
    button <= blank;
    kphit <= '1;
end else begin
    button <= blank;
end
end

active: begin
dbnc_cnt <= dbnc_cnt - 'd1; // counts down
  if (kpr_reg == 4'b1111)
    kphit <= 0;
  end
endcase

// control path
always_comb begin
  case(state)
  rst: begin
    if (reset)
      state_next = rst;
    else
      state_next = idle;
  end

  idle: begin
    if (reset)
      state_next = rst;
    else if (IDLE_EXIT)
      state_next = active;
    else
      state_next = idle;
  end

  active: begin
    if (reset)
      state_next = rst;
    else if (ACTIVE_EXIT)
      state_next = idle;
    else
      state_next = active;
  end
  endcase
end
endmodule

Listing 11: keypad.c

// Author: Nathaniel Rohrick
// Date: April 7, 2017

#include <stdint.h>

#include "../musicplayer_bsp/system.h"  // defines mapping memory of project
#include "keypad.h"

static volatile uint32_t * const keypad_button =
  (volatile uint32_t*)(KEYPAD_0_BASE);

int keypad_get_button(void)
{
  return *keypad_button;
Listing 12: keypad.h

// Author: Nathaniel Rohrick
// Date: April 7, 2017

#ifndef _KEYPAD_H
#define _KEYPAD_H

enum Keypad.Buttons {
    key_blank,
    key_next,
    key_v_up,
    key_v_down,
    key_mute,
    key_prev
};

int keypad_get_button(void);

#endif

Listing 13: main.c

// Author: Preston Thompson
// Date: April 7, 2017

#include <stdint.h>
#include "system.h"
#include "sdc.h"
#include "sdc_cmd.h"
#include "keypad.h"

#define DAC_BUFFER_SIZE 2048

static volatile uint16_t * const ledc =
    (volatile uint16_t *) LEDC_0_BASE;
static volatile uint32_t * const dac_sample_read_ptr =
    (volatile uint32_t *) DAC_0_BASE;
static volatile uint32_t * const dac_sample_write_ptr =
    (volatile uint32_t *)(DAC_0_BASE + 0x4);
static volatile uint32_t * const dac_sample =
    (volatile uint32_t *)(DAC_0_BASE + 0x8);

struct track_info {
    uint32_t start;
    uint32_t size;
};

static struct track_info tracks[64];
static uint8_t buf[512];
static int current_track;
static uint32_t current_sector;
static uint32_t last_sector;
static int num_tracks;

int get_num_samples(void) {
    volatile uint32_t write_ptr = *dac_sample_write_ptr;
    volatile uint32_t read_ptr = *dac_sample_read_ptr;
    if (write_ptr >= read_ptr)
        return write_ptr - read_ptr;
    else
        return (DAC_BUFFER_SIZE - read_ptr) + write_ptr;
}

void change_track(void) {
    *ledc = current_track;
    current_sector = tracks[current_track].start;
    if (current_track != num_tracks - 1)
        last_sector = tracks[current_track+1].start - 1;
    else
        last_sector = tracks[current_track].start + tracks[current_track].size/512;
    sdc_read(current_sector, 1, buf);
    while (keypad_get_button() != key_blank);
}

int main(void) {
    int i;
    *ledc = 0xff;
    sdc_init();
    sdc_read(0, 1, (uint8_t *)tracks);
    current_track = 0;
    for (i = 0; i < 64; i++) {
        if (tracks[i].size > 0)
            num_tracks++;
    }
    change_track();
    while (1) {
        int current_key = keypad_get_button();
        if (current_key == key_next) {
            current_track++;
            if (current_track == num_tracks)
                current_track = 0;
            change_track();
            continue;
        }
        else if (current_key == key_prev) {
            current_track--;
            if (current_track < 0)
current_track = num_tracks - 1;
change_track();
continue;
}
else if (current_sector == last_sector) {
    int dac_room = DAC_BUFFER_SIZE - get_num_samples();
    if (dac_room >= (tracks[current_track].size % 512)/4) {
        for (i = 0; i < tracks[current_track].size % 512; i+= 4)
            *dac_sample = *(uint32_t*)(buf + i);
        if (current_track == num_tracks - 1)
            current_track = 0;
        else
            current_track++; 
        change_track();
        sdc_read(current_sector, 1, buf);
    }
}
else {
    int dac_room = DAC_BUFFER_SIZE - get_num_samples();
    if (dac_room >= 128) {
        for (i = 0; i < 512; i += 4)
            *dac_sample = *(uint32_t*)(buf + i);
        current_sector++;
        sdc_read(current_sector, 1, buf);
    }
}
return 0;

Listing 14: fat.c

// Author: Preston Thompson
// Date: April 7, 2017

// This is an incomplete FAT32 driver.

#include <stddef.h>
#include <stdint.h>

#include "fat.h"
#include "sdc.h"

struct Partition_Desc {
    uint8_t type_code;
    uint32_t lba_begin;
};

struct Volume_ID {
    uint16_t BytsPerSec;
    uint8_t SecPerClus;
    uint16_t RsvdSecCnt;
    uint8_t NumFATs;

uint32_t FATSz32;
uint32_t RootClus;
);

static uint8_t sector_buf[512];
static struct Partition_Desc pd;
static struct Volume_ID vid;
static uint32_t cluster_begin_lba;
static uint32_t fat_begin_lba;

static void read_word(const uint8_t *src, void *dst, size_t len) {
  size_t i;
  for (i = 0; i < len; i++)
    *(uint8_t*)(dst++) = *src++;
}

void fat_init(void) {
  sdc_read(0, 1, sector_buf);
  read_word(sector_buf + 446 + 0x04, &pd.type_code, 1);
  read_word(sector_buf + 446 + 0x08, &pd.lba_begin, 4);
  sdc_read(pd.lba_begin, 1, sector_buf);
  read_word(sector_buf + 0x0b, &vid.BytesPerSec, 2);
  read_word(sector_buf + 0x0d, &vid.SecPerClus, 1);
  read_word(sector_buf + 0x0e, &vid.ResvdSecCnt, 2);
  read_word(sector_buf + 0x10, &vid.NumFATs, 1);
  read_word(sector_buf + 0x24, &vid.FATSz32, 4);
  read_word(sector_buf + 0x2c, &vid.RootClus, 4);
  fat_begin_lba = pd.lba_begin + vid.ResvdSecCnt;
  cluster_begin_lba = fat_begin_lba + (vid.NumFATs * vid.FATSz32);
}

int fat_get_file_info(uint32_t index, struct File_Info *fi) {
  uint8_t record[32];
  uint16_t first_cluster_hi, first_cluster_lo;
  uint32_t lba_addr;
  lba_addr = cluster_begin_lba + (index / (vid.BytesPerSec / 32));
  sdc_read(lba_addr, 1, sector_buf);
  read_word(sector_buf + index * 32, record, 32);
  if (record[0] == 0)
    return FAT_END_OF_FILES;
  if (((record[0x0b] & 0xF) == 0xF)
    return FAT_LONG_FILENAME;
  read_word(record, &fi->name, 11);
  fi->name[11] = '\0';
  fi->is_dir = (record[11] & (1 << 4)) ? 1 : 0;
  read_word(record + 0x1a, &first_cluster_lo, 2);
read_word(record + 0x14, &first_cluster_hi, 2);
fi->first_cluster = first_cluster_lo | (first_cluster_hi << 16);
read_word(record + 0x1c, &fi->size, 4);
return FAT_SUCCESS;
}

static uint32_t lba_addr_from_cluster(uint32_t cluster) {
  return (cluster - vid.RootClus) * vid.SecPerClus + cluster_begin_lba;
}

size_t fat_read_file(uint32_t index, uint32_t sector, uint8_t *dst) {
  uint32_t lba_addr, cluster_num, cluster, i;
  struct File_Info fi;
  if (fat_get_file_info(index, &fi) != FAT_SUCCESS)
    return 0;
  cluster_num = sector / vid.SecPerClus;
  cluster = fi.first_cluster;
  for (i = 0; i < cluster_num; i++) {
    sdc_read(fat_begin_lba + (cluster >> 7), 1, sector_buf);
    read_word(sector_buf + (cluster & 0x7f) * 4, &cluster, 4);
    if (cluster >= 0xFFFFFFFF)
      return 0;
  }
  lba_addr = lba_addr_from_cluster(cluster) + (sector % vid.SecPerClus);
  sdc_read(lba_addr, 1, dst);
  read_word(sector_buf + (cluster & 0x7f) * 4, &cluster, 4);
  if (cluster >= 0xFFFFFFFF)
    return fi.size % BLOCK_SIZE;
  else
    return BLOCK_SIZE;
}
```c
#define FAT_LONG_FILENAME 2

struct File_Info {
    char name[12];
    uint8_t is_dir;
    uint32_t first_cluster;
    uint32_t size;
};

void fat_init(void);
int fat_get_file_info(uint32_t index, struct File_Info *fi);
size_t fat_read_file(uint32_t index, uint32_t sector, uint8_t *dst);

#endif

Listing 16:(deadtime.m)
clc;  
clear;  
fp = fopen('deadtime.txt', 'w');  
s = tf('s');  
R_low = 100;  
R_high = 1000;  
C = 100e-12;  
H = (1/(R_low*C))/(s+(1/(R_low*C)));  
S_min = stepinfo(H,'RiseTimeLimits',[0.00,0.52]);  
S_max = stepinfo(H,'RiseTimeLimits',[0.00,0.68]);  
S_typ = stepinfo(H,'RiseTimeLimits',[0.00,0.60]);  
fprintf(fp,'Given:
R_low : %.3f Ohms 
R_max = %.3f Ohms 
C = %.3 f pF 
H = (1/(R_low*C))/(s+(1/(R_low*C)));
S_min = stepinfo(H,'RiseTimeLimits',[0.00,0.52]);
S_max = stepinfo(H,'RiseTimeLimits',[0.00,0.68]);
S_typ = stepinfo(H,'RiseTimeLimits',[0.00,0.60]);
fprintf(fp,'Low end dead time:
Typical :%nS 
Min :%nS 
Max :%nS ', S_min.RiseTime*1e9, S_max.RiseTime*1e9);
H = (1/(R_high*C))/(s+(1/(R_high*C)));  
S_min = stepinfo(H,'RiseTimeLimits',[0.00,0.52]);  
S_max = stepinfo(H,'RiseTimeLimits',[0.00,0.68]);  
S_typ = stepinfo(H,'RiseTimeLimits',[0.00,0.60]);  
fprintf(fp,'High end dead time:
Typical :%nS 
Min :%nS 
Max :%nS ', S_typ.RiseTime*1e9, S_min.RiseTime*1e9, S_max.RiseTime*1e9);

Listing 17: (V12switcher.m)
%Author: Nathaniel Rohrick

56
clc; clear;
filep = fopen('V12switcher.txt', 'w');

Rdc = 0.0488; % Inductor resistance, assume 25mOhm
Vout = 12; % Output voltage
io = 1.5; % Nominal current value
icl = 3.5; % Current limit
Vin = 24; % Input voltage
Vin_max = 48; % Maximum expected input voltage
Vd = 0.3; % Forward diode voltage (assumption)
Vf = 0.8; % Internal forward voltage drop
Cd = 300e-12; % Diode capacitance estimate
Rds_on = 0.092; % Maximum internal rdson from datasheet
Ton_min = 135e-9; % Minimum controllable on time from datasheet
Vref = 0.8; % Internal reference voltage
tr = 4.9e-9; % Estimated rise time of internal gate drive
Qg = 3e-9; % Estimated gate charge of internal fet
Iq = 146e-6; % Quiescent current of chip
f_div = 8; % Frequency divider of control loop
Vsc = 0.1; % Short circuit output voltage
Kind = 0.3 % Allowable ripple current as fraction of output current
f_maxskip = (1/Ton_min) * (((io*Rdc)+Vout+Vd)/(Vin_max-(io*Rds_on)+Vd))
f_shift = (f_div/Ton_min) * (((icl*Rdc)+Vsc+Vd)/(Vin_max-(icl*Rds_on)+Vd))
dI = 0.5; % Allowable change in load current
dV = Vout*0.01; % Allowable change in load voltage
Ioh = 1.5; % Output current under heavy load
Iol = 0.9; % Output current under light load
Ihys = 3.4e-6; % Hysteresis current specified in datasheet
Iein = 1.2e-6; % Enable input current
Vstart = 20; % PSU starts at input voltage of 20V
Vstop = 18; % PSU stops at input voltage of 16V
Ven = 1.2; % Enable voltage level without hysteresis
gmps = 12; % COMP to SW current transconductance
gmea = 250e-6; % Error amplifier transconductance

% Pick 10^log10x below lowest frequency found above.
if (f_maxskip > f_shift)
    fsw = roundn(f_shift - 10^floor(log10(f_shift)), floor(log10(f_shift)));
else
    fsw = roundn(f_maxskip - 10^floor(log10(f_maxskip)), floor(log10(f_maxskip)));
end

RT = rpv((92417 / ((fsw/1000)^0.991))*1000);
RT = RT(1,1); % R3

% Inductor value calculations
Lo_min = ((Vin-Vout)/(io*Kind))*((Vout)/(Vin*fsw)); % L1
Lo_min = 33e-6; % Chose 513-1351-1-ND
Irip = ((Vout*(Vin - Vout))/(Vin*Lo_min*fsw));
Il_rms = sqrt((io^2)+((1/12)*(((Vout*(Vin-Vout))/(Vin*Lo_min*fsw))^2)));
\[ I_{pk} = io + \left( \frac{I_{rip}}{2} \right) \]

% Capacitor value calculations
\[ C_{o1} = \left( \frac{(2*di)/(fsw*dV)}{} \right) \]
\[ C_{o2} = \frac{L_{min}}{((Ioh^2)-(Iol^2))/(((V_{out}+dV)^2)-(V_{out}^2))} \]
\[ C_{o3} = \frac{1}{8*(fsw)} \times \frac{1}{(dV/I_{rip})} \]
\[ C_{o} = 8 \times \max \left[ C_{o1} C_{o2} C_{o3} \right] \]

\[ C_{o} = 44e^{-6}; \% chose 22u in parallel HHXA500ARA220MF61G \]
\[ R_{esr} = \left( \frac{dV}{I_{rip}} \right) ; \% C_{o} ESR \]
\[ R_{esr} = 40e^{-3}; \% chosen capacitor HHXA500ARA220MF61G in parallel \]
\[ I_{cout\_rms} = \frac{\left( V_{out}/\sqrt{12} \right) \times (Vin - V_{out})}{Vin \times L_{min} \times fsw} ; \% \text{irms} \]

% Diode value calculations
\[ P_{d} = \frac{((Vin - V_{out}) \times io \times Vd)}{Vin} + \frac{((Cd \times fsw \times (Vin + Vd)^2))}{2} \]

% Input capacitor value
\[ C_{in} = 4.7e^{-6}; \% C_{1}/C_{2} \]
\[ I_{ic} = io \times \sqrt{\left( \frac{Vin}{Vin} \right) \times \left( \frac{Vin - V_{out}}{Vin} \right)} \]
\[ dVin = \left( \frac{io \times 0.25}{C_{in} \times fsw} \right) \]

% Bootstrap capacitor value
\[ C_{boot} = 0.1e^{-6}; \% \text{specified in datasheet C4} \]

% Under and overvoltage lockout resistor selection
\[ R_{uvlo1} = \text{rpv}((V_{start} - V_{stop}) / \text{I}_{hys}) \]
\[ R_{uvlo1} = \text{Ruvlo1}(1,1); \% R_{1} \]
\[ R_{uvlo2} = \text{rpv}(\text{Ven} / (((V_{start} - Ven) / \text{Ruvlo1}) + \text{I}_{enin})) \]
\[ R_{uvlo2} = \text{Ruvlo2}(1,1); \% R_{2} \]

% Output feedback resistor selector
\[ R_{ls} = 10e3; \% \text{pick 10k for low side resistor R6} \]
\[ R_{hs} = \text{rpv}(R_{ls} \times (Vout - Vf) / Vf) \]
\[ R_{hs} = R_{hs}(1,1); \% R_{5} \]

% Compensation values for voltage control loop
\[ f_{p} = \frac{io}{(2*\pi*Vout*C_{o})} \]
\[ f_{z} = \frac{1}{(2*\pi*Resr*C_{o})} \]
\[ F_{co1} = \sqrt{f_{p} \times f_{z}} \]
\[ F_{co2} = \sqrt{f_{p} \times (fsw/2)} \]
\[ f_{co} = \min(F_{co1}, F_{co2}) \]
\[ R_{4} = \text{rpv}(\{(2*\pi*f_{co}*C_{o})/gmps\)*(Vout/(Vref*gmea))\}) \]
\[ R_{4} = R_{4}(1,1) \]
\[ C_{5} = 1/(2*\pi*R_{4}*f_{p}) \]
\[ C_{8\_1} = (C_{o}*Resr) / R_{4} \]
\[ C_{8\_2} = 1/(R_{4}*fsw*pi) \]
\[ C_{8} = \max(C_{8\_2}, C_{8\_1}) \]

% Power dissipation estimates for RPS54360
\[ P_{cond} = (io^2)*R_{ds\_on}*(Vout/Vin) \]
\[ P_{sw} = Vin*fsw*io*tr \]
\[ P_{gd} = Vin*Qg*fsw \]
\[ P_{q} = Vin*Iq \]
\[ P_{t} = P_{cond} + P_{q} + P_{gd} + P_{sw} \]
fprintf (filep, 'This file computes component values for TPS54360DDA');
fprintf (filep, ', refer to datasheet for more information');
fprintf (filep, '\n\nR1 = %.2f \nR2 = %.2f \nR3 = %.2f \nR4 = %.2f \nR5 = %.2f \nR6 = %.2f \nC1/C2 = %.2f \nC4 = %.2f \nC5 = %.2f \nC6/C7 = %.2f \nC8 = %.2f \nL = %.2f \n\nTotal = %.2f', Ruvlo1, Ruvlo2, RT, R4, Rhs, Rls, Cin*1e6, Cboot*1e6, C5*1e9, Co*1e6, C8*1e9, Lomin*1e6, Pt);

Listing 18: GATE_DRV_DESIGN.m
%Author: Nathaniel Rohrick
%Date: February 28, 2017
%Description: Creates a list of component values needed for TI's UCC27714 gate driver IC

fp = fopen('gate_drive.txt', 'w');
Qg = 35e-9; %11 nC gate charge AUJRF7640S2CT-ND from digikey
Rg = 3.5; %Intrinsic gate resistance is 3.50hm from ds
fsw = 500e3; %500 kHz switching frequency
Vhv = 24; %High voltage supply
Iqdd = 1050e-6; %Quiescent current of gate driver Vdd to Vss
Iqbs = 300e-6; %Quiescent current of HB-HS supplies of gate driver
Ibl = 20e-6; %Bootstrap leakage current
Vf = 0.3; %Boot diode Vf
Vdd = 12; %Gate drive voltage supply
Cg = Qg / (Vdd - Vf); %Equivalent gate capacitance
Cboot = 10 * Cg; %Bootstrap cap cant drop 10% in voltage, spec
%cap to be 10x gate capacitance
Cvdd = 10 * Cboot; %Vdd capacitor recommended by TI to be 10x Cboot
Rbias = 5; %To allow Vdd pins voltage to ramp longer than 50uS
Rboot = 2.2; %Recommended value between 2.2-10, pick minimum
Idbootpk = (Vdd - Vf) / Rboot; %Max current boot diode sees on startup
Pqc = Vdd * (Iqdd + Iqbs); %Quiescent current power dissipation
Pibl = ((Vdd+Vhv) - Vf) * Ibl * 0.5; %Static losses due to HB leakage
%current at 50% duty cycle PWM
Pq1q2 = 2 * Vdd * Qg * fsw; %Primary power dissipation element turning on and off fets.
Plvlshft = (Vdd+Vhv) * (0.5e-9) * fsw; %Level shift power dissipation
%assuming 0.5nC of parasitic charge
P_INT = Plvlshft + Pq1q2 + Pibl + Pqc; %Total internal power dissipation

%Find ripple current of gate driver
%Assume maximum allowable current
T = 1/fsw; %Period of switching waveform
I = (Vdd - Vf)/Rg; %Max current from gate driver
\[ T_c = \frac{Q_g}{I}; \quad \% \text{Charge up time} \]

\[ t = 0: \left( \frac{T}{10 \times 10^3} \right); T; \]
\[ I_{wv1} = I \cdot \text{heaviside}(t - (T/2)) - I \cdot \text{heaviside}(t - (T/2 + T_c)); \]
\[ I_{wv2} = -I \cdot \text{heaviside}(t - (T - T_c)) + I \cdot \text{heaviside}(t - T); \]
\[ I_{wv} = I_{wv1} + I_{wv2}; \]
\[ \text{RMS}_{I_{wv}} = \sqrt{\text{mean}(I_{wv}^2)}; \]

\[ \text{RMS}_{\text{TOTAL}} = 2 \times \text{RMS}_{I_{wv}}; \quad \% 2 \text{ drivers per chip} \]

fprintf(fp, 'Cboot is: %.4fnF\nRboot is: %.4fOhms\n', (Cboot*1e9), Rboot);
fprintf(fp, '
Total internal Power dissipation Tabulated Below\n');
fprintf(fp, 'Switching losses: %.5fW\n', Pq1q2);
fprintf(fp, 'Level shifting losses: %.5fW\n', Plvlshft);
fprintf(fp, 'Quiescent current losses %.5fW\n', (Pqc + Pibl));
fprintf(fp, '
Ripple Current Rating for bulk decoupler:%.5fA\n', RMS_TOTAL);
fclose(fp);
## 6.4 Bill of Materials

### 6.4.1 BOM billed to BCIT

<table>
<thead>
<tr>
<th>Group No.</th>
<th>Supplier</th>
<th>Part URL</th>
<th>Description</th>
<th>Quantity</th>
<th>Unit Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Adafruit</td>
<td><a href="https://www.adafruit.com/products/1294">https://www.adafruit.com/products/1294</a></td>
<td>MicroSD card breakout board</td>
<td>1</td>
<td>9.95</td>
</tr>
<tr>
<td>16</td>
<td>Adafruit</td>
<td><a href="https://www.adafruit.com/products/1210">https://www.adafruit.com/products/1210</a></td>
<td>SMT breakout board for DAC</td>
<td>1</td>
<td>4.95</td>
</tr>
<tr>
<td>16</td>
<td>Sparkfun</td>
<td><a href="https://www.sparkfun.com/products/9117">https://www.sparkfun.com/products/9117</a></td>
<td>Rotary encoder</td>
<td>1</td>
<td>2.95</td>
</tr>
<tr>
<td>16</td>
<td>Sparkfun</td>
<td><a href="https://www.sparkfun.com/products/11570">https://www.sparkfun.com/products/11570</a></td>
<td>TRRS 3.5mm breakout</td>
<td>1</td>
<td>3.95</td>
</tr>
</tbody>
</table>

**Figure 18: Components used in final design**
# 6.4.2 Class-D Amplifier BOM

<table>
<thead>
<tr>
<th>Name</th>
<th>Manufacturer</th>
<th>Description</th>
<th>Digikey P/N</th>
<th>Type</th>
<th>Value</th>
<th>Package</th>
<th>Unit Cost</th>
<th>Design Quantity</th>
<th>Purchase Quantity</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>110 Ohm Resistor</td>
<td>On Shore Technology</td>
<td>10% 10k Resistor</td>
<td>R3</td>
<td>1206</td>
<td>10k</td>
<td>Through-Hole</td>
<td>0.04</td>
<td>100</td>
<td>100</td>
<td>10.00</td>
</tr>
<tr>
<td>3.6k Ohm Resistor</td>
<td>On Shore Technology</td>
<td>3.6k Ohm Resistor</td>
<td>R1</td>
<td>1206</td>
<td>3.6k</td>
<td>Through-Hole</td>
<td>0.08</td>
<td>100</td>
<td>100</td>
<td>8.00</td>
</tr>
<tr>
<td>56 Ohm Resistor</td>
<td>On Shore Technology</td>
<td>56 Ohm Resistor</td>
<td>R2</td>
<td>1206</td>
<td>56 Ohm</td>
<td>Through-Hole</td>
<td>0.12</td>
<td>100</td>
<td>100</td>
<td>12.00</td>
</tr>
<tr>
<td>10k Ohm Resistor</td>
<td>On Shore Technology</td>
<td>10k Ohm Resistor</td>
<td>R4</td>
<td>1206</td>
<td>10k</td>
<td>Through-Hole</td>
<td>0.04</td>
<td>100</td>
<td>100</td>
<td>10.00</td>
</tr>
</tbody>
</table>

**TOTAL:** $172.58