Nexys4 Audio Player

Andrew Reiff, Daniel French, Daniel Vega
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
E-mails: ajreiff@oakland.edu, dlfrench@oakland.edu, dgvega@oakland.edu

Abstract* – *The purpose of our project is to create an audio player that will load music tracks from an SD card and play them when the user selects the indicated song using the buttons and display on the FPGA board. This paper will give a short introduction, go through the methodology of how we constructed this project, go over the research needed, and explain our results.

I. INTRODUCTION

Music is a big part of everyone's life and there is constant pressure in the engineering industry to improve hardware and overall sound quality for consumers in all areas from consumer electronics to automobiles. The motivation for this project was to conduct in depth research as to how audio signals are constructed and played. In order to do so, we decided to create and audio player using the Nexsys 4 DDR board that will allow one to import previously saved .wav files from an SD card and put them each into a bank. From these banks, the user would be allowed to play a single track or potentially, given the scope of the project, be able to mix and match audio signals and output one signal using Pulse Width Modulation (PWM), through the AUX output on the Nexsys 4 DDR board, to an external speaker in order to hear the mixed audio signal.

This report will cover many topics that we have learned in our computer hardware design course. In order to incorporate audio, we would have to have components such as state machines, multiplexors, registers, a look up table (LUT), a control circuit, as well as the use of the 7-segment displays on the FPGA board. But this project covered much more material than we went over in the course and there was a need to do some in depth research. We had to incorporate a micro SD card into our design. In order to do so, we had to research many things about our Sandisk 2.0GB micro SD card such as the pinouts, storage format, communication protocols, and the timing of the communication. The .wav files on the SD card were saved as an 8-bit unsigned Pulse Code-Modulation (PCM) signal which needed to be converted to a PWM signal in order to output through the Aux port on the Nexsys 4 board. We had to research how to create this converter using modules to incorporate into our top level design. Lastly, one of the last items we had to independently research was how the PWM signal was going to drive the speaker, in other words, how to enable the Aux port on the board, in order to get sound. Overall, there is much to be learned from designing this project and we can see it having many applications in the future such as a template for other students to continue and improve upon our work, and even as a simple media player for one's home. In the reset of this report, we will go in depth about how our design was constructed, the methodology, and the experimental setup we did in order to test our designs.

II. METHODOLOGY

A. Top File

First, let us begin with a brief walkthrough of the program from the user's perspective. Once the board is programmed, powered up, and automatically initialized, the user is presented with a menu in which the user may use the pushbuttons to scroll through a list of files containing sounds or songs which can be played through the board's audio amplifier. Once a file is chosen, the center pushbutton will start the playback. At the same time, a stopwatch will begin recording how long the file has been playing. At this point the user may pause the playback, stop the playback and return to the menu, or wait for the file to finish and automatically return to the menu.

The code for the program implements seven modules into the top file. A complete list of the top file modules is as follows: User Interface, SD Interface, Seven Segment Control, Seven Segment Display, Stopwatch, and a digital PCM to PWM audio converter. These seven module work together to create the program explained above.

The central module of the system is the User Interface. It acts as the main controlling unit as well as the top level FSM. It acts as a control signal by sending 'enable' signals to the sub modules and registers that are needed for the processes in each state. These signals are controlled by the top level state machine in the User Interface. The board's hardware, controlled by the user, sends signals to the User Interface, which navigates through the high level states of the program. The observable high level states are as follows: initialization, menu navigation, play mode, pause mode, and a stop mode.

The SD Interface is designed to receive an 8 bit command signal from the User Interface and perform a specific operation depending on the command it receives. The design utilizes the SPI protocol of the SD card. SPI has a slower response than standard SD protocol, but it is much easier to implement, which is ideal for small scale applications such as this program.

The Seven Segment Display module receives a string and displays it on the Nexys 4 board's 8-digit Seven Segment Display. It does this by sending the string through a decoding machine that outputs eight 8-bit segment codes. The built-in state machine cycles through each digit one at a time at 1 kHz. The module receives its input from the Seven Segment Control module, which outputs a string based on its communication with the User Interface. For example, in the 'menu navigation' mode it will change the word it displays whenever the user presses the left or right pushbuttons.

Once a file is selected for playback, it is pulled from the SD card and the data stream is sent to the PCM/PWM converter. One of the most common modulation techniques for storing audio data in a portable storage device is Pulse Code Modulation (PCM). However, the board's built-in DAC circuit requires Pulse Width Modulation (PWM), so the data stream from the file must be converted before it is sent to the output audio jack.

Lastly the Stopwatch was added to display how long a file has been playing. It uses five counters to divide the rate at which the Q values change. Four Q values are sent to the Seven Segment Control module to display the output of the Stopwatch.

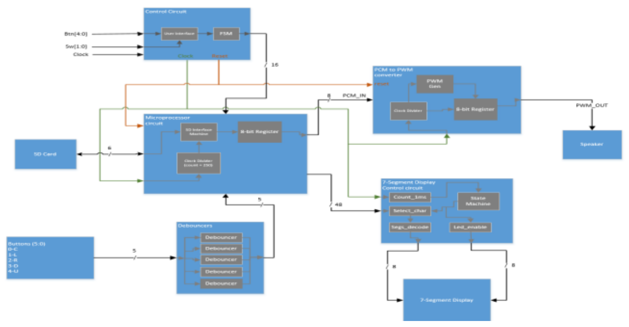


Figure 1. Top level design

B. *Clock*

The project required the use of many clock signals in order for separate components to gather the correct data for each audio signal. We used a global 100MHz clock signal "Clock" and each event occurred on the positive or negative edge of this clock signal. The micro SD card, however, needed a 100-400KHz clock so we had to use a clock divider in order to step-down our 100MHz clock to be able to communicate properly with the SD card.

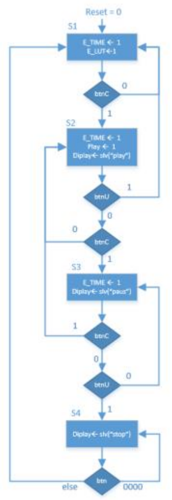
C. *State Machines*

The entire program has a total of 5 different state machines. They are found in the User Interface (main FSM), SD Interface, Stopwatch, Seven Segment Display module, and the Debouncers. The state machines in the Seven Segment Display module and the Debouncers will not be discussed since they remain constant throughout the execution and navigation of the program. Additionally, the state machine that controls the Stopwatch closely resembles part of the User Interface state machine, so it will not be explained individually either.

The state machine in the SD Interface is tightly integrated with the central FSM in the User Interface. It is designed to input an 8-bit instruction received from the central FSM, and output a 'done' and a 'ready' signal to the central FSM, depending on its state. In its post-idle state, the FSM is sending a high 'ready', which communicates that the FSM is ready to receive a new instruction. The first 2 bits of the instruction signal are high at this point. As soon as the FSM reads that the first 2 bits of the instruction are low, it begins its process. First it assembles a 48-bit command to send to the SD card. This command contains the remaining 6 bits from the original instruction. Next it sends the command by enabling a shift register. Next it waits for a low signal from the SD card that signifies the beginning of a response. Once a low bit is read, the FSM enables a second shift register to pull in the response. The response contains information about any errors that may have occurred. If there are no errors, the FSM continues with its process. The remainder of the process depends on the command that was sent, and there may be nothing more to do. Once the process is completed, the FSM advances to its pre-idle state and sends a high 'done' signal to the User Interface. It will remain in the pre-idle state until the first 2 bits of the instruction signal read high. Once they do, the FSM returns to its post-idle state and outputs the ready signal.

The main state machine communicates with the SD Interface state machine during the initialization stage and the playback states. When it sends the 8-bit instruction, it retains that output until it receives the 'done' signal. It responds asynchronously by setting the instruction output to all '1's. (The SD Interface state machine then responds synchronously to the high bits by returning to its post-idle state.).

Starting from power up, the main FSM instigated the initialization of the SD card by sending two different instructions, sending the second only after receiving the 'done' signal. Once initialization is complete, it advances to the menu state and activates the Seven Segment Control module. In this state, the FSM will cycle through the file names when it receives a signal from the left or right pushbuttons. A signal from the center pushbutton causes it to advance to the pre-play state as it sends instructions to the SD Interface. It advances to the play state once the 'done' signal is received. Here, the FSM has 2 possible paths. Another center button signal sends it to the pause state where it sets the SD Interface enable signal low, stopping the bit stream. Here, another center button signal will re-enable the SD Interface and send the FSM back to the play state. In either the play or pause state, a signal from the top button sends it to the pre-stop state, where in sends instructions to stop the data transmission altogether. Once the 'done' signal is received, it advances to the stop state, and the automatically back to the menu state.



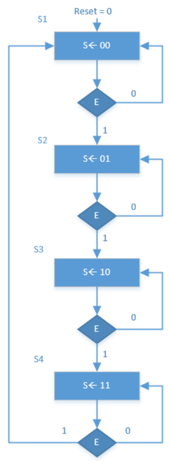


Figure 2. 7-Seg State Machine Figure 3. Timer display state machine

D. *External Storage (SD Card)*

The first thing we had to do for this project in terms of research was to figure out how to access the micro SD card from the Nexys 4 board. This was going to prove to be the most difficult step. We figured out that Secure Digital (SD) cards have two modes of communication, Secure Digital Protocol and Serial Peripheral Interface (SPI) protocol. In order to communicate with the SD card, we could need to figure out which of the two modes would suite our needs the best.

First step was to determine the pin outs for each mode, which can be seen in table 1 below. From reading the user manual for the Nexys 4 DDR board from Digelent, we were able to find the diagram in figure 21 that depicts how we need to connect the micro SD card to the appropriate ports on the board. Fortunately, the board already comes with the connections pre-made and with internal pull-up resistors. Without the pull-up resistors, the SD card would not be able to initialize or send data. Our specific micro SD card was made by SanDisk and had a capacity of 2 GB. From the datasheet, we found that it could be powered by 2.7v to 3.6v. The Nexys board powers the SD card with 3.3v that should be perfect.

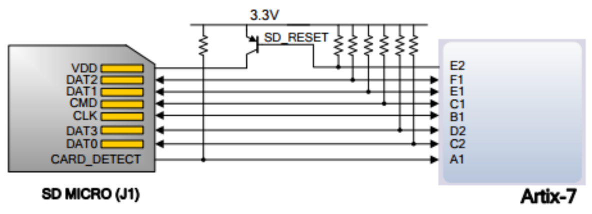


Figure 21. Artix-7 microSD card connector interface (PIC24 connections not shown).

Pin	Name	Function (SD Mode)	Function (SPI Mode)
1	DAT3/CS	Data Line 3	Chip Select/Slave Select (SS)
2	CMD/DI	Command Line	Master Out Slave In (MOSI)
3	VSS1	Ground	Ground
4	VDD	Supply Voltage	Supply Voltage
5	CLK	Clock	Clock (SCK)
6	VSS2	Ground	Ground
7	DAT0/DO	Data Line 0	Master In Slave Out (MISO)
8	DAT1/IRQ	Data Line 1	Unused or IRQ
9	DAT2/NC	Data Line 2	Unused

Table 1: SD Card Pin Assignments [2].

1) *SD Protocol*

Based on our research, utilizing the SD protocol method would allow us to send data from one to up to four data lines (DAT0-DAT4) as seen in table 1. First, we would have to send a command to the SD card and wait for a response. Both the command and the response are transferred serially on the Command (CMD) line on pin 2. Also, both the command and response utilize the same format, but not all bits are needed for each response. The format for these blocks of data is shown below in figure 5. The command and response data blocks are comprised of 48 bits were the first bit is a logic low start bit which will tell either the SD card or the Nexys 4 board when to start reading since the last bit will be set to logic high and will keep the line active high until another packet is to be read. The second bit indicates which device, whether the master or slave, is talking on the line. Logic high indicates that the board is sending a command to the SD card, which a logic low indicates that the SD card is sending a response back to the board. The next 6 bits after the host bit indicates the command that we want to execute, whether it is reading, writing, deleting, etc. This section is only necessary when sending a command and can be ignored when receiving a response from the SD card. The next 32 bits dictate the argument of the message which specifies the data address in the SD card which we want to begin reading from or writing to. And lastly, the 7 bits before the end bit which is specified as CRC7 in the figure below, are used by the SD card to tell the board the status of the SD card and/or whether the command was successfully received. This section can be ignored when sending a command.

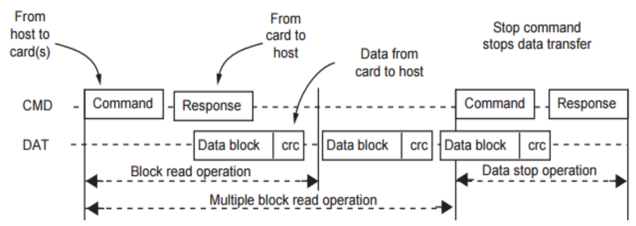
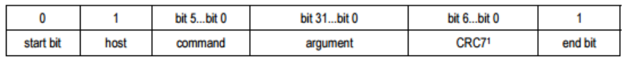


Figure 4. process diagram for SD mode.



0	1	bit 5...bit 0	bit 31...bit 0	bit 6...bit 0	1
start bit	host	command	argument	CRC7[1]	end bit

Figure 5. Command and Response format

One thing that we noticed from our research about using SD mode is that data is transferred both on the rising and falling edges of the clock signal. As shown in figure 4, this means that we would have to incorporate in our code a method from filtering out the data block that would be received from the SD since it will begin to send data part way through sending a response on the command line.

2) SPI Protocol

Now, using the SPI communication method was very similar to the SD protocol. However, in order to use SPI, first, one would have to set the DI and SD pins high and wait a minimum of 90 clock cycles before sending command CMD0 in order to initialize the card into SPI mode. After sending this command, one would have to wait a few clock cycles (not specified in documentation) and then send command CMD1 in order to check the status of the card. By sending CMD1, we would then get a response from the SD card check the CRC bits (7 down to 1), which would tell us whether the board was still in "Idle" mode (0x01) or in "ready" mode (0x00). As seen in figure 6 below, this protocol, after getting through the initialization process, is more straightforward as we would only be sending and receiving data on the rising clock edges. After sending the commands through the DataIN line, we would then get a response back through the DataOUT line with no overlap in the clock signals as when using the SD mode protocol. Using this communication method, we would only need to use the chip select (CS), data in (DI), data out (DO), and clock pins which are pins 1, 2, 7, and 5 respectively as shown previously in table 1.

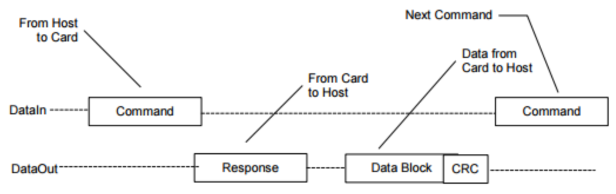


Figure 6. Single block read operation (SPI protocol)

3) Storage Format (FAT16)

The last thing we needed to know from our SD card was how the information was stored on it. SD cards use a File Allocation Table (FAT) format to store data in different locations on the card. In order to send the SD card a command, we would need to fill the argument section by finding the location on the SD card where our audio tracks are being stored. File Allocation Tables divide the disk storage in to clusters where different cards have a different number of sectors per cluster. Our card had about 64 sectors per cluster and the first 512 bytes of our card indicate the boot sector that tells us all the relevant information pertaining to our specific SD card; this information is permanent and cannot be changed. According to our boot sector we were able to find out that the disk we were using had the ability to have about 236 separate entries. The card immediately makes two copies of these entries and stores them in different locations so that if one were to get deleted the files were backed up. Due to this duality feature, the FAT on our disk was located in two locations pertaining to what is called the "offset." The offset is a way for users to address specific bytes within the card and our FAT's were located at offsets 512 and 121,344 and each were roughly 236 sectors long. These FAT were a way to find where our files were located by reading the individual clusters. Once a cluster was located, it gave an offset address that could be directly referenced to find where a file was located on the disk. Once located we were able to use the offset address in our argument sent to the SD card to directly load the file from the memory of the chip.

III. Experimental Setup

To set up and test our project, first we had to set up our SD card in the FAT16 format. We then found an audio file using a sound from the classic video game Super Mario Bro's. To use this file with our hardware we had to convert it to an 8-bit PCM file using a program called Audacity. Once the file was converted we loaded the file on the chip and had to use a Hex Dump of the FAT-16 format of the card to find the offset at which our sound was located. We then would be able to call to that address using SPI protocol on the board and load the signal in 8 bits at a time. Figuring out the SPI protocol was where we were stumped however due to the timing constraints necessary for proper functionality. After determining that this process worked, we would load multiple sound files into the SD card and complete the code for the final presentation.

IV. Results

Due to the complexity of the SD interface modules and actually getting the timing correct for reading from the SD card to the board, we were not able to get a fully functional project. We tested most of the modules in a simulation to make sure they were working. Our presentation demonstrated the overall functionality of our project and how the user will interface with the board to select the tracks. It also showed a fully functional timer system that activated when the user played, stopped, and reset the system. Overall, this project still has a short way to go but can always be improved upon.

Conclusions

Interfacing with a micro SD card to the FPGA board proved to be a very difficult task. This was a time consuming process as there was much research to be done since this material was never presented in our course. Not only did we learn the basics behind how an SD card works, the file allocation system, communication protocols, and process flow, but we also have a much greater knowledge of how this interaction in VHDL.

There are a few issues/bugs to be worked out. As seen in our functional simulation. We need to make sure we are capturing all the data being transmitted from the SD card so that we try to minimize the error in sound quality. Also, we have to find a way to improve our PCM to PWM code in order to get the most accurate conversion and there is minimal loss in sound quality.

There are any improvements that we would like to make for this project in the future. Our group would like to turn this audio player into a “beat” mixer. Essentially, we would like to be able to store multiple beats/sounds into ROM on the Nexys board and then be able to overlap and loop them so that we can create music just like music producers do in actually music production labs.

REFERENCES

[1] SandDisk Corporation. (2003). SandDisk Secure Digital Card: Product Manual. Version 1.9. Document No. 80-13-00169.

[2] SD Card Association. http://sdcard.org/.

[3] Digilent Inc. (2014). Nexys4 DDR FPGA Board Reference Manual. Revision C. Document No. 502-292.

[4] "How to use MMC/SDC". Website. February 18, 2013. http://elm-chan.org/docs/mmc/mmc_e.html. Accessed April 4, 2016.