Audio Transmission via S/PDIF over TOSLINK

1st Eric Bui
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
eqbui@mit.edu

2nd Alyssa Keirn
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
akeirn@mit.edu

3rd Luis Martinez
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA
luism@mit.edu

***Abstract*—Fiber optics are an important method of communication. Fiber is vastly faster than other forms of long-distance data transmission and as a result forms the basis for modern communication. We use two FPGAs, one to collect and transmit audio data, and one to receive it and output it to a speaker.**

The implementation consists of an audio WAV file stored on an SD card on the FPGA. This digital data is converted to an optical data source and coupled to a TOSLINK cable with a JIS F05 connector. Data is transferred through the cable via fast-traveling pulses of light in the 650 nm range from an LED and transmitter circuit. The data link protocol is S/PDIF, a serial uni-directional interface over short distances. S/PDIF format is preferable for audio as it allows the transfer of data between two pieces of digital audio equipment without needing analog connection that would lessen the quality. A receiver device with a photodiode receives and converts the optical information to digital to the second FPGA. It is then edited and sent over another TOSLINK cable to a speaker.

***Index Terms*—S/PDIF, TOSLINK**

I. Physical Construction

The setup consists of:

- An optical TOSLINK cable
- A transmitter circuit consisting of TOTX1950A optical transmitter[1] with data rate of 10MBd with a control signal coming from our FPGA digital I/O at a bit rate of 6 MHz.
- A receiving circuit including the TORX1950A optical receiver[2] that requires a simple voltage drop from 5 V to 3.3V by using a signal diode and load resistor. To reduce parasitics, we tightly couple the signal output into the digital I/O pins for the FPGA by using header pins.
- Two Nexys 4 DDR FPGAs
- One 2GB micro-sd card
- A speaker with optical input

This setup may be modified slightly in the final version, with additional hardware such as a screen.

A. Hardware Construction

The receiver circuit is based on the implementation of the TORX circuit. We included a load resistor and a signal diode in order to step down the voltage to 3.3V, instead of the 5.0V measured at the output. For the transmitter circuit, the implementation is instead based on the TOTX implementation circuit provided. We included a 5600Ω load resistor and a 0.1μF bypass capacitor.

In order to verify and obtain the proper configuration values for the S/PDIF i.e. default consumer channel words, auxiliary bits, and user bits, we implemented a Python script used to scope a single packet of audio data from a reliable source —the reliable source being a DVD disk player. For the speaker, we purchased an optical TOSLINK compatible speaker that uses S/PDIF as audio transmission protocol. It is to note that in order to initially scope the signal from the disk player, we used a digital logic analyzer in order to figure out the values and transfer them to a Python format that we could parse and analyze.

II. SD Card Audio

A. SD Card Controller

The setup uses a SD card module written by Johnathan Matthews[3]. This module controls the direct interfacing to the SD card. The rest of the setup was assembled by the group. The module reads 8 bytes at a time, in 512 byte chunks. The data is in .WAV PCM 8-bit unsigned raw format. The SD card runs on a 25Mhz clock created by passing through a clock wizard generated using the IP. This prevents the original clock from being overdriven. The controller module is controlled by a state machine waiting for user input and ready signals.

The state machine takes in addresses and a ready signal to begin a read. The ready signal and addresses are controlled by the file system module. The output of the state machine is then passed to an intermediate module (sd card addressor) that sorts the data output. Each 512 block is either cluster information, directory information, or audio data. The addressor sorts the output by information type. Cluster data is accumulated into a three byte chunk, and then sent to the file system module. Directory data is sent to a block ram so it can be accessed by the VGA (this information contains song titles and start addresses for the file system). Audio data is sent to a FIFO buffer for transmission. This more modular system was easier for us to debug, and allowed us to implement the project in stages.

B. FIFO Buffers

The data from the SD card is sent to one FIFO buffer. This buffer allows the transmission module to catch up the the SD card, which reads much faster than the transmission rate. The FIFO stores three 512 byte blocks at a time. Two thresholds are

set: full512, which goes high when the FIFO has at least one block, ans empty512, which goes high when the transmission module has sent at least one block of data. These signals are used in the top level state machine to determine when to ask the file system module for more data.

C. System State Machine and User Interface

A simple state machine in the top level module coordinates the ready, read, write, and valid bits between the file system and SD modules, the FIFO buffers, and S/PDIF transmitter module. It ensures that the SD card is read in time for the S/PDIF transmitter module's needs. The system starts playing audio after receiving user input. The top module also contains all the variable initilizations and connections between other modules.

Using btnu (up), btnd (down), and btnc (select), a user can scroll through songs on the SD card. Directory data is stored on a bram using the SD addressor module. This bram can then be read by the extractor module according to the user's selection, allowing song names to be displayed on the VGA. The top level state machine keeps track of the current display song and whether is has been selected to be played.

D. Clock Domain Crossing

Because we read off the SD card at 25mhz, transmit data at 6.144mhz, and display to the VGA at 75mhz, modules were needed to handle clock domain crossing. To solve this issue, we used the IP generator to create several true dual port block rams. We instantiated one block ram for each variable that needed to be moved across clock domains. This allowed us to forgo state machines and just tie the addresses and read and write enable inputs to hardcoded values.

III. FILE SYSTEM

In order to allow the simple transferring of song information into the SD card, we implemented a FAT12 file system, which allows the user to "drag and drop" file from a computer onto the SD card. This is possible given that the SD cards are fragmented into sectors of 512 bytes, where information can be stored.

A. File Addressing on the SD card

Every SD card contains a Master Boot Record (MBR) at the beginning of the file. This record holds the partition table, which in itself contains four partitions. We formatted the FAT12 partition onto the first partition, where there is a Linear Block Address that when multiplied by 512, provides us the resulting address of the start of the FAT formatted partition (in our case, at 0x100000).

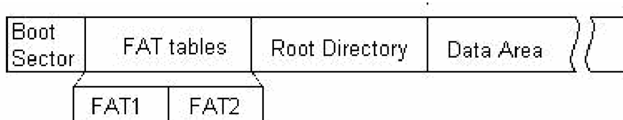


Fig. 1. Structure of FAT12 block

B. FAT12

The FAT12 is a fragmented file system characterized for its simplicity. The system is fragmented into clusters which specify the address of the stored data. In general, the FAT12 is organized in four structures:

- *BIOS Parameter Block (BPB)*: this structure is 2 sectors wide and contains information about the format of its specific FAT details. After the BPB there are Reserved Sectors, which are already specified in the BPB.
- *FAT (File Allocation Table)*: there are typically two FAT —both identical to avoid corruption. The data in the FAT12 is organized in cluster objects which in this case, are 32 sectors wide. The FAT thus, works essentially as a lookup table. Once you have a cluster number, you lookup the value of the cluster number in the corresponding index of the FAT to get the next cluster number. It is to note that the cluster number is used to find the appropriate locations of the sectors in the data.
  As an example shown in the following diagram, we get our starting cluster number from the dictionary at entry 2. We then read off the data at the logical cluster with value 2. Following, we check FAT table 2, obtain the value 4. Given that we have the value 4, we read the data at Logical Cluster Value 4 and then go to entry 6 and so forth, until we reach entry 7; which has an ending code that specifies the end of the file —typically values that are greater than or equal to 0xFF8.
  Finally, we also note that the FAT entries are only 12 bytes wide —hence the name, which is not a power of 2. Therefore, it is possible to run into issues where the sectors, of size 512 bytes, are not perfectly divisible by 12. This prompts us to find an alternate implementation. Furthermore, they are organized in such a way that it takes these letters as hex values 0xBCZAXY. If the cluster number was even, then we would get the cluster value 0xABC. If odd, then the cluster value would be 0xXYZ.
- *Dictionary*: following the FAT tables, each dictionary entry is 32 bytes wide and contains the name of the files, the file length, and the file's starting cluster.
- *Data sector*: this comes after the Dictionary, and includes all of the data that the cluster data points to.

The user can scroll through the directory entries where each directory entry is 32 bytes, where the song name occupies the upper 11 bytes and the song starting cluster is at bytes 26 and 27. Thus, the user can select the song by iterating through the appropriate directory entries.

At the beginning of every FAT partition, we have certain important information to consider. First, we make further note that the data displayed are all represented in little Endian format. Second, at the BIOS Parameter Block of the FAT partition, we note that the number of bytes per sector is 512, and most importantly, the number of sectors per cluster is 32. Furthermore, the number of FATs are two, and the number of hidden sectors is 32. Finally, the size of each FAT

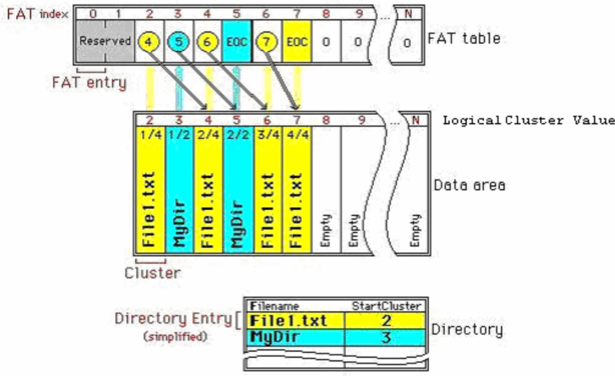


Fig. 2. FAT12 Allocation table

table, reserved sectors, including root directory is 32 sectors large. The beginning of the data sector starts after these; and therefore, we start 128 sectors after the beginning of the FAT partition.

IV. S/PDIF

A. Structure

The S/PDIF protocol[4] consists of blocks that are 192 frames wide; with every frame divided into two 32 bit subframes —one for the left audio channel and another one for the right audio channel respectively. In our implementation, as we reproduce monophonic sound, the two audio channels shared the same information, aside from the preamble that characterizes each channel. Excluding the 8-bits that form the preamble, every logical bit is sent using biphase-mark encoding (BMC); thus, sending two bits for every logical bit at a clock rate of 6.144MHz. The structure of every subframe is as follows:

- *Preamble*: 8 bits that mark the start of every subframe. It can vary between three unique alternating sequences: one for the right channel, one for the left channel, and one for the start of a block.
- *Auxiliary*: 4 optional bits are used to include extra audio data.
- *Audio Sample word*: 20 bits include the audio data. In mono, both subframes carry the exact same audio data; in stereo, they have different audio data.
- *Validity Bit*: single bit that indicates whether the audio sample word is invalid.
- *User Bit*: single bit of user data channel corresponding to the associated channel in the subframe.
- *Channel Bit*: single bit that contains important information about the transmission channel (e.g. clock frequency, user mode, etc.) that is collected in a 192-bit word, including an 8-bit CRC-8/EBU that allows for checking its validity once the entire block is received.
- *Parity Bit*: single bit that is assigned such that the entire subframe, excluding the preamble, contains an even number of ones and zeros.

B. Frame Assembler

The frame assembler is the principal module that, given 20 bits of audio data at the start of every frame, constructs the aforementioned structure and sends it encoded in BMC to the transmitter circuit through the JA PMOD pins. In order to obtain the audio data, the frame assembler sends a ready signal to the audio FIFO at the start of every frame, which will send the 20 bits before the module finishes sending the preamble. We send the following sections —auxiliary, audio word, and the last four bits— through the use of a module that encodes every bit in BMC. In our design, the 192 bits that conform the channel bits are set to a default value that allows for monophonic audio to be played in the speaker. Given that we are always sending valid data and are not making use of the user data bits, we set these two to 0. Through the process of sending the first 27 bits, we track the parity such that we set the parity-bit to 0.

This entire process is repeated multiple times as long as the ready signal from the FIFO —the one that indicates us that the FIFO contains enough information for a block— is present.

C. Receiving Modules

Once the audio message is sent through the TOSLINK optical cable, we use the receiver circuit to gather the information through the JC PMOD pins. As the information is received at a frequency of 6.144MHz, we sample the bits at a frequency of 60MHz, assessing the validity of each bit by considering 8 similar samples. Once we obtain a complete bit, we send it to the receiving modules.

In order to decode the bits received, which are a combination of preambles and BMC encoded data, we introduce a *biphase-mark* decoder module. As the preambles include sequences of ones and zeros not reproducible in BMC encoding ('000' and '111'), we detect the preamble corresponding to the start of a block and then continue to decode each dibit of the subframe.

Then, we send to decoded bits to a *frame dismantler* module, which carries the purpose of extracting the audio data, and checking both the validity of the subframe and the entire block i.e. checking the validity bit, the parity bit, and the CRC-8/EBU at the end of the block. This module sends the extracted data to a FIFO buffer at the end of every subframe, given that the data is valid and the parity is preserved. At the end of a block however, the module compares the last 8 channel bits (containing the checksum) and with the CRC-8; this will trigger a done signal indicating the end of a block, and a kill signal indicating that the block is invalid. Upon finding an invalid block, the kill signal will tell the FIFO buffer to delete the audio data information received. Once received, in addition to the done signal, the FPGA will show the last 32 bits of the channel bit word (including the checksum) in the seven-segment display.

For re-transmission, we use another *frame assembler* module that will receive the audio data from the FIFO buffer, in addition to the valid 192-bit channel word. This design allows

for verification of our transmission design before transmitting audio data to an external speaker.

V. VIDEO INTERFACE

A. VGA Display

Given a title stored in the SD card, we want to provide the user an interface that allows them to select a song through the use of the pushbuttons embedded in the Nexys board. In order to achieve this, we made use of a series of modules designed by the Project F group[4]. Through the use of these modules, we constructed an ASCII title with small stars being displayed in the background.

The titles can be up to 10 letters long, for a total size of 10 bytes for each input title and each byte representing an ASCII code point that we use as address for a BRAM that contains our bitmap fonts (in this case using Unscii). By pushing the up button or the down button, we find the corresponding address at which the the title is contained, and then, we send it to the video interface module.

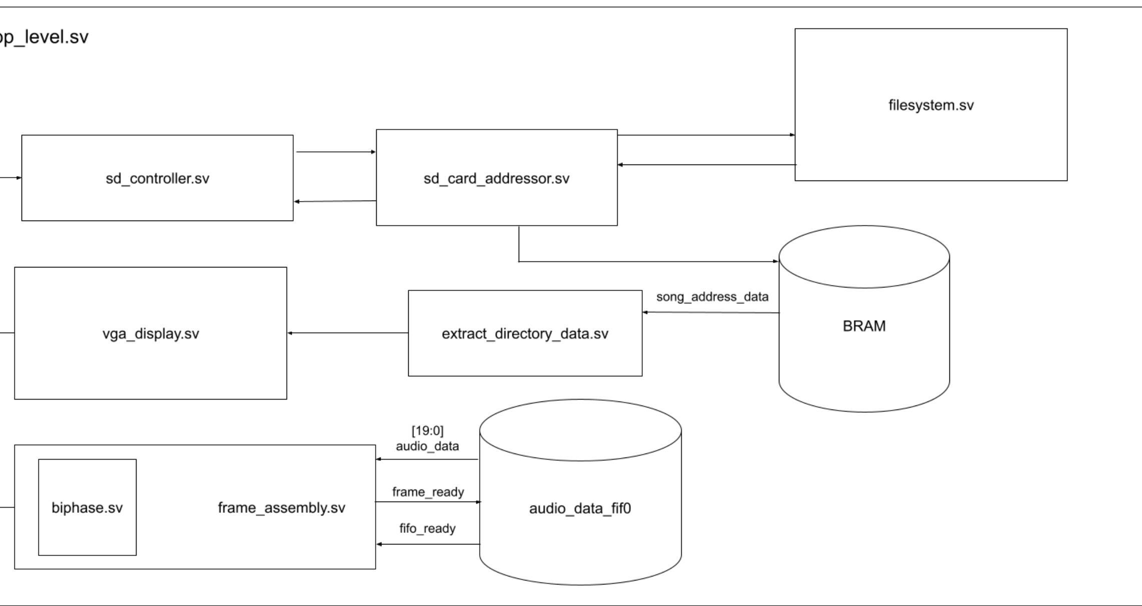
VI. RETROSPECTIVE

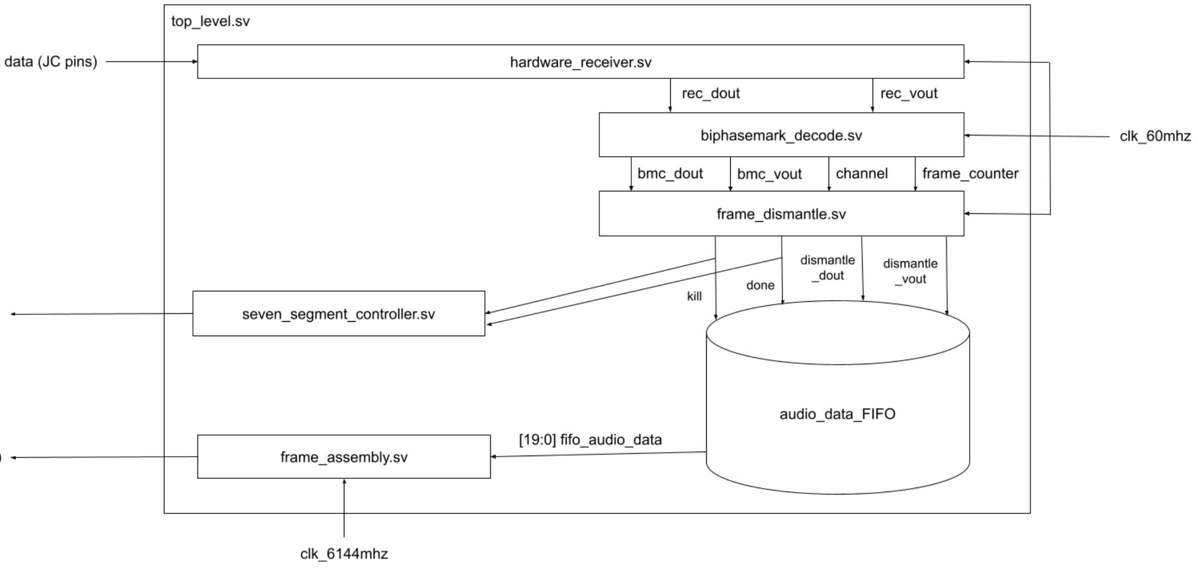
We learned quite a few things throughout this process:

- *Filesystem:* Implementing file systems is difficult. Probing the SD card was critical. If we were to do the project again, it would be helpful to work out a way to simulate the SD without having to debug everything with an ILA.
- *Hardware:* We had a few hardware faults that left us confused as to whether software of hardware was the problem. In the future, we would like to buy multiple copies of each part.

REFERENCES

[1] Mouser datasheet, TOTX1950. April 04, 2016.
[2] Mouser datasheet, TORX1950, April 04, 2016.
[3] Jonathan Matthews' Github repository
[4] European Broadcasting Union, Tech 3250: Specification of the digital audio interface, 3rd edition. June 08, 2004.
[5] Project F's Github repository
[6] https://wiki.osdev.org/FAT
[7] https://wiki.osdev.org/MBR_(x86)
[8] https://www.eit.lth.se/fileadmin/eit/courses/eitn50/Literature/fat12_description.pdf
[9] Our repository containing the code and the modules designed for this project.


op_level.sv
filesystem.sv
sd_controller.sv
sd_card_addressor.sv
BRAM
song_address_data
vga_display.sv
extract_directory_data.sv
[19:0]
audio_data
biphase.sv
frame_assembly.sv
frame_ready
audio_data_fif0
fifo_ready


top_level.sv
data (JC pins)
hardware_receiver.sv
rec_dout
rec_vout
biphasemark_decode.sv
clk_60mhz
bmc_dout
bmc_vout
channel
frame_counter
frame_dismantle.sv
dismantle
_dout
dismantle
_vout
done
kill
seven_segment_controller.sv
audio_data_FIFO
[19:0] fifo_audio_data
frame_assembly.sv
clk_6144mhz