Implementing an SPI to APB interface for the Leon CPU

Alex Bolinsky and Jeremy Tang
Washington University in St. Louis, Saint Louis, MO 63130
(Dated: November 6, 2014)

In this paper we present a design and implementation of a general Serial Peripheral Interface (SPI) bus that enables communication between the processor and the ADC and DAC onboard the Xilinx Spartan3 FPGA. While the general design of an SPI peripheral is similar across platforms, the challenge in the design lies with creating an interface compatible with components with distinct timing requirements. In this case, the ADC and DAC timing requirements differed from each other, so we needed to design an interface with these differences in mind. Our solution involved condensing a timing diagram from the datasheets of both the ADC and DAC, translating the timing diagram to state machine logic, and implementing the state machine in hardware. In the end, we were able to design and implement an efficient SPI that properly interfaces the Leon processor with the ADC and DAC and which supports a sampling rate of 127k samples/sec aggregate, or 63.5k samples/sec per channel. Ultimately this design for the SPI will play a key role in our overarching design of an audio equalizer.

I. INTRODUCTION

The Serial Peripheral Interface (SPI) bus is a full-duplex serial link used for short-range communication between devices. It is a protocol developed by Motorola that has since become a de facto standard for communication between master and slave devices in embedded systems, and is also used by sensors and SD cards. The SPI protocol consists of only four signals, shown in Figure 1. SCLK (serial clock) is output from the master device to the slave device and controls the rate of the data transfer. MOSI (master output slave input) and MISO (master input slave output) are the two data transfer signals, each of which transmits one bit per SCLK cycle. SS (slave select) is the signal from the master which enables the slave device for serial data transfer.

A Digital-to-Analog converter, or DAC, is a device which takes in a digital value as input, and outputs an analog voltage signal corresponding to the magnitude of the digital value. They are a key component of digital music players, which must convert a music files digital encoding into an analog signal that can be processed by speakers or headphones. An Analog-to-Digital converter, or ADC, is the opposite; it takes in an analog voltage signal and outputs a digital value corresponding to the amplitude of the analog voltage. Like how a DAC can be used to playback a digital music file as music, an ADC can be used to record an analog audio signal into a digital format, after which the audio can be digitally processed or saved for future playback.

For this project, our objective was to design a peripheral for the Xilinx Spartan3 FPGA to act as a bridge between the AMBA bus protocol and the SPI protocol, and use the SPI peripheral to interface the Leon3 processor with the Spartan3 boards LTC1654 DAC and LTC1865L ADC. By default, the Leon3 processor has no ability to communicate with the DAC and ADC. By developing a SPI peripheral device, the Leon3 could send data to the peripheral over the AMBA bus, and the SPI peripheral could forward the data over SPI bus to the DAC or ADC. The SPI peripheral could then receive data from the DAC or ADC, and send the data back to the Leon3. We used Verilog HDL to design our peripheral, and we wrote a C program for the Leon3 to test and operate the peripheral.

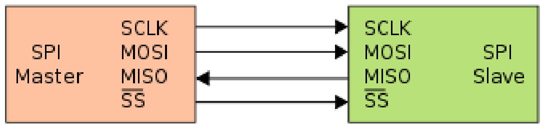

FIG. 1. Illustration of two example devices communicating over the SPI bus.

II. DESIGN

Our design for the SPI and the manner in which it connects the Leon Processor to both the ADC and the DAC is illustrated by the high level block diagram in Figure 2. On the left of the figure, the PSel13 and PSel14 signals are asserted by the Leon Processor to select which SPI peripheral to write to, PRData13 and PRData14 are 32 bit data lines that the ADC SPI and DAC SPI use to push data back for the Leon to read, and PWData is a 32 bit data line that the Leon uses to write values to the SPI peripherals. On the right of the figure we have a general SPI design where the SPI is the master and the ADC and DAC are slaves. The master asserts SCK to clock the slaves, CS to select one of two main states in the DAC and ADC, and a single bit data line, SDI, to write data serially to the slave. In our design, the slave asserts a single bit data line, SDO, to write data serially back to the SPI master.

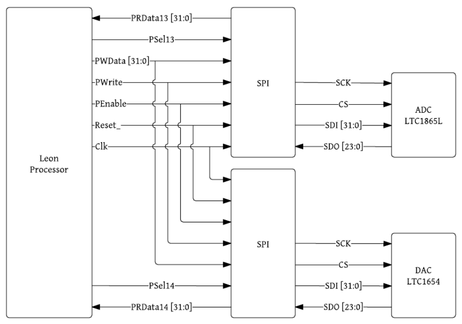

FIG. 2. High Level Block Diagram of our SPI Peripheral.

A. Timing

Our SPI peripheral was designed to accommodate the different timing specifications of the ADC and DAC onboard the Nu Horizons FPGA. The timing diagram used in designing the state machine is shown in Figure 3. The processor clock signal is shown at the top, and the four master-slave signals are below: CS, SDO, SCK, and SDI. The propagation delay from clock output to the rising and falling edges of CS and SCK is 0-14ns. There is a minimum hold time requirement from the falling edge of SCK to when SDO is valid of 14ns (clock propagation delay) + 8ns (offset from valid to changing SDO signal) + 60ns = 82ns. There is a minimum setup time requirement from when SDI goes valid to the rising edge of SCK of 66ns 14ns (clock propagation delay) = 52ns. The period of the processor clock is 33ns, and we designed the high and low time for SCK to be 3 processor clock periods, or 99ns each to meet the timing requirements above and to meet the 100k samples/sec sample rate design specification. There is a minimum setup time of 99ns (SCK period) 14ns (clock output propagation delay) = 85ns between the falling edge of CS and the first rising edge of SCK, so we designed the first rising edge of SCK to occur 3 clock periods, or 99ns, after CS goes low. A new bit from SDI is written from the peripheral a clock tick after CS goes low or a clock tick after each falling edge of SCK. Similarly, a new bit from SDO is sent to the SPI peripheral two clock ticks after the rising edge of SCK.

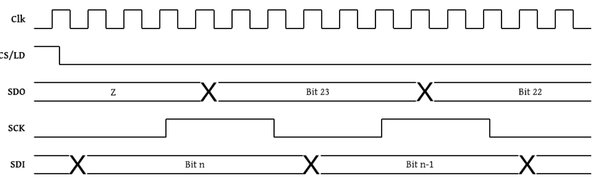

FIG. 3. Timing Diagram of our SPI Peripheral.

B. State Machine

After determining the timing diagram for the SPI that works with both the ADC and DAC timing specifications, designing the state machine was a relatively straightforward translation of the timing considerations. The state machine for this SPI can be seen in Figure 4. Given that the ADC and DAC have two cycles, Load and Convert, we designed a state machine with three main higher-level states: Init, where the state machine initially begins or returns to from a reset or disabling, Convert, where the state machine enables communication for the ADC or DAC to complete a conversion cycle, and Talk, where the state machine enables communication for the ADC or DAC to complete a load cycle. We designed two initial states, Init and InitParse.

In Init, the peripheral continually stores the PWData sent by the Leon. Only if the peripheral receives PSel, PEnable, and PWrite from the Leon do we progress to InitParse. If the peripheral received the command from the Leon to disable, the machine progresses back to Init. If on the other hand the peripheral is enabled, it stores the number of bits per sample and the sample rate that was passed by the Leon, and calculates and stores a counter based on the sample rate and number of bits that determines how long our machine allows the ADC or DAC to stay in its conversion cycle. The calculation for the counter is as follows:

$$Counter = clkrate - samplerate - 6 * BitsPerSample * samplerate$$

If the SPI peripheral is interfacing with the ADC, the number of bits/sample is 16, and if the SPI is interfacing with the DAC, the number of bits/sample is 24, according to the LTC1654 and LTC1865L data sheets. The ADC Load cycle should take 16 (bits) * 6 (processor clock ticks per SCK period) / 30MHz = 3.2 microseconds. If we want a sample rate of 50k samples/sec aggregate, then the total time for a load cycle + a conversion cycle should take 1/50000 = 20 microseconds. Therefore, in the case of the ADC, the length of the conversion cycle should be 20 microseconds 3.2 microseconds = 16.7 microseconds. After the counter is set, and the values from the Leon have been stored, our machine enters the first of the conversion states. The state machine enters ConvertReady, lowers the ready signal to be sent back to the Leon, and remains here while decrementing the counter by the sample rate until the Leon asserts a write to send the peripheral a new sample. If the Leon does not send a new sample before the counter finishes, the state machine calculates a new counter based on the number of bits/sample and progresses to Talk0 to initiate the Load cycle. By design, this results in a dropped sample, but it prevents the machine from introducing delays that would alter the sample rate that we specified. If the CPU controlling communication to the Leon is fast enough, we expect a new sample to always be written before the

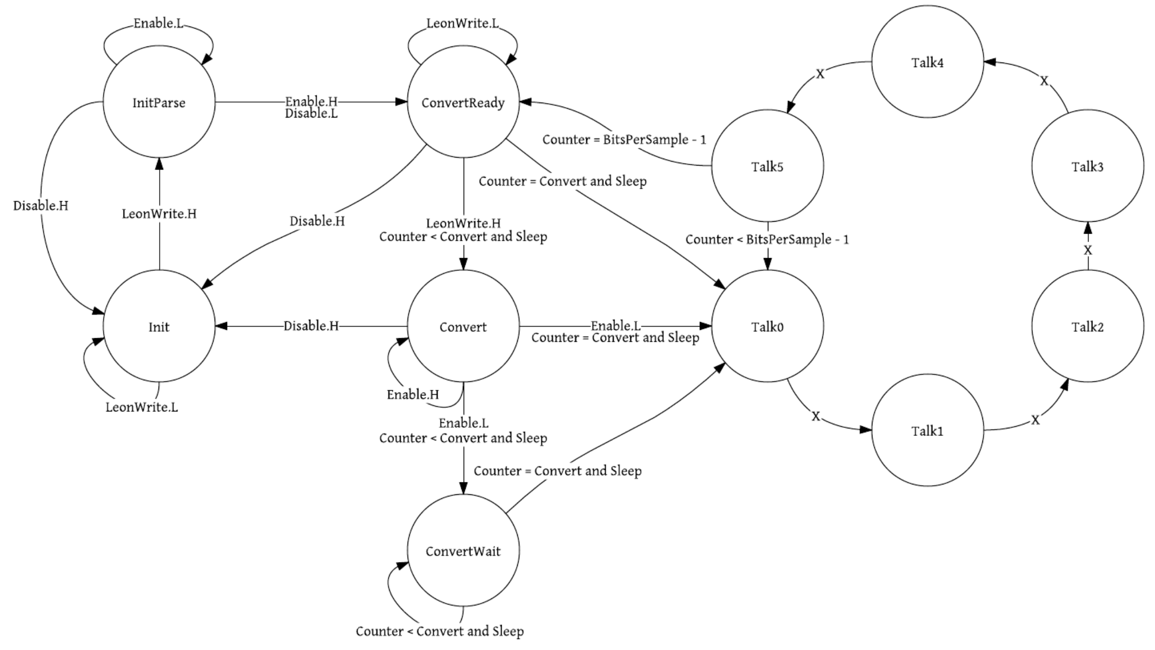
Enable.L
LeonWrite.L
Talk4
InitParse
Enable.H
Disable.L
ConvertReady
X
X
Counter = BitsPerSample - 1
Talk5
Talk3
Disable.H
LeonWrite.H
Counter = Convert and Sleep
Disable.H
LeonWrite.H
Counter < Convert and Sleep
Counter < BitsPerSample - 1
X
Init
Disable.H
Convert
Enable.L
Counter = Convert and Sleep
Talk0
Talk2
Enable.H
LeonWrite.L
X
Talk1
X
Enable.L
Counter < Convert and Sleep
Counter = Convert and Sleep
ConvertWait
Counter < Convert and Sleep

FIG. 4. State Machine of our SPI Peripherall.

counter finishes. If the SPI does receive a sample before the counter finishes, the machine progresses to the Convert state. In this state the SPI peripheral stores the data previously sent from the Leon into SDI_Data. If the counter is not finished, the counter is decremented and the machine progresses to ConvertWait. The machine loops in this state, decrementing the counter, until the counter finishes. In any of the Convert states, when the counter finishes, a new counter is calculated based on the number of bits/sample, the peripheral asserts CS low, SCK low, and progresses to Talk0 to start the load cycle.

The last collection of states we designed for the state machine include Talk0 to Talk5, six states that control the Load cycle of the ADC or DAC. We chose to implement this in six states instead of one in order to break down the sequence of timing events by individual clock ticks, which simplified the design process. Based on the number of bits/sample passed to the Leon prior, the machine will loop through these six states either 16 or 24 times for a full Load cycle, and then return to ConvertReady to begin the next conversion cycle. In these six states we control SCK (low for three clock ticks, high for three clock ticks) by asserting SCK high in Talk2 and SCK low in Talk5. In Talk0, we write a single bit selected by the counter to SDI and progress to Talk1. Nothing occurs in Talk1 and we progress to Talk2. In Talk2 we assert SCK high and progress to Talk3. Nothing occurs in Talk3 and we progress to Talk4. In Talk4 we read a single bit from SDO and store it in its proper ordered place in a register within SPI denoted by the counter, and we progress to Talk5. In Talk5, we assert SCK low. If the counter is not finished, we progress to Talk0 and continue the cycle, but if the counter is finished, we set the counter to the previously calculated value determining the duration of the convert cycle, we assert ready back to the Leon in order to allow a new sample to be written to the peripheral, raise CS high, and progress to ConvertReady to begin the next conversion cycle.

C. Hardware Implementation

Finally, with our timing diagram translated to a state machine, we implemented the conceptual model in hardware. The simplified block diagram of the SPI peripheral itself is shown in Figure 5. The hardware implementation was mostly a direct mapping of the state machine logic to Verilog. Of interest to note is the way that we interpreted data received by the peripheral from the Leon via PWData and data sent from the peripheral to the Leon via PRData. We parsed data sent from the Leon to the SPI as follows: If loading the sample rate and number of bits/sample: bit 31 = enable signal, bit 30 = disable signal, bits 29..22 = dont care, bits 21..5 = sample rate (up to 100000), and bits 4..0 = number of bits/sample. If specifying control and address bits to the DAC or ADC: bits 31..24 = don't care, bits 23..20 = control bits, bits 19..16 = address bits, bits 15..0 = don't care. Similarly, we parsed data sent to the Leon from the SPI as follows: bit 31 = ready signal, bit 30 = enabled signal, bits 29..25 = current state, bit 24 = dont care, and bits

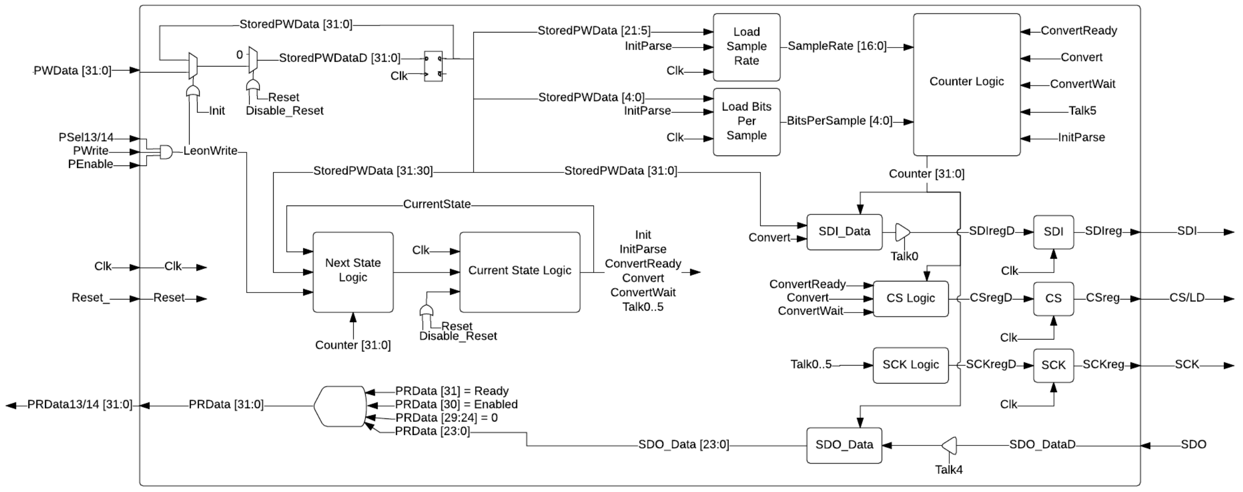

FIG. 5. Simplified Block Diagram of our SPI Peripheral.

23..0 = SDO_Data, or the converted sample. The bits for the enabled signal and the current state exist for debugging purposes, but do not have any cost associated with keeping them in the final design. In hardware, we tied the PRData output directly to these signals and registers with assign statements in order to make data always available to be read by the Leon. In addition, to simplify the code, we designed the reset conditional within the synchronous block to be the logical OR of the Reset_ signal coming from the Leon and StoredPWData[30] == 1, which is the signal to disable the peripheral.

III. OPERATION AND TESTING

A. Testing in simulation using ModelSim

To test our implementation, we created a custom simulation-only copy of our Top file, called "Tiptop," in which we instantiated our APBBusMaster and two instances of our SPI peripheral while removing all synthesis-only code. Additionally, we connected Tiptop to Ross Williamson's ADC tester and DAC tester devices, which we had downloaded from the "Useful Documents" section of the course website, and then connected the ADC tester and DAC tester each to an instance of our SPI peripheral. With this setup, we were able to write simulated APB bus calls in APBBusMaster to PSel13 and PSel14, allowing us to interface with an SPI peripheral handling the ADC tester and an SPI peripheral handling the DAC tester, respectively.

To test how our SPI peripheral interfaced with the DAC tester, we wrote the code in our APBBusMaster to write out the ENABLE command on PSel14. This ENABLE command also contained the sample rate parameter and the bits-per-sample parameter, which were set to 50000 samples/second aggregate and 24 bits/sample, respectively. Upon enabling our peripheral, we then wrote the DAC commands to set both DAC channels to FAST mode. We then wrote out 1000 instances of the hexadecimal word "dead" to each DAC channel, for 2000 total writes. Finally, we sent the DISABLE command to our peripheral. After each command to the ADC or DAC, we paused the APBBusMaster code with a while loop to wait for our peripheral to output the Ready bit before proceeding to write out the next command.

$$sin(x/25000 * 500 * 2\pi) \tag{1}$$

To test how our SPI peripheral interfaced with the ADC tester, we first had to supply the ADC tester with data to write to our peripheral. We used Microsoft Excel to generate 1000 points of a 500Hz sine wave at 25000 samples/second, using the equation above. We then transformed the sine wave to have a range from 0 to 65535, converted the resulting numbers to four-digit hexadecimal numbers, and then copied the resulting hexadecimal numbers into a text file to be read by the ADC tester. Our APBBusMaster command sequence was similar to the command sequence we used with the DAC tester, with a few key differences. First, our ENABLE command set the peripheral to read/write 16 bits per sample, instead of 24. Second, we did not send commands for FAST mode to our peripheral as the ADC does not have such a mode. Third, instead of writing "dead", we wrote the ADC commands to make the ADC tester record a sample from one of the two ADC channels, and at the same time passing it back to the Leon as PRData.

We used ModelSim to simulate our design and verify our timing before we implemented the design in hardware. With an aggregate sample rate of 50000 samples/second and a clock rate of 30MHz, we expected our peripheral to have $30 * 10^6 \div 50000 = 600$ clock cycles in between samples. With ModelSim, we verified that our SPI peripheral for the DAC tester enabled and disabled correctly, that each write to the DAC tester wrote 24 bits, that the DAC tester received the data word dead after each sample write, and that the total number of clock cycles in between sample writes was 600 clock cycles. Thus, we verified that our SPI peripheral design worked correctly with a DAC in simulation.

Similarly, we verified that our SPI peripheral for the ADC tester enabled and disabled correctly, that each write to and read from the ADC tester was 16 bits, and that the total number of clock cycles in between sample reads was 600 clock cycles. Additionally, we checked and confirmed that data delivered from the ADC tester to the SPI peripheral was being written out to PRData and read by the APBBusMaster, and we also printed out the integer representations of the data read from the SPI peripheral. We then compared the printed data to our input sine wave data, and found the integer values to all be identical, thus confirming that our SPI peripheral design worked correctly with an ADC in simulation.

B. Testing in hardware using the Leon

To test our implementation in hardware, we wrote a C program for the Leon that communicated with our two SPI peripherals using the APB bus. The SPI peripheral handling our ADC could be accessed by reading and writing to memory address 0x800000d00, and the SPI peripheral handling our DAC could be accessed by reading and writing to memory address 0x800000e00. Our objectives were to use our SPI peripherals to write a sine wave to our DAC and to read 25000 samples of data from our ADC. In order to write a sine wave to our DAC, we first needed to prepare data points for a sine wave. We used Microsoft Excel to generate 2500 points of a 500Hz sine wave, or 100ms worth of data at 25000 samples per second, using the equation that we had previously used to test the ADC in simulation. We then took the hexadecimal conversion of the sine wave samples and stored them into an int array of size 2500.

Our C code was modeled after our APBBusMaster code, in that after every command we waited for the peripheral to output a high ready bit before writing the next command. We easily detected if the ready bit was high or not by taking advantage of 2s complement; if the ready bit was high, the int value of the SPI peripherals output would be negative, else the output would be positive. Thus, after sending a command, we waited for the peripherals output to become negative before moving on to the next command.

To test our SPI peripheral with the DAC, we needed to output the 2500 digital samples of our 500Hz sine wave through the DAC. We first sent the command to disable the associated SPI peripheral to reset it and ensure it was in the Init state and not elsewhere due to a garbage write by the Leon upon initially starting up. We then sent the ENABLE command, along with the parameters for 50000 samples per second and 24 bits per sample. The next command sent was the DAC command to set both DAC A and DAC B to fast mode. We then constructed a for loop that iterated through the sine wave array loading DAC A then DAC B with the same sine wave value per iteration through the loop. For 2500 digital samples, we expected to see the DAC output data onto the oscilloscope for 100ms. We connected probes to the DAC A and DAC B terminals on the board and used the analog signal and timing on the oscilloscope to verify that our output was correct.

To test our SPI peripheral with the ADC, we passed a 50Hz sine wave to the ADC from a tone generator, recorded 25000 samples of the wave per channel, and printed 1000 samples to the console window. We first sent the command to disable the associated SPI peripheral to reset it and ensure it was in the Init state and not elsewhere due to a garbage write by the Leon upon initially starting up. We then sent the ENABLE command, along with the parameters for 50000 samples per second and 16 bits per sample. We then constructed a for-loop that iteratively wrote data from ADC channel 0 into an array while sending the command to convert the analog signal in ADC channel 1, then wrote data from ADC channel 1 into another array while sending the command to convert the analog signal in ADC channel 0. This for-loop ran 25000 times, for a total of 25000 samples per channel being read and stored into arrays. We then printed 1000 data points from each array to the console, for later analysis.

To test if we were recording correct voltages from the ADC, we used our test boards potentiometers to send constant voltages to the ADC. By connecting the ADC inputs to the oscilloscope and by adjusting the potentiometers, we verified that our ADC has a maximum range of 3.3V. We then adjusted the potentiometers such that ADC channel 0 had a 1.0040V input and ADC channel 1 had a 2.0094V input. We then ran our C code and output the recorded digital values to the console window. Using Microsoft Excel, we extracted the hexadecimal data output and converted it to integer. Since the ADC output 16-bit values, we could transform the digital signal to its corresponding voltage signal by multiplying each data value by the constant 3.3/65536. By doing so, we saw that our device recorded an average voltage of 1.0696V from ADC channel 0, and an average voltage of 1.9359V from ADC channel 2. The percent error for ADC channel 0 was $(1.0696 - 1.0040)/1.0040 = 6.534\%$, and the percent error for channel 1 was $(2.0094 - 1.9359)/2.0094 = 3.658\%$. Considering the imperfectness of our ADC device and the amount of electrical noise likely present in our system, we figured that the discrepancy between our actual ADC input and our recorded ADC input was reasonable.

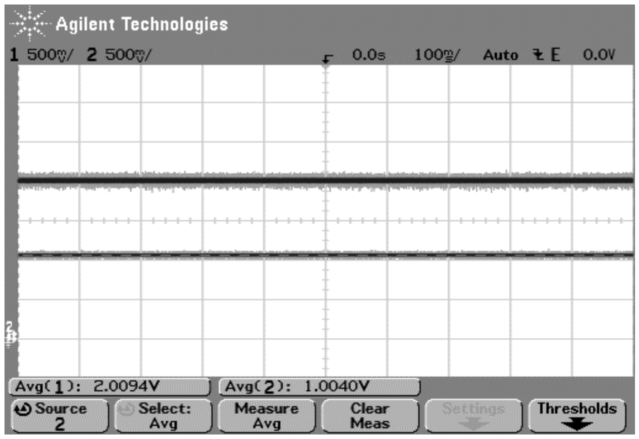

FIG. 6. Oscilloscope reading of our actual input voltages to the ADC channels.

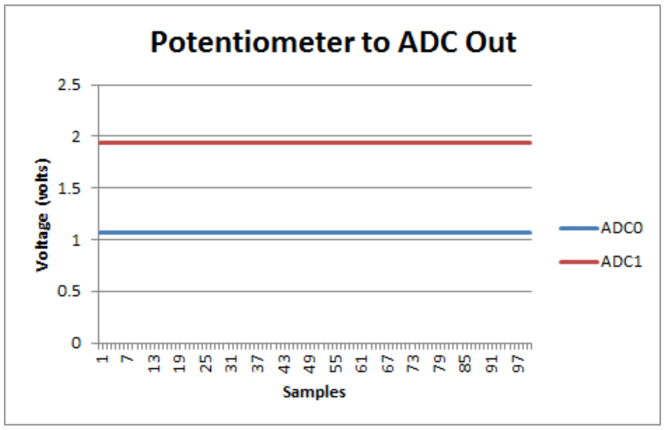

FIG. 7. Excel graph showing the recorded input voltages to the ADC channels.

After verifying that the SPI works with the ADC and DAC individually, we wanted to bring everything together. In order to allow our C code to keep up with our SPI peripherals, we disabled the two peripherals together and enabled the ADC SPI immediately after enabling the DAC SPI. We did this to keep both SPIs in sync as much as possible and to allow the CPU to wait for the ADC SPI to become ready instead of both peripherals. After setting the DAC to fast mode, we made the Leon enter an infinite loop where we endlessly performed the following: send command to read from ADC channel 1, wait for ADC to become ready, write data from ADC channel 1 to DAC A and then send the command to read from ADC channel 0, wait for ADC to become ready, then write data from ADC channel 0 to DAC B. A full cycle of the DAC and ADC can be seen in Figure 8 With this sequence of commands, we could play music from a laptop through the ADC analog input and be able to hear the music come out of the DAC analog output through computer speakers. A comparison between the original signal input and the output from the DAC can be seen in Figure 9.

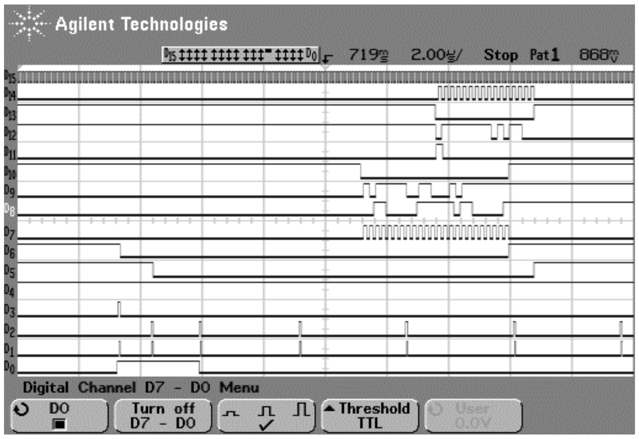

FIG. 8. Oscilloscope output showing a full cycle of the DAC and ADC. For reference, D0 is PWrite, D1 is PEnable, D2 is PSel13, D3 is PSel14, D4 is Reset_, D5 is PRData13[31] (ready), D6 is PRData14[31] (ready), D7 is SPI_DAC_CLK (SCK), D8 is SPI_DAC_MOSI (SDI), D9 is SPI_DAC_MISO (SDO), D10 is DAC_LD (CS), D11 is SPI_AD_SDI (SDI), D12 is SPI_AD_SDO (SDO), D13 is AD_CONV_ST (CS), D14 is SPI_AD_SCK (SCK), and D15 is a half processor clock signal.

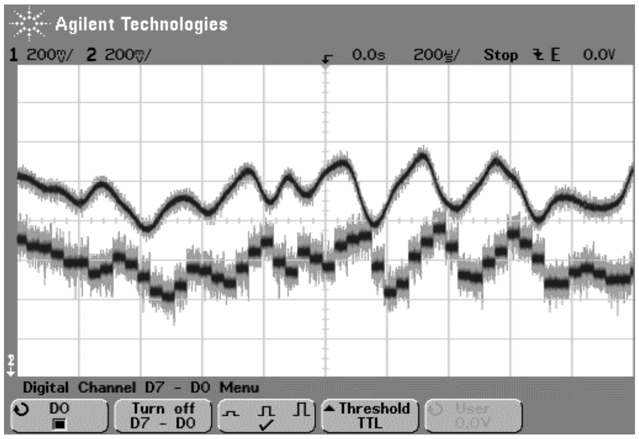
Agilent Technologies
1 200mV/ 2 200mV/ 0.0s 200us/ Stop E 0.0V
Digital Channel D7 - D0 Menu
D0
Turn off D7 - D0
Threshold TTL
User 0.0V

FIG. 9. Oscilloscope output comparing the input signal to the ADC with the output signal from the DAC. Notice how the output signal has a much lower sampling rate than the source signal.

We verified the correctness of our C code and our peripherals behavior in multiple ways. First, when sending the 500Hz sine wave data to our DAC, we simply listened to the speaker output of our DAC and audibly compared the sound to a 500Hz sine wave generated from a laptop, and concluded that the sounds had identical pitch. Second, we output 250 samples of the sine wave onto the oscilloscope and used the cursors to verify that 10ms of time elapsed from the beginning of the wave to the end, which can be seen in Figure 10.

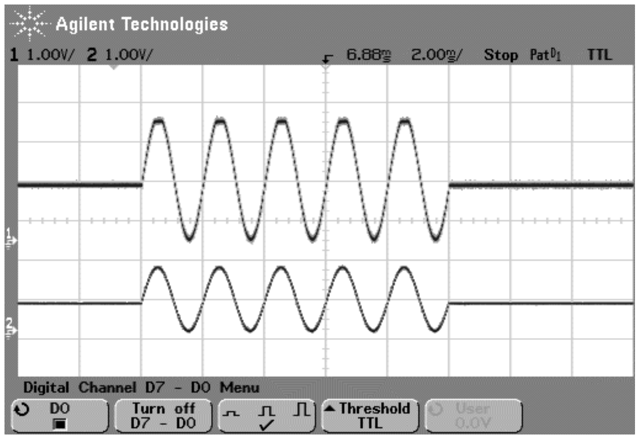
Agilent Technologies
1 1.00V/ 2 1.00V/ 6.88ms 2.00ms/ Stop Pat D1 TTL
Digital Channel D7 - D0 Menu
D0
Turn off D7 - D0
Threshold TTL
User 0.0V

FIG. 10. Oscilloscope output of our DAC reading 250 samples of 500Hz sine wave. The output of the two channels of the DAC differ by 1.65V due to one channel driving 0-3.3V and the other driving 0-1.65V.

Third, we took the 1000 data samples from our ADC and plotted them. These plots can be seen in Figure 11. For a 50Hz sine wave at 25000 samples a second, we expected 25000/50 = 500 samples to occur in between each period of the wave, or for 1000 samples to occur for 2 periods. From the output from our ADC, we can confirm that exactly two periods of the wave elapsed during those 1000 samples, meaning that our ADC SPI peripheral collected samples at the correct frequency and did not miss any of the 1000 samples that it collected. Additionally, we modified our C code to play back the 25000 collected samples through our DAC, and were unable to notice any audible difference between the output sound and the original 50Hz sound.

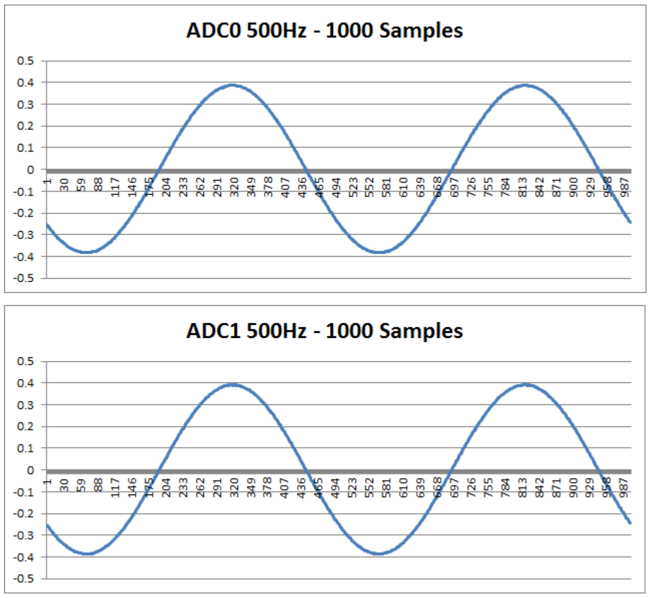

FIG. 11. Excel plot of 1000 data points of a 50Hz sine wave captured from our ADC.

Fourth, we used the Test pins on the Spartan board to output the SPI signals onto the oscilloscope in order to verify that the timing of the digital signals was correct and that no samples were being dropped or written more than once. To do this, we used the oscilloscope cursors to measure with a high degree of resolution (on the order of hundredths of a nanosecond) the time between subsequent falling edges of CS for both the ADC and the DAC. A sample rate of 50k samples/sec translates to 1/50000 sec/sample, which is 20 microseconds. We verified that the time between subsequent falling edges of CS for both the DAC and ADC in most samples was 20 microseconds, which can be seen in Figure 12. Interestingly, we did observe a small delay of 5 nanoseconds between signals in about 1/6 of the samples. This delay appeared not just with CS but with all of the signals associated with the two SPIs, however the delay did not necessarily manifest itself in every signal for a particular sample. It is important to note that this delay does not cause the ADC and DAC cycles to become out of sync with each other, and after a long detailed timing analysis of a long stretch of samples on the oscilloscope, we found that this delay was not causing any samples to be dropped or be duplicated.

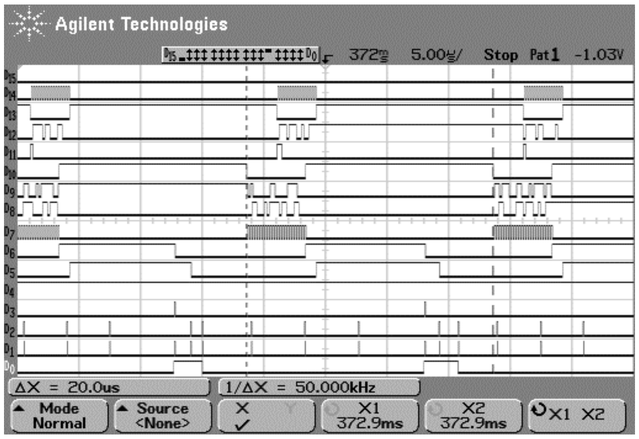

FIG. 12. Oscilloscope showing exactly 20us between falling edges of CS (D10).

To glean more insight into the possible sources of this delay, we added a Test signal that represented half of the processor clock. The delay was not a result of any errors on our part, as any timing mistakes in our state diagram would result in delays on the order of a clock period, or 33ns, so we suspected an issue with the clock signal that the Leon was sending to our peripheral. We triggered the scope on the pulse width of CS and observed a jittering of the half processor clock signal, which is very strange. Our best assessment as to the source of the delay is a fluctuation of the power supply voltage to the FPGA board, but further testing is required before conclusions can be drawn.

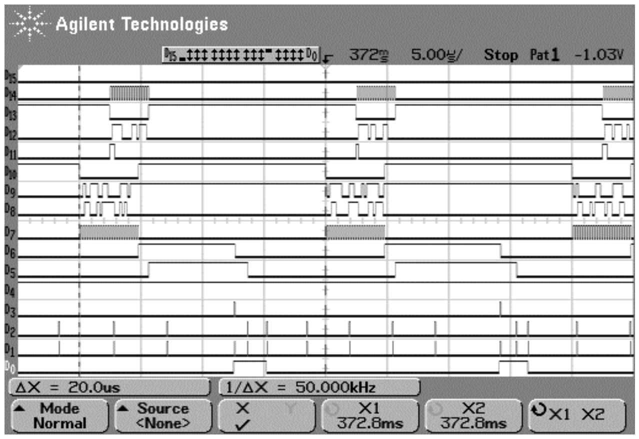

FIG. 13. Oscilloscope seemingly showing 20us between falling edges of CS (D10).

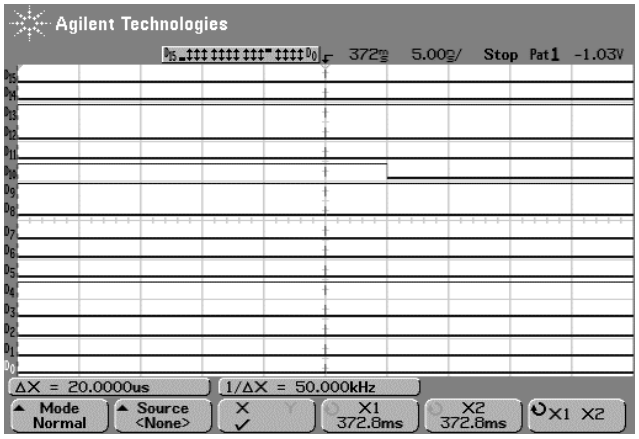

FIG. 14. Oscilloscope output showing that the time between CS (D10) falling edges is actually 20.005us; a 5ns delay.

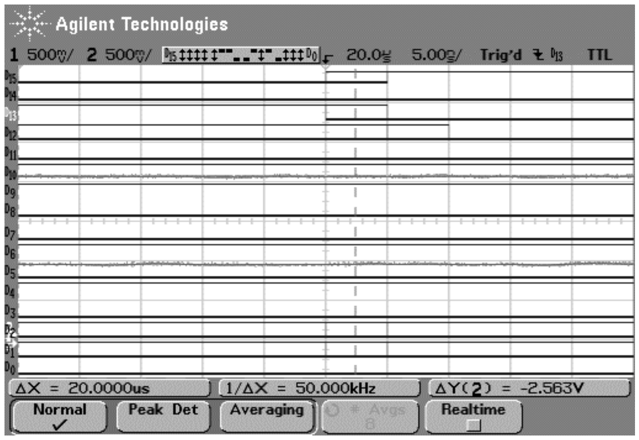

FIG. 15. Oscilloscope output showing jitter in the CS signal (D13) and our half-clock signal (D15).

IV. DISCUSSION/CONCLUSIONS

We designed and successfully implemented a general SPI in hardware and verified that the design was able to sustain a sample rate of 25k samples/sec per each of two audio channels (50k samples/sec aggregate). Theoretically, the maximum sample rate our design is able to support is limited by ADC. The minimum time required for the ADC convert cycle is 4.66us, and the ADC load cycle takes 6 (clock ticks) * 16 (bits/sample) / 30MHz = 3.2 microseconds. So our hardware can support 1 / (3.2us + 4.66us) $\approx$ 127k samples/second, aggregate, or 63.5k samples/sec per channel. Ultimately, the only bottleneck that prevented sampling at such a high rate during testing was the speed of the CPU used to run the code that communicated to the Leon. For example, we tested a sample rate of 88.2k samples/sec aggregate, which is CD quality, and found many instances of lost samples due to the inability of the CPU to catch up to hardware operations. This bottleneck will vary depending on the CPU used to communicate with the Leon processor, but our hardware is well designed to ensure it meets the base specification of a sample rate of 50k samples/sec aggregate.

We have discerned that our design is sufficiently well built for its application within the audio equalizer, but we acknowledge that a couple of modifications could be made to minorly improve performance. The first modification would be to consolidate the init and initParse states into a single state to save a clock cycle, simplify the finite state machine, and potentially make for more readable verilog. The second modification would be to consolidate and simplify the convert states so as to reduce the logic and again simplify the finite state machine and improve readability. These consolidations proved difficult and time-consuming to implement.

This project required us to become familiar with and learn a significant number of concepts and skills. We learned the mechanisms behind the ADC and the DAC, and we spent a good deal of time digging deep in the bowels of datasheets for the two components, which was the most in-depth either of us had probed into hardware documentation. We became familiar with general SPI protocol and learned how to design an interface between two components with incompatible timing. Additionally, we became more adept at working the oscilloscope by learning to use the probes and how to manipulate and view analog signals, and we learned how to trigger on a pulse width, which proved useful in debugging and accurately measuring the delays between signals.

V. CITATIONS AND REFERENCES

en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus
Wikipedia article on the Serial Peripheral Interface

https://www.sharelatex.com/templates/journals/aps

LaTeX template used to format this report.

http://cds.linear.com/docs/en/datasheet/1654fb.pdf LTC1654 DAC datasheet by Linear Technology

http://cds.linear.com/docs/en/datasheet/18645lfs.pdf LTC1864L ADC datasheet by Linear Technology