Axi4Lite SPI Report

CSE 465 Fall 2020

Ben Guan, Will Hamic

Abstract
In this project, we developed a SPI peripheral in verilog which will interact with the ADC and DAC. We verify the functional correctness of the ADC by collecting and analyzing 25,000 samples from a function generator. We verify the functional correctness of the DAC by writing 1,000 samples from the microblaze to the DAC and analyzing with an oscilloscope. Finally we verify their correctness together with a loopback test where values of the ADC are read from the microblaze and written back to the DAC. Both the ADC and DAC will run at 50k Hz.

Introduction

In this project, we implemented the Axi4Lite Serial Peripheral Interface (SPI) in verilog. SPI will be instantiated for both the ADC and DAC to process the signals. The DAC used in this lab is LTC1654 and the ADC is LTC1865L. They are capable of sampling signals ranging from 1Khz to 100Khz. By changing the chip select (conv), we can control what device to operate. The sck bit will determine when the datas are asserted. The conv bit and sck bit are tightly controlled using a FSM which is discussed in the Design session.

Design

The FSM consists of 5 states: SLEEP, INIT, IDLE, HIGH, and LOW. When SPI receives an enable signal from microblaze, the state will transition to INIT. It will stay in INIT for 3 clock cycles, then transition to HIGH state. In HIGH state, SPI will assert sck, then output data to sdi. Three cycles later, we decrement the index and the state will transition to LOW state. In LOW state, SPI will set SCK =0 and stay in LOW for 3 clock cycles. Once the index is 0, that means we are done with sending or receiving one sample, then the state will transition back to IDLE to wait for another sample. We also have a counter that keeps track of the number of cycles for conv to be high. For a 50k sampling rate, the number of cycles for one period of conv is 600. The time that conv remains low stays the same for various sampling frequencies, which is 96 clock cycles (16*6 = 96 cycles) for ADC and 144 clock cycles for DAC (24*6 = 144 cycles).The sck will be running at 5 MHz which is 6 times slower than the system clock (30 MHz).

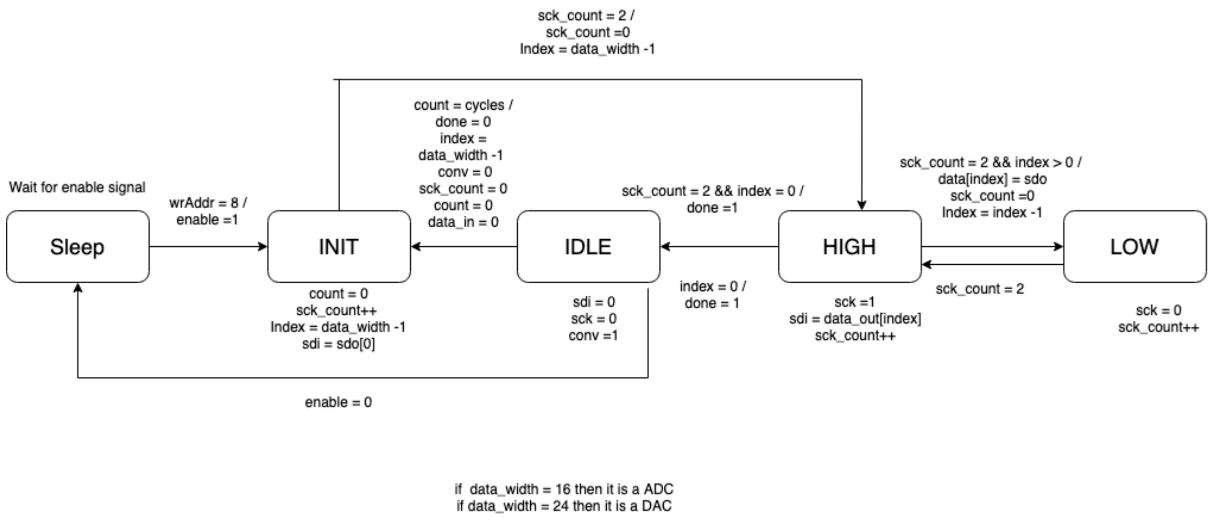


Figure 1. Finite State Machine of FIR Module

The ADC has a data width of 16 bits. We will set the first two bits of sdi to be “10” for channel 0 and “11” for channel 1. This pattern will alternate so each channel will receive samples at the same sampling rate.

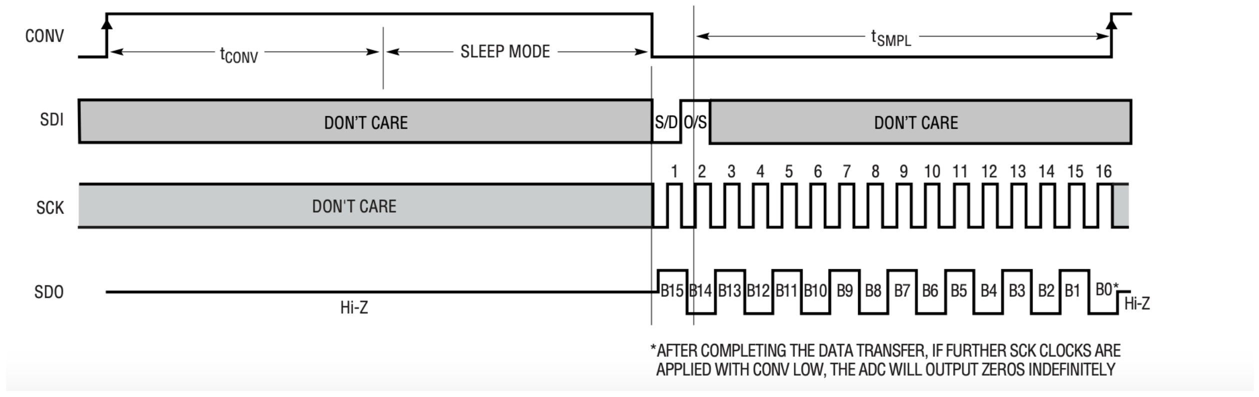
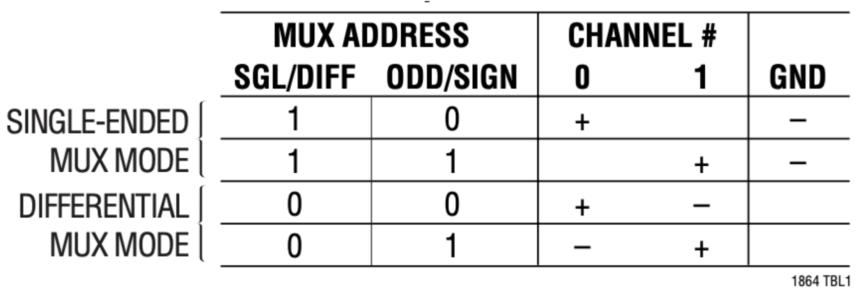

CONV
t_{CONV}
SLEEP MODE
t_{SMPL}
SDI
DON'T CARE
S/D
O/S
DON'T CARE
SCK
DON'T CARE
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
SDO
Hi-Z
B15 B14 B13 B12 B11 B10 B9 B8 B7 B6 B5 B4 B3 B2 B1 B0*
Hi-Z
*AFTER COMPLETING THE DATA TRANSFER, IF FURTHER SCK CLOCKS ARE APPLIED WITH CONV LOW, THE ADC WILL OUTPUT ZEROS INDEFINITELY

Figure 2. Timing Diagram of ADC



	MUX ADDRESS		CHANNEL #		
	SGL/DIFF	ODD/SIGN	0	1	GND
SINGLE-ENDED MUX MODE	1	0	+		–
	1	1		+	–
DIFFERENTIAL MUX MODE	0	0	+	–	
	0	1	–	+	

1864 TBL1

Table 1. Multiplexer Channel Selection

The DAC has a data width of 24 bits. There are 4 control bits, 4 address bits, 14 data bits, and 2 don’t care bits. Since we will be using single-ended mode, to load in data to channel 0, we output 0011 0000 + DATA to the DAC. To load in data to channel 1, we output 0011 0001 + DATA.

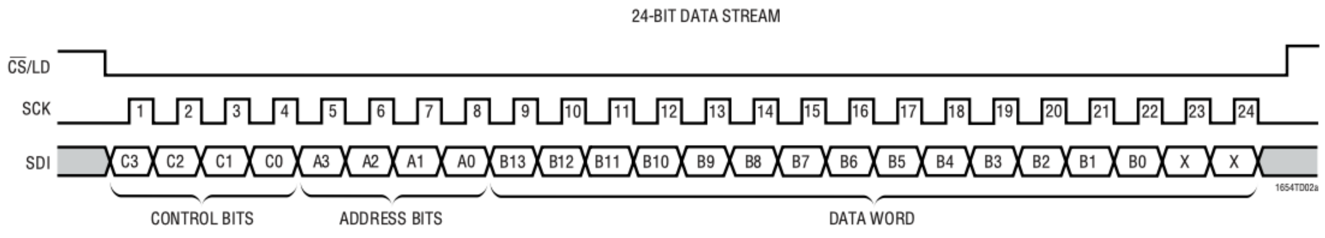


Figure 3. Timing Diagram of DAC

Two instances of SPI are instantiated in the block diagram. One is for ADC and the other is for DAC. The conv, sdi, sdo, and sck are set to be external ports for observation purposes. They are connected to the JA1 and JA2 on the FPGA board. The microblaze is connected to the SPI peripheral through the Axi4Lite Manager.

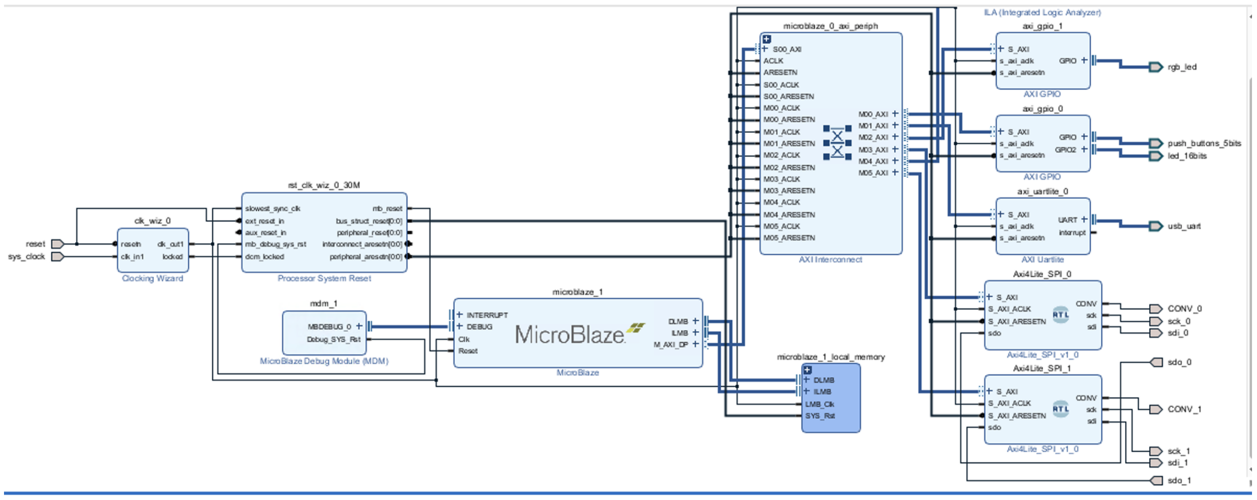
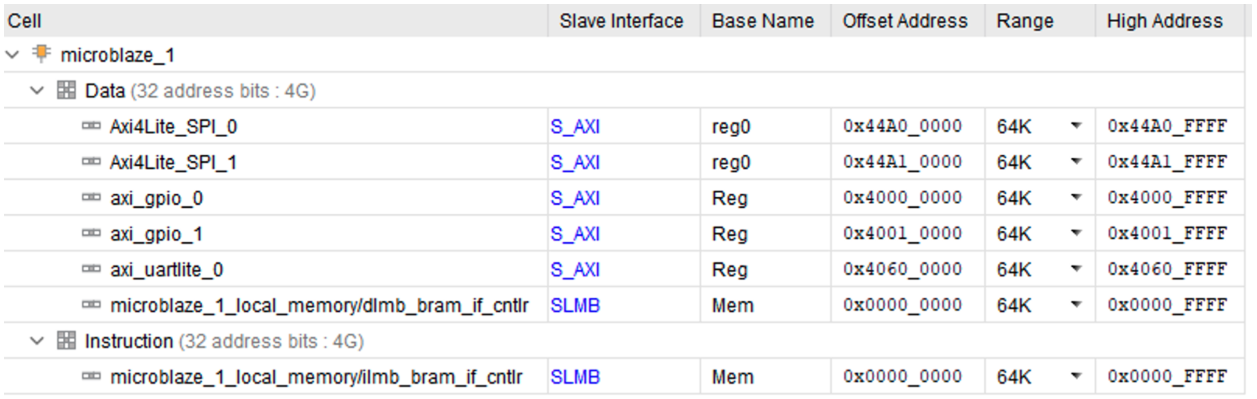


Figure 4. System Block Diagram

Axi4Lite_SPI_0 with base address of 0x44A0_0000 is connected to the ADC whereas the Axi4Lite_SPI_1 with base address of 0x44A1_0000 is connected to the DAC. When we read from or write to the FIR, we will use the offset addresses.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_1					
Data (32 address bits : 4G)					
Axi4Lite_SPI_0	S_AXI	reg0	0x44A0_0000	64K	0x44A0_FFFF
Axi4Lite_SPI_1	S_AXI	reg0	0x44A1_0000	64K	0x44A1_FFFF
axi_gpio_0	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
axi_gpio_1	S_AXI	Reg	0x4001_0000	64K	0x4001_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
microblaze_1_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	64K	0x0000_FFFF
Instruction (32 address bits : 4G)					
microblaze_1_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	64K	0x0000_FFFF

Figure 5. Addresses of Modules

Operation and Testing

An ADC tester and DAC tested are implemented by a previous student. A module-level testbench is developed to test the module. Then we instantiated the testers in the testbench for validation purposes. We first write the number of cycles for one period of the CONV signal. For a 50K aggregate sample rate, we will write 600 to the SPI given that the system clock is 30MHz.

$$30 \times 10^{6} / 50 \times 10^{3} = 600 \text{ cycles}$$

We then write data width to SPI, 16 bits for ADC and 24 bits for DAC. For ADC, we can start reading samples from the left and right channels. We implemented a handshake so that we only treat the data as valid data when the flag is set. For DAC, we also need to set the component to be in fast mode by setting the control bits to be b’0101. We then can spoon feed sine wave samples to the left and right channels of DAC. It has shown that the DAC and ADC are functioning correctly.

We also developed a SDK program to test the hardware implementation. From the microblaze, we will write the number of cycles to the SPI module.

Several helper functions for the ADC and DAC were made to allow easy communication with them. For the ADC the following were created:

- setAdcChannel(int ch): writes a control command to switch to channel ch
- setAdcCycles: sets clock rate for the peripheral (e.g. 600 for 50kHz)
- setAdcStatus(int status): enable or disables the ADC
- waitAdc(): waits for the ADC to return a done signal

Identical functions for the DAC were created except for a set channel helper as this is instead controlled when the data is written to the DAC.

Then we tested the ADC by collecting 25k samples from the function generator at 50Khz aggregate. Each channel is sampling signals at a 25Khz rate.

To test the DAC, we write a full-scale sine wave to the SPI and alternate the channels that we are writing to. We can then observe the output on an oscilloscope. We can see the output of DAC resembles the function generator output, but at half the frequency as expected, shown in Figure 6.

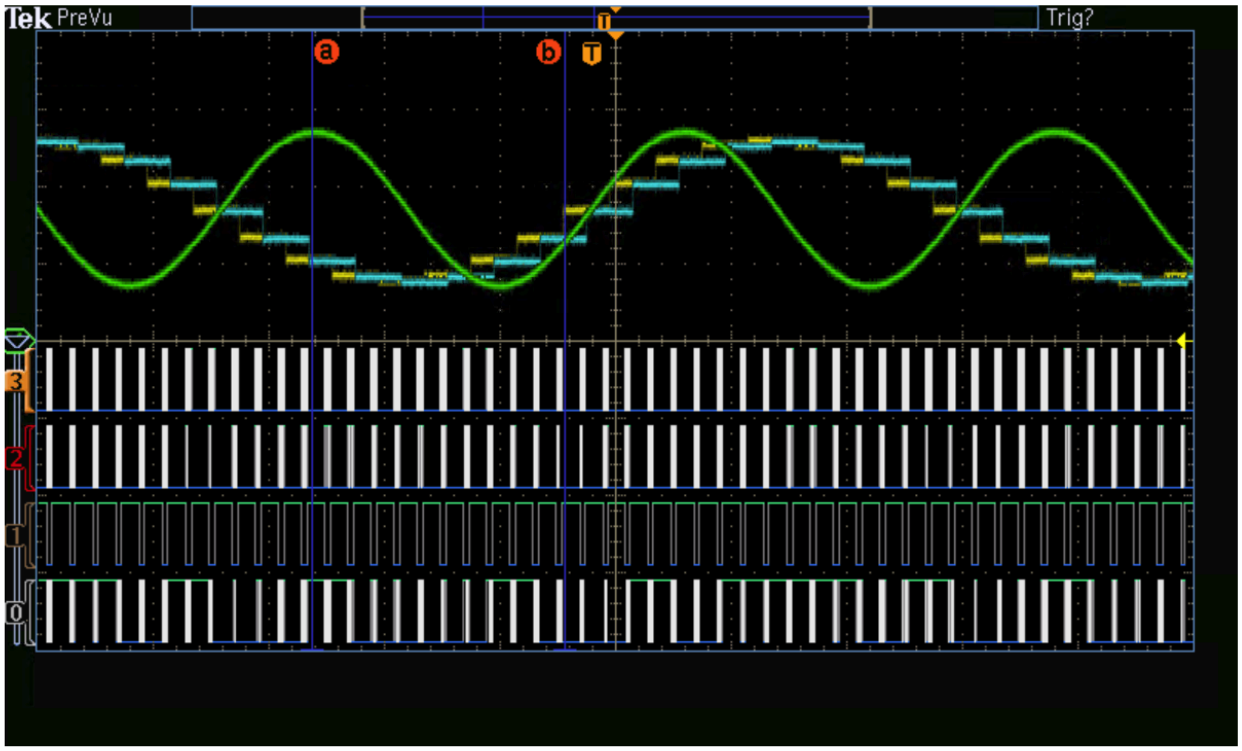


Figure 6. Oscilloscope Image of output of DAC

To implement the loop-back test, we first read data from the ADC channels: left and right. Then we write the exact same data to the left and right channels of the DAC, respectively.

The following code block shows the function that implements this behavior. After some initial configuration the function enters an infinite loop. To make the synchronization between the two simpler they both are controlled by completely separate blocks. If at any point done is read from the ADC, the data is sampled on a channel, that channel's variable is updated and then the channel is updated. A similar process for the DAC occurs where anytime it is ready it writes the stored left or right value to the left or right channel, alternating between the two.

```
void loopback() {
    setAdcCycles(600); // set both to 50kHz
    setDacCycles(600);
    setAdcStatus(1);   // enable both
    setDacStatus(1);

    int adcLeft = 0;    // last left value
    int adcRight = 0;   // last right value
    int adcChannel = 0; // next channel to read
    int dacChannel = 0; // next channel to write

    while(1) { // loop infinitely

        if (*adcDone == 1) {       // if adc is ready
            if (adcChannel == 0) { // left
                 setAdcChannel(0);
                 adcLeft = ((int)*adcData) & 0x00FFFF;
                 adcChannel = 1;
            } else {               // right
                 setAdcChannel(1);
                 adcRight = ((int)*adcData) & 0x00FFFF;
                 adcChannel = 0;
            }
        }

        if (*dacDone == 1) {       // if dac is ready
            if (dacChannel == 0) { // left
                 *dacData = 0x300000 | adcLeft;
                 dacChannel = 1;
            } else {               // right
                 *dacData = 0x310000 | adcRight;
                 dacChannel = 0;
            }
        }
    }
}
```

Now as the function generator is adjusted the DAC output adjusts in real-time. This can be seen in Figure 7. The cursors show a measurement of five steps on a single channel reading 199.8 microseconds. At 25kHz per channel, we have 40 microseconds per step and 200 microseconds for five as we observed.

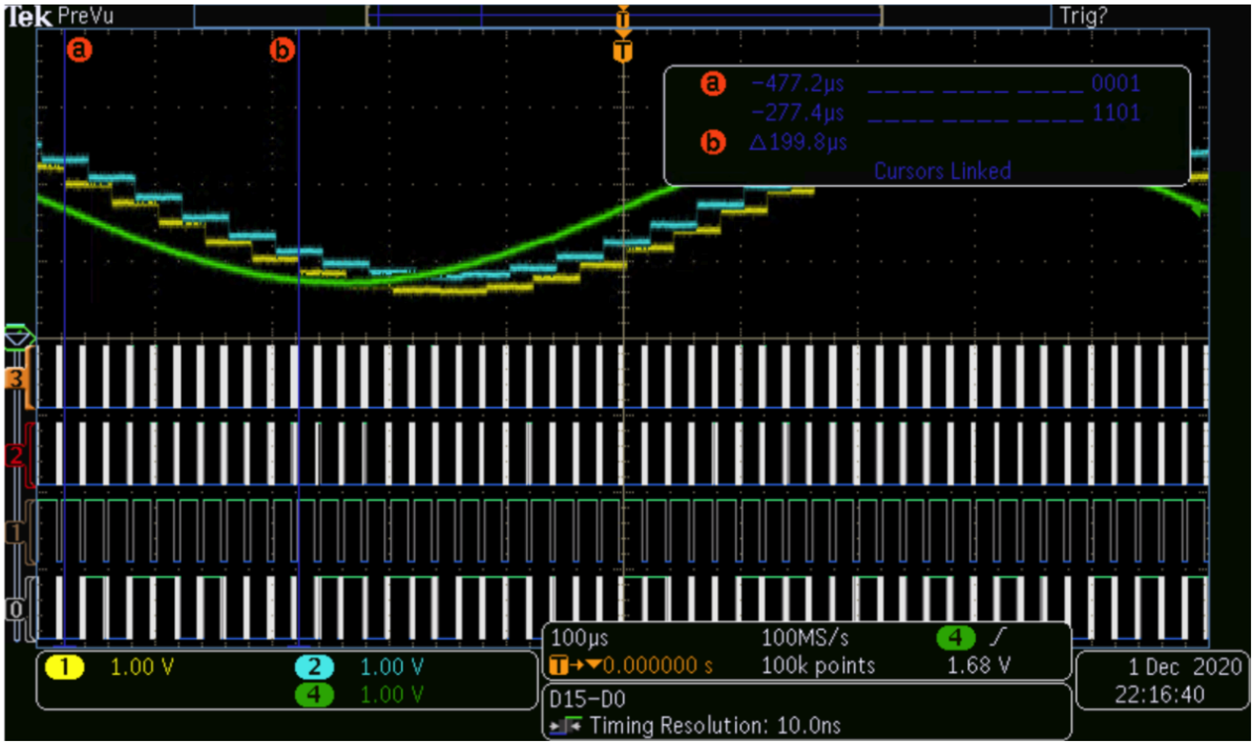


Figure 7: Oscilloscope Image of Loopback test

Discussion and Conclusion

Overall, we successfully implemented a SPI module that will be used for both ADC and DAC. There is a Axi4Lite supporter instantiated inside the SPI which will communicate with the Axi4Lite master in the testbench. On the other hand, there are 4 signals: sdi, sdo, sck, and conv which will communicate with the ADC or DAC. We first write the data width to the SPI to distinguish ADC and DAC. Then we will collect samples from the ADC. Then we write the samples to the DAC, the output of which will be displayed on the scope.

A module-level testbench is developed to test the functional correctness of the verilog code. Then we developed a SDK program to test the hardware implementation. A loop-back test was implemented which allows the ADC to collect samples then digitized those samples and send them to the DAC. Using an oscilloscope we can observe the output of the DAC and verify if that is consistent with the output of the function generator. As discussed in the previous section, the output is consistent with the output of the function generator.