EECS 151/251A FPGA Lab
Lab 6: AC97 (Audio) Controller and FPGA - IC Communication

Prof. Borivoje Nikolic
TA: Vighnesh Iyer
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

Contents

1 Before You Start This Lab

These documents will be helpful as you work through this lab.

1. **labs_fa16/docs/AC97_Spec.pdf**

The official specification of the AC97 protocol which you will partially implement in this lab. This is a large document but there are only a few sections that are important to us. They will be referenced later in the spec.

2. **labs_fa16/docs/AD1981B_Datasheet.pdf**

   The datasheet of the AD1981B AC97 Audio Codec. We will be writing a controller that lives on the FPGA which will communicate with this IC using the AC97 protocol. This IC receives audio data over the AC97 protocol and converts the digital audio data to an analog signal which is sent through the headphone jack.

We strongly recommend you read through this entire document before coming to lab.

2 Lab Overview

Run `git pull` in your **git cloned** `labs_fa16` directory to **fetch the latest skeleton files for this lab.**

In this lab, we will create a AC97 controller that lives on the FPGA. The controller will interact with an AC97 audio codec IC on the ML505 development board. Your controller will implement a portion of the AC97 protocol that will allow it to send digital audio data and control signals to the IC.

3 Introduction to the AC97 Protocol

AC97 is a protocol which is used for communication between a source/producer of audio data and an audio codec. The audio codec present on the FPGA development board (ML505) is the Analog Devices AD1981B Codec. Our processor will send digital packets to this codec using the AC97 protocol.

Refer to the AD1981B Codec datasheet and the official AC97 spec when working through this lab. They are in the `labs_fa16/docs` folder.

3.1 Protocol Connections

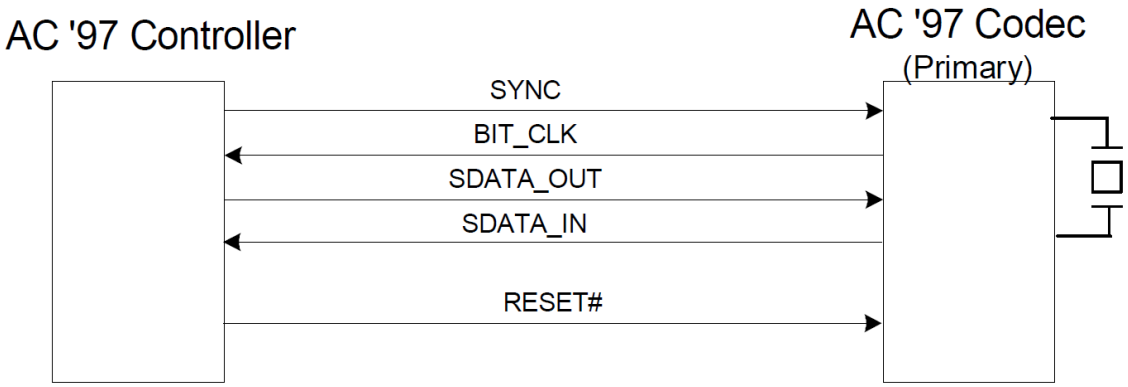

Figure 1: Codec - Controller Connections

There are 5 wires involved in the AC97 protocol. They are

1. `SYNC` - tells the codec when a frame is about to start
2. `BIT_CLK` - a clock from the codec which your controller should synchronize its data transfers with
3. `SDATA_OUT` - the serial line on which your controller transmits data to the codec
4. `SDATA_IN` - the serial line on which the codec transmits data to your controller (not used in this lab)
5. `RESET#` - a signal used by your controller to reset the codec

The way in which your controller transmits audio samples (linear PCM) to the codec is through the `SDATA_OUT` wire. There is a specific method of framing the samples so that the codec knows how to read them which is defined by the AC97 spec.

3.2 How Data is Transmitted

AC97 is a **serial interface**: data is transmitted to and from the codec one bit at a time. On every cycle of the AC97 bit clock (`BIT_CLK`), one bit of data is transfered from the AC97 controller (on the FPGA) to the codec over the `SDATA_OUT` wire, and one bit of data is transfered from the codec to the FPGA over the `SDATA_IN` wire.

The constant streams of data passing between the codec and the FPGA are divided into frames. The bit clock is generated by the codec, and runs at 12.288MHz. There are 256 bits per frame, so 48,000 frames are sent per second. This is where the 48kHz sampling rate of the codec comes from. Each frame sent to the codec provides one 20-bit audio sample for each of the DACs in the codec, and each frame sent by the codec provides one 20-bit sample from each of the codec's ADCs.

Frames are divided into twelve slots of 20 bits each, plus a 16-bit tag field, which serves as the frame header. Each slot serves a different purpose and contains various types of data to be sent to the codec. Each slot should be sent with the MSB first going down to the LSB. For example if you were to send data[19:0] in slot 1, you would begin by sending data[19] and finish by sending data[0].

The start of each frame is indicated by a rising edge of the SYNC signal. The SYNC signal goes high one clock cycle before the first bit of a frame, and goes low at the same time as the last bit of the tag field is sent. The diagrams below will summarize how frames are sent using the AC97 protocol.

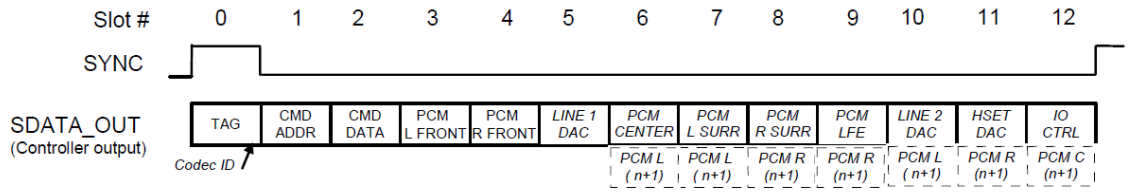

Figure 2: Framing for AC97

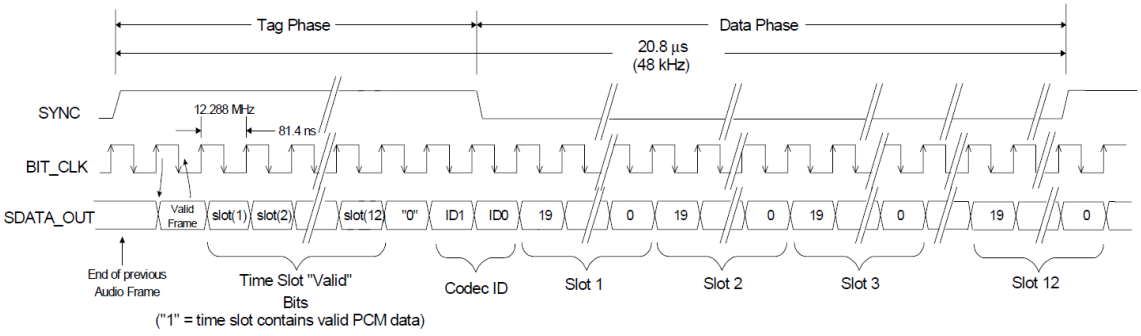

Figure 3: Framing for AC97 with Timing Details

3.2.1 Sending the Frame Tag

The bits in the tag slot are indicate which, if any, of the other slots in the frame are valid. The tag bits are assigned as follows:

Bit	Description	Value
15	Frame Valid	Should be 1 always
14	Slot 1: Valid Register Address	Should be 1 if we are writing or reading from a control register in the codec
13	Slot 2: Valid Register Data	Should be 1 if we are writing to a control register in the codec
12	Slot 3: PCM Left Channel Valid Data	Should be 1 if we have a sample to send to the codec
11	Slot 4: PCM Right Channel Valid Data	Should be 1 if we have a sample to send to the codec
10-0	Etc. Valid Bits	Should be set to 0

3.2.2 Setting Control Registers With Slot 1 + 2

After you send the correct tag through SDATA_OUT, you will then need to send slots 0 and 1. These slots contain an address and a value which refer to some control register on the codec. The codec contains a multitude of registers which control various features (volume, muting, etc.). Our controller will need to manipulate some registers to unmute the codec and to set the volume appropriately.

You should refer to the codec datasheet, specifically the table on page 12 to get the details of these control registers. We have specified below the registers that you will need to manipulate.

Reg Address	Description	Value
0x02	Master Volume	Should unset the mute bit and set right and left volume
0x04	Headphone Volume	Should unset the mute bit and set right and left volume
0x18	PCM-Out Volume	Should unset the mute bit and set right and left volume

In slot 1, the command register address needs to be specified as follows in 20 bits.

1. Bit[19] - Read/Write Command (1 = read, 0 = write)
2. Bit[18:12] - Control Register Index/Address (64 16-bit locations, addressed on even byte boundaries)
3. Bit[11:0] - Set to 0

The first bit (MSB) sampled by AC97 indicates whether the current control transaction in this frame is a read or write operation. The following 7 bit positions communicate the targeted control register address. The trailing 12 bits should be 0.

In slot 2, the command register data needs to be specified as follows in 20 bits:

1. Bit[19:4] - Control Register Write Data
2. Bit[3:0] - Set to 0

If we are writing data to a register, you must send the data with the MSB first in the first 16 bits of slot 2. If you are reading data, the entire slot 2 must be filled with 0s.

3.2.3 Sending Linear PCM (pulse code modulation) Samples in Slot 3 + 4

The next 2 slots (slots 3 and 4) are used for sending the actual audio samples you want the codec to push to the headphone output. Remember that the samples must be transmitted MSB first and each sample is 20 bits wide. Also keep in mind that each sample is a signed integer encoded with 2s complement, so the total range is roughly $2^{20}/2$.

Fill the remaining slots (slots 5-12) with all 0s for each 20-bit slot.

3.3 Codec Timing

The datasheet is useful for figuring out how the timing works with the codec. Pay close attention to the timing parameters and diagrams on pages 6-7 of the datasheet as your controller needs to be coded with those in mind.

Your controller will need to send bits on the rising edges of BIT_CLK, and the codec will sample them on the falling edge of the BIT_CLK. The BIT_CLK is provided by the codec.

3.3.1 Codec Reset

Your controller will have to perform a cold reset of the codec when it receives a reset signal from FPGA board (when you press the center compass button). The cold reset timing is critical, since if your controller doesn't hold the reset properly, the codec may lose its clock.

This timing diagram below is crucial to understanding how to reset the codec.

You will notice from the diagram, that the reset signal that your controller will send to the codec is active low, which means that to assert reset, you have to pull the signal low. You will need to pull the reset signal low for at least t_{RST_LOW} which is defined in the datasheet. While the reset

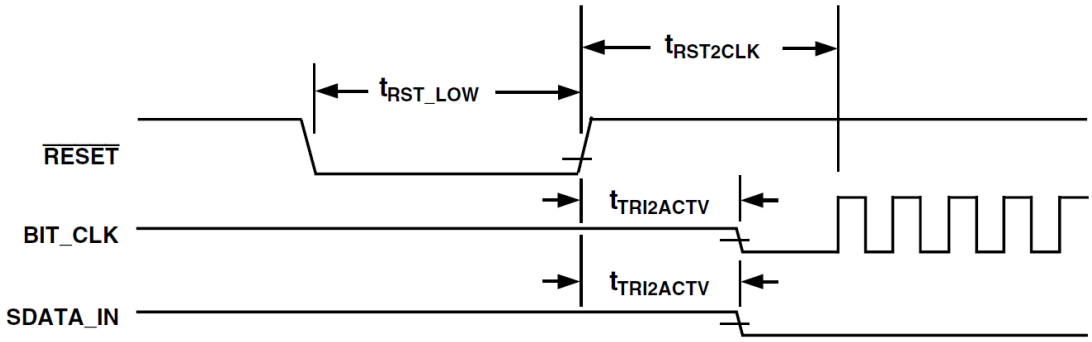

Figure 1. Cold Reset Timing (Codec Is Supplying the BIT_CLK Signal)

Figure 4: Reset Timing Diagram from Datasheet

signal is being asserted, keep in mind that the codec will stop producing the BIT_CLK and thus you cannot rely on it during reset; you must instead rely on the clock going into your processor for reset.

Once the reset signal has been asserted for the required time, after a delay of $t_{RST2CLK}$, the BIT_CLK will begin to oscillate, after which you can begin transmission. When the codec undergoes a cold reset, all its registers are set to their default values.

3.3.2 SYNC Signal for Frame Timing

The codec needs a way to know when a frame is about to begin so that it can interpret its content appropriately. This is done using the SYNC signal which is sent from your controller to the codec. You should assert SYNC for a total duration of 16 BIT_CLKs at the beginning of each audio frame. Refer back to Figure 6 to understand how the SYNC signal should be asserted with each frame.

3.4 Interfacing With RISC-V Processor

Our processor will talk with the AC97 controller through memory mapped IO, just as it does with the UART. The new memory map is below.

The AC97 controller is different from the UART in an important way. The AC97 controller interfaces the processor clock domain (50/100 Mhz) with the AC97 codec clock domain (12.228 Mhz). There could be registers in your design that are written to and read from using different clocks; you will need to make sure that your registers don't enter a metastable state.

Handling the bridge between the two clock domains (BIT_CLK and cpu_clk_g) is a difficult task that you will need to make sure goes smoothly as frequently as possible. To this end, to allow the CPU to write data to the codec, you can use an asynchronous FIFO if you are having a hard time implementing the ready/valid interface directly using synchronizers. The FIFO will isolate the read and write domains by having a separate read clock (BIT_CLK) and a separate write clock

Table 1: I/O Memory Map

Address	Function	Access	Data Encoding
32'h80000000	UART control	Read	{30'b0, DataOutValid, DataInReady}
32'h80000004	UART receiver data	Read	{24'b0, DataOut}
32'h80000008	UART transmitter data	Write	{24'b0, DataIn}
32'h80000010	Cycle counter	Read	Total number of cycles
32'h80000014	Instruction counter	Read	Number of instructions executed
32'h80000018	Reset counters to 0	Write	N/A
32'h80000050	AC97 Controller Control	Read	{31'b0, controller_ready}
32'h80000054	AC97 Transmit Sample	Write	{12'b0, sample}

(cpu_clk_g). This will make setting the ready/valid control signals easy.

We encourage you to look online for resources on crossing clock domains, metastability, and the design of an asynchronous FIFO. We are allowing you to use any resources to figure out the general block diagrams necessary to build the async FIFO; however, you cannot use any Verilog code you find online, and the entire implementation must be done by your group only. You can implement a FIFO of any number of entries, but powers of two are usually easier. Your FIFO should be able to hold at least 2 entries, but you should write your FIFO using parameters so you can extend its depth to be used in more situations.

Here are some articles that you may find useful for crossing clock domains (you can implement the ready/valid interface without using an async FIFO):

- Basic Intro to Crossing Clock Domains (using synchronizers)
- Synchronizer Techniques
- Understanding Clock Domain Crossing Issues
- Clock Domain Crossing
- Synchronization Techniques for FIFOs

3.5 Testing

You should begin by pulling the latest skeleton code from the staff Github repository. You should edit the `MakeFile` and set the `EECS151_CHKPNT` constant to `CHKPNT_2`.

3.5.1 AC97 Controller Without Processor

Your first goal should be to get the AC97 controller to work without using your processor. You should generate a square wave of a known frequency inside your controller and should output that square wave through the SDATA_OUT line. You should instantiate your AC97 controller in the top level module (ml505top.v) and should not synthesize your CPU (you can comment it out). You should be able to verify that your controller is performing properly by plugging in headphones to the board and hearing a clean square wave at your test frequency.

To get this working properly, it will help to build a model of the AC97 codec in Verilog and run simulations against this model. This checkpoint involves writing tests for your controller, which will be necessary to get the timing correct. Once you get this task working, you should think about ways to implement the ready/valid interface.

3.5.2 Testing AC97 Controller Ready/Valid Interface Without Processor

Next, move the square wave generation to the top level module and feed the square wave into the AC97 controller through the ready/valid interface. Try impacting this design to the board and you should be able to hear the same square wave as in the previous design. You should add tests for your ready/valid interface here.

3.5.3 Testing AC97 Controller With Processor Memory Map

Now, reintegrate the processor into your design and connect your AC97 controller to the processor in parallel with the UART. You should write a small test assembly or C program which generates a square wave and sends it to the AC97 controller through memory mapped IO. Once you impact this design on the board, you should still be able to hear the same square wave, but this time, your processor is the one that is generating it.

3.5.4 Testing Using Piano Program

Finally, load the program that we will supply to you which uses the keyboard connected to your workstation as a piano. Based on which keys you press, the program will instruct the processor to generate a sine wave of a particular frequency and send it to the controller.

4 Conclusion + Checkoff

You are done with lab 6! Please write down any and all feedback and criticism of this lab and share it with the TA. This is a brand new lab and we welcome everyone's input so that it can be improved.

4.1 Checkoff Tasks

1. Show the TA your working design with the FSM. Be able to transition states by clicking on the north and center buttons and show that your `music_streamer` matches the spec.