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.

This lab will be done in teams of two; it will involve building a peripheral used in the project.

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 AC97 controller will send digital packets to this codec using the AC97 protocol. The AC97 codec will decode the signals sent to it and send a voltage out through the headphone jack on the back of the ML505 board. You can plug in your headphones and listen to its output.

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

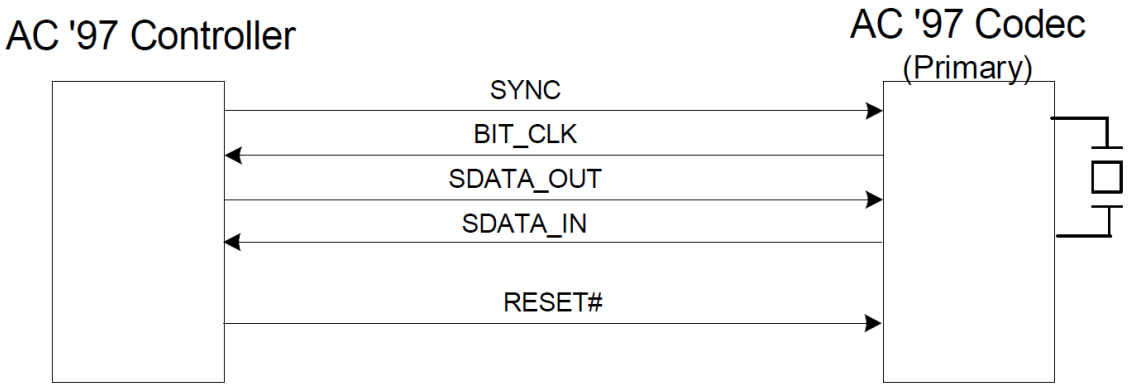

Figure 1: Codec - Controller Connections

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.

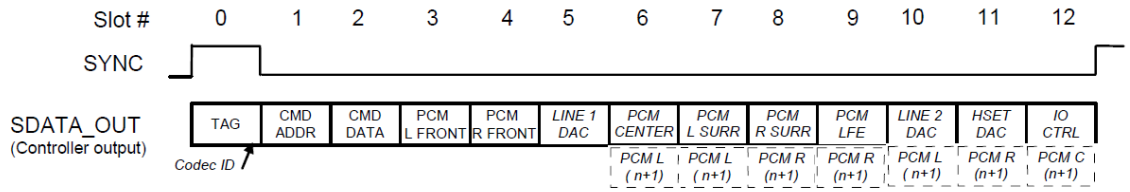

Figure 2: Framing for AC97

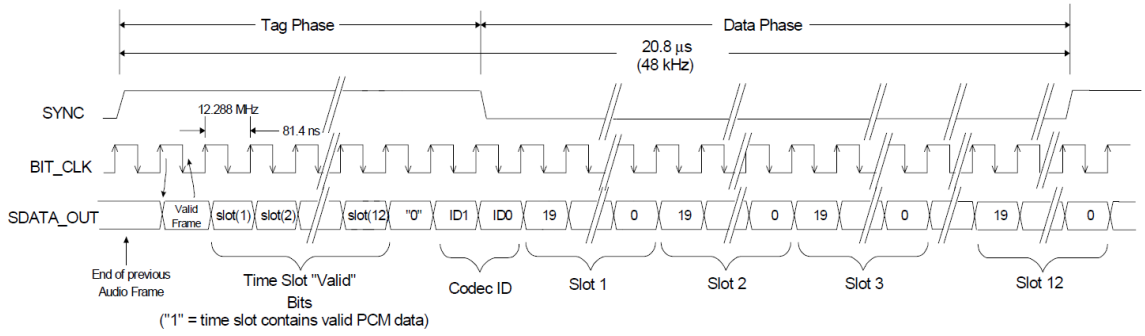

Figure 3: Framing for AC97 with Timing Details

3.3 Sending the Frame Tag (Slot 0)

The bits in the tag (slot 0) 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

3.4 Setting Control Registers With Slots 1 and 2

After you send the correct tag through `SDATA_OUT`, you will then need to fill slots 1 and 2. These slots contain an address and a value which refer to some control register on the codec IC. The codec contains a multitude of registers which control various features (volume/gain, mute, 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 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 (16 bits)
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.

3.5 Sending Linear PCM (pulse code modulation) Samples in Slots 3 and 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 note that each sample is a signed integer encoded with 2s complement, so the total range is roughly $\pm 2^{19}$. Each sample represents the amplitude of the audio wave to be played.

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

3.6 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`. Recall, that the `BIT_CLK` is provided by the codec.

3.6.1 Codec Reset

Your controller will have to perform a cold reset of the codec when it receives a reset signal from the FPGA board (when you press the `CPU_RESET` 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 critical to understanding how to reset the codec.

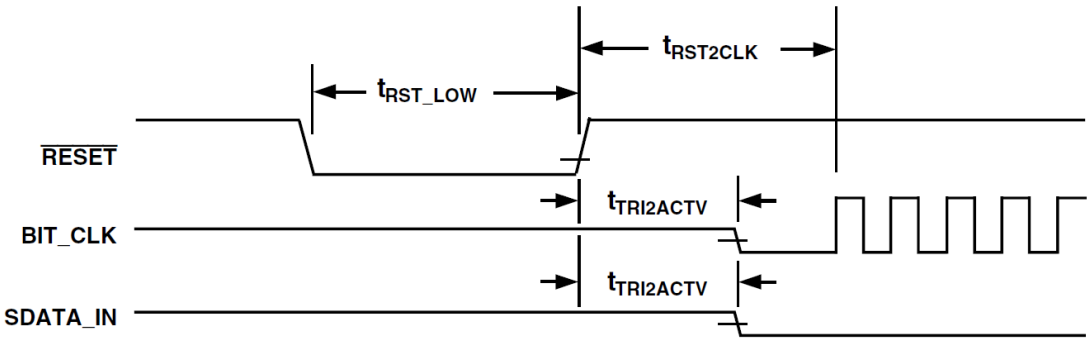

Figure 1. Cold Reset Timing (Codec Is Supplying the BIT_CLK Signal)

Figure 4: Reset Timing Diagram from Datasheet

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 the reset signal, you have to pull the line 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 signal is being asserted, the codec will stop producing the `BIT_CLK` and thus you cannot rely on it during reset; you must instead rely on a different clock to time the 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 the controller can begin transmitting frames to the codec. When the codec undergoes a cold reset, all its registers are set to their default values as specified in the datasheet.

3.6.2 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_CLK`s at the beginning of each audio frame. Refer back to Figure 3 in Section 3.2 to understand how the `SYNC` signal should be asserted with each frame.

4 Copy Over (Modify) Lab 4 Designs

You should copy over your lab 4 input processing circuits. Copy over your implementations of `synchronizer.v, debouncer.v, edge_detector.v`, and `rotary_decoder.v`. Then copy over your `tone_generator.v` and `music_streamer.v`. You can directly copy these files over from your `labs_fa16/lab4` directory.

If you use the 'rst' input in your synchronizer, debouncer, or edge detector, you should NOT use it for this lab. Instead of using a reset signal to initialize your module's registers, use a reg assignment so that your simulations continue to work, like this:

```
reg [width-1:0] prev_values = 0;
```

You might have to add this code to reset the 2D saturating counter register in your debouncer if you are getting Xs in simulation:

```
initial begin:INITIALIZE_SAT_COUNTER
        integer k;
        for (k = 0; k < width; k = k + 1) begin
                saturating_counter[k] <= 0;
        end
end
```

The reason for this is due to a feedback loop from the edge detector's pulse from a reset button press back to the debouncer, resetting the saturating counter and triggering another reset pulse. This was OK for the previous labs, but the AC97 codec is very sensitive to being reset frequently.

5 Building the AC97 Controller

Open up `labs_fa16/lab6/src/ac97_controller.v`. You will be implementing your AC97 controller in this module.

5.1 I/O Ports

- `sdata_in, bit_clk, sdata_out, sync, reset_b`

  These are the signals comprising the AC97 protocol. They behave as mentioned in the previous section. The `reset_b` signal flows directly to the AC97 codec and the `sdata_out, sync` signals go through a register in `ml505top` clocked by the `bit_clk` before going to the AC97 codec. The `reset_b` signal is an active low reset, which means that to assert the reset to the codec, you must pull the signal low.

- `system_clock, system_reset`

  The `system_clock` is a 33Mhz clock that will continuously operate. The `system_reset` is a 1 clock cycle reset pulse that is generated when you press the `CPU_RESET` button. The `system_reset` comes from a synchronizer → debouncer → edge detector chain.

- `volume_control [3:0]`

  This input will be explained in more detail in the Volume Control section. This signal is tied to the GPIO DIP switches 3, 4, 5, 6.

- `square_wave`

  This input comes from your `tone_generator` and is either 0 or 1, indicating the state of the square wave to play. Your AC97 controller will output this square wave input to the codec using the AC97 protocol.

5.2 General Operation

In the general case, the `bit_clk` will operate at its specified frequency. Your controller's job is to send AC97 frames to the codec. It will need to set `sdata_out` and `sync` in accordance to the AC97 specification.

For sending data to the codec, the AC97 protocol defines 2 20-bit slots for PCM data per AC97 frame. The data you send corresponds to digital inputs to the DACs on the codec. These slots should be filled with -250000 or 250000 based on whether the `square_wave` input signal is high or low. Since we are sending signed PCM samples to the codec, you will need a way of holding a signed number. To do this in Verilog, we can use the 'signed' keyword on a wire or reg net as such:

```
wire signed [19:0] pcm_data;
assign pcm_data = -20'd250000;
```

You are also in charge of setting the three previously specified registers of the AC97 codec so that the volume level is set properly and so that the output is unmuted. It is recommended that you

write to these registers in a loop so that you are performing a register write every frame; this will make your controller simpler.

5.3 Resetting the Codec

When we assert the `reset_b` signal to the codec, the `bit_clk` stops oscillating and only begins again after the `reset_b` signal has been deasserted. However, we need to precisely time how long the reset signal is asserted. To do this, we will use the `system_clock` which is always oscillating, and use the `system_reset` input to trigger a codec reset.

You should read the datasheet for the codec and understand how to reset it. Make sure you reset the codec for the required amount of time.

Be careful, however, since resetting the codec too frequently or for too long can cause the codec to lose its clock and stop operating. If this happens to you, you can recover by re-running `make impact`.

5.4 Resetting Controller Registers

When the `system_reset` is asserted, you should only touch one register, which is your reset clock counter that times the reset. Other registers that are modified by the posedge of the `bit_clk` shouldn't be touched.

In order to get simulation working and your other design registers (frame shift register, frame counter, etc.) in your design to a default state, you should use reg assignments.

```
reg [255:0] frame_to_send = 256'd0;
```

5.5 Clock Crossing

The design of the AC97 controller is tricky since it involves 2 clocks. Generally, you do not want registers that are set by one clock, to be used in combinational logic and then registered by another clock. However, we have that circumstance in this design.

Our `music_streamer` and `tone_generator` are being clocked by the 33 Mhz system clock, while the logic that sends data over `sdata_out` is being clocked by the `bit_clk`. Ordinarily, this would pose a problem, since the `square_wave` register is being set in the system clock domain, and its value is being read, used, and registered in the `bit_clk` domain.

However, in this particular case, since the `square_wave` register updates are so infrequent with respect to both the clocks, and a brief glitch in the square wave would make hardly any impact on the audio output, we can use the `square_wave` value in the `bit_clk` domain without much worry.

This happy circumstance will not persist into the project. In the project you will design a more sophisticated synchronization scheme to move multiple bits in parallel from the system clock domain to the `bit_clk` domain using an async FIFO.

5.6 Volume Control

Your AC97 controller needs to unmute and set the volume of the audio output using three registers in the codec. These registers should be set as such (look at the datasheet for the details):

Reg Address	Description	Value
0x02	Master Volume	Unmute and set R/L attenuations equal to `{1'b1, volume_control}`
0x04	Headphone Volume	Unmute and set R/L attenuations equal to `{1'b1, volume_control}`
0x18	PCM-Out Volume	Unmute and set R/L gain to 0dB

WARNING: If sufficient attenuation isn't applied, the codec may supply too much voltage to your headphones and destroy them. Check with the TA if you are unsure about the values written to the registers.

5.7 SYNC Signal Timing

As a note, if you create registers for the `sync` and `sdata_out` signals, make sure that they are both registered the same number of times and march in line.

You will need to make sure the `sync` signal is applied properly as you send each AC97 frame. Refer to Figure 3 in Section 3.2 for details. You should raise the `sync` signal on the last bit of the previous frame and hold it high until bit 15 has been sent of the next frame.

5.8 Testing in Simulation

To test our AC97 controller, we will be using a **block-level testbench**. This testbench will simulate our AC97 controller in isolation to see if it performs to spec.

The source for this testbench is `lab6/src/ac97_controller_testbench.v`. Run the testbench as usual by running `make` in the `lab6/sim` directory. This testbench isn't self checking so you should look at the Verilog file and manually inspect the output waveform to verify that all specifications are met. You should extend the testbench so that it sends more stimulus to the AC97 controller (like changing the volume).

You should make sure that ALL the signals in your AC97 controller are well defined (not X) for the entirety of the simulation.

6 Connect the music_streamer and tone_generator

6.1 Testing in Simulation

We previously used a block-level testbench to just test the AC97 controller in isolation. Now we will use a **system/chip-level testbench** to test our entire system at a higher level. This dichotomy

is similar to the difference between unit tests and integration tests in software.

Take a look at the source for this testbench in `lab6/src/system_testbench.v`. You will notice that instead of instantiating the AC97 controller, it instantiates `ml505top`. Now, the stimulus that we supply to our DUT represent signals coming into the FPGA.

This testbench will simulate the entire system which connects your AC97 controller to the `tone_generator`, which is connected to the `music_streamer`, which has connections from your input conditioning circuits (synchronizer, debouncer, edge detector, rotary decoder). This testbench isn't self-checking either, so you should add stimulus to the DUT to make it respond in some way and see that it is reflected in the waveform.

You should make sure that ALL of the signals in your entire system are well defined (not X) for the entirety of the simulation.

7 Try it on the FPGA!

With both simulations indicating that everything seems to be working, run `make` and `make impact` from the `lab6` directory. When your design is on the FPGA, your music streamer FSM from Lab 4 should still work as usual. By turning on the 1st and last DIP switches, the piezo output will be enabled. But now, as long as the 1st DIP switch is turned on, you can plug in a pair of headphones to listen to the output of the music streamer FSM.

You can use DIP switches 3-6 to change the volume of the audio output. Keep in mind that making all the switches high will give you the max attenuation, which corresponds to the quietest volume.

8 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.

The differences between a block and system level testbench will be critical in the testing of your project. You will use both types of testbenches extensively to verify correct behavior of all the modules you write.

8.1 Checkoff Tasks

1. Show the TA your working AC97 controller by demonstrating the state machine in the `music_streamer` working and sending audio to the AC97 codec.