

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	2
2	Lab Overview	2
3	Introduction to the AC97 Protocol	2
3.1	Protocol Connections	3
3.2	How Data is Transmitted	3
3.3	Sending the Frame Tag (Slot 0)	4
3.4	Setting Control Registers With Slots 1 and 2	5
3.5	Sending Linear PCM (pulse code modulation) Samples in Slots 3 and 4	6
3.6	Codec Timing	6
3.6.1	Codec Reset	6
3.6.2	SYNC Signal for Frame Timing	7
4	Copy Over (Modify) Lab 4 Designs	7
5	Building the AC97 Controller	8
5.1	I/O Ports	8
5.2	General Operation	8
5.3	Resetting the Codec	8
5.4	Resetting Controller Registers	8
5.5	Clock Crossing	8
5.6	Volume Control	8
5.7	SYNC Signal Timing	9
5.8	Testing in Simulation	9
6	Connect the music_streamer and tone_generator	9
6.1	Testing in Simulation	9
7	Try it on the FPGA!	9

8 Conclusion + Checkoff	9
8.1 Checkoff Tasks	9

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 later 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 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

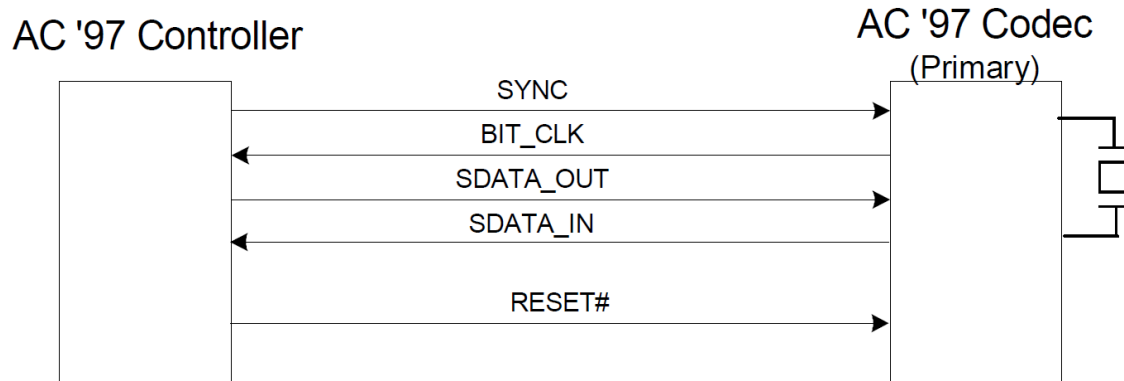


Figure 1: Codec - Controller Connections

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 transferred from the AC97 controller (on the FPGA) to the codec over the **SDATA_OUT** wire, and one bit of data is transferred 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.

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.



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.

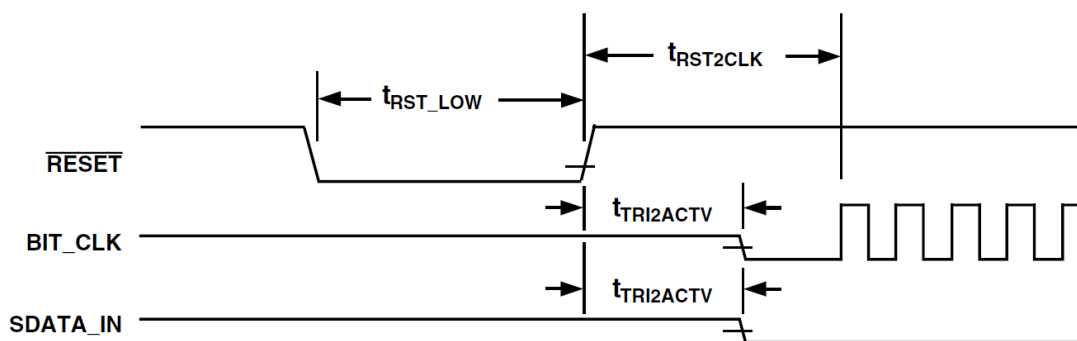


Figure 1. Cold Reset Timing (Codec Is Supplying the BIT_CLK Signal)

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_CLKs 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 should 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 `m1505top` 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

5.3 Resetting the Codec

WARNING: Do not assert your ac97 controller's codec reset signal (`reset_b`) longer than `tRSTLOW`. If you assert cold resets to the codec IC too often it will lose its clock and refuse to operate correctly. If this happens to you, just run `make impact` again to restore the ac97 codec.

5.4 Resetting Controller Registers

5.5 Clock Crossing

5.6 Volume Control

WARNING: Attenuate the audio using the control registers otherwise you may destroy your headphones by blowing the drivers with too much power!!! Check with the TA if you are unsure if you have attenuated your volume levels appropriately.

5.7 SYNC Signal Timing

WARNING: If you make sync, reset, and sdata_out registers in the controller, they must ALL be registers and they should march in line.

5.8 Testing in Simulation

block-level testbench

6 Connect the music_streamer and tone_generator

6.1 Testing in Simulation

system/chip-level testbench

7 Try it on the FPGA!

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.

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.