M A S S A C H U S E T T S I N S T I T U T E O F T E C H N O L O G Y
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.111 Lab #3 Checkoff List

Please be ready with the following when checking off Lab #3:

1. A printout of your Verilog code (put your name at the top!) that will be collected by the staff and evaluated.

2. Have a sketch of a timing diagram showing how your circuit works during record and playback. Show what happens in each mode over a sequences of eight *ready* cycles.

3. Be prepared to demonstrate your recorder *without* the interpolating filter.

4. Be prepared to demonstrate your recorder *with* the interpolating filter.

During checkoff you may be asked to discuss one or more of the following questions:

1. The linear interpolator is a simple example of digital signal processing (DSP). How can the FPGA be used to implement more sophisticated DSP functions, e.g., how would you add echo (reverb) to your playback audio signal?

2. How would you use the labkit's on-board ZBT memory for audio signal storage, instead of the Xilinx's BRAM? What is the longest recording sample you could have using all the ZBT memory? (See the labkit page for info about the ZBT configuration.)

3. Can you use the labkit's Flash ROM for storing audio data? (See the Flash ROM datasheet on the labkit page for info about how that components works.)

4. Explain why adding the linear interpolator makes a qualitative difference in what one hears during playback.

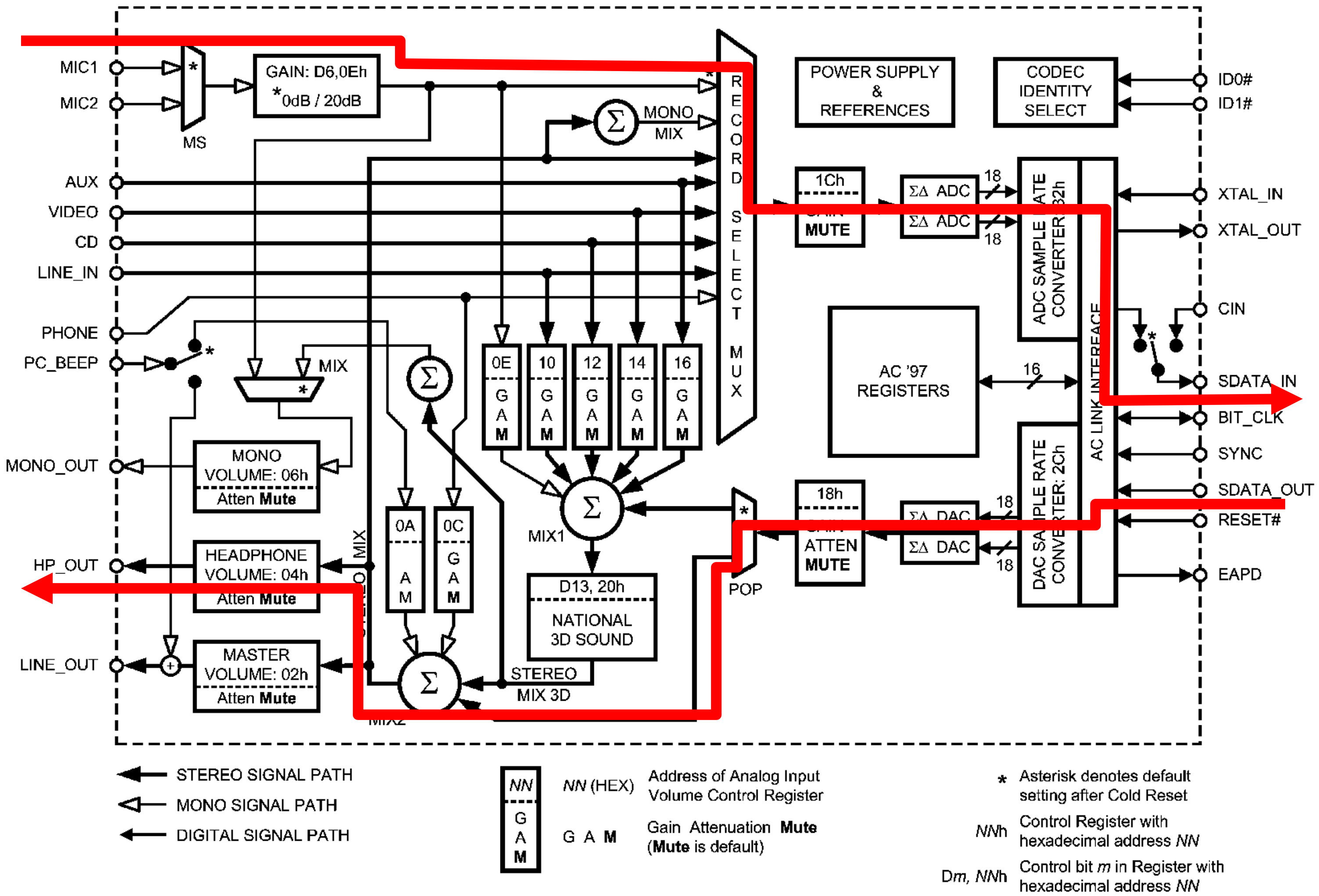
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.111 Lab #3

Goal: Build a voice recorder that records and plays back 8-bit digital audio samples.

Digital Audio Interface

Our labkit is equipped with an AC97 Audio Codec chip (a National Semiconductor LM4550) which serves as an interface between the analog world of traditional audio components (e.g., headphones and microphones) and the digital world of the FPGA. The block diagram of the LM4550 shown below has been marked up to show the processing paths we'll be using for this lab:



Incoming audio from microphone (top arrow, pointing left-to-right): the incoming audio signal from the microphone is boosted by +20dB by an on-chip amplifier and then selected as the input source for the two (one for each of the stereo channels) 18-bit sigma-delta analog-to-digital converters (ΣΔ ADCs). The ADCs sample the analog waveforms at 48KHz, digitize the sampled voltages, and output sequences of 18-bit two's complement numbers (referred to as the pulse-code modulated or PCM data). Each pair (left and right channel) of PCM samples is packaged along with other status data into a 256-bit frame which is then transmitted serially at 12.288Mhz (= 256 * 48Khz) to the FPGA via the SDATA-IN pin.

Outgoing audio to headphones (bottom arrow, pointing right-to-left): the FPGA transmits a 256-bit frame of serial data to the AC97 chip via the SDATA-OUT pin. Each frame contains two 18-bit fields with PCM data for the left and right audio channels. The PCM data is converted to two 48KHz analog waveforms by the sigma-delta digital-to-analog converters (ΣΔ DACs). The analog waveforms are amplified and sent to the stereo headphones.

So 48,000 times per second the AC97 codec provides two stereo PCM samples from the microphone and accepts two stereo PCM samples for the headphones. (Actually the microphone is a monaural source and so the same data appears on both the left and right incoming data streams.) It's the FPGA's job to keep up with the codec's data rates since the codec does not have on-chip buffering for either the incoming or outgoing data streams.

Voice recorder

The goal of this lab is to implement a voice recorder using the labkit's AC97 codec and FPGA. The top-level plan is pretty simple – when recording, store the stream of incoming samples in a memory, when playing back feed the stored data stream back to the codec.

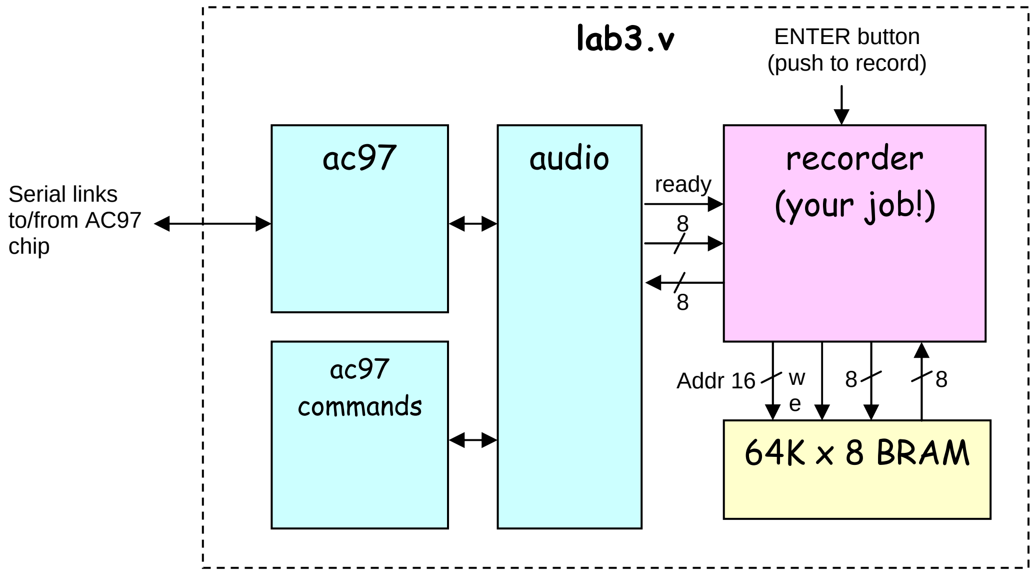
There are (of course) some interesting details:

- Let's use the FPGA's block rams (BRAMs) to build the memory for saved audio samples. A good size (i.e., one that fits in the FPGA we have) for the memory is 64K locations of 8 bits. To store a recording of respectable length let's store 8-bit samples at 6Khz, i.e., one eighth of the data rate of the incoming stream.

- The outgoing data stream wants samples every 48KHz , which we can produce by replicating each stored 6KHz sample eight times. But if we do that we'll hear lots of high frequency noise introduced by the large steps between successive samples in the 6KHz waveform. We can improve the sound dramatically by building a 48KHz data stream using linear interpolation between the 6KHz samples.

Implementation steps

1. Download lab3.v from the Handouts page. This file contains several Verilog modules:



lab3: a modified version of labkit.v that includes instances of the *audio* and *recorder* modules, hooking them up appropriately to each other and the pins connecting to the AC97 codec. The labkit's ENTER pushbutton is used as the record/playback button (push to record). You shouldn't need to modify this module.

debounce: used to debounce and synchronize pushbuttons.

audio: a wrapper around the *ac97* and *ac97commands* modules which implement the low-level interface to codec. This module has three ports of interest to us: a *ready* output that signals users of this module that a new sample is ready, and two 8-bit data ports, one for incoming monaural PCM data and one for outgoing monaural PCM data. You shouldn't need to modify this module.

ac97: interfaces with the AC97 codec, transmitting and receiving the 256-bit serial data streams. It has ports for both incoming and outgoing 18-bit stereo PCM data. You shouldn't need to modify this module.

ac97commands: generates a repeating sequence of writes to the AC97 command registers that perform the appropriate initialization. In this case, that includes selecting the microphone as the input source, setting the correct amplifier gains, etc. You shouldn't need to modify this module.

tone750hz: supplies a 20-bit PCM stream which if played at 48KHz produces a 750Hz sinewave. You shouldn't need to modify this module.

recorder: You'll be modifying this module to implement the necessary functionality. The supplied module tests the basic functionality of the microphone and headphones. In playback mode, this dummy module sends a

750Hz tone to the headphones. In record mode, it loops incoming samples back to the outgoing data stream, so you should hear your voice in the headphones. The module has the following ports all of which are synchronous with clock_27mhz:

clock_27mhz	input	system clock
reset	input	1 to reset the module to its initial state
playback	input	1 for playback, 0 for record
ready	input	transitions from 0 to 1 when a new sample is available
from_ac97_data[7:0]	input	8-bit PCM data from the microphone
to_ac97_data[7:0]	output	8-bit PCM data to the headphones

2. Using the Xilinx tools, build a lab3 project, compile lab3.v and load lab3.bit into the labkit. Plug in the headphone and microphone plugs from the headset into the appropriate jacks on the left-hand side of the labkit. You should a hear a 750Hz tone in the headset. Pushing the ENTER pushbutton should silence the tone and instead you should hear sounds picked up by the microphone.

3. Using the steps outlined in lecture, build a single-port 64K x 8 memory component using BRAMs.

4. Modify the *recorder* module to implement basic record and playback functionality.

   Record mode: When entering record mode, reset the memory address. When the *ready* input transitions from 0 to 1, a new sample from the microphone is available on the *from_ac97_data[7:0]* inputs. Store every eighth sample in the memory, incrementing the memory address after each write. You should also keep track of the highest memory address that's written.

   Playback mode: When entering playback mode, reset the memory address. When the *ready* input transitions from 0 to 1, supply an 8-bit sample on the *to_ac97_data[7:0]* outputs. For now, read a new sample from the memory every eight transitions of *ready*, i.e., upsample the 6KHz samples to 48KHz using simple replication. When you reach the last stored sample (compare the memory address to the highest memory address written which you saved in record mode), reset the address to 0 and continue – this will loop through the saved data again and again.

   Test your code. The playback will not be very intelligible given the high frequency noise introduced by the large steps in the 6KHz waveform.

- Modify your *recorder* module to generate playback samples using an 8-step linear interpolation between successive samples in memory. Use one of the switches to control if your interpolator is used during playback. If the successive samples are

S1 and S2 (S1 being the older of the two), then your module should output the following data over the course of 8 cycles of *ready:*

S1 = (8 * S1) >> 3   (the right shift by 3 divides by 8)
.875 * S1 + .125 * S2 = (7 * S1 + S2) >> 3
.750 * S1 + .250 * S2 = (6 * S1 + 2 * S2) >> 3
.625 * S1 + .375 * S2 = (5 * S1 + 3 * S2) >> 3
.500 * S1 + .500 * S2 = (4 * S1 + 4 * S2) >> 3
.375 * S1 + .625 * S2 = (3 * S1 + 5 * S2) >> 3
.250 * S1 + .750 * S2 = (2 * S1 + 6 * S2) >> 3
.125 * S1 + .875 * S2 = (S1 + 7 * S2) >> 3
In general, on cycle *i* (for *i* from 0 to 7) you should output ((8–i)*S1 + i*S2)>>3.

At this point S1 ← S2 and S2 ← new memory sample, and the whole process repeats for another eight cycles of *ready*.

NOTE: the samples from the AC97 are in 8-bit two's complement format, i.e., they range in value from -128 to +127. BUT the default Verilog data type is unsigned and arithmetic circuits for unsigned operands are, in general, different than arithmetic circuits for two's complement operands.

Using a feature introduced in the Verilog 2001 specification (which, happily, issupported by the Xilinx tools), you can solve this problem by adding a **signed** modifier to your register declarations so that Verilog knows that those quantities are in two's complement format, e.g.,

```
reg signed [7:0] s1, s2;
```

Now when, say, you use the "*" operator in your Verilog code, the Xilinx tools will generate the circuitry for signed multiplication instead of unsigned multiplication.

5. [optional] Have your recorder module record continuously when in record mode and then playback the last 11 seconds when you switch to playback mode – sort of an instant reply of the most recent part of a conversation.

Implementation Tips

After coding, examining the waveforms in simulation before attempting to program everything onto the FPGA can save you a lot of time. In particular, closely examine what happens when processing an incoming sample and generating a new outgoing sample (i.e., what your logic does just after a low-to-high transition of *ready*). It's pretty easy to generate a known sequence of *from_ac97_data* values and ensure that they get written to your memory in record mode and get played back correctly in playback mode. Check that all control signals rise and fall as you would expect them to. Another good time to use the simulator: examining the values produced by your linear interpolator – you should see the appropriate intermediate values created over 8 *ready* cycles using two successive stored values.

If your circuit seems to work under simulation but not when loaded into the labkit, try bringing critical signals out to the logic analyzer connectors, e.g., the signals for your 64Kx8 memory.

A good way to debug the interpolator is to use *ready* and *to_ac97_data[7:0]* to drive one of the logic analyzer connectors. Hook up the analyzer clock lead to *ready* and the data leads to *to_ac97_data*, and clock the data on the falling edge of *ready*. You can display the 8-bit data as a "magnitude waveform" in which the logic analyzer will plot the captured data values as a waveform. Zooming in, you should see the waveform as short straight line segments each made up of 8 points as your linear interpolator interpolates between the stored samples. There shouldn't be any big jumps between one captured value and the next if your interpolator is doing its job correctly.

In general, using the logic analyzer to examine what's happening is a quick way to "see inside" your chip and get some idea of what's going on.