525.742 SOC FPGA Design Lab

Laboratory Project 2

Audio Playback System

**Introduction**

In this laboratory assignment, the student will build the foundations of an audio playback system for the Embedded FPGA processor.  This will involve using your Lab 1 design to act as a low-level controller for the DAC on the Zybo Z7-020, interfacing it to the processor processor through an AXI / AXI Stream interface, and then writing some simple software to demonstrate the setup.  The result will be enabling the processor to easily play CD quality audio.

**Learning Goals**

* gain experience with software/hardware co-design
* get a feel for how quickly large tasks can be accomplished using an SOC approach.
* gain some experience interfacing to a typical Audio CODEC and using standard serial interfaces (I2S, I2C)
* gain experience using the AXI and AXI-Stream interfaces as means to connect IP

**Hardware Needed**

* Zybo Z7-020
* 3.5 mm stereo cable or any pair of headphones or amplified speakers
* USB Cable

**Useful References**

* [Zybo Reference Manual](https://reference.digilentinc.com/_media/reference/programmable-logic/zybo-z7/zybo-z7_rm.pdf)
* Analog Devices SSM2603 Audio Codec [datasheet](https://www.analog.com/media/en/technical-documentation/data-sheets/SSM2603.pdf).
* Xilinx AXI IIC peripheral [documentation](https://www.xilinx.com/support/documentation/ip_documentation/axi_iic/v2_0/pg090-axi-iic.pdf)

**Functional Requirements**

The system shall be controlled via a serial port console interface. That is, a user will be able to control the operation of the program through numeric/text input and output over a serial port.

The system shall start by printing a welcome message with the student name and any instructions necessary to the user on the serial port console interface.

The system shall provide a mechanism for a user to load a small (< 16kbytes) digital audio file via the serial port.

The system shall be able to playback that file in two ways, both controlled via a command from the serial port :

    -- single playback of the file

    -- a continuous loop which plays the file over and over with no gap between iterations until interrupted by another serial command.

The system shall print the total number of samples loaded from the file.

Audio File Format:

* 4 bytes indicating number of samples in the file (valid range for this lab will be = 0 to 4096).  This length is stored Little Endian.  (This is the default endian-ness of the processor when used with AXI bus.  Conveniently it is also the same as your PC.)
* 4 \* number of samples bytes, which will be LeftLSB LeftMSB RightLSB RightMSB ...etc.

Two files in the above described format are attached to the assignment which will be used for checking out the audio player. tone48\_le.dat: which is 48 points of one cycle of a sine-wave. When played continuously, this should produce a perfect 1kHz sinewave on both channels. (or to be more precise, 125MHz/2560/48 = 1017.25kHz). The second file: cowbell\_le.dat plays the sound of a short cowbell clang.

Example:

System: "Welcome to the audio playback system,  press L to load a file, C to playback continuously."

User: "L"

System: "Send the file now from your terminal"

User: Sends a binary file that has size information followed by samples.

System: "Done Loading"

User: "C"

System: "Starting loop, press any key to stop"

<system plays the sound continuously>

**Suggested Approach (Not required - just one option for an incremental approach.  Student can certainly jump straight to the whole design if they are confident)**

Task 1: Instantiate your Lab 1: lowlevel\_dac\_intfc component in your top level HDL. Create a simple 8-point lookup table in hardware such that the lowlevel dac interface will cycle through the values  (0,7070, 10000, 7070, 0, -7070, -10000, -7070). This will eventually create a sinewave on the output of approximately 6kHz. (but not till the CODEC is set up properly). Write the necessary HDL to cycle through the samples in the table, putting each on the data\_word input to your lowlevel\_dac\_intfc at the appropriate time. (Hint: your component already provides a signal that indicates when it is ready for you to switch to the next sample.) Simulate to verify the system is behaving as desired.

Task 2.1 (HW): Add an IIC peripheral to your block diagram, use connection automation to wire it to the processor, and bring the clock and data signals out to your top level. Additionally create the necessary top level signals to wire signals from your lowlevel\_dac\_intf component to the corresponding CODEC pins. Consult the Zybo Reference Manual and uncomment/edit the necessary lines in your constraints file. Note that there is one additional signal you need to provide to the codec in order for audio playback to work that is not provided by your dac interface or the IIC peripheral. This is described in the Zybo reference manual.

Task 2.2 (SW): Now that you have everything connected correctly, write the software for to send new values to the codec config registers over the IIC interface as shown in the lecture slides. When this code is working, you should hear the 6kHz tone from the lookup table you built in Task 1.

Hint 1: Look at the Xilinx provided xiic\_low\_level\_eeprom\_example for one option on how to control the IIC peripheral from your software.

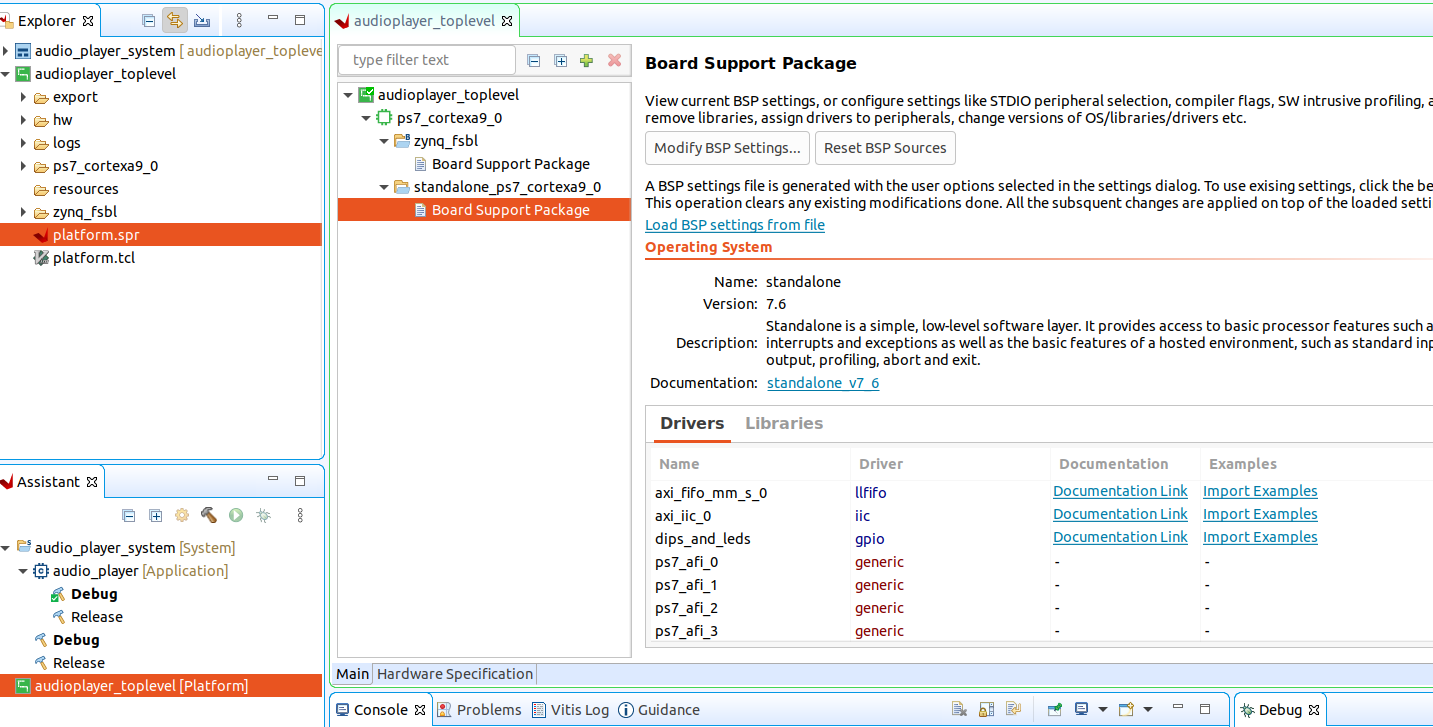
Hint 2: Consider using an Internal Logic Analyzer and comparing the signals sent by the IIC peripheral to the timing diagrams shown in the codec datasheet if you aren’t getting the results you expect

Task 3: Once you are confident in your DAC interface component and connection to the CODEC, it is time to hook it to the processor, such that the processor can provide the audio samples to it as they are needed. To enable the processor to be able to create axi-stream transactions,  Add an “AXI-Stream fifo” to your design as described in the lecture notes and the documentation.  Set this up so that writes from the processor (via AXI) go into the fifo, and the dac interface component reads from the fifo every time it is ready for a new data word.

Task 4: Test your Processor / DAC interface by writing the same 8 points to the FIFO over and over again, which should produce the same tone as in Task 2.  Once this works, the rest of the lab is a matter of writing the software in C to handle reading and writing the audio files.

Some References for Writing the Software:

When deciding how to use a peripheral, there are usually a couple of general options. The first option would be to take a look at the manual for the actual peripheral itself. This will usually describe the hardware, how it works, the registers, and the instructions for getting the peripheral to do something (write this register, then that...etc.). This is always an option, but for the more complex peripherals, it can be relatively complicated. For each of the peripherals, there are also drivers that are provided for you, that get built when you make the board support package. In some cases, there are even two sets of drivers – “low level” and “high level”. These drivers can often simplify things, and abstract the hardware well, but the downside is that they can be more complicated than necessary for simple peripherals where it is sometimes just easier to do it yourself. For this lab, we will take a look at both of these methods since we have a few different peripherals to use. Note, the place to find all of the documentation that I’ll be referencing here is in your “Platform” project, click on the .spr file and you will get a list of all of the peripherals in your hardware design, followed by the documentation and examples for each of the drivers that the BSP makes for you. (Screenshot below)



UART

So far, we’ve used the UART only by using the standard C printing functions. To extend the use of the UART to some of the things we need to do in lab 2, lets look at the low-level drivers

To read a byte from the ps7 uart, #include "xuartps.h” and there are a couple of nice functions you will see there: “XUartPs\_IsReceiveData” and “XUartPs\_RecvByte”. One can be used to see if there is a character/byte waiting from the serial port, and another to actually read a character/byte. This should be all you need to handle the user interface for the lab.

I2C

To use the i2c port, again, look in the file xiic\_l.h for a helpful function, xiic\_send. This function is passed a base address of the iic peripheral

AXI Stream FIFO

As with many of the peripherals , there are a some higher level drivers included in the board support package. That said, this is a simple enough peripheral to use (at least the way we are using it) that it makes more sense to just read the reference manual for the hardware peripheral itself and write registers in there ourselves to make it work.

A read from the “transmit vacancy register” will tell you whether there is space in the FIFO (and how much) , and a write to the transmit data register will write to the FIFO itself. As long as the FIFO was created to have CUT-THROUGH mode, then this should be all you need to do. Look at and #include "xllfifo.h” to find the addresses of these registers.