Objective

The educational objective of lab 9 is to investigate the effects of signal processing algorithms implemented in VHDL on an audio signal. Additionally, the process for creating and using a custom component that interfaces with the NIOS II on the Avalon bus will be reinforced.

Note: You must have lab 8 and the audio demo completed in order to complete lab 9.

Procedure

Part 1

In part 1, you will create an audio filter IP to be instantiated in QSYS to interface with the NIOS II. The audio filter is based on the filter designed in the previous lab, but will have an internal signal that chooses between the high pass coefficients and the low pass coefficients.

1. Create a new project in Quartus and copy into it one of the filter.vhd files from lab 8.
2. Modify the filter so that it can be either a high pass filter or a low pass filter based on the value of an internal signal. One suggestion is to put the coefficients as constants in two different arrays and multiplex between the two. The synthesizer will infer two ROMs.
3. Since this filter will become a custom component, there is only one way to write data into it, the writedata input. As such, you will need to have two registers that receive data from the writedata input. The first register stores the switch signal to determine high pass or low pass and the second register stores the incoming audio sample data. Each of these registers will need to have an address associated with them to indicate where the writedata should go when write (write enable) is active. The C program will be writing to the sample data register much more often than the switch register so it makes sense to give the sample data register address 0 and the switch register address 1.
4. Edit the entity of the filter to match the signal names and types expected by QSYS. The entity should match the following and connect with your modified filter as shown in figure 1.

```
entity audio_filter is
    Port (clk, reset_n     : in std_logic;
          write            : in std_logic;
          address          : in std_logic;
          writedata        : in std_logic_vector(15 downto 0);
          readdata         :out std_logic_vector(15 downto 0)
          );
    end audio_filter;
```

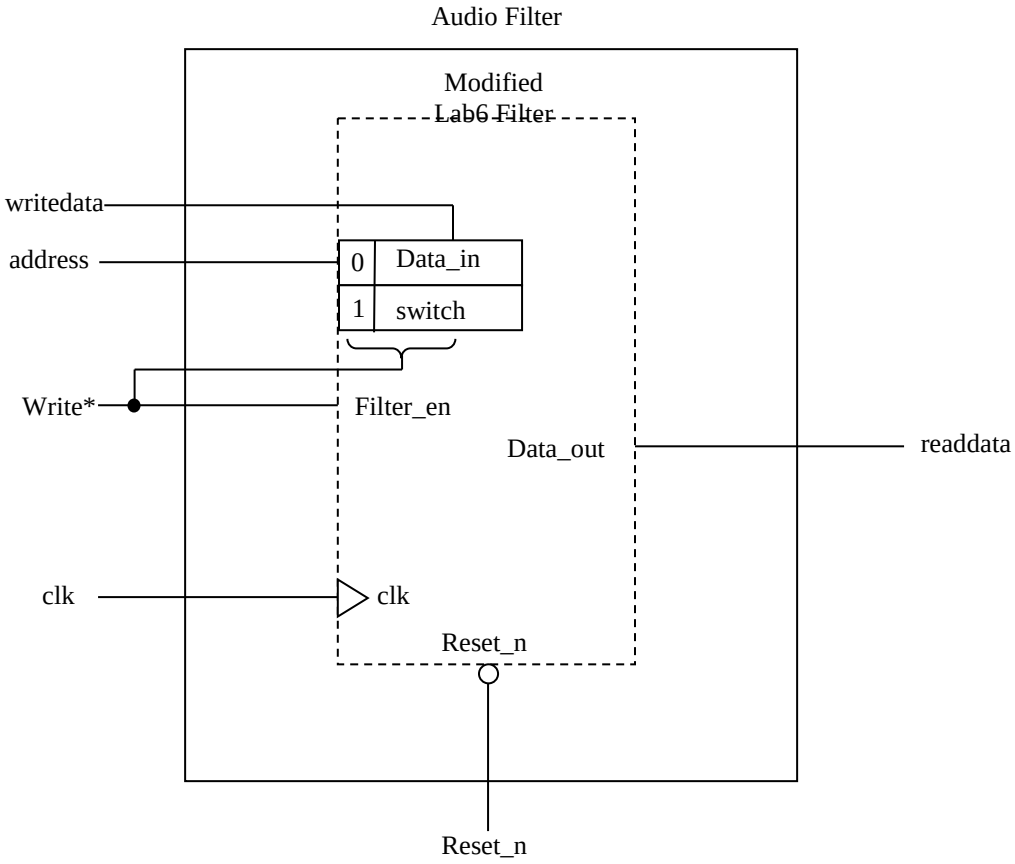

Figure 1. Connections of audio filter entity to the modified lab 8 filter

*Note that write is a reserved word, but it can and should be used

5. Verify that the audio filter compiles without errors or warnings
6. Open Qsys and copy the nios_system from the audio demo. Select New Component (in IP Catalog tab) for your audio filter and complete each screen. Don't forget to add the .vhd file and analyze it to verify it compiles without errors. The signal assignments should be recognized by the new component

editor. You may need to specify the reset association for the Avalon interface. At this point there should not be any errors or warnings.

7. Add the filter component to the system, connect the clk, resets and slave port. Save the system (remember there will be a warning) and generate the VHDL.
8. Add a PIO interface for the switches on the DE1-SoC board. Enable the PIO to generate an interrupt on either a rising or falling edge.
9. Generate the HDL.
10. Copy the top_level VHDL file from the audio demo into the Quartus project. Modify this file to include your newly generated nios_system.
11. Import the pin assignments, compile the top_level and download to the DE1-SoC board.

Part 2

In part 2 you will be writing a C program that does the following:

- Loads data from a .wav file into the SDRAM
- Reads audio samples from the SDRAM and sends them to the audio component at a 48K rate
- Interrupts when SW0 or SW1 changes state and sets the mode appropriately
  - If neither switch is high, there is no filtering
  - If SW0 is high, the audio should be filtered with the low pass filter
  - If SW1 is high, the audio should be filtered with the high pass filter
- Sends the input audio samples from the SDRAM to the audio filter and sends the output from the audio filter to the audio component when the switches indicate either type of filtering is enabled.

1. You may start with the modified audio_demo.c file provided in MyCourses.
2. Open NIOS II Software Build Tools for Eclipse. Create a new Application and BSP from template.
3. Open the BST editor (right click on Audio_Demo_App_bsp > NIOS II > bsp editor). Click on the **"Software Packages"** tab and click to enable **alter_hostfs**. This will allow you to read a file from the computer via the C program.
4. Click on the **"Linker Script"** tab and left click on each reference to **new_sdram_controller_0** in **Linker region names** area and change them to **onchip_memory**. This will ensure that all of the code exists in onchip memory so it does not get overwritten when the audio data is loaded into SRAM.
5. Generate the bsp and copy the new system.h to the App folder.
6. Create a C program to meet the specifications outlined above.

7. Build the project and test your system. If it is not working as expected, use Signal Tap and the NIOS II Software Build Tools for Eclipse to help with debugging. For example, you can use Signal Tap to verify the correct switch is set and the samples are being written correctly.
8. When debugging your system with the filter component, you may want to change MAX_SAMPLES from 0X80000 to 0X10000 to reduce the wait time for the audio file to load to the SDRAM.
9. You can use the .wav file provided with the audio demo, or you can supply another one. If you google it, directions for the process to convert a song to .wav through itunes can be found.

Demonstration:

1. Audio file plays back with no filtering.

2. Audio file plays back with noticeable low pass filtering.

3. Audio file plays back with noticeable high pass filtering.