Field Programmable Digital Audio Effects Rack

Andrew B. Shapiro        Marc P. Resnick

6.111 Final Project, Fall 2010

Abstract

This report presents the design and implementation for a Field Programmable Digital Audio Effects Rack using the Xilinx 2 series FPGA. With an 18-bit AC’97-compatible codec (audio) and a 24-bit VGA output (video), the FPGA can be used to synthesize the large, even cumbersome devices normally used by musicians and sound engineers to modify sound in real-time. Common audio effect tools such as signal generators, delay, and filters will be available to the user. Modular effects blocks and configurable dataflow allow for an extensible library and numerous arrangements of effects. A routing module allows for the use of a patch bay to configure the signal path, and a video display shows the activity of each effect’s input. The final result is a modular system for emulation of audio effect devices many times the size of an FPGA which is compatible and interchangeable with the devices it emulates.
## Contents

1. **Introduction** .................................................. 4

2. **Overview** ......................................................
   - 2.1 Effects .................................................. 5
   - 2.2 Routing .................................................. 6
   - 2.3 Display .................................................. 6
   - 2.4 Composition and Extensions .............................. 6

3. **Description** ..................................................
   - 3.1 Patchbay (Drew) ........................................... 7
   - 3.2 Routing (Drew) ........................................... 8
   - 3.3 Effects ................................................... 8
     - 3.3.1 Mix (Marc) ........................................... 8
     - 3.3.2 Signal Generator (Marc) .............................. 9
     - 3.3.3 FIR Filter (Marc) .................................... 9
     - 3.3.4 Delay (Marc) ........................................ 9
     - 3.3.5 Pan (Drew) ......................................... 11
   - 3.4 Display (Drew) ........................................... 11
   - 3.5 Demo Configuration ...................................... 11

4. **Testing and Troubleshooting** ............................... 11
   - 4.1 Use of the Signal Generator for Testing .................. 12
   - 4.2 Routing .................................................. 12
   - 4.3 Delay and ZBT RAM ...................................... 12

5. **Conclusions** .................................................. 13

6. **Appendices** .................................................. 13
   - A. Code Listing (alpha-order) ............................... 14
     - A.1 Main Audio Support (audio.v) ........................... 14
     - A.2 Router (audio_router.v) ............................... 21
     - A.3 Main Labkit File (audiofxbox.v) ....................... 23
List of Figures

1  Overview of Demonstration Configuration. LPF is an instance of a Low-Pass Filter. The effects such as Delay (D), Signal Generator (SG), and other Filters (FIR) receive their inputs from \texttt{input\_vector} and pipe their outputs into \texttt{output\_vector}. 7

2  Sliding Window of ZBT Memory used by Delay. 10
1 Introduction

The Field Programmable Digital Audio Effects Rack is a system designed to emulate, replace, or work alongside a rack of dedicated audio effect devices. The patch bay and signal routing provide a familiar method of directing an audio signal through many effects. This implementation includes a signal generator, delay, filters, pan, and mix, allowing for both the modulation of sound input and synthesis of new sound. Additionally, multiple modules can be composed to create new effects, and the entire system is designed to be extensible; additional modules can easily be created and integrated into the system.

Background

Since the early 20th century, electronic music and instruments have risen in popularity. From the theremin, to the tape recorder, to the synthesizer, and so on, new musical techniques have accompanied technological advances. In order to integrate music with computers and other digital hardware, a technique called Digital Signal Processing (DSP) is used to manipulate audio in the form of quantized samples, rather than a continuous analog signal. Using an FPGA, the Field Programmable Digital Audio Effects Rack performs DSP to create and route electronic music effects.

Motivation

The Field Programmable Digital Audio Effect Rack emulates a “patch panel.” This type of devices allows the user to reconfigure the order of effects in real time using a physical interface. Most patchbays literally connect the ports of separate audio modules such as the effects we have implemented for this project. However, we forego sending audio data into an external routing of physical cables and instead use a routing table to internally pass signals to the appropriate audio effects. This approach avoids the issue of noise due to cable or patchbay electrical imperfections, i.e., the click one would here when disconnecting the input to the speaker.

2 Overview

The Audio Effects Rack can be divided into four main components: Audio Support, Audio Effects, Routing, and User Interaction. The Audio Support modules interact with an AC’97 codec chip which uses 18-bit resolution ADC’s and DAC’s; audio samples are converted from the audio input jack and produced
at the audio output jack at a rate of 48kHz. The **Audio Effects** that this project implements are a foundational set of modules essential for an effective demonstration of the live routing features. The **Routing**-related modules orchestrate the updating and enforcing of routing configurations, and the **User Interaction** component allows audio channel activity to be visually monitored.

### 2.1 Effects

The Audio Effects Rack implements several modules which synthesize and modulate sound based on parameters specific to each intended audio effect. All signals coming into and out of audio effects are signed 18-bit buses which can be interpreted as audio PCM samples or parameter settings. Sound effect primitives are composed to make higher-order blocks which may be more familiar to electronic musicians.

**Signal Generator**

The signal generator is the main producer of audio signals besides the microphone input. Lookup tables for sine, sawtooth, triangle, and square waves are used to provide a choice in the timbre of the produced sound. The frequency of all of these sounds is selectable by indicating a count which is used to increment the index into the appropriate look-up table. An optional input is also available which will be mixed equally with the generated signal (see Mix).

**Finite Input Response (FIR) Filter**

In order to attenuate unwanted frequencies the filter module applies a convolution of a 31-tap FIR filter with a historical buffer of input samples. This effect is not only useful in live performance, but doubles as an audio support module. Instances of this module are used on the main input and output signals to/from the AC’97 codec chip (see Figure 1) to counteract aliasing and reconstruction artifacts. Malformed filters can be used to generate irregular sounding transformations for more experimental sounds.

**Delay**

The delay introduces intentional latency to the signal path by buffering audio samples in ZBT memory. The size of this buffer and speed of playback are ideally parameterized, but have been set to constants for this demonstration. Delay is an useful effect for higher-order composition of effects such as Chorus, Flanger, and Echo.
2.2 Routing

Routing in the Audio Effects Rack is achieved using two components with a shared memory to store a routing table. The patchbay scans the physical interface and updates the routing table. The router uses the information in the routing table to pass input signals to the appropriate destination.

Patchbay

The patchbay uses a set of scanning and probing channels which are normally pulled up to VDD. To scan for active connections, a scan pattern of one bit on is shifted through scan channels. If there is a connection from Output 0 to Input 1, the FPGA-driven low signal on the Output 0 scan channel will be read at the probe channel corresponding to Input 1. All combinations of sources and destinations are tested continuously using the high and low halves of a 1kHz 8-bit counter. When a scan hit is sensed, the write enable for the routing table is asserted. In the case of a scan miss, the source channel set the highest patchbay address to indicate a no-connect.

Router

Information describing the connectivity of modules is stored in the Routing Table. This data is used on the rising edge of the AC’97s ready signal to synchronously assign slices of data from the input vector to the appropriate slices of the routing output vector. Generally, the outputs of the effects blocks are sent to the input vector of the routing table, and the inputs to the effects blocks are connected to slices of the output vector of the router.

2.3 Display

A VGA display provides visual feedback of audio channel activity to aid in live performance. The values of input samples to effects blocks are displayed at 60Hz so that a signal that isn’t necessarily heard can be visually checked by a performer. In addition to the volumetric displays, the routing table is also outputted to the 16-segment displays on the 6.111 labkit.

2.4 Composition and Extensions

Each effect module is designed for easy composition with other effects. All input selectors have an 18-bit resolution, meaning that the output of any module can be used as a selector, and not just the signal input,
to another module. For example, a sine wave from the signal generator can vary the buffer size of the delay module.

The 18-bit standardization of all effects modules also allows for extensibility of the system as a whole. If the specifications set by the modules already implemented are followed, any new effect module can be implemented and added into the routing system. This design therefore represents more than simply a series of implemented digital effects, but rather a framework for the routing of a digital signal through any included modules.

3 Description

Figure 1: Overview of Demonstration Configuration. LPF is an instance of a Low-Pass Filter. The effects such as Delay (D), Signal Generator (SG), and other Filters (FIR) receive their inputs from input_vector and pipe their outputs into output_vector.

3.1 Patchbay (Drew)

The patchbay is operated using two of the user ports on the lab kit as scan and probe channels. For the sixteen channel example we configured for the lab demonstration, an 8-bit counter is used to address every
pair of input and output channels. Each cycle augments the test circuit, exposing the correct address, data, and write enable signals to the routing table. For each source address, a scan pattern with an active low signal on the given source channel is driven. Then each destination channel is probed to find if its value is zero (meaning a connection is present).

The physical ports of the patchbay must be connected to pull-up resistors in order to ensure a high logic value throughout no-connect scenarios. The external user buses on the labkit also require a slower clock speed of 1kHz. These periodic updates of connectivity directly effect the internal routing table of the router module.

### 3.2 Routing (Drew)

The routing provided by the Audio Effects Rack takes advantage of register/wire-arrays, input/output-vector slicing, and a lookup table for routing. The module receives all the outputs from the effects blocks as one concatenated vector of 18-bit audio samples (called the output vector). Similarly, the router exposes a concatenated vector of samples to go into the effects blocks (called the input vector). Generate blocks are used to assign 18-bit slices of the input and output vectors to wire and reg arrays which can be indexed by src/dest channel.

To synchronously assign all outputs to their appropriate input channels, a for-loop iterates through all routing table address and performs the appropriate deferred assignment using the more convenient array data structure. This allows arbitrary routing changes from the patchbay to be applied seamlessly in the next 48kHz frame.

### 3.3 Effects

#### 3.3.1 Mix (Marc)

Mix allows for two input signals to be combined and output as one. An 18-bit input selector is used to calculate the relative weight of each signal in the combination. The signals are multiplied by the appropriate weight, then summed into the output signal.

This module is especially useful when instantiated by other modules, providing for a “wet/dry” selector commonly found in audio effects devices. “Wet” and “dry” represent the modulated and unaffected input signals, respectively.
3.3.2 Signal Generator (Marc)

The signal generator is capable of creating 1024-sample, 18-bit sine, square, sawtooth, and triangle waves at varying frequencies. The module takes a frequency and shape selector as input, and produces the desired digital signal as its output.

To generate the shapes, look-up tables for sine, sawtooth, and triangle waves are instantiated. The tables return the value of a sample for an index between 0 and 1023. The incrementation speed of the index is controlled by a counter that increments on every clock cycle. When the counter reaches a certain value, decided by the frequency select input, the index increments, and the counter resets. This allows the user to control the frequency of the output signal. The sine look-up table was generated in Xilinx ISE using coregen, and the triangle and sawtooth tables were created using MATLAB. The square shape is generated by combinational logic in the signal generator module. It is low for the first 512 samples, and high for the rest.

A mix module is also instantiated by the signal generator in order to allow other signals to pass through it. Different sounds can then be chained together, to generate polyphonic tones.

3.3.3 FIR Filter (Marc)

The Finite Impulse Response filter module attenuates and intensifies certain frequencies of an input signal given the filter coefficients of the desired frequency response. The output is the convolution of the coefficients with the input signal. An accumulator performs the convolution by adding incrementing by the product of a coefficient and sample on every clock cycle. An offset is used to step through the samples backwards, in order to properly calculate the convolved output.

MATLAB was used to generate filter coefficients, and a python script was written to generate a case statement given a list of coefficients. The case statement is implemented as a ROM on the FPGA. Like the signal generator look-up tables, a coefficient is returned given an index.

3.3.4 Delay (Marc)

Delay can best be described as a window that moves along a set of memory addresses with time. When a new sample is ready from the AC’97, it is written to the address specified by front end of the window, while the address at the back end of the window is read from and sent to the AC’97 for playback. Using this method, a sample which has been recorded into memory will be played back after the full length of the window has passed over it. Therefore, the delay time is controlled by the length of the window, specified
by the number of samples in the window by an input to the module, as shown in Figure 2. For example, a window size of 48,000 samples results in a one second delay, since the sample rate is 48KHz.

Figure 2: Sliding Window of ZBT Memory used by Delay.

The labkit’s onboard ZBT SRAM is used as the memory for the delay. In order to account for clock skew between the FPGA and memory modules, a helper module, provided by the 6.111 staff in 2005, was used to invert the clock to the SRAM. It also provided an easy interface to the RAM by allowing write data to be input and providing read data as output, even though the memory is single-port. A write-enable signal to this module pulls low the write-enable signal to the RAM, indicating that it should write the current data in the bus to the specified address. Otherwise, the memory returns the data in the specified address along the data bus. Each of these operations has a two-cycle delay. Therefore, the data from a read request cannot be latched until two cycles later, and the written data is not available in memory until two cycles after write functionality was enabled.

To accomplish the reading, writing, and moving of the window along the memory, a finite state machine manages the module’s current task. The state machine operates as follows:

1. Do nothing until ready signal is asserted.
2. On ready signal, write current sample to highest address in window.
3. After write, read sample from lowest address in window. Wait two clock cycles, then latch data.
4. Latched data is next sample out from module. Wait again for ready signal.

A mix module is also instantiated to allow for both the original input sample and the delayed output sample to be output, allowing for a “wet/dry” signal selector to be used.
3.3.5 Pan (Drew)

The pan module duplicates a mono signal and weights left and right output channels to reflect the currently selected pan value. The transfer curve for pan is equal-power which means that the gain applied to the stereo channel does not grow linearly. To achieve this equal-power curve, a 16-bin lookup table is used. The appropriate gain coefficient is selected for left and right channels, and the new weighted values are output to the left and right output channels.

3.4 Display (Drew)

The display is a simple feedback mechanism for the potential musician who would use the Audio Effects Rack. The DCM is used to generate a buffer 65MHz clock which is used as a pixel clock with an xvga module (this builds the appropriate control signals for the VGA display). Each channel’s input is latched at 60Hz and displayed as a vertical value bar of a color matching a patchbay channel. Appropriate spacing is coded for the given modules setup during the Demo Configuration. Due to undersampling and the resolution of the current video setup, this display is mostly useful for check alive-or-dead activity on a given channel (e.g., microphone in).

3.5 Demo Configuration

The labkit setup for the demo was designed for a live performance. We included 4 signal generators (with selected shape mapped to the labkit switched) of frequencies in an A Major chord. Each one of the signal generators had a equal-gain mix passing upstream signals through to the next effect. Two FIR filters were also mapped to the patchbay to show the effects of not including the anti-aliasing and reconstruction filters. Finally a delay module was provided to utilize during feedback improvisation. The main in/out on the patchbay represented the live microphone samples and the outgoing speaker samples. Using all of these effects together made for an entertaining demonstration of patchable sounds.

4 Testing and Troubleshooting

All modules were initially tested with GTKwave and/or ModelSim, using a Verilog test module that provided the appropriate clock, ready, and other input signals. However, simulation proved insufficient in several cases, most of which involved components that ModelSim could not simulate. Each module presented new challenges and issues that needed to be overcome for successful implementation. After simulation,
modules were tested individually on the FPGA, and when they proved functional, were integrated with the rest of the system.

4.1 Use of the Signal Generator for Testing

The signal generator was intentionally one of the first modules to be implemented. All of the available wave shapes are useful for testing audio-related modules, both to hear using the AC’97, and to see with the logic analyzer or modelsim. It therefore became the primary testing tool for most other modules.

4.2 Routing

To develop a scheme for updating the routing table, many aproaches were explored in simulation. It became clear that each pair of source and destination channels needed to be tested, so a simulation was first developed which would iterate through all possible pairs. This was achieved by the use of a counter that was twice as wide as an value in the routing table. This meant that the higher four bits could be regarded as a steady address while the lower four bits changed every clock cycle. Simulation also helped confirm that syntax for slicing the input and output vectors was working correctly. There were generally two test cases for the Patchbay simulations: (1) every channel \( n \) was fed into channel \( Max - n \), or (2) only one connection existed between the out of channel 4 and the in of channel 2. These scenarios were all that needed to be confirmed in order to expect functional behavior. Unfortunately, the unforeseen circumstance of run an external bus faster than it can be read is not revealed through simulation. Only once the program was running on the FPGA did we notice off-by-one and off-by-two errors which symptoms of such a timing problem

4.3 Delay and ZBT RAM

The delay effect experience several phases of memory management before settling on the use of the onboard ZBT SRAM. Initially, it was written to operate using coregen-created dual port BRAM. However, the synthesis of the amount of memory desired \( 2^{18} \) bits was taking almost an hour to complete. The same was apparently true when creating an inferred amount of BRAM using arrays of registers. Therefore, it was decided that the delay’s data would be located in ZBT RAM.

The ZBT RAM presented several new obstacles to be surmounted in order to have a working delay. Many of these were taken care of by the ZBT helper module provided by the 6.111 2005 website. It inverts the clock to the ZBT RAM in order to insure that data is held for a sufficient period of time. The main
issue with interfacing with the memory was latching data at the correct time. Read requests to the RAM do not yield data until two clock cycles later. It was therefore necessary for the delay’s state machine to use a counter to remain in its read state until it read data from the correct address. The other major problem involved some of the signals that needed to be held low in order for the RAM to function properly. After realizing that I had not assigned some of them to a value, it became apparent that this was the problem, and the delay began working soon after I corrected it.

5 Conclusions

The Field Programmable Digital Audio Effects Rack is a modular, extensible system designed to integrate many audio effects easily into one device. The provided, on-the-fly signal routing system provides a familiar method for choosing and arranging effects, and the high sample rate and resolution are aimed at preserving sound quality. The FPGA provides a unique angle from which to approach an audio effects rack such as this one. It can, for instance, be thought of as a series of pre-programmed digital effects, with a provided routing system. We prefer to think of it as a framework that provides a musician or developer with an environment for managing digital effects on an FPGA, routing signals, and synthesizing new sounds.

6 Appendices
A Code Listing (alpha-order)

A.1 Main Audio Support (audio.v)

```verilog
// bi-directional monaural interface to AC97
// augmented by Drew Shapiro
// TODO: Make input channel binaural and (18-bit) [11/28/2010]

module audio_support (/*AUTOARGS*/
    // Outputs
    ready, mono_channel_in, ac97_sdata_out, ac97_synch, audio_reset_b,
    // Inputs
    reset, clock_27mhz, volume, left_channel_out, right_channel_out,
    ac97_sdata_in, ac97_bit_clock
);

input reset, clock_27mhz;
input [4:0] volume;
input [17:0] left_channel_out, right_channel_out;
input ac97_sdata_in, ac97_bit_clock; //ac97 interface signal

output ready;
output [17:0] mono_channel_in;
output ac97_sdata_out, ac97_synch; //ac97 interface signal
output reg audio_reset_b; //ac97 interface signal

wire [7:0] command_address;
wire [15:0] command_data;
wire command_valid;
wire [19:0] left_in_data, right_in_data;
wire [19:0] left_out_data, right_out_data;

// wait a little before enabling the AC97 codec
reg [9:0] reset_count;
always @((posedge clock_27mhz) begin
    if (reset) begin
```
audio_reset_b = 1'b0;
reset_count = 0;
end else if (reset_count == 1023)
audio_reset_b = 1'b1;
else
reset_count = reset_count+1;
end

wire ac97_ready;
ac97 ac97(.ready(ac97_ready),
    .command_address(command_address),
    .command_data(command_data),
    .command_valid(command_valid),
    .left_data(left_out_data), .left_valid(1'b1),
    .right_data(right_out_data), .right_valid(1'b1),
    .left_in_data(left_in_data), .right_in_data(right_in_data),
    .ac97_sdata_out(ac97_sdata_out),
    .ac97_sdata_in(ac97_sdata_in),
    .ac97_synch(ac97_synch),
    .ac97_bit_clock(ac97_bit_clock));

// ready: one cycle pulse synchronous with clock_27mhz
reg [2:0] ready_sync;
always @(posedge clock_27mhz) ready_sync <= {ready_sync[1:0], ac97_ready};
assign ready = ready_sync[1] & ~ready_sync[2];

reg [17:0] l_left_data, l_right_data; //latch incoming audio data
always @(posedge clock_27mhz)
    if (ready) begin
        l_left_data <= left_channel_out;
        l_right_data <= right_channel_out;
    end
assign mono_channel_in = left_in_data[19:2]; //FIX - monaural
assign left_out_data = {l_left_data, 2'b00};
assign right_out_data = {l_right_data, 2'b00};

// generate repeating sequence of read/writes to AC97 registers
ac97commands cmds(.clock(clock_27mhz), .ready(ready),
module ac97 (
    output reg ready,
    input wire [7:0] command_address,
    input wire [15:0] command_data,
    input wire command_valid,
    input wire [19:0] left_data,
    input wire left_valid,
    input wire [19:0] right_data,
    input wire right_valid,
    output reg [19:0] left_in_data, right_in_data,
    output reg ac97_sdata_out,
    input wire ac97_sdata_in,
    output reg ac97_synch,
    input wire ac97_bit_clock
);

reg [7:0] bit_count;
reg [19:0] l_cmd_addr;
reg [19:0] l_cmd_data;
reg [19:0] l_left_data, l_right_data;
reg l_cmd_v, l_left_v, l_right_v;

initial begin
    ready <= 1'b0;
    // synthesis attribute init of ready is "0";
    ac97_sdata_out <= 1'b0;
    // synthesis attribute init of ac97_sdata_out is "0";
    ac97_synch <= 1'b0;
    // synthesis attribute init of ac97_synch is "0";
    bit_count <= 8'h00;
endmodule // audio

// assemble/disassemble AC97 serial frames

reg [7:0] bit_count;
// synthesis attribute init of bit_count is "0000";
l_cmd_v <= 1'b0;
// synthesis attribute init of l_cmd_v is "0";
l_left_v <= 1'b0;
// synthesis attribute init of l_left_v is "0";
l_right_v <= 1'b0;
// synthesis attribute init of l_right_v is "0";

left_in_data <= 20'h00000;
// synthesis attribute init of left_in_data is "000000";
right_in_data <= 20'h00000;
// synthesis attribute init of right_in_data is "000000";
end

always @(posedge ac97_bit_clock) begin

// Generate the sync signal
if (bit_count == 255)
    ac97_synch <= 1'b1;
if (bit_count == 15)
    ac97_synch <= 1'b0;

// Generate the ready signal
if (bit_count == 128)
    ready <= 1'b1;
if (bit_count == 2)
    ready <= 1'b0;

// Latch user data at the end of each frame. This ensures that the first frame after reset will be empty.
if (bit_count == 255) begin
    l_cmd_addr <= {command_address, 12'h000};
    l_cmd_data <= {command_data, 4'h0};
    l_cmd_v <= command_valid;
    l_left_data <= left_data;
    l_left_v <= left_valid;
    l_right_data <= right_data;
    l_right_v <= right_valid;
end
if ((bit_count >= 0) && (bit_count <= 15))
    // Slot 0: Tags
    case (bit_count[3:0])
        4'h0: ac97_sdata_out <= 1'b1;  // Frame valid
        4'h1: ac97_sdata_out <= 1_cmd_v; // Command address valid
        4'h2: ac97_sdata_out <= 1_cmd_v; // Command data valid
        4'h3: ac97_sdata_out <= 1_left_v; // Left data valid
        4'h4: ac97_sdata_out <= 1_right_v; // Right data valid
        default: ac97_sdata_out <= 1'b0;
    endcase
else if ((bit_count >= 16) && (bit_count <= 35))
    // Slot 1: Command address (8-bits, left justified)
    ac97_sdata_out <= 1_cmd_v ? l_cmd_addr[35−bit_count] : 1'b0;
else if ((bit_count >= 36) && (bit_count <= 55))
    // Slot 2: Command data (16-bits, left justified)
    ac97_sdata_out <= 1_cmd_v ? 1_cmd_data[55−bit_count] : 1'b0;
else if ((bit_count >= 56) && (bit_count <= 75)) begin
    // Slot 3: Left channel
    l_left_in_data = { l_left_in_data[18:0], ac97_sdata_in };
    l_left_data <= { l_left_data[18:0], l_left_data[19] };
end
else if ((bit_count >= 76) && (bit_count <= 95))
    // Slot 4: Right channel
    ac97_sdata_out <= 1_right_v ? 1_right_data[95−bit_count] : 1'b0;
else
    ac97_sdata_out <= 1'b0;

bit_count <= bit_count+1;
end // always @(posedge ac97_bit_clock)

always @(negedge ac97_bit_clock) begin
    if ((bit_count >= 57) && (bit_count <= 76))
        // Slot 3: Left channel
        l_left_in_data <= { l_left_in_data[18:0], ac97_sdata_in };
    else if ((bit_count >= 77) && (bit_count <= 96))
        // Slot 4: Right channel
        l_right_in_data <= { l_right_in_data[18:0], ac97_sdata_in };
end
endmodule
module ac97commands (
    input wire clock,
    input wire ready,
    output wire [7:0] command_address,
    output wire [15:0] command_data,
    output reg command_valid,
    input wire [4:0] volume,
    input wire [2:0] source
);

reg [23:0] command;

reg [3:0] state;

initial begin
    command <= 4'h0;
    // synthesis attribute init of command is "0";
    command_valid <= 1'b0;
    // synthesis attribute init of command_valid is "0";
    state <= 16'h0000;
    // synthesis attribute init of state is "0000";
end

assign command_address = command[23:16];
assign command_data = command[15:0];

wire [4:0] vol;
assign vol = 31-volume; // convert to attenuation

always @(posedge clock) begin
    if (ready) state <= state+1;
    case (state)
        4'h0: // Read ID
            begin
                command <= 24'h80_0000;
                command_valid <= 1'b1;
            end
        4'h1: // Read ID
command <= 24'h80_0000;
4'h3: // headphone volume
    command <= { 8'h04, 3'b000, vol, 3'b000, vol };
4'h5: // PCM volume
    command <= 24'h18_0808;
4'h6: // Record source select
    command <= { 8'h1A, 5'b00000, source, 5'b00000, source };
4'h7: // Record gain = max
    command <= 24'h1C_0F0F;
4'h9: // set +20db mic gain
    command <= 24'h0E_8048;
4'hA: // Set beep volume
    command <= 24'h0A_0000;
4'hB: // PCM out bypass mix1
    command <= 24'h20_8000;
default:
    command <= 24'h80_0000;
endcase // case(state)
end // always @(posedge clock)
endmodule // ac97commands
A.2 Router (audio_router.v)

```verilog
module audio_router(//AUTOARG*/
  // Outputs
  block_output_vector, current_routing,
  // Inputs
  reset, clk, ready, block_output_vector, route clk, route data, route address,
  update_routing
);

parameter WIDTH = 18;
parameter LOG_N = 4;
localparam N = 1 << LOG_N;

input reset, clk, ready;
// accept a concatenated bus of output samples
input [WIDTH*N-1:0] block_output_vector;
wire [WIDTH-1:0] rcv_from_block[N-1:0];
// provide a concatenated bus of input samples
output [WIDTH*1:0] block_input_vector;
reg [WIDTH-1:0] snd_to_block[N-1:0];

// maintain a routing table which can be updated externally
input route_clk, update_routing;
input [LOG_N-1:0] route_data, route_address;
output [LOG_N*1:0] current_routing;
reg [LOG_N-1:0] routing_table [N-1:0];

// slice input and output vectors
generate
  genvar i;
  for(i=0;i<N;i =i+1) begin: slice
```

21
assign rcv_from_block[i] = block_output_vector[(i+1)*WIDTH−1:i*WIDTH];
assign block_input_vector[(i+1)*WIDTH−1:i*WIDTH] = snd_to_block[i];
assign current_routing[(i+1)*LOG_N−1:i*LOG_N] = routing_table[i];
end

class block (input [WIDTH−1:0] block_output,
           output [WIDTH−1:0] block_input)
begin
  // maintain a routing table which can be updated externally
  always @(posedge route_clk)
    if(update_routing) routing_table[route_address] <= route_data;
  // enforce routing table
  integer j;
  always @(posedge clk) begin
    for(j=0;j<N;j =j+1) begin
      snd_to_block[j] <= rcv_from_block[routing_table[j]]; // handle no-connects?
    end
  end
endmodule
A.3 Main Labkit File (audiofxbox.v)

```vhdl
'timescale 1ns / 1ps

// Audio Effects Box

module audiofxbox(
  // AC97
  output wire beep, audio_reset_b, ac97_synch, ac97_sdata_out,
  input wire ac97_bit_clock, ac97_sdata_in,

  // VGA
  output wire [7:0] vga_out_red, vga_out_green, vga_out_blue,
  output wire vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
  vga_out_vsync,

  // PS2
  inout wire mouse_clock, mouse_data, keyboard_clock, keyboard_data,

  // FLUORESCENT DISPLAY
  output wire disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b,
  input wire disp_data_in,
  output wire disp_data_out,

  // BUTTONS, SWITCHES, LEDS
  //input wire button0,
  //input wire button1,
  //input wire button2,
  //input wire button3,
  //input wire button_enter,
  //input wire button_right,
  //input wire button_left,
  //input wire button_down,
  //input wire button_up,
  input wire [7:0] switch,
  output wire [7:0] led
);```
// USER CONNECTORS, DAUGHTER CARD, LOGIC ANALYZER

inout wire [11:0] user1,
inout wire [31:0] user2,
inout wire [31:0] user3,
inout wire [31:0] user4,

//inout wire [43:0] daughtercard,
output wire [15:0] analyzer1_data, output wire analyzer1_clock,
//output wire [15:0] analyzer2_data, output wire analyzer2_clock,
output wire [15:0] analyzer3_data, output wire analyzer3_clock,
output wire [15:0] analyzer4_data, output wire analyzer4_clock,
inout wire clock_27mhz,

// ZBT RAM
inout [35:0] ram0_data,
output [18:0] ram0_address,
output ram0_adv_ld,
output ram0_clk, ram0_cen_b,
output ram0_ce_b, ram0_oe_b,
output ram0_we_b,
output [3:0] ram0_bwe_b

);

// ZBT Signals
//

assign ram0_ce_b = 0;
assign ram0_oe_b = 0;
assign ram0_adv_ld = 0;
assign ram0_bwe_b = 3'b0;

// Timing and Reset
//
wire clock_65mhz_unbuf, clock_65mhz;

// use FPGA's digital clock manager to produce a
// 65MHz clock (actually 64.8MHz)
DCM vclk1 (.CLKIN(clock_27mhz), .CLKFX(clock_65mhz_unbuf));

// synthesis attribute CLKFX_DIVIDE of vclk1 is 10
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2 (.O(clock_65mhz), .I(clock_65mhz_unbuf));

// power-on reset generation
wire power_on_reset; // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset), .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

// divide down a 10Hz clock for visual display
reg clock_10hz, clock_1khz;
reg [20:0] count_20hz;
reg [13:0] count_2khz;
assign led = {8{clock_10hz}};

always @(posedge clock_27mhz) begin
  if(power_on_reset) begin
    clock_10hz <= 1;
clock_1khz <= 1;
count_20hz <= 0;
count_2khz <= 0;
  end
  else begin
    clock_10hz <= ((count_20hz+1) >= 1350000) ? ˜clock_10hz : clock_10hz;
count_20hz <= ((count_20hz+1) >= 1350000) ? 0 : count_20hz+1;
clock_1khz <= ((count_2khz+1) >= 13500) ? ˜clock_1khz : clock_1khz;
count_2khz <= ((count_2khz+1) >= 13500) ? 0 : count_2khz+1;
  end
end
// 1 MHz Clock for ADC
one_mhz_clock one_mhz (.clock_27mhz(clock_27mhz), .clock_1mhz(clock_1mhz));

assign user2[18] = clock_1mhz;
wire [7:0] knob_value;
assign knob_value = user2[26:19];

/////////////////////////////////////////////////////////////////////////////////////////////////

// Audio Support

/////////////////////////////////////////////////////////////////////////////////////////////////

wire ready;

// assign beep = power_on_reset;
wire [17:0] mono_channel_in,
     left_channel_out, right_channel_out,
     left_channel_raw, right_channel_raw, raw_in;
wire [27:0] antiAlias_out, antiimage_out1, antiimage_out2;

// assign left_channel_out = left_channel_raw;
// assign right_channel_out = right_channel_raw;

// filter 64 antiAlias (.clock(clock_27mhz), .reset(reset), .ready(ready), .x(left_channel_raw)
    , .y(left_channel_out));

audio_support basic_audio_interface ( .reset(reset),
         .clock_27mhz(clock_27mhz),
         .ready(ready),
         .mono_channel_in(mono_channel_in),
         .left_channel_out(left_channel_out),
         .right_channel_out(right_channel_out),
         .ac97_bit_clock(ac97_bit_clock),
         .ac97_sdata_in(ac97_sdata_in),
         .ac97_sdata_out(ac97_sdata_out),
         .ac97_synch(ac97_synch),
         .audio_reset_b(audio_reset_b),
         .volume(5'h11111)
);

/////////////////////////////////////////////////////////////////////////////////////////////////
wire [16*18−1:0] input_vector, output_vector;

// Channel 0: Main I/O
wire [17:0] main_out_mono;
assign left_channel_out = main_out_mono;
assign right_channel_out = main_out_mono;
assign main_out_mono = input_vector[17:0];

// Channel 1: Delay
wire [17:0] delay_signal;
delay delay1 (  
  .clock(clock,27mhz),  
  .reset(reset),  
  .ready(ready),  
  .sample_in(input_vector[18*2−1:18]),  
  .buffer_size(19'd24000),  
  .mix_out(delay_signal),  
  .ram_clk(ram0_clk),  
  .ram_we_b(ram0_we_b),  
  .ram_address(ram0_address),  
  .ram_data(ram0_data),  
  .ram_cen_b(ram0_cen_b),  
  .mix_select(0)  
);  

// Channel 2: FIR 1
wire [17:0] fir1_signal;
wire [27:0] fir1_signal_full;
assign fir1_signal = fir1_signal_full[27:10];
filter fir1 (  
  .clock(clock,27mhz),
.reset(reset),
.ready(ready),
.x(input_vector[18*3−1:18*2]),
.y(fir1_signal_full)
);

//Channel 3: FIR 2 assign
wire [17:0] fir2_signal;

//Channel 4: Wave 1 A440 – 61 363 counts
reg [17:0] wave1_count;
wire [17:0] wave1_signal;
signal_generator wave1 (  
.freq_select(18’d61,363),
.clock(clock,27mhz),
.reset(reset), .wave_in(input_vector[18*5−1:18*4]),
.wave_out(wave1_signal),
.shape(switch[1:0])
);

//Channel 5: Wave 2 C#550 – 49 090 counts
reg [17:0] wave2_count;
wire [17:0] wave2_signal;
signal_generator wave2 (  
.freq_select(18’d49,090),
.clock(clock,27mhz),
.reset(reset), .wave_in(input_vector[18*6−1:18*5]),
.wave_out(wave2_signal),
.shape(switch[3:2])
);

//Channel 6: Wave 3 E660 – 40 909 counts
reg [17:0] wave3_count;
wire [17:0] wave3_signal;
signal_generator wave3 (  
.freq_select(18’d40,909),
.clock(clock,27mhz),
.reset(reset), .wave_in(input_vector[18*7−1:18*6]),
.wave_out(wave3_signal),
)
```vhdl
// Channel 7: Wave 4 A880 – 30 681 counts
reg [17:0] wave4_count;
wire [17:0] wave4_signal;
signal_generator wave4(
    .freq_select(18'd30_681),
    .clock(clock_27mhz),
    .reset(reset),
    .wave_in(input_vector[18*8-1:18*7]),
    .wave_out(wave4_signal),
    .shape(switch[7:6])
);

reg [17:0] select;
wire signed [17:0] mix_signal;
always @(posedge clock_10hz)
    select <= reset ? 0 : select + 15'b100000000000000;

mix mix_fifth(
    .reset(reset),
    .clock(clock_27mhz),
    .ready(ready),
    .mix_select(0),
    .signal1(wave1_signal),
    .signal2(wave2_signal),
    .mixed(mix_signal)
);

wire signed [17:0] select_knob = {user1[11:0], 6'b0};

wire signed [17:0] right_panned, left_panned;
pan pan_main_out(
    .clk(clock_27mhz), .reset(reset),
    .ready(ready),
```
assign right_channel_raw = switch[7] ? right_panned : main_out_mono;

wire update_routing;
wire [3:0] src, dest;
wire [63:0] data;
wire [14*18-1:0] nothing;

assign output_vector = {0, wave4_signal, wave3_signal, wave2_signal, wave1_signal, // Waves
fir2_signal, fir1_signal, // Filters
delay_signal, // Delay
mono_channel_in}; // Line-In

audio_router #(LOG N(4), WIDTH(18)) router1 (
.route_address(dest),
.update_routing(update_routing)
);

// physical patchbay updates routing table inside the router
patchbay physical_interface(
  .scan_channels(user3[15:0]),
  .probe_channels(user4[15:0]),
  .update_routing(update_routing),
  .src(src),
  .dest(dest),
  .reset(reset),
  .clock(clock_1khz)
);

assign user4[31:8] = {24'1'b1};

/****************************************************************************

// Keyboard and Mouse Input
/****************************************************************************
wire [11:0] mouse_x, mouse_y;
wire [2:0]  mouse_click;
ps2_mouse_xy  mouse(.clk(clock_65mhz), .reset(reset),
  .ps2_clk(mouse_clock), .ps2_data(mouse_data),
  .mx(mouse_x), .my(mouse_y), .btn_click(mouse_click));
wire [2:0]  command;
wire new_command;
assign beep = new_command;
keyboard asdw_commands(
  .clock_27mhz(clock_27mhz),
  .reset(reset),
  .kbd_data(keyboard_data),
  .kbd_clock(keyboard_clock),
  .command(command),
  .new_command(new_command)
);
```vhdl
-- //
-- // VGA Display
-- //
-- //**************************************************************

-- // generate basic XVGA video signals
wire [10:0] hcount;
wire [9:0] vcount;
wire hsync, vsync, blank;
xvg a xvg a1 (.vclock (clock_65mhz), .hcount (hcount), .vcount (vcount),
    .hsync (hsync), .vsync (vsync), .blank (blank));
wire [2:0] pixelmainout, pixelmainin, pixel1, pixel2, pixel3, pixel4, pixel5, pixel6,
    pixel7, pixel8;
channel_display #( .LEFT (100), .COLOR (7) )
    main_out_display (.
        .vga_clock (clock_65mhz), .reset (reset), .pixel (pixelmainout),
        .hcount (hcount), .vcount (vcount), .vsync (vsync), .value (main_out_mono));
channel_display #( .LEFT (175), .COLOR (3) )
    main_in_display (.
        .vga_clock (clock_65mhz), .reset (reset), .pixel (pixelmainin),
        .hcount (hcount), .vcount (vcount), .vsync (vsync), .value (mono_channel_in));
channel_display #( .LEFT (350), .COLOR (2) )
    wave1_display (.
        .vga_clock (clock_65mhz), .reset (reset), .pixel (pixel1),
        .hcount (hcount), .vcount (vcount), .vsync (vsync), .value (input_vector[18*2 − 1:18]));
channel_display #( .LEFT (425), .COLOR (1) )
    wave2_display (.
        .vga_clock (clock_65mhz), .reset (reset), .pixel (pixel2),
        .hcount (hcount), .vcount (vcount), .vsync (vsync), .value (input_vector[18*3 − 1:18*2]));
channel_display #( .LEFT (500), .COLOR (4) )
    wave3_display (.
        .vga_clock (clock_65mhz), .reset (reset), .pixel (pixel3),
```
.hcount (hcount), .vcount (vcount), .vsync (vsync), .value (input_vector[18*4 – 1:18*3]);

channel_display #(.LEFT(675), .COLOR(6))
wave4_display (.vga_clock (clock_65mhz), .reset (reset), .pixel (pixel4),
.hcount (hcount), .vcount (vcount), .vsync (vsync), .value (input_vector[18*5 – 1:18*4]));

channel_display #(.LEFT(750), .COLOR(1))
wave5_display (.vga_clock (clock_65mhz), .reset (reset), .pixel (pixel5),
.hcount (hcount), .vcount (vcount), .vsync (vsync), .value (input_vector[18*6 – 1:18*5]));

channel_display #(.LEFT(825), .COLOR(4))
wave6_display (.vga_clock (clock_65mhz), .reset (reset), .pixel (pixel6),
.hcount (hcount), .vcount (vcount), .vsync (vsync), .value (input_vector[18*7 – 1:18*6]));

channel_display #(.LEFT(900), .COLOR(2))
wave7_display (.vga_clock (clock_65mhz), .reset (reset), .pixel (pixel7),
.hcount (hcount), .vcount (vcount), .vsync (vsync), .value (input_vector[18*8 – 1:18*7]));

reg [2:0] rgb;
reg b, hs, vs;
always @(posedge clock_65mhz) begin
hs <= hsync;
vs <= vsync;
b <= blank;
rgb <= ((hcount == mouse_x | vcount == mouse_y) ? 7 : 0) |
(pixelmainout | pixelmainin | pixel1 | pixel2 | pixel3 | pixel4 | pixel5 | pixel6 |
pixel7 | pixel8) |
end
// VGA Output. In order to meet the setup and hold times of the
// AD7125, we send it ˜clock_65mhz.
assign vga_out_red = {8{rgb[2]}};
assign vga_out_green = {8{rgb[1]}};
assign vga_out_blue = {8{rgb[0]}};
assign vga_out_sync_b = 1'b1; // not used
assign vga_out_blank_b = ˜b;
assign vga_out_pixel_clock = ˜clock_65mhz;
assign vga_out_hsync = hs;
assign vga_out_vsync = vs;

///////////////////////////////////////////

// Hex Display

///////////////////////////////////////////

display_16hex disp16 (.reset(reset), .clock,27mhz(clock_27mhz),
.data(data), .disp_blank(disp_blank), .disp_clock(disp_clock),
.disp_r_s(disp_r_s), .disp_c_e_b(disp_c_e_b),
.disp_reset_b(disp_reset_b), .disp_data_out(disp_data_out));
assign analyzer1_clock = clock_65mhz;
assign analyzer1_data[3:0] = src;
assign analyzer1_data[7:4] = dest;
assign analyzer1_data[8] = update_routing;
assign analyzer1_data[15:9] = {7{1'b1}};
assign analyzer3_clock = clock_65mhz;
assign analyzer3_data[15:0] = user3[15:0];
assign analyzer4_clock = clock_65mhz;
assign analyzer4_data[15:0] = user4[15:0];
endmodule
A.4 Audio-Visual Feedback (channel_display.v)

module channel_display(vga_clock, vsync, reset, pixel, hcount, vcount, value);

parameter COLOR = 7;
parameter LEFT = 100;
parameter BOTTOM = 100;
parameter WIDTH = 20;
parameter HEIGHT = 512;
localparam TOP = BOTTOM + HEIGHT;

input vga_clock;
input reset;
input [17:0] value;
wire [8:0] disp_value;
reg [17:0] l_value; // grab pieces of the hi-res reading for visual feedback
assign disp_value[8:6] = {l_value[17], l_value[15], l_value[13]};
assign disp_value[5:3] = {l_value[11], l_value[9], l_value[7]};
assign disp_value[2:0] = {l_value[4], l_value[2], l_value[0]};
output reg [2:0] pixel;
input [10:0] hcount;
input [9:0] vcount;
input vsync;
reg [1:0] vsync_buffer;
wire frame_refresh; // only latch once every frame
assign frame_refresh = vsync_buffer[0] & ~vsync_buffer[1];
wire within_bounds, on_border;
assign within_bounds = (hcount >= LEFT & hcount < LEFT + WIDTH) &
                     (vcount >= BOTTOM & vcount < BOTTOM + HEIGHT);
assign on_border = within_bounds & ((hcount == LEFT) | (hcount == LEFT + WIDTH-1)) |
                  ((vcount == BOTTOM) | (vcount == BOTTOM + HEIGHT-1));
always @(posedge vga_clock) begin
    vsync_buffer <= {vsync_buffer[0], vsync};
    l_value <= frame_refresh ? value : l_value;
    if (within_bounds) begin
        pixel <= (vcount >= (BOTTOM + 512 - disp_value)) ? COLOR : 0; // only paint COLOR if value visible
    end
    else
        pixel <= 0;
end
endmodule
A.5 Delay (delay.v)

```
module delay (input clock, reset, ready, input signed [17:0] sample_in,
               input [18:0] buffer_size, input signed [17:0] mix_select,
               output signed [17:0] mix_out, output ram_clk,
               output ram_we_b, output [18:0] ram_address,
               inout [35:0] ram_data, output ram_cen_b);

parameter SAMPLE_RATE = 480000;

wire [35:0] write_data;
wire [35:0] read_data;
reg [18:0] cur_addr;
wire [18:0] write_addr;
reg [17:0] sample_out;
wire we;
reg [1:0] rwstate = 2'b0;
reg [1:0] next_rwstate = 2'b0;
wire [18:0] addr;

assign addr = we ? write_addr : cur_addr; // if write enabled, address is write address (front of window), otherwise, it’s read address (back of window)
assign write_data = sample_in; // write data always incoming sample

zbt_6111 delay_zbt ( // helper module for zbt ram.clk(clock),
```

38
.cen(1'b1),
 .we(we),
 .addr(addr),
 .write_data(write_data),
 .read_data(read_data),
 .ram_clk(ram_clk),
 .ram_we_b(ram_we_b),
 .ram_address(ram_address),
 .ram_data(ram_data),
 .ram_cen_b(ram_cen_b)
);

mix delay_mix(       // instantiate mix for wet/dry signal control
  .reset(reset),
  .clock(clock),
  .ready(ready),
  .mix_select(mix_select),
  .signal1(sample_in),
  .signal2(sample_out),
  .mixed(mix_out)
);

assign write_addr = cur_addr + (buffer_size - 1'b1);

parameter WAIT = 2'b00;
parameter WRITE = 2'b01;
parameter READ = 2'b10;
parameter COUNT = 2'b11;

reg [18:0]  next_addr;
reg [17:0]  next_sample_out;
reg [1:0]   rcount, next.rcount;

always @( * ) begin         // state machine for delay
case (rwstate)
  WAIT : begin
    next_addr = cur_addr;          // wait for ready, then go to WRITE
    if (ready) begin
      next_rwstate = WRITE;
  end
endcase

end

else begin
    next_rwstate = WAIT;
end

WRITE: begin
    next_addr = cur_addr;  // write current sample to high end of buffer, go to READ
    if (!ready) begin
        next_rwstate = READ;
        next_rcount = 2'd3;
    end
    else begin
        next_rwstate = WRITE;
    end
end

READ: begin  // wait two cycles for read
    // wait two cycles for read
    information to be retrieved, then latch, output, and go back to WAIT
    if (rcount > 0) begin
        next_addr = cur_addr;
        next_rcount = rcount - 1;
        next_rwstate = READ;
    end
    else begin
        next_addr = cur_addr+1;
        next_sample_out = read_data;
        next_rwstate = WAIT;
    end
end

default: begin
    next_rwstate = WAIT;
    next_addr = cur_addr;
end
endcase

assign we = (rwstate==WRITE);

always @(posedge clock) begin  // transition registers on posedge clock
    if (reset) begin

end
rwstate <= 0;
cur_addr <= 0;

end

else begin
rwstate <= next_rwstate;
cur_addr <= next_addr;
sample_out <= next_sample_out;
rcount <= next_rcount;

end

endmodule
module zbt_6111(clk, cen, we, addr, write_data, read_data, 
        ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);

    input clk; // system clock
    input cen; // clock enable for gating ZBT cycles
    input we; // write enable (active HIGH)
    input [18:0] addr; // memory address
    input [35:0] write_data; // data to write
    output [35:0] read_data; // data read from memory
    output ram_clk; // physical line to ram clock
    output ram_we_b; // physical line to ram we_b
    output [18:0] ram_address; // physical line to ram address
    inout [35:0] ram_data; // physical line to ram data
    output ram_cen_b; // physical line to ram clock enable

    // clock enable (should be synchronous and one cycle high at a time)
    wire ram_cen_b = ~cen;

    // create delayed ram_we signal: note the delay is by two cycles!
    // ie we present the data to be written two cycles after we is raised
    // this means the bus is tri-stated two cycles after we is raised.

    reg [1:0] we_delay;

    always @(posedge clk)
    we_delay <= cen ? {we_delay[0],we} : we_delay;
// create two-stage pipeline for write data

reg [35:0] write_data_old1;
reg [35:0] write_data_old2;
always @(posedge clk)
    if (cen)
        {write_data_old2, write_data_old1} <= {write_data_old1, write_data};

// wire to ZBT RAM signals

assign ram_we_b = ~we;
assign ram_clk = ~clk;  // RAM is not happy with our data hold
                        // times if its clk edges equal FPGA's
                        // so we clock it on the falling edges
                        // and thus let data stabilize longer
assign ram_address = addr;
assign ram_data = we_delay[1] ? write_data_old2 : {36'bZ};
assign read_data = ram_data;

endmodule // zbt_6111
module filter(
    input wire clock, reset, ready,
    input wire signed [17:0] x,
    output reg signed [27:0] y);

reg [4:0] index, offset;
wire signed [9:0] coeff;
reg signed [17:0] sample[30:0];
coeffs31 coeffs (.index(index), .coeff(coeff));

always @(posedge clock) begin
    if (reset) begin
        y <= 0;
        offset <= 0;
        index <= 0;
    end
    else if (ready) begin
        //On ready, increment offset, read new sample, reset accumulator
        offset <= offset + 1;
        sample[offset] <= x;
        y <= 0;
        index <= 0;
    end
    else if (!ready && (index < 5'b11111)) begin
        y <= y + (coeff * sample[offset−index]);
        //perform convolution using accumulator
        index <= index + 1;
    end
    else begin
        y <= y;
    end
end
```verilog
module coeffs31(
    input wire [4:0] index,
    output reg signed [9:0] coeff
);

// tools will turn this into a 31x10 ROM
always @(index)
    case (index)
      5'd0: coeff = -10'd1;
      5'd1: coeff = -10'd1;
      5'd2: coeff = -10'd3;
      5'd3: coeff = -10'd5;
      5'd4: coeff = -10'd6;
      5'd5: coeff = -10'd7;
      5'd6: coeff = -10'd5;
      5'd7: coeff = 10'd0;
      5'd8: coeff = 10'd10;
      5'd9: coeff = 10'd26;
      5'd10: coeff = 10'd46;
      5'd11: coeff = 10'd69;
      5'd12: coeff = 10'd91;
      5'd13: coeff = 10'd110;
      5'd14: coeff = 10'd123;
    endcase
endmodule
```

```markdown
// Coefficients for a 31-tap low-pass FIR filter with Wn=.125 (eg., 3kHz for a
// 48kHz sample rate). Since we’re doing integer arithmetic, we’ve scaled
// the coefficients by 2**10
// Matlab command: round(fir1(30,.125)*1024)
```

```verilog
// /////////////////////////////////////////////////////////////////////////////
```

```markdown
module coeffs31(
    input wire [4:0] index,
    output reg signed [9:0] coeff
);
```

```verilog
// tools will turn this into a 31x10 ROM
always @(index)
```

```markdown
// /////////////////////////////////////////////////////////////////////////////
```
5'd15: coeff = 10'sd128;
5'd16: coeff = 10'sd123;
5'd17: coeff = 10'sd110;
5'd18: coeff = 10'sd91;
5'd19: coeff = 10'sd69;
5'd20: coeff = 10'sd46;
5'd21: coeff = 10'sd26;
5'd22: coeff = 10'sd10;
5'd23: coeff = 10'sd0;
5'd24: coeff = -10'sd5;
5'd25: coeff = -10'sd7;
5'd26: coeff = -10'sd6;
5'd27: coeff = -10'sd5;
5'd28: coeff = -10'sd3;
5'd29: coeff = -10'sd1;
5'd30: coeff = -10'sd1;

default: coeff = 10'hXXX;
endcase
endmodule
module mix(input reset, clock, ready, input signed [17:0] mix_select, 
    input signed [17:0] signal1, signal2, 
    output wire signed [17:0] mixed);

reg signed [17:0] sigout_d, sigout;
reg signed [17:0] l_mix_select; // latch mix-select on positive edge of clock

always @(posedge ready)
    l_mix_select <= mix_select;

wire signed [4:0] mix_mask;
assign mix_mask = (l_mix_select[17:13]); // mask the lower bits, get 5 bit mix selector
assign mixed = sigout;
wire signed [5:0] signal1_weight, signal2_weight;

assign signal1_weight = {1'b0,((mix_mask >>> 1) + 8)}; // calculate weights of both signals
assign signal2_weight = {1'b0,(15-((mix_mask >>> 1) + 8))}; // based on mask

reg signed [31:0] sig_weight1, sig_weight2;
reg signed [31:0] sig1_mult, sig2_mult;

always @(posedge clock) begin
    sig1_mult <= signal1*signal1_weight; // multiply by weight (out of 16)
sig2_mult <= signal2*signal2_weight; // divide by 16 to get actual weight
    sig_weight1 <= (sig1_mult/16); //
sig_weight2 <= (sig2_mult/16);
sigout_d <= sig_weight1[17:0] + sig_weight2[17:0]; // add two weighted signals
    sigout <= sigout_d; // pipeline calculation
end
endmodule
module pan (clk, reset, ready, pan_select, input_signal, left_signal_out, right_signal_out);

input clk, reset, ready;
input signed [17:0] pan_select, input_signal;
output signed [17:0] left_signal_out, right_signal_out;

// 5-bits yields -16 (full left) to 15 (full right) pan
wire signed [4:0] pan_mask;
reg signed [17:0] l_pan_select;
assign pan_mask = l_pan_select[17:13];

wire signed [5:0] left_index, right_index;
reg signed [5:0] left_weight, right_weight;

assign left_index = {1'b0,((pan_mask >>> 1) + 8)};
assign right_index = {1'b0,(15-((pan_mask >>> 1) + 8))};

reg signed [31:0] left_weighted, right_weighted, left_mult, right_mult;
assign left_signal_out = left_weighted[17:0];
assign right_signal_out = right_weighted[17:0];

always @(posedge clk) begin
    if(ready) l_pan_select <= reset ? 0 : pan_select;
    left_mult <= (input_signal*left_weight);
    right_mult <= (input_signal*right_weight);
    left_weighted <= (left_mult/16);
    right_weighted <= (right_mult/16);
end

// equal-power gain lookup table
always @(*) begin
    case(left_index)

48
0: begin
   left_weight <= 16;
   right_weight <= 0;
end
1: begin
   left_weight <= 15;
   right_weight <= 4;
end
2: begin
   left_weight <= 15;
   right_weight <= 5;
end
3: begin
   left_weight <= 14;
   right_weight <= 7;
end
4: begin
   left_weight <= 14;
   right_weight <= 8;
end
5: begin
   left_weight <= 13;
   right_weight <= 9;
end
6: begin
   left_weight <= 12;
   right_weight <= 10;
end
7: begin
   left_weight <= 11;
   right_weight <= 11;
end
8: begin
   left_weight <= 11;
   right_weight <= 11;
end
9: begin
   left_weight <= 10;
   right_weight <= 12;
end
10:  begin
    left_weight <= 9;
    right_weight <= 13;
end
11:  begin
    left_weight <= 8;
    right_weight <= 14;
end
12:  begin
    left_weight <= 7;
    right_weight <= 14;
end
13:  begin
    left_weight <= 5;
    right_weight <= 15;
end
14:  begin
    left_weight <= 4;
    right_weight <= 15;
end
15:  begin
    left_weight <= 0;
    right_weight <= 16;
end
dcase
end
endmodule
A.10 Patchbay (patchbay.v)

```verilog
//timescale 1ns / 1ps

// Patchbay — provides physical interface for routing table update
// for the Field-Programmable Audio Effects Rack

module patchbay (/*AUTOARG*/
    // Outputs
    scan_channels, update_routing, src, dest, scan_hit, scan_miss, match_history,
    // Inputs
    reset, clock, probe_channels
);

// TODO: read external routing table to silence redundant updates...

parameter LOG_N = 4;
localparam N = (1 << LOG_N);

output [LOG_N-1:0] src, dest;
input reset, clock;
reg [2*LOG_N-1:0] src_dest;
input [N-1:0] probe_channels;
output [N-1:0] scan_channels;
reg [N-1:0] scan_pattern;
assign scan_channels = ~(1'b1 << (src));

output update_routing;

reg update;
output reg [N-1:0] match_history;
output wire scan_hit, scan_miss;
assign scan_hit = (probe_channels[dest] == 0);
assign src = src_dest[LOG_N-1:0]; // routing table address
assign scan_miss = (src == {LOG_N{1'b1}}) & (match_history);
assign dest = src_dest[2*LOG_N-1:LOG_N]; // routing table data
assign update_routing = scan_miss | scan_hit;
```
always @(posedge clock) begin
    if(reset) begin
        src_dest <= 0;
        match_history <= {N1'b1};
        end
    else begin
        src_dest <= src_dest + 1;
        match_history <= {match_history[14:0], probe_channels[dest]};
        end
    end
endmodule
A.11 Signal Generator (signal_generator.v)

```vhdl
//timescale 1ns / 1ps
/////////////////////////////////////////////////////////
// Signal Generator Module for 6.111 2010 Final Project
// Marc Resnick/Drew Shapiro
/////////////////////////////////////////////////////////
module signal_generator(freq_select, clock, reset, ready, shape, wave_in, wave_out);
parameter CLK_FREQ = 27_000_000; //27Mhz clock on lab kit
input signed [17:0] freq_select;
localparam SINE = 2'b00;
localparam SQUARE = 2'b01;
localparam SAWTOOTH = 2'b10;
localparam TRIANGLE = 2'b11;
input clock, reset, ready;
input [1:0] shape;
input signed [17:0] wave_in;
output signed [17:0] wave_out;
// select the appropriate signal to expose
reg signed [17:0] value;
// assign wave = value;
wire signed [17:0] mix_select;
assign mix_select = wave_in == 0 ? 18'b100000000000000000 : 0;
mix wet_dry_mix ( //allow signals to be chained/combined
.reset(reset),
.clock(clock),
.ready(ready),
.mix_select(18'b0),
signal1(value),
signal2(wave_in),
mixed(wave_out)
);
```
reg [9:0] index;
reg [24:0] count, count_stop;

always @(posedge clock) begin
  if(reset) begin
    index <= 0;
    count <= 0;
    count_stop <= 0;
  end else begin
    count <= (count == count_stop) ? 0 : count + 1;
    // count until count_stop is reached
    count_stop <= $unsigned(freq_select) >> 10;
    // then increment index and reset count
    index <= (count == count_stop) ? ((index == 10'd1023) ? 10'b0 : index + 1) : index;
    // also reset index when it reaches 1023
  end
end

wire signed [17:0] square_value, sine_value, sawtooth_value, triangle_value;
// make 4 different shapes available
assign square_value = (index > 10'd511)? 18'b10000000000000000 : 18'b01111111111111111;
// logic for square wave
sinelut lut_sine(.THETA(index), .SINE(sine_value)); // and lookup tables for others
sawtoothlut lut_sawtooth(.index(index), .wave(sawtooth_value)); // input is index, output is value
trianglelut lut_triangle(.index(index), .wave(triangle_value)); // LUTs are full-wave, 1024 samples

always @( *) // select wave to output
  case(shape)
    SINE: value = sine_value;
    SQUARE: value = square_value;
    SAWTOOTH: value = sawtooth_value;
    TRIANGLE: value = triangle_value;
  endcase
endmodule