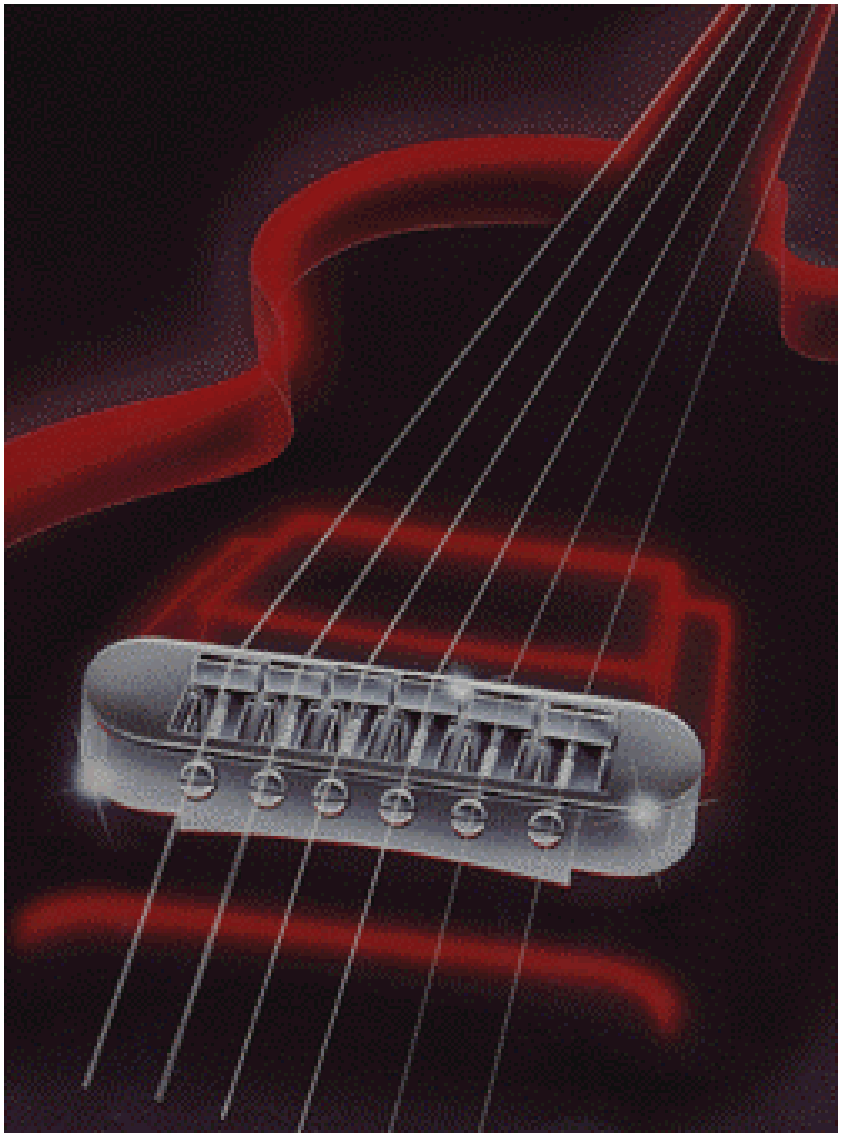
# THE GUITAR EFFECTOR



Jason Cardillo (jjc2124)
Jun Hao Ip (ji2115)
Chih-Chieh Lin (cl2291)

# Table of Contents

**Abstract**

The Guitar Effector simulates a guitar effects pedal. The stereo audio port, AKM AK4565 Low Power Audio Codec on the Xilinx XESS XSB 300E is utilized to receive analog sound from an electric guitar. The onboard signal processing utilities to digitize the sound, implement a digital effect and output analog sound to a speaker in real-time. This design allows the user to select between the following digital effects: Clean (no effect), Distortion, Delay, Echo, Flange and Flange with feedback. All the digital effects will be written in VHDL. The User Interface, which allows the user to select the digital effect, will be written in C. The SRAM will be used to store the sound bits for the digital effects involving delay (Delay, Echo, Flange).

## Design Overview

The design of the guitar effector involved four major components: Audio Codec, Effect Generation, SRAM, and the Effect Selection. The Audio Codec component involves the real-time playback of the guitar sound. This involved initializing the codec and simultaneously reading and playing the sound back in real-time. The Effects Generation component as its name indicates deals with generating the individual effects. As the 16-bit audio signal is received from the codec all of the effects are performed on the signal. Then based on the user selection the proper effect is chosen by using a multiplexor and sent back to the Audio Codec module. The SRAM module as its name describes the interface with the SRAM. The effects involving delay require that the sound be stored in the SRAM. The last component is the Effect Selector. This module is contains both C and VHDL code. The user is prompted on the Minicom window to select an effect that they would like to experience. This selection is entered on the keypad and sent via the UART to the VHDL hardware module.

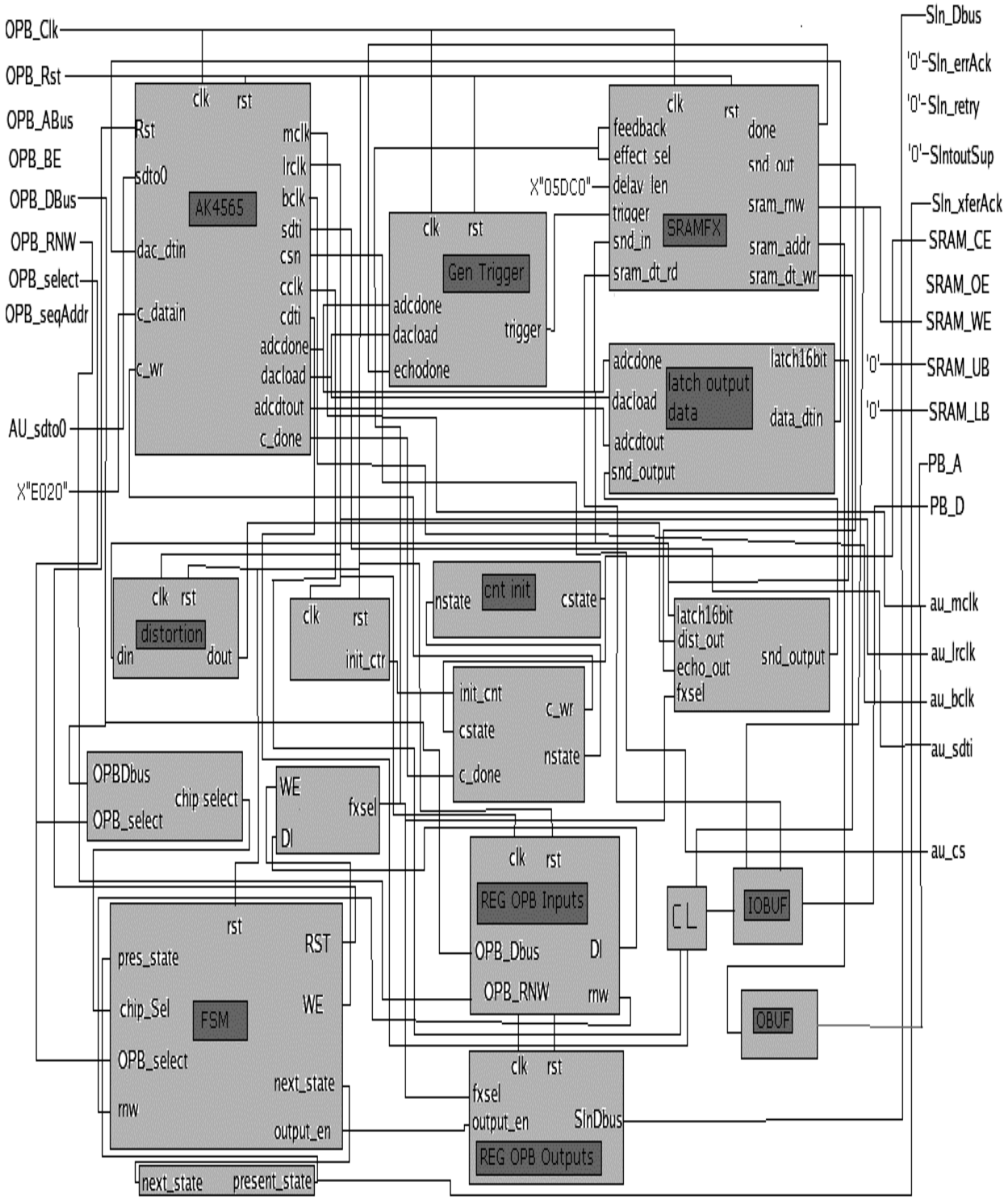



Figure 1.Block diagram of the Guitar Effector

## Real-Time Playback: The Audio Codec

### AK4565 Codec controller

The basis audio codec controller was implemented according to the given data sheet. The first step was to generate a set of required clocks. Next, we had to get red/green (in/out) jacks on the Xilinx board working. This was achieved by sending control signals. This module was broken into two parts: Data handling and control.

### Clock Generation

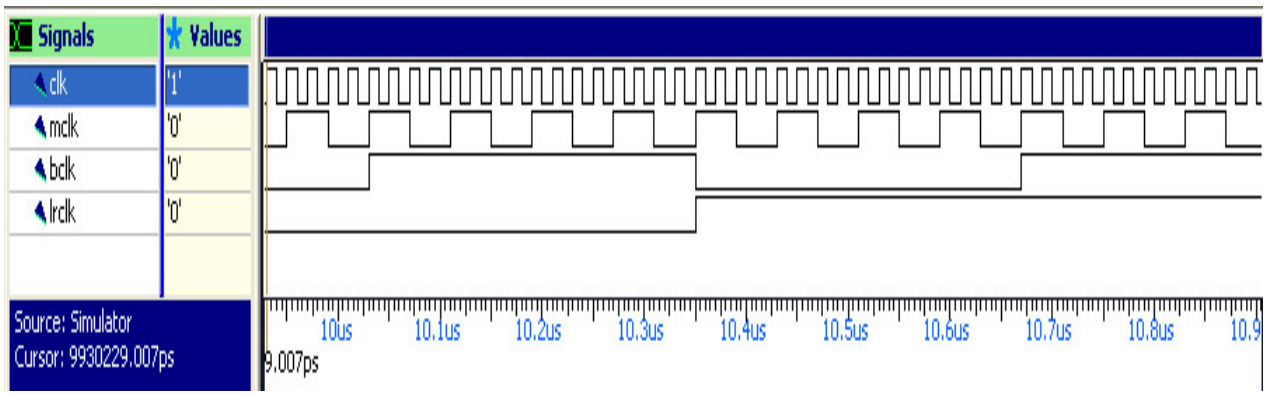



Figure 2. Clock Simulation

The manual tells us none of the clocks need to be in phase, but as was discovered later the mclk must be synchronized with lrclk. In fact, all clocks needed to be synchronized with the rising edge of mclk. Also, the manual wrongly described the channel processing sequence. In the manual, left channel (mono input) is being processed when lrclk is high. After hours of experiment, this sequence described in the Xilinx manual happens to be the reverse. This clock generation was attained by using one 2-bit and 8-bit counter. The 2-bit counter generates the master clock for the AK4565 chip. This clock is then fed into the 8-bit counter to generate the bclk and lrclk. This process is described in Figure 3.

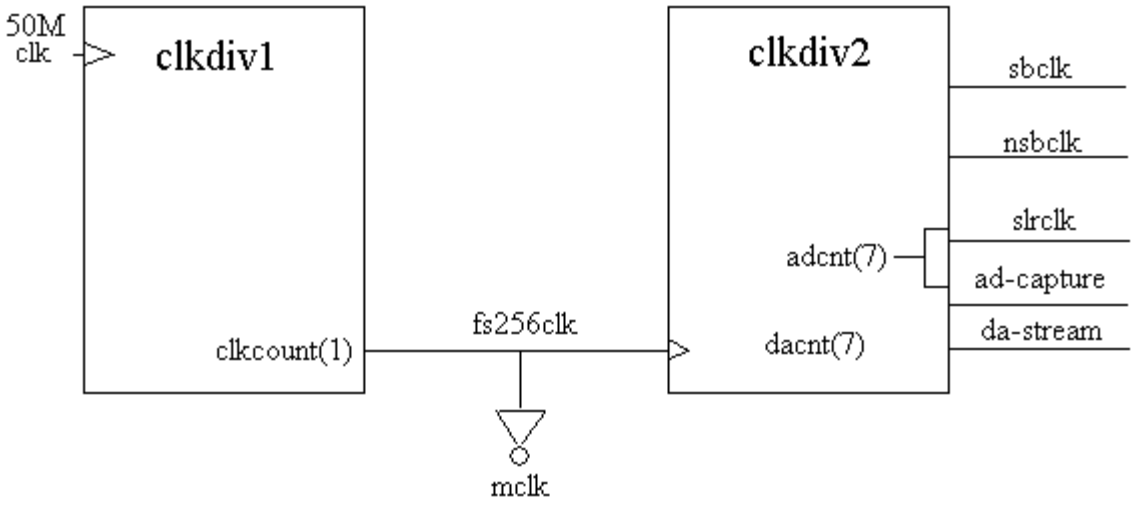


Figure 3. Clock Generation

Data Handling

The first part of the Audio Codec module is the data handling. When lrclk='0' (left channel), serial sound data is being latched into (serial to parallel) input shift register. After 16 cycles and lrclk='1', the 16-bit sound input data is being latched into the adc_dtout register. Also when lrclk='0', parallel loaded (parallel to serial) output shift register will shift out its MSB each cycle to the ak4565 serial input port.

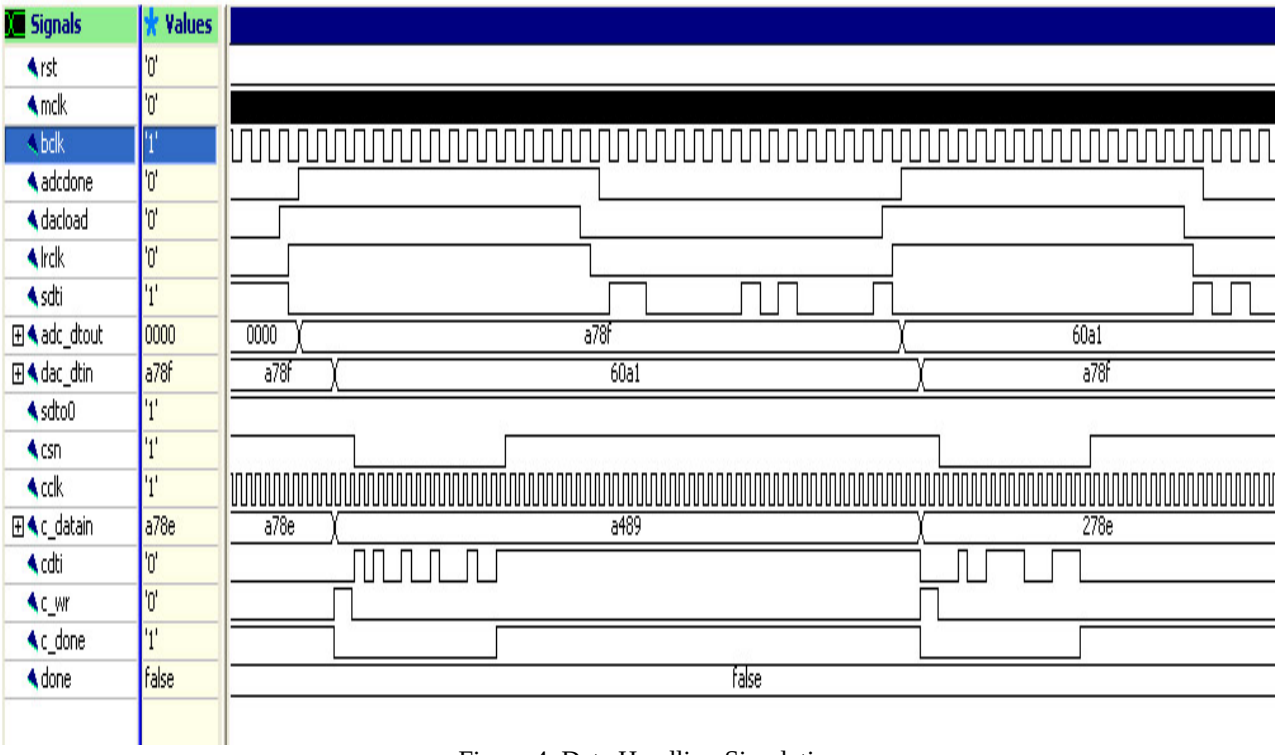


Figure 4. Data Handling Simulation

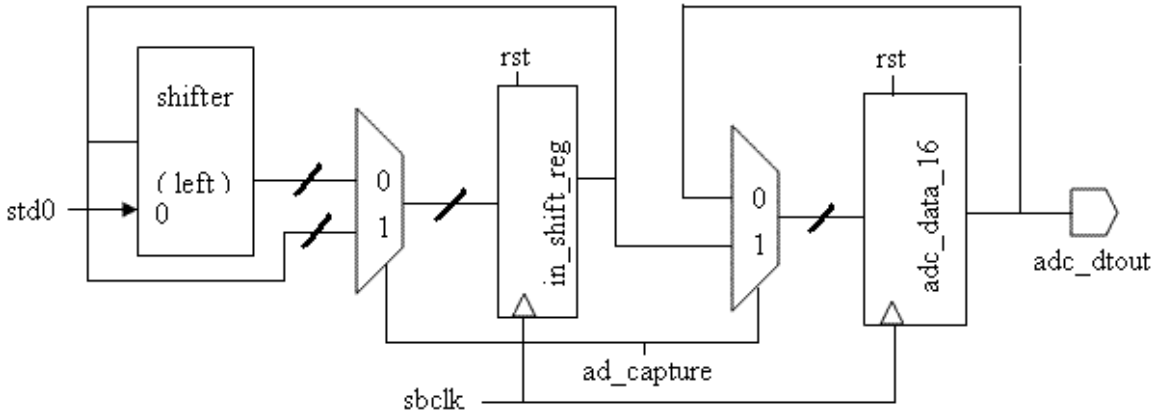



Figure 5. Audio Serial to parallel

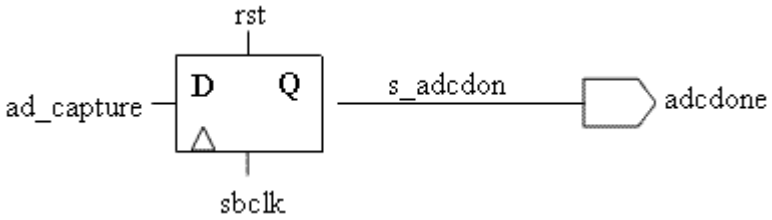



Figure 6. Audio serial to parallel done signal

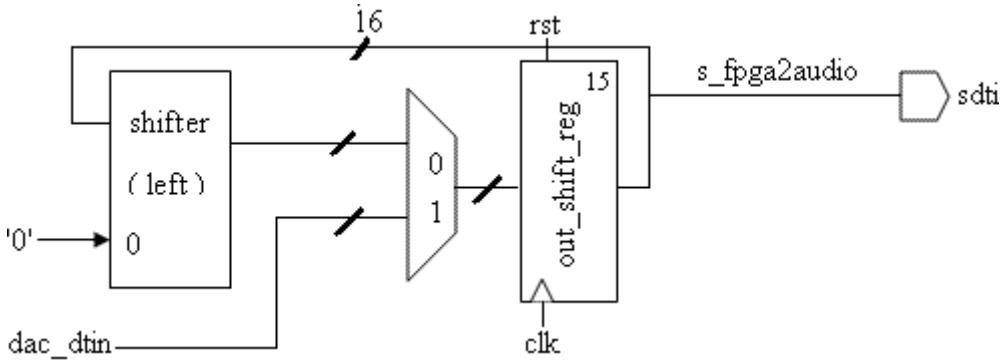



Figure 7 .Audio parallel to serial output

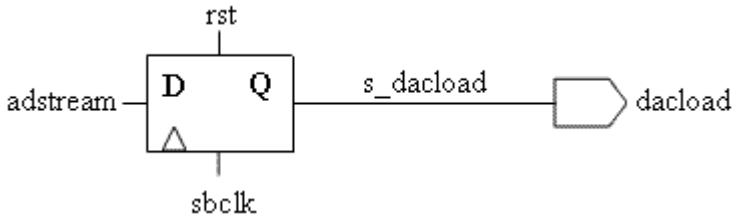



Figure 8 .Audio output parallel load signal

AK4565 Control Data

The second part of the Audio codec module is the control. Control signals can be separate from the data signals. It is a 16-bit serial output, with 3 bits opcode, 5 bit address and 8 bits data. When control data is being sent, it must be sent on the falling edge of control clock, and chip select signal must be zero. When transmission is finished, chip select returns back to one. This can be implemented by a finite state machine where each bit sent represents a state. A 4-bit counter and s_done signal to establish a finite state machine structure for control data transmission. When c_wr signal is triggered, the state machine is activated. When s_done is zero, s_csn chip select signal is also zero signaling ak4565 to grab input signal. As soon as the counter reaches 15, s_done is triggered, thus driving s_csn back to one to stop transmission. Since minimum 200ns clock requirement is needed, 3MHz clock was used for control transmission. It was assumed that control signal is completely asynchronous with data clocks.

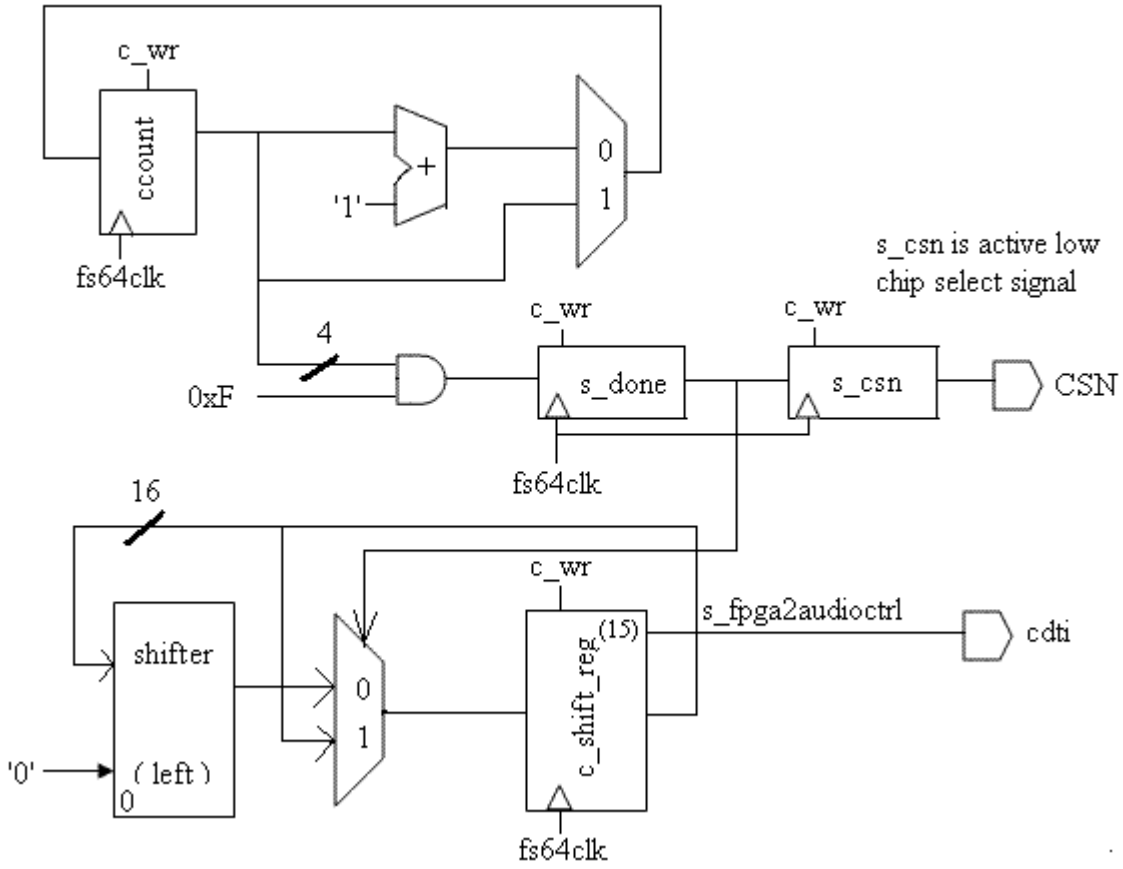



Figure 9. AK4565 Controller

## Implementation of Effects

The following effects were implemented: Distortion, Delay, Echo and two types of Flange. The process of implementing an effect began with its design. Research was done on many different digital effects. The effects were then chosen based on the sound of the effect, ease of implementation (due to the fact that there is a limited amount of clock cycles that can be utilized to implement an effect) and the project deadline. Sample guitar clips were recorded using the Sound Recorder in the Accessories folder of windows. The desired effect was programmed in Matlab and the sound was passed through this effect. From these Matlab simulations, it was possible to fine-tune the effect before it was synthesized in hardware. By listening to the sound the particular effect, the various parameters of the effect (length of delay, decay of signal) could be adjusted without having to synthesize the VHDL code in hardware. Also this allowed parallelism as our group divided the labor as some members could use the board to test other components of our design while another worked on the effect generation in Matlab.

### Distortion

There are two well-known methods exist for performing distortion of an audio signal: Clipping and absolute value. Like many decisions for this project, there was a trade-off. While easier to implement, the absolute value solution has a noisier sound. The clipping method, while much more difficult to implement has a much better sound. Our design team decided that absolute value was the better solution, because the implementation takes less clock cycles and is not an amplitude-dependent algorithm. For the clipping method, a threshold at which the sound was not to surpass has to be set. Lower volumes, where the entire value of the signal falls below this threshold would not experience the distortion effect. The clipping threshold would have to change depending on the amplitude of the signal. There would be many clock cycle-consuming comparisons in order to enact this method. The distortion algorithm that was implemented simply checks the sign bit of the 16bit signal and if this bit is a 1, all 16 bits in the signal are negated.

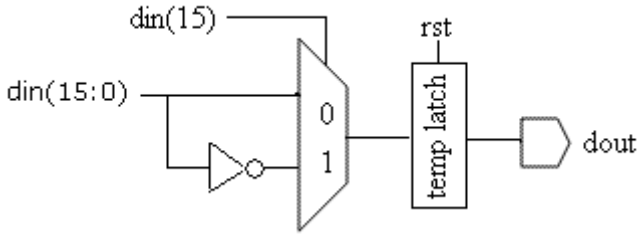



Figure 10 . Distortion

### SRAM Effects

SRAM_FX is the main digital signal processing unit in our project. By using SRAM as a circular buffer, we can achieve various effects by doing time domain calculations. The most common effects are delay, echo, reverberation, and flanging. Delay, Echo, and flanging were implemented because they have similar structures. Reverberation was not implemented for following reasons:

the electric guitar strings are intrinsically flexible and have a long settling time. Reverb is the sound you hear in a room with hard surfaces where the sound bounces off the floors and walls for a period of time after the initial sound stops. The problem is that the initial sound of the guitar does not stop because the strings keep vibrating. This posed a problem because the various reverberation techniques we implemented did not sound good. Also, reverberation has a tendency to amplify the noise signal without using a compandor. If we are to do reverberation, we need to increase number of states to include additional delays.

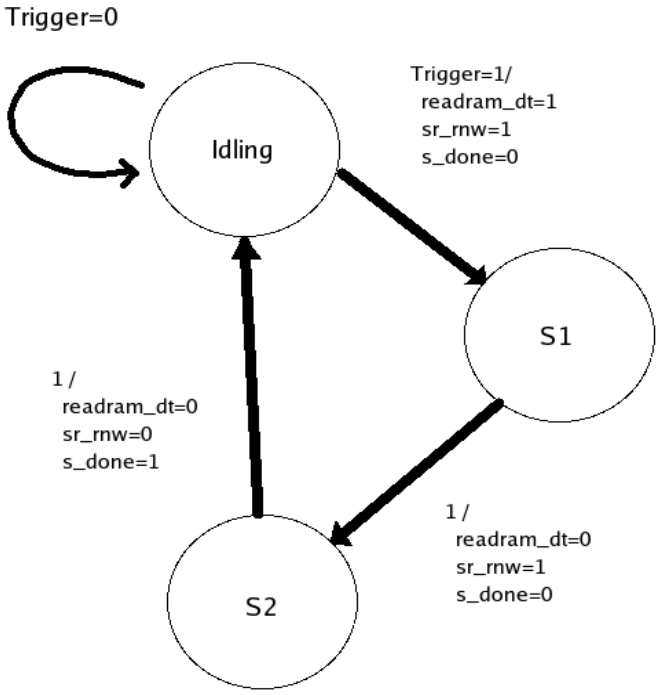



Figure 11 . Basic calculation FSM

The state machine (Figure 11 ) describes our SRAMFX protocol. By default, the state machine starts in the idle state until it receives a trigger signal. Upon receiving this signal SRAM data is latched. In state 1, the appropriate delay effect output is calculated and latched. In state 2, depending on the effect selection (feedback or not), either the incoming data (from the codec) or the delay effect output is stored in the SRAM.

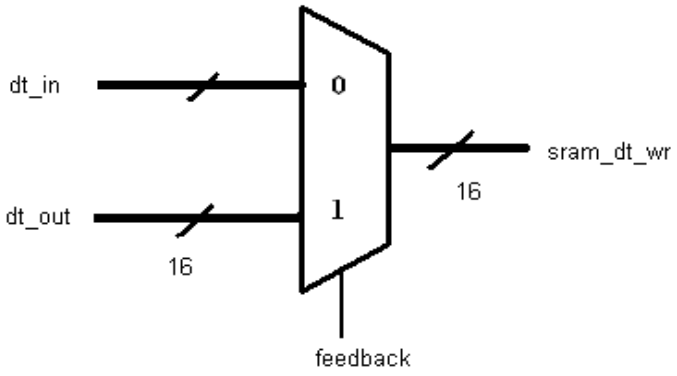


Figure 12. Feedback MUX

The delay calculation is achieved using the scheme described in Figure 13. Every clock cycle the SRAM address is calculated continuously except when cstate is S2. Depending on what effect is selected the delay goes through a series of multiplexers.

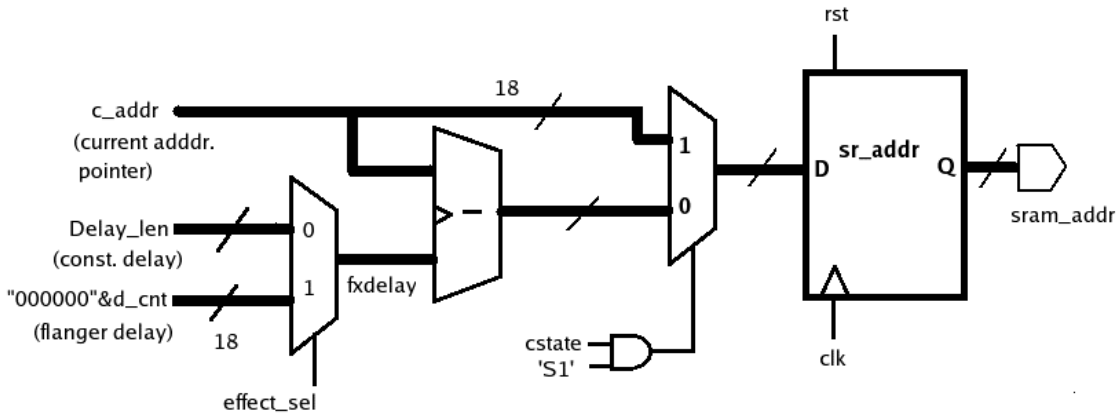



Figure 13.SRAM address (Delay) processing unit

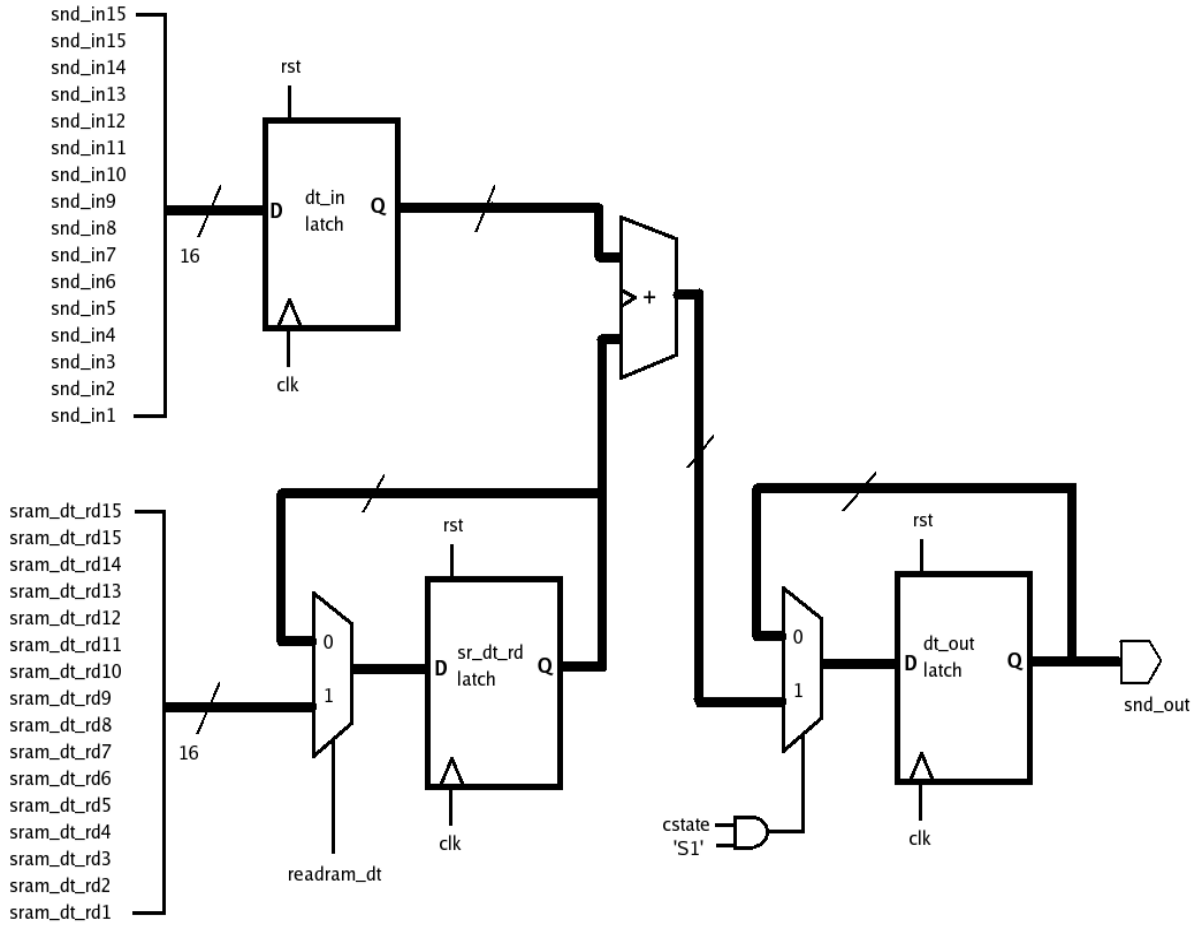



Figure 14. Feedback system

The feedback system is described in Figure 14. The following formula is implemented

$$\mathbf{Y(n) = \tfrac{1}{2}\, X(n) + \tfrac{1}{2}\, Y(n\text{-}d)}$$

A divide by 2 is used because the addition will cause the output signal to saturate. The output level must always be less than or equal to the max 16-bit signed number. This division is carried out by a shift right operation.

Delay

The delay effect is implemented by adding the current 16-bit signal with an attenuated signal that was played a constant period of time ago. The amount of delay and decay are constant parameters. The delay that was used was .5 seconds. This delay was achieved by hardwiring X”05DC0” into the delay_len input to the SRAM_FX module.

Echo

The Echo effect is similar to that of delay except that the signal has feedback. The same delay of .5 seconds was used.

Flange

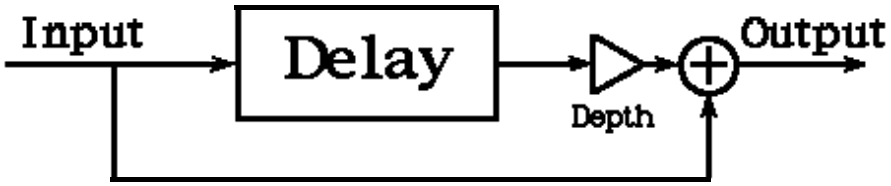



Figure 15 .Block diagram for a flange implementation

The Flange effect is created by mixing a signal with delayed copies of itself, where the length of the delay is changing continuously. This delay is usually ranges from 0 to 5ms. There are three primary controls for the flange effect. They are referred to as depth, rate, and decay. Depth refers to how long the delay is, Rate refers to how fast the delay changes, and Decay describes the attenuation of the delayed sound. In order to achieve these delays the 50 MHz master clock is divided down to 381Hz. This corresponds to a 2.6 ms period. This number is the delay variation rate. This controls the depth. Then, this slow clock is fed into a 12-bit up/down counter, which we limited it from 0 to 144, which it is corresponded to the depth. This caused the various delays to range from 0 to 3ms.

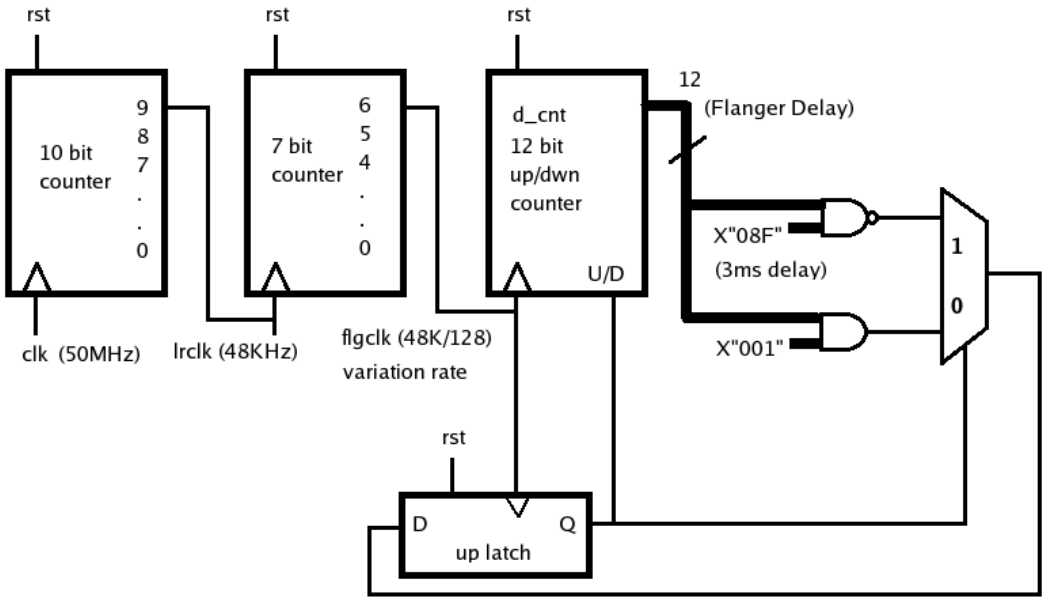



Figure 16 .Flange hardware – Delay calculation

## User Interface

The user interface is simply a menu that prompts the user to select one of six guitar effects. The user can enter any numbers 1-6, which correspond to clean tone, distortion, delay, echo, flange w/o feedback and flange w/ feedback respectively. The default setting is clean tone. If the user attempts to press any other button on the keyboard except the buttons 1-6, the program does not respond and the current effect remains playing. Once the effect is selected a three-bit sequence is loaded onto the OPB data bus and sent to our GDSP hardware module. This three-bit sequence provides the control bits for the SRAMFX module and the demultiplexor that sends one of three sounds back to the audio codec module for playback. The keystroke is received and an interrupt is generated. The user selection is read from the two-bit array. The 3-bit sequence is then generated. This piece of code was very similar to lab2, where a typewriter was implemented.

## Conclusions

Lessons Learned

Jason.

This was my first project where as a group we could pick our own project. In my previous courses, the specific project was assigned and I as the student had to complete it. There was no choice or maybe the professor proposed three possible projects and I had to complete one of them. This was the first project that as a group we had to essentially create ourselves. Not only create we basically had to implement it ourselves also. While the Professor Edwards and TA Marcio are familiar with the Xilinx boards and peripherals, they may not familiar with the details of implementing unique projects that the students have come up with. Thus the burden of implementation falls more strictly on the student as there is limited help you could receive from the professor or TA. This is good because it forces you to be more resourceful. I also learned a great deal about VHDL. While I have taken a previous course in VHDL, it was nothing this advanced. We had to design a simple component. Throughout the course of this project I learned a great deal about how to implement VHDL. I learned how to interpret manuals and data sheets in order to make our project work.

Jun Hao

This is the second course I have taken, where the majority of the work involved VHDL. This was an excellent design course for me. I learned to read and make sense (try to) out of the data sheets. I also got much experience in coding in VHDL. Many times, the components I designed worked great in simulation. The code is logically correct; however, in order to synthesize this VHDL code into hardware, timing issues needed to be seriously considered as not all code that works in simulation is synthesizable. Also, I have learned that it is important to break VHDL into small processes. This makes things easier for the synthesizer to handle. In addition, I gained experience with implementing inter-chip communication protocols such as Audio port, SRAM, and OPB. This is a fun practical project. If I get more time, I could of make it much more interesting by including some Frequency domain filtering.

Chih-Chieh

This project is the first where I was able to design, create and implement my(our) own idea. All of my previous projects were designed and assigned by a professor. We always had directions to follow and a clear scope in mind and never had to start from scratch. While some of my previous projects involved C programming, this project was my first using VHDL. I have never programmed an FPGA or Microprocessor. I have never even written any VHDL code before. The last time I learned anything about digital design or logic was six years ago. The skeleton of the project was in VHDL and a lot of timing diagrams. I learned in details how to complete a digital design project and learned the functions of the various files (.vhd,.mss, .mpd,.mhs.,etc). I learned not only “read” the manual and timing diagram but also implement it. (Timing issue is really a trouble in digital design.) I learned how to manipulate the board, peripherals and

processor. For example, since our project was concerning the delay parts of our signals, we had to manipulate the SRAM. Also, I learned how to communicate with peripherals by using a combination of C and VHDL.

Future advice

The board takes a while to compile and synthesize. Each time we compiled and synthesize it took about 5-10 minutes to complete although it seemed like an eternity. We suggest that future students use your brains for debugging and implementing. Do what you can without synthesizing. We simulated all of our digital effects in Matlab, fine tuning all of the various parameters such as delays, and decays, while we listened to the sound. This saved us a lot of time as we could fine-tune our digital effects in minutes rather than hours. Besides the simulations of the sound effects, we simulated each one of our components before simulation. We verified the waveforms were what we wanted before synthesizing the hardware. We accomplished this by using Xilinx Student Edition software that came with the John F. Wakerly book Digital Design Principles & Practices purchased for Professor Novick's class: Advanced Logic Design. We also strongly suggest that students take a C and VHDL course before you enroll in Embedded Systems. We feel that this is not a beginner VHDL course and it would be very beneficial if you had prior experience with both of theses languages. The reviews in the first two lectures just are not enough if you have never been exposed to either language.

Future Work

As is, the SRAM data processing technique implemented functions properly. However, there is still space for improvement. There may be a bug within the address calculation process. For some reason, the counter generated address only works with certain number of bits. If we arbitrary modify the number of address counter bits, our system will fail. One suggested solution is to attempt to use as little IF statements as possible, and use as much combinational logic as possible due to our team's uncertainty about how 'IF' statements are synthesized into hardware. In many cases, IF statements should be simplified to combinational logic, but sometimes it will be turn into one big MUX (like what our diagrams shown. The SRAM state machine can be improved so that it may include higher order delays. The effect functions can be parameterized. This will allow the user to control how SRAM effects work. The user could control all the delays and decays.

## **Responsibilities**

Jason

- AK4565 Controller
- Distortion
- User Interface(C,VHDL)
- OPB Protocol
- SRAMFX Module

Jun Hao

- Data Handling
- Clock Generation
- SRAMFX Module
- VHDL Integration of components

Chih-Chieh

- Matlab simulations and fine-tuning of all effects
- User Interface (UART)
- System File Modification
- SRAMFX Module

**References**

GM Arts Home Page. http://users.chariot.net.au/~gmarts/index.html

Lehman, Scott  Flanging. http://www.harmony-central.com/Effects/Articles/Flanging/

## Appendix

C program for User interface

```
#include "xparameters.h"
#include "xbasic_types.h"
#include "xio.h"
#include "xintc_l.h"
#include "xuartlite_l.h"

/* get character, or say, selection from keyboard */
unsigned char buff;

/*
 * Interrupt service routine for the UART
 */
void uart_handler(void *callback)
{
  Xuint32 IsrStatus;
  Xuint8 incoming_character;
  /* Check the ISR status register so we can identify the interrupt source */
  IsrStatus = XIo_In32(XPAR_MYUART_BASEADDR + XUL_STATUS_REG_OFFSET);
  /* disable interrupts */
  microblaze_disable_interrupts();

  if ((IsrStatus & (XUL_SR_RX_FIFO_FULL | XUL_SR_RX_FIFO_VALID_DATA)) != 0)
  {
    incoming_character =
      (Xuint8) XIo_In32( XPAR_MYUART_BASEADDR + XUL_RX_FIFO_OFFSET );
      buff= incoming_character;
  }
  /* Enable interrupts */
  microblaze_enable_interrupts();
}

int main()
{
  int j, k, addr;

  /* Use a 2-character array to store input character(1st bit) and null
character(2nd bit) */
  char disp_char[2];
  char last='\0';
  disp_char[1]='\0';

  /* Enable UART interrupts and register uart_handler as the ISR */
  XIntc_RegisterHandler( XPAR_INTC_BASEADDR, XPAR_MYUART_DEVICE_ID,
                  (XInterruptHandler)uart_handler, (void *)0);
  XIntc_mEnableIntr( XPAR_INTC_BASEADDR, XPAR_MYUART_INTERRUPT_MASK);
  XIntc_mMasterEnable( XPAR_INTC_BASEADDR );
  XIntc_Out32(XPAR_INTC_BASEADDR + XIN_MER_OFFSET,
XIN_INT_MASTER_ENABLE_MASK);
  microblaze_enable_interrupts();
  XUartLite_mEnableIntr(XPAR_MYUART_BASEADDR);
```

```
/* USER INTERFACE SCREEN */
   print("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
   print("\r\n\tWELCOME TO THE GUITAR EFFECTOR\r\n\n\n");

   print("Select which effect you want to experience:\r\n\n");

   print(" 1 - Clean\r\n");
   print(" 2 - Distortion\r\n");
   print(" 3 - Delay\r\n");
   print(" 4 - Echo\r\n");
   print(" 5 - Flange w/o feedback\r\n");
   print(" 6 - Flange w/ feedback\r\n\n\n");

   print("Enter a number 1-6: \r\n");


  for(;;)
  {
    /* get character from buffer to display */
      microblaze_disable_interrupts();
      disp_char[0] = buff;
      microblaze_enable_interrupts();

      /* Force the input to be one of the selections */
      if(disp_char[0]=='1' || disp_char[0]=='2'|| disp_char[0]=='3' ||
disp_char[0]=='4' || disp_char[0]=='5' || disp_char[0]=='6')
      {
          if(disp_char[0]!=last)
          {
             /* print to minicom */
             print(disp_char);
             print("\r");

             /* set mux controls based on input, first 3 bits are
                the mux controls */
                   if(disp_char[0]=='1')
                    j=0x00000000;

             if(disp_char[0]=='2')
              j=0x20000000;

             if(disp_char[0]=='3')
              j=0x80000000;

             if(disp_char[0]=='4')
                    j=0xC0000000;

             if(disp_char[0]=='5')
                    j=0xA0000000;

             if(disp_char[0]=='6')
                    j=0xE0000000;

             addr = 0xFEFF1001;
             XIo_Out32(addr, j);

             k = XIo_In32(addr);
```

```
            print("\r");
           }
       }
/* make sure that if user enters selects same effect as current effect
nothing happens */
  last=disp_char[0];
  }
 return 0;
}
```

**system.mhs**

```
# Parameters
PARAMETER VERSION = 2.1.0

# Global Ports

PORT FPGA_CLK1 = FPGA_CLK1, DIR = IN
PORT RS232_TD = RS232_TD, DIR=OUT
PORT RS232_RD = RS232_RD, DIR=IN

PORT SRAM_CE = SRAM_CE, DIR = OUT
PORT SRAM_OE = SRAM_OE, DIR = OUT
PORT SRAM_WE = SRAM_WE, DIR = OUT
PORT SRAM_UB = SRAM_UB, DIR = OUT
PORT SRAM_LB = SRAM_LB, DIR = OUT
PORT PB_A = PB_A, DIR = OUT, VEC = [17:0]
PORT PB_D = PB_D, DIR = INOUT, VEC = [15:0]

PORT au_cs = au_cs, DIR = OUT
PORT au_mclk = au_mclk, DIR = OUT
PORT au_lrclk = au_lrclk, DIR = OUT
PORT au_bclk = au_bclk, DIR = OUT
PORT au_sdti = au_sdti, DIR = OUT
PORT au_sdto0 = au_sdto0, DIR = IN

# Hint: Put your peripheral first in this file so it will be analyzed
# first and will generate errors faster.

# gDSP example peripheral

BEGIN gDSP
 PARAMETER INSTANCE = gDSP_peripheral
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_BASEADDR = 0xFE100000
 PARAMETER C_HIGHADDR = 0xFE1fffff
 PORT OPB_Clk = sys_clk
 PORT SRAM_CE = SRAM_CE
 PORT SRAM_OE = SRAM_OE
 PORT SRAM_WE = SRAM_WE
 PORT SRAM_UB = SRAM_UB
 PORT SRAM_LB = SRAM_LB
 PORT PB_A = PB_A
 PORT PB_D = PB_D
 BUS_INTERFACE SOPB = myopb_bus
 PORT au_cs = au_cs
 PORT au_mclk = au_mclk
 PORT au_lrclk = au_lrclk
 PORT au_bclk = au_bclk
 PORT au_sdti = au_sdti
 PORT au_sdto0 = au_sdto0
END

# Interrupt controller for dealing with interrupts from the UART

BEGIN opb_intc
```

```
 PARAMETER INSTANCE = intc
 PARAMETER HW_VER = 1.00.c
 PARAMETER C_BASEADDR = 0xFFFF0000
 PARAMETER C_HIGHADDR = 0xFFFF00FF
 PORT OPB_Clk = sys_clk
 PORT Intr =  uart_intr
 PORT Irq =   intr
 BUS_INTERFACE SOPB = myopb_bus
END

# The main processor core

BEGIN microblaze
 PARAMETER INSTANCE = mymicroblaze
 PARAMETER HW_VER = 2.00.a
 PARAMETER C_USE_BARREL = 1
 PARAMETER C_USE_ICACHE = 0
 PORT Clk = sys_clk
 PORT Reset = fpga_reset
 PORT Interrupt = intr
 BUS_INTERFACE DLMB = d_lmb
 BUS_INTERFACE ILMB = i_lmb
 BUS_INTERFACE DOPB = myopb_bus
 BUS_INTERFACE IOPB = myopb_bus
END

# Block RAM for code and data is connected through two LMB busses
# to the Microblaze, which has two ports on it for just this reason.

# Data LMB bus

BEGIN lmb_v10
 PARAMETER INSTANCE = d_lmb
 PARAMETER HW_VER = 1.00.a
 PORT LMB_Clk = sys_clk
 PORT SYS_Rst = fpga_reset
END

BEGIN lmb_bram_if_cntlr
 PARAMETER INSTANCE = lmb_data_controller
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00000FFF
 BUS_INTERFACE SLMB = d_lmb
 BUS_INTERFACE BRAM_PORT = conn_0
END

#Instruction LMB bus

BEGIN lmb_v10
 PARAMETER INSTANCE = i_lmb
 PARAMETER HW_VER = 1.00.a
 PORT LMB_Clk = sys_clk
 PORT SYS_Rst = fpga_reset
END

BEGIN lmb_bram_if_cntlr
```

```
 PARAMETER INSTANCE = lmb_instruction_controller
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_BASEADDR = 0x00000000
 PARAMETER C_HIGHADDR = 0x00000FFF
 BUS_INTERFACE SLMB = i_lmb
 BUS_INTERFACE BRAM_PORT = conn_1
END

#The actual block memory

BEGIN bram_block
 PARAMETER INSTANCE = bram
 PARAMETER HW_VER = 1.00.a
 BUS_INTERFACE PORTA = conn_0
 BUS_INTERFACE PORTB = conn_1
END

# Clock divider to make the whole thing run

BEGIN clkgen
 PARAMETER INSTANCE = clkgen_0
 PARAMETER HW_VER = 1.00.a
 PORT FPGA_CLK1 = FPGA_CLK1
 PORT sys_clk = sys_clk
 PORT pixel_clock = pixel_clock
 PORT fpga_reset = fpga_reset
END

# The OPB bus controller connected to the Microblaze
# All peripherals are connected to this

BEGIN opb_v20
 PARAMETER INSTANCE = myopb_bus
 PARAMETER HW_VER = 1.10.a
 PARAMETER C_DYNAM_PRIORITY = 0
 PARAMETER C_REG_GRANTS = 0
 PARAMETER C_PARK = 0
 PARAMETER C_PROC_INTRFCE = 0
 PARAMETER C_DEV_BLK_ID = 0
 PARAMETER C_DEV_MIR_ENABLE = 0
 PARAMETER C_BASEADDR = 0x0fff1000
 PARAMETER C_HIGHADDR = 0x0fff10ff
 PORT SYS_Rst = fpga_reset
 PORT OPB_Clk = sys_clk
END

# UART: Serial port hardware

BEGIN opb_uartlite
 PARAMETER INSTANCE = myuart
 PARAMETER HW_VER = 1.00.b
 PARAMETER C_CLK_FREQ = 50_000_000
 PARAMETER C_USE_PARITY = 0
 PARAMETER C_BASEADDR = 0xFEFF0100
 PARAMETER C_HIGHADDR = 0xFEFF01FF
 PORT OPB_Clk = sys_clk
 BUS_INTERFACE SOPB = myopb_bus
```

```
 PORT Interrupt = uart_intr
 PORT RX=RS232_RD
 PORT TX=RS232_TD
END
```

**System.mss**

```
PARAMETER VERSION = 2.2.0
PARAMETER HW_SPEC_FILE = system.mhs

BEGIN PROCESSOR
 PARAMETER HW_INSTANCE = mymicroblaze
 PARAMETER DRIVER_NAME = cpu
 PARAMETER DRIVER_VER = 1.00.a
END

BEGIN OS
  PARAMETER PROC_INSTANCE = mymicroblaze
  PARAMETER OS_NAME = standalone
  PARAMETER OS_VER = 1.00.a
  PARAMETER STDIN = myuart
  PARAMETER STDOUT = myuart
END

BEGIN DRIVER
 PARAMETER HW_INSTANCE = myuart
 PARAMETER DRIVER_NAME = uartlite
 PARAMETER DRIVER_VER = 1.00.b
END

BEGIN DRIVER
 PARAMETER HW_INSTANCE = intc
 PARAMETER DRIVER_NAME = intc
 PARAMETER DRIVER_VER = 1.00.c
END

# Use null drivers for peripherals that don't need them
# This supresses warnings

BEGIN DRIVER
 PARAMETER HW_INSTANCE = gDSP_peripheral
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
END

BEGIN DRIVER
 PARAMETER HW_INSTANCE = lmb_data_controller
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
END

BEGIN DRIVER
 PARAMETER HW_INSTANCE = lmb_instruction_controller
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
END
```

**Microprocess Peripheral Definition File**

```
##############################################################################
##
## Microprocessor Peripheral Definition
##
##############################################################################

BEGIN gDSP, IPTYPE = PERIPHERAL, EDIF=TRUE

BUS_INTERFACE BUS = SOPB, BUS_STD = OPB, BUS_TYPE = SLAVE

## Generics for VHDL
PARAMETER c_baseaddr     = 0xFFFFFFFF, DT = std_logic_vector, MIN_SIZE = 0xFF
PARAMETER c_highaddr     = 0x00000000, DT = std_logic_vector
PARAMETER c_opb_awidth   = 32,         DT = integer
PARAMETER c_opb_dwidth   = 32,         DT = integer

## FPGA Internal Ports, connects Microblaze and FPGA module
PORT opb_abus    = OPB_ABus,    DIR = IN, VEC = [0:(c_opb_awidth-1)],     BUS
= SOPB
PORT opb_be      = OPB_BE,      DIR = IN, VEC = [0:((c_opb_dwidth/8)-1)], BUS
= SOPB
PORT opb_clk     = "",          DIR = IN,                                 BUS
= SOPB
PORT opb_dbus    = OPB_DBus,    DIR = IN, VEC = [0:(c_opb_dwidth-1)],     BUS
= SOPB
PORT opb_rnw     = OPB_RNW,     DIR = IN,                                 BUS
= SOPB
PORT opb_rst     = OPB_Rst,     DIR = IN,                                 BUS
= SOPB
PORT opb_select  = OPB_select,  DIR = IN,                                 BUS
= SOPB
PORT opb_seqaddr = OPB_seqAddr, DIR = IN,                                 BUS
= SOPB
PORT sln_dbus    = Sl_DBus,     DIR = OUT, VEC = [0:(c_opb_dwidth-1)],    BUS
= SOPB
PORT sln_errack  = Sl_errAck,   DIR = OUT,                                BUS
= SOPB
PORT sln_retry   = Sl_retry,    DIR = OUT,                                BUS
= SOPB
PORT sln_toutsup = Sl_toutSup,  DIR = OUT,                                BUS
= SOPB
PORT sln_xferack = Sl_xferAck,  DIR = OUT,                                BUS
= SOPB

# Ports connecting the module to the SRAM
PORT SRAM_CE     = "",          DIR = OUT
PORT SRAM_OE     = "",          DIR = OUT
PORT SRAM_WE     = "",          DIR = OUT
PORT SRAM_UB     = "",          DIR = OUT
PORT SRAM_LB     = "",          DIR = OUT
PORT PB_A        = "",          DIR = OUT, VEC=[17:0], IOB_STATE=BUF
PORT PB_D        = "",          DIR = INOUT, VEC=[15:0], THREE_STATE=FALSE,
IOB_STATE=BUF
```

```
PORT au_cs         = "",          DIR = OUT
PORT au_mclk       = "",          DIR = OUT
PORT au_lrclk      = "",          DIR = OUT
PORT au_bclk       = "",          DIR = OUT
PORT au_sdti       = "",          DIR = OUT
PORT au_sdto0      = "",          DIR = IN

END
```

**GDSP pao file**

```
#####################################################################
#
# gDSP pao file
#
#####################################################################

lib gDSP_v1_00_a gDSP
lib gDSP_v1_00_a ak4565
lib gDSP_v1_00_a distortion
lib gDSP_v1_00_a SRAM_FX
```

**ak4565.vhd**

```
-- JunHao Ip
-- Jason Cardillo
-- Chih-Chieh Lin
--------------------
--FPGA rule, don't mix clock with signal
--use separate counter for each clock
--always work on either up latch or down latch

--------------ak4565 Audio Interface----------------------------------------
-------
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity ak4565 is
  port (
    clk      : in  std_logic;
    rst      : in  std_logic;
    mclk     : out std_logic;
    bclk     : out std_logic;
    lrclk    : out std_logic;
    sdti     : out std_logic;
    sdto0    : in  std_logic;
    csn      : out std_logic;
    cclk     : out std_logic;
    cdti     : out std_logic;

    adcdone : out std_logic;
    dacload : out std_logic;
    adc_dtout : out std_logic_vector(15 downto 0);  --data out to fpga
    dac_dtin  : in  std_logic_vector(15 downto 0);  --parallelload data from
fpga
    c_datain : in std_logic_vector(15 downto 0);  --parallelload control from
fpga
    c_wr     : in std_logic;
    c_done   : out std_logic
    );
end ak4565;

architecture Behavioral of ak4565 is

  signal fs256clk : std_logic;  --12.5Mhz = 256fs
  signal fs64clk, sbclk, snbclk : std_logic;  --1.5625Mhz = 32fs
  signal slrclk: std_logic;  --48.8Khz = fs
  --signal scclk : std_logic;  --3.125Mhz

  signal clkcount: std_logic_vector(1 downto 0):="00";
  signal adcount,dacount: std_logic_vector(7 downto 0):="00000000";
  signal ccount : std_logic_vector(3 downto 0):="0000";
  signal ad_capture, da_stream : std_logic;
  signal s_csn, sdone : std_logic;
  signal s_adcdone, s_dacload: std_logic;
```

```
  signal adc_data16: std_logic_vector(15 downto 0) := x"0000";
  signal in_shift_reg,out_shift_reg,c_shift_reg: std_logic_vector(15 downto
0) := x"0000";
  signal s_fpga2audio, s_fpga2audcntrl : std_logic;

----output to signal mapping-------------------------
begin
    mclk  <=not fs256clk;
    bclk  <=sbclk;
    lrclk <=slrclk;

    sdti  <=s_fpga2audio;

    csn   <=s_csn;
    cclk  <=not fs64clk;
    cdti  <=s_fpga2audcntrl;

    adcdone <= s_adcdone;
    dacload <= s_dacload;

    adc_dtout(15 downto 0)<=adc_data16(15 downto 0);
    c_done  <= sdone;

----clocks and counters------------------------------
  clkdiv1 : process(clk,rst)
  begin
    if rst = '1' then
      clkcount <= "00";
    elsif clk'event and clk = '1' then
      clkcount <= clkcount + 1;

      -- phase doesn't matter? ak4565 spec page 11
      fs256clk<=clkcount(1);  --sync with lrclk?
    end if;

  end process clkdiv1;

  clkdiv2 : process(fs256clk,rst)
  begin
    if rst = '1' then
      adcount <= "00000000";
      dacount <= "00000100";
    elsif fs256clk'event and fs256clk = '1' then
      adcount <= adcount+1;
      dacount <= dacount+1;

        fs64clk<=adcount(1);
      sbclk<=adcount(2);  --clocks
      snbclk<=dacount(2);
      slrclk<=adcount(7);

        ad_capture <=adcount(7);  --regular signal
        da_stream <=dacount(7);
    end if;
  end process clkdiv2;
```

```
------grab audio data---------------------------------
  get_audio : process(sbclk,rst)
  begin
    if rst = '1' then
      in_shift_reg <= X"0000";
      adc_data16 <= X"0000";
    elsif sbclk'event and sbclk = '1' then
      if ad_capture = '0' then
          in_shift_reg(15)<=in_shift_reg(14);
          in_shift_reg(14)<=in_shift_reg(13);
          in_shift_reg(13)<=in_shift_reg(12);
          in_shift_reg(12)<=in_shift_reg(11);
          in_shift_reg(11)<=in_shift_reg(10);
          in_shift_reg(10)<=in_shift_reg(9);
          in_shift_reg(9)<=in_shift_reg(8);
          in_shift_reg(8)<=in_shift_reg(7);
          in_shift_reg(7)<=in_shift_reg(6);
          in_shift_reg(6)<=in_shift_reg(5);
          in_shift_reg(5)<=in_shift_reg(4);
          in_shift_reg(4)<=in_shift_reg(3);
          in_shift_reg(3)<=in_shift_reg(2);
          in_shift_reg(2)<=in_shift_reg(1);
          in_shift_reg(1)<=in_shift_reg(0);
          in_shift_reg(0)<=sdto0;
      else
          adc_data16(15 downto 0)<= in_shift_reg(15 downto 0);
      end if;
    end if;
  end process get_audio;

  get_audio2 : process(sbclk,rst)
  begin
    if rst = '1' then
      s_adcdone<='0';
    elsif sbclk'event and sbclk = '1' then
      if ad_capture = '0' then
        s_adcdone<='0';
      else
        s_adcdone<='1';
      end if;
    end if;
 end process get_audio2;


--------push data to output port-----------------------

  output_audiostream : process(snbclk,rst)
  begin
    if rst ='1' then
      out_shift_reg(15 downto 0)<=X"0000";
      s_fpga2audio<='0';
    elsif snbclk'event and snbclk = '1' then
      if da_stream='0' then
          out_shift_reg(15)<=out_shift_reg(14);
          out_shift_reg(14)<=out_shift_reg(13);
          out_shift_reg(13)<=out_shift_reg(12);
          out_shift_reg(12)<=out_shift_reg(11);
```

```
          out_shift_reg(11)<=out_shift_reg(10);
          out_shift_reg(10)<=out_shift_reg(9);
          out_shift_reg(9)<=out_shift_reg(8);
          out_shift_reg(8)<=out_shift_reg(7);
          out_shift_reg(7)<=out_shift_reg(6);
          out_shift_reg(6)<=out_shift_reg(5);
          out_shift_reg(5)<=out_shift_reg(4);
          out_shift_reg(4)<=out_shift_reg(3);
          out_shift_reg(3)<=out_shift_reg(2);
          out_shift_reg(2)<=out_shift_reg(1);
          out_shift_reg(1)<=out_shift_reg(0);
          out_shift_reg(0)<='0';
          s_fpga2audio <= out_shift_reg(15);
      else
          out_shift_reg(15 downto 0)<= dac_dtin(15 downto 0);
          s_fpga2audio<='0';
      end if;
    end if;
  end process output_audiostream;

 output_audiostream2 : process(snbclk,rst)
  begin
    if rst = '1' then
      s_dacload<='0';
    elsif snbclk'event and snbclk = '1' then
      if da_stream = '0' then
        s_dacload<='0';
      else
        s_dacload<='1';
      end if;
    end if;
 end process output_audiostream2;



----- audio port initialization -------------------------
process (fs64clk,c_wr)
   begin
   if c_wr='1' then
            ccount<="0000";
            sdone<='0';
            s_csn<='1';
   elsif fs64clk'event and fs64clk='1' then
            if sdone<='0' then
                  ccount<=ccount+1;
                  s_csn<='0';
            end if;

            if ccount = "1111" then
                  sdone<='1';
            end if;

            if sdone='1' then
                  s_csn<='1';
            end if;
   end if;
end process;
```

```
process (fs64clk,c_wr,c_datain)
   begin
   if c_wr='1' then
            c_shift_reg(15 downto 0) <= c_datain(15 downto 0);
      s_fpga2audcntrl<='0';
   elsif fs64clk'event and fs64clk='1' then
            if sdone ='0' then
                  c_shift_reg(15) <= c_shift_reg(14);
                  c_shift_reg(14) <= c_shift_reg(13);
                  c_shift_reg(13) <= c_shift_reg(12);
                  c_shift_reg(12) <= c_shift_reg(11);
                  c_shift_reg(11) <= c_shift_reg(10);
                  c_shift_reg(10) <= c_shift_reg(9);
                  c_shift_reg(9) <= c_shift_reg(8);
                  c_shift_reg(8) <= c_shift_reg(7);
                  c_shift_reg(7) <= c_shift_reg(6);
                  c_shift_reg(6) <= c_shift_reg(5);
                  c_shift_reg(5) <= c_shift_reg(4);
                  c_shift_reg(4) <= c_shift_reg(3);
                  c_shift_reg(3) <= c_shift_reg(2);
                  c_shift_reg(2) <= c_shift_reg(1);
                  c_shift_reg(1) <= c_shift_reg(0);
        c_shift_reg(0) <= '0';
                  s_fpga2audcntrl <= c_shift_reg(15);
            end if;
   end if;
end process;



end Behavioral;
```

**distortion.vhd**

```
library ieee;
use ieee.std_logic_1164.all;

-- distortion is taking the absolute value of the data.

entity distortion is
  port(clk: in STD_LOGIC;
    rst: in std_logic;
    din:in std_logic_vector(15 downto 0);
    dout:out std_logic_vector(15 downto 0));
end distortion;

architecture behavioral of distortion is
  signal temp: std_logic_vector(15 downto 0):=x"0000";
begin

  process(clk,rst)
    begin
      if rst = '1' then
          temp<=x"0000";
      elsif clk='1' and clk'event then
          if din(15)='1' then
            temp(15)<=not(din(15));
            temp(14)<=not(din(14));
            temp(13)<=not(din(13));
            temp(12)<=not(din(12));
            temp(11)<=not(din(11));
            temp(10)<=not(din(10));
            temp(9)<=not(din(9));
            temp(8)<=not(din(8));
            temp(7)<=not(din(7));
            temp(6)<=not(din(6));
            temp(5)<=not(din(5));
            temp(4)<=not(din(4));
            temp(3)<=not(din(3));
            temp(2)<=not(din(2));
            temp(1)<=not(din(1));
            temp(0)<=not(din(0));
          else
            temp(15 downto 0)<=din(15 downto 0);
          end if;
      end if;
  end process;

  dout<=temp(15 downto 0);
end behavioral;
```

**gDSP.vhd**

```
------Guitar Effector-------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity gDSP is

  generic (
    C_OPB_AWIDTH : integer                   := 32;
    C_OPB_DWIDTH : integer                   := 32;
    C_BASEADDR   : std_logic_vector(0 to 31) := X"00000000";
    C_HIGHADDR   : std_logic_vector(0 to 31) := X"FFFFFFFF"
    );

  port (
    OPB_Clk     : in  std_logic;
    OPB_Rst     : in  std_logic;
    OPB_ABus    : in  std_logic_vector(0 to C_OPB_AWIDTH-1);  --(31:0)
    OPB_BE      : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);--(3:0)
    OPB_DBus    : in  std_logic_vector(0 to C_OPB_DWIDTH-1);  --(31:0)
    OPB_RNW     : in  std_logic;
    OPB_select  : in  std_logic;
    OPB_seqAddr : in  std_logic;        -- Sequential Address
    Sln_DBus    : out std_logic_vector(0 to C_OPB_DWIDTH-1);  --(31:0)
    Sln_errAck  : out std_logic;        -- (unused)
    Sln_retry   : out std_logic;        -- (unused)
    Sln_toutSup : out std_logic;        -- Timeout suppress
    Sln_xferAck : out std_logic;        -- Transfer acknowledge

    SRAM_CE     : out std_logic;        --sram chip enable (active low)
    SRAM_OE     : out std_logic;        --sram output enable (active low)
    SRAM_WE     : out std_logic;        --sram write enable (active low)
    SRAM_UB     : out std_logic;        --sram enable upper-byte(active low)
    SRAM_LB     : out std_logic;        --sram enable lower-byte(active low)
    PB_A        : out std_logic_vector(17 downto 0);  --sram 18 bit address
    PB_D        : inout std_ulogic_vector(15 downto 0);  --sram 16 bit data


    au_mclk     : out std_logic;
    au_lrclk    : out std_logic;
    au_bclk     : out std_logic;
    au_sdti     : out std_logic;
    au_sdto0    : in std_logic;
    au_cs       : out std_logic
    );
end gDSP;

architecture Behavioral of gDSP is

  component wr_audio_control
    port (
    clk_in: in std_logic;               -->200ns clock
    init: in std_logic;                 --pulse input
```

```
    done: out std_logic;
    D_in: in std_logic_vector(15 downto 0);        --control data
    cntrl_out: out std_logic);
  end component;

  component OBUF_F_24
    port (
      O : out STD_ULOGIC;               -- the pin
      I : in  STD_ULOGIC);              -- signal to pin
  end component;

  component IOBUF_F_24
    port (
      O : out STD_ULOGIC;               -- signal from pin
      IO : inout STD_ULOGIC;            -- the pin
      I : in  STD_ULOGIC;               -- signal to pin
      T : in STD_ULOGIC);               -- 1-drive IO with I
  end component;

  component ak4565
    port (
      clk      : in  std_logic;
      rst      : in  std_logic;
      mclk     : out std_logic;
      bclk     : out std_logic;
      lrclk    : out std_logic;
      sdti     : out std_logic;
      sdto0    : in  std_logic;
      csn      : out std_logic;
      cclk     : out std_logic;
      cdti     : out std_logic;

      adcdone : out std_logic;  --AD indicator
      dacload : out std_logic;  --DA indicator
      adc_dtout : out std_logic_vector(15 downto 0);  --data out to fpga
      dac_dtin  : in  std_logic_vector(15 downto 0);  --parallelload data
from fpga
      c_datain : in std_logic_vector(15 downto 0);  --parallelload control
from fpga
      c_wr     : in std_logic :='0';  --write to control
      c_done   : out std_logic  --write control done
      );
  end component;

  component distortion
    port(clk: in STD_LOGIC;
    rst: in STD_LOGIC;
    din:in std_logic_vector(15 downto 0);
    dout:out std_logic_vector(15 downto 0)
    );
  end component;

  component SRAM_FX
  port (
            clk:in std_logic; --50Mhz system clk
            rst:in std_logic;
```

```
            feedback: in std_logic;
            effect_sel: in std_logic;
            delay_len: in std_logic_vector(17 downto 0);  --must be in 2's
comp negative, this reduce unnessessary computation in vhdl
            trigger: in std_logic;
            done: out std_logic;

            snd_in: in std_logic_vector(15 downto 0);
            snd_out:out std_logic_vector(15 downto 0);

            sram_rnw: out std_logic;
            sram_addr: out std_logic_vector(17 downto 0);
            sram_dt_wr: out std_logic_vector(15 downto 0);
            sram_dt_rd: in std_logic_vector(15 downto 0)
            );
  end component;


  signal ABus: std_logic_vector(17 downto 0);
  signal pbDIn: std_logic_vector(15 downto 0);

  signal tri_state : std_logic;

  signal mclk : std_logic;  --12.5Mhz
  signal bclk : std_logic;  --3.125Mhz
  signal lrclk: std_logic;  --48.8Khz
  signal cclk: std_logic;

  signal sdti: std_logic;
  signal latch16bit,dist_out,echo_out,delay_out,snd_output:
std_logic_vector(15 downto 0) := x"0000";

  signal adcdone : std_logic;
  signal dacload : std_logic;
  signal adc_dtout : std_logic_vector(15 downto 0);  --data out to fpga
  signal dac_dtin  : std_logic_vector(15 downto 0) :=x"0000";  --parallelload
data from fpga
  signal c_datain : std_logic_vector(15 downto 0) :="1110000011100111";  --
parallelload control from fpga
  signal csn : std_logic;
  signal c_wr     : std_logic;
  signal c_done   : std_logic;

  signal feedback, effect_sel: std_logic;
  signal en_echoproc,trigger,echobegin: std_logic;
  signal echodone: std_logic;
 -- signal echoout: std_logic_vector(15 downto 0);
  signal sram_rnw: std_logic;
  signal sram_addr: std_logic_vector(17 downto 0);
  signal sram_dt_wr: std_logic_vector(15 downto 0);
  signal sram_dt_rd: std_logic_vector(15 downto 0);

  signal initcnt:std_logic_vector(15 downto 0):=X"0000";  --has to wait >
4128/fs=90ms for initialization
  constant pre_init : std_logic_vector(1 downto 0):="00";
  constant c_wr_wait: std_logic_vector(1 downto 0):="01";
  constant wait_done: std_logic_vector(1 downto 0):="10";
```

```
  constant norm : std_logic_vector(1 downto 0):="11";
  signal c_state, n_state :  std_logic_vector(1 downto 0):="00";
  signal c_serial_data: std_logic;


----------------------------OPB SIGNALS
---------------------------PART FOR C PROGRAM CHOICE OF EFFECT _______________
  signal RNW : std_logic;--read not write
  --signal ABus : std_logic_vector(0 to 15);
  signal chip_select : std_logic;
  signal output_enable : std_logic;
  signal WE, RST : std_logic;

  -- Critical: Sln_xferAck is generated directly from state bit 0!
  constant STATE_BITS : integer := 3;
  constant Idle     : std_logic_vector(0 to STATE_BITS-1) := "000";
  constant Selected : std_logic_vector(0 to STATE_BITS-1) := "001";
  constant Read     : std_logic_vector(0 to STATE_BITS-1) := "011";
  constant Xfer     : std_logic_vector(0 to STATE_BITS-1) := "111";

  signal present_state, next_state : std_logic_vector(0 to STATE_BITS-1);

  signal DI: std_logic_vector(0 to 15);
  signal fx_sel:std_logic_vector(0 to 2);
  --------------------------END OPB SIGNALS---------------------------------
--

-------------------------------------------------------------------------------
-----
-------------------------------------------------------------------------------
-----
-----------BEGIN ARCHITECTURE-----------------------------------------------
------
-------------------------------------------------------------------------------
------
-------------------------------------------------------------------------------
------
begin
------Audio Interface Signals and Buffering---------------------------------
----------------

    au_mclk<=mclk;  -----VERY IMPORTANT---- mclk must be sync with LRCLK
    au_bclk<=bclk;
    au_lrclk<=lrclk;
    au_sdti <=sdti;
    au_cs<=csn;

------SRAM pin assignment---------------------------------------------------
-------------------
    SRAM_CE <='0' when c_state=norm else  -- *** must enable SRAM!!!
              '1';
    SRAM_OE <='0';
    SRAM_WE <=sram_rnw;
    SRAM_UB <='0';
    SRAM_LB <='0';

    gen1: for m in 0 to 17 generate
```

```
    sramAddrpin:OBUF_F_24 port map (O=>PB_A(m), I=>ABus(m));
    end generate;

    gen2: for m in 0 to 15 generate
    sramDatapin:IOBUF_F_24 port map (O=>sram_dt_rd(m),IO=>PB_D(m),
I=>pbDIn(m), T=>tri_state );
    end generate;
    tri_state <=sram_rnw when c_state=norm else
                '0'; --always writing


    ABus<=sram_addr(17 downto 0);
    pbDIn(15 downto 2) <=sram_dt_wr(15 downto 2) when c_state=norm else
                     (others => '0');
    pbDIn(0)<=sram_dt_wr(0) when c_state=norm else
              cclk;
    pbDIn(1)<=sram_dt_wr(1) when c_state=norm else
              c_serial_data;

-----------------------------------------------------------------

    ak:ak4565 port map(clk =>OPB_Clk,
      rst      =>OPB_Rst,
      mclk     =>mclk,
      bclk     =>bclk,
      lrclk    =>lrclk,
      sdti     =>sdti,
      sdto0    =>au_sdto0,
      csn      =>csn,
      cclk     =>cclk,
      cdti     =>c_serial_data,
      adcdone =>adcdone,
      dacload =>dacload,
      adc_dtout => adc_dtout,
      dac_dtin  => dac_dtin,
      c_datain =>c_datain,
      c_wr    =>c_wr,
      c_done  =>c_done);

      ak_distort:distortion port map(
      clk=>OPB_Clk,
      rst =>OPB_Rst,
      din => latch16bit,
      dout=> dist_out
      );

    echobegin<=trigger when c_state = norm else
                '0';
  delaygen:SRAM_FX port map(
            clk=>OPB_Clk,
            rst=>OPB_Rst,
            feedback=>feedback,
            effect_sel=>effect_sel,
            delay_len=>"0001011101110000000",
            trigger=>echobegin,
            done=>echodone,
            snd_in=>latch16bit,
```

```
            snd_out=>echo_out,
            sram_rnw=>sram_rnw,
            sram_addr=>sram_addr,
            sram_dt_wr=>sram_dt_wr,
            sram_dt_rd=>sram_dt_rd
            );

   ---latch output data----------------------
   process (OPB_Clk,OPB_Rst)
   begin
     if OPB_Rst = '1' then
       latch16bit<=x"0000";
       dac_dtin<=x"0000";
     elsif OPB_clk'event and OPB_clk ='1' then
       if adcdone = '1' and dacload = '1' then
         latch16bit(15 downto 0)<=adc_dtout(15 downto 0);
         dac_dtin(15 downto 0)<=snd_output(15 downto 0);
       end if;
     end if;
   end process;

   --- generate effect trigger-----------------
   process (OPB_Clk,OPB_Rst)
   begin
     if OPB_Rst = '1' then
       trigger<='0';
       en_echoproc<='0';
     elsif OPB_clk'event and OPB_clk ='1' then
       if adcdone = '1' and dacload = '1' then
         if echodone='0' and en_echoproc='1' then
           trigger<='1';
         else
           en_echoproc<='0';
           trigger<='0';
         end if;
       else
         en_echoproc<='1';
       end if;
     end if;
   end process;


----control initialization-------
--op0 op1 op2 a0 a1 a2 a3 a4 d0 d1 d2 d3 d4 d5 d6 d7
c_datain<=   "1110000000100000";

process (bclk, OPB_Rst)
begin
  if OPB_Rst = '1' then
     c_state <= pre_init;
  elsif bclk'event and bclk='1' then
     c_state<=n_state;
  end if;
end process;

process (lrclk, OPB_Rst)
begin
```

```
   if OPB_Rst = '1' then
      initcnt <= X"0000";
   elsif lrclk'event and lrclk='1' then
      initcnt <= initcnt + 1;
   end if;
 end process;

process (initcnt(15),c_state,c_done)
begin
  c_wr<='0';
  case c_state is
  when pre_init =>
  --wait for initcnt
    if initcnt(15) = '1' then
      n_state<=c_wr_wait;
    else
      n_state<=pre_init;
    end if;
  when c_wr_wait =>
  --hold c_wr for 1 cycle
      c_wr<='1';
      n_state<=wait_done;
  when wait_done =>
      if c_done='1' then
        --normal operation
        n_state<=norm;
      else
        n_state<=wait_done;
      end if;
  when norm =>
    n_state<=norm;

  when others =>
    n_state<=pre_init;
  end case;
end process;


register_opb_inputs: process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      DI <= (others => '0');
      --ABus <= (others => '0');
      RNW <= '0';
    elsif OPB_Clk'event and OPB_Clk = '1' then
      DI <= OPB_DBus(0 to 15);
      --ABus <= OPB_ABus(C_OPB_AWIDTH-3-(RAM_AWIDTH-1) to C_OPB_AWIDTH-3);
      RNW <= OPB_RNW;
    end if;
  end process register_opb_inputs;

register_opb_outputs: process (OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      Sln_DBus(0 to 15) <= (others => '0');
    elsif OPB_Clk'event and OPB_Clk = '1' then
     if output_enable = '1' then
```

```
        Sln_DBus(0 to 15) <= fx_sel &"0000000000000"; --test purposes for now
      else
        Sln_DBus(0 to 15) <= (others => '0');
      end if;
     end if;
  end process register_opb_outputs;

  -- Unused outputs
  Sln_errAck  <= '0';
  Sln_retry   <= '0';
  Sln_toutSup <= '0';
  Sln_DBus(16 to 31) <= (others => '0');

  chip_select <=
    '1' when OPB_select = '1' and OPB_ABus(0 to 31) = x"FEFF1001"
     else '0';

--latch accumulator --for testing
another : process(OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      fx_sel<="000";
    elsif OPB_Clk'event and OPB_Clk = '1' then
      if WE= '1' then
         fx_sel <= DI(0 to 2);
      end if;
    end if;
  end process another;


  -- Sequential part of the FSM
  fsm_seq : process(OPB_Clk, OPB_Rst)
  begin
    if OPB_Rst = '1' then
      present_state <= Idle;
    elsif OPB_Clk'event and OPB_Clk = '1' then
      present_state <= next_state;
    end if;
  end process fsm_seq;

  -- Combinational part of the FSM
  fsm_comb : process(OPB_Rst, present_state, chip_select, OPB_Select, RNW)
  begin
    RST <= '1';                          -- Default values
    WE <= '0';
    output_enable <= '0';
    if OPB_RST = '1' then
      next_state <= Idle;
    else
      case present_state is
        when Idle =>
          if chip_select = '1' then
            next_state <= Selected;
          else
            next_state <= Idle;
          end if;
```

```
        when Selected =>
          if OPB_Select = '1' then
            if RNW = '1' then
              RST <= '0';
              next_state <= Read;
            else
              WE <= '1';
              next_state <= Xfer;
            end if;
          else
            next_state <= Idle;
          end if;

        when Read =>
          if OPB_Select = '1' then
            output_enable <= '1';
            next_state <= Xfer;
          else
            next_state <= Idle;
          end if;

        -- State encoding is critical here: xfer must only be true here
        when Xfer =>
          next_state <= Idle;

        when others =>
          next_state <= Idle;
      end case;
    end if;
  end process fsm_comb;

  Sln_xferAck <= present_state(0);


snd_output<=latch16bit(15 downto 0) when fx_sel="000" else
          dist_out(15 downto 0) when fx_sel="001" else
          echo_out(15 downto 0);

feedback<=fx_sel(1);
effect_sel<=fx_sel(2);

end Behavioral;
```

**SRAM_FX.vhd**

```
library ieee;
use IEEE.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

-- Since effects are involved with delays, we use SRAM to RW delay data.
-- We exploit the ideas of circular buffer to store data.
-- Here, it is performed with counters.

entity SRAM_FX is
  port (
            clk:in std_logic; --50Mhz system clk
            rst:in std_logic;

            feedback: in std_logic;
            effect_sel: in std_logic;
            delay_len: in std_logic_vector(17 downto 0);  --must be in 2's
comp negative, this reduce unnessessary computation in vhdl
            trigger: in std_logic;
            done: out std_logic;

            snd_in: in std_logic_vector(15 downto 0);
            snd_out:out std_logic_vector(15 downto 0);

    --SRAM should always assume to be enabled
    --and it is controlled by RNW and Address
            sram_rnw: out std_logic;
            sram_addr: out std_logic_vector(17 downto 0);
            sram_dt_wr: out std_logic_vector(15 downto 0);
            sram_dt_rd: in std_logic_vector(15 downto 0)

            );
end SRAM_FX;

architecture Behavioral of SRAM_FX is
  signal s_done: std_logic;
  signal dt_in: std_logic_vector(15 downto 0);  --input latch
  signal dt_out: std_logic_vector(15 downto 0);  --output latch
  signal sr_rnw, readram_dt: std_logic;
  signal sr_addr: std_logic_vector(17 downto 0);  --address latch
  signal sr_dt_rd: std_logic_vector(15 downto 0); --sram latch

  signal c_addr: std_logic_vector(17 downto 0):="000000000000000000";
  signal delay_addr: std_logic_vector(17 downto 0);--:="111010001001000000";

  signal dt_mux: std_logic;
  signal mux_dt,temp_sum,accum: std_logic_vector(15 downto 0);

  signal up:std_logic:='1';
  signal lrcnt: std_logic_vector(9 downto 0);
  signal lrclk:std_logic;
  signal cnt: std_logic_vector(6 downto 0):="0000000";
  signal d_cnt: std_logic_vector(11 downto 0):=x"000";
  signal fxdelay:std_logic_vector(17 downto 0):="000000000000000000";
```

```
  signal flgclk:std_logic;

  constant idling : std_logic_vector(1 downto 0):="00";
  constant s1: std_logic_vector(1 downto 0):="01";
  constant s2 : std_logic_vector(1 downto 0):="10";
  --constant s3 : std_logic_vector(1 downto 0):="11";
  signal c_state,n_state: std_logic_vector(1 downto 0) :="00";

begin
            done<=s_done;
            snd_out<=dt_out(15 downto 0);
            sram_rnw<=sr_rnw;
    sram_addr(17 downto 0)<=sr_addr(17 downto 0);

    sram_dt_wr<= dt_in(15 downto 0) when feedback='0' else
                 temp_sum(15 downto 0);


  process(clk,rst)
      begin
            if rst='1' then
      lrcnt<=(others=>'0');
    elsif clk'event and clk='1' then
      lrcnt<=lrcnt+1;
      lrclk<=lrcnt(9);
    end if;
  end process;

  process(lrclk,rst)
      begin
            if rst='1' then
      cnt<="0000000";
    elsif lrclk'event and lrclk='1' then
      cnt<=cnt+1;
      flgclk<=cnt(6);
    end if;
  end process;

  --variation from 0 to .003sec, with rate=128/48000 per step
  process(flgclk,rst)
  begin
    if rst='1' then
      d_cnt<="000000000000";
      up<='1';
    elsif flgclk'event and flgclk='1' then
      if up='1' then
        d_cnt<=d_cnt+1;
        if d_cnt="000010001111" then
          up<='0';
        end if;
      else
        d_cnt<=d_cnt-1;
        if d_cnt="000000000001" then
          up<='1';
        end if;
      end if;
    end if;
```

```
  end process;

  fxdelay<= delay_len when effect_sel='0' else
            "000000" & d_cnt;

----latch sound input------------
--data input is divided by 2 to avoid saturation
--at first we divide data ip by 4 but get nothing!
      process(clk,rst)
      begin
            if rst='1' then
                  dt_in<=X"0000";
            elsif clk'event and clk='1' then
      dt_in(15)<=snd_in(15);
      dt_in(14)<=snd_in(15);
      dt_in(13)<=snd_in(14);
      dt_in(12)<=snd_in(13);
      dt_in(11)<=snd_in(12);
      dt_in(10)<=snd_in(11);
      dt_in(9)<=snd_in(10);
      dt_in(8)<=snd_in(9);
      dt_in(7)<=snd_in(8);
      dt_in(6)<=snd_in(7);
      dt_in(5)<=snd_in(6);
      dt_in(4)<=snd_in(5);
      dt_in(3)<=snd_in(4);
      dt_in(2)<=snd_in(3);
      dt_in(1)<=snd_in(2);
      dt_in(0)<=snd_in(1);
                  --dt_in(14 downto 0)<=snd_in(15 downto 1);
            end if;
      end process;

      --generate delay and current addresses
      process(clk,rst)
      begin
            if rst='1' then
      sr_addr<="0000000000000000";
            elsif clk'event and clk='1' then
      if c_state=s1 then
        sr_addr<=c_addr;
      else
        sr_addr<=delay_addr;
      end if;
            end if;
      end process;

    delay_addr<=c_addr-fxdelay;


      --latch sram output------------
  --data input is divided by 4 to avoid saturation
      process(clk,rst)
      begin
            if rst='1' then
                  sr_dt_rd<=X"0000";
            elsif clk'event and clk='1' then
```

```
    if readram_dt<='1' then
      sr_dt_rd(15)<=sram_dt_rd(15);
      sr_dt_rd(14)<=sram_dt_rd(15);
      sr_dt_rd(13)<=sram_dt_rd(14);
      sr_dt_rd(12)<=sram_dt_rd(13);
      sr_dt_rd(11)<=sram_dt_rd(12);
      sr_dt_rd(10)<=sram_dt_rd(11);
      sr_dt_rd(9)<=sram_dt_rd(10);
      sr_dt_rd(8)<=sram_dt_rd(9);
      sr_dt_rd(7)<=sram_dt_rd(8);
      sr_dt_rd(6)<=sram_dt_rd(7);
      sr_dt_rd(5)<=sram_dt_rd(6);
      sr_dt_rd(4)<=sram_dt_rd(5);
      sr_dt_rd(3)<=sram_dt_rd(4);
      sr_dt_rd(2)<=sram_dt_rd(3);
      sr_dt_rd(1)<=sram_dt_rd(2);
      sr_dt_rd(0)<=sram_dt_rd(1);
      --sr_dt_rd(14 downto 0)<=sram_dt_rd(15 downto 1);
    end if;
          end if;
    end process;

--fsm of delay process-----
    process(clk,rst)
    begin
          if rst='1' then
                c_state<=idling;
          elsif clk'event and clk='1' then
                c_state<=n_state;
          end if;
    end process;

    --state machine combinational logic
    FSM_Comb: process(rst,trigger,c_state)
    begin
  sr_rnw<='1';  --always reading
  s_done<='0';
  readram_dt<='0';

  if rst='1' then
    n_state <= idling;
  else
    case c_state is
    when idling =>
      if trigger='1' then
        readram_dt<='1';
        n_state<=s1;
      else
        n_state<=idling;
      end if;
    when s1 =>
      --wait for calculation to be done
      n_state<=s2;
    when s2 =>
      --store new data to SRAM
      sr_rnw<='0';  --write to sram
      s_done<='1';
```

```
        n_state<=idling;
      when others =>
        n_state<=idling;
      end case;
    end if;
      end process;


      --latch output-------
      process(clk, rst)
      begin
            if rst='1' then
                  dt_out<=X"0000";
            elsif clk'event and clk='1' then
      if c_state=s1 then
        dt_out<=temp_sum(15 downto 0);
      end if;
            end if;
      end process;

  --update current SRAM address-----------
      process(clk,rst)
      begin
      if rst='1' then
            c_addr<="0000000000000000";
      elsif clk'event and clk='1' then
        if c_state=s2 then
          c_addr<=c_addr+1;
        end if;
            end if;
      end process;


----delay calculation--------
--temp_sum is floating at c_state=2
  temp_sum<=dt_in + sr_dt_rd;


end Behavioral;
```