# DSP Dude: A DSP Audio Pre-Amplifier

*6.111 Project Proposal*

**Yanni Coroneos and Valentina Chamorro**

## Overview

The problem we would like to address with DSP Dude is the scenario where there are multiple speakers in a single room but each can optimally produce only one set of frequencies. We imagine DSP Dude as a quickly-reprogrammable digital signal processor that can filter out frequencies a speaker can't reproduce. By chaining multiple of these units together, we can allow every speaker to only produce the frequencies it is good at reproducing. The digital nature and reprogrammability is integral to our design because we intend for this system to be deployable on short notice and in very different speaker arrangements—as is the case in public spaces which host a variety of social events.

## Motivation: Digital Signal Processing

A valid question one might ask is why must these audio signals be digitally processed when there exist working analog solutions? Analog signal processors are large, difficult, and ineffective to produce because of the cost and space associated with continuous-time (CT) analog filters. A generally applicable analog signal processor will either need an array of CT filters, which takes up space, or a few CT filters and a complicated voltage multiplier in order to shift the center frequencies of the filters. This completed scheme still doesn't even solve the problem of arbitrary frequency response: what if the user needs a highly specific frequency response whose shape cannot be achieved by just a few linear combinations of CT filters? In the analog domain, this requires a completely custom circuit design—which is prohibitively expensive for just about everyone. There is also the issue of noise and unbalanced signals. Analog audio outputs on most consumer electronics are single-ended, which means that the negative voltage is also the reference ground. This can cause issues at high frequencies or high power because the current running through the ground wire will cause a voltage drop, or a back EMF in the case of a very long cable. When the ground is no longer 0 volts the reference is lost. In extreme cases this can cause a noticeable DC offset on the input to the amplifier which can result in clipping or DC current through the speaker. A digital signal processor with user re-programmable DSP functions could solve every issue previously mentioned while still remaining small in size. A system consisting of only an FPGA and a codec can take the place of all the analog electronics.

## Design

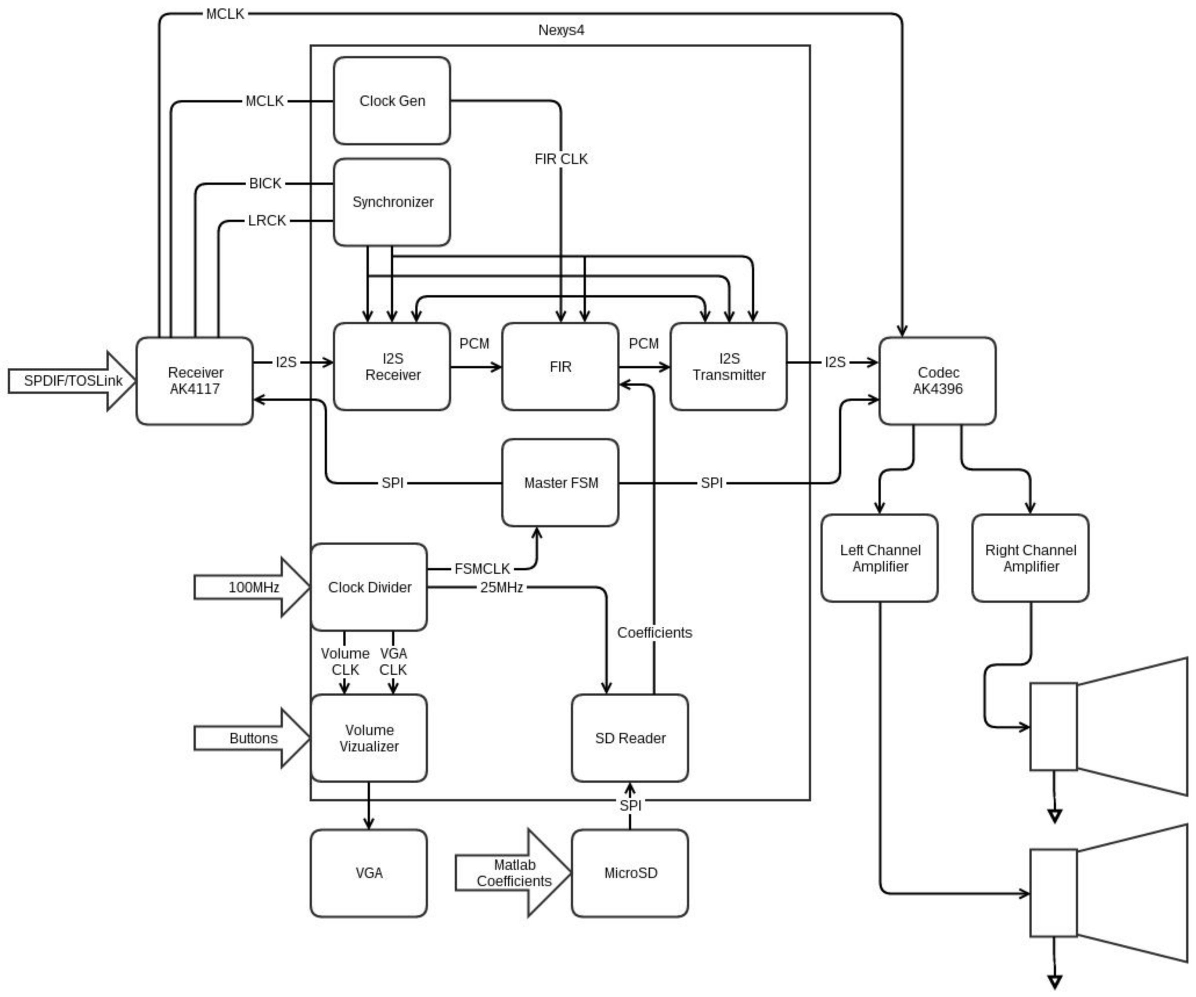


Figure 1. Block diagram of DSP Dude. Generally, sound, clocks, and button inputs come in on the left and filtered audio comes out on the right. On the bottom are the SDcard and VGA data.

DSP Dude lives on a an Artix7 XC7A-100t-CSG321 which is sitting on a Digilent Nexys4 development kit. Please refer to our attached schematic sheets. Ports JA and JB are used for input/output of the audio data signals, switches are used for selecting operating modes, and the push buttons are used for adjusting volume, which is displayed on a vga screen. FIR coefficients are loaded from the SDcard plugged into the Nexys4.

There are two external IC’s that DSP Dude interacts with to manipulate an audio stream: the AK4117 SPDIF receiver and the AK4396 dual-channel codec.

The AK4117 receiver IC decodes an SPDIF input audio stream into 24bit audio PCM data and outputs it over an I2S bus. Its internal registers must be configured over 16bit SPI in Mode 3 in order to set the desired sampling frequency and output resolution. The AK4117 has to be the I2S master so that way it can generate the I2S bit clock, left-right clock, and master clock. These clocks, but not the bit data itself, are directly passed onto the AK4396 audio codec over wires because it must operate in the same clock domain as the AK4117.

The AK4396 audio codec encodes a PCM I2S audio stream into dual-channel analog audio output. It takes as input an I2S audio bus and outputs signal-level balanced analog audio. Like the AK4117, its internal registers must be configured over 16bit SPI in Mode 3 in order to set the desired sampling frequency and input resolution. As previously said, the I2S clock signals are generated by the AK4117 and fed to the AK4396 over a wire.

The Nexys4 is responsible for transforming the digital audio bitstream according to the FIR filter and then outputting it to the AK4396. On a broad level, the AK4117 outputs I2S audio data, then the Nexys4 reads it and transforms it before outputting the transformed data to the AK4396. Needless to say, most of the work is done by the Nexys4.

Configuring both the AK4117 and AK4396 happens over a shared 16bit SPI bus. This is possible because both IC's have a chip-select line, which is driven active-low by the fpga depending on which IC data is intended for.

# Verilog Modules

## Clockgen

DSP Dude operates on two different clock domains: the 24.576MHz AK4117 falling-edge synchronous  master clock and the internal 100MHz rising-edge synchronous control system clocks. The AK4117 generates the master clock and a 3.72MHz bit rate clock from the 48KHz sample rate clock which is automatically set by detecting the sample rate of the incoming audio data. The digital signal processing clocks are derived from the I2S bus output of the AK4117. In order to perform fast DSP we had to synthesize a 98.304MHz clock from the 24.576MHz AK4117 master clock. The 98.302MHz clock is falling edge synchronous with the 24.576MHz clock in order to produce a clear output. We did this with the Vivado clock wizard module.

There are two internal configuration clock signals that are derived from the 100MHz internal oscillator: the 100Hz FSM clock and 9.6KHz SPI clock. The FSM clock is what the Master FSM uses for rising edges in order to advance the state machine. The 9.6KHz SPI clock is required for SPI communications to the AK4117 and AK4396 slave devices.

## Clockgen Complications

We initially did not realize that the I2S bus clocks were falling edge synchronous. We designed our clockgen module assuming they were rising-edge synchronous and this had the effect of offsetting our output I2S bit data signal by 90 degrees in phase compared to the clocks. This caused the most significant bit of 1 channel to be discarded, making one speaker sound much softer compared to another.

## Master FSM (Yanni Coroneos)

On start-up, the AK4117 must be configured before the AK4396 because the AK4117 provides the AK4396 with its master clock. Additionally, volume commands must not be sent to the AK4396 until it has been properly configured or else it will ignore them. A “Master FSM” module was written to execute these events in the right order. The clock used for state transitions was set to be at 100Hz so that states waiting for an SPI transmission to finish could be removed. A simplified version of the state transition diagram is reproduced in Figure 2.

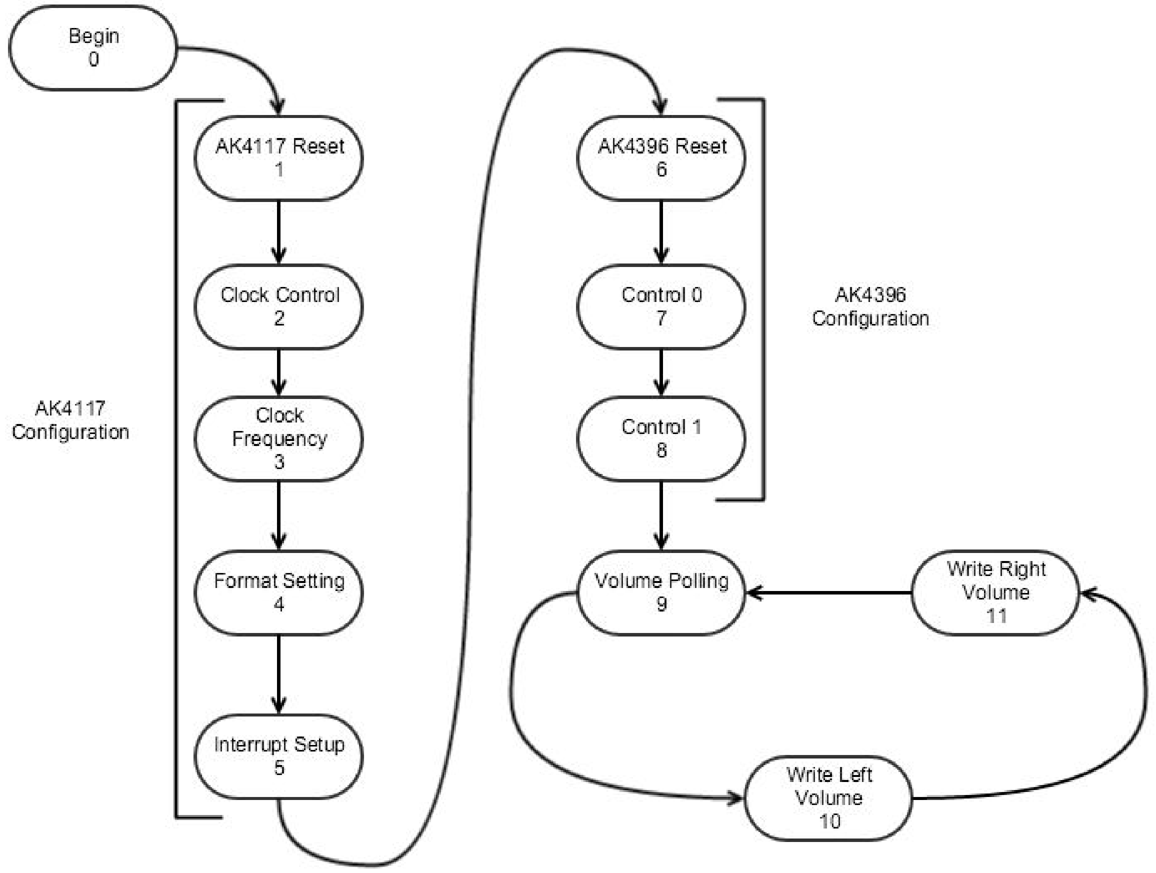


Figure 2. Simplified diagram of the Master FSM which configures the receiver and codec.

## Master FSM Complications (Yanni Coroneos)

We initially ran our Master FSM at 100MHz, the maximum clock rate of the system. This proved hopelessly difficult to manage because we had to constantly poll the SPI module to see if it was done transmitting, adding many redundant states to our state machine and complicating the entire mechanism. Reducing the clock rate to 100Hz means that, in between the 100Hz clock period, an entire SPI transmission can finish. This means we don’t have to poll the SPI module to see if it has finished transmitting.

## SPI Transmitter (Yanni Coroneos)

In order for the AK4117 and the AK4396 to work correctly, their internal registers must be configured to operate at the proper audio sampling frequency. This is done by our SPI module which can produce a 16bit SPI signal in Mode 3 operation. A single module is instantiated for both IC's and we time multiplex the SPI bus through use of the chip select lines.

## SPI Transmitter Complications (Yanni Coroneos)

We initially instantiated two SPI Transmitter modules, one for each IC. Since configuring both IC's simultaneously is impossible due to startup constraints mentioned earlier, the complicated parallel scheme was abandoned in favor of our current time-division scheme which works great.

## I2S Receiver (Valentina Chamorro)

DSP Dude must decode the I2S audio stream in order to be useful. As shown in Figure 3,, I2S consists of a three-wire bus: left-right clock, bit clock, and bit data. All clocks are falling-edge synchronous. 1 bit of audio data is shifted in on every falling edge of bit clock. When left-right clock is low, input data belongs to the left channel. When left-right clock is high, input data belongs to the right channel. There are 32 bit clocks per half-cycle of the left-right clock but only 24bits of audio data. The solution is to shift in all 32 bits but then discard the lower 8 least-significant bits because they are garbage. Thus at every period of left-right clock there are two new samples of PCM audio data on the output of the I2S receiver module.

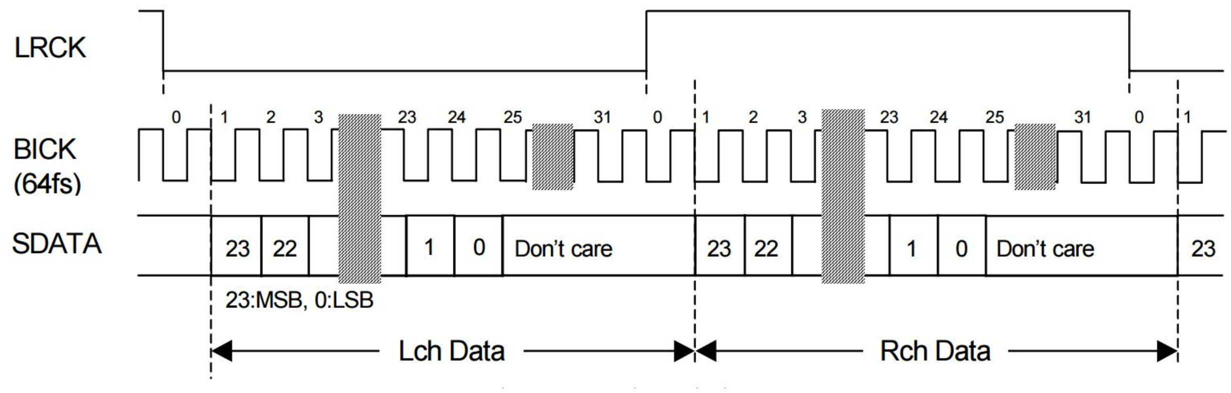


Figure 3. I2S protocol

## I2S Receiver Complications (Valentina Chamorro)

We could not edge trigger on the bit clock like a normal I2S reception mechanism operates. The Nexys4 only exposed 1 single-ended clock-capable input pin on the JB header. We chose to input the AK4117 master clock into this pin because it is synchronous with all the other I2S clocks and faster. The bit clock was given to a regular GPIO pin that was not capable of edge detection. In order to effectively sample at the falling edge of the bit-clock we generate a ready signal from the combination of the input master clock and the input bit clock. At every falling edge of the master clock, the bit clock is also sampled, and, if the bit clock experienced a falling-edge transition, a ready signal was asserted. This ready signal was used in the I2S receiver module to indirectly sample the bit clock and recover the PCM audio data.

## I2S Transmitter (Valentina Chamorro)

DSP Dude must output an I2S audio stream in order to be useful. The implementation of the I2S transmitter module is exactly the opposite of how the I2S receiver works. Instead of shifting *in* 1 bit of audio data on every bit clock edge, it shifts *out* 1 bit of audio on every bit clock edge. In fact, the code for the two modules is identical aside from the shift direction. The audio data which is shifted out is the result of the FIR computation, which is discussed next.

## I2S Transmitter Complications (Valentina Chamorro)

The I2S transmitter suffered from the same initial problem as the I2S receiver: synchronicity on the rising edge instead of the falling edge. It was difficult initially discovering this problem because we didn’t know if the I2S transmitter or receiver was broken.

## FIR Module (Yanni Coroneos)

This is the most important part of DSP Dude. The FIR module performs a convolution of the incoming audio data with a set of pre-generated coefficients. In the time-domain, the FIR filter coefficients comprise a causal, BIBO stable system as shown below.

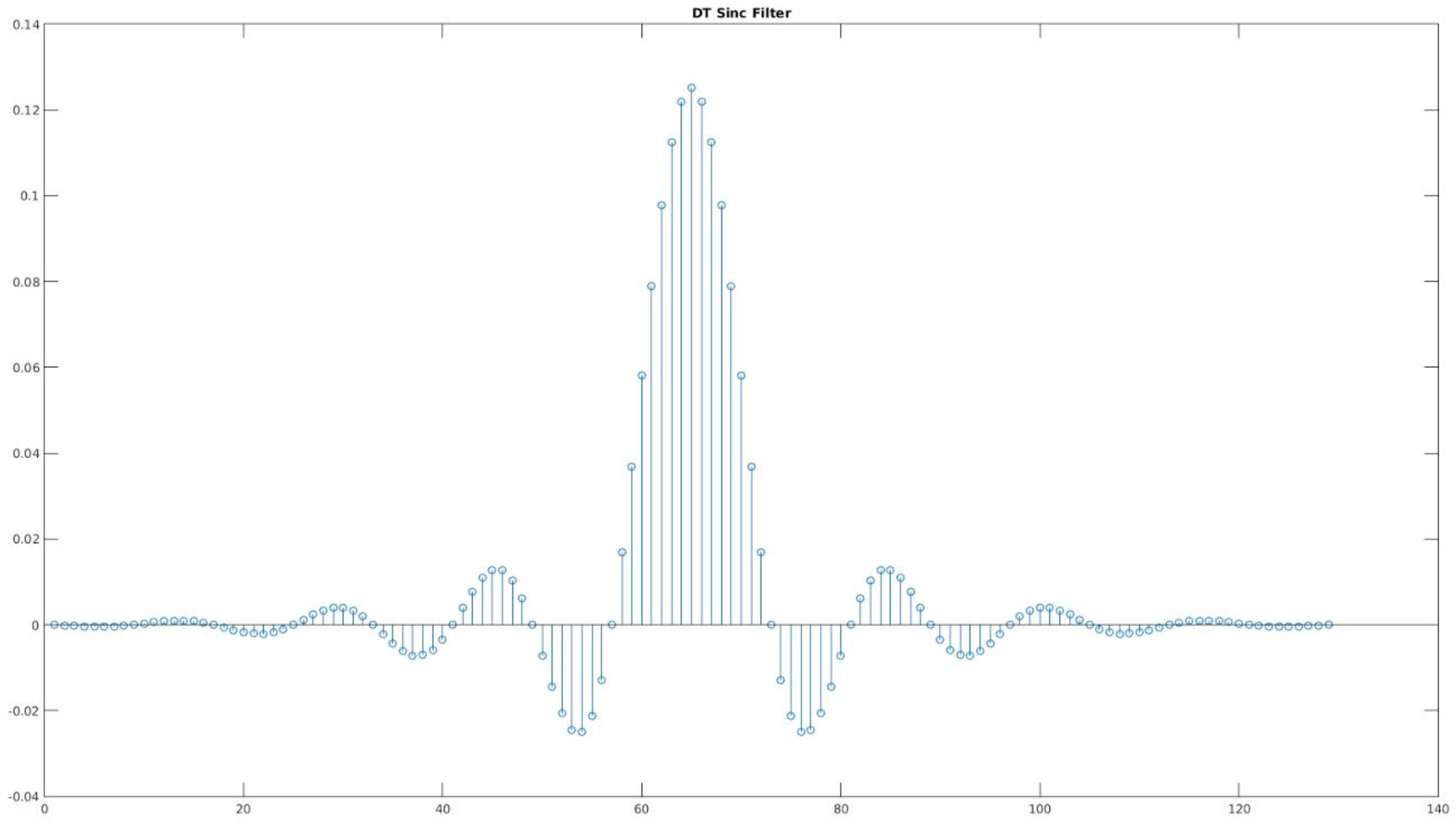


Figure 4. FIR coefficients for a 127-tap low pass filter

The causality of the FIR spectrum can be explained by the fact that is impossible to sample a signal in negative time. The effects of the time-shifted signal can be explained in frequency space:

$$x[n - n_0] \ <=> \ X(e^{jw})e^{-jw_0}$$

The fact that coefficients are shifted in time does not affect the magnitude of the spectrum, but introduces linear phase. For audio, linear phase is acceptable.

A naive implementation of the FIR module consists of storing the last N received audio samples, where N is the number of FIR coefficients. Then, in-between clock cycles of the sample-rate clock, multiply the Nth FIR coefficient with the Nth stored sample and accumulate the result. This result becomes the next output on the next sample-rate clock. This called a multiply-accumulate architecture (MAC for short).

$$y[n] = h_0x[n] + h_1x[n-1] + h_2x[n-2] + \ldots + h_Nx[n-N]$$

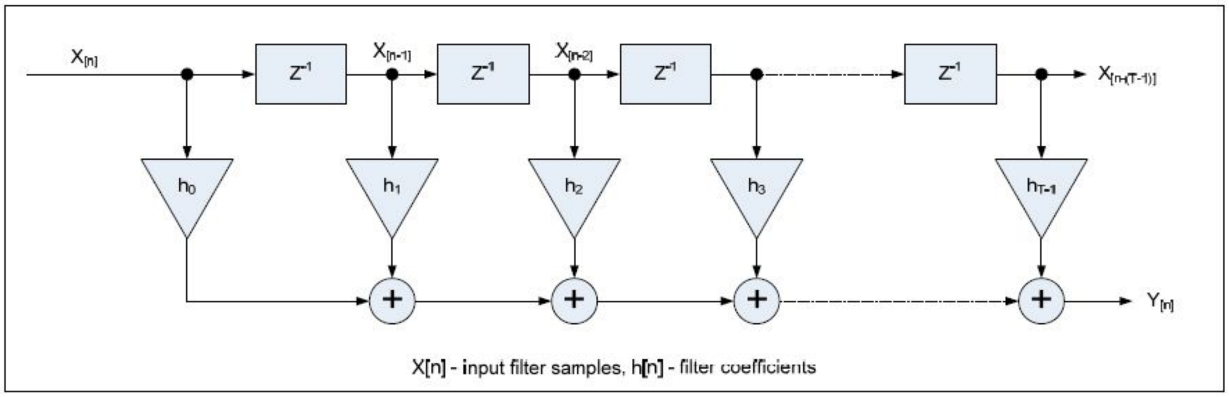


Figure 5. FIR MAC architecture
http://renesasrulz.com/cfs-file.ashx/__key/communityserver-blogs-components-weblogfiles/00-00-00-00-68/FIR-filter.jpg

Since DSP Dude operates with odd-order FIR filters, the coefficients are symmetric and a better pipelined Distributed Arithmetic (DA) architecture can be used to calculate the output as shown below.

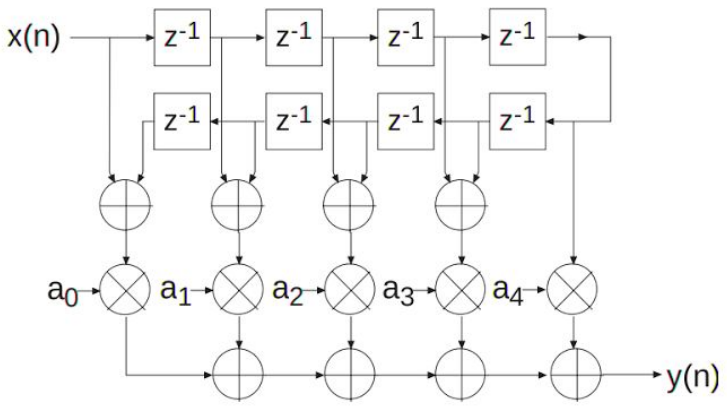


Figure 6. Symmetric Distributed Arithmetic Pipeline for 9-tap FIR

Coefficient symmetry means that the Nexys4 only has to store half of the coefficients. The pipelined nature of the DA architecture means that the FPGA is doing less work between sample-rate clock cycles because the work is distributed among different hardware in different stages.

DSP Dude can currently use either a 31-tap MAC FIR architecture or a 31-tap symmetric DA FIR architecture. In the case of the DA architecture, adding more taps is easy because the module is easily parameterizable.

Since FPGA's typically operate on integers, the FIR coefficients for DSP Dude are scaled by a power of 2. The result of the FIR filter computation is then scaled by the inverse of that amount. DSP Dude can operate on 24bit coefficients, but the scale factor has conservatively been set to 2^10 (1024). It is also very important to properly size the registers of the FIR computation. For a 31-tap filter with 24bit data and coefficients, the maximum result can be:

$$(2^{24} - 1) * (2^{24} - 1) * 31 = \text{0x1effffc200001f}$$

This is a 56bit number. To be extremely safe, all intermediate registers inside DSP Dude's FIR module are set to widths of 56bits. Unfortunately, the number of taps in the FIR filter causes an extremely confusing scaling problem. It's unclear if the final FIR result should also be scaled by the number of taps in the filter. One would expect the FIR filter design program to pre-bake this scale factor into the coefficients themselves but that would be sacrificing resolution. This uncertainty is the achilles heal of DSP Dude. DSP Dude requires hand-tuning for successful operation because the only thing known about the final 56bit result is that the correct 24bit window of audio data lies *somewhere* inside.

## FIR Module Complications (Yanni Coroneos)

As briefly explained above, the FIR module caused at least two sleepless nights. We observed that DSP Dude works great as a high-pass filter or a low-pass filter with a very high frequency cutoff (3KHz) but DSP Dude does not perform well as a low pass filter. There are two issues that can cause this. Firstly, as mentioned before, the output is scaled by 31, which is not a power of two. This means that correcting for the scale cannot easily be done by bit shifts and this causes distortion because we do use bit shifts. Additionally, low pass filters require more FIR taps in order to function effectively and DSP Dude currently only uses 31 taps. Since parameterizing the DA FIR module is very easy, we can probably increase the amount of FIR taps at a later date.

## Coefficient DB Module (Valentina Chamorro)

The goal of DSP Dude is to change FIR filters without re-synthesizing verilog. We accomplish this by storing computed FIR filter coefficients on an SDcard and then loading them with the Nexys4. The SDcard is operated in SPI mode for ease but this means that reads are quantized to 1 block at 1 byte at a time. 1 block is 512bytes but our coefficients are 24bits, or 3 bytes. Thus, we can pack 170 FIR coefficients into a single block. Three successive 8-byte reads yield 1 FIR coefficient. Since DSP Dude currently only uses 31 coefficients, the other 139 read are ignored. Future updates to DSP Dude will use the currently ignored coefficients.

## Coefficient DB Module Complications (Valentina Chamorro)

Packing coefficients was a fun problem to solve because no conventional programming language has 24bit data types. The solution came through using 32bit integers and simply ignoring the MSB before writing the data out to the sdcard.

**in Python:**

```
#input is a 32bit data type that will be truncated to 24bits
def getbytes(data):
    return [(data>>16)&0xFF, (data>>8)&0xFF, (data)&0xFF]

#write 24bit coefficients to a file or device
with open(filename, 'wb+') as f:
        for a,b,c in map(getbytes, [coeffs]):
            f.write(str(bytearray([a])))
            f.write(str(bytearray([b])))
            f.write(str(bytearray([c])))
```

**in Matlab:**

```
fileID = fopen('coeffs.bin','wb');
for i = 1:length(b)
   for j = [16 8 0]
      coeffs_shift = bitshift(coeffs32(i),-j,'int32');
      coeffs24 = bitand(coeffs_shift,-1,'int32');
      if coeffs24<128
         fwrite(fileID,coeffs24,'int8');
      else
         fwrite(fileID,coeffs24);
      end
   end
end
fclose(fileID);
```

## Volume Module (Yanni Coroneos)

DSP Dude has left/right channel volume adjust capability. The user selects the channel by pressing the Nexys4 BTNC and then adjusts volume up or down with BTNU or BTND, respectively. In order to do this all 3 buttons are first debounced, then they are sampled at 100MHz to detect rising edges, and finally this information is integrated into 8bit volume registers for the left channel and the right channel. Volume output is given to the Master FSM, which adjusts volume, and the VGA display module which displays the volume information on the screen.

## Volume Module Complications (Yanni Coroneos)

There were none. The volume module was very straightforward to implement.

## VGA Display Module (Yanni Coroneos)

DSP Dude can display current left/right channel volumes on an attached VGA screen. VGA operates on the idea of horizontal and vertical scan lines, hcount and vcount for short. The vector $(hcount, vcount)$ gives the current pixel in the image whose color is to be determined, as shown below.

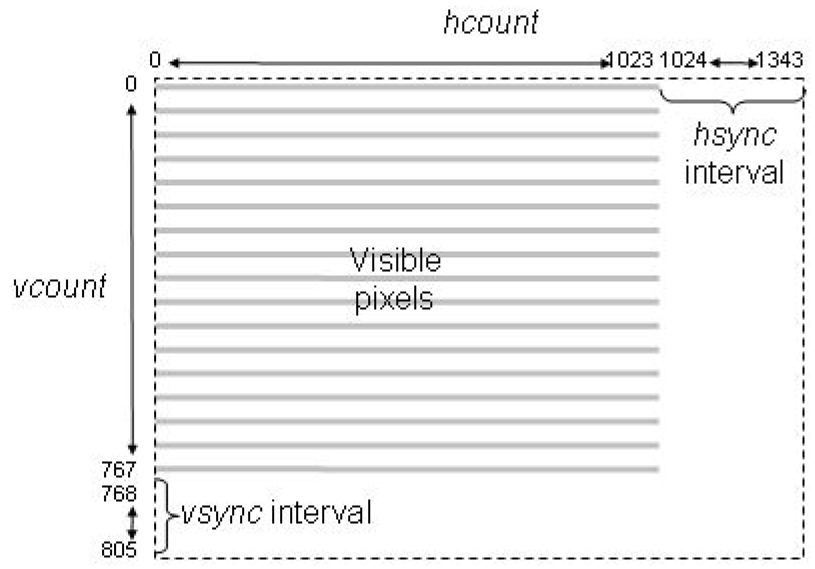


Figure 7. VGA pixel position

Displaying 2D images, or sprites, is then accomplished by checking whether $(hcount, vcount)$ is within the outline of the object and then setting the pixel value accordingly. DSP Dude represents volume with two boxes whose height is the magnitude of the volume. The left box on screen is for the left channel and the right box on screen is for the right channel. The active channel box is highlighted red.

## VGA Display Module Complications (Yanni Coroneos)

The VGA module from the old 6.111 labkit was ported to the Nexys4. The only difference is that the Nexys4 can only output 12bit color instead of 24bits. The labkit VGA module requires a 64MHz clock so we generated one with the Vivado clock wizard.

## Evaluation and Conclusion

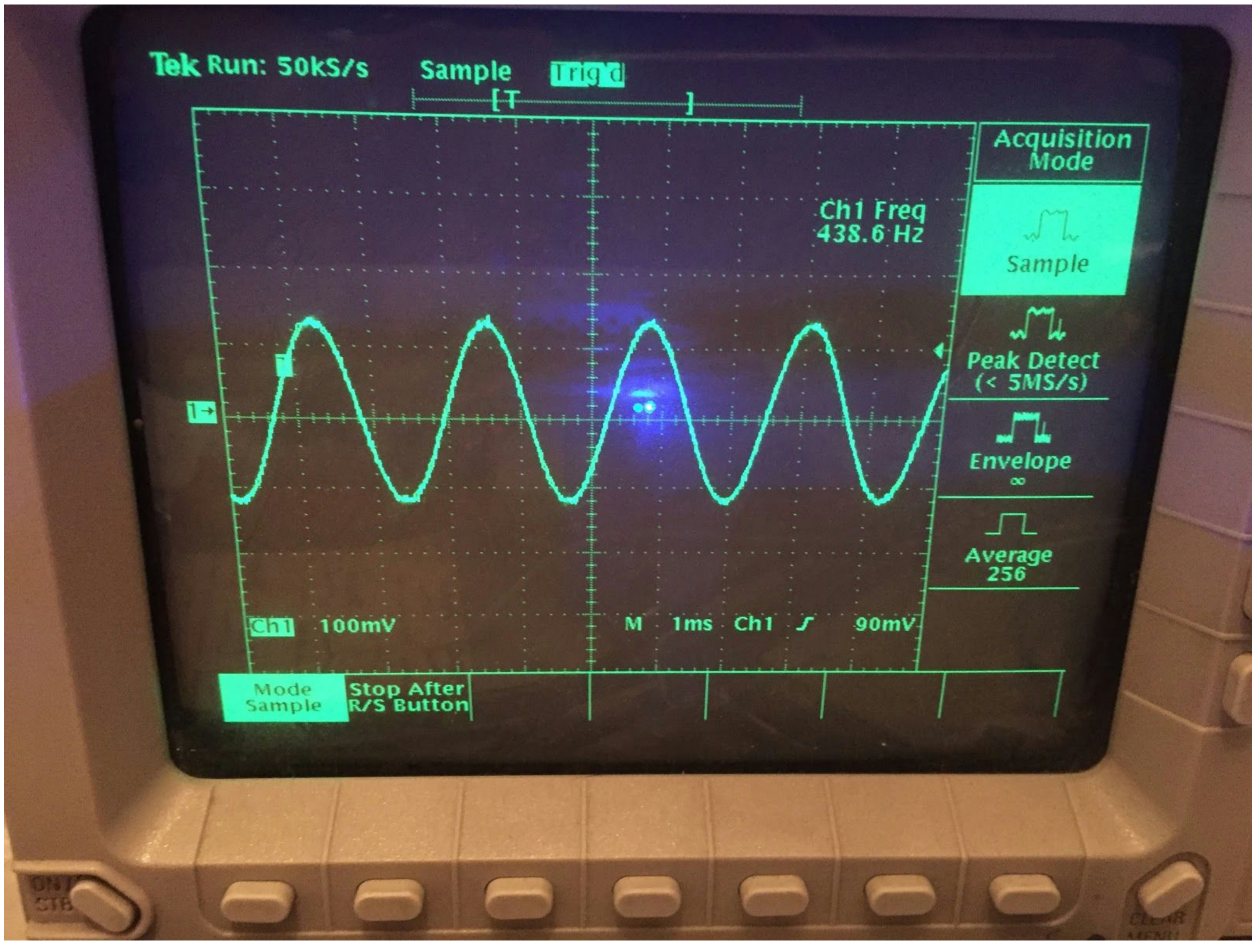


Figure 8. Scope shot of DSP Dude producing a 440Hz sinusoid

Currently, DSP Dude can work as described with certain filter cutoffs. Physically, DSP Dude consists of the Nexys4, a switching power supply and two bread boards with the AK4117 and AK4396 on them. Audio input comes in over SPDIF/TOSLink and DSP Dude can either reproduce the exact input or transform it according to an FIR filter loaded on an sd card. DSP Dude's one limitation is the class of filters that it can implement. As mentioned, only high-frequency cutoff filters work best but we believe that extending the number of FIR taps in DSP Dude is a reasonably achievable task.

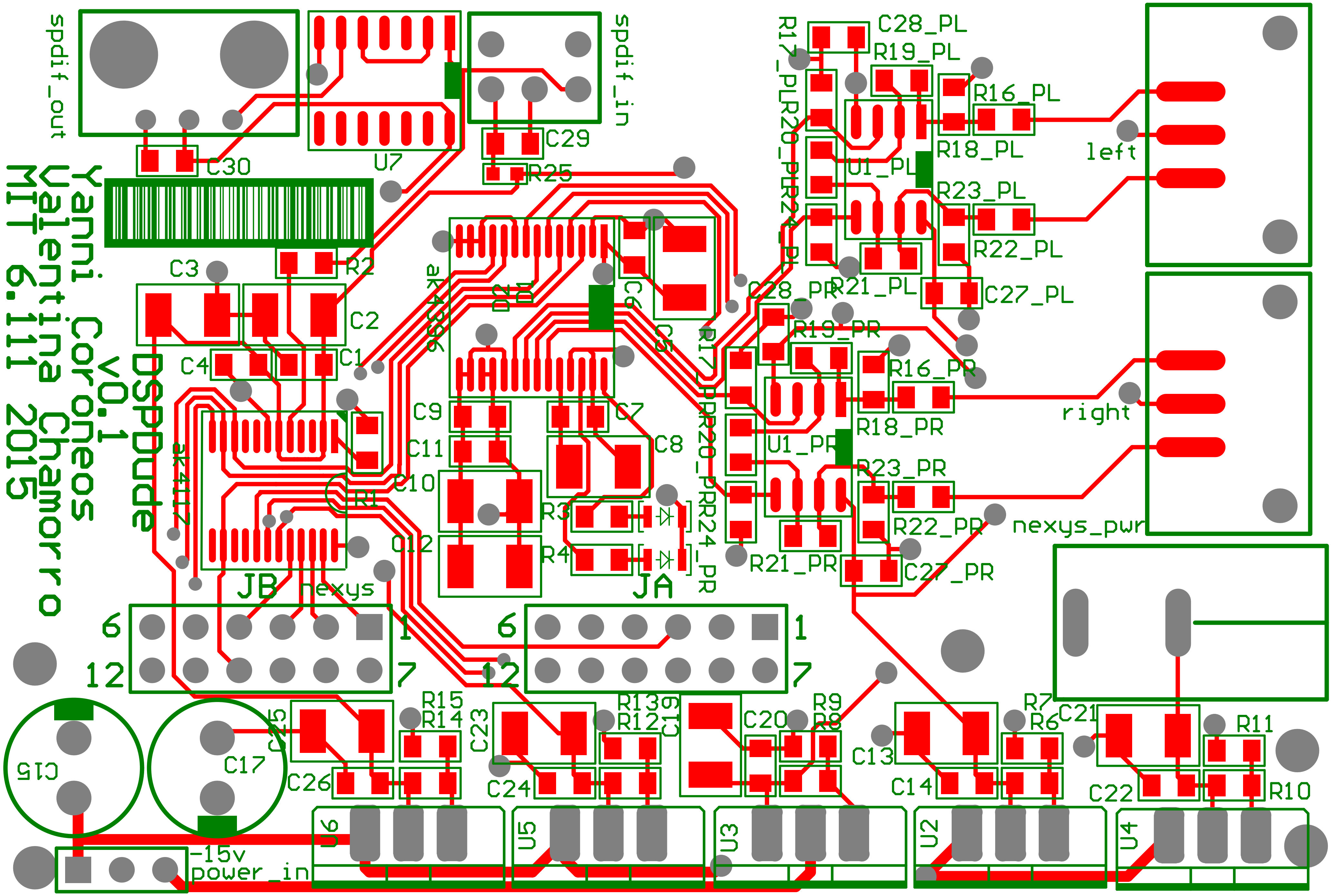
As far as usability goes, we have been using DSP Dude in our personal audio system for the past 2 weeks and it has proven to be very reliable. Development of DSP Dude into a cleaner product is definitely on the agenda.

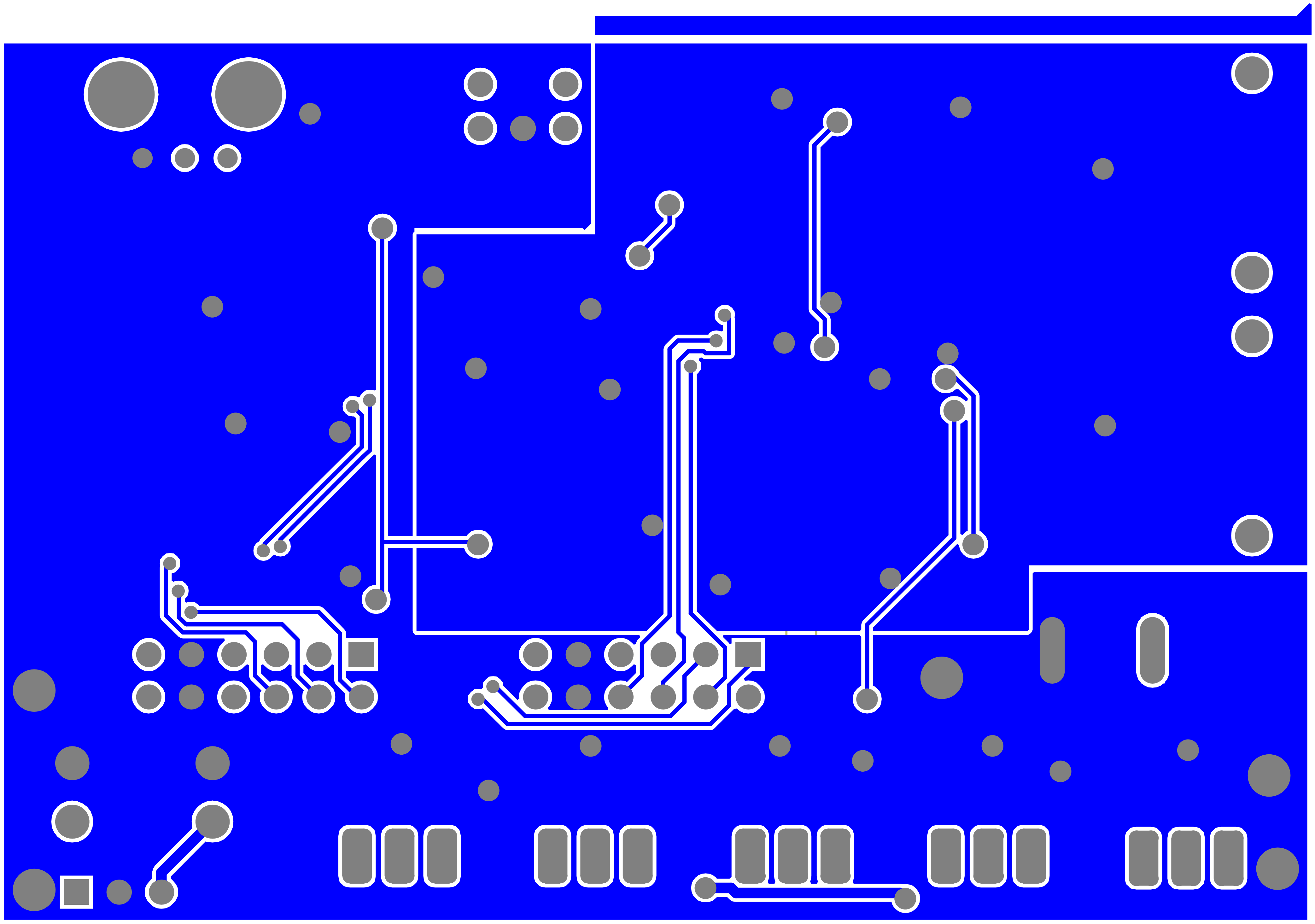
# Things That Didn’t Quite Make It

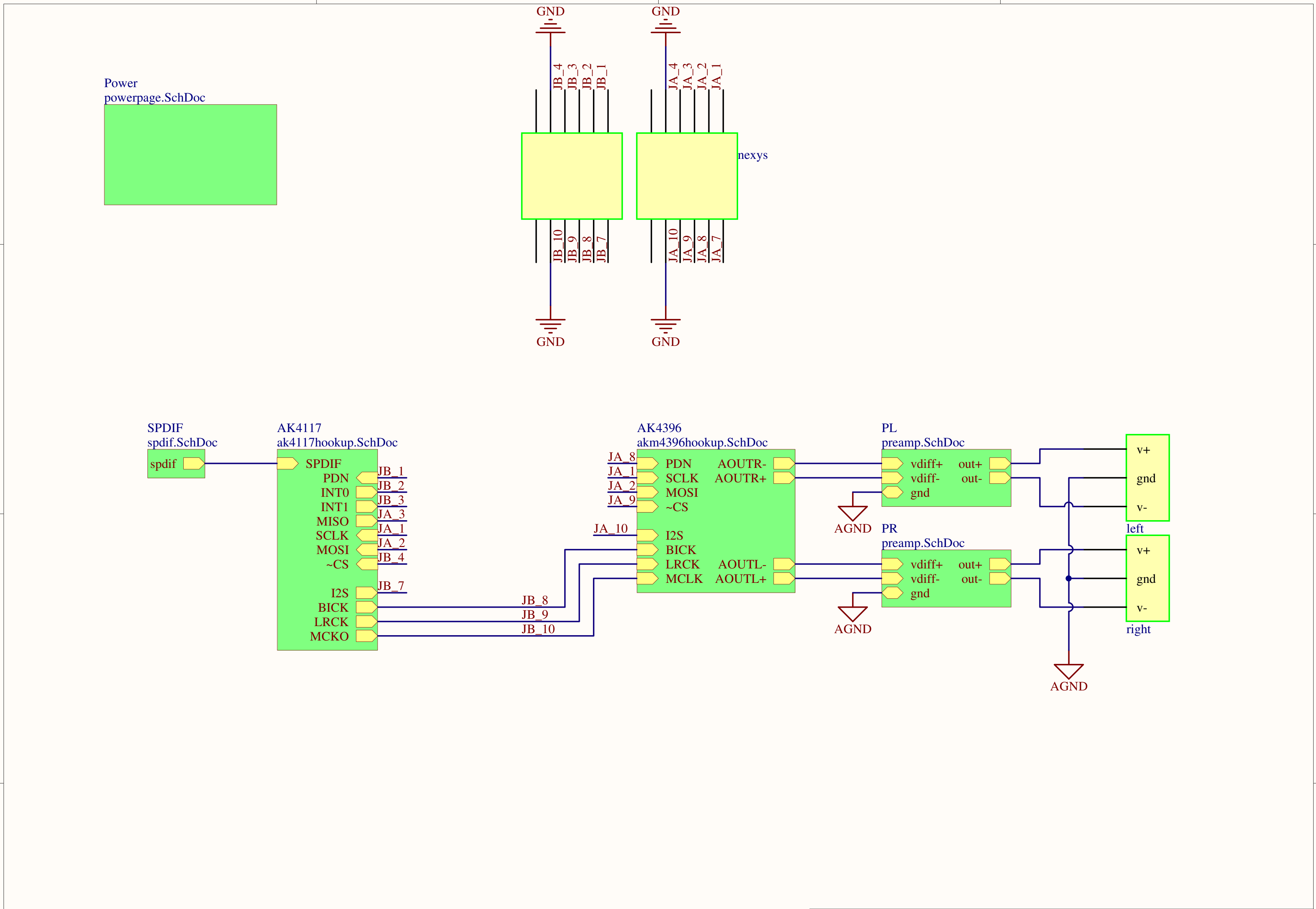
## PCB

In order to improve performance and reduce area, DSP Dude was fabricated onto a PCB that can be attached to the JB and JC ports of the Nexys4. The PCB improves performance because it reduces the length and parasitic effects that wires on the I2S bus must experience. On the current DSP Dude, jostling the wires will result in a momentary PLL lock loss. This causes bit-clock to become out of phase with MCLK and results in the most significant bit of one channel being lost, which causes one speaker to become quiet compared to the other.

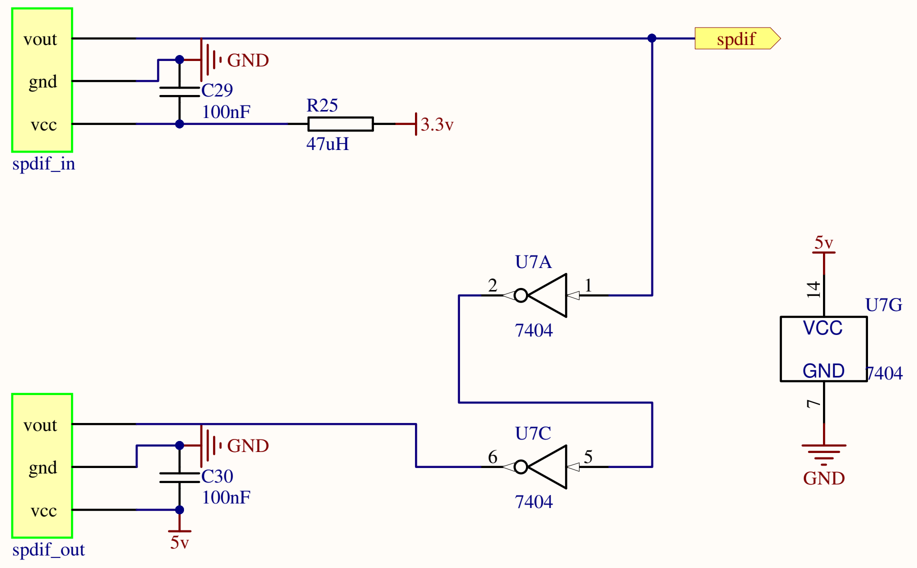
Although the PCB’s arrived on time, Digikey sent the wrong XLR connector so assembly before checkoff was impossible. Reproduced below is the schematic capture as well as the top/bottom copper layers.


Yanni Coroneos
Valentina Chamorro
MIT 6.111 2015
DSPDude v0.1
spdif_out
spdif_in
U7
C30
C29
R25
C3
C2
C1
C4
R2
ak4396
ak4117
JB
nexys
JA
left
right
nexys_pwr
-15v
power_in
C15
C17
U6
U5
U3
U2
U4
C28_PL
R19_PL
R16_PL
R18_PL
U1_PL
R23_PL
R22_PL
R21_PL
C27_PL
C28_PR
R19_PR
R16_PR
R18_PR
U1_PR
R23_PR
R22_PR
R21_PR
C27_PR
R17_PLR20_PLR24_PL
R17_PRR20_PRR24_PR
C9
C11
C10
C12
C7
C8
C6
C5
R3
R4
R1
C26
C25
R15
R14
C23
C24
R13
R12
C19
C20
R9
R8
C13
C14
R7
R6
C21
R11
C22
R10
6
12
1
7

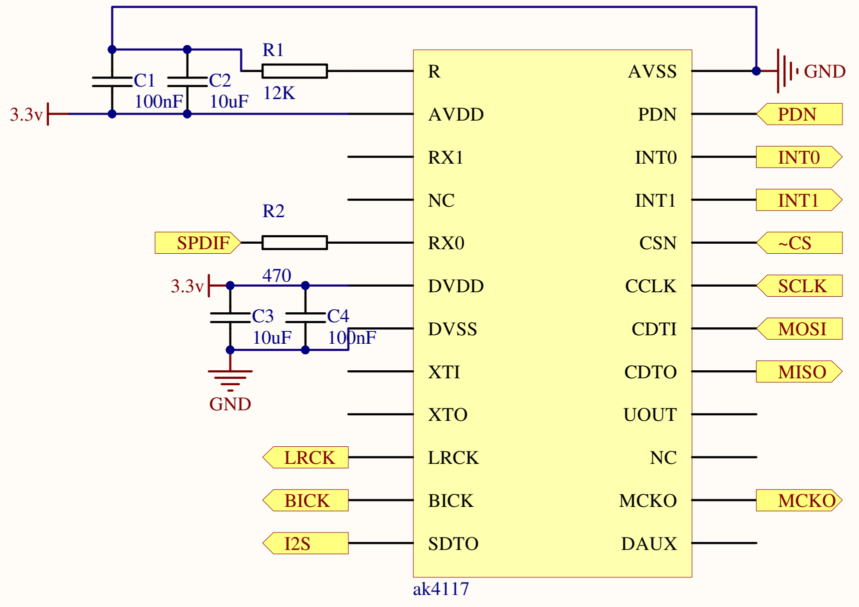




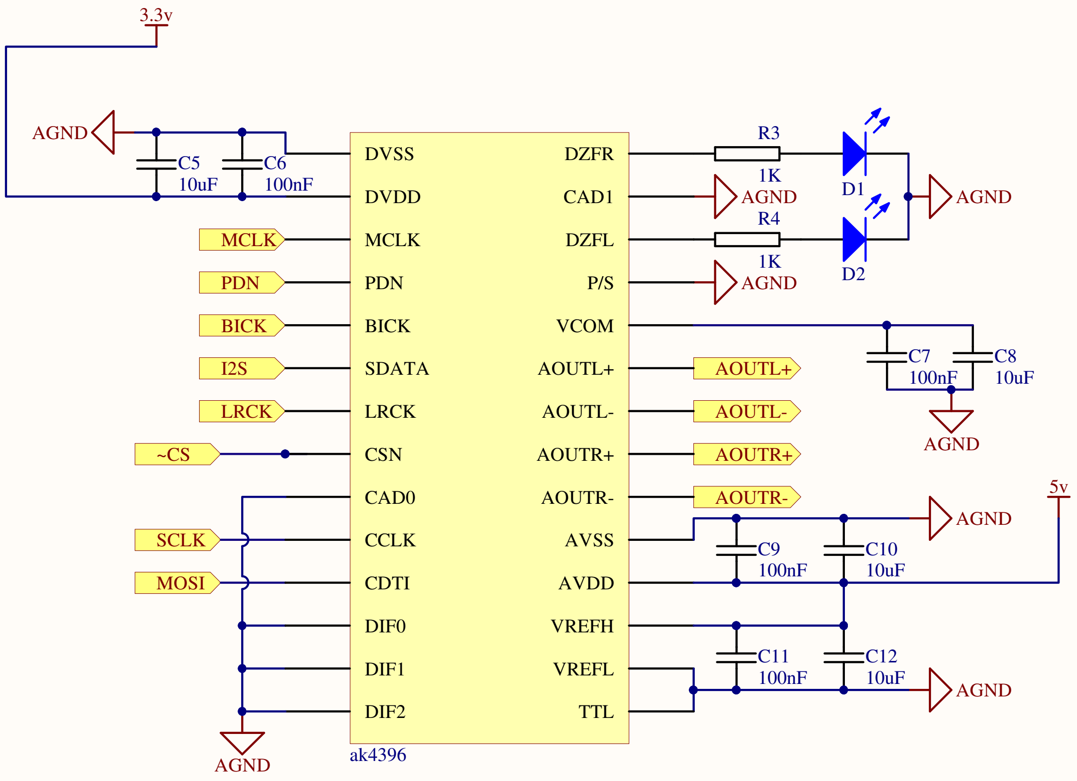
| Title | | |
|---|---|---|
| Size<br>A | Number | Revision |
| Date: 12/7/2015 | | Sheet of |
| File: C:\Users\..\master.SchDoc | | Drawn By: |



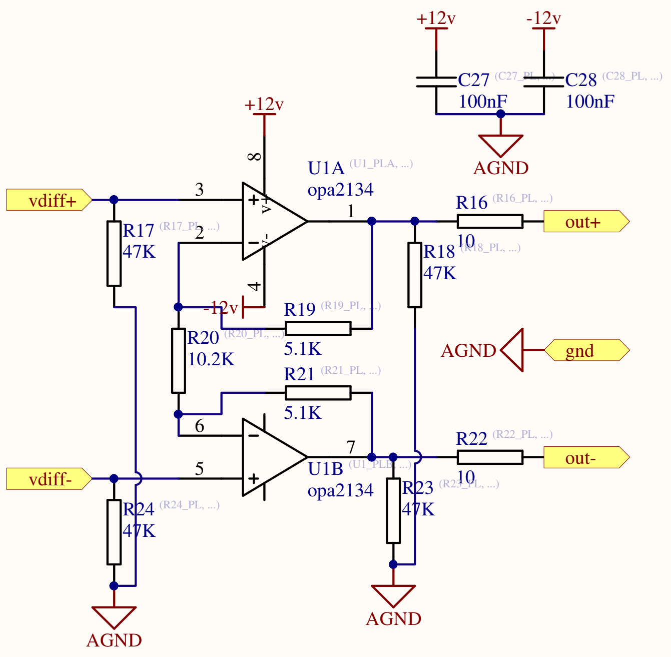
| Title | | |
|---|---|---|
| Size A | Number | Revision |
| Date: 12/7/2015 | | Sheet of |
| File: C:\Users\..\spdif.SchDoc | | Drawn By: |



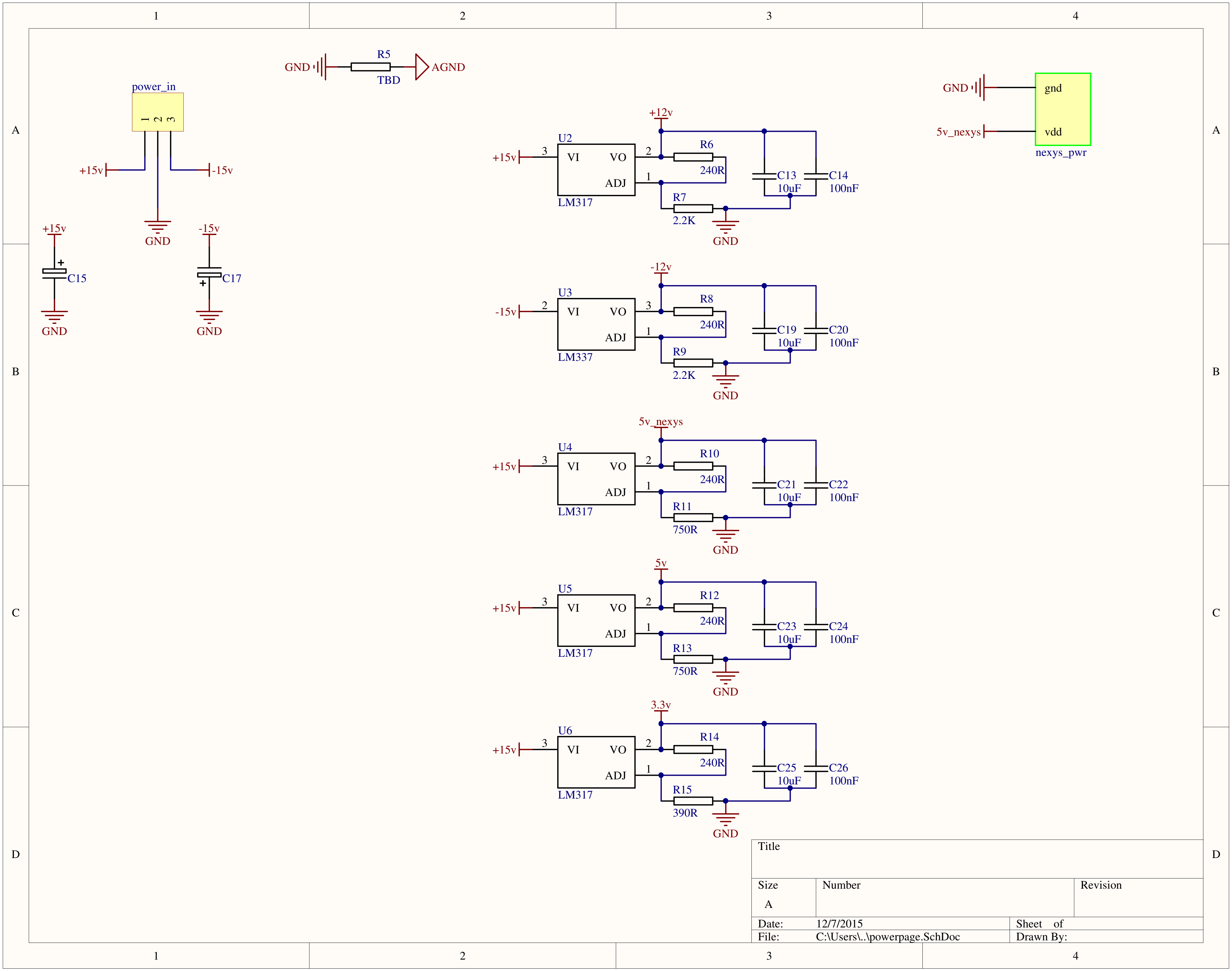
| | | |
|---|---|---|
| Title | | |
| Size A | Number | Revision |
| Date: 12/7/2015 | | Sheet of |
| File: C:\Users\..\ak4117hookup.SchDoc | | Drawn By: |



| Title | | |
|---|---|---|
| Size: A | Number | Revision |
| Date: 12/7/2015 | | Sheet of |
| File: C:\Users\..\akm4396hookup.SchDoc | | Drawn By: |



| | | | |
|---|---|---|---|
| Title | | | |
| Size<br>A | Number | | Revision |
| Date: | 12/7/2015 | Sheet of | |
| File: | C:\Users\..\preamp.SchDoc | Drawn By: | |



| Title | | |
|---|---|---|
| Size<br>A | Number | Revision |
| Date: 12/7/2015 | | Sheet of |
| File: C:\Users\..\powerpage.SchDoc | | Drawn By: |

```
‘default_nettype none

//////////////////////////////////////////////////////////////////
////////////////
////
//// dspdude on a digilent nexys 4 DDR
////
////
////
//// Author: Yanni Coroneos, Valentina Chamorro
////
//////////////////////////////////////////////////////////////////
////////////////

module dspdude(
input wire CLK100MHZ, //input clock
//input wire RESET, //switches
//input wire MUTE,
input wire [15:0] SW,
input wire BTNU, BTNC, BTND,
input wire BICK_IN, //input ports
input wire LRCLK_IN,
input wire MCLK_IN,
input wire I2S_IN,
input wire PLL_LOCKED,
inout wire [3:0] SD_DAT,
output wire SD_RESET,
//output wire [3:1] SD_DAT,
output wire SD_CMD,
output wire SD_SCK,
output wire MOSI,
output wire SCLK,
output wire CODEC_CS,
output wire CODEC_PDN,
output wire AK_CS,
output wire AK_PDN,
output wire CODEC_I2S,
output wire [15:0] LED, //indicators
output wire LED16_B,
output wire LED17_B,
output wire [3:0] VGA_R, VGA_G, VGA_B,
output wire VGA_HS, VGA_VS);
```

```
    assign SD_RESET=0;
    assign SD_DAT[2:1]=2'b11;
    assign LED[14:3] = 12'b000000000000;
    assign LED[15] = PLL_LOCKED;
    wire RESET=SW[0];
    //assign CODEC_I2S=I2S_IN;
    wire serial_clk, fsm_clk, fir_clk, vgaclk, volume_clk,
sdcard_clk;
    //clock generation
    clockmaker clocks(.reset(0), .clk_100Mhz(CLK100MHZ),
.ak4117_mclk(MCLK_IN), .serial_clk(serial_clk),
.fsm_clk(fsm_clk), .fir_clk(fir_clk), .vga_clk(vgaclk),
.volume_clk(volume_clk), .sdcard_clk(sdcard_clk));

    // UP and DOWN and SWITCH buttons for volume
    wire up,down,center;
    debounce
db1(.reset(RESET),.clock(CLK100MHZ),.noisy(BTNC),.clean(center));
    debounce
db2(.reset(RESET),.clock(CLK100MHZ),.noisy(BTNU),.clean(up));
    debounce
db3(.reset(RESET),.clock(CLK100MHZ),.noisy(BTND),.clean(down));

    //volume controller
    wire [7:0] leftvolume, rightvolume;
    wire channelselect;
    volumecontrol vc(.clk(volume_clk), .upvolume(up),
.downvolume(down), .switchchannel(center),
.leftvolume(leftvolume), .rightvolume(rightvolume),
.channelselect(channelselect));

    //volume display
    volumedisplay vdisp(.reset(RESET), .clk_65mhz(vgaclk),
.leftvolume(leftvolume), .rightvolume(rightvolume),
.channelselect(channelselect), .vga_r(VGA_R), .vga_g(VGA_G),
.vga_b(VGA_B), .vga_hs(VGA_HS), .vga_vs(VGA_VS));

    //master fsm
    //master fsm for configuring 4117,4397,polling interrupts,
and sending volume
    wire ak4117_config_done, ak4396_config_done;
    assign LED16_B = ak4117_config_done;
    assign LED17_B = ak4396_config_done;
    //assign SCLK = serial_clk;
    masterfsm thefsm(.reset(RESET), .clk100mhz(CLK100MHZ),
.fsmclk(fsm_clk), .leftvolume(leftvolume),
.rightvolume(rightvolume), .serial_clk(serial_clk),
.spi_sclk(SCLK), .spi_mosi(MOSI), .ak4117_cs(AK_CS),
.ak4117_pdn(AK_PDN), .ak4396_cs(CODEC_CS),
.ak4396_pdn(CODEC_PDN), .ak4117_idle(ak4117_config_done),
.ak4396_idle(ak4396_config_done));

    // ready signals
    wire i2sready, dataready, shiftready, bitwait;
    synchronizer i2sreadymaker(.reset(RESET),
.fasterclk(~fir_clk), .slowerclk(~BICK_IN), .ready(i2sready));
// pulse high on falling edge
    synchronizer firreadymaker(.reset(RESET),
.fasterclk(~fir_clk), .slowerclk(LRCLK_IN), .ready(bitwait));  //
pulse high on rising edge of lrclk
    synchronizer bitwaitmaker(.reset(RESET),
.fasterclk(~fir_clk), .slowerclk(~LRCLK_IN), .ready(dataready));
// pulse high on falling edge or lrclk
    synchronizer shiftreadymaker(.reset(RESET),
.fasterclk(~fir_clk), .slowerclk(LRCLK_IN), .ready(shiftready));
  // pulse high on rising edge



    //ak 4117 i2s receiver
    wire signed [31:0] leftchannel, rightchannel;
    i2s_receiver receiver(.reset(RESET), .firclk(fir_clk),
.bick(i2sready), .lrck(LRCLK_IN), .sdto(I2S_IN),
.ready(dataready), .bitwait(bitwait), .data_left(leftchannel),
.data_right(rightchannel));



    //fir module configured for MAC architecture
    wire signed [23:0] leftchannelout, rightchannelout;
    wire signed [9:0] coefficient;
    wire [4:0] leftcoeffindex, rightcoeffindex;
    //symmetricdafir betterleftfir(.reset(RESET),
.ready(dataready), .clk(fir_clk), .x(leftchannel[31:8]),
.y(leftchannelout));
    fir leftchannelfir(.reset(RESET), .ready(dataready),
.clk(fir_clk), .x(leftchannel[31:8]), .y(leftchannelout),
.coeffindex(leftcoeffindex), .coeff(coefficient));
    //dualfir bothchannels(.reset(RESET), .ready(dataready),
.clk(fir_clk), .xleft(leftchannel[31:8]),
.xright(rightchannel[31:8]), .yleft(leftchannelout),
.yright(rightchannelout), .coeffindex(coeffindex),
.coeff(coefficient));
    //symmetricdafir betterrightfir(.reset(RESET),
.ready(dataready), .clk(fir_clk), .x(rightchannel[31:8]),
.y(rightchannelout));
    fir rightchannelfir(.reset(RESET), .ready(dataready),
.clk(fir_clk), .x(rightchannel[31:8]), .y(rightchannelout),
.coeffindex(rightcoeffindex), .coeff(coefficient));

    //coefficient module for the fir module
    coeffdatabasev2 database(.reset(RESET),
.clk_25mhz(sdcard_clk), .index(leftcoeffindex),
.coefficient(coefficient), .status(LED[2:0]), .sd_cs(SD_DAT[3]),
.sd_mosi(SD_CMD), .sd_sclk(SD_SCK), .sd_miso(SD_DAT[0]));
    //coeffs31 filter(.index(leftcoeffindex),
.coeff(coefficient));

    // akm4396 driver+i2s tx
    wire outnoise;
    assign CODEC_I2S = SW[15] ? I2S_IN : outnoise;
    i2s_tx_symm i2x_tx(.reset(RESET), .firclk(fir_clk),
.bick(i2sready), .bitwait(bitwait), .lrck(LRCLK_IN),
.ready(dataready), .data_left({leftchannelout,{8{1'b0}}}),
.data_right({rightchannelout,{8{1'b0}}}), .sdto(outnoise));

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////
/////////////////
// Company:
// Engineer: yo mama
//
// Create Date: 10/31/2015 02:46:19 AM
// Design Name:
// Module Name: clockmaker
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////
/////////////////

//mclk is 24.5760MHz
//lrclk is 48KHz
//bclk is 12.2880MHz
//serial_clk is 9.6KHz

module clockmaker(
input wire reset, clk_100Mhz, ak4117_mclk,
output wire serial_clk, fsm_clk, fir_clk, vga_clk, volume_clk,
sdcard_clk
    );
    clkgen0 someclocks(.clk_in1(ak4117_mclk), .reset(reset),
.clk_out1(fir_clk)); //akm4117_mclk is exactly 24.5760mhz,
fir_clk is exactly 98.304MHz
    clkgenvga moreclks(.clk_in1(clk_100Mhz), .reset(reset),
.clk_out1(vga_clk));
    slowdivider #(.LOGLENGTH(15), .COUNTVAL(5200))
serialclkgen(.inclk(clk_100Mhz), .reset(reset),
.newclk(serial_clk)); //serial_clk is 9.6KHz
    slowdivider #(.LOGLENGTH(32), .COUNTVAL(1000000))
fsmclkgen(.inclk(clk_100Mhz), .reset(reset), .newclk(fsm_clk));
```

```
//fsm_clk 100hz
    slowdivider #(.LOGLENGTH(21), .COUNTVAL(100000))
volclkgen(.inclk(clk_100Mhz), .reset(reset),
.newclk(volume_clk)); //volume_clk
    slowdivider #(.LOGLENGTH(4), .COUNTVAL(2))
sdcardclkgen(.inclk(clk_100Mhz), .reset(reset),
.newclk(sdcard_clk)); //sdcard_clk

endmodule
```

```
‘timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////
/////////////////
// Company:
// Engineer:
//
// Create Date: 11/08/2015 12:09:02 AM
// Design Name:
// Module Name: fir
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////
/////////////////

////////////////////////////////////////////////////////////////////
//////////////
//
// 512-tap FIR filter, 24-bit signed data, 23-bit signed
coefficients.
// ready is asserted whenever there is a new sample on the X
input,
// the Y output should also be sampled at the same time.
// coefficients have been scaled by 2**10, so has the output (it’
// expanded from 24 bits to 34 bits).  To get an 24-bit result
from the
// filter just divide by 2**10, ie, use ac[33:10].
//
////////////////////////////////////////////////////////////////////
//////////////

module fir(
input wire clk, ready, reset,
input wire signed [23:0] x,
output reg signed [23:0] y,
output wire [4:0] coeffindex,
input wire signed [9:0] coeff
    );
    //if we scale coefficients by 2**10, then the output will be
34 bits
    reg signed [23:0] sample [31:0];
    reg signed [52:0] acc=0;
    reg [4:0] offset=0;
    reg [4:0] index=0; //index into fir coefficients
    assign coeffindex=index;

    reg [1:0] state=0;
    integer i;
    always @(negedge clk)
        begin
        if(reset)
            begin
            for (i=0; i<32; i=i+1) sample[i] <= 0;
            index<=0;
            offset<=0;
            y<=0;
            end
        else if (ready)
            begin
            y<=acc[47:24];
            acc<=0;
            sample[offset] <= x; //store data at the head pointer
            offset<=offset+1; //loops over to 0 when overflowed
            index<=0;
            state<=0;
            end

        else //do MACs
            begin
            case(state)
            0:
                begin
                index <=index+1;

acc<=acc+{{14{coeff[9]}},coeff[9:0]}*sample[(offset-index)&31];

                if (index>=30)
                    state<=state+1;
                end
            1:
                begin

                end
            endcase
            end
        end
endmodule
```

```
'timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////
/////////////////
// Company:
// Engineer:
//
// Create Date: 11/07/2015 03:03:25 PM
// Design Name:
// Module Name: i2s_receiver
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////
/////////////////
// takes AKM4117 IC receiver i2s and converts it into a 24bit
number for fpga
module i2s_receiver(
    input wire reset,
    input wire firclk,
    input wire bick,                // ready pulse on every
falling edge of BICK
    input wire lrck,
    input wire sdto,                // i2s data from ic receiver
    input wire ready,               // data ready
    input wire bitwait,
    output reg [31:0] data_left, data_right    // data that goes
to fpga
    );

    // additional registers and parameters
    reg [31:0] shiftreg_l = 32'b0;    // shift registers
    reg [31:0] shiftreg_r = 32'b0;

    always @(negedge firclk)
        begin
        if (reset)
            begin
            //state <= START;
            shiftreg_l <= 32'b0;
            shiftreg_r <= 32'b0;
            end
        else
            begin
            if (bick)
                begin
                if (ready)
                    begin
                    data_left <= shiftreg_l;
                    data_right <= shiftreg_r;
                    end
                    else if (bitwait)
                    begin
                    //do nothing on rising edge of lrck
                    end
                    else if (lrck)
                    begin
                        shiftreg_r <= {shiftreg_r[30:0], sdto};
//shift in r channel when lrck high
                    end
                    else if (!lrck)
                    begin
                        shiftreg_l <= {shiftreg_l[30:0], sdto};
//shift in l channel when lrck low
                    end
                end
            end
        end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////
/////////////////
// Company:
// Engineer:
//
// Create Date: 11/26/2015 06:10:07 PM
// Design Name:
// Module Name: masterfsm
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////
/////////////////


module masterfsm(
input wire clk100mhz,
input wire fsmclk,
input wire reset,
input wire [7:0] leftvolume, rightvolume,
input wire serial_clk,
output wire spi_sclk, spi_mosi,
output reg ak4117_cs=1, ak4396_cs=1, ak4117_pdn=1, ak4396_pdn=1,
ak4117_idle=0, ak4396_idle=0
    );

    //ak4117 configuration
    //data order is
C1,C0,R/W,A4,A3,A2,A1,A0,D7,D6.D5,D4,D3,D2,D1,D0
    parameter CONTROL0_CLOCK_SOURCE =       16'b0010000000001111;
  // select PLL as master clock source
    parameter CONTROL1_CLOCK_FREQ =         16'b0010000100000000;
  // select master clk for PLL mode, fs = 48kHz
    parameter CONTROL2_FORMAT_SETTING =     16'b0010001000001101;
  // select I2S as comm format and channel 1 for fs
    parameter CONTROL3_INT_0 =             16’b0010001101111111;
  // enable all the interrupts on INT0

    //ak4396 configuration
    //data order is
C1,C0,R/W,A4,A3,A2,A1,A0,D7,D6.D5,D4,D3,D2,D1,D0
    parameter CONTROL0_AK4396 = 16’b0010000010000111;
    parameter CONTROL1_AK4396 = 16’b0010000110000010;
    parameter LCH_VOLUME = 8’b00100011;
    parameter RCH_VOLUME = 8’b00100100;

    //spi modules
    reg [15:0] controldata=0;
    //wire spistart=~(ak4117_cs ^ ak4396_cs);
    reg spistart=0;
    //spicontroller spimodule(.reset(spistart),
.spiclk(serial_clk), .sclk(spi_sclk), .mosi(spi_mosi),
.data(controldata));
    wire unused_cs;
    serialcontroller sp2(.reset(reset), .serialclk(serial_clk),
.start(spistart), .data(controldata), .outclk(spi_sclk),
.shiftout(spi_mosi), .cs(unused_cs));
    //fsm variables
    reg [4:0] curstate =0;
    reg [7:0] oldleftvolume, oldrightvolume;
    always @(posedge fsmclk)
    begin
        if(reset)
            begin
            curstate <= 0;
            ak4117_cs <= 1;
            ak4396_cs <= 1;
            ak4117_pdn <= 1;
            ak4396_pdn <= 1;
            ak4117_idle <= 0;
            ak4396_idle <= 0;
            end
        else
            begin
            curstate <= curstate+1;
            case(curstate)
                5’d0:
                    begin
                    ak4117_cs <= 1;
                    ak4396_cs <= 1;
                    ak4117_pdn <= 1;
                    ak4396_pdn <= 1;
                    curstate <= curstate +1;
                    end
                5’d1: //configure ak4117
                    begin
                    ak4117_pdn <= 0;
                    curstate <= curstate +1;
                    end
                5’d2:
                    begin
                    ak4117_pdn <= 1;
                    curstate <= curstate +1;
                    end
                5’d3:
                    begin
                    ak4117_cs <= 0;
                    spistart<=1;
                    controldata <= CONTROL0_CLOCK_SOURCE; //write
control 0
                    curstate <= curstate +1;
                    end
                5’d4:
                    begin
                    ak4117_cs <= 1;
                    spistart<=0;
                    curstate <= curstate+1;
                    //curstate <=3;
                    end
                5’d5:
                    begin
                    ak4117_cs <= 0;
                    spistart<=1;
                    controldata <= CONTROL1_CLOCK_FREQ; //write
control 1
                    curstate <= curstate+1;
                    end
                5’d6:
                    begin
                    ak4117_cs <= 1;
                    spistart<=0;
                    curstate <= curstate+1;
                    end
                5'd7:
                    begin
                    ak4117_cs <= 0;
                    spistart<=1;
                    controldata <= CONTROL2_FORMAT_SETTING;
//write control 2
                    curstate <= curstate+1;
                    end
                5'd8:
                    begin
                    ak4117_cs <= 1;
                    spistart<=0;
                    curstate <= curstate+1;
                    end
                5'd9:
                    begin
                    ak4117_cs <= 0;
                    spistart<=1;
                    controldata <= CONTROL3_INT_0; //write
control 3
                    curstate <= curstate+1;
                    end
                5'd10:
                    begin
                    ak4117_cs <= 1;
                    spistart<=0;
                    curstate <= curstate+1;
                    ak4117_idle <= 1;
                    end
                5'd11:
                    begin
                    ak4396_pdn <= 0;
                    curstate <= curstate+1;
                    end
                5'd12: //configure ak4396
                    begin
                    ak4396_pdn <= 1;
                    curstate <= curstate+1;
                    end
                5'd13:
                    begin
                    ak4396_cs <= 0;
                    spistart<=1;
                    controldata <= CONTROL0_AK4396; //write
control 0
                    curstate <= curstate+1;
                    end
                5'd14:
                    begin
                    ak4396_cs <= 1;
                    spistart<=0;
                    curstate <= curstate+1;
                    end
                5'd15:
                    begin
                    ak4396_cs <= 0;
                    spistart<=1;
                    controldata <= CONTROL1_AK4396; //write
control 1
                    curstate <= curstate+1;
                    end
                5'd16:
                    begin
                    ak4396_cs <= 1;
                    spistart<=0;
                    curstate <= curstate+1;
                    ak4396_idle <= 1;
                    end
                5'd17: //both chips configured, drop into polling
mode and volume adjust
                    begin
                    if (oldleftvolume != leftvolume ||
oldrightvolume != rightvolume)
                        begin
                        oldleftvolume <= leftvolume;
                        oldrightvolume <= rightvolume;
                        curstate <= curstate+1;
                        end
                    end
                5'd18: //adjust volume
                    begin
                    ak4396_cs <= 0;
                    spistart<=1;
                    //write left volume
                    controldata<={LCH_VOLUME, leftvolume};
                    curstate <= curstate+1;
                    end
                5'd19:
                    begin
                    ak4396_cs<=1;
                    spistart<=0;
                    curstate <= curstate+1;
                    end
                5'd20:
                    begin
                    ak4396_cs <= 0;
                    spistart<=1;
                    //write right volume
                    controldata<={RCH_VOLUME, rightvolume};
                    curstate <= curstate+1;
                    end
                5'd21:
                    begin
                    ak4396_cs <= 1;
                    spistart<=0;
                    curstate <= 5'd17; //back to polling
                    end
            endcase
            end
    end
endmodule
```

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////
/////////////////
// Company:
// Engineer:
//
// Create Date: 11/07/2015 11:53:01 PM
// Design Name:
// Module Name: bicksync
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////
/////////////////


module synchronizer(
input wire fasterclk,
input wire slowerclk,
input wire reset,
output reg ready=0
    );
    reg [1:0] memory=2’b00;
    always @(posedge fasterclk)
        begin
        if (reset)
            memory<=2’b00;
        memory<={memory[0],slowerclk};
        end
    always @*
    begin
        if (memory=={1’b0, 1’b1})
            ready=1;
        else
            ready=0;
    end
endmodule
```

```
`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////
/////////////////
// Company:
// Engineer:
//
// Create Date: 11/26/2015 02:56:47 PM
// Design Name:
// Module Name: volumedisplay
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////////////////////////////////////
/////////////////


module volumedisplay(
input wire reset,
input wire clk_65mhz,
input wire [7:0] leftvolume, rightvolume,
input wire channelselect,
output wire [3:0] vga_r, vga_g, vga_b,
output wire vga_hs,
output wire vga_vs
    );
    // generate basic XVGA video signals
    wire [10:0] hcount;
    wire [9:0]  vcount;
    wire hsync,vsync,blank;
    xvga
xvga1(.vclock(clk_65mhz),.hcount(hcount),.vcount(vcount),
.hsync(hsync),.vsync(vsync),.blank(blank));
    // feed XVGA signals to user's pong game
   wire [11:0] pixel;
   wire phsync,pvsync,pblank;
   volumebars vb(.vclock(clk_65mhz),.reset(reset),
.hcount(hcount),.vcount(vcount),
.hsync(hsync),.vsync(vsync),.blank(blank),
.phsync(phsync),.pvsync(pvsync),.pblank(pblank),.pixel(pixel),
.leftvolume(leftvolume), .rightvolume(rightvolume),
.channelselect(channelselect));
   reg [11:0] rgb;
   reg b,hs,vs;
  always @(posedge clk_65mhz)
    begin
       hs <= phsync;
       vs <= pvsync;
       b <= pblank;
       rgb <= pixel;
     end
     assign vga_r = rgb[11:8];
     assign vga_g = rgb[7:4];
     assign vga_b = rgb[3:0];
     assign vga_hs = hs;
     assign vga_vs = vs;
endmodule
```

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////
/////////////////
// Company:
// Engineer:
//
// Create Date: 11/08/2015 12:09:02 AM
// Design Name:
// Module Name: fir
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////
/////////////////

//////////////////////////////////////////////////////////////////
//////////////
//
// 512-tap FIR filter, 24-bit signed data, 23-bit signed
coefficients.
// ready is asserted whenever there is a new sample on the X
input,
// the Y output should also be sampled at the same time.
// coefficients have been scaled by 2**10, so has the output (it’
// expanded from 24 bits to 34 bits).  To get an 24-bit result
from the
// filter just divide by 2**10, ie, use ac[33:10].
//
//////////////////////////////////////////////////////////////////
//////////////

module fir(
input wire clk, ready, reset,
input wire signed [23:0] x,
output reg signed [23:0] y,
output wire [4:0] coeffindex,
input wire signed [9:0] coeff
    );
    //if we scale coefficients by 2**10, then the output will be
34 bits
    reg signed [23:0] sample [31:0];
    reg signed [52:0] acc=0;
    reg [4:0] offset=0;
    reg [4:0] index=0; //index into fir coefficients
    assign coeffindex=index;

    reg [1:0] state=0;
    integer i;
    always @(negedge clk)
        begin
        if(reset)
            begin
            for (i=0; i<32; i=i+1) sample[i] <= 0;
            index<=0;
            offset<=0;
            y<=0;
            end
        else if (ready)
            begin
            y<=acc[47:24];
            acc<=0;
            sample[offset] <= x; //store data at the head pointer
            offset<=offset+1; //loops over to 0 when overflowed
            index<=0;
            state<=0;
            end

        else //do MACs
            begin
            case(state)
            0:
                begin
                index <=index+1;

acc<=acc+{{14{coeff[9]}},coeff[9:0]}*sample[(offset-index)&31];

                if (index>=30)
                    state<=state+1;
                end
            1:
                begin

                end
            endcase
            end
        end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////
/////////////////
// Company:
// Engineer:
//
// Create Date: 12/06/2015 04:26:11 AM
// Design Name:
// Module Name: symmetricdafir
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////
/////////////////

/*
*optional symmetric distributed arithmetic FIR
*/

module symmetricdafir(
    input wire clk, reset, ready,
    input wire signed [23:0] x,
    output reg signed [23:0] y
    );

    reg signed [23:0] coeffs [15:0];
    reg signed [23:0] dreg [29:0]; //30 pipeline stages
    reg signed [60:0] sum[14:0];
    reg signed [60:0] mul[15:0];
    reg signed [60:0] acc[14:0];

    reg [3:0] count=0;

    integer i;
    always @(negedge clk)
```

```
begin
    if(reset)
        begin
        count<=0;
        for(i=0; i<16; i=i+1) coeffs[i]=24’sd0;
        for(i=0; i<30; i=i+1) dreg[i]=0;
        for(i=0; i<15; i=i+1) sum[i]=0;
        for(i=0; i<16; i=i+1) mul[i]=0;
        for(i=0; i<15; i=i+1) acc[i]=0;
        //coeffs[15]<=24’sd1;
        /*coeffs[0] = -24’sd10871;
        coeffs[1] = -24’sd24226;
        coeffs[2] = -24’sd45336;
        coeffs[3] = -24’sd74503;
        coeffs[4] = -24’sd103876;
        coeffs[5] = -24’sd116755;
        coeffs[6] = -24’sd90105;
        coeffs[7] = 24’sd0;
        coeffs[8] = 24’sd171242;
        coeffs[9] = 24’sd428196;
        coeffs[10] = 24’sd757820;
        coeffs[11] = 24’sd1128921;
        coeffs[12] = 24’sd1496199;
        coeffs[13] = 24’sd1808259;
        coeffs[14] = 24’sd2017818;
        coeffs[15] = 24’sd2091654;
              */
        end
    else
        if (ready)
        begin
        dreg[0]<=x;
        //y<=acc[14][47-:24];
        y<=acc[14][23:0];
            //30 delay regs
            //stage 1
            dreg[1]<=dreg[0];
            dreg[2]<=dreg[1];
            dreg[3]<=dreg[2];
            dreg[4]<=dreg[3];
            dreg[5]<=dreg[4];
            dreg[6]<=dreg[5];
            dreg[7]<=dreg[6];
```

```
dreg[8]<=dreg[7];
dreg[9]<=dreg[8];
dreg[10]<=dreg[9];
dreg[11]<=dreg[10];
dreg[12]<=dreg[11];
dreg[13]<=dreg[12];
dreg[14]<=dreg[13];
dreg[15]<=dreg[14];
dreg[16]<=dreg[15];
dreg[17]<=dreg[16];
dreg[18]<=dreg[17];
dreg[19]<=dreg[18];
dreg[20]<=dreg[19];
dreg[21]<=dreg[20];
dreg[22]<=dreg[21];
dreg[23]<=dreg[22];
dreg[24]<=dreg[23];
dreg[25]<=dreg[24];
dreg[26]<=dreg[25];
dreg[27]<=dreg[26];
dreg[28]<=dreg[27];
dreg[29]<=dreg[28];

//next steps
sum[0]<=x+dreg[29];
sum[1]<=dreg[0]+dreg[28];
sum[2]<=dreg[1]+dreg[27];
sum[3]<=dreg[2]+dreg[26];
sum[4]<=dreg[3]+dreg[25];
sum[5]<=dreg[4]+dreg[24];
sum[6]<=dreg[5]+dreg[23];
sum[7]<=dreg[6]+dreg[22];
sum[8]<=dreg[7]+dreg[21];
sum[9]<=dreg[8]+dreg[20];
sum[10]<=dreg[9]+dreg[19];
sum[11]<=dreg[10]+dreg[18];
sum[12]<=dreg[11]+dreg[17];
sum[13]<=dreg[12]+dreg[16];
sum[14]<=dreg[13]+dreg[15];

//next steps
mul[0]<=sum[0]*coeffs[0];
mul[1]<=sum[1]*coeffs[1];
```

```
                mul[2]<=sum[2]*coeffs[2];
                mul[3]<=sum[3]*coeffs[3];
                mul[4]<=sum[4]*coeffs[4];
                mul[5]<=sum[5]*coeffs[5];
                mul[6]<=sum[6]*coeffs[6];
                mul[7]<=sum[7]*coeffs[7];
                mul[8]<=sum[8]*coeffs[8];
                mul[9]<=sum[9]*coeffs[9];
                mul[10]<=sum[10]*coeffs[10];
                mul[11]<=sum[11]*coeffs[11];
                mul[12]<=sum[12]*coeffs[12];
                mul[13]<=sum[13]*coeffs[13];
                mul[14]<=sum[14]*coeffs[14];
                mul[15]<=dreg[14]*coeffs[15];

                //n-stage output pipeline
                acc[0]<=mul[0]+mul[1];
                acc[1]<=acc[0]+mul[2];
                acc[2]<=acc[1]+mul[3];
                acc[3]<=acc[2]+mul[4];
                acc[4]<=acc[3]+mul[5];
                acc[5]<=acc[4]+mul[6];
                acc[6]<=acc[5]+mul[7];
                acc[7]<=acc[6]+mul[8];
                acc[8]<=acc[7]+mul[9];
                acc[9]<=acc[8]+mul[10];
                acc[10]<=acc[9]+mul[11];
                acc[11]<=acc[10]+mul[12];
                acc[12]<=acc[11]+mul[13];
                acc[13]<=acc[12]+mul[14];
                acc[14]<=acc[13]+mul[15];

                //done
            end
    end
endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////
/////////////////
// Company:
// Engineer:
//
// Create Date: 11/07/2015 03:03:25 PM
// Design Name:
// Module Name: i2s_receiver
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////
/////////////////
// takes AKM4117 IC receiver i2s and converts it into a 24bit
number for fpga
module i2s_tx_symm(
    input wire reset,
    input wire firclk,
    input wire bick,                // ready pulse on every
falling edge of BICK
    input wire lrck,
    output wire sdto,                // i2s data from ic receiver
    input wire ready,               // data ready
    input wire bitwait,
    input wire signed [31:0] data_left, data_right    // data
that goes to fpga
    );


    // additional registers and parameters
    reg [31:0] shiftreg_l = 32'b0;    // shift registers
    reg [31:0] shiftreg_r = 32'b0;

    assign sdto = lrck ? shiftreg_r[31] : shiftreg_l[31];
    always @(negedge firclk)
        begin
        if (reset)
            begin
            end
        else
            begin
            if (bick)
                begin
                if (ready)
                    begin
                    //sdto becomes right channel
                    shiftreg_l <= data_left;
                    shiftreg_r <= data_right;
                    end
                    else if (bitwait)
                    begin
                    //sdto becomes left channel
                    end
                    else if (lrck)
                    begin
                        shiftreg_r <= {shiftreg_r[30:0], 1'b0};
//shift out r channel when lrck high
                    end
                    else if (!lrck)
                    begin
                        shiftreg_l <= {shiftreg_l[30:0], 1'b0};
//shift out l channel when lrck low
                    end
                end
            end
        end
endmodule
```