Final Report for

CSEE 4840 Embedded Systems

Project: AudioSampler

Avik Dhupar (UNI: ad3910)
Spandan Das (UNI: sd3506)

Spring 2022

1. Introduction

We built a sound sampler and playback system which supports the following features:

1. MIDI input using a USB-MIDI Keyboard
2. Sample storage
3. Sample Playback

The system receives input from a USB-MIDI keyboard, which is processed by the HPS, and is forwarded through a memory mapped device to the FPGA fabric. The FPGA side loads a pre-saved sample through the HPS, which will then be processed on the FPGA itself, since it will allow us to optimize for signal processing and real-time audio reproduction. Once the FPGA processes the audio, it will also control and drive the WM8731 audio CODEC, which allows for easier playback through the integrated DAC. The DE1-SoC is packed with any necessary audio signal processing circuitry, which will allow us to simply load the audio buffer into the WM8731 which can then be output on the 3.5mm audio output jack.

2. System Overview

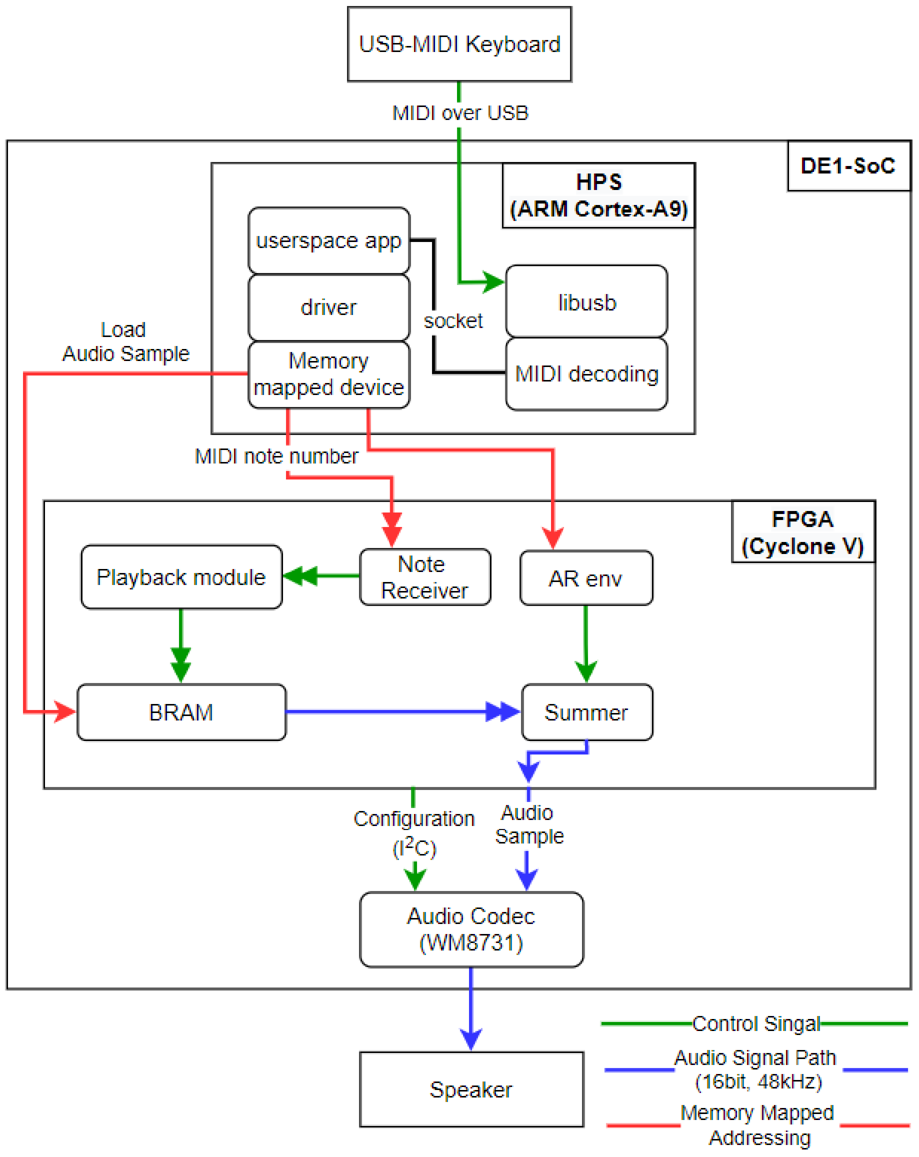
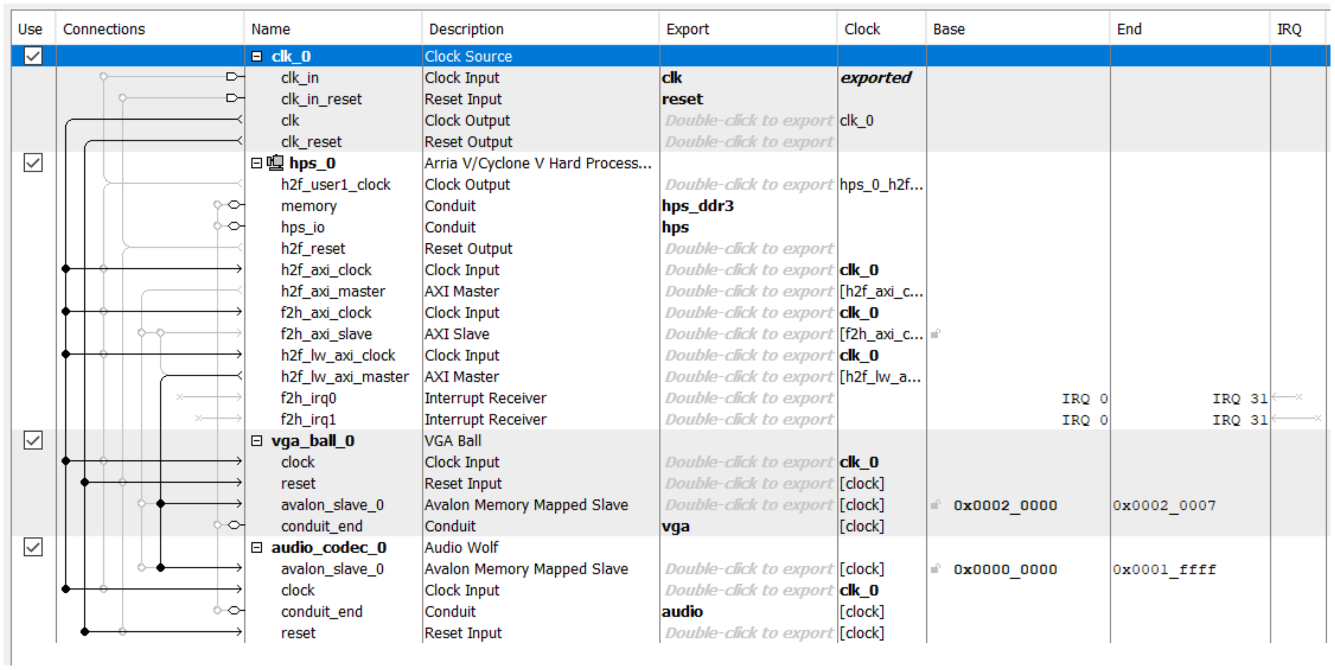


Fig. 1

Figure 1 above shows the updated system block diagram. Each component is described in the following sections.



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
✓		clk_0	Clock Source					
		clk_in	Clock Input	clk	exported			
		clk_in_reset	Reset Input	reset				
		clk	Clock Output	Double-click to export	clk_0			
		clk_reset	Reset Output	Double-click to export				
✓		hps_0	Arria V/Cyclone V Hard Process...					
		h2f_user1_clock	Clock Output	Double-click to export	hps_0_h2f...			
		memory	Conduit	hps_ddr3				
		hps_io	Conduit	hps				
		h2f_reset	Reset Output	Double-click to export				
		h2f_axi_clock	Clock Input	Double-click to export	clk_0			
		h2f_axi_master	AXI Master	Double-click to export	[h2f_axi_c...			
		f2h_axi_clock	Clock Input	Double-click to export	clk_0			
		f2h_axi_slave	AXI Slave	Double-click to export	[f2h_axi_c...			
		h2f_lw_axi_clock	Clock Input	Double-click to export	clk_0			
		h2f_lw_axi_master	AXI Master	Double-click to export	[h2f_lw_a...			
		f2h_irq0	Interrupt Receiver	Double-click to export		IRQ 0	IRQ 31	
		f2h_irq1	Interrupt Receiver	Double-click to export		IRQ 0	IRQ 31	
✓		vga_ball_0	VGA Ball					
		clock	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clock]			
		avalon_slave_0	Avalon Memory Mapped Slave	Double-click to export	[clock]	0x0002_0000	0x0002_0007	
		conduit_end	Conduit	vga	[clock]			
✓		audio_codec_0	Audio Wolf					
		avalon_slave_0	Avalon Memory Mapped Slave	Double-click to export	[clock]	0x0000_0000	0x0001_ffff	
		clock	Clock Input	Double-click to export	clk_0			
		conduit_end	Conduit	audio	[clock]			
		reset	Reset Input	Double-click to export	[clock]			

The figure above shows the system configuration, the avalon bus connections between the HPS and the hardware modules.

USB-MIDI

Musical Instrument Digital Interface is a well-established and old standard that describes the communication protocols, digital interface, and electrical connectors to connect a variety of electronic musical instruments [1]. MIDI supports some of the key features required for any electronic musical keyboard. For example, key-press, key-release, clock signal (tempo), note's pitch, velocity and aftertouch. Additionally, MIDI also allows for generic and specific 'control-change' messages, which can denote parameters such as portamento, modulation wheel values, and effect amount.

Despite the MIDI standard's age, most electronic musical instruments continue to support MIDI, over forty years after its advent. While today's computers do not directly support MIDI without specialized hardware such as sound cards or USB-MIDI converters, recent advancements have helped introduce support for MIDI over USB. In devices that support MIDI over USB, the underlying MIDI standard is maintained, i.e. the devices still speak the same protocol, however the MIDI data is encapsulated in a USB packet, and the electrical interface also follows the USB standard.

We reuse code from lab2 for USB-MIDI to capture the data being generated by the MIDI keyboard. This data is then enumerated in a limited range. For example, when the note 'G2' is pressed, this will be enumerated to 1. We intend to support 3 octaves, therefore, the note 'C#5' will be the last playable note, and will be enumerated to 31. This enumeration will be performed on the HPS side and the values will be passed on to the FPGA over a memory mapped device. On the FPGA side, these values will be used to compute parameters for the sample playback.

Sample Parameter Computation

Once the FPGA receives the value of the note being played, with 1 corresponding to G2, and 31 corresponding to C#5, we will perform computation to 'pitch shift' the sample in order to allow for 'playability' of the sample.
We stick with the 12-tone equal temperament scale, which means that an octave is divided into 12 parts, all of which are equal on a logarithmic scale, with a ratio equal to the 12th root of 2. [2]

Finally we were able to come up with the sample address undersampling using the Equal Temperament scale.

Sample Storage

We use the HPS/SD card for on-board storage of the samples. These samples are pushed to the FPGA fabric during system initialization. During runtime, these samples are stored in the BRAM of the FPGA. To enable polyphony, each element in a sample needs to be read as many times as the number of keys pressed. Since we support a polyphony of 3, and the FPGA doesn't support more than 2 port RAM, and we have enough memory budget, we simply save 3 copies of each sample in 3 different locations.

Attenuator & Summer

Since we are using more than one sample during playback, a simple summation would lead to an increase in the amplitude. In order to maintain a fixed, maximum output level, we use weighted amplitude of the two signals. This is implemented using attenuators.
For example:
For two input signals, W_1 and W_2 , the amplitude for each is $A_1 = A_2 = 1$
On adding both signals, amplitude will increase to $A_1 + A_2 = 2$. This is undesirable, and would lead to a much louder output from the speakers.

We fix this by using attenuators, who's gain is determined by the mixer inputs. If the mixer is set to 0.75 for W_1 and 0.75 for W_2, the output should be weighted based on these values, such that the total output never exceeds 1. In this case, the output would be $0.75 \times 0.66 + 0.75 \times 0.66 = 1.0$

With the final implementation, we were able to manage the mixing with an arithmetic summation and dividing the summed output by the number of notes used. The outputs are configured for a single note input-output, a two note input-output or a 3-note input and output. This depends on how the keys are pressed on the MIDI keyboard.

Audio CODEC

DE1-SoC ships with an onboard audio CODEC, the Wolfson Audio WM8731. These devices are used to encode or decode audio signals using DACs and ADCs respectively. The datasheet provides specific details of the CODEC, which are as follows. [4]

WM8731 is a low power stereo CODEC with an integrated headphone driver. It also supports line and mono microphone level audio inputs, programmable line level volume control, and a bias voltage output suitable for electret microphones. The WM8731 uses stereo 24-bit sigma delta ADCs and DACs. It supports audio input word lengths from 16-32 bits and sampling rates from 8kHz to 96kHz. We use 48KHz sampling rate for the DAC. The CODEC can be configured using 2 or 3 wire bus. We will use the 2 wire I2C bus in our project.

3. Algorithms

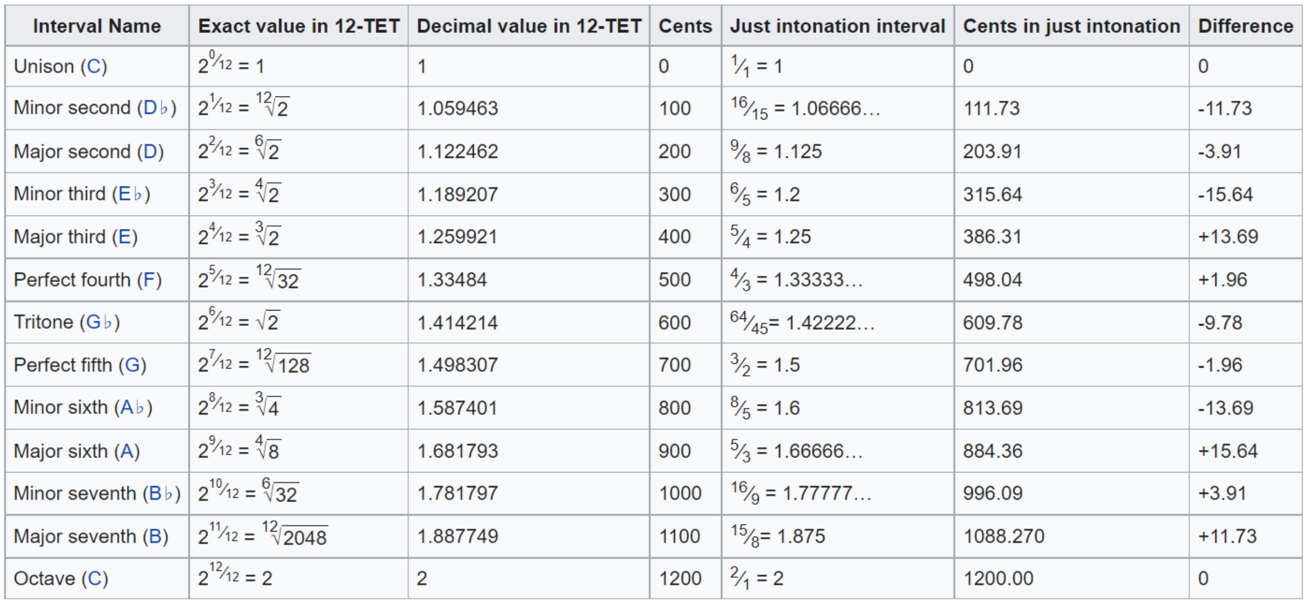
1. Playing with Undersampling

The samples are written to the embedded memory(BRAM) in the FPGA from the HPS. The samples are for a sine wave with a particular frequency. While playing back the samples from the audio codec, we provide the samples to the audio codec using Undersampling to create different frequencies from the stored one.

We implement this in our hardware by generating the appropriate addresses for the memory.

The frequency is selected by the input coming in from the MIDI keyboard (from the HPS). And according to the input the correct address is generated.

The address is calculated according to the following frequency ratios :-



Interval Name	Exact value in 12-TET	Decimal value in 12-TET	Cents	Just intonation interval	Cents in just intonation	Difference
Unison (C)	$2^{0/12} = 1$	1	0	$1/1 = 1$	0	0
Minor second (D♭)	$2^{1/12} = \sqrt[12]{2}$	1.059463	100	$16/15 = 1.06666\ldots$	111.73	-11.73
Major second (D)	$2^{2/12} = \sqrt[6]{2}$	1.122462	200	$9/8 = 1.125$	203.91	-3.91
Minor third (E♭)	$2^{3/12} = \sqrt[4]{2}$	1.189207	300	$6/5 = 1.2$	315.64	-15.64
Major third (E)	$2^{4/12} = \sqrt[3]{2}$	1.259921	400	$5/4 = 1.25$	386.31	+13.69
Perfect fourth (F)	$2^{5/12} = \sqrt[12]{32}$	1.33484	500	$4/3 = 1.33333\ldots$	498.04	+1.96
Tritone (G♭)	$2^{6/12} = \sqrt{2}$	1.414214	600	$64/45 = 1.42222\ldots$	609.78	-9.78
Perfect fifth (G)	$2^{7/12} = \sqrt[12]{128}$	1.498307	700	$3/2 = 1.5$	701.96	-1.96
Minor sixth (A♭)	$2^{8/12} = \sqrt[3]{4}$	1.587401	800	$8/5 = 1.6$	813.69	-13.69
Major sixth (A)	$2^{9/12} = \sqrt[4]{8}$	1.681793	900	$5/3 = 1.66666\ldots$	884.36	+15.64
Minor seventh (B♭)	$2^{10/12} = \sqrt[6]{32}$	1.781797	1000	$16/9 = 1.77777\ldots$	996.09	+3.91
Major seventh (B)	$2^{11/12} = \sqrt[12]{2048}$	1.887749	1100	$15/8 = 1.875$	1088.270	+11.73
Octave (C)	$2^{12/12} = 2$	2	1200	$2/1 = 2$	1200.00	0

Source: https://en.wikipedia.org/wiki/Equal_temperament

As the next frequency is some fraction of the previous frequency, we deal with fractional numbers in verilog through a naive implementation of fixed point.

To increment the address we calculated the step size for each frequency. This step size is a fraction value of 32 bits with MSB 16 bits representing the decimal part and the rest the fraction part. As the step gets incremented on every clock cycle and the actual address is limited to the last 16 bits [31:16], some of the samples do get passed on. This is how undersampling is implemented.

2. Reverse Playback

Reverse playback of an audio sample is achieved by playing back audio samples in the reverse order from the BRAM. For example, in regular playback, the samples are played starting 0, all the way up to 32,767. In reverse playback, samples are played back starting from 32767, down to 0, and so on in a loop.

3. Audio summation

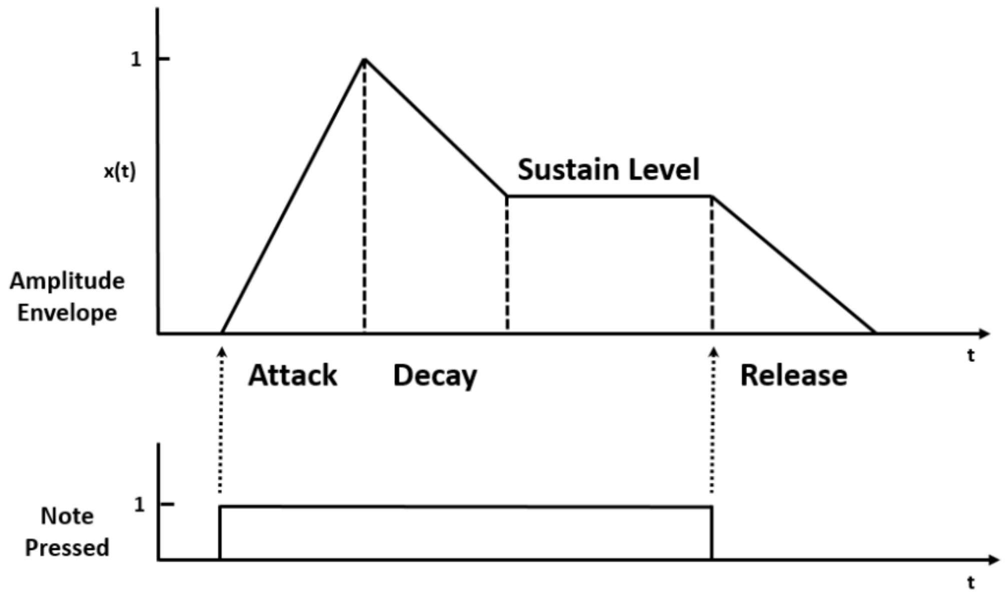
We store 3 such samples as described above stored in the memory. There can be 3 input keys coming from the MIDI keyboard simultaneously. It can also be one key or 2 keys pressed together. Depending on the number of keys being pressed the outputs are generated (read from the memory) and input into the summer module.

In the summer module, the arithmetic summation of the incoming signals is done and the amplitude is adjusted by dividing the summed output by the number of keys being pressed.

4. ADSR Amplitude Envelope Generator

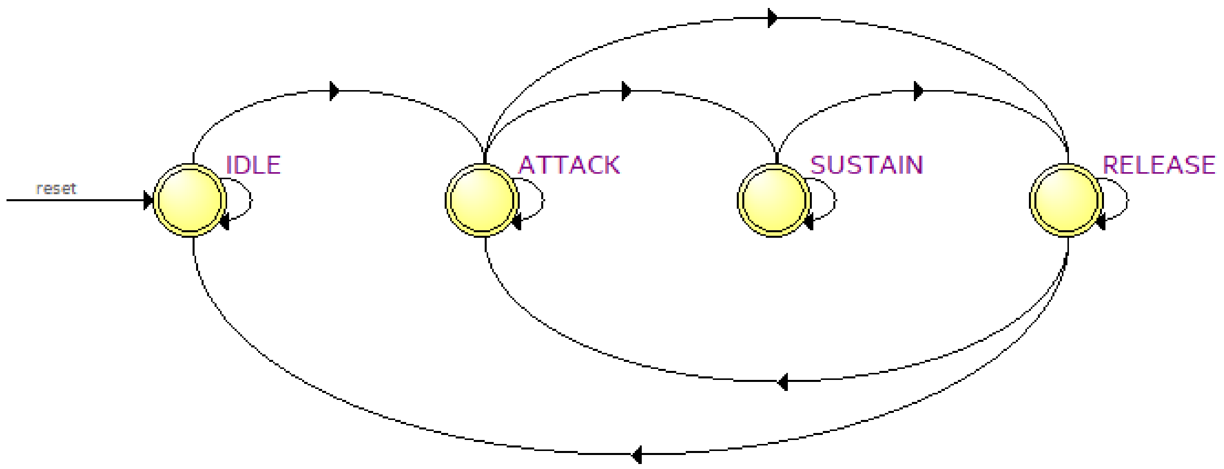
The ADSR (Attack-Decay-Sustain-Release) Amplitude Modulation Envelope is an important component of any modern synthesizer.This is used to modulate the amplitude of each note being pressed depending on how long or short the key is pressed down. This is used to model the physical response of an instrument such as a piano, and its change in volume over time. For example, the press of a note on a piano can be represented by a sharp increase in amplitude followed by a long, drawn out attenuation as the note fades. An ADSR envelope consists of four stages which give it its name. Attack, the first stage, begins the moment a note is pressed. During this phase, the amplitude of the signal rises from its idle state at zero to a maximum value of one. Once the maximum value is reached, the envelope enters the decay stage, during which the amplitude falls to a user defined level, called sustain. The envelope remains in the sustain stage with constant amplitude until the user stops playing the note. At this point, the release stage begins, during which the amplitude goes down to zero.

Below is the figure for the ADSR envelope response -

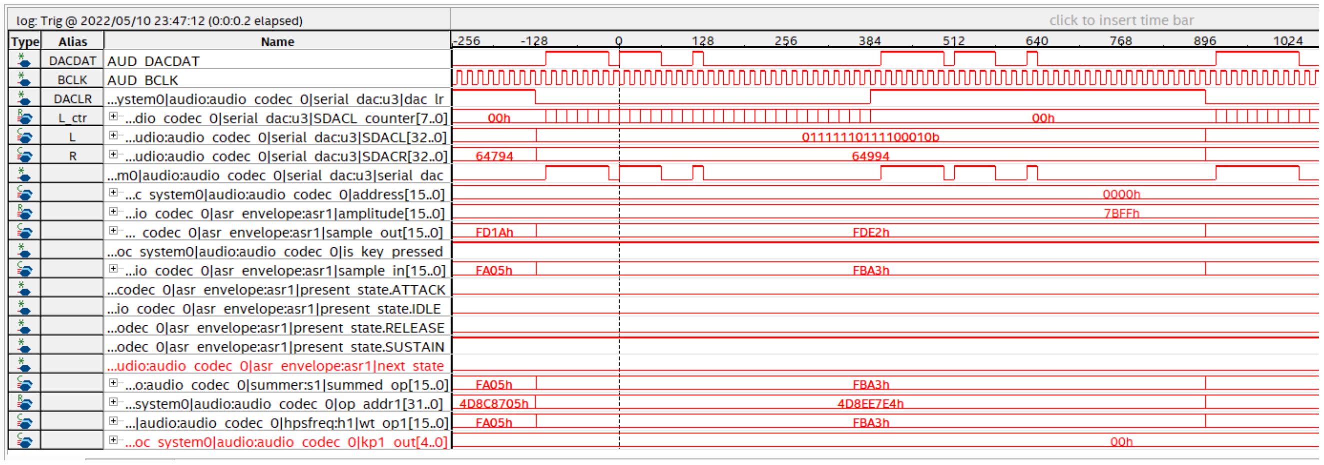


Source: https://web.wpi.edu/Images/CMS/ECE/Veilleux_Briggs_FPGA_Digital_Music_Synthesizer.pdf

We have implemented a similar envelope but have reduced the number of stages by eliminating decay. Following is the state diagram of our FSM, which includes the stages: IDLE, ATTACK, SUSTAIN and RELEASE.



Following is the snapshot from SignalTap depicting our state machine and its stages, along with other CODEC signals such as DACLR, DACDAT, BCLK.



log: Trig @ 2022/05/10 23:47:12 (0:0:0.2 elapsed)

click to insert time bar

Type	Alias	Name	-256 -128 0 128 256 384 512 640 768 896 1024
	DACDAT	AUD DACDAT	
	BCLK	AUD BCLK	
	DACLR	...ystem0|audio:audio codec 0|serial dac:u3|dac lr	
	L_ctr	...dio codec 0|serial dac:u3|SDACL counter[7..0]	00h 00h
	L	...udio:audio codec 0|serial dac:u3|SDACL[32..0]	0111111011110010b
	R	...udio:audio codec 0|serial dac:u3|SDACR[32..0]	64794 64994
		...m0|audio:audio codec 0|serial dac:u3|serial dac	
		...c system0|audio:audio codec 0|address[15..0]	0000h
		...io codec 0|asr envelope:asr1|amplitude[15..0]	7BFFh
		... codec 0|asr envelope:asr1|sample out[15..0]	FD1Ah FDE2h
		...oc system0|audio:audio codec 0|is key pressed	
		...io codec 0|asr envelope:asr1|sample in[15..0]	FA05h FBA3h
		...codec 0|asr envelope:asr1|present state.ATTACK	
		...io codec 0|asr envelope:asr1|present state.IDLE	
		...odec 0|asr envelope:asr1|present state.RELEASE	
		...odec 0|asr envelope:asr1|present state.SUSTAIN	
		...udio:audio codec 0|asr envelope:asr1|next state	
		...o:audio codec 0|summer:s1|summed op[15..0]	FA05h FBA3h
		...system0|audio:audio codec 0|op addr1[31..0]	4D8C8705h 4D8EE7E4h
		...|audio:audio codec 0|hpsfreq:h1|wt op1[15..0]	FA05h FBA3h
		...oc system0|audio:audio codec 0|kp1 out[4..0]	00h

4. Resource Budgets

Actual Memory Usage on FPGA

$$\underbrace{3}_{Sample\ Banks} \times \underbrace{16\,bit}_{Bit\ Depth} \times \underbrace{32768}_{Number\ of\ Samples} = 192\ KiB$$

Total = **192 KiB**

Total available embedded memory = 4450 Kbits = 552 KiB

While we initially planned to use 3 samples of size 48000, but quickly realized that we will occupy more than 60% of the available embedded memory. We now use only 3 samples of size 32768 each, and leave some memory for other overheads.

5. Hardware-Software Interface

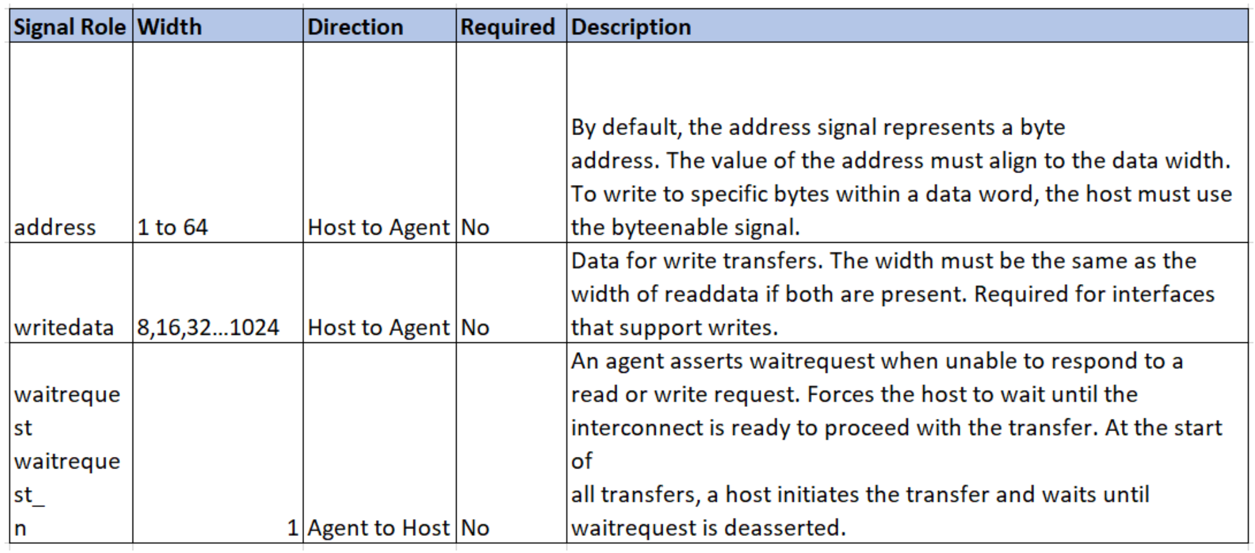
The primary hardware-software interfacing is done over the Avalon Bus.
We have introduced a memory-mapped device for the sample, and the HPS is able to directly write to this area and change the samples.
Further, we use the same memory mapped device to transfer parameters such as envelope ASR values, and key-presses received from the MIDI keyboard, over to the FPGA.

Avalon Bus:
Avalon interfaces simplify system design by allowing you to easily connect components in Intel® FPGA.

Avalon Memory Mapped Interface (Avalon-MM)—an address-based read/write interface typical of Host-Agent connections.

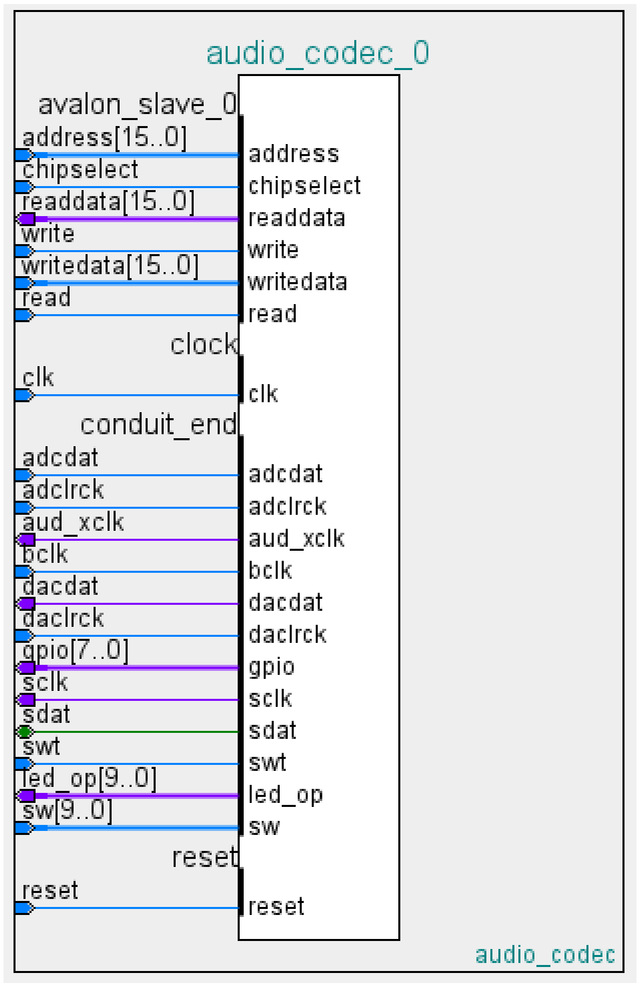
We can use Avalon Memory-Mapped (Avalon-MM) interfaces to implement read and write interfaces for Host and Agent components. The following are examples of components that typically include memory-mapped interfaces:

• Microprocessors
• Memories
• UARTs
• DMAs
• Timers



Signal Role	Width	Direction	Required	Description
address	1 to 64	Host to Agent	No	By default, the address signal represents a byte address. The value of the address must align to the data width. To write to specific bytes within a data word, the host must use the byteenable signal.
writedata	8,16,32...1024	Host to Agent	No	Data for write transfers. The width must be the same as the width of readdata if both are present. Required for interfaces that support writes.
waitreque st waitreque st_ n	1	Agent to Host	No	An agent asserts waitrequest when unable to respond to a read or write request. Forces the host to wait until the interconnect is ready to proceed with the transfer. At the start of all transfers, a host initiates the transfer and waits until waitrequest is deasserted.

The USB-MIDI keyboard connects to the HPS over a USB bus. MIDI devices typically show up as HID devices, and we plan to use existing HID device drivers to access the bytes being sent by the MIDI device. Once we capture these bytes, they will are converted to a numeric value between 0 to 31, since we support only 5 octaves ranging from G#2 to C5. These numeric values are written to the FPGA through a memory-mapped device. We reuse some of the existing code from Lab3 for this. While an existing driver may support various other MIDI controls, such as control change and modulation wheel, we will only send the note's numeric value, and note on/off signal.



The above figure represents the connection between the audio codec and the hps and the signals between the audio codec and the FPGA pins.

6. Simulations

The simulations below show what we initially planned to implement. However, further discussions with the professor led to development of a slightly different system, as described in the previous sections. The information below is left for legacy reasons. The Simulink simulations, however, are up-to-date, and were enormously helpful during the development.

Simulink

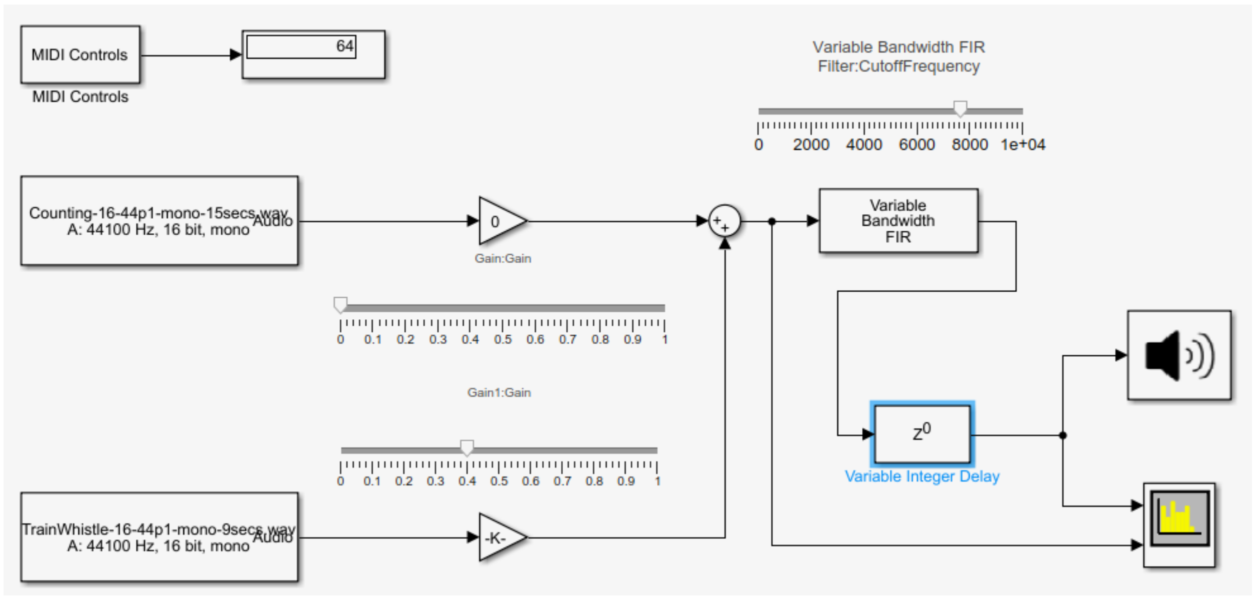

MIDI Controls
MIDI Controls
64
Variable Bandwidth FIR Filter:CutoffFrequency
0 2000 4000 6000 8000 1e+04
Counting-16-44p1-mono-15secs.wav
A: 44100 Hz, 16 bit, mono
Audio
0
Gain:Gain
Variable Bandwidth FIR
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
Gain1:Gain
z^0
Variable Integer Delay
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
TrainWhistle-16-44p1-mono-9secs.wav
A: 44100 Hz, 16 bit, mono
Audio
-K-

Fig 2: Simulink simulation diagram

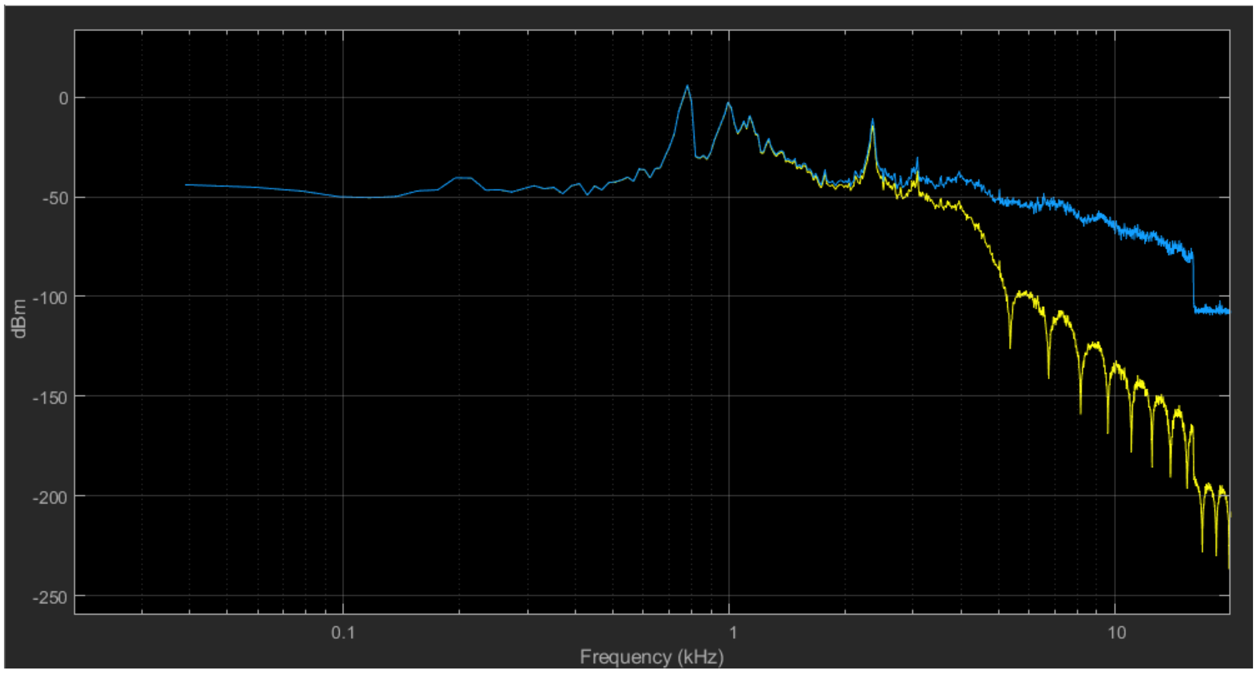


Fig 3: Spectrogram when Fcutoff = 2907 Hz

Legend:
Blue: Original Input
Yellow: Filtered and delayed output

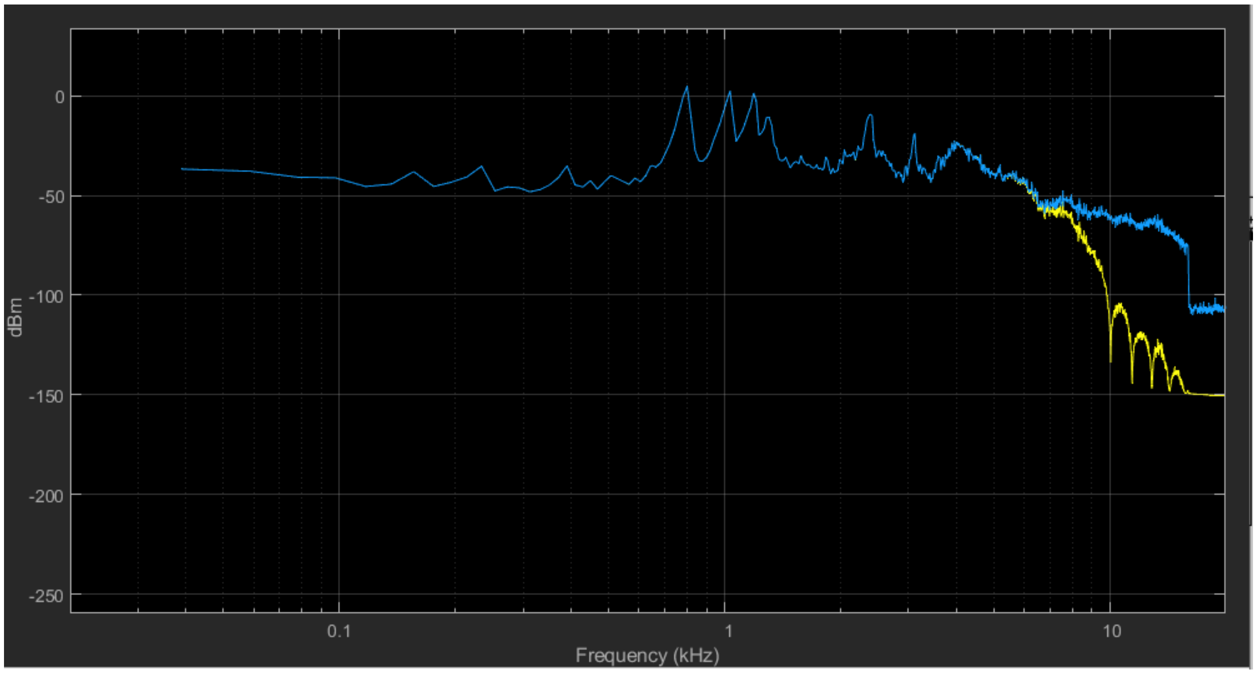


Fig 4: Spectrogram when Fcutoff = 7601 Hz

Legend:
Blue: Original Input
Yellow: Filtered and delayed output

VCV Rack

While the simulation in Simulink is sufficient to verify our design, our low-end Intel i5 systems clocked at 3.15GHz with 16GB RAM were not good enough to run the simulation without jitter. Therefore, to get a better understanding of how the system would sound, we created a VCVRack simulation.
VCVRack runs jitter free, and allows us to listen to the output. However, since it is focussed on music production, as opposed to scientific research, most components are (virtual) voltage-controlled, as is typical in the world of analog synthesizers. In our system, however, we will not have voltage control since the entire control and signal path is in the digital domain, except the CODEC output.

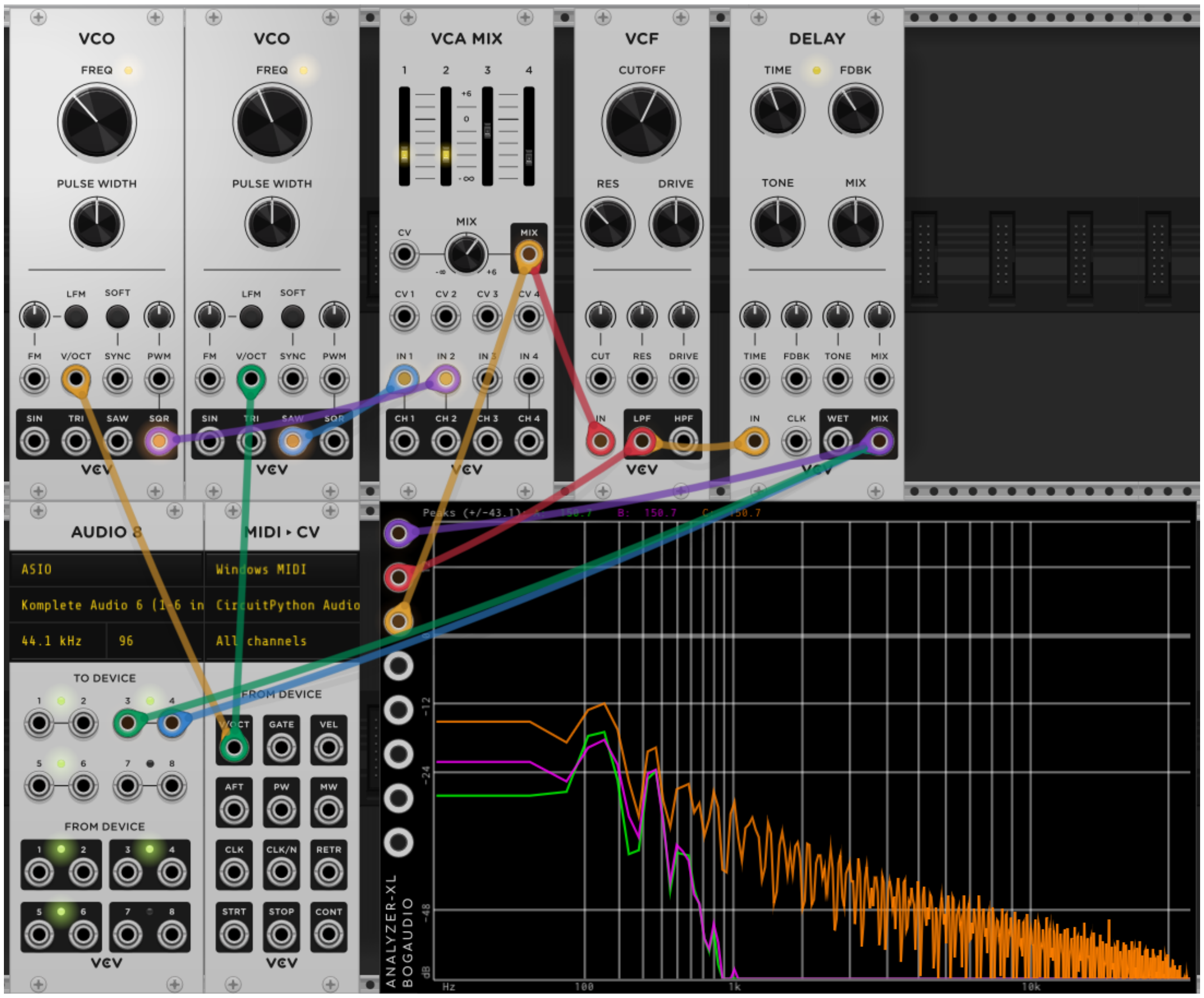
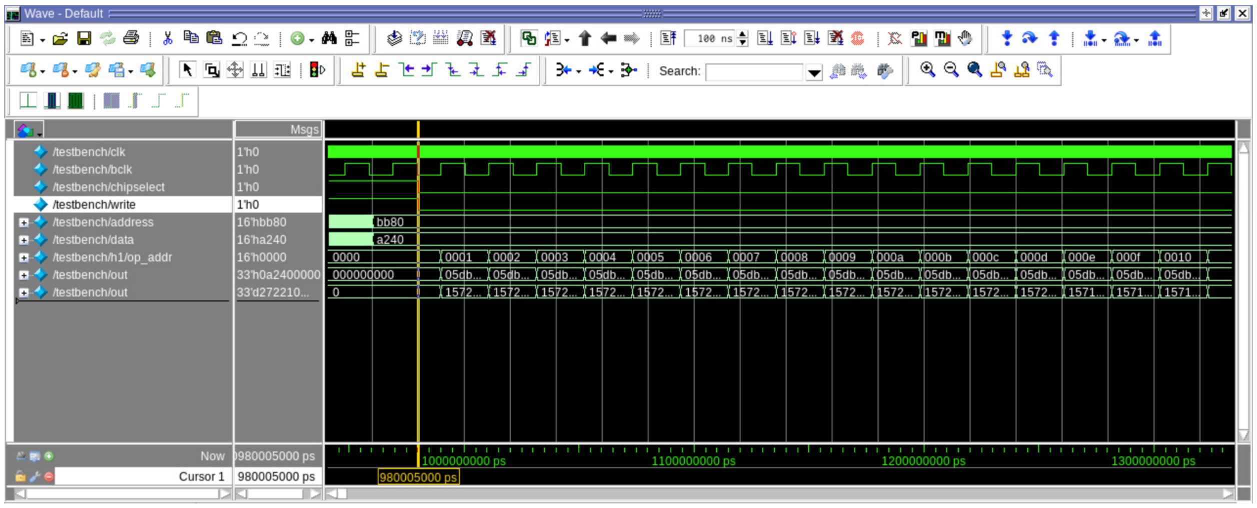


Fig 5: VCVRack simulation

Legend:
Orange: Unprocessed output of samples (Saw and square wave in this case)
Purple: Filtered output
Green: Filtered and Delayed Output

Modelsim

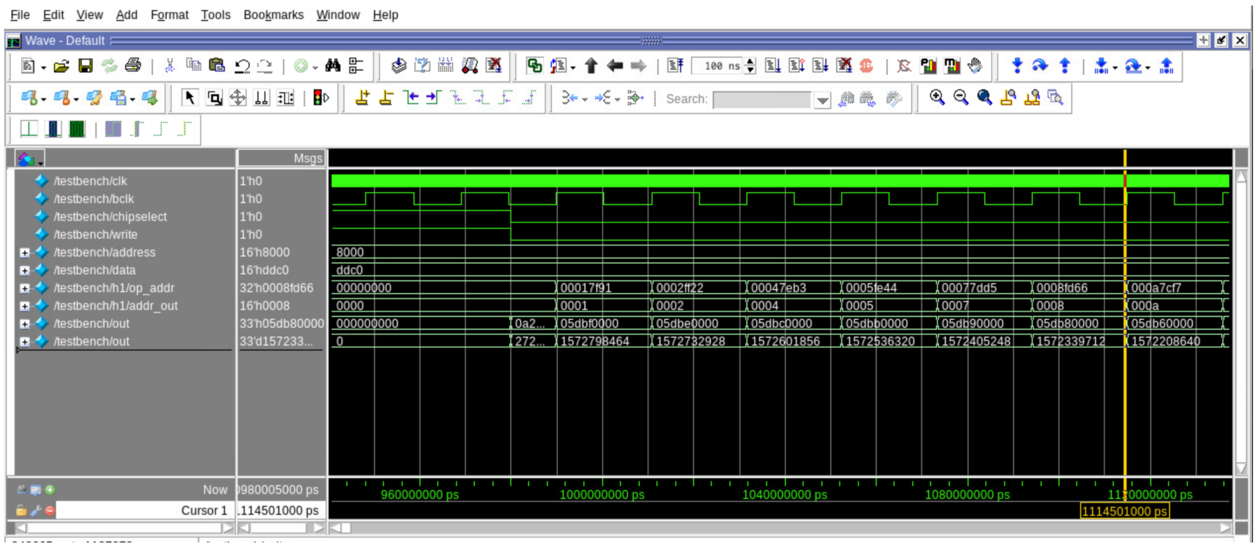
Following are the modelsim simulation results. We simulated the hardware on Modelsim for saving to the memory and reading out the samples from memory.



```
/testbench/clk              1'h0
/testbench/bclk             1'h0
/testbench/chipselect       1'h0
/testbench/write            1'h0
/testbench/address          16'hbb80          bb80
/testbench/data             16'ha240          a240
/testbench/h1/op_addr       16'h0000          0000  0001  0002  0003  0004  0005  0006  0007  0008  0009  000a  000b  000c  000d  000e  000f  0010
/testbench/out              33'h0a2400000     000000000  05db...  05db...  05db...  05db...  05db...  05db...  05db...  05db...  05db...  05db...  05db...  05db...  05db...  05db...  05db...  05db...
/testbench/out              33'd272210...     0  1572...  1572...  1572...  1572...  1572...  1572...  1572...  1572...  1572...  1572...  1572...  1572...  1572...  1571...  1571...  1571...

Now       980005000 ps      1000000000 ps   1100000000 ps   1200000000 ps   1300000000 ps
Cursor 1  980005000 ps      980005000 ps
```

Similarly we simulated reading the samples out from the memory for different frequencies of the stored audio sample.



```
/testbench/clk              1'h0
/testbench/bclk             1'h0
/testbench/chipselect       1'h0
/testbench/write            1'h0
/testbench/address          16'h8000          8000
/testbench/data             16'hddc0          ddc0
/testbench/h1/op_addr       32'h0008fd66      00000000  00017f91  0002ff22  00047eb3  0005fe44  00077dd5  0008fd66  000a7cf7
/testbench/h1/addr_out      16'h0008          0000  0001  0002  0004  0005  0007  0008  000a
/testbench/out              33'h05db80000     000000000  0a2...  05dbf0000  05dbe0000  05dbc0000  05dbb0000  05db90000  05db80000  05db60000
/testbench/out              33'd157233...     0  272...  1572798464  1572732928  1572601856  1572536320  1572405248  1572339712  1572208640

Now       980005000 ps      960000000 ps   1000000000 ps   1040000000 ps   1080000000 ps   1120000000 ps
Cursor 1  1114501000 ps     1114501000 ps
```

7. References

[1] https://en.wikipedia.org/wiki/MIDI
[2] https://en.wikipedia.org/wiki/Equal_temperament
[3] https://cdrdv2.intel.com/v1/dl/getContent/654277?explicitVersion=true
[4] https://www.digikey.com/en/datasheets/cirruslogicinc/cirrus-logic-inc-wm8731_v49
[5] https://en.wikipedia.org/wiki/Delay_(audio_effect)
[6] https://en.wikipedia.org/wiki/Clipping_(audio)
[7] https://www.elprocus.com/fir-filter-for-digital-signal-processing/

8. Appendix

8.1 Audio.v

```
module audio (
    address,
    chipselect,
    readdata,
    read,
    write,
    writedata,
    aud_xclk     , // clock 12. MHz?
    bclk         , // bit stream clock
    adclrck      , // left right clock ADC
    adcdat       , // data stream ADC
    daclrck      , // left right clock DAC
    dacdat       , // data stream DAC
    sclk         , // serial clock I2C
    sdat         , // serail data I2C
    swt          ,
    clk          ,
    reset,
    gpio,
    led_op,
    sw
);

input [15:0] address;
output reg [15:0] readdata;
input [15:0] writedata;
input chipselect;
input write;
input read;

input adcdat;
input swt;
input clk;
input bclk;

input reset;

input adclrck;
input daclrck;
```

```
inout sdat;

output aud_xclk;
output sclk;
output dacdat;
output[7:0] gpio;
output [9:0] led_op;
input [9:0] sw;


// THIS DOESN'T AFFECT ANYTHING
parameter n_samples = 2048;
// LEFT FOR QUARTUS, DON'T DELETE


I2C_programmer u1(
   .RESET(swt),                       //  clock enable
   .i2c_clk(clk),                     //  50 Mhz clk from DE1-SoC
   .I2C_SCLK(sclk),           // I2C clock 40K
   .TRN_END(TRN_END),
   .ACK(ACK),
   .ACK_enable(ACK_enable),
   .I2C_SDATA(sdat)           // bi directional serial data
);

//serial_adc u2 (
//    .serial_adc(adcdat),            // 32 bit serial in data
//    .SADCL(serial_lf),
//    .SADCR(serial_rt),
//    .adc_lr(adclrck),
//    .clk(clk),                      // 50 KHz clock
//    .enable(swt)                    // master reset
//);

serial_dac u3(
   .serial_dac(dacdat),               // 32 bit serial in data
   .SDACL(hps_op),
   .SDACR(hps_op),
   .dac_lr(daclrck),
   .clk(bclk),                        // 50 KHz clock
   .enable(swt)                       // master reset
);

hpsfreq h1(
    .clk(clk),
    .chipselect(chipselect),
    .write(write),
    .addr(address),
    .writedata(writedata),
    .readdata(readout),
    .read(read),
    .wt_op1(sine_op1),
    .wt_addr1(ad1),
    .wt_op2(sine_op2),
    .wt_addr2(ad2),
    .wt_op3(sine_op3),
    .wt_addr3(ad3)
);

key_parser k1(
    .clk(clk),
    .chipselect(chipselect),
    .write(write),
    .address(address),
    .writedata(writedata),
    .kp_1(kp1_out),
    .kp_2(kp2_out),
    .kp_3(kp3_out)
);

summer s1(
    .ip1(sine_op1),
    .ip2(sine_op2),
    .ip3(sine_op3),
    .summed_op(summed_op),
    .key_1(kp1_out_temp),
    .key_2(kp2_out_temp),
    .key_3(kp3_out_temp)
);

asr_envelope asr1(
    .clk(clk),
    .chipselect(chipselect),
    .write(write),
    .address(address),
    .writedata(writedata),
    .sample_in(summed_op),
    .pressed(is_key_pressed),
    .sample_out(sample_out)
);

wire is_key_pressed ;

assign is_key_pressed = (kp1_out > 0 || kp2_out > 0 || kp3_out > 0) ? 1'b1 : 1'b0;

// Unused but left in place
parameter S1 = 0;
parameter S2 = 0;

wire [15:0] readout;

wire [4:0] kp1_out;
wire [4:0] kp2_out;
wire [4:0] kp3_out;

// Data being written to DAC
// It expects 33bit
wire [32:0] hps_op;

wire [15:0] summed_op;
wire [15:0] sample_out;

// AD and op_addrs
wire [15:0] ad1 ;
assign ad1 = op_addr1[31:16];
reg [31:0] op_addr1 = 0;

wire [15:0] ad2 ;
assign ad2 = op_addr2[31:16];
reg [31:0] op_addr2 = 0;

wire [15:0] ad3 ;
assign ad3 = op_addr3[31:16];
reg [31:0] op_addr3 = 0;
// AD and op_addrs

wire [15:0] sine_op1;
wire [15:0] sine_op2;
wire [15:0] sine_op3;


// op from hpsfreq
```

```
assign hps_op = {1'b0, sample_out, {16'b0}};
// op from hpsfreq

wire [4:0] sel;
assign sel = {sw[0], sw[1], sw[2], sw[3], sw[4]};

always @(posedge clk)
   readdata <= readout;


// remember last keypress to support env Release
reg [4:0] prev_kp1_out;
reg [4:0] prev_kp2_out;
reg [4:0] prev_kp3_out;
reg [4:0] kp1_out_temp;
reg [4:0] kp2_out_temp;
reg [4:0] kp3_out_temp;
// remember last keypress to support env Release
always @(posedge clk) begin
   if (is_key_pressed) begin
      kp1_out_temp <= kp1_out;
      prev_kp1_out <= kp1_out;
      kp2_out_temp <= kp2_out;
      prev_kp2_out <= kp2_out;
      kp3_out_temp <= kp3_out;
      prev_kp3_out <= kp3_out;
   end
   else begin
      kp1_out_temp <= prev_kp1_out;
      kp2_out_temp <= prev_kp2_out;
      kp3_out_temp <= prev_kp3_out;
   end
end


reg [31:0] incr1 = 0;
reg [31:0] incr2 = 0;
reg [31:0] incr3 = 0;

// voodoo voodoo
always @(*) begin
   case(kp1_out_temp)
      5'd0  : incr1   = 32'b0;
      5'd1  : incr1   = 32'b0000_0000_0000_0001_0000_0000_0000_0000;
```

```
        5'd2  : incr1   = 32'b0000_0000_0000_0001_0000_1111_0011_1000;
        5'd3  : incr1   = 32'b0000_0000_0000_0001_0001_1111_0101_1001;
        5'd4  : incr1   = 32'b0000_0000_0000_0001_0011_0000_0110_1111;
        5'd5  : incr1   = 32'b0000_0000_0000_0001_0100_0010_1000_1010;
        5'd6  : incr1   = 32'b0000_0000_0000_0001_0101_0101_1011_1000;
        5'd7  : incr1   = 32'b0000_0000_0000_0001_0110_1010_0000_1001;
        5'd8  : incr1   = 32'b0000_0000_0000_0001_0111_1111_1001_0001;
        5'd9  : incr1   = 32'b0000_0000_0000_0001_1001_0110_0101_1111;
        5'd10 : incr1   = 32'b0000_0000_0000_0001_1010_1110_1000_1001;
        5'd11 : incr1   = 32'b0000_0000_0000_0001_1100_1000_0010_0011;
        5'd12 : incr1   = 32'b0000_0000_0000_0001_1110_0011_0100_0011;
        5'd13 : incr1   = 32'b0000_0000_0000_0010_0000_0000_0000_0000;
        5'd14 : incr1   = 32'b0000_0000_0000_0010_0001_1110_0111_0001;
        5'd15 : incr1   = 32'b0000_0000_0000_0010_0011_1110_1011_0011;
        5'd16 : incr1   = 32'b0000_0000_0000_0010_0110_0000_1101_1111;
        5'd17 : incr1   = 32'b0000_0000_0000_0010_1000_0101_0001_0100;
        5'd18 : incr1   = 32'b0000_0000_0000_0010_1010_1011_0111_0000;
        5'd19 : incr1   = 32'b0000_0000_0000_0010_1101_0100_0001_0011;
        5'd20 : incr1   = 32'b0000_0000_0000_0010_1111_1111_0010_0010;
        5'd21 : incr1   = 32'b0000_0000_0000_0011_0010_1100_1011_1111;
        5'd22 : incr1   = 32'b0000_0000_0000_0011_0101_1101_0001_0011;
        5'd23 : incr1   = 32'b0000_0000_0000_0011_1001_0000_0100_0111;
        5'd24 : incr1   = 32'b0000_0000_0000_0011_1100_0110_1000_0110;
        5'd25 : incr1   = 32'b0000_0000_0000_0100_0000_0000_0000_0000;
        5'd26 : incr1   = 32'b0000_0000_0000_0100_0011_1100_1110_0011;
        5'd27 : incr1   = 32'b0000_0000_0000_0100_0111_1101_0110_0110;
        5'd28 : incr1   = 32'b0000_0000_0000_0100_1100_0001_1011_1111;
        5'd29 : incr1   = 32'b0000_0000_0000_0101_0000_1010_0010_1000;
        5'd30 : incr1   = 32'b0000_0000_0000_0101_0101_0110_1110_0000;
        5'd31 : incr1   = 32'b0000_0000_0000_0101_1010_1000_0010_0111;

      default:
          incr1 = 32'b0;
    endcase
end
// voodoo voodoo

// voodoo voodoo
always @(*) begin
    case(kp2_out_temp)
        5'd0  : incr2   = 32'b0;
        5'd1  : incr2   = 32'b0000_0000_0000_0001_0000_0000_0000_0000;
        5'd2  : incr2   = 32'b0000_0000_0000_0001_0000_1111_0011_1000;
        5'd3  : incr2   = 32'b0000_0000_0000_0001_0001_1111_0101_1001;
        5'd4  : incr2   = 32'b0000_0000_0000_0001_0011_0000_0110_1111;
        5'd5  : incr2   = 32'b0000_0000_0000_0001_0100_0010_1000_1010;
        5'd6  : incr2   = 32'b0000_0000_0000_0001_0101_0101_1011_1000;
        5'd7  : incr2   = 32'b0000_0000_0000_0001_0110_1010_0000_1001;
        5'd8  : incr2   = 32'b0000_0000_0000_0001_0111_1111_1001_0001;
        5'd9  : incr2   = 32'b0000_0000_0000_0001_1001_0110_0101_1111;
        5'd10 : incr2   = 32'b0000_0000_0000_0001_1010_1110_1000_1001;
        5'd11 : incr2   = 32'b0000_0000_0000_0001_1100_1000_0010_0011;
        5'd12 : incr2   = 32'b0000_0000_0000_0001_1110_0011_0100_0011;
        5'd13 : incr2   = 32'b0000_0000_0000_0010_0000_0000_0000_0000;
        5'd14 : incr2   = 32'b0000_0000_0000_0010_0001_1110_0111_0001;
        5'd15 : incr2   = 32'b0000_0000_0000_0010_0011_1110_1011_0011;
        5'd16 : incr2   = 32'b0000_0000_0000_0010_0110_0000_1101_1111;
        5'd17 : incr2   = 32'b0000_0000_0000_0010_1000_0101_0001_0100;
        5'd18 : incr2   = 32'b0000_0000_0000_0010_1010_1011_0111_0000;
        5'd19 : incr2   = 32'b0000_0000_0000_0010_1101_0100_0001_0011;
        5'd20 : incr2   = 32'b0000_0000_0000_0010_1111_1111_0010_0010;
        5'd21 : incr2   = 32'b0000_0000_0000_0011_0010_1100_1011_1111;
        5'd22 : incr2   = 32'b0000_0000_0000_0011_0101_1101_0001_0011;
        5'd23 : incr2   = 32'b0000_0000_0000_0011_1001_0000_0100_0111;
        5'd24 : incr2   = 32'b0000_0000_0000_0011_1100_0110_1000_0110;
        5'd25 : incr2   = 32'b0000_0000_0000_0100_0000_0000_0000_0000;
        5'd26 : incr2   = 32'b0000_0000_0000_0100_0011_1100_1110_0011;
        5'd27 : incr2   = 32'b0000_0000_0000_0100_0111_1101_0110_0110;
        5'd28 : incr2   = 32'b0000_0000_0000_0100_1100_0001_1011_1111;
        5'd29 : incr2   = 32'b0000_0000_0000_0101_0000_1010_0010_1000;
        5'd30 : incr2   = 32'b0000_0000_0000_0101_0101_0110_1110_0000;
        5'd31 : incr2   = 32'b0000_0000_0000_0101_1010_1000_0010_0111;

      default:
         incr2 = 32'b0;
    endcase
end
// voodoo voodoo

// voodoo voodoo
always @(*) begin
    case(kp3_out_temp)
        5'd0  : incr3   = 32'b0;
        5'd1  : incr3   = 32'b0000_0000_0000_0001_0000_0000_0000_0000;
        5'd2  : incr3   = 32'b0000_0000_0000_0001_0000_1111_0011_1000;
        5'd3  : incr3   = 32'b0000_0000_0000_0001_0001_1111_0101_1001;
        5'd4  : incr3   = 32'b0000_0000_0000_0001_0011_0000_0110_1111;
        5'd5  : incr3   = 32'b0000_0000_0000_0001_0100_0010_1000_1010;
        5'd6  : incr3   = 32'b0000_0000_0000_0001_0101_0101_1011_1000;
        5'd7  : incr3   = 32'b0000_0000_0000_0001_0110_1010_0000_1001;
        5'd8  : incr3   = 32'b0000_0000_0000_0001_0111_1111_1001_0001;
        5'd9  : incr3   = 32'b0000_0000_0000_0001_1001_0110_0101_1111;
        5'd10 : incr3   = 32'b0000_0000_0000_0001_1010_1110_1000_1001;
        5'd11 : incr3   = 32'b0000_0000_0000_0001_1100_1000_0010_0011;
        5'd12 : incr3   = 32'b0000_0000_0000_0001_1110_0011_0100_0011;
        5'd13 : incr3   = 32'b0000_0000_0000_0010_0000_0000_0000_0000;
        5'd14 : incr3   = 32'b0000_0000_0000_0010_0001_1110_0111_0001;
        5'd15 : incr3   = 32'b0000_0000_0000_0010_0011_1110_1011_0011;
        5'd16 : incr3   = 32'b0000_0000_0000_0010_0110_0000_1101_1111;
        5'd17 : incr3   = 32'b0000_0000_0000_0010_1000_0101_0001_0100;
        5'd18 : incr3   = 32'b0000_0000_0000_0010_1010_1011_0111_0000;
        5'd19 : incr3   = 32'b0000_0000_0000_0010_1101_0100_0001_0011;
        5'd20 : incr3   = 32'b0000_0000_0000_0010_1111_1111_0010_0010;
        5'd21 : incr3   = 32'b0000_0000_0000_0011_0010_1100_1011_1111;
        5'd22 : incr3   = 32'b0000_0000_0000_0011_0101_1101_0001_0011;
        5'd23 : incr3   = 32'b0000_0000_0000_0011_1001_0000_0100_0111;
        5'd24 : incr3   = 32'b0000_0000_0000_0011_1100_0110_1000_0110;
        5'd25 : incr3   = 32'b0000_0000_0000_0100_0000_0000_0000_0000;
        5'd26 : incr3   = 32'b0000_0000_0000_0100_0011_1100_1110_0011;
        5'd27 : incr3   = 32'b0000_0000_0000_0100_0111_1101_0110_0110;
        5'd28 : incr3   = 32'b0000_0000_0000_0100_1100_0001_1011_1111;
        5'd29 : incr3   = 32'b0000_0000_0000_0101_0000_1010_0010_1000;
        5'd30 : incr3   = 32'b0000_0000_0000_0101_0101_0110_1110_0000;
        5'd31 : incr3   = 32'b0000_0000_0000_0101_1010_1000_0010_0111;

      default:
          incr3 = 32'b0;
    endcase
end
// voodoo voodoo


// access 3 memories
always @(negedge daclrck)  begin
    if(!write) begin
        if (!sw[0]) begin
            if (op_addr1[31:16] >= 32767) op_addr1 <= 0;
            else op_addr1 <= op_addr1 + incr1;
        end
        else begin
            if (op_addr1[31:16] == 0) op_addr1[31:16] <= 32767;
            else op_addr1 <= op_addr1 - incr1;
        end
    end
end

always @(negedge daclrck)  begin
    if(!write) begin
        if (!sw[0]) begin
            if (op_addr2[31:16] >= 32767) op_addr2 <= 0;
            else op_addr2 <= op_addr2 + incr2;
        end
        else begin
            if (op_addr2[31:16] == 0) op_addr2[31:16] <= 32767;
            else op_addr2 <= op_addr2 - incr2;
        end
    end
end

always @(negedge daclrck)  begin
    if(!write) begin
        if (!sw[0]) begin
            if (op_addr3[31:16] >= 32767) op_addr3 <= 0;
            else op_addr3 <= op_addr3 + incr3;
        end
        else begin
            if (op_addr3[31:16] == 0) op_addr3[31:16] <= 32767;
            else op_addr3 <= op_addr3 - incr3;
        end
    end
end



 //////////////////////////////////////////////////
 /// variables and parameter for state machines  ///
 //////////////////////////////////////////////////


 parameter clk_freq = 50000000;  // 50 Mhz
 parameter i2c_freq = 12288000;  // 12.288 Mhz

 wire[32:0] serial_lf;
 wire[32:0] serial_rt;

 //////////////////////////////////////////////////////
```

```
  ////// I2C clock (50 Mhz)used for DE1-SoC video in chip ///
  //////////////////////////////////////////////////////
 reg clk_by2 = 0;
 reg clk_by4 = 0;

always @(posedge clk)
    clk_by2 = ~clk_by2;

always @(posedge clk_by2)
    clk_by4 = ~clk_by4;

        wire sclk;
        wire sdat;

        ////////////////////////////////////////
        //// internal signals
        ////////////////////////////////////////
        wire ACK ;
        wire ACK_enable;
        wire [23:0] data_23;
        wire TRN_END;
        reg ctrl_clk;
        reg [15:0] clk_div;  // clock divider

        assign aud_xclk = clk_by4;// ctrl_clk;

        assign gpio[0] = clk_by4;//ctrl_clk;
        assign gpio[1] = bclk;
        assign gpio[2] = dacdat;
        assign gpio[3] = daclrck;
        assign gpio[4] = adcdat;
        assign gpio[5] = adclrck;
        assign gpio[6] = chipselect;
        assign gpio[7] = write;
        //assign readdata[7:0] = serial_lf[14:7];

endmodule
```

8.2 hpsfreq.v

```
module hpsfreq(
input    clk,
input    chipselect,
input    write,
input [15:0] addr,
input [15:0] writedata,
output [15:0] readdata,
input read,
output [15:0] wt_op1,
input [15:0] wt_addr1,
output [15:0] wt_op2,
input [15:0] wt_addr2,
output [15:0] wt_op3,
input [15:0] wt_addr3
);

// THIS DOESN'T AFFECT ANYTHING
parameter n_samples = 2048;
// LEFT FOR QUARTUS, DON'T DELETE

reg [15:0] wave_table_as [32767 : 0];
reg [15:0] wave_table_bs [32767 : 0];
reg [15:0] wave_table_cs [32767 : 0];

// readdata
always @(posedge clk) begin
   if (chipselect && read)
   readdata <= wave_table_as[addr];
end

// LOL sorry
// Receive data from HPS and store in local memory
always_ff @(posedge clk) begin
   if (chipselect && write) begin
      wave_table_as[addr] <= writedata;
      wave_table_bs[addr] <= writedata;
      wave_table_cs[addr] <= writedata;
   end
end

// Output wav table to DAC @ bclk=48khz
always_ff @(posedge clk) begin
```

```
   wt_op1 <=  wave_table_as[wt_addr1];
end

// Output wav table to DAC @ bclk=48khz
always_ff @(posedge clk) begin
   wt_op2 <=  wave_table_bs[wt_addr2];
end

// Output wav table to DAC @ bclk=48khz
always_ff @(posedge clk) begin
   wt_op3 <=  wave_table_cs[wt_addr3];
end

endmodule
```

8.3 key_parser.sv

```
module key_parser(
   input clk,
   input chipselect,
   input write,
   input [15:0]    address,
   input [15:0]    writedata,
   output [4:0] kp_1,
   output [4:0] kp_2,
   output [4:0] kp_3
);


always_ff @(posedge clk) begin
   if (chipselect && write) begin
      // if address is 0x8000 then
      // writedata contains midi keypress
      if (address == 16'h8000) begin
         kp_1 <= writedata[4:0];
         kp_2 <= writedata[9:5];
         kp_3 <= writedata[14:10];
      end
   end
end
endmodule
```

8.4 summer.sv

```
module summer(
   input signed [15:0] ip1,
   input signed [15:0] ip2,
   input signed [15:0] ip3,
   output signed [15:0] summed_op,
   input [4:0] key_1,
   input [4:0] key_2,
   input [4:0] key_3

   );


reg [17:0] summed_op_temp;


  always @(*) begin
     if (key_1 == 0 && key_2 == 0 && key_3 != 0) begin
        summed_op_temp = ip3;
        summed_op = summed_op_temp[15:0];
     end
     else if (key_1 == 0 && key_2 != 0 && key_3 == 0) begin
        summed_op_temp = ip2;
        summed_op = summed_op_temp[15:0];
     end
     else if (key_1 != 0 && key_2 == 0 && key_3 == 0) begin
        summed_op_temp = ip1;
        summed_op = summed_op_temp[15:0];
     end

     else if (key_1 == 0 && key_2 != 0 && key_3 != 0) begin
        summed_op_temp = (ip2 + ip3) / 2;
        summed_op = {summed_op_temp[16], summed_op_temp[14:0]};
     end
     else if (key_1 != 0 && key_2 == 0 && key_3 != 0) begin
        summed_op_temp = (ip1 + ip3) / 2;
        summed_op = {summed_op_temp[16], summed_op_temp[14:0]};
     end
     else if (key_1 != 0 && key_2 != 0 && key_3 == 0) begin
        summed_op_temp  = (ip1 + ip2) / 2;
        summed_op = {summed_op_temp[16], summed_op_temp[14:0]};
     end
```

```
      else begin
         summed_op_temp = (ip1 + ip2 + ip3) / 3 ;
         summed_op = {summed_op_temp[17], summed_op_temp[14:0]};
      end
   end


endmodule
```

8.5 asr_envelope.sv

```
module asr_envelope (
        input clk,
        input chipselect,
        input write,
        input [15:0] address,
        input [15:0] writedata,
        input logic signed [15:0] sample_in,
        input pressed,
        output logic [15:0] sample_out
    );

logic signed [15:0] amplitude;
typedef enum logic [2:0] {IDLE, ATTACK, SUSTAIN, RELEASE} statetype;

statetype next_state;
statetype present_state = IDLE;

logic [7:0] in_attack  = 8'd64;
logic [7:0] in_release = 8'd32;

always_ff @(posedge clk) begin
   if (chipselect && write) begin
      // if address is 0x8800 then
      // writedata contains attack/release values
      if (address == 16'h8800) begin
         in_attack  <= writedata[7:0];
         in_release <= writedata[15:8];
      end
   end
end

reg [30:0] ctr_attack;
always @(posedge clk) begin
   if (ctr_attack == 24'd5000) begin
      ctr_attack <= 24'd0;
   end
   else ctr_attack <= ctr_attack + 31'd1;
end


always_ff @(posedge clk) begin
   //if(!pressed) present_state <= IDLE;
```

```
    //else present_state <= next_state;
    present_state <= next_state;
end// adsr_envelope state

// State Transitions
always_comb begin
    case(present_state)
        IDLE : if(pressed) next_state = ATTACK;
        else next_state = IDLE;

        ATTACK: begin
            if(amplitude >= 16'd31743) next_state = SUSTAIN;
            else if (!pressed) next_state = RELEASE;
        else next_state = ATTACK;
        end

        SUSTAIN: if(!pressed) next_state = RELEASE;
        else next_state = SUSTAIN;

        RELEASE: begin
        if (amplitude <= 16'd1024 )
            next_state = IDLE;
        else if (amplitude > 16'd0 && pressed)
            next_state = ATTACK;
        else next_state = RELEASE;
        end
        default:
            next_state = IDLE;
endcase // present_state
end

// Output definitions
always_ff @(posedge clk ) begin
    case(present_state)
        IDLE :  amplitude <= 16'd0;

        ATTACK: if (ctr_attack == 24'd1000) amplitude <= amplitude + in_attack;

        SUSTAIN: amplitude <= 16'd31743;

        RELEASE: begin
                if (ctr_attack == 24'd1000) amplitude <= amplitude - in_release;
                else if (amplitude <= 16'd1024 ) amplitude <= 16'd0;
         end
    endcase // present_state
end

logic signed [31:0] temp_mult;

always_comb begin : proc_sample_in
    temp_mult = (amplitude * sample_in);
    sample_out  = temp_mult[31:16];
end

endmodule
```

8.6 serial_dac.v

```
module serial_dac(

   serial_dac,  // 24 bit serial in data
   SDACL,                        // left channel DAC
   SDACR,                        // right channel DAC
   dac_lr,                       // left right channel enable
   clk,                          // 50 KHz clock
   enable                        // master reset

);


input[32:0] SDACL;    // Stored ADC left
input[32:0] SDACR;    // Stored ADC right

output serial_dac;
input clk;
input enable;
input dac_lr;



//////////////////////////////////////
// internal register
//////////////////////////////////////

reg [7:0] SDACL_counter;      // counter for DAC left channel
reg [7:0] SDACR_counter;      // counter for DAC right channel

///////////////////////////////////////
// state machine for serial counter   //
///////////////////////////////////////

assign serial_dac = (dac_lr)? (serial_2) : (serial_1) ;
//assign serial_dac = (dac_lr)? (serial_1 | serial_2) : (serial_1 | serial_2) ;
reg serial_1;
reg serial_2;

always @(negedge enable or negedge clk) begin
   if  (!enable)
   begin
      SDACL_counter <= 7'b0; // reset left channel counter
```

```
    end
    else begin
        if (dac_lr)
            SDACL_counter <= 7'b0;
        else
            SDACL_counter <= SDACL_counter + 1; // left channel captures audio
    end
end

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

always @(negedge enable or negedge clk) begin
    if  (!enable)
    begin
        SDACR_counter <= 7'b0; // reset right channel counter
    end
    else begin
        if (!dac_lr)
            SDACR_counter <= 7'b0;
        else
            SDACR_counter <= SDACR_counter + 1; // right channel captures audio
    end
end

always @ (*) begin
//always @ (negedge enable or negedge clk) begin

    case (SDACL_counter)

        // msb first
        7'd0        : begin serial_1 = SDACL[32]  ;  end // bit 0 - start

        7'd1        : begin serial_1 = SDACL[31]  ;  end //  valid audio 31 left channel

        7'd2        : begin serial_1 = SDACL[30] ;  end // valid audio 30 left channel

        7'd3        : begin serial_1 = SDACL[29]  ;  end // valid audio 29 left channel

        7'd4        : begin serial_1 = SDACL[28]  ;  end // valid audio 28 left channel

        7'd5        : begin serial_1 = SDACL[27]  ;  end // valid audio 27 left channel

        7'd6        : begin serial_1 = SDACL[26]  ;  end // valid audio 26 left channel
```

```
7'd7        : begin serial_1 = SDACL[25]  ;  end // valid audio 25 left channel

7'd8        : begin serial_1 = SDACL[24]  ;  end // valid audio 24 left channel

7'd9        : begin serial_1 = SDACL[23]  ;  end // valid audio 23 left channel

7'd10       : begin serial_1 = SDACL[22]  ;  end // valid audio 22 left channel

7'd11       : begin serial_1 = SDACL[21]  ;  end // valid audio 21 left channel

7'd12       : begin serial_1 = SDACL[20]  ;  end // valid audio 20 left channel

7'd13       : begin serial_1 = SDACL[19]  ;  end // valid audio 19 left channel

7'd14       : begin serial_1 = SDACL[18]  ;  end // valid audio 18 left channel

7'd15       : begin serial_1 = SDACL[17]  ;  end // valid audio 17 left channel

7'd16       : begin serial_1 = SDACL[16]  ;  end // valid audio 16 left channel

7'd17       : begin serial_1 = SDACL[15]  ;  end // valid audio 15 left channel

7'd18       : begin serial_1 = SDACL[14]  ;  end // valid audio 14 left channel

7'd19       : begin serial_1 = SDACL[13]  ;  end // valid audio 13 left channel

7'd20       : begin serial_1 = SDACL[12] ;  end // valid audio 12 left channel

7'd21       : begin serial_1 = SDACL[11]  ;  end // valid audio 11 left channel

7'd22       : begin serial_1 = SDACL[10]  ;  end // valid audio 10 left channel

7'd23       : begin serial_1 = SDACL[9]  ;  end // valid audio 9 left channel

7'd24       : begin serial_1 = SDACL[8]  ;  end // valid audio 8 left channel

7'd25       : begin serial_1 = SDACL[7]  ;  end // valid audio 7 left channel

7'd26       : begin serial_1 = SDACL[6]  ;  end // valid audio 6 left channel

7'd27   : begin serial_1 = SDACL[5]  ;  end // valid audio 5 left channel

7'd28       : begin serial_1 = SDACL[4]  ;  end // valid audio 4 left channel
```

```
        7'd29       : begin serial_1 = SDACL[3]  ;  end // valid audio 3 left channel

        7'd30       : begin serial_1 = SDACL[2]  ;  end // valid audio 2 left channel

        7'd31       : begin serial_1 = SDACL[1]  ;  end // valid audio 1 left channel

        7'd32       : begin serial_1 = SDACL[0]  ;  end // valid audio 0 left channel

        default : begin serial_1 = 0; end

    endcase

end

always @ (*) begin


    case (SDACR_counter)

        // msb first
        7'd0        : begin serial_2 = SDACR[32] ;  end // bit 0 - start

        7'd1        : begin serial_2 = SDACR[31] ;  end // valid audio 31 right channel

        7'd2        : begin serial_2 = SDACR[30] ;  end // valid audio 30 right channel

        7'd3        : begin serial_2 = SDACR[29] ;  end // valid audio 29 right channel

        7'd4        : begin serial_2 = SDACR[28] ;  end // valid audio 28 right channel

        7'd5        : begin serial_2 = SDACR[27] ;  end // valid audio 27 right channel

        7'd6        : begin serial_2 = SDACR[26] ;  end // valid audio 26 right channel

        7'd7        : begin serial_2 = SDACR[25] ;  end // valid audio 25 right channel

        7'd8        : begin serial_2 = SDACR[24] ;  end // valid audio 24 right channel

        7'd9        : begin serial_2 = SDACR[23] ;  end // valid audio 23 right channel

        7'd10       : begin serial_2 = SDACR[22] ;  end // valid audio 22 right channel

        7'd11       : begin serial_2 = SDACR[21] ;  end // valid audio 21 right channel
    7'd12       : begin serial_2 = SDACR[20] ;  end // valid audio 20 right channel

    7'd13       : begin serial_2 = SDACR[19] ;  end // valid audio 19 right channel

    7'd14       : begin serial_2 = SDACR[18] ;  end // valid audio 18 right channel

    7'd15       : begin serial_2 = SDACR[17] ;  end // valid audio 17 right channel

    7'd16       : begin serial_2 = SDACR[16] ;  end // valid audio 16 right channel

    7'd17       : begin serial_2 = SDACR[15] ;  end // valid audio 15 right channel

    7'd18       : begin serial_2 = SDACR[14] ;  end // valid audio 14 right channel

    7'd19       : begin serial_2 = SDACR[13] ;  end // valid audio 13 right channel

    7'd20       : begin serial_2 = SDACR[12] ;  end // valid audio 12 right channel

    7'd21       : begin serial_2 = SDACR[11] ;  end // valid audio 11 right channel

    7'd22       : begin serial_2 = SDACR[10] ;  end // valid audio 10 right channel

    7'd23       : begin serial_2 = SDACR[9] ;  end // valid audio 9 right channel

    7'd24       : begin serial_2 = SDACR[8] ;  end // valid audio 8 right channel

    7'd25       : begin serial_2 = SDACR[7] ;  end // valid audio 7 right channel

    7'd26       : begin serial_2 = SDACR[6] ;  end // valid audio 6 right channel

    7'd27   : begin serial_2 = SDACR[5] ;  end // valid audio 5 right channel

    7'd28       : begin serial_2 = SDACR[4] ;  end // valid audio 4 right channel

    7'd29       : begin serial_2 = SDACR[3] ;  end // valid audio 3 right channel

    7'd30       : begin serial_2 = SDACR[2] ;  end // valid audio 2 right channel

    7'd31       : begin serial_2 = SDACR[1] ;  end // valid audio 1 right channel

    7'd32       : begin serial_2 = SDACR[0] ;  end // valid audio 0 right channel

    default : begin serial_2 = 0; end
endcase
```

```
end

endmodule
```

8.7 i2c_av_cfg.v

```
module i2c_av_cfg (
                        clk, // 50k clock
                        reset, // switch 0 on DE1-SoC board
                        mend,  // end of load
                        mstep,
                        SCLK,
                        mack,  // acknowledge reference bit
                        mgo,   // go transfer for each value
                        i2c_data // 23 bit register (8 command, 8 address, 8 data)

                        );

input           clk;
input           reset;
input           SCLK;

input   mend;
input   mack;

output[3:0] mstep;
output  mgo;
output[23:0] i2c_data;

// internal register

reg             mgo;
reg[23:0] i2c_data;
reg[15:0] LUT_data;
reg[5:0]  LUT_index;
reg[3:0]  mstep;


// LUT data size value for both audio and video register

parameter       LUT_size                = 10; // number of values loaded both audio and serial

//Audio register values  ( 9 in total)

parameter       set_lin_l       = 0;
parameter       set_lin_r       = 1;
parameter       set_head_l      = 2;
parameter       set_head_r      = 3;
parameter   a_path_cntrl = 4;
parameter   d_path_cntrl = 5;
parameter   power_on          = 6;
parameter   set_format    = 7;
parameter   sample_cntrl = 8;
parameter   set_active    = 9;


// config controllers [Audio]

always @ (posedge clk or negedge reset)

begin

        if (!reset)

                begin

                LUT_index     <= 0;
                mstep                 <= 0;
                mgo                   <= 0;

                end
                else

                begin

                if (LUT_index < LUT_size)
                begin

                case(mstep)
                        0: begin
                                        if (SCLK)

                                        i2c_data <= {8'h34,LUT_data};

                                        mgo <= 1;
                                        mstep <= 1;
                                end

                        1: begin

                                        if (mend)
                                        begin
                                                if (mack)
                                                mstep <= 2;
                                                else
                                                mstep <= 0;
                                                mgo <= 0;
                                                end
                                        end

                        2: begin

                                        LUT_index <= LUT_index + 1;
                                        mstep <= 0;

                                        end

                                endcase
                        end
                end
        end


always

begin

case ( LUT_index)

// audio config values

set_lin_l               : LUT_data <= 16'h001a;
set_lin_r               : LUT_data <= 16'h021a;
set_head_l              : LUT_data <= 16'h0479;
set_head_r              : LUT_data <= 16'h0679;
//a_path_cntrl  : LUT_data <= 16'h08fc; // Mic in
//a_path_cntrl  : LUT_data <= 16'h08f8; // Linein
a_path_cntrl    : LUT_data <= 16'h0810; // Disable bypass, enable DAC in
d_path_cntrl    : LUT_data <= 16'h0a06;
power_on                        : LUT_data <= 16'h0c00;
//set_format            : LUT_data <= 16'h0e4a; // 24bit
set_format              : LUT_data <= 16'h0e42; // 16bit depth
sample_cntrl    : LUT_data <= 16'h1000;
set_active              : LUT_data <= 16'h1201;
```

```
endcase

end

endmodule
```

8.8 I2C_programmer.v

```
//----------------------------------------------------
// Design Name : encoder_using_case
// File Name   : encoder_using_case.v
// Function    : Encoder using Case
// Coder       : Fred Aulich
// last update Dec 19th 2013 changed the way program clock is set up SCLK  and SDO
// March 2016 updated for audio only set master mode
//----------------------------------------------------
module I2C_programmer(


RESET,                          //  2 MHz clock enable
i2c_clk,                //  27 Mhz clk from DE1-SoC
I2C_SCLK,               // I2C clock 40K
TRN_END,
ACK,
ACK_enable,
I2C_SDATA               // bi directional serial data
);

output I2C_SCLK;
output  TRN_END;
output ACK;
output  ACK_enable;
inout   I2C_SDATA;

input i2c_clk;
input RESET; // 27 MHz clock enable



/////////////////////////////
//// internal register ////////
/////////////////////////////

reg [15:0] mi2c_clk_div;  // clock divider
reg [23:0] SD;                    // serial shift register I2C data
reg [6:0] SD_COUNTER;             // shift counter

reg mi2c_ctrl_clk;                // output 40k clock
```

```
 reg SCLK;                                              // serial clock variable
 reg TRN_END;                                           // end of each serial load
 reg SDO;
 reg CLOCK;
/////////////////////////////////////////////////////////////////
/// module for loading values to the video and audio register  /////
/////////////////////////////////////////////////////////////////

 i2c_av_cfg  u0 (
                         .clk(mi2c_ctrl_clk),
                         .reset(RESET),
                         .mend(TRN_END),
                         .mack(ACK),
                         .mgo(GO),
                         .SCLK(SCLK),
                         .mstep(mstep),
                         .i2c_data(data_23)
                         );

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

wire I2C_SCLK = !TRN_END ? SCLK : 1;
 wire I2C_SDATA = ACK_enable ? SDO : 1'bz;

 reg ACK_enable;
 reg ACK1,ACK2,ACK3;
 wire ACK = ACK1 | ACK2 | ACK3;
 wire [23:0] data_23;
 wire GO;
 wire[3:0] mstep;

 parameter clk_freq = 50000000;  // 50 Mhz
 parameter i2c_freq = 40000;     // 40 Khz



 ////////////////////////////////////////////////////
 ////// I2C clock (50 Mhz)used for DE1-SoC video in chip ///
 ////////////////////////////////////////////////////

  always @ (posedge i2c_clk or negedge RESET)
  begin
                 if (!RESET)
		begin
			mi2c_clk_div <= 0;
			mi2c_ctrl_clk <= 0;
		end

		else

		begin

			if (mi2c_clk_div <  (clk_freq/i2c_freq) )  // keeps dividing until reaches desired
frequency
			mi2c_clk_div <= mi2c_clk_div + 1;

			else
			begin
					mi2c_clk_div <= 0;
					mi2c_ctrl_clk <= ~mi2c_ctrl_clk;
			end
		end
	end

	always @(negedge RESET or posedge CLOCK) begin
	if (!RESET) SD_COUNTER =  7 'b1111111;
	else begin
	if (GO==0)
		SD_COUNTER=0;
		else
		if ((SD_COUNTER < 7 'b1110111) & (TRN_END ==0)) SD_COUNTER =
SD_COUNTER + 1;
	end
	end

 /////////////////////////////////////////
 // counter to serially shift bits into  ///
 // I2C data register                    ///
 /////////////////////////////////////////

 always @ (negedge RESET or posedge CLOCK) begin

 if (!RESET) begin ACK1 = 0; ACK2 = 0; ACK3 = 0; TRN_END = 1;  ACK_enable = 1; SCLK = 1; SDO
= 1; end
 else
 case (SD_COUNTER)
```

```
        7'd0    : begin ACK1 = 0; ACK2 = 0; ACK3 = 0; TRN_END = 0; SDO = 1; SCLK = 1;
ACK_enable =1; end
                7'd1    : begin SD= (data_23); SDO = 0; end
                // begin load
                // slave address
                7'd2    : begin SDO = SD[23]; SCLK = 0; end
                7'd3    : begin SDO = SD[23]; SCLK = 1; end
                7'd4    : begin SDO = SD[23]; SCLK = 1; end
                7'd5    : begin SDO = SD[23]; SCLK = 0; end

                7'd6    : begin SDO = SD[22]; SCLK = 0; end
                7'd7    : begin SDO = SD[22]; SCLK = 1; end
                7'd8    : begin SDO = SD[22]; SCLK = 1; end
                7'd9  : begin SDO = SD[22]; SCLK = 0; end

                7'd10   : begin SDO = SD[21]; SCLK = 0; end
                7'd11   : begin SDO = SD[21]; SCLK = 1; end
                7'd12   : begin SDO = SD[21]; SCLK = 1; end
                7'd13 : begin SDO = SD[21]; SCLK = 0; end

                7'd14   : begin SDO = SD[20]; SCLK = 0; end
                7'd15 : begin SDO = SD[20]; SCLK = 1; end
                7'd16 : begin SDO = SD[20]; SCLK = 1; end
                7'd17 : begin SDO = SD[20]; SCLK = 0; end

                7'd18 : begin SDO = SD[19]; SCLK = 0; end
                7'd19 : begin SDO = SD[19]; SCLK = 1; end
                7'd20 : begin SDO = SD[19]; SCLK = 1; end
                7'd21   : begin SDO = SD[19]; SCLK = 0; end

                7'd22 : begin SDO = SD[18]; SCLK = 0; end
                7'd23 : begin SDO = SD[18]; SCLK = 1; end
                7'd24 : begin SDO = SD[18]; SCLK = 1; end
                7'd25 : begin SDO = SD[18]; SCLK = 0; end

                7'd26   : begin SDO = SD[17]; SCLK = 0; end
                7'd27   : begin SDO = SD[17]; SCLK = 1; end
                7'd28   : begin SDO = SD[17]; SCLK = 1; end
                7'd29 : begin SDO = SD[17]; SCLK = 0; end

                7'd30 : begin SDO = SD[16]; SCLK = 0; end
                7'd31 : begin SDO = SD[16]; SCLK = 1; end
```

```
7'd32 : begin SDO = SD[16]; SCLK = 1; end
7'd33   : begin SDO = SD[16]; SCLK = 0; end
// acknowledge cycle begin
7'd34 : begin SDO = 0; SCLK = 0; end
7'd35 : begin SDO = 0; SCLK = 1; end
7'd36 : begin SDO = 0; SCLK = 1; end
7'd37 : begin ACK1=I2C_SDATA; SCLK = 0; ACK_enable = 0 ;  end // tri state
7'd38 : begin ACK1=I2C_SDATA; SCLK = 0; ACK_enable = 0 ; end
7'd39 : begin SDO = 0; SCLK = 0; ACK_enable = 1 ; end
// sub address
7'd40   : begin SDO = SD[15]; SCLK = 0; end
7'd41   : begin SDO = SD[15]; SCLK = 1; end
7'd42   : begin SDO = SD[15]; SCLK = 1; end
7'd43   : begin SDO = SD[15]; SCLK = 0; end

7'd44   : begin SDO = SD[14]; SCLK = 0; end
7'd45   : begin SDO = SD[14]; SCLK = 1; end
7'd46   : begin SDO = SD[14]; SCLK = 1; end
7'd47 : begin SDO = SD[14]; SCLK = 0; end

7'd48   : begin SDO = SD[13]; SCLK = 0; end
7'd49   : begin SDO = SD[13]; SCLK = 1; end
7'd50   : begin SDO = SD[13]; SCLK = 1; end
7'd51 : begin SDO = SD[13]; SCLK = 0; end

7'd52   : begin SDO = SD[12]; SCLK = 0; end
7'd53 : begin SDO = SD[12]; SCLK = 1; end
7'd54 : begin SDO = SD[12]; SCLK = 1; end
7'd55 : begin SDO = SD[12]; SCLK = 0; end

7'd56 : begin SDO = SD[11]; SCLK = 0; end
7'd57 : begin SDO = SD[11]; SCLK = 1; end
7'd58 : begin SDO = SD[11]; SCLK = 1; end
7'd59   : begin SDO = SD[11]; SCLK = 0; end

7'd60 : begin SDO = SD[10]; SCLK = 0; end
7'd61 : begin SDO = SD[10]; SCLK = 1; end
7'd62 : begin SDO = SD[10]; SCLK = 1; end
7'd63 : begin SDO = SD[10]; SCLK = 0; end

7'd64   : begin SDO = SD[9]; SCLK = 0; end
7'd65   : begin SDO = SD[9]; SCLK = 1; end
7'd66   : begin SDO = SD[9]; SCLK = 1; end
7'd67 : begin SDO = SD[9]; SCLK = 0; end
```

```
7'd68 : begin SDO = SD[8]; SCLK = 0; end
7'd69 : begin SDO = SD[8]; SCLK = 1; end
7'd70 : begin SDO = SD[8]; SCLK = 1; end
7'd71   : begin SDO = SD[8]; SCLK = 0; end
// acknowledge cycle begin
7'd72 : begin SDO = 0; SCLK = 0; end
7'd73 : begin SDO = 0; SCLK = 1; end
7'd74 : begin SDO = 0; SCLK = 1; end
7'd75 : begin ACK1=I2C_SDATA; SCLK = 0; ACK_enable = 0 ;  end // tri state
7'd76 : begin ACK1=I2C_SDATA; SCLK = 0; ACK_enable = 0 ; end
7'd77 : begin SDO = 0; SCLK = 0; ACK_enable = 1 ; end
// data
7'd78   : begin SDO = SD[7]; SCLK = 0; end
7'd79   : begin SDO = SD[7]; SCLK = 1; end
7'd80   : begin SDO = SD[7]; SCLK = 1; end
7'd81   : begin SDO = SD[7]; SCLK = 0; end

7'd82   : begin SDO = SD[6]; SCLK = 0; end
7'd83   : begin SDO = SD[6]; SCLK = 1; end
7'd84   : begin SDO = SD[6]; SCLK = 1; end
7'd85 : begin SDO = SD[6]; SCLK = 0; end

7'd86   : begin SDO = SD[5]; SCLK = 0; end
7'd87   : begin SDO = SD[5]; SCLK = 1; end
7'd88   : begin SDO = SD[5]; SCLK = 1; end
7'd89 : begin SDO = SD[5]; SCLK = 0; end

7'd90   : begin SDO = SD[4]; SCLK = 0; end
7'd91 : begin SDO = SD[4]; SCLK = 1; end
7'd92 : begin SDO = SD[4]; SCLK = 1; end
7'd93 : begin SDO = SD[4]; SCLK = 0; end

7'd94 : begin SDO = SD[3]; SCLK = 0; end
7'd95 : begin SDO = SD[3]; SCLK = 1; end
7'd96 : begin SDO = SD[3]; SCLK = 1; end
7'd97   : begin SDO = SD[3]; SCLK = 0; end

7'd98 : begin SDO = SD[2]; SCLK = 0; end
7'd99 : begin SDO = SD[2]; SCLK = 1; end
7'd100: begin SDO = SD[2]; SCLK = 1; end
7'd101: begin SDO = SD[2]; SCLK = 0; end

7'd102: begin SDO = SD[1]; SCLK = 0; end
```

```
                7'd103: begin SDO = SD[1]; SCLK = 1; end
                7'd104: begin SDO = SD[1]; SCLK = 1; end
                7'd105: begin SDO = SD[1]; SCLK = 0; end

                7'd106: begin SDO = SD[0]; SCLK = 0; end
                7'd107: begin SDO = SD[0]; SCLK = 1; end
                7'd108: begin SDO = SD[0]; SCLK = 1; end
                7'd109: begin SDO = SD[0]; SCLK = 0; end
                // acknowledge cycle begin
                7'd110: begin SDO = 0; SCLK = 0; end
                7'd111: begin SDO = 0; SCLK = 1; end
                7'd112: begin SDO = 0; SCLK = 1; end
                7'd113: begin ACK1=I2C_SDATA; SCLK = 0; ACK_enable = 0 ;  end // tri state
                7'd114: begin ACK1=I2C_SDATA; SCLK = 0; ACK_enable = 0 ; end
                7'd115: begin SDO = 0; SCLK = 0; ACK_enable = 1 ; end
                // stop
                7'd116: begin SCLK = 1'b0; SDO = 1'b0; end
                7'd117: SCLK = 1'b1;
                7'd118: begin SDO = 1'b1; TRN_END = 1'b1; end

                endcase
                end

////////////////////////////////////////////
// directing signals to GPIO bus //////////
////////////////////////////////////////////

always          CLOCK <= mi2c_ctrl_clk;

endmodule
```

8.9 soc_system_top.sv

```
// ============================================================================
// Copyright (c) 2013 by Terasic Technologies Inc.
// ============================================================================
//
// Modified 2019 by Stephen A. Edwards
//
// Permission:
//
//   Terasic grants permission to use and modify this code for use
//   in synthesis for all Terasic Development Boards and Altera
//   Development Kits made by Terasic.  Other use of this code,
//   including the selling ,duplication, or modification of any
//   portion is strictly prohibited.
//
// Disclaimer:
//
//   This VHDL/Verilog or C/C++ source code is intended as a design
//   reference which illustrates how these types of functions can be
//   implemented.  It is the user's responsibility to verify their
//   design for consistency and functionality through the use of
//   formal verification methods.  Terasic provides no warranty
//   regarding the use or functionality of this code.
//
// ===========================================================================
//
//  Terasic Technologies Inc

//  9F., No.176, Sec.2, Gongdao 5th Rd, East Dist, Hsinchu City, 30070. Taiwan
//
//
//                    web: http://www.terasic.com/
//                    email: support@terasic.com
module soc_system_top(

 ///////// ADC /////////
 inout        ADC_CS_N,
 output       ADC_DIN,
 input        ADC_DOUT,
 output       ADC_SCLK,

 ///////// AUD /////////
 input        AUD_ADCDAT,
inout        AUD_ADCLRCK,
inout        AUD_BCLK,
output       AUD_DACDAT,
inout        AUD_DACLRCK,
output       AUD_XCK,

///////// CLOCK2 /////////
input        CLOCK2_50,

///////// CLOCK3 /////////
input        CLOCK3_50,

///////// CLOCK4 /////////
input        CLOCK4_50,

///////// CLOCK /////////
input        CLOCK_50,

///////// DRAM /////////
output [12:0] DRAM_ADDR,
output [1:0]  DRAM_BA,
output       DRAM_CAS_N,
output       DRAM_CKE,
output       DRAM_CLK,
output       DRAM_CS_N,
inout [15:0]  DRAM_DQ,
output       DRAM_LDQM,
output       DRAM_RAS_N,
output       DRAM_UDQM,
output       DRAM_WE_N,

///////// FAN /////////
output       FAN_CTRL,

///////// FPGA /////////
output       FPGA_I2C_SCLK,
inout        FPGA_I2C_SDAT,

///////// GPIO /////////
//inout [35:0]  GPIO_0,
output [35:0]  GPIO_0,
inout [35:0]  GPIO_1,

///////// HEX0 /////////
```

```
output [6:0]  HEX0,

///////// HEX1 /////////
output [6:0]  HEX1,

///////// HEX2 /////////
output [6:0]  HEX2,

///////// HEX3 /////////
output [6:0]  HEX3,

///////// HEX4 /////////
output [6:0]  HEX4,

///////// HEX5 /////////
output [6:0]  HEX5,

///////// HPS /////////
inout         HPS_CONV_USB_N,
output [14:0] HPS_DDR3_ADDR,
output [2:0]  HPS_DDR3_BA,
output        HPS_DDR3_CAS_N,
output        HPS_DDR3_CKE,
output        HPS_DDR3_CK_N,
output        HPS_DDR3_CK_P,
output        HPS_DDR3_CS_N,
output [3:0]  HPS_DDR3_DM,
inout [31:0]  HPS_DDR3_DQ,
inout [3:0]   HPS_DDR3_DQS_N,
inout [3:0]   HPS_DDR3_DQS_P,
output        HPS_DDR3_ODT,
output        HPS_DDR3_RAS_N,
output        HPS_DDR3_RESET_N,
input         HPS_DDR3_RZQ,
output        HPS_DDR3_WE_N,
output        HPS_ENET_GTX_CLK,
inout         HPS_ENET_INT_N,
output        HPS_ENET_MDC,
inout         HPS_ENET_MDIO,
input         HPS_ENET_RX_CLK,
input [3:0]   HPS_ENET_RX_DATA,
input         HPS_ENET_RX_DV,
output [3:0]  HPS_ENET_TX_DATA,
output        HPS_ENET_TX_EN,
inout         HPS_GSENSOR_INT,
inout         HPS_I2C1_SCLK,
inout         HPS_I2C1_SDAT,
inout         HPS_I2C2_SCLK,
inout         HPS_I2C2_SDAT,
inout         HPS_I2C_CONTROL,
inout         HPS_KEY,
inout         HPS_LED,
inout         HPS_LTC_GPIO,
output        HPS_SD_CLK,
inout         HPS_SD_CMD,
inout [3:0]   HPS_SD_DATA,
output        HPS_SPIM_CLK,
input         HPS_SPIM_MISO,
output        HPS_SPIM_MOSI,
inout         HPS_SPIM_SS,
input         HPS_UART_RX,
output        HPS_UART_TX,
input         HPS_USB_CLKOUT,
inout [7:0]   HPS_USB_DATA,
input         HPS_USB_DIR,
input         HPS_USB_NXT,
output        HPS_USB_STP,

///////// IRDA /////////
input         IRDA_RXD,
output        IRDA_TXD,

///////// KEY /////////
input [3:0]   KEY,

///////// LEDR /////////
output [9:0]  LEDR,

///////// PS2 /////////
inout         PS2_CLK,
inout         PS2_CLK2,
inout         PS2_DAT,
inout         PS2_DAT2,

///////// SW /////////
input [9:0]   SW,

///////// TD /////////
```

```
 input         TD_CLK27,
 input [7:0]   TD_DATA,
 input         TD_HS,
 output        TD_RESET_N,
 input         TD_VS,


 ///////// VGA /////////
 output [7:0]  VGA_B,
 output        VGA_BLANK_N,
 output        VGA_CLK,
 output [7:0]  VGA_G,
 output        VGA_HS,
 output [7:0]  VGA_R,
 output        VGA_SYNC_N,
 output        VGA_VS
);

  soc_system soc_system0(
    .clk_clk                    ( CLOCK_50 ),
    .reset_reset_n              ( 1'b1 ),

    .hps_ddr3_mem_a             ( HPS_DDR3_ADDR ),
    .hps_ddr3_mem_ba            ( HPS_DDR3_BA ),
    .hps_ddr3_mem_ck            ( HPS_DDR3_CK_P ),
    .hps_ddr3_mem_ck_n          ( HPS_DDR3_CK_N ),
    .hps_ddr3_mem_cke           ( HPS_DDR3_CKE ),
    .hps_ddr3_mem_cs_n          ( HPS_DDR3_CS_N ),
    .hps_ddr3_mem_ras_n         ( HPS_DDR3_RAS_N ),
    .hps_ddr3_mem_cas_n         ( HPS_DDR3_CAS_N ),
    .hps_ddr3_mem_we_n          ( HPS_DDR3_WE_N ),
    .hps_ddr3_mem_reset_n       ( HPS_DDR3_RESET_N ),
    .hps_ddr3_mem_dq            ( HPS_DDR3_DQ ),
    .hps_ddr3_mem_dqs           ( HPS_DDR3_DQS_P ),
    .hps_ddr3_mem_dqs_n         ( HPS_DDR3_DQS_N ),
    .hps_ddr3_mem_odt           ( HPS_DDR3_ODT ),
    .hps_ddr3_mem_dm            ( HPS_DDR3_DM ),
    .hps_ddr3_oct_rzqin         ( HPS_DDR3_RZQ ),

    .hps_hps_io_emac1_inst_TX_CLK ( HPS_ENET_GTX_CLK ),
    .hps_hps_io_emac1_inst_TXD0   ( HPS_ENET_TX_DATA[0] ),
    .hps_hps_io_emac1_inst_TXD1   ( HPS_ENET_TX_DATA[1] ),
    .hps_hps_io_emac1_inst_TXD2   ( HPS_ENET_TX_DATA[2] ),
    .hps_hps_io_emac1_inst_TXD3   ( HPS_ENET_TX_DATA[3] ),
.hps_hps_io_emac1_inst_RXD0   ( HPS_ENET_RX_DATA[0] ),
.hps_hps_io_emac1_inst_MDIO   ( HPS_ENET_MDIO  ),
.hps_hps_io_emac1_inst_MDC    ( HPS_ENET_MDC   ),
.hps_hps_io_emac1_inst_RX_CTL ( HPS_ENET_RX_DV ),
.hps_hps_io_emac1_inst_TX_CTL ( HPS_ENET_TX_EN ),
.hps_hps_io_emac1_inst_RX_CLK ( HPS_ENET_RX_CLK ),
.hps_hps_io_emac1_inst_RXD1   ( HPS_ENET_RX_DATA[1]  ),
.hps_hps_io_emac1_inst_RXD2   ( HPS_ENET_RX_DATA[2]  ),
.hps_hps_io_emac1_inst_RXD3   ( HPS_ENET_RX_DATA[3]  ),

.hps_hps_io_sdio_inst_CMD     ( HPS_SD_CMD         ),
.hps_hps_io_sdio_inst_D0      ( HPS_SD_DATA[0]      ),
.hps_hps_io_sdio_inst_D1      ( HPS_SD_DATA[1]      ),
.hps_hps_io_sdio_inst_CLK     ( HPS_SD_CLK          ),
.hps_hps_io_sdio_inst_D2      ( HPS_SD_DATA[2]      ),
.hps_hps_io_sdio_inst_D3      ( HPS_SD_DATA[3]      ),

.hps_hps_io_usb1_inst_D0      ( HPS_USB_DATA[0]     ),
.hps_hps_io_usb1_inst_D1      ( HPS_USB_DATA[1]     ),
.hps_hps_io_usb1_inst_D2      ( HPS_USB_DATA[2]     ),
.hps_hps_io_usb1_inst_D3      ( HPS_USB_DATA[3]     ),
.hps_hps_io_usb1_inst_D4      ( HPS_USB_DATA[4]     ),
.hps_hps_io_usb1_inst_D5      ( HPS_USB_DATA[5]     ),
.hps_hps_io_usb1_inst_D6      ( HPS_USB_DATA[6]     ),
.hps_hps_io_usb1_inst_D7      ( HPS_USB_DATA[7]     ),
.hps_hps_io_usb1_inst_CLK     ( HPS_USB_CLKOUT      ),
.hps_hps_io_usb1_inst_STP     ( HPS_USB_STP         ),
.hps_hps_io_usb1_inst_DIR     ( HPS_USB_DIR         ),
.hps_hps_io_usb1_inst_NXT     ( HPS_USB_NXT         ),

.hps_hps_io_spim1_inst_CLK    ( HPS_SPIM_CLK  ),
.hps_hps_io_spim1_inst_MOSI   ( HPS_SPIM_MOSI ),
.hps_hps_io_spim1_inst_MISO   ( HPS_SPIM_MISO ),
.hps_hps_io_spim1_inst_SS0    ( HPS_SPIM_SS   ),

.hps_hps_io_uart0_inst_RX     ( HPS_UART_RX     ),
.hps_hps_io_uart0_inst_TX     ( HPS_UART_TX     ),

.hps_hps_io_i2c0_inst_SDA     ( HPS_I2C1_SDAT     ),
.hps_hps_io_i2c0_inst_SCL     ( HPS_I2C1_SCLK     ),

.hps_hps_io_i2c1_inst_SDA     ( HPS_I2C2_SDAT     ),
.hps_hps_io_i2c1_inst_SCL     ( HPS_I2C2_SCLK     ),
      .hps_hps_io_gpio_inst_GPIO09  ( HPS_CONV_USB_N ),
      .hps_hps_io_gpio_inst_GPIO35  ( HPS_ENET_INT_N ),
      .hps_hps_io_gpio_inst_GPIO40  ( HPS_LTC_GPIO ),

      .hps_hps_io_gpio_inst_GPIO48  ( HPS_I2C_CONTROL ),
      .hps_hps_io_gpio_inst_GPIO53  ( HPS_LED ),
      .hps_hps_io_gpio_inst_GPIO54  ( HPS_KEY ),
      .hps_hps_io_gpio_inst_GPIO61  ( HPS_GSENSOR_INT ),

      .vga_r (VGA_R),
      .vga_g (VGA_G),
      .vga_b (VGA_B),
      .vga_clk (VGA_CLK),
      .vga_hs (VGA_HS),
      .vga_vs (VGA_VS),
      .vga_blank_n (VGA_BLANK_N),
      .vga_sync_n (VGA_SYNC_N),

      .audio_aud_xclk (AUD_XCK),
      .audio_bclk (AUD_BCLK),
      .audio_adclrck (AUD_ADCLRCK),
      .audio_adcdat (AUD_ADCDAT),
      .audio_daclrck (AUD_DACLRCK),
      .audio_dacdat (AUD_DACDAT),
      .audio_sclk (FPGA_I2C_SCLK),
      .audio_sdat (FPGA_I2C_SDAT),
      .audio_sw(SW),
      .audio_swt (1'b1),
      .audio_gpio (GPIO_0[7:0]),
      .audio_led_op(LEDR)
//     .audio_hex0(HEX0),
//     .audio_hex1(HEX1),
//     .audio_hex2(HEX2),
//     .audio_hex3(HEX3),
//     .audio_hex4(HEX4),
//     .audio_hex5(HEX5)

  );



   assign swt = 1;
   // The following quiet the "no driver" warnings for output
   // pins and should be removed if you use any of these peripherals
   assign ADC_CS_N = SW[1] ? SW[0] : 1'bZ;
   assign ADC_DIN = SW[0];
   assign ADC_SCLK = SW[0];

//   assign AUD_ADCLRCK = SW[1] ? SW[0] : 1'bZ;
//   assign AUD_BCLK = SW[1] ? SW[0] : 1'bZ;
//   assign AUD_DACDAT = SW[0];
//   assign AUD_DACLRCK = SW[1] ? SW[0] : 1'bZ;
//   assign AUD_XCK = SW[0];

   assign DRAM_ADDR = { 13{ SW[0] } };
   assign DRAM_BA = { 2{ SW[0] } };
   assign DRAM_DQ = SW[1] ? { 16{ SW[0] } } : 16'bZ;
   assign {DRAM_CAS_N, DRAM_CKE, DRAM_CLK, DRAM_CS_N,
           DRAM_LDQM, DRAM_RAS_N, DRAM_UDQM, DRAM_WE_N} = { 8{SW[0]} };

   assign FAN_CTRL = SW[0];

//   assign FPGA_I2C_SCLK = SW[0];
//   assign FPGA_I2C_SDAT = SW[1] ? SW[0] : 1'bZ;

   assign GPIO_0[35:8] = SW[1] ? { 28{ SW[0] } } : 28'bZ;
   assign GPIO_1 = SW[1] ? { 36{ SW[0] } } : 36'bZ;

//   assign HEX0 = { 7{ SW[1] } };
//   assign HEX1 = { 7{ SW[2] } };
//   assign HEX2 = { 7{ SW[3] } };
//   assign HEX3 = { 7{ SW[4] } };
//   assign HEX4 = { 7{ SW[5] } };
//   assign HEX5 = { 7{ SW[6] } };

   assign IRDA_TXD = SW[0];

   // assign LEDR = { 10{SW[7]} };

   assign PS2_CLK = SW[1] ? SW[0] : 1'bZ;
   assign PS2_CLK2 = SW[1] ? SW[0] : 1'bZ;
   assign PS2_DAT = SW[1] ? SW[0] : 1'bZ;
   assign PS2_DAT2 = SW[1] ? SW[0] : 1'bZ;

   assign TD_RESET_N = SW[0];
```

```
endmodule
```

8.10 hello.c

```
/*
 * Userspace program that communicates with the vga_ball device driver
 * through ioctls
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * scokets ref: https://www.geeksforgeeks.org/socket-programming-cc/
 */

#include <stdio.h>
#include "vga_ball.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include<netinet/in.h>

#include<stdlib.h>
#include<string.h>

#define SERVER_HOST "127.0.0.1"
#define SERVER_PORT 42000

// special address
// 1000_0000_0000_0000
// FPGA interprets writedata as
// MIDI note values when data
// is written to this address
#define MIDI_WRITEADDR 0x8000


// special address for env data
// 0100_0000_0000_0000
#define ENV_WRITEADDR 0x8800

#define NSAMPLES 32768
signed int wav_table[NSAMPLES];
```

```
int vga_ball_fd;

#define MAX_LINESIZE 15

int line_num = 0;
char line [MAX_LINESIZE+1];
char * next_field;

char *ptr ;

int i = 0;

int load_wave_table(char * fn){
   FILE * f = fopen (fn, "r");
   while (1){
      fgets(line, MAX_LINESIZE, f);

      if(feof(f))
         break;

      line_num++;
      next_field = strtok(line, " \n");
      while (next_field != NULL)
      {
         short v = (int)(strtol(next_field, &ptr, 10) * 0.5);
         wav_table[i++] = v;
         //       printf("Line %d:%s \t int:%d\n", line_num, next_field, v);
         next_field = strtok(NULL, " \n");
      }
   }
   fclose(f);
   return 0;
}



/* Read and print the background color */
void print_background_color(const vga_ball_color_t *c) {
   vga_ball_arg_t vla;
   vla.background = *c;
   if (ioctl(vga_ball_fd, VGA_BALL_READ_BACKGROUND, &vla)) {
      perror("ioctl(VGA_BALL_READ_BACKGROUND) failed");
```

```
        return;
    }
    //vla.background = *c;
    printf("%d %02x \n",
            vla.background.sin, vla.background.loc);
}

/* Set the background color */
void set_background_color(const vga_ball_color_t *c)
{
    vga_ball_arg_t vla;
    vla.background = *c;
    if (ioctl(vga_ball_fd, VGA_BALL_WRITE_BACKGROUND, &vla)) {
        perror("ioctl(VGA_BALL_SET_BACKGROUND) failed");
        return;
    }
}


void write_sample_data(){
    vga_ball_color_t c;

    for (int i = 0; i< NSAMPLES; i++){
        c.sin = wav_table[i];
        c.loc = i;
        set_background_color(&c);
    }

}


void socket_server_loop(){
    printf("\nStarting socket server\n");

    // data to write, and address to write
    vga_ball_color_t c;

    int server_fd, new_socket, valread;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);
    char buffer[32] = {0};

    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
```

```
{
    perror("socket failed");
    exit(EXIT_FAILURE);
}

if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
            &opt, sizeof(opt)))
{
    perror("setsockopt");
    exit(EXIT_FAILURE);
}
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_addr.s_addr = inet_addr(SERVER_HOST);
address.sin_port = htons( SERVER_PORT );


if (bind(server_fd, (struct sockaddr *)&address,
            sizeof(address))<0)
{
    perror("bind failed");
    exit(EXIT_FAILURE);
}

if (listen(server_fd, 3) < 0)
{
    perror("listen failure");
    exit(EXIT_FAILURE);
}

if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
                (socklen_t*)&addrlen))<0)
{
    perror("accept failure");
    exit(EXIT_FAILURE);
}

for (;;){
    // Read socket
    valread = read( new_socket , buffer, 32);
    //printf("%s\n",buffer );

    // packed struct for storing keypress
    // values in 16 bit wide val
```

```
        typedef union {
            uint16_t val;
            struct{
                unsigned int na: 5;
                unsigned int nb: 5;
                unsigned int nc: 5;
                unsigned int x: 1;
            };
        }packed_un_t;

        packed_un_t u1 = {.val = 0};

        short vals[3] = {0};
        int i=0;

        // Extract comma sep values and convert to int
        const char s[2] = ",";
        char *token;

        token = strtok(buffer, s);
        while(token!= NULL){
            int xval =  (int)strtol(token, NULL, 16); // str to int
            //printf("\nExtracted val: %02x\n",xval);
            vals[i++] = (short)xval;
            token = strtok(NULL, s);  // Get next token
        }

        u1.na = vals[0];
        u1.nb = vals[1];
        u1.nc = vals[2];

        printf (">>>> %06x\n", u1.val);

        // load values to be written to FPGA
        c.sin = u1.val;
        // Addr is the secret address, writing to which makes
        // fpga interpret writeaddr values as midi note
        c.loc = MIDI_WRITEADDR;
        set_background_color(&c);

    } // for(;;) loop

}
```

```
int main(int argc, char *argv[])
{
    vga_ball_arg_t vla;
    int i;
    //static const char filename[] = "/dev/vga_ball";
    static const char filename[] = "/dev/audio_codec";

    printf("VGA ball Userspace program started\n");

    if ( (vga_ball_fd = open(filename, O_RDWR)) == -1) {
        fprintf(stderr, "could not open %s\n", filename);
        return -1;
    }

    printf("initial state: not priting ");

    // No args passed: receive-keypress-over-socket mode
    if (argc == 1)
    {
        socket_server_loop();
   }

        // If filename passed, then
        // read sin_table and writedata
        if (argc == 2){
            char *sin_file= argv[1];
            load_wave_table(sin_file);
            write_sample_data();
        }

        if (argc == 3){
    vga_ball_color_t c;
        char *a_val = argv[1];
        char *r_val = argv[2];
      short attack = (short) strtol(a_val, NULL, 10);
      short release = (short) strtol(r_val, NULL, 10);
      printf("\nWriting Attack:%d\tRelease:%d\n", attack, release);

      typedef union {
            uint16_t val;
            struct{
                unsigned int att: 8;
                unsigned int rel: 8;
            };
        }packed_un_t2;

        packed_un_t2 u2 = {.val = 0};
        u2.att = attack;
        u2.rel = release;
        c.sin = u2.val;
        c.loc = ENV_WRITEADDR;
        set_background_color(&c);

        }


    printf("VGA BALL Userspace program terminating\n");
    return 0;
}
```

8.11 vga_ball.c

```
/* * Device driver for the VGA video generator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *              drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod vga_ball.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree vga_ball.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "vga_ball.h"

//#define DRIVER_NAME "vga_ball"
#define DRIVER_NAME "audio_codec"

/* Device registers */
#define BG_RED(x) (x)
#define BG_GREEN(x) ((x)+1)
#define BG_BLUE(x) ((x)+2)
```

```
/*
 * Information about our device
 */
struct vga_ball_dev {
        struct resource res; /* Resource: our registers */
        void __iomem *virtbase; /* Where registers can be accessed in memory */
      vga_ball_color_t background;
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_background(vga_ball_color_t *background)
{
        iowrite16(background->sin, (dev.virtbase + 2*background->loc ));
   dev.background = *background;
}

static void read_background(vga_ball_color_t *background)
{
   background->sin = (__u16) ioread16( (dev.virtbase + 2*background->loc) );
   printk("bg->sin @ %d = %d", background->loc, background->sin);

}

        vga_ball_arg_t vla;

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long vga_ball_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{

        switch (cmd) {
        case VGA_BALL_WRITE_BACKGROUND:
                if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                                   sizeof(vga_ball_arg_t)))
                        return -EACCES;
                write_background(&vla.background);
                break;
        case VGA_BALL_READ_BACKGROUND:
           read_background(&vla.background);
                if (copy_to_user((vga_ball_arg_t *) arg, &vla,
                                 sizeof(vga_ball_arg_t)))
                        return -EACCES;
                break;

        default:
                return -EINVAL;
        }

        return 0;
}

/* The operations our device knows how to do */
static const struct file_operations vga_ball_fops = {
        .owner          = THIS_MODULE,
        .unlocked_ioctl = vga_ball_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice vga_ball_misc_device = {
        .minor          = MISC_DYNAMIC_MINOR,
        .name           = DRIVER_NAME,
        .fops           = &vga_ball_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init vga_ball_probe(struct platform_device *pdev)
{
      //vga_ball_color_t beige = { 0xf9, 0xe4, 0xb7 };
        int ret;

        /* Register ourselves as a misc device: creates /dev/vga_ball */
        ret = misc_register(&vga_ball_misc_device);

        /* Get the address of our registers from the device tree */
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
        if (ret) {
                ret = -ENOENT;
```

```
		goto out_deregister;
	}

	/* Make sure we can use these registers */
	if (request_mem_region(dev.res.start, resource_size(&dev.res),
			       DRIVER_NAME) == NULL) {
		ret = -EBUSY;
		goto out_deregister;
	}

	/* Arrange access to our registers */
	dev.virtbase = of_iomap(pdev->dev.of_node, 0);
	if (dev.virtbase == NULL) {
		ret = -ENOMEM;
		goto out_release_mem_region;
	}

	/* Set an initial color */
       //write_background(&beige);

	return 0;

out_release_mem_region:
	release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
	misc_deregister(&vga_ball_misc_device);
	return ret;
}

/* Clean-up code: release resources */
static int vga_ball_remove(struct platform_device *pdev)
{
	iounmap(dev.virtbase);
	release_mem_region(dev.res.start, resource_size(&dev.res));
	misc_deregister(&vga_ball_misc_device);
	return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id vga_ball_of_match[] = {
	//{ .compatible = "csee4840,vga_ball-1.0" },
	{ .compatible = "csee4840,audio_codec-1.0" },
	{},
};
MODULE_DEVICE_TABLE(of, vga_ball_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver vga_ball_driver = {
	.driver  = {
		.name  = DRIVER_NAME,
		.owner = THIS_MODULE,
		.of_match_table = of_match_ptr(vga_ball_of_match),
	},
	.remove		= __exit_p(vga_ball_remove),
};

/* Called when the module is loaded: set things up */
static int __init vga_ball_init(void)
{
	pr_info(DRIVER_NAME ": init\n");
	return platform_driver_probe(&vga_ball_driver, vga_ball_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit vga_ball_exit(void)
{
	platform_driver_unregister(&vga_ball_driver);
	pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_ball_init);
module_exit(vga_ball_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA ball driver");
```

8.12 vga_ball.h

```
#ifndef _VGA_BALL_H
#define _VGA_BALL_H

#include <linux/ioctl.h>

#define NSAMPLES 32768


typedef struct {
            short sin;//, squ, tri;
        int loc;
} vga_ball_color_t;


typedef struct {
  vga_ball_color_t background;
} vga_ball_arg_t;

#define VGA_BALL_MAGIC 'q'

/* ioctls and their arguments */
#define VGA_BALL_WRITE_BACKGROUND _IOW(VGA_BALL_MAGIC, 1, vga_ball_arg_t *)
#define VGA_BALL_READ_BACKGROUND  _IOR(VGA_BALL_MAGIC, 2, vga_ball_arg_t *)

#endif
```

8.13 lab2.c

```
/*
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <unistd.h>
#include "usbkeyboard.h"

//#define SERVER_HOST "0.0.0.0"
#define SERVER_HOST "127.0.0.1"
#define SERVER_PORT 42000

#define BUFFER_SIZE 16

#define USE_SOCKET


int sockfd; /* Socket file descriptor */

struct libusb_device_handle *keyboard;
uint8_t endpoint_address;

pthread_t network_thread;
void *network_thread_f(void *);


struct libusb_device_handle *keyboard;
uint8_t endpoint_address;

void onKeyChange(struct usb_keyboard_packet packet){
    int key_sig = packet.keycode[0];

    // -2A cuz we only support G2 to C#5 = 30 keys
    // 30 < 2^5 which nicely fits in a single 16bit wide bus
    // To support polyphony, we pass 5 bits of each keypress in a
    // single 16bit-wide writedata
    // let's say we press G2,A3,A#3. Our writedata will be
    // 00001 00010 00011
    // lets say we press C#5, C5, B5. Our Writedata will be
    // 11111 11110 11101
    // BTW, in MIDI, G2 = d43 = 0x2B
    int key_val = packet.keycode[1] - 0x2A;
    //printf("%02x %02x\n", key_sig, key_val);

    // below algorithm leads to stuck keys
    // when more than 2 keys are released at the same time
    static int n_keypress;
    static int keys_pressed[3];

    // Key press
    if (key_sig == 0x90)
        if(n_keypress <3){
            for (int i=0; i < 3;i++){
                if (keys_pressed[i] == 0){
                    keys_pressed[i] = key_val;
                    n_keypress++;
                    break;}
            }
        }

    // Key release
    if (key_sig == 0x80)
        for (int i = 0; i < 3; i++)
            if (keys_pressed[i] == key_val)
            {
                keys_pressed[i] = 0;
                n_keypress--;
            }

    //printf("Current array: %02x %02x %02x\n", keys_pressed[0], keys_pressed[1], keys_pressed[2]);
    //printf("n_keypress: %d\t", n_keypress);

    char buffer[BUFFER_SIZE];

    //sprintf(buffer, "%02x,%02x,%02x", keys_pressed[0], keys_pressed[1], keys_pressed[2]);
    sprintf(buffer, "%02x,%02x,%02x", keys_pressed[0], keys_pressed[1], keys_pressed[2]);
#ifdef USE_SOCKET
    if (key_val <= 0x1f && key_val >= 0x01){
        int n = write(sockfd, buffer, BUFFER_SIZE);
        if (n != BUFFER_SIZE)
            fprintf(stderr, "Error: write() failed. Is the server running?\n");
        printf("\n***\nTx size: %d\nTx data:%s\n***\n", n, buffer);
    }
    else
        printf("\nVALID RANGE G2-C#5\n");
#endif
 }

int main()
{


  struct usb_keyboard_packet packet;
  int transferred;
  //char keystate[12];
  /* Open the keyboard */

  if ( (keyboard = openkeyboard(&endpoint_address)) == NULL ) {
    fprintf(stderr, "Did not find a keyboard\n");
    exit(1);
  }
 #ifdef USE_SOCKET

  struct sockaddr_in serv_addr;
  /* Create a TCP communications socket */
  if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
    fprintf(stderr, "Error: Could not create socket\n");
    exit(1);
  }


  /* Get the server address */
  memset(&serv_addr, 0, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_port = htons(SERVER_PORT);
  if ( inet_pton(AF_INET, SERVER_HOST, &serv_addr.sin_addr) <= 0) {
    fprintf(stderr, "Error: Could not convert host IP \"%s\"\n", SERVER_HOST);
    exit(1);
  }

  /* Connect the socket to the server */
  if ( connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
    fprintf(stderr, "Error: connect() failed.  Is the server running?\n");
    exit(1);
  }
#endif

// Dont need network thread, since i only wanna send to server?


  /* Look for and handle keypresses */
  for (;;) {
    libusb_bulk_transfer(keyboard, endpoint_address,
                              (unsigned char *) &packet, sizeof(packet),
                              &transferred, 0);
    //if (transferred == sizeof(packet)) {
   //   sprintf(keystate, " %02x %02x %02x %02x %02x %02x %02x %02x", packet.modifiers,
packet.keycode[0],
   //          packet.keycode[1], packet.keycode[2], packet.keycode[3],
packet.keycode[4],packet.keycode[5], packet.keycode[6]);
   //   printf("%s\n", keystate);
      onKeyChange(packet);

    //}
  }

  return 0;
}
```

8.14 usbkeyboard.c

```
#include "usbkeyboard.h"

#include <stdio.h>
#include <stdlib.h>

/* References on libusb 1.0 and the USB HID/keyboard protocol
 *
 * http://libusb.org
 * http://www.dreamincode.net/forums/topic/148707-introduction-to-using-libusb-10/
 * http://www.usb.org/developers/devclass_docs/HID1_11.pdf
 * http://www.usb.org/developers/devclass_docs/Hut1_11.pdf
 */

/*
 * Find and return a USB keyboard device or NULL if not found
 * The argument con
 *
 */
struct libusb_device_handle *openkeyboard(uint8_t *endpoint_address) {
   libusb_device **devs;
   struct libusb_device_handle *keyboard = NULL;
   struct libusb_device_descriptor desc;
   ssize_t num_devs, d;
   uint8_t i, k;
   printf("1\n");

   /* Start the library */
   if ( libusb_init(NULL) < 0 ) {
      fprintf(stderr, "Error: libusb_init failed\n");
      exit(1);
   }
   printf("2\n");

   /* Enumerate all the attached USB devices */
   if ( (num_devs = libusb_get_device_list(NULL, &devs)) < 0 ) {
      fprintf(stderr, "Error: libusb_get_device_list failed\n");
      exit(1);
   }

   /* Look at each device, remembering the first HID device that speaks
     the keyboard protocol */
   printf("3\n");
```

```
  for (d = 0 ; d < num_devs ; d++) {
    libusb_device *dev = devs[d];
    if ( libusb_get_device_descriptor(dev, &desc) < 0 ) {
      fprintf(stderr, "Error: libusb_get_device_descriptor failed\n");
      exit(1);
    }
    printf("4\n");
    printf("dev class for dev %x: %x\n",d, desc.bDeviceClass);
    if (desc.bDeviceClass == 0){//LIBUSB_CLASS_PER_INTERFACE) {
      struct libusb_config_descriptor *config;
      libusb_get_config_descriptor(dev, 0, &config);
      for (i = 0 ; i < config->bNumInterfaces ; i++){
        printf("5\n");
        for ( k = 0 ; k < config->interface[i].num_altsetting ; k++ ) {
          printf("6\n");
          const struct libusb_interface_descriptor *inter =
            config->interface[i].altsetting + k ;
          printf("\nDetails for IF#%x : AltSetting=%x, IFclass=%x, IFSubClass=%x, IFproto=%x\n",
               i, config->interface[i].altsetting, inter->bInterfaceClass, inter->bInterfaceSubClass,
inter->bInterfaceProtocol);
          if ( inter->bInterfaceClass ==  1 &&
            inter->bInterfaceSubClass == 3 &&
            inter->bInterfaceProtocol == 0 ){
              printf("7\n");
              int r;
              if ((r = libusb_open(dev, &keyboard)) != 0) {
                fprintf(stderr, "Error: libusb_open failed: %d\n", r);
                exit(1);
              }
              printf("8\n");
              if (libusb_kernel_driver_active(keyboard,i))
                libusb_detach_kernel_driver(keyboard, i);
              libusb_set_auto_detach_kernel_driver(keyboard, i);
              printf("9\n");
              if ((r = libusb_claim_interface(keyboard, i)) != 0) {
                fprintf(stderr, "Error: libusb_claim_interface failed: %d\n", r);
                exit(1);
              }
              printf("10\n");
              printf("Nb Endpoints: %x, bEndptAddr:%x\n",inter->bNumEndpoints,
inter->endpoint[0].bEndpointAddress);
              //for (int aa = 0; aa < inter->bNumEndpoints; aa++ )
```

```
                    //    printf("Endpoint#: %d, bEndptAddr:%x\n",aa,
inter->endpoint[aa].bEndpointAddress);
                    printf("Endpoing Address: %x\n", inter->endpoint[0].bEndpointAddress);
                    *endpoint_address = inter->endpoint[1].bEndpointAddress;
                    printf("11\n");
                    goto found;
                 }
                 }
                 }
           }
           }

found:
           libusb_free_device_list(devs, 1);

           return keyboard;
           }
```

8.15 usbkeyboard.h

```
#ifndef _USBKEYBOARD_H
#define _USBKEYBOARD_H

#include <libusb-1.0/libusb.h>

#define USB_HID_KEYBOARD_PROTOCOL 1
#define USB_MIDI_PROTOCOL 1

struct usb_keyboard_packet {
  uint8_t modifiers;
  uint8_t keycode[7];
};

/* Find and open a USB keyboard device.  Argument should point to
   space to store an endpoint address.  Returns NULL if no keyboard
   device was found. */
extern struct libusb_device_handle *openkeyboard(uint8_t *);

#endif
```

8.16 memread.sv (TB)

```
module hpsfreq(
input    clk,
input    chipselect,
input    write,
input bclk,
input [15:0] addr,
input [15:0] writedata,
output [32:0] op
);

parameter num_samples = 48000;
logic [15:0] sin_table [num_samples-1 : 0];

//logic op_val;

logic [27:0] cntr_f;
// Receive data from HPS and store in local memory
always_ff @(posedge clk) begin
   if (chipselect && write) begin
      sin_table[addr] <= writedata;
   end
end

logic [15:0] op_addr = 16'b0;
// Output sin table to DAC @ bclk=48khz

always_ff @(posedge bclk)  begin

   if(!write) begin
         if (op_addr == 47999) op_addr <= 0;
      else op_addr <= op_addr + 1;
   end

end


// write the sample
assign    op = (!write) ? { 1'b0, sin_table[op_addr], 16'b0 } : {33{1'b0}};

endmodule
```

8.17 memread_diff.sv (TB)

```
module hpsfreq(
input    clk,
input    chipselect,
input    write,
input bclk,
input [15:0] addr,
input [15:0] writedata,
input [3:0] sel,
output [32:0] op
);

parameter num_samples = 48000;
logic [15:0] sin_table [num_samples-1 : 0];

//logic op_val;

logic [27:0] cntr_f;
// Receive data from HPS and store in local memory
always_ff @(posedge clk) begin
   if (chipselect && write) begin
      sin_table[addr] <= writedata;
   end
end

logic [31:0] op_addr = {32{1'b0}};
logic [15:0] addr_out;
// Output sin table to DAC @ bclk=48khz

logic [31:0] incr = {32{1'b0}};


always_comb begin
   case(sel)
     4'd0 : incr  = 1;
     4'd1 : incr  = 32'b0000_0000_0000_0001_0000_1111_0011_1000;
     4'd2 : incr  = 32'b0000_0000_0000_0001_0000_0101_1100_0000;
     4'd3 : incr  = 32'b0000_0000_0000_0001_0011_0000_0110_1111;
     4'd4 : incr  = 32'b0000_0000_0000_0001_0100_0010_1000_1010;
     4'd5 : incr  = 32'b0000_0000_0000_0001_0101_0101_1011_1000;
     4'd6 : incr  = 32'b0000_0000_0000_0001_0110_1010_0000_1001;
     4'd7 : incr  = 32'b0000_0000_0000_0001_0111_1111_1001_0001;
     4'd8 : incr  = 32'b0000_0000_0000_0001_1001_0110_0101_1111;
```

```
      4'd9 : incr  = 32'b0000_0000_0000_0001_1010_1110_1000_1010;
      4'd10 : incr = 32'b0000_0000_0000_0001_1100_1000_0010_0011;
      4'd11 : incr = 32'b0000_0000_0000_0001_1110_0011_0100_0011;
      4'd12 : incr = 32'b0000_0000_0000_0010_0000_0000_0000_0000;
   endcase
end

always_ff @(posedge bclk)  begin

   if(!write) begin
         if (op_addr[31:16] > 32767) op_addr <= 0;
      op_addr <= op_addr + incr;
   end

end

assign addr_out = op_addr[31:16];



// write the sample
assign    op = (!write) ? { 1'b0, sin_table[addr_out], 16'b0 } : {33{1'b0}};

endmodule
```

8.18 memread_diff_tb.sv (TB)

```
`timescale 1ns/1ns
module testbench();


logic clk=1'b0;
logic chipselect;
logic write;
logic bclk=1'b0;
logic [15:0] address = 0;
logic [15:0] data = -24000;
logic [32:0] out;
logic [27:0] counter = 0;
logic [3:0] sel = 0;



parameter DIV = 1042;
always #10 clk = ~clk;

always @(posedge clk) begin
        if( (counter >= 28'd1042-1)) begin
                counter <= 0;
        end
        else
                counter <= counter +1;
   bclk <= (counter < 28'd521 ) ? 1'b0 : 1'b1;
end




hpsfreq h1 (.clk(clk),
            .chipselect(chipselect),
            .write(write),
            .bclk(bclk),
        .addr(address),
        .writedata(data),
            .sel(sel),
            .op(out)
);
```

```
int i;

initial begin
	for ( i = 0 ; i <32768 ; i=i+1) begin
		@(posedge clk); address = address + 1; data = 23999 - i;
	end
end

initial begin

		write = 0; chipselect = 0; sel = 4'd7;
	#5	write = 1; chipselect = 1;
	#980000 write = 0; chipselect = 0;
	#1000000000 $finish();
end

endmodule
```

8.19 memread_tb.sv (TB)

```
`timescale 1ns/1ns
module testbench();


logic clk=1'b0;
logic chipselect;
logic write;
logic bclk=1'b0;
logic [15:0] address = 0;
logic [15:0] data = -24000;
logic [32:0] out;
logic [27:0] counter = 0;



parameter DIV = 1042;
always #10 clk = ~clk;

always @(posedge clk) begin
        if( (counter >= 28'd1042-1)) begin
                counter <= 0;
        end
        else
                counter <= counter +1;
   bclk <= (counter < 28'd521 ) ? 1'b0 : 1'b1;
end




hpsfreq h1 (.clk(clk),
            .chipselect(chipselect),
            .write(write),
            .bclk(bclk),
        .addr(address),
        .writedata(data),
            .op(out)
);

int i;

initial begin
```

```
        for ( i = 0 ; i <48000 ; i=i+1) begin
                @(posedge clk); address = address + 1; data = 23999 - i;
        end
end

initial begin

                write = 0; chipselect = 0;
        #5      write = 1; chipselect = 1;
        #980000 write = 0; chipselect = 0;
        #1000000000 $finish();
end

endmodule
```