6.111 Synthesized Drum

Abstract

The goal of this project is to simulate the membrane of a drum and synthesize a passable drum sound. We will do so by taking analog input, produced by a piezo grid, which encodes the input force and strike zone (or “epicenter”) of the strike. Using this data we will infer the location of the strike in order to simulate the drum membrane and produce a sound resembling what would happen if playing an actual drum. We hope to be able to do this with a small latency using the Nexys4 DDR board enabled with an Artix-7 FPGA chip.

Background

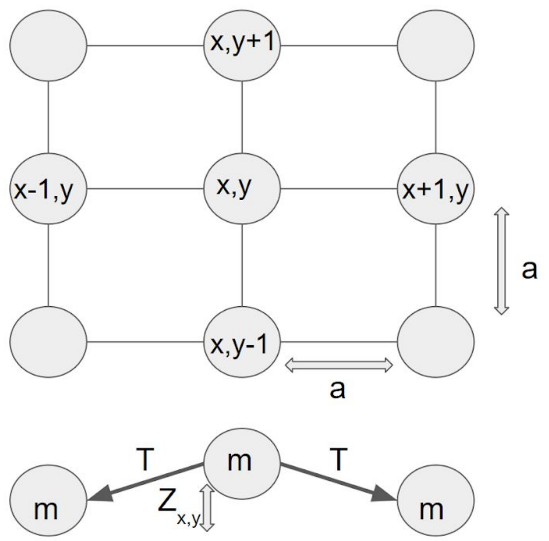
Electronic drum pads use analog signals from piezoelectric components to produce percussive noises using specialized hardware.
Early (“low quality”) electric drums synthesize the sounds by means of simulating the response of the drum as opposed to newer (“higher quality”) percussion instruments, which use mesh heads that closely mimic the response of typical drum membranes. The difference between these models is that the higher quality drums use a single piezo (or very few), in conjunction with tensioned applied mesh heads, and acts as a glorified amplifier. The “lower” quality drums have a peculiar problem in that they need to synthesize the sound somehow.

Our goal is to mimic the perturbation of this membrane using and synthesizing the resulting sound. We believe this to be an interesting problem since simulating multiparticle systems can be computationally expensive; but with some good hardware we can produce a decent sounding circular percussive membrane within a reasonably small latency. Real drums create slightly different sounds when struck/dampened at different locations. We believe that real-time simulation of a drum membrane coupled with a position-sensitive drum pad could very realistically recreate the position sensitivity of a real drum, which is a property typically lacking in drum pads on the market today.

Numerical Simulation of Drum Membrane

- In order to make this a tractable problem we will be making some reasonable assumptions:
    - The membrane can be represented as discrete regions of 2D space, which we will call “particles”.
    - The membrane is only one “particle” in depth or is composed of a single layer of particles.
    - The air column beneath the membrane does not add to the sound but does damp the movement of the membrane somewhat
    - The tension is strong enough to avoid having to use a damping constant. Although we may explore this later in later iterations.
    - For now we will assume that only the boundary conditions are 0, but may scale by some gaussian function time permitting.
- This is shown by the following:

Discretized Drum Membrane



- Each mass has mass m
- Vertical and horizontal spacing is a
- Tension between each mass is T
- $Z_{x,y}$ represents displacement out of page for each mass

$$F_{x,y} = ma_{x,y} = \frac{T}{a}\left(Z_{x+1,y} + Z_{x-1,y} + Z_{x,y-1} + Z_{x,y+1} - 4Z_{x,y}\right) - \Gamma v_{x,y} + F_d$$

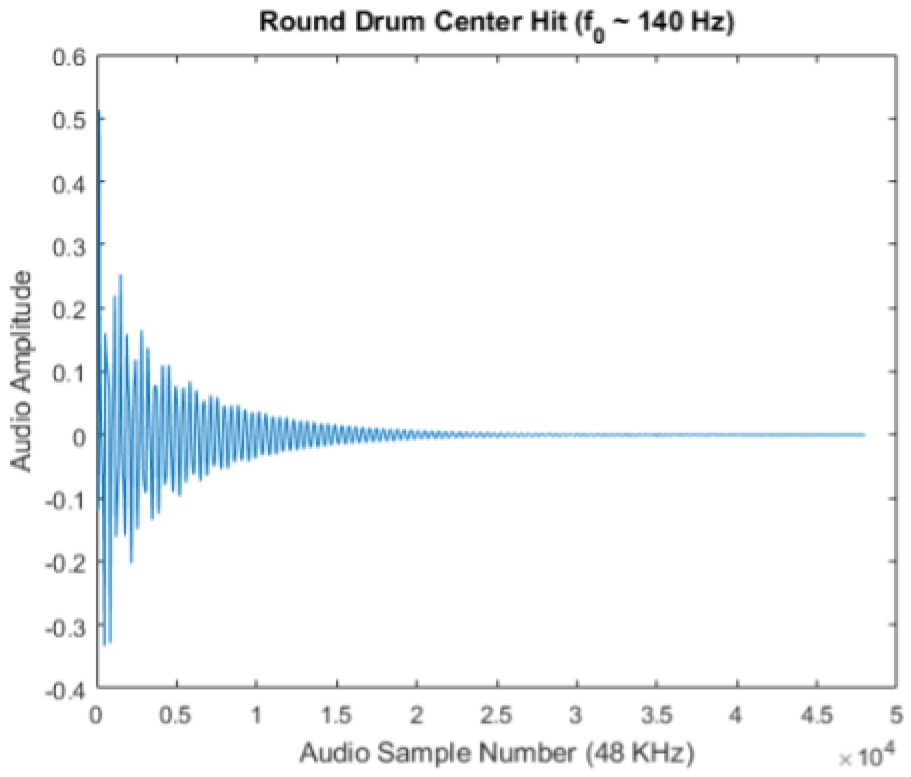
Equations of motion for each mass in the coupled oscillator.

$$v_{x,y}(t+dt) = v_{x,y}(t) + \frac{F_{x,y}}{m}dt$$

$$Z_{x,y}(t+dt) = Z_{x,y}(t) + v_{x,y}(t)dt + \frac{F_{x,y}}{2m}(dt)^2$$

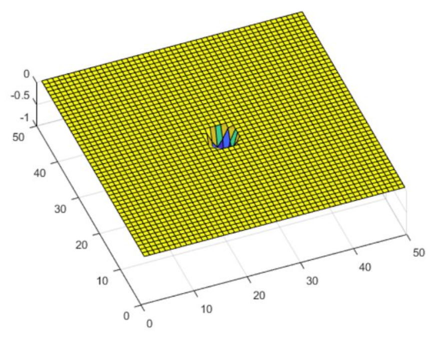
Taylor approximations of the position and velocity of each mass in the coupled oscillator based on the equations of motion. These calculations were repeated for each time step of duration dt. The highlighted term was included in the Matlab simulations, but neglected in the Verilog version.

- To characterize the motion of a 2D, finite, membrane we assume the discretized particles interact only with their nearest neighbors.
  - We assume that the force applied by each neighbor is proportional to the positional difference between that neighbor and center particle.
  - Additionally we assume a circular (radius N/2) boundary condition, inscribed an NxN grid of particles.
- Using Euler's method, with small time step corresponding to sampling interval, we calculate the new position vectors for each particle.
- Another method we will try to explore is using the normal modes of the system to characterize the time evolution given an input. We have decomposed the eigenvectors for a 50x50 grid of particle system and found the normal modes using Matlab, shown below.
  - Using the normal modes we can characterize the time evolution of the system given some input (strike on the drum pad).

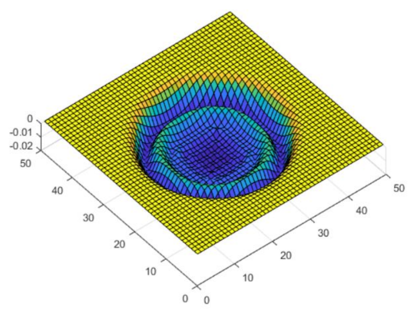


Audio signal from a simulated drum hit. Matlab was used to simulate a 50x50 2D coupled oscillator. The audio signal was derived from the sum of all masses in the oscillator.

Drum Simulation in Matlab (early results)



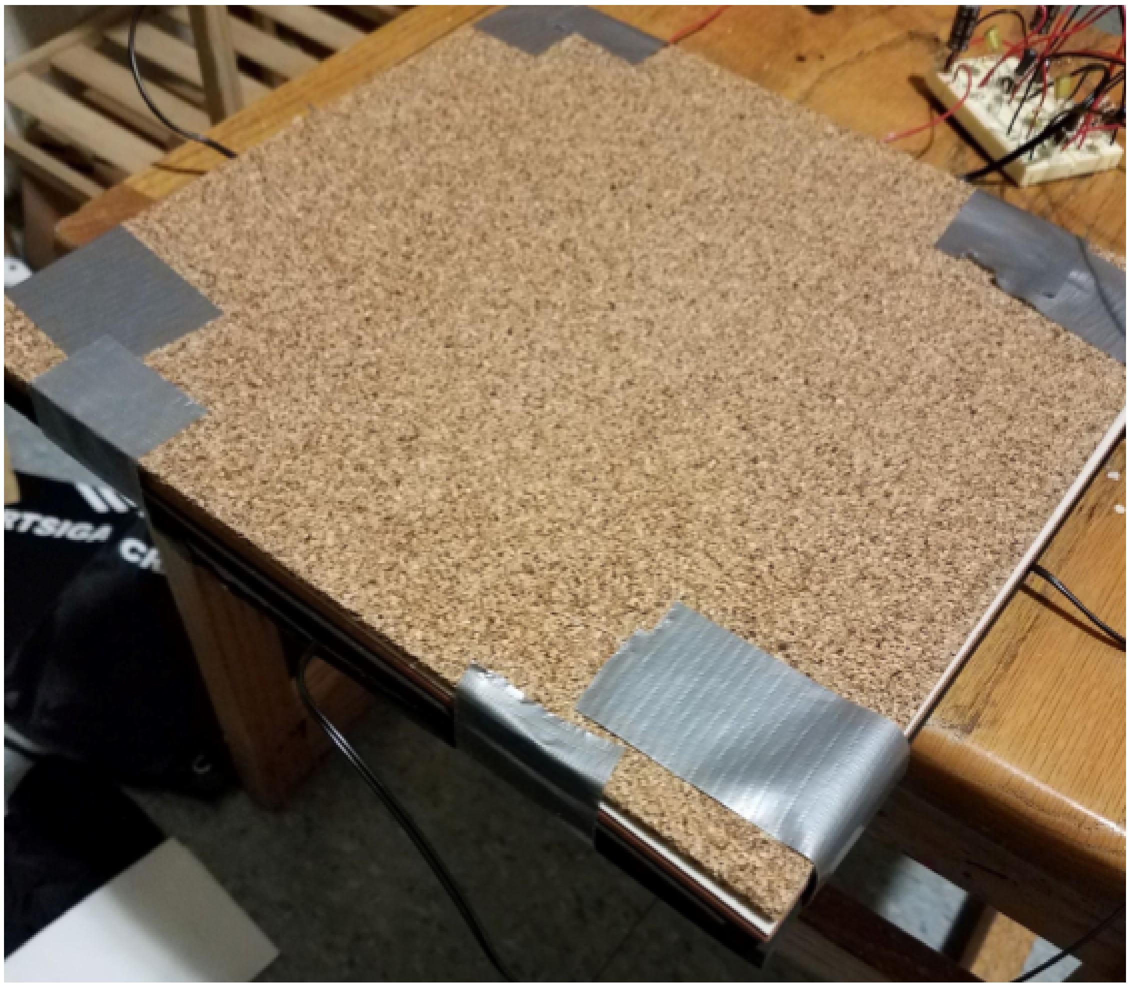
Initial



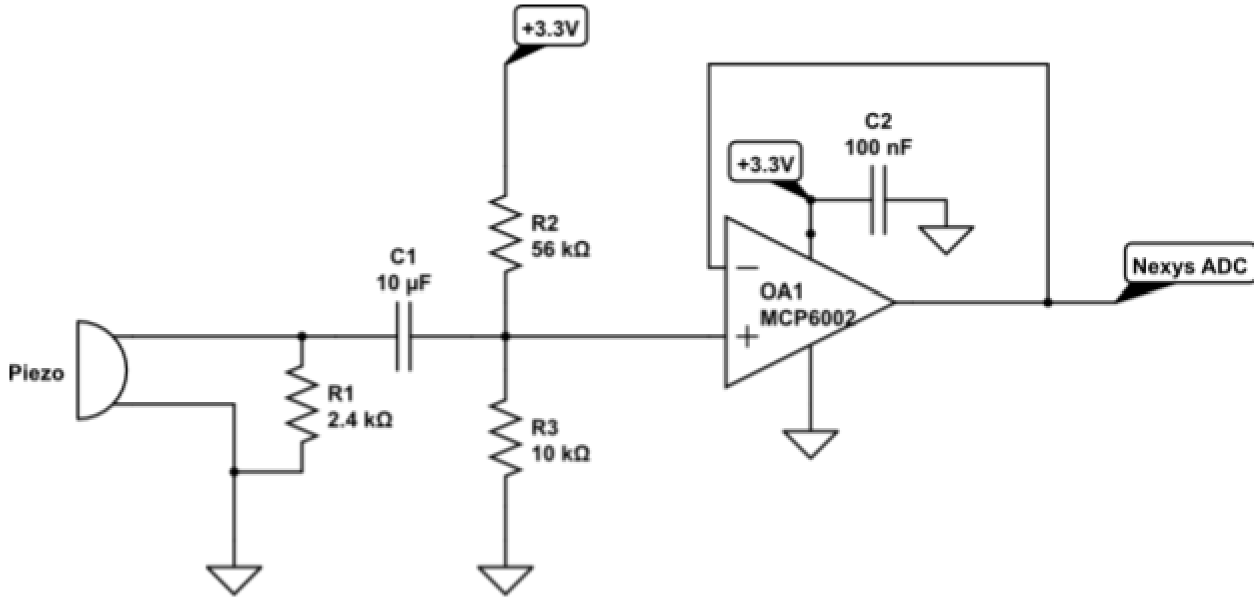
After some time...

The Physical Setup

- **Drum Head**
    - We tested a number of possible drum pad configurations. The main part of each were 4 piezos that were arranged with two along two perpendicular axes. This allowed us to determine a x and a y position for a drum strike for determining the sound our drum would make. In order to accomplish this, we needed a drum head that would remain relatively stable so the piezos would not move as well as material that would pass the vibrations of a strike along to all 4 piezos relatively quickly.
    - Materials we tried were a silicone pad used to place pet food bowls on, foam similar to that of a mouse pad, ceramic tiling, and cork board
    - Our final configuration consisted of a combination of foam, ceramic tiling, and cork board layers bound together with tape.


Photograph of the drum pad. The 4 piezo transducers were sandwiched between two ¼ inch layers of neoprene foam. A ceramic tile was placed on top of the foam to distribute the force of the drum hits, and a layer of cork board

- **ADC Interface**
    - For receiving this piezo outputs in a meaningful form to our FPGA, we needed to scale the voltage output down to the range of 0 - 1V. This is the range that is able to be received by the ADC on the Artix 7.
    - To accomplish this we used a number of voltage dividers for scaling and capacitors to account for noise.



Circuit to interface piezos to the Nexys Board's ADC. R1 reduces the voltage of the signal from the piezo to approximately 1 Vpp, and C1 AC couples the piezo signal. R1 and R2 bias the signal to approximately 0.5 V, and OA1 buffers the input signal.

- **Sound Output**

In order to “see” what the simulation was doing we thought it would be natural to listen to how the simulation is oscillating. We decided a good representation of this would be the sum of the positions of the pseudo-particles.

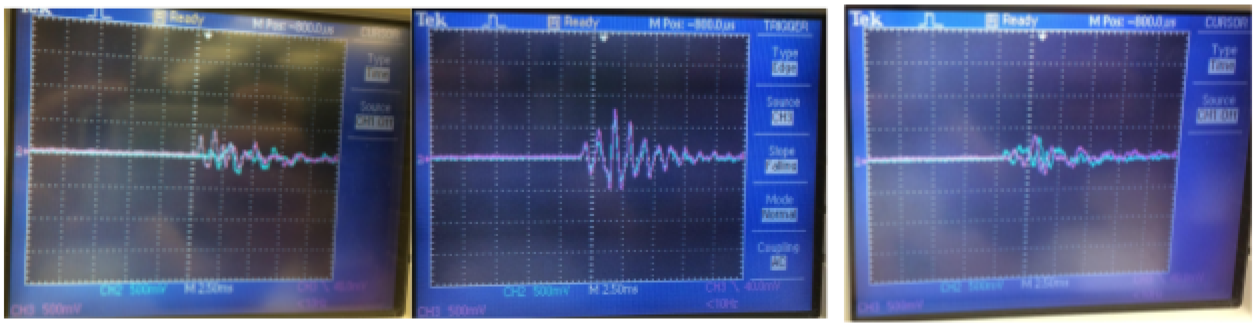
We then had to figure out a smart way to truncate this 32-bit number to 8 bits to feed in as inputs to the on-board PWM. Although we could have come up with a scheme that would let us avoid this truncation we decided it be best to focus our efforts on the simulation aspect of the project.

Modules

We need to run a time step of the simulation by solving for the new positions and velocities of each simulated particle. The ADC sampling rate is 48kHz, and if we use a 100MHz clock, then we have 2083 clock cycles to perform these tasks.

- **Infer Epicenter [Robbie]**

  This module makes use of the on-board 4-channel ADC . Each channel was 16-bits wide, but ignored the 4 LSB auxiliary bits. The remaining 12-bits had corresponded to discretized voltages between 0V-1V. The ADC reads are then measured across a time window following a trigger threshold being surpassed, indicating a strike. Across this time window, the values are averaged in order to extrapolate the likely origin of the strike. This is then mapped to an x, y coordinate across a range of 50 by 50. From this information we also come up with a 12 bit measure of the force that is fed into the image generation and time evolution modules. Some issues in making this module included finding a proper threshold and determining the best means of inferring the origin of the drum hit, whether it be a difference in force experienced or difference in time of triggering. What was decided was essentially integrating the values across the time window.



a b c

Two piezos were placed on the x-axis of the drum pad and their signals fed into the oscilloscope. The drum pad was then struck on a) the left side, b) the center, and c) the right side. Note that the signals line up for the center hit, but are offset from one another on the side hits. The time delay between the signals were used to estimate the position of the strike on the drum pad.

Issue with Infer epicenter

First issue was stepping down the 30V p-p signal of the piezos, then biasing that signal about 0.5V. The imprecision of this operation introduced some artifacts in the behavior which we were able to better detect with the display module.

Additionally, the module makes a lot of assumptions about how the wave travels through the board and to the piezos. Although most waves should have travelled relatively consistently, we believe the way we were sampling the piezos could be greatly improved by relying less on physical assumptions.

One minor (but humorous) artifact was that the piezos were in some cases sensitive to noise, both electrical and physical. As a result there were times when the drum would randomly go off from electrical noise around it or pick up the vibrations of you putting your phone on the table.

- **Fixed Point Multiplier [Ben]**

  In order to carry out Euler's method, multiplication by small numbers is necessary. A fixed point multiplier was designed that allows us to represent fractional numbers with a fixed resolution. This module carries out regular integer multiplication between its two multiplicands, the result of which has double the bits. The module then rounds the result back to the original bit length of the inputs and sign-extends the result.

Complications with the Fixed Point Multiplier

We initially used a 16-bit format to represent all numbers in the used in the coupled oscillator simulation (4 bits to the left of the decimal place, and 12 to the right), but eventually a 24-bit format was used to increase resolution (6 bits to the left of the decimal point and 18 to the right). Even higher precision would have been desirable, but any format exceeding 24 bits resulted in over-utilization of the FPGA's DSPs.

- **Time Evolution Solver [Ben]**

  This module was used to update the position and velocities of the particles in the simulated coupled oscillator using Euler's method to solve the equations of motion for each particle over a small time step (i.e. the time between audio samples). This module parallelized the update of the position and velocity for an entire column of the system at once (which in this case ranged from 25 to 50 particles long). In order to parallelize RAM access, the positions and velocities for an entire column were concatenated into a single large register. Hardware was then synthesized in a generate block to update each section of the position and velocity register according to the corresponding mass's current velocity, position, and the positions of its nearest neighbors.

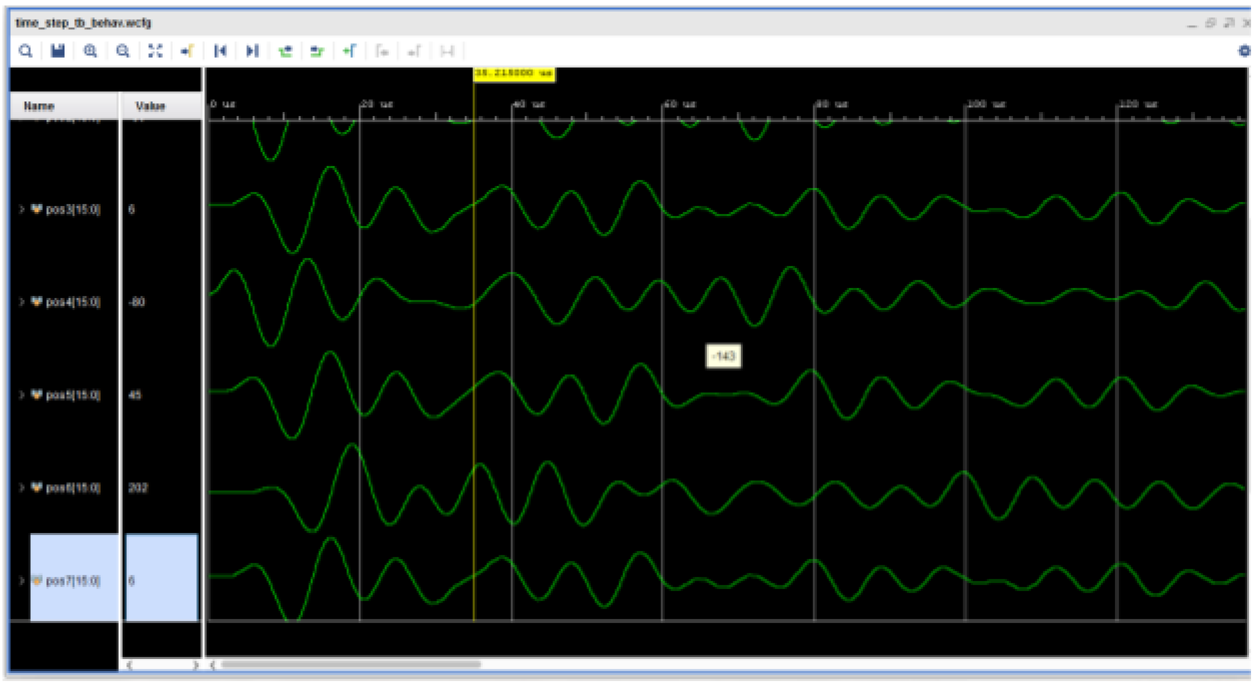
  This module was meant to have its inputs changed in the top-level to iterate through the columns of a 2D coupled oscillator and speed up the time evolution calculation compared to updating each mass individually. With slight modifications to the math in this module, it could be used to solve for a 1D coupled oscillator in only a few clock cycles.

  Development flow of the Time Evolution Module

  Several instances of the fixed point multiplier were used to construct the first order Taylor approximation of the position and velocity derived from the equations of motion for a spring-mass system/single-mass system. This simplified version was then tested with a test bench. We found that round-off error would accumulate over many time steps and cause the simulation to diverge if no damping was applied.

  The simulation was then made iteratively more complex. First, it was modified to calculate updated positions and velocities for a 1D coupled oscillator over one time step within. This calculation was originally not pipelined, and took place in one clock cycle. Second, it was modified to accept inputs for the positions of the column to the left and right of the update column. This allows it to be used to calculate the time evolution of a 2D system. Both the 1D and 2D systems were tested using test benches before proceeding.

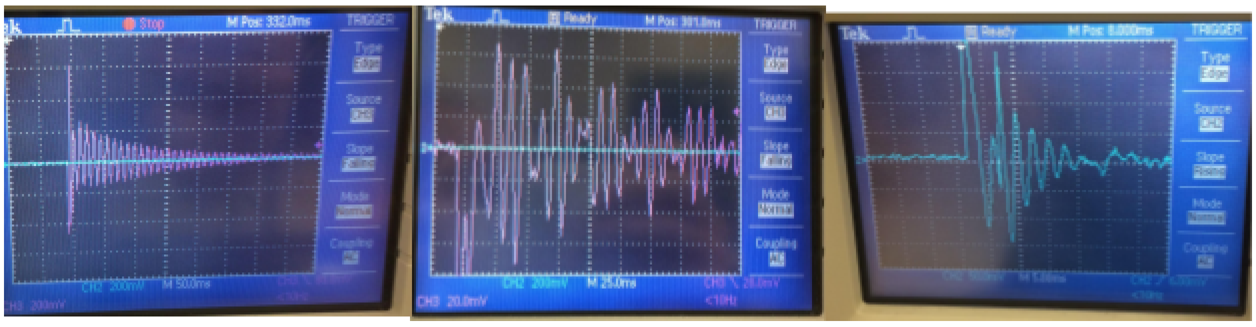
The audio signal from the simulation was derived by summing the positions of all particles used in the simulation. Test benches were used to determine what constant to add to the sum to make it unsigned, and how much to shift it to convert it to an 8-bit amplitude value for the audio PWM module.



A test bench for a 3x3 coupled oscillator. Initially only the center mass (pos5) was displaced. Note that the upper right and lower left corner positions (pos3 and pos7) match up, which is what one would expect given the symmetries of this particular system

Once the test benches verified that the math was being performed correctly, we tried uploading the code to an FPGA and tested it by setting the audio output to the sum of all positions in the simulated coupled oscillator. Due to complications, which will be discussed below, we were not able to run the 2D simulations on the FPGA, but we were able to run a simulation of a 50-mass 1D system which only required 3 clock cycles to update.

Once the simulation was working and producing an audio output, it was modified to take a force_in input, which stored the force applied to every mass. Infer_epicenter would provide the data necessary to determine which mass to apply a force to, and the magnitude of said force.



a b c

Nexys audio output from the simulation of a) a single-mass system, b) a 1D 3-mass system, c) the final 1D 50-mass system.

Later iterations of the time evolution module allowed for the values of Gamma and k (k = T/(m*a)) to be modified by means of btnu, btnd, btnl, and btnr. Values of Gamma and k were displayed on the 7-segment hex displays. This allowed for easy adjustment of the drum noise produced by the pad, particularly the dominant pitch of the sound, and its decay time.

Complications with the Time Evolution Module

Some of the terms in the coupled oscillator simulation required 3 multiplications. Thus, our initial design that solved for an entire column in one clock cycle contained multipliers that fed into other multipliers. This resulted in high latency and timing violations. We solved this problem by breaking the calculations into 3 separate steps, each performed during separate clock cycles. While we got this working for 1D systems, we were unable to get this pipelined time evolution module to work successfully for 2D systems.

The time evolution module was plagued by round-off error, which ended up being the most fundamental limitation of this module. This round off error was relatively small for the single mass system, and, so long as damping was included in the simulation, could safely be ignored. In the case of the 1D 3-mass system and the 2D 3x3 system, this led to the interesting behavior where, once perturbed, none of the particle positions in the system would return to their equilibrium position.

As the size of the system was scaled up to larger and larger numbers of particles, the complications from round-off error became increasingly severe. By the time we were working with a 1D 25-mass system, the sum of positions was no longer oscillating about zero, and would remain negative once it was perturbed. The problem was even worse for our final 50-mass system. None-the-less, the sum would still oscillate when the simulated system was perturbed by a force and could still be used to generate an audio signal.

Further improvements of this module would include modification to use floating point numbers and optimization of the DSP utilization. Both would allow for a much more accurate implementation of the larger coupled-oscillator systems we hoped to be able to simulate. Updates would also need to be made to perform a 2D simulation with our pipelined version of the time evolution module

- **Sound Generator [Ben] (handled in top level)**

  The audio signal was derived from the sum of the position of all masses in the coupled-oscillator simulation. The sum was converted to an unsigned value by adding a constant slightly larger than the smallest negative sum value. The sum was then converted to an 8-bit amplitude value via a bit shift. The 8-bit amplitude value was used to set the duty cycle of the PWM module used in lab 5A. This PWM signal is fed into a 4th-order analog low-pass filter on the Nexys board to create an analog audio signal. The analog audio signal was then fed into either headphones or a set of portable speakers for listening.

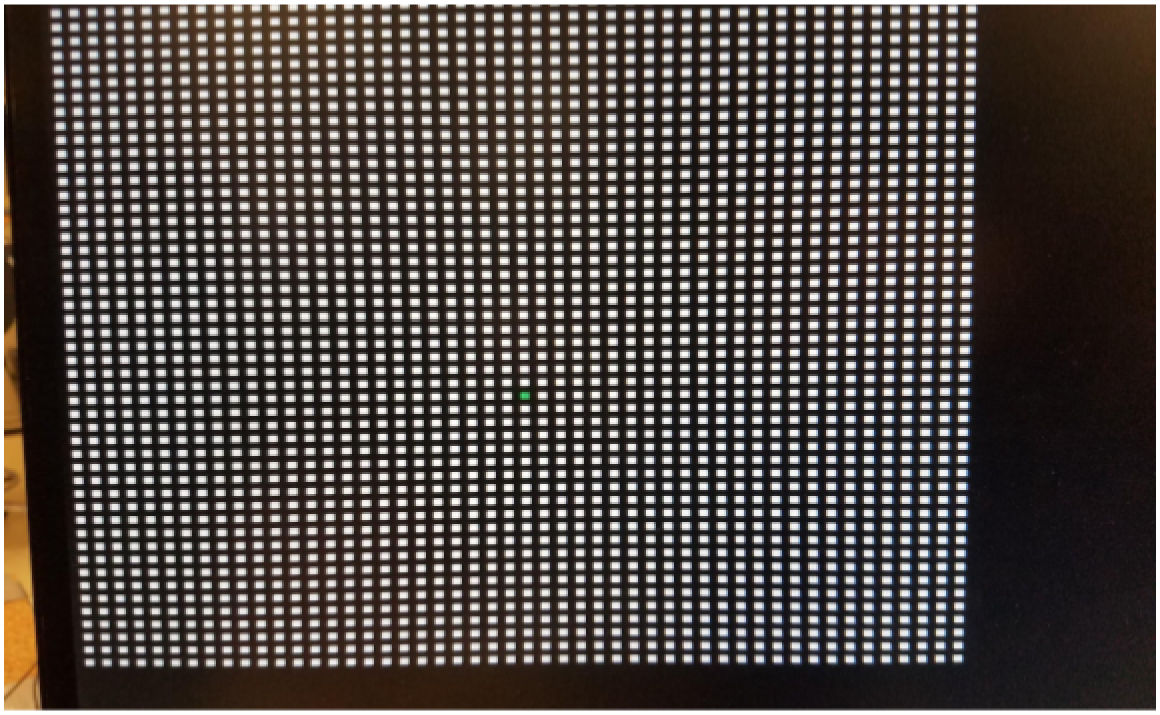
- **XVGA † [Evan /6.111 Lab 3]**

  This module is used to handle the display signals of the FPGA, namely the blank, vsync, hsync, and rgb signals. Each of these directly feeds to one of the pins of the VGA output on the board. We used the same module in a previous lab and simply incorporated it into our display module

- **Image Generation [Evan]**

  This consists of two display modes, both synced to a clock generated specifically for this module as VGA at 60fps on a 1024 by 768 display is easily updated using a 65 MHz clock. We produced this module by having inputs of information necessary for vga display directly from the board, the value of a switch in order to control what mode is currently displayed, a value for the size of the array to be displayed, the current force and x and y positions from the infer epicenter module, and the values of positions determined in the time evolution module. From this, the module outputs the relevant display information.

  If the value of the relevant switch is off, then the display mode is display mode 1. Pixels on the screen are categorized as either a mass or blank space. In this mode, what is displayed in regions of mass are points of an n by n array of discrete particles where n is passed into the module from our top level. The particles are by default colored solid white, but the mass at position x, y as determined by infer epicenter has a color proportional to the 12 bit force passed in, up to a max value of 111100000000, corresponding to the color of solid red. Regions determined to represent blank space are left black. This updates once per frame if the force surpasses a threshold defined in the top level to mitigate conflicts of data since they update at different intervals. We neglected to get pictures of this module in action but this is an example of a medium strength strike determined to be at point (26, 29):



A 50x50 grid representing the possible locations of the drum strike. One pixel at the location of the drum strike as determined by the infer_epicenter module is colored according to the force of the hit. In this case, it is the green pixel.

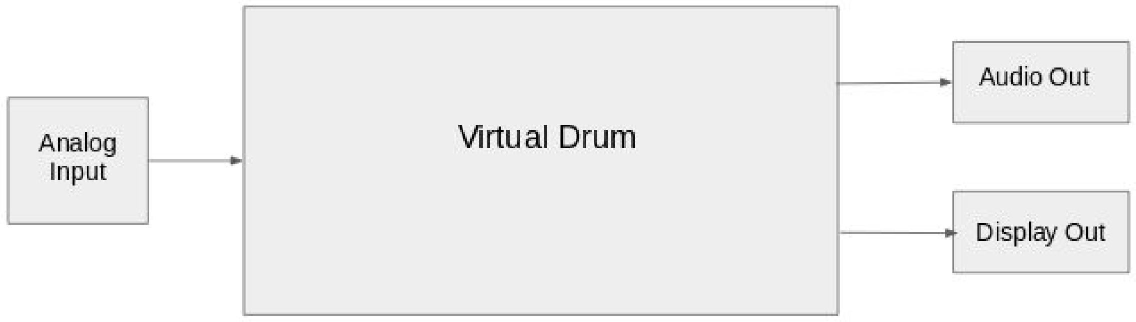
If the switch value is on, then the display is in display mode 2. In this mode we display the same grid of the same size as in the previous mode. This time, all particles are displayed as white in mass regions unless they fall in the middle row. This is because we initially planned for a 2-dimensional time evolution module but instead had to make due with a 1-dimensional model for computation purposes. However, if given full functionality of the time evolution module, we are prepared to display the full grid's time evolution. Each particle along the middle row has a color corresponding to the most significant 12 bits of the 24 bit value of position for that x coordinate. Since these values consistently hovered below 0, we flipped the sign of the signed input and made the maximum value red as we did in the previous mode. This module also updates once per frame but since it was difficult to determine an adequate threshold for an array of 50 24 bit values it simply relied on the last values passed in.

- **Top Level**

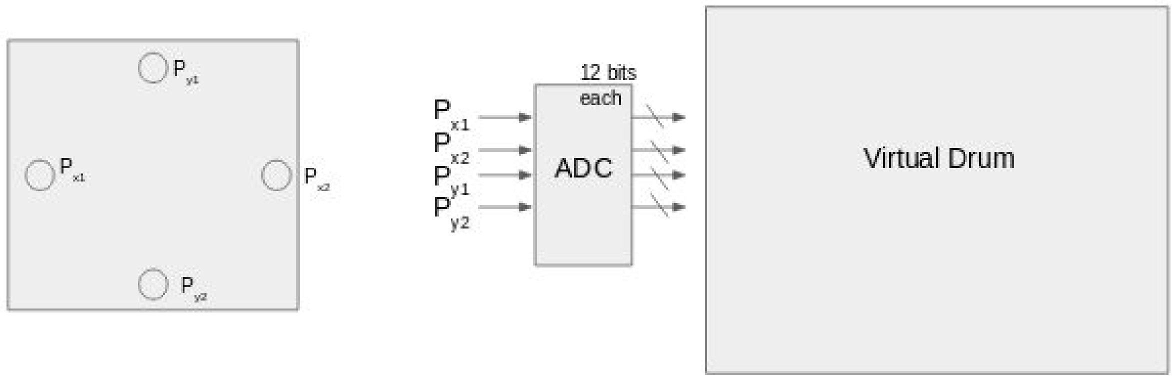
  Our top level module managed to connect all the inputs and outputs of each of the other modules, also handling some thresholding and timing. It handles all of the inputs from the ADC as well as the switches on the board and the clock. The top level is also where we implemented our sound generation. This was done by taking the 32 bit signed output from our time evolution modul, converted it to a positive, unsigned value, and selected a range of 8 bits that would model the generated sound wave between 0 and 1 volt for the aux output on the board. Further, we made use of plenty of the on-board IO for fine tuning. This allowed us to reach a stretch goal of adjusting the damping and tension values that were being used in the simulation.

Block Diagram:

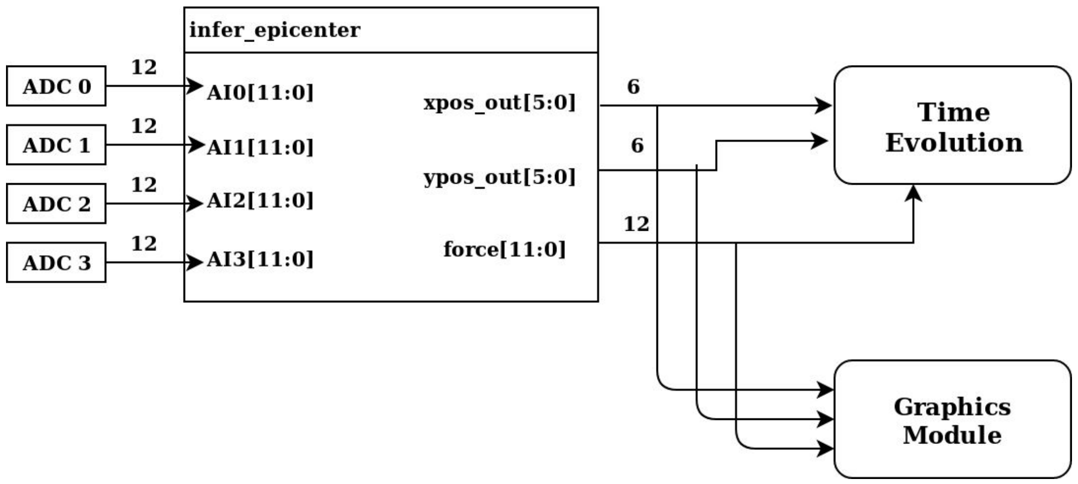
- **Overall**



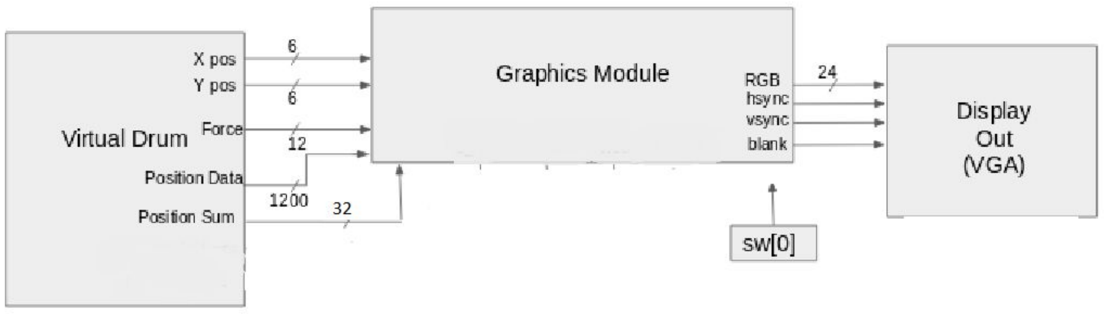
- **Interface**



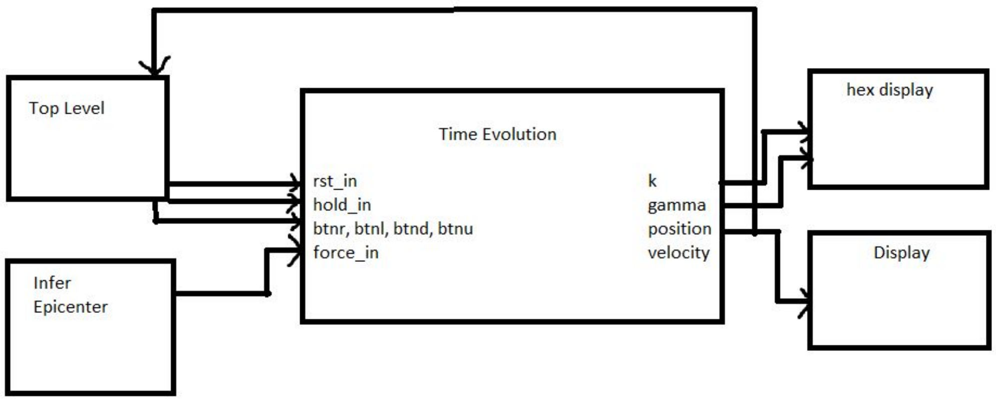
- **Infer Epicenter**



- **Graphics**



* **Time Evolution**



External Components

- We used 4 piezo transducers to readout the force of the drum hit and infer the position of the hit.

- To get the voltage range of the piezos to match that of the Nexys board's ADCs, we used 2 MCP6002 opamps as buffers,capacitors to ac-couple the signal, and some resistors to divide the signal and bias it.

- We used a ceramic plate or a high-density foam slab as the drum pad itself.

- We sandwiched the piezos between layers of ¼" thick neoprene foam underneath the ceramic tile
- A layer of cork board was placed on top of the ceramic tile to dampen drum hits slightly.

Ideas for Improvement

- Use Floating Point
- Migrate away from piezos for positional data
  - Could still use piezo for force data
- Migrate away from an XY coordinate system
  - Migrate towards radial coordinate system
- **Migrate towards capacitive touch technology**
  - Found some companies use CTT for some drum designs

Appendix:

Matlab Test Simulation:

```
%Author: Ben Sheffer
clear;
clc;

m = 1;
a = 16;
T = 3*80E6;%6*80E6;
k = 4;
fig = 1;

Nsamples = 48E3;
N = 1600;
gamma = 8;
X = zeros(1,N);
Vel = zeros(1,N);
mVec = m.*ones(1,N);

K = zeros(N,N); %declare the K matrix
%X(N/2) = 8;
tStep = 0.000028; %at 48KHz
A = zeros(1, Nsamples); %hold a second of audio data

Boundary = zeros(1,N); %use this to store the boundary conditions

for y = ((sqrt(N)/2)-2):((sqrt(N)/2)+2)
   for x = ((sqrt(N)/2)-2):((sqrt(N)/2)+2)
       sqrt(N)*(y-1)+x
      X(sqrt(N)*(y-1) + x) = -0.01*exp(-(((x-(sqrt(N)/2))^2+(y-(sqrt(N)/2))^2))^2/(2*3^2));
   end
end


for y = 1:sqrt(N)
   for x = 1:sqrt(N)
       if(sqrt((x-sqrt(N)/2)^2 + (y-sqrt(N)/2)^2) > sqrt(N)/2) mVec(sqrt(N)*(y-1)+x) = 1E9;       else
mVec(sqrt(N)*(y-1)+x) = m;
       end
   end
end
M = diag(mVec); %find M and inverse M
invM = inv(M);

C=zeros(1,N);

%fill K matrix
x=1;
y=1;
for y = 1:N
    for x = 1:N
```

```
        if(x == y) K(y,x) = 4*T/a; %if the matrix element corresponds to the mass we are considering,
add 4T/a
        elseif(x == y+1 && mod(y,sqrt(N)) ~= 0) K(y,x) = -T/a ; %else if the matrix element is a nearest
neighbor to the mass we are considering, add -T/a
        elseif(x == y-1 && mod(y - 1,sqrt(N)) ~= 0) K(y,x) = -T/a;
        elseif(x == y+sqrt(N)) K(y,x) = -T/a;
        elseif(x == y-sqrt(N)) K(y,x) = -T/a;
        end
    end
end

sparseK = sparse(K);

X = transpose(X);
Vel = transpose(Vel);

Xplot = zeros(sqrt(N), sqrt(N));
y=1;
for y = 1:sqrt(N)
    Xplot(y,:) = transpose(X((y-1)*sqrt(N)+1:y*sqrt(N)));
end
figure(fig)
fig = fig + 1;
surf(1:sqrt(N),1:sqrt(N),Xplot);

KMinv = sparse(invM*K);
for t = 1:Nsamples
    Vel = (Vel + (-KMinv*X-gamma.*invM*Vel).*tStep);%.*Boundary;
    X = X + Vel*tStep + 1/2*(-KMinv*X-gamma*invM*Vel)*tStep*tStep;
    A(t) = sum(X);
    %print('sample =');print(t); print(' A = ');print(A(t));
    if(mod(t,10) == 0)fprintf('sample = %f, A = %f \n', t, A(t));end
end
figure(fig)
fig = fig + 1;
plot(1:Nsamples, A);

Xplot = zeros(sqrt(N), sqrt(N));
y=1;
for y = 1:sqrt(N)
    Xplot(y,:) = transpose(X((y-1)*sqrt(N)+1:y*sqrt(N)));
end
figure(fig)
fig = fig + 1;
surf(1:sqrt(N),1:sqrt(N),Xplot);
```

<u>Top Level Verilog File:</u>

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 11/13/2019 02:30:41 PM
// Design Name:
// Module Name: top_level
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module top_level(input clk_100mhz,
                 input vauxp2, vauxn2,
                 input vauxp3, vauxn3,
                 input vauxp10, vauxn10,
                 input vauxp11, vauxn11,
                 input [15:0] sw,
                 input btnc, btnr, btnl, btnu, btnd,
                 output [15:0] led,
                 output [3:0] vga_r, vga_b, vga_g,
                 output vga_hs, vga_vs,
                 output logic aud_pwm, aud_sd,
                 output logic[7:0] an,
                 output logic ca,
                 output logic cb,
                 output logic cc,
                 output logic cd,
                 output logic ce,
                 output logic cf,
                 output logic cg
                );

    logic btnl_clean, btnr_clean, btnu_clean, btnd_clean;
    logic btnl_debounce, btnr_debounce, btnd_debounce, btnu_debounce;
    logic btnl_old, btnr_old, btnd_old, btnu_old;
    logic rst;
    assign rst = btnc;

    assign btnu_clean = btnu_debounce & !btnu_old;
    assign btnd_clean = btnd_debounce & !btnd_old;
    assign btnr_clean = btnr_debounce & !btnr_old;
    assign btnl_clean = btnl_debounce & !btnl_old;
```

```
always_ff @(posedge clk_100mhz)begin
    if (rst)begin
        btnu_old <= 1'b0;
        btnd_old <= 1'b0;
        btnr_old <= 1'b0;
        btnl_old <= 1'b0;
    end else begin
        btnu_old <= btnu_debounce;
        btnd_old <= btnd_debounce;
        btnr_old <= btnr_debounce;
        btnl_old <= btnl_debounce;
    end
end

debounce dbl(.reset_in(0), .clock_in(clk_100mhz), .noisy_in(btnl), .clean_out(btnl_debounce));
debounce dbr(.reset_in(0), .clock_in(clk_100mhz), .noisy_in(btnr), .clean_out(btnr_debounce));
debounce dbu(.reset_in(0), .clock_in(clk_100mhz), .noisy_in(btnu), .clean_out(btnu_debounce));
debounce dbd(.reset_in(0), .clock_in(clk_100mhz), .noisy_in(btnd), .clean_out(btnd_debounce));

logic signed [11:0] UT = (sw[13:12] == 3) ? 500 :
                         (sw[13:12] == 2) ? 400 :
                         (sw[13:12] == 1) ? 300 : 150;
logic signed [11:0] LT = 0 - UT;
parameter stepsPerSample = 50;
parameter N = 50;
parameter decPrec = 18;
parameter intPrec = 6;
parameter len = intPrec+decPrec;

wire [6:0] adc2 = 8'h12; // x1
wire [6:0] adc3 = 8'h13; // x2
wire [6:0] adc10 = 8'h1a; // y1
wire [6:0] adc11 = 8'h1b; // y2

// I think do_out is 16 bits wide
logic [11:0] adc_data;
logic [6:0] adc_address = adc2;
logic fresh_values = 0;
logic eos_out;
logic [1:0] delay;

xadc_wiz_0 my_adc0(.dclk_in(clk_100mhz), .daddr_in(adc_address),
                   .vauxp2(vauxp2), .vauxn2(vauxn2), .vauxn3(vauxn3), .vauxp3(vauxp3),
                   .vauxp10(vauxp10), .vauxn10(vauxn10), .vauxp11(vauxp11), .vauxn11(vauxn11),
                   .do_out(adc_data), .di_in(0), .den_in(1), .dwe_in(0), .reset_in(0), .vp_in(0), .vn_in(0),
                   .eos_out(eos_out));

logic ready;
logic [11:0] x1; // adc2
logic [11:0] x2; // adc3
logic [11:0] y1; // adc10
logic [11:0] y2; // adc11
logic [5:0] x;
logic [5:0] y;
logic signed [11:0] f;
```

```
infer_epicenter inf_epi(.clk_100mhz(clk_100mhz), .ready(ready), .sw(sw[4:3]),
                        .x1(x1), .x2(x2), .y1(y1), .y2(y2),
                        .x(x), .y(y), .f(f), .led(led));

image_generation display(.clk_100mhz(clk_100mhz), .sw(sw), .btnc(btnc),
                         .xcoord(x), .ycoord(y), .resol(N), .force_in(f), .positions(position),
                         .vga_r(vga_r), .vga_b(vga_b), .vga_g(vga_g), .vga_hs(vga_hs), .vga_vs(vga_vs));


/**
 *  logic for handling the pwm audio
 **/
logic signed ao_temp0, ao_temp1;
logic [7:0] audio_out;
logic [11:0] sample_delay;
logic pwm_val;
pwm (.clk_in(clk_100mhz), .rst_in(btnc), .level_in(audio_out), .pwm_out(pwm_val));

/* Splits the summation of all the positions */
logic signed [31:0] sum;
logic signed [31:0] audio_out_temp;
logic signed [31:0] offset_sum = sw[15] ? 2_000_000 : 4_000_000;
logic [4:0] shift = sw[14] ? 16 : 14;
assign audio_out_temp = ((sum) + offset_sum) >> shift;

assign aud_pwm = pwm_val;
assign aud_sd = 1;

/**
 * logic for running the simulation
 **/
logic signed [(intPrec+decPrec)*N-1:0] position;
logic signed [(intPrec+decPrec)*N-1:0] velocity;

logic [6:0] n = 0;
logic hold = 0;

logic signed [63:0] colSum;
logic signed [63:0] accum = 0;
logic signed [63:0] audioSig = 0;


logic [7:0] stepCount;
logic [4:0] delayCount;
logic [1:0] step = 0;
logic signed [N*len-1:0] force_in = 0;
logic signed [15:0] multiplyTest = -100;
logic [11:0] x_shift;
logic signed [23:0] gamma;
logic signed [23:0] k;
time_evolution #(.N(N), .intPrec(intPrec), .decPrec(decPrec))
  evolution(.clk_in(clk_100mhz), .rst_in(rst),
            .hold_in(hold), .Gamma_in(gamma), .k_in(k),
            .btnl(btnl_clean), .btnr(btnr_clean), .btnd(btnd_clean), .btnu(btnu_clean),
            .position(position), .velocity(velocity), .force_in(force_in));
ila_0 my_ila(.clk(clk_100mhz), .probe0(audio_out_temp), .probe1(sum), .probe2(gamma), .probe3(k));
summer #(.N(N),.len(len)) my_sum(.clk(clk_100mhz), .rst(rst), .position(position), .out(sum));
```

```
    seven_seg_controller my_seven_seg_controller(.val_in({gamma[23:8], k[23:8]}), .rst_in(rst), .clk_in(clk_100mhz),
.cat_out({cg,cf,ce,cd,cc,cb,ca}), .an_out(an));

    always_ff @(posedge clk_100mhz) begin

        ////////////////////////////////////////////////////////////////
        //
        // Begin infer_epicenter logic
        //
        if (eos_out) begin
            fresh_values <= 1;
            delay <= 0;
        end

        if (!fresh_values & !delay) begin
            ready <= 0;
        end

        if (fresh_values) begin
            case (adc_address)
                adc2: begin
                    adc_address <= adc3;
                    x1 <= adc_data; // typically always ready since adc_address defaults to adc2
                    ready <= 0;
                end
                adc3: begin
                    adc_address <= adc10;
                end
                adc10: begin
                    adc_address <= adc11;
                end
                adc11: begin
                    adc_address <= adc2;
                    fresh_values <= 0;
                    delay <= 1;
                end
                default: begin
                    adc_address <= adc2;
                    ready <= 0;
                    fresh_values <= 0;
                end
            endcase
        end
        if (delay) begin
            case (delay)
                1: x2 <= adc_data;
                2: y1 <= adc_data;
                3: y2 <= adc_data;
            endcase
            delay <= delay + 1;
            if (delay == 3) begin
                ready <= 1;
            end
        end
        //
        // End of infer_epicenter logic
        //
        ////////////////////////////////////////////////////////////////

        ////////////////////////////////////////////////////////////////
        //
        // Begin time_evolution logic
        //
        if(!rst)begin

            if(stepCount < stepsPerSample)begin
                if(hold == 1)begin
                     hold <= 0;
                     stepCount <= stepCount+1;
               end else begin

                if(delayCount == 8)begin
                    hold <= 1;
                    delayCount <= 0;
                end else begin
                    delayCount <= delayCount + 1;
                end

               end
            end else begin
                hold <= 1;
            end

            if(sample_delay < 2080) begin
                sample_delay  <= sample_delay + 1;
                // hold <= 1;
            end else if (sample_delay == 2080) begin
                sample_delay <= sample_delay + 1;
                x_shift <= x*len;
            end else if (sample_delay == 2083) begin
                force_in <= (f > UT | f < LT) ? f<<(x_shift+1) : 0;
                sample_delay <= 12'b0;
                //hold <= 0;
                audio_out <= audio_out_temp[7:0];
                stepCount <= 0;
                //hold <= 1;
            end else begin
                sample_delay <= sample_delay + 1;
            end
        end
        //
        // End of time_evolution logic
        //
        ////////////////////////////////////////////////////////////////
    end
endmodule
```

Time Evolution Module:

```
module time_evolution #(parameter N = 25, intPrec = 4, decPrec = 12, len = intPrec+decPrec)(
                        input clk_in,
                        input rst_in, hold_in,
                        input btnr, btnl, btnu, btnd,
                        input logic signed [N*len-1:0] force_in,
                        output logic [23:0] k_in, Gamma_in,
                        output logic signed [N*len-1:0] position,
                        output logic signed [N*len-1:0] velocity
    );

    parameter gamma_add = {1'b1, 8'b0};
    parameter k_add = {1'b1, 15'b0};

    logic [1:0] n; //use this to iterate through the masses

    parameter maxPos = 250000;
    parameter maxVel = 3000;
    parameter timeStep = 512;//16'b0000_0000_0000_1000;

    logic signed [len-1:0] Gamma;
    logic signed [len-1:0] k;

    always_ff @(posedge clk_in) begin
        if (btnl) Gamma_in <= Gamma_in - gamma_add;
        else if (btnr) Gamma_in <= Gamma_in + gamma_add;
        else if (btnd) k_in <= k_in - k_add;
        else if (btnu) k_in <= k_in + k_add;
        else begin
            Gamma <= Gamma_in;
            k <= k_in;
        end
    end

    genvar i;
    generate
        for(i = 1; i <= N; i=i+1) begin : N_block
            //use this to generate logic for a single row of coupled oscillators
            //declare all logics and what not that are necessary
            logic signed [len-1:0] velTemp1;
            logic signed [len-1:0] velTemp2;
            logic signed [len-1:0] posTemp;
            logic signed [len-1:0] posDouble;
            logic signed [len-1:0] velSum;
            logic signed [len-1:0] posSum;

            logic signed [len-1:0] inBVelTemp1;
            logic signed [len-1:0] inBVelTemp2;

            logic signed [len-1:0] inBVelocity;
            logic signed [len-1:0] inBPosition;

            logic signed [len-1:0] posSumTemp1;
            logic signed [len-1:0] posSumTemp2;

            logic signed [len-1:0] velSumTemp1;
            logic signed [len-1:0] velSumTemp2;

            logic signed [len-1:0] posN;
            logic signed [len-1:0] posNPlus;
            logic signed [len-1:0] posNMinus;

            logic signed [len-1:0] positionTemp1;
            logic signed [len-1:0] positionTemp2;
            logic signed [len-1:0] velocityTemp1;
            logic signed [len-1:0] velocityTemp2;

            //use this to keep track of which point in the calcuyation we are
            logic [1:0] state;

            assign posDouble = signed'(position[((i)*len)-1:(i-1)*len]) <<< 1;

            fixed_point_mult #(.len(len), .decimals(decPrec)) velocityTempMult2(.clk_in(clk_in),
.rst_in(rst_in), .inA(-Gamma), .inB(inBVelTemp2), .out(velTemp2)); //multiply damping by velocity
            fixed_point_mult #(.len(len), .decimals(decPrec))
velocityMult(.clk_in(clk_in),.rst_in(rst_in), .inA(timeStep), .inB(inBVelocity), .out(velSum)); //multiply
force by timeStep
            fixed_point_mult #(.len(len), .decimals(decPrec))
positionMult(.clk_in(clk_in),.rst_in(rst_in),.inA(timeStep), .inB(inBPosition), .out(posSum)); //multiply
velocity by timeStep
            fixed_point_mult #(.len(len), .decimals(decPrec))
velocityTempMult1(.clk_in(clk_in),.rst_in(rst_in), .inA(k), .inB(inBVelTemp1), .out(velTemp1));

            always_ff @(posedge clk_in) begin
                if(rst_in)begin
                    position[i*len-1:(i-1)*len] <= 0;

                    velocity[i*len-1:(i-1)*len] <= 0;

                    inBVelTemp1 <= 0;
                    inBVelTemp2 <= 0;
                    inBVelocity <= 0;
                    inBPosition  <= 0;
                    state <= 0;

                end else if(!hold_in) begin
                    case (state)
```

```
                      0: begin
                          //prepare the inputs for velMult1 and velMult2
                          inBVelTemp2 <= signed'(velocity[i*len-1:(i-1)*len]);
                          if(i == 1)begin
                              inBVelTemp1 <= signed'(position[(i+1)*len-1:(i)*len]) - posDouble;
                          end else if (i == N) begin
                              inBVelTemp1 <= signed'(position[(i-1)*len-1:(i-2)*len]) - posDouble;
                          end else begin
                              inBVelTemp1 <= signed'(position[(i+1)*len-1:(i)*len]) +
signed'(position[(i-1)*len-1:(i-2)*len]) - posDouble;
                          end
                          state <= state + 1; // move to next part of calculation
                      end
                      1: begin
                          //prepare the inputs for position and velocity multipliers
                          inBVelocity <= velTemp1 + velTemp2 + signed'(force_in[i*len-1:(i-1)*len]);
                          inBPosition <= signed'(velocity[i*len-1:(i-1)*len]);
                          state <= state + 1; //move to next step in calculation
                      end
                      2: begin
                          //add first order approximation of position and velocity functions for this time
step
                          if((signed'(position[i*len-1:(i-1)*len]) + posSum) > maxPos) begin
                              position[i*len-1:(i-1)*len] <= maxPos;//(velocity * 11)/524288;// +
(((-4*T/a*position + Gamma*velocity)/2)*(11**2))/(524288**2);
                          end else if((signed'(position[i*len-1:(i-1)*len]) + posSum) < (-maxPos))begin
                              position[i*len-1:(i-1)*len] <= -maxPos;//(velocity * 11)/524288;// +
(((-4*T/a*position + Gamma*velocity)/2)*(11**2))/(524288**2);
                          end else begin
                              position[i*len-1:(i-1)*len] <= signed'(position[i*len-1:(i-1)*len]) +
posSum;

                              //position[i*len-1:(i-1)*len] <= positionTemp1[i*len-1:(i-1)*len];
                          end

                          if((signed'(velocity[i*len-1:(i-1)*len]) + velSum) > maxVel)begin
                              velocity[i*len-1:(i-1)*len] <= maxVel;
                          end else if((signed'(velocity[i*len-1:(i-1)*len]) + velSum) < -maxVel) begin
                              velocity[i*len-1:(i-1)*len] <= -maxVel;
                          end else begin
                              velocity[i*len-1:(i-1)*len] <= signed'(velocity[i*len-1:(i-1)*len]) +
velSum;
                          end
                          state <= 0; //reset the state machine
                      end
                      default: state <= 0;
                    endcase
                end
            end
        end
    endgenerate
```

```
endmodule
```

Infer Epicenter Module:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Rob G
//
// Create Date: 11/13/2019 02:30:41 PM
// Design Name:
// Module Name: infer_epicenter
// Project Name: 6.111 Final Project
// Target Devices:  Nexys 4 DDR
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

/**
 inputs are biased to 0.5V - 12'b0
**/


module infer_epicenter (
        input clk_100mhz,
        input ready,
        input [1:0] sw,
        input signed [11:0] x1, x2, y1 ,y2,
        output logic [5:0] x, y,
        output logic signed [11:0] f,
        output logic [15:0] led
    );
    wire signed [11:0] THRESHOLD = sw[1] ? 750 : 650;
    wire signed [11:0] OFFSET = sw[0] ? -50 : -20;

    logic signed [11:0] UPPER_THRESHOLD = OFFSET + THRESHOLD;
    logic signed [11:0] LOWER_THRESHOLD = OFFSET - THRESHOLD;

    logic [5:0] xtemp, ytemp;
    logic signed [11:0] ftemp;

    /* Individual ADC triggers */
    logic x1_trigger;
    logic x2_trigger;
    logic y1_trigger;
    logic y2_trigger;
    logic trigger = x1_trigger | x2_trigger | y1_trigger | y2_trigger;

    logic [9:0] x1_trigger_time;
    logic [9:0] x2_trigger_time;
    logic [9:0] y1_trigger_time;
    logic [9:0] y2_trigger_time;


    /**
    * Ready goes high every 416 clock cycles
    * ready_count counts 2**10 = 1024 of those
    * At a 100 MHz clk we count over an interval of about 5ms
    **/
    logic [9:0] ready_count;

    signal_trigger x1_module(.lower(LOWER_THRESHOLD), .upper(UPPER_THRESHOLD),
                             .signal(x1), .signal_triggered(x1_trigger));

    signal_trigger x2_module(.lower(LOWER_THRESHOLD), .upper(UPPER_THRESHOLD),
                             .signal(x2), .signal_triggered(x2_trigger));

    signal_trigger y1_module(.lower(LOWER_THRESHOLD), .upper(UPPER_THRESHOLD),
                             .signal(y1), .signal_triggered(y1_trigger));

    signal_trigger y2_module(.lower(LOWER_THRESHOLD), .upper(UPPER_THRESHOLD),
                             .signal(y2), .signal_triggered(y2_trigger));

    time_diff x_module(.clk(clk_100mhz), .bottom(x1_trigger_time), .top(x2_trigger_time),
.position(xtemp));
    time_diff y_module(.clk(clk_100mhz), .bottom(y1_trigger_time), .top(y2_trigger_time),
.position(ytemp));

    average_signal signal_module(.clk(clk_100mhz), .x1(x1), .x2(x2), .y1(y1), .y2(y2), .average(ftemp));

    always_ff @(posedge clk_100mhz) begin
        if (ready) begin
            /** Top loop **/
            if (!x1_trigger_time | !x2_trigger_time | !y1_trigger_time | !y2_trigger_time) begin
                if (ready_count) ready_count <= ready_count + 1;
                else begin // ready_count overflows, end of integration
                    x <= xtemp;
                    y <= ytemp;
                    f <= ftemp;
                    x1_trigger_time <= 1023;
                    x2_trigger_time <= 1023;
                    y1_trigger_time <= 1023;
                    y2_trigger_time <= 1023;
                    led[11:0] <= 0;
                end
                if (trigger) begin
                    if (!led[0] & x1_trigger) led[0] <= 1;
                    if (!led[1] & x2_trigger) led[1] <= 1;
                    if (!led[2] & y1_trigger) led[2] <= 1;
                    if (!led[3] & y2_trigger) led[3] <= 1;
                    /* Sets "triggered" time if not already triggered */
                    x1_trigger_time <= ((x1_trigger_time == 1023) & x1_trigger) ? ready_count :
x1_trigger_time;
                    x2_trigger_time <= ((x2_trigger_time == 1023) & x2_trigger) ? ready_count :
x2_trigger_time;
                    y1_trigger_time <= ((y1_trigger_time == 1023) & y1_trigger) ? ready_count :
y1_trigger_time;
                    y2_trigger_time <= ((y2_trigger_time == 1023) & y2_trigger) ? ready_count :
y2_trigger_time;
                end
            end else begin
                if (trigger) begin
                    ready_count <= 1;
                    /*
                     * One or all of these are the first trigger,
                     * so set their "triggered" time to 1
                     * Controls the top loop
                    */
                    if (x1_trigger) x1_trigger_time <= 0;
                    if (x2_trigger) x2_trigger_time <= 0;
                    if (y1_trigger) y1_trigger_time <= 0;
                    if (y2_trigger) y2_trigger_time <= 0;
                end
            end
        end // ready
    end // always
Endmodule
```

Image Generation Module:

```
module image_generation(input clk_100mhz,
                        input[15:0] sw,
                        input btnc,
                        input signed [1199:0] positions,
                        input [23:0] [49:0] pos_col,
                        input [5:0] col, xcoord, ycoord, resol, resol2,
                        input [11:0] force_in,
                        input pos_we,
                        output[3:0] vga_r, vga_b, vga_g,
                        output vga_hs, vga_vs
                      );
    // create 65mhz system clock, happens to match 1024 x 768 XVGA timing
    logic clk_65mhz;
    clk_wiz_graphics clkdivider(.clk_in1(clk_100mhz), .clk_out1(clk_65mhz));

    //test inputs//
    logic [5:0] resol_test, xcoord_test, ycoord_test;
    logic [11:0] force_in_test;

    logic [1199:0] positions_test ;
    logic [23:0] [49:0] pos_col_test;
    logic pos_we_test;
    logic [5:0] col_test;
    logic [5:0] resol2_test;

    /*
     * Test inputs coming from infer epicenter & time evolution modules
     */
    //display1//
    assign resol_test = 6'd50;
    assign xcoord_test = 6'd24;
    assign ycoord_test = 6'd24;
    assign force_in_test = 12'b1111_0000_0000;

    //display2//
    assign positions_test = {24'b111100000000000000000000, 24'b111000000000000000000000, 24'b0, 24'b0, 24'b0, 24'b0,
24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0,
24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0,
24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0, 24'b0};
    assign pos_col_test = 0;
    assign pos_we_test = 0;
    assign col_test = 0;
    assign resol2_test = 6'd50;


    /** VGA Setup **/

    wire [10:0] hcount;    // pixel on current line
    wire [9:0] vcount;     // line number
    wire hsync, vsync, blank;
    wire [11:0] pixel1, pixel2;
    reg [11:0] rgb;
    xvga xvga1(.vclock_in(clk_65mhz),.hcount_out(hcount),.vcount_out(vcount),
          .hsync_out(hsync),.vsync_out(vsync),.blank_out(blank));
```

```
// btnc button is user reset
wire reset;
debounce db1(.reset_in(reset),.clock_in(clk_65mhz),.noisy_in(btnc),.clean_out(reset));

wire phsync,pvsync,pblank;

/*
 * display mode 1 with real inputs
 */
display1 d1(.vclock_in(clk_65mhz),.reset_in(reset), .sw(sw[1]),
            .xcoord(xcoord), .ycoord(ycoord), .force_in(force_in), .resol(resol),
            .hcount_in(hcount),.vcount_in(vcount),
            .hsync_in(hsync),.vsync_in(vsync),.blank_in(blank),
            .phsync_out(phsync),.pvsync_out(pvsync),.pblank_out(pblank),.pixel_out(pixel1));


/*
 * display mode 1 with test inputs
 */
/**
display1 d1(.vclock_in(clk_65mhz),.reset_in(reset),
            .xcoord(xcoord_test), .ycoord(ycoord_test), .force_in(force_in_test),
            .resol(resol_test),
            .hcount_in(hcount),.vcount_in(vcount),
            .hsync_in(hsync),.vsync_in(vsync),.blank_in(blank),
            .phsync_out(phsync),.pvsync_out(pvsync),.pblank_out(pblank),.pixel_out(pixel1));
**/

/*
 * display 2 test instance
 */
/**
display2 d2(.vclock_in(clk_65mhz),.reset_in(reset),
            .hcount_in(hcount),.vcount_in(vcount), .positions(positions_test), .resol(resol2_test),
            .hsync_in(hsync),.vsync_in(vsync),.blank_in(blank),
            .phsync_out(phsync),.pvsync_out(pvsync),.pblank_out(pblank),.pixel_out(pixel2));
**/

/*
 * display 2 real instance
 */
display2 d2(.vclock_in(clk_65mhz),.reset_in(reset),
            .hcount_in(hcount),.vcount_in(vcount), .positions(positions), .resol(resol2_test),
            .hsync_in(hsync),.vsync_in(vsync),.blank_in(blank),
            .phsync_out(phsync),.pvsync_out(pvsync),.pblank_out(pblank),.pixel_out(pixel2));



wire border = (hcount==0 | hcount==1023 | vcount==0 | vcount==767 |
               hcount == 512 | vcount == 384);

reg b,hs,vs;
always_ff @(posedge clk_65mhz) begin
    if (sw[0]) begin
        // 1 pixel outline of visible area (white)
        hs <= hsync;
```

```
            vs <= vsync;
            b <= blank;
            rgb <= pixel2;
        end else begin
            // default: pong
            hs <= phsync;
            vs <= pvsync;
            b <= pblank;
            rgb <= pixel1;
        end
    end

    // the following lines are required for the Nexys4 VGA circuit - do not change
    assign vga_r = ~b ? rgb[11:8]: 0;
    assign vga_g = ~b ? rgb[7:4] : 0;
    assign vga_b = ~b ? rgb[3:0] : 0;

    assign vga_hs = ~hs;
    assign vga_vs = ~vs;

endmodule

////////////////////////////////////////////////////////////////////////////////
//
// display mode 1
//
////////////////////////////////////////////////////////////////////////////////

module display1 (
   input vclock_in,        // 65MHz clock
   input reset_in,         // 1 to initialize module
   input [5:0] resol, xcoord, ycoord,
   input signed [11:0] force_in,
   input [10:0] hcount_in, // horizontal index of current pixel (0..1023)
   input [9:0]  vcount_in, // vertical index of current pixel (0..767)
   input hsync_in,         // XVGA horizontal sync signal (active low)
   input vsync_in,         // XVGA vertical sync signal (active low)
   input blank_in,         // XVGA blanking (1 means output black pixel)
   input [0:0] sw,
   output phsync_out,       // pong game's horizontal sync
   output pvsync_out,       // pong game's vertical sync
   output pblank_out,       // pong game's blanking
   output [11:0] pixel_out  // pong game's pixel  // r=23:16, g=15:8, b=7:0
   );

   parameter mass_side = 3'd5;
   parameter spacing = 4'd10;
   parameter red = 12'b1111_0000_0000;

   logic hsync_out, vsync_out, blank_out;
   logic[11:0] rgb, color;

   assign phsync_out = hsync_out;
   assign pvsync_out = vsync_out;
   assign pblank_out = blank_out;
   assign pixel_out = rgb;
```

```
logic[10:0] right_edge;
logic[9:0] bottom_edge;

logic [10:0] x_loc, xtemp;
logic [9:0] y_loc, ytemp;

logic [9:0] last_vcount_in;
logic vertical_flag = 0;
logic[2:0] hcount;
logic[2:0] vcount;
logic mass = 0;

assign right_edge = (spacing * resol + mass_side);
assign bottom_edge = (spacing * resol);

wire signed [11:0] THRESHOLD = sw ? 250 : 200;

assign x_loc = xtemp + mass_side;
assign y_loc = ytemp + mass_side;

always_ff @(posedge vsync_in) begin
     if (force_in > THRESHOLD) begin
        xtemp <= spacing * xcoord;
        ytemp <= spacing * ycoord;
        color <= force_in;
     end
end


always_ff @(posedge vclock_in) begin

    if(hcount_in == 0)begin
        hcount <= 0;
        mass <= 0;
    end
    else if(hcount != (mass_side - 1)) hcount <= hcount + 1;
    else if(hcount == (mass_side - 1)) begin
        mass <= !mass;
        hcount <= 0;
    end

    if (last_vcount_in != vcount_in) begin
        if(vcount_in == 0)begin
            vcount <= 0;
            vertical_flag <= 0;
        end
        else if(vcount != (mass_side - 1)) vcount <= vcount + 1;
        else if(vcount == (mass_side - 1)) begin
             vcount <= 0;
             vertical_flag <= !vertical_flag;
        end
    end

    last_vcount_in <= vcount_in;
    hsync_out <= hsync_in;
    vsync_out <= vsync_in;
    blank_out <= blank_in;
```

```
   end

   always_comb begin

       if((hcount_in <= right_edge) && (vcount_in <= bottom_edge)) begin
           if(mass && vertical_flag)begin
               if((hcount_in >= x_loc) && (hcount_in <= (x_loc + mass_side)) && (vcount_in >= y_loc) && (vcount_in <= 
(y_loc + mass_side))) rgb = color;
               else rgb = 12'b1111_1111_1111;
           end
           else rgb = 12'b0;
       end
       else rgb = 12'd0;
   end

endmodule


////////////////////////////////////////////////////////////////////////////////
//
// display mode 2
//
////////////////////////////////////////////////////////////////////////////////

module display2 (
   input vclock_in,        // 65MHz clock
   input reset_in,         // 1 to initialize module
   input [10:0] hcount_in, // horizontal index of current pixel (0..1023)
   input [9:0]  vcount_in, // vertical index of current pixel (0..767)
   input hsync_in,         // XVGA horizontal sync signal (active low)
   input vsync_in,         // XVGA vertical sync signal (active low)
   input blank_in,         // XVGA blanking (1 means output black pixel)
   input [5:0] resol,
   input signed [1199:0] positions,
   output phsync_out,       // pong game's horizontal sync
   output pvsync_out,       // pong game's vertical sync
   output pblank_out,       // pong game's blanking
   output [11:0] pixel_out  // pong game's pixel  // r=23:16, g=15:8, b=7:0
   );



   assign phsync_out = hsync_in;
   assign pvsync_out = vsync_in;
   assign pblank_out = blank_in;

   //display mode 2//
   logic hsync_out;
   logic vsync_out;
   logic blank_out;

   assign phsync_out = hsync_out;
   assign pvsync_out = vsync_out;
   assign pblank_out = blank_out;

   logic[11:0] rgb;
   assign pixel_out = rgb;
   parameter mass_side = 3'd5;
   parameter spacing = 4'd10;

   logic[10:0] right_edge;
   logic[9:0] bottom_edge;

   logic [10:0] x_loc;
   logic [9:0] y_loc;
   logic [8:0] xcoord = 0;
   logic [8:0] ycoord = 48;
   logic signed [23:0] positions1[49:0];
   logic [7:0] hold_count;

   logic mass = 0;
   logic [9:0] last_vcount_in;
   logic vertical_flag = 0;
   logic[2:0] hcount;
   logic[2:0] vcount;


   logic [3:0] r;
   logic [3:0] g;
   logic [3:0] b;
   logic [11:0] color = 0;
   logic [11:0] color_out;
   parameter red = 12'b1111_0000_0000;



   assign right_edge = (spacing * resol + mass_side);
   assign bottom_edge = (spacing * resol);
   assign x_loc = (spacing * (xcoord[8:1]) + mass_side);
   assign y_loc = (spacing * (ycoord[8:1]) + mass_side);


   always_ff @(posedge vsync_in) begin
     if((positions <= -24'sd4096)) begin
           hold_count <= 0;
           positions1 <= {positions[1199:1176], positions[1175:1152], positions[1151:1128], positions[1127:1104],
positions[1103 : 1080],
           positions[1079:1056], positions[1055:1032], positions[1031:1008], positions[1007:984], positions[983:960],
positions[959:936],
           positions[935:912], positions[911:888], positions[887:864], positions[863:840], positions[839:816],
positions[815:792], positions[791:768],
           positions[767:744], positions[743:720], positions[719:696], positions[695:672], positions[671:648],
positions[647:624], positions[623:600],
           positions[599:576], positions[575:552], positions[551:528], positions[527:504], positions[503:480],
positions[479:456], positions[455:432],
           positions[431:408], positions[407:384], positions[383:360], positions[359:336], positions[335:312],
positions[311:288], positions[287:264],
           positions[263:240], positions[239:216], positions[215:192], positions[191:168], positions[167:144],
positions[143:120], positions[119:96],
           positions[95:72], positions[71:48], positions[47:24], positions[23:0]};
      end
      else hold_count <= hold_count + 1;
   end
```

```
  always_ff @(posedge vclock_in) begin

    r <= ~positions1[xcoord[8:1]][22:19];
    g <= ~positions1[xcoord[8:1]][18:15];
    b <= ~positions1[xcoord[8:1]][14:11];
    color <= ({r, g, b} > 12'b0) ? {r, g, b} : 12'b111111111111;

      if(hcount_in == 0)begin
          hcount <= 0;
          mass <= 0;
          xcoord <= 0;
      end
      else if(hcount != (mass_side - 1)) hcount <= hcount + 1;

      else if(hcount == (mass_side - 1)) begin
       mass <= !mass;
       hcount <= 0;
       xcoord <= xcoord + 1;
      end

      if (last_vcount_in != vcount_in) begin
          if(vcount_in == 0)begin
              vcount <= 0;
              vertical_flag <= 0;
          end

          else if(vcount != (mass_side - 1)) vcount <= vcount + 1;

          else if(vcount == (mass_side - 1)) begin
               vcount <= 0;
               vertical_flag <= !vertical_flag;
               //ycoord <= ycoord + 1;

          end
      end

      last_vcount_in <= vcount_in;
      hsync_out <= hsync_in;
      vsync_out <= vsync_in;
      blank_out <= blank_in;
  end

  always_comb begin
      color_out = ((red > color) || (color == 12'b111111111111)) ? color:red;

      if((hcount_in <= right_edge) && (vcount_in <= bottom_edge)) begin

          if(mass && vertical_flag)begin
              if((hcount_in >= x_loc) && (hcount_in <= (x_loc + mass_side)) && (vcount_in >= y_loc) && (vcount_in <=
(y_loc + mass_side))) rgb = color_out;
              else rgb = 12'b1111_1111_1111;
          end
          else rgb = 12'b0;

      end
```

```
       else rgb = 12'd0;
   end
endmodule


module synchronize #(parameter NSYNC = 3)  // number of sync flops.  must be >= 2
                   (input clk,in,
                    output reg out);

  reg [NSYNC-2:0] sync;
  always_ff @ (posedge clk)
  begin
    {out,sync} <= {sync[NSYNC-2:0],in};
  end
endmodule


////////////////////////////////////////////////////////////////////////////////
// Update: 8/8/2019 GH
// Create Date: 10/02/2015 02:05:19 AM
// Module Name: xvga
//
// xvga: Generate VGA display signals (1024 x 768 @ 60Hz)
//
//                              ---- HORIZONTAL -----     ------VERTICAL -----
//                              Active                    Active
//                    Freq      Video   FP  Sync   BP      Video   FP  Sync  BP
//   640x480, 60Hz    25.175    640     16    96   48       480    11    2    31
//   800x600, 60Hz    40.000    800     40   128   88       600     1    4    23
//   1024x768, 60Hz   65.000    1024    24   136  160       768     3    6    29
//   1280x1024, 60Hz  108.00    1280    48   112  248       768     1    3    38
//   1280x720p 60Hz   75.25     1280    72    80  216       720     3    5    30
//   1920x1040 60Hz   148.5     1920    88    44  148      1080     4    5    36
//
// change the clock frequency, front porches, sync's, and back porches to create
// other screen resolutions
////////////////////////////////////////////////////////////////////////////////

module xvga(input vclock_in,
            output reg [10:0] hcount_out,    // pixel number on current line
            output reg [9:0] vcount_out,     // line number
            output reg vsync_out, hsync_out,
            output reg blank_out);

   parameter DISPLAY_WIDTH  = 1024;      // display width
   parameter DISPLAY_HEIGHT = 768;       // number of lines

   parameter  H_FP = 24;                 // horizontal front porch
   parameter  H_SYNC_PULSE = 136;        // horizontal sync
   parameter  H_BP = 160;                // horizontal back porch

   parameter  V_FP = 3;                  // vertical front porch
   parameter  V_SYNC_PULSE = 6;          // vertical sync
   parameter  V_BP = 29;                 // vertical back porch

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   reg hblank,vblank;
   wire hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount_out == (DISPLAY_WIDTH -1));
   assign hsyncon = (hcount_out == (DISPLAY_WIDTH + H_FP - 1));  //1047
   assign hsyncoff = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE - 1));  // 1183
   assign hreset = (hcount_out == (DISPLAY_WIDTH + H_FP + H_SYNC_PULSE + H_BP - 1));  //1343

   // vertical: 806 lines total
   // display 768 lines
   wire vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount_out == (DISPLAY_HEIGHT - 1));   // 767
   assign vsyncon = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP - 1));  // 771
   assign vsyncoff = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE - 1));  // 777
   assign vreset = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP + V_SYNC_PULSE + V_BP - 1)); // 805

   // sync and blanking
   wire next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always_ff @(posedge vclock_in) begin
      hcount_out <= hreset ? 0 : hcount_out + 1;
      hblank <= next_hblank;
      hsync_out <= hsyncon ? 0 : hsyncoff ? 1 : hsync_out;  // active low

      vcount_out <= hreset ? (vreset ? 0 : vcount_out + 1) : vcount_out;
      vblank <= next_vblank;
      vsync_out <= vsyncon ? 0 : vsyncoff ? 1 : vsync_out;  // active low

      blank_out <= next_vblank | (next_hblank & ~hreset);
   end

endmodule
```

Helper Modules:

```
//////////////////////////////////////////////////////////////////
// Time Evolution Helper Modules
//

/*
 * Fixed point mutliplier module
 * With trade-off precision between integer and fraction
*/
module fixed_point_mult #(parameter len, parameter decimals) (
                   input signed [len-1:0] inA,
                   input signed [len-1:0] inB,
                   output logic signed [len-1:0] out
    );

  //output is delayed 2 clock cycles...
  logic signed [len*2 - 1:0] temp0;// = 0;
  assign temp0 = inA*inB >>> decimals;
  assign out = temp0[len-1:0];

endmodule

/* Breaks up large sums into smaller sums */
module summer #(parameter N = 50, len = 16)
              (input clk, rst,
               input signed [N*len - 1:0] position,
               output logic signed [31:0] out
              );
    logic [1:0] step;
    logic signed [31:0] classic_sum0, classic_sum1, classic_sum2, classic_sum3, classic_sum4,
                       classic_sum5, classic_sum6, classic_sum7, classic_sum8, classic_sum9;
    logic signed [31:0] intermediate0, intermediate1;

    always_ff @(posedge clk) begin
        case (step)
            0: begin
               classic_sum0 <= signed'(position[len-1:0]) + signed'(position[len*(2)-1:len*(2-1)]) +
signed'(position[len*(3)-1:len*(3-1)]) +
signed'(position[len*(4)-1:len*(4-1)])+signed'(position[len*(5)-1:len*(5-1)]);
               classic_sum1 <=
signed'(position[len*(6)-1:len*(6-1)])+signed'(position[len*(7)-1:len*(7-1)])+signed'(position[len*(8)-1:l
en*(8-1)])+signed'(position[len*(9)-1:len*(9-1)])+signed'(position[len*(10)-1:len*(10-1)]);
               classic_sum2 <=
signed'(position[len*(11)-1:len*(11-1)])+signed'(position[len*(12)-1:len*(12-1)])+signed'(position[len*(13
)-1:len*(13-1)])+signed'(position[len*(14)-1:len*(14-1)])+signed'(position[len*(15)-1:len*(15-1)]);
               classic_sum3 <=
signed'(position[len*(16)-1:len*(16-1)])+signed'(position[len*(17)-1:len*(17-1)])+signed'(position[len*(18
)-1:len*(18-1)])+signed'(position[len*(19)-1:len*(19-1)])+signed'(position[len*(20)-1:len*(20-1)]);
```

```
               classic_sum4 <=
signed'(position[len*(21)-1:len*(21-1)])+signed'(position[len*(22)-1:len*(22-1)])+signed'(position[len*(23
)-1:len*(23-1)])+signed'(position[len*(24)-1:len*(24-1)])+signed'(position[len*(25)-1:len*(25-1)]);
               classic_sum5 <=
signed'(position[len*(26)-1:len*(26-1)])+signed'(position[len*(27)-1:len*(27-1)])+signed'(position[len*(28
)-1:len*(28-1)])+signed'(position[len*(29)-1:len*(29-1)])+signed'(position[len*(30)-1:len*(30-1)]);
               classic_sum6 <=
signed'(position[len*(31)-1:len*(31-1)])+signed'(position[len*(32)-1:len*(32-1)])+signed'(position[len*(33
)-1:len*(33-1)])+signed'(position[len*(34)-1:len*(34-1)])+signed'(position[len*(35)-1:len*(35-1)]);
               classic_sum7 <=
signed'(position[len*(36)-1:len*(36-1)])+signed'(position[len*(37)-1:len*(37-1)])+signed'(position[len*(38
)-1:len*(38-1)])+signed'(position[len*(39)-1:len*(39-1)])+signed'(position[len*(40)-1:len*(40-1)]);
               classic_sum8 <=
signed'(position[len*(41)-1:len*(41-1)])+signed'(position[len*(42)-1:len*(42-1)])+signed'(position[len*(43
)-1:len*(43-1)])+signed'(position[len*(44)-1:len*(44-1)])+signed'(position[len*(45)-1:len*(45-1)]);
               classic_sum9 <=
signed'(position[len*(46)-1:len*(46-1)])+signed'(position[len*(47)-1:len*(47-1)])+signed'(position[len*(48
)-1:len*(48-1)])+signed'(position[len*(49)-1:len*(49-1)])+signed'(position[len*(50)-1:len*(50-1)]);
               step <= step + 1;
            end
            1: begin
                intermediate0 <= classic_sum0 + classic_sum1 + classic_sum2 + classic_sum3 + classic_sum4;
                intermediate1 <= classic_sum5 + classic_sum6 + classic_sum7 + classic_sum8 + classic_sum9;
                step <= step + 1;
            end
            2: begin
                out <= intermediate0 + intermediate1;
                step <= 0;
            end
            default: step <= 0;
        endcase
    end
endmodule

//
// End Time Evolution Helper Modules
//////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////
// Infer Epicenter Helper Modules
//

/**
 Module that takes in 2 signals that represent the "time" when
  the corresponding hardware was triggered.
  Assumes the "length" is 50 "particles"
  Assumes that the maximum time for a wave to cross the board is ~0.75 ms or 750 microsec
  The ADC takes 416 clk cycles to convert all 4 signals

  bottom - x1, y1
  top - x2, y2

  If the difference between each piezo being triggered is really high
  we should just assume the wave traveled across the LENGTH of the board

  Follows these equations:
    (L-2d)/t = V
    Where L is Length
    t is the time between triggers
    d is the distance away from the piezo triggered first
**/
module time_diff(input clk,
                 input [9:0] bottom,
                 input [9:0] top,
                 output logic [5:0] position);
    logic [9:0] MAX_TIME = 400;
    logic [9:0] LENGTH = 50;

    logic [2:0] step = 0;
    logic [9:0] intermediate_A0, intermediate_B0,
                intermediate_A1, intermediate_B1,
                intermediate_A2, intermediate_B2,
                intermediate_A3, intermediate_B3,
                intermediate_A4, intermediate_B4;

    always_ff @(posedge clk) begin
        case (step)
            0: begin
                intermediate_A0 <= top - bottom;
                intermediate_B0 <= bottom - top;
                step <= step + 1;
            end
            1: begin
                /*
                   400 % 50 = 8 = 2^3
                   bit shifting by 3 is easier and less expensive than mult/div
                */
                intermediate_A1 <= intermediate_A0>>3;
                intermediate_B1 <= intermediate_B0>>3;
                step <= step + 1;
            end
            2: begin
                intermediate_A2 <= LENGTH - intermediate_A1;
                intermediate_B2 <= LENGTH - intermediate_B1;
                step <= step + 1;
            end
            3: begin
                intermediate_A3 <= intermediate_A2>>1;
                intermediate_B3 <= intermediate_B2>>1;
                step <= step + 1;
            end
            4: begin
                if (top > bottom) begin
                    position <= (intermediate_A0 < MAX_TIME) ? LENGTH - intermediate_A3 : LENGTH;
                end else if (bottom > top) begin
                    position <= (intermediate_B0  < MAX_TIME) ? intermediate_B3 : 0;
                end else begin
                    position <= 25;
                end
                step <= 0;
            end
            default: step <= 0;
        endcase
    end
endmodule

/**
`   Signal_trigger goes high at END of clock cycle
**/
module signal_trigger(input signed [11:0] lower, upper, signal,
                      output logic signal_triggered);
    always_comb begin
        signal_triggered = (signal < lower) | (upper < signal);
    end
endmodule

/**
    Outputs the average (positive) of the 4 digital signals inputed.
    Updates in 3 clock cycles
**/
module average_signal(input clk,
                      input signed [11:0] x1, x2, y1, y2,
                      output logic signed [11:0] average);
    logic signed [13:0] intermediate;
    logic signed [11:0] x1temp, x2temp, y1temp, y2temp;
    always_ff @(posedge clk) begin
        x1temp <= (x1 < 0) ? ~(x1-1) : x1;
        y1temp <= (y1 < 0) ? ~(y1-1) : y1;
        x2temp <= (x2 < 0) ? ~(x2-1) : x2;
        y2temp <= (y2 < 0) ? ~(y2-1) : y2;
        intermediate <= (x1temp+x2temp+y1temp+y2temp)>>2;
        average <= intermediate;
    end
endmodule
//
// End Infer Epicenter Helper Modules
//////////////////////////////////////////////////////////////////
```

```
/////////////////////////////////////////////////////////////////
// Top Level Helper Modules
//  Most of these modules were taken from previous labs
//

//PWM generator for audio generation!
module pwm (input clk_in, input rst_in, input [7:0] level_in, output logic pwm_out);
  logic [7:0] count;
  assign pwm_out = count<level_in;
  always_ff @(posedge clk_in)begin
    if (rst_in) count <= 8'b0;
    else count <= count+8'b1;
  end

endmodule

///////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module (video version - 24 bits)
//
///////////////////////////////////////////////////////////////////////////////

module debounce (input reset_in, clock_in, noisy_in,
                 output reg clean_out);

   reg [19:0] count;
   reg new_input;

   always_ff @(posedge clock_in)
     if (reset_in) begin
        new_input <= noisy_in;
        clean_out <= noisy_in;
        count <= 0; end
     else if (noisy_in != new_input) begin new_input<=noisy_in; count <= 0; end
     else if (count == 650000) clean_out <= new_input;
     else count <= count+1;

endmodule

module seven_seg_controller(input               clk_in,
                            input               rst_in,
                            input [31:0]        val_in,
                            output logic[7:0]   cat_out,
                            output logic[7:0]   an_out
    );

    logic[7:0]      segment_state;
    logic[31:0]     segment_counter;
    logic [3:0]     routed_vals;
    logic [6:0]     led_out;

    binary_to_seven_seg my_converter ( .bin_in(routed_vals), .led_out(led_out));
    assign cat_out = ~led_out;
    assign an_out = ~segment_state;

    // Chooses one of the 8 LED displays to provide power to
    always_comb begin
        case(segment_state)
            8'b0000_0001:   routed_vals = val_in[3:0];
            8'b0000_0010:   routed_vals = val_in[7:4];
            8'b0000_0100:   routed_vals = val_in[11:8];
            8'b0000_1000:   routed_vals = val_in[15:12];
            8'b0001_0000:   routed_vals = val_in[19:16];
            8'b0010_0000:   routed_vals = val_in[23:20];
            8'b0100_0000:   routed_vals = val_in[27:24];
            8'b1000_0000:   routed_vals = val_in[31:28];
            default:        routed_vals = val_in[3:0];
        endcase
    end

    // Increments the segment counter for 100000 clock cycles,
    // then left shifts the segment state.
    always_ff @(posedge clk_in)begin
        if (rst_in)begin
            segment_state <= 8'b0000_0001;
            segment_counter <= 32'b0;
        end else begin
            if (segment_counter == 32'd100_000)begin
                segment_counter <= 32'd0;
                segment_state <= {segment_state[6:0],segment_state[7]};
            end else begin
                segment_counter <= segment_counter +1;
            end
        end
    end

endmodule //seven_seg_controller

module binary_to_seven_seg(
                            input[3:0] bin_in,
                            output logic[6:0] led_out
);

    logic num0, num1, num2, num3,
          num4, num5, num6, num7,
          num8, num9, numA, numB,
          numC, numD, numE, numF;

    assign num0 = !bin_in[3] && !bin_in[2] && !bin_in[1] && !bin_in[0];
    assign num1 = !bin_in[3] && !bin_in[2] && !bin_in[1] && bin_in[0];
    assign num2 = !bin_in[3] && !bin_in[2] && bin_in[1] && !bin_in[0];
    assign num3 = !bin_in[3] && !bin_in[2] && bin_in[1] && bin_in[0];
    assign num4 = !bin_in[3] && bin_in[2] && !bin_in[1] && !bin_in[0];
    assign num5 = !bin_in[3] && bin_in[2] && !bin_in[1] && bin_in[0];
    assign num6 = !bin_in[3] && bin_in[2] && bin_in[1] && !bin_in[0];
    assign num7 = !bin_in[3] && bin_in[2] && bin_in[1] && bin_in[0];
    assign num8 = bin_in[3] && !bin_in[2] && !bin_in[1] && !bin_in[0];
    assign num9 = bin_in[3] && !bin_in[2] && !bin_in[1] && bin_in[0];
    assign numA = bin_in[3] && !bin_in[2] && bin_in[1] && !bin_in[0];
    assign numB = bin_in[3] && !bin_in[2] && bin_in[1] && bin_in[0];
    assign numC = bin_in[3] && bin_in[2] && !bin_in[1] && !bin_in[0];
    assign numD = bin_in[3] && bin_in[2] && !bin_in[1] && bin_in[0];
    assign numE = bin_in[3] && bin_in[2] && bin_in[1] && !bin_in[0];
    assign numF = bin_in[3] && bin_in[2] && bin_in[1] &&bin_in[0];

    // Use inverse logic to reduce number of terms
    assign led_out[0] = !(num1 || num4 || numB || numD);
    assign led_out[1] = !(num5 || num6 || numB || numC || numE || numF);
    assign led_out[2] = !(num2 || numC || numE || numF);
    assign led_out[3] = !(num1 || num4 || num7 || numA || numF);
    assign led_out[4] = !(num1 || num3 || num4 || num5 || num7 || num9);
    assign led_out[5] = !(num1 || num2 || num3 || num7 || numD);
    assign led_out[6] = !(num0 || num1 || num7 || numC);

endmodule //binary_to_hex

//
// End Top Level Helper Modules
//////////////////////////////////////////////////////////////////
```