# 6.111 Final Report - Voice Controlled Game Sprite

Aakanksha Sarda and William Huffman

December 11, 2013

## Project Overview

The primary goal of this project was to implement a system which produces a display of a game sprite on a computer monitor that responds to voice commands given by the user. This specific implementation was to use four spoken directional commands ("up", "down", "left", and "right") as well as one additional movement-halting command ("stop") to drive a pixel representation of a car on a computer screen. The completed project did not entirely attain this goal: the visualization component works as intended on hardware, and the speech component processes simulated inputs correctly in software, but the implementation of the speech component in hardware was not able to interpret real data correctly.

The motivation behind creating such a system was that speech is a very natural way for humans to communicate with a system (as it is a primary means of communication between humans), and can often be faster than other input methods, such as a keyboard. Specifically, an FPGA implementation is sensible because the system must, for every voice input, do computation for each word in its vocabulary to determine the likelihood of the voice input being that word. This is a very parallel system, and thus lends itself to a hardware implementation. Such a system was also certainly considered feasible, as many relatively simple software implementations of voice recognition exist, with the complexity of the system scaling roughly as the generality of the problem to be solved (i.e. recognizing commands by multiple people with accents is harder than recognizing commands by one person). By restricting this project to five commands spoken by one person, much of the complexity of general speech recognition is avoided.

At a high level, this project displays a maze, a car, and a finish line on the computer screen. The car's position on the screen was to be controlled by five voice commands, with four directions ('up', 'down', 'left', and 'right') causing the car to move in those directions, and a 'stop' command stopping the car. If the car hits a wall, it will stop movement in that direction. If the car crosses the finish line, a done screen will be briefly displayed, before resetting the system. The purpose of this visualization was to give visual feedback about the accuracy and effectiveness of the underlying speech recognition system, and is sensible as controlling video game characters is one major application of speech recognition.

In addition to requiring the use of the FPGA to implement the logic component of this project, a mic was required to record user input and a computer monitor was required to display the output of the FPGA. Both the mic and the computer monitor are part of the standard 6.111 lab kit, and have been used successfully. The mic was incorporated through the FPGA's audio microphone input in a way similar to that used in lab 5. The computer monitor received data via the VGA output on the FPGA in a way similar to that used in lab 3.

The bulk of the project was broken into two major components: a speech recognition component and a visualization component (See Figure 1 on page 2 for the block diagram). The speech recognition component takes as input the audio in from the mic, as well as a listening signal which indicates that the user is ready to input a command (this is done via holding down the enter key, as in lab 5). From this, the speech recognition module was to interpret the mic input into one of the five legal commands and output that command into a 3-bit bus connecting to the visualization module. The speech recognition module originally was also to output a 1-bit signal that is high for one clock cycle when a new legal command has been received, but this was found to be unneccessary if the current voice command was kept constant by the speech recognition module until the user changed commands.

The visualization component takes as input the 3-bit command bus from the speech recognition module, and produces as output XVGA data for a 1024x768 pixel display. Upon a new command from the speech recognition module, the visualization module waits until the current frame has been drawn and then updates its internal state to begin executing the new command. While the main purpose of the visualization module is to give a visual indicator that the speech recognition system is functioning correctly (in this case by moving a pixel car in the appropriate direction), two additional features were added in order to give the visual component some amount of interesting behavior and to emphasize the use of this system in display applications such as gaming. The first is a maze, drawn on the screen as a collection of colored rectangles. This maze acts as a solid object through which the car cannot pass, requiring the user to maneuver the car in multiple different directions. The second is a finish line, which provides a goal for the car to reach and a win-state (in this case, a screen reading 'DONE') once the car has reached that area on the screen. The full visualization output can be seen in Figure 2 on page 3.

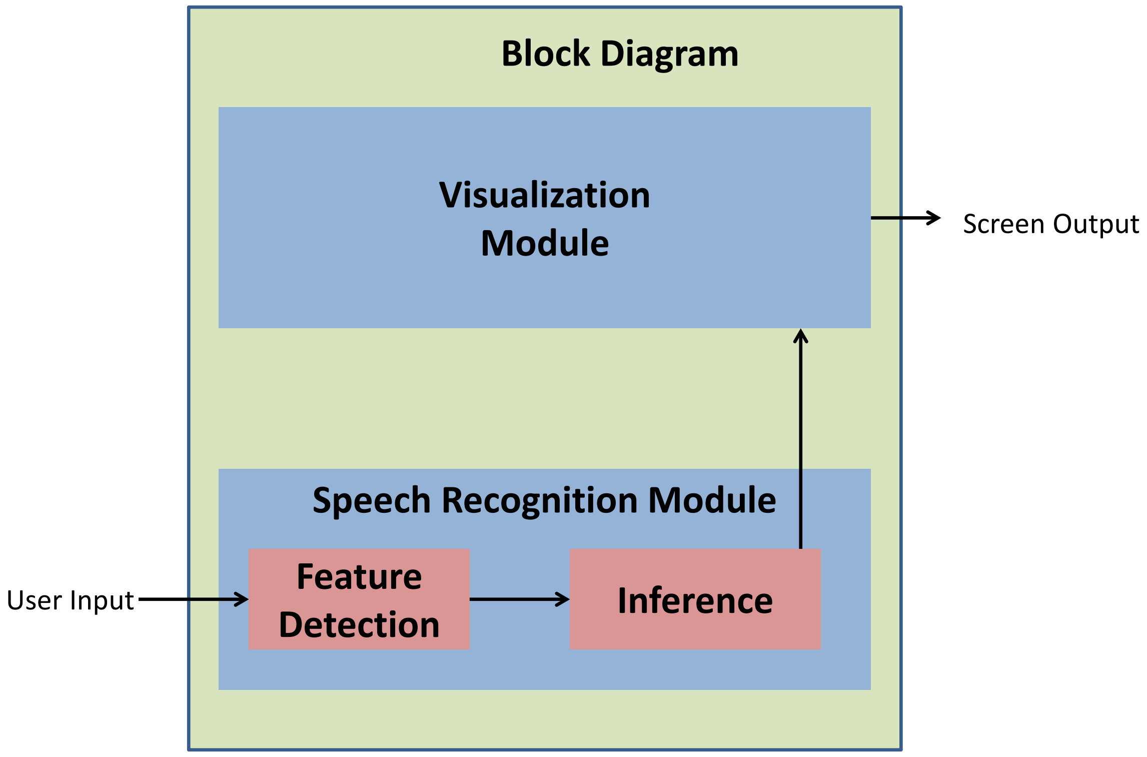

**Figure 1: Block Diagram Describing the Relationship between the Visualization and Speech Recognition Components**. The project uses a linear flow of data from the initial user input into the speech recognition component. Inside, the speech recognition component uses two main sub-components: one for detecting user input features, and the other for conducting inference on the user input using these features. The inference results are sent to the visualization component, which translates the identified command into an on-screen change in the car sprite's location.

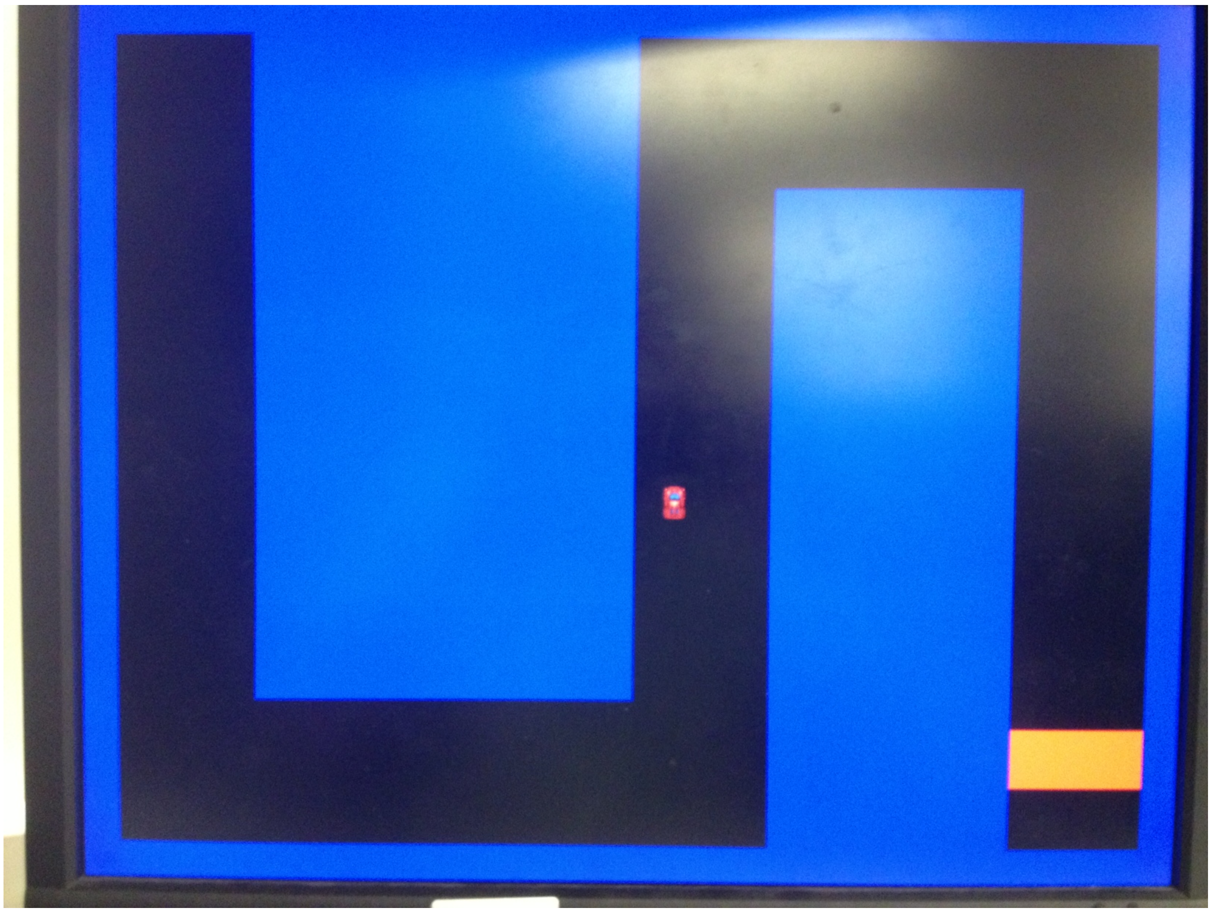

**Figure 2: Visualization Module Final Output**. This is the display output of the visualization module, containing a car which was to be driven by voice commands in different directions, a maze whose walls the car cannot pass through, and a finish line (here in orange) which will cause the module to display a screen with the word 'DONE' on it upon coming into contact with the car sprite.

## Speech Recognition Module - led by Aakanksha Sarda

(Instructor Note: Aakanksha dropped this class midway through the completion of this project. While she still contributed her code to the project, she was not able to contribute to writing the report. The following section uses a combination of descriptions from her used in the project proposal and explanations from myself (in the case of the GMM PDF module, which I implemented) in order to elaborate on the experience of implementing the project itself. However, this section is not as complete as the visualization section, due to the lack of first-person experience in developing the speech recognition module.)

The speech recognition module is broken into two major submodules: the feature detection and the inference components (See Figure 3 on page 6 for the wiring diagram). The feature detection component is responsible for taking audio input from the microphone, and converting it into a stream of feature vectors, which is an intermediate representation used by the inference component to decide which word was spoken.

### Feature Detection

The feature detection component attempted to decompose general voice input into a feature vector, which can be used by the inference module to determine which word has been spoken. It took as input a signal indicating that the user is speaking (from a button press), and audio sampled at 44.1 KHz from the labkit mic input. The audio recording module is responsible for down-sampling the audio to 16 KHz (which is standard for human speech signals), breaking it up into chunks of 512 samples, and passing the chunks on to the FFT module. The chunks were stored temporarily in the labkit BRAM, as lab 5. We used a pre-existing implementation for the FFT module. The output of the FFT module (the amplitudes of the different frequencies) were also temporarily stored in the labkit BRAM, and passed on to the peak detection module. The peak detection module made use of a common comparator tree structure to detect the 3 highest-amplitude frequency components in the speech sample. The tuple of frequencies corresponding to the 3 peaks comprise the "feature vector", which was passed on to the inference module.

## Inference

The inference component aimed to use a given feature vector to reason about the phoneme being spoken, and then use that to determine which word was being spoken. It took in a stream of feature vectors, and output a recognized command. The feature vectors were first stored in a buffer, which was later accessed by the decision module. The decision module also took pre-calculated, hard-coded, parameters, which correspond to each of the words in the vocabulary and were pre-calculated on a computer with MATLAB.

The decision module then passed on the received feature vectors and the the vocabulary parameters to the Gaussian Mixture Model PDF (GMM PDF) module. The GMM PDF module was doing a few mathematical operations on large data structures - in particular it computes a dot product between a feature vector and an inverse covariance matrix. Luckily, the inverse covariance matrix, average values, normalization factors, etc. were all parameters of the software model that was trained off of voice inputs, and so therefore could simply be hardcoded into each GMM PDF module. Unfortunately, this required passing in as parameters to a module a variable length two dimensional input matrix of variable precision, along with other vector quantities, which required a lot of careful planning in terms of figuring out how to best access this data. It was determined that the best way to do this was to pass each parameter in as a large (in the case of the inverse covariance matrix, large means on the order of 150+ bits) integer which would be parsed internally into a sequence of signed values and arranged into a logical matrix. Matters were complicated further by the requirement to upscale the precision on each of the involved constants by a factor of three, in order to ensure that operations involving them would not overflow inside of the module. Initially, overlaying the bits from the paramter into the vector buffers was disallowed by the Xilinx compiler, so this was done in two steps: the first involving reading in the values into logical vector data structures of normal precision, and the second involving transferring these values into a high precision register structure. Most difficult was sorting out which data type to use in each case, where the first ended up being wires initialized inside of a generated for loop and the second ended up being registers initialized from an initial begin statement.

Once the values for the GMM PDF were imported correctly into the module, the actual computation could be done either in the normal domain or the logarithmic domain. The normal domain was much more complicated, in that it required taking an exponential function, whereas the logarithmic domain merely required the dot product. However, working in the normal domain had the advantage of allowing the Hidden Markov Model (explained below) module to be able to be used, while the logarithmic domain calculation did not permit itself for use with the HMM. Because the logarithmic domain calculation is almost entirely a subset of the normal domain calculation, it was decided that the normal domain calculation would be implemented at first. This was relatively straightforward until it came time to take the exponential function of the computed value. In software, the arguments to the exponential tended to be quite small, so it was thought that a combination of a Taylor expansion and a thresholding of negative value magnitudes would be sufficient to do the computation. However, because of the precision requirements of dealing with physical hardware, as opposed to software where floating point values could be used, the coefficients used in the computation were required to be scaled by a large amount in order to accomodate the values as integers in the calculation. This meant that the exponential arguments were no longer small, and while it was possible to reverse the scaling after the taylor approximation, thereby reducing the problem back to a small-value approximation, there was not enough time to test this in software in order to determine that that scaling would still result in integer values that could be distinguished from each other (as opposed to undoing the scaling and ending up with all values being zero, because the dividing factor was so large in comparison with the calculation size). Because of this reason (and the fact that time constraints made implementing the HMM seem unlikely), it was decided that the normal domain calculation would be abandoned in favor of the logarithmic domain calculation.

An optional extension was to have a Hidden Markov Model module producing the similarity score instead. This extension was optional due to the comlexity of an HMM over a GMM, and due to the fact that the system would have already worked with a GMM. It performs better in the more general case of multiple users, accents, etc. of speech recognition; however, there was not enough time to attempt the HMM while attempting to get the GMM PDF based model to work correctly. In either case, the similarity scores were to be passed on to a common comparison tree module which would return the closest-matching word (the word with the highest similarity score). The decision module would then pass this matched word on to the Visualization module.

## Testing and Implementation

The testing and implementation of the speech recognition module followed a heirarchical strategy. First, the speech recognition algorithm was prototyped and tested in MATLAB. The MATLAB implementation was mostly to use fixed point arithmetic and use no built-in functions in order to model the FPGA setup as closely as possible, though the exponential function ended up still requiring floating point values, leading to the difficulties in the GMM PDF module described above. The algorithm was then be translated into Verilog, and implemented on the FPGA. Each module (audio recording, FFT, peak detection etc) was separately tested against the corresponding MATLAB implementation in software on simulated inputs, to make sure that it produced reasonable outputs (one such test on a GMM PDF submodule is shown in Figure 4 on page 6). Unfortunately testing on the actual hardware with real outputs did not begin until very late in the project, which heavily contributed to the project not reaching completion. This is due to the large amount of thresholding and optimization of a few parameters specific to the system which was required, including setting thresholds for significance and for filtering out background noise. These thresholds could not be tested until actual data was available, at which point the project was already assembled into the full speech recognition module and could not be tested separately. Thus, testing the speech recognition system was difficult and time consuming, leading to the project eventually running out of time.

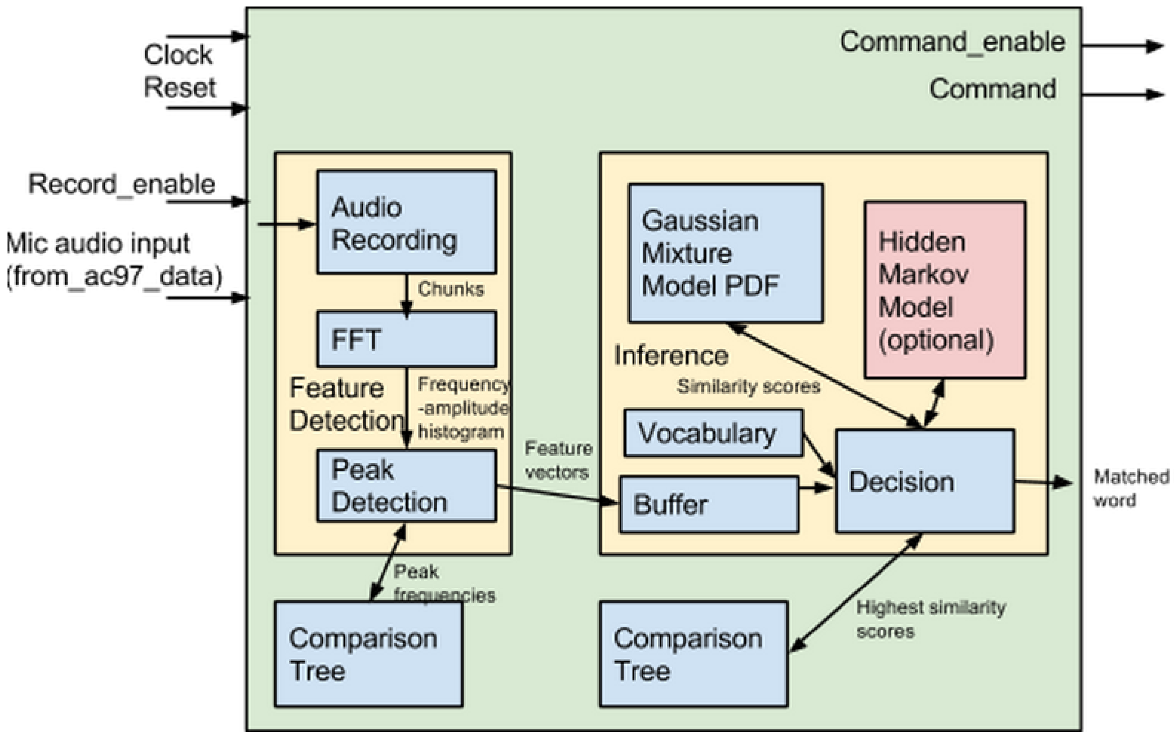

**Figure 3: Organization of the Speech Recognition Module into Feature Detection and Inference components**. The Feature Detection component is responsible for recording the audio input and processing it to produce an intermediate representation (corresponding to the dominant frequencies of the speech sample), which is used by the Inference module to decide which word from the vocabulary was spoken. The Comparison Tree module is a common parameterized module that was instantiated by both the Feature Detection and Inference components (with different parameters).

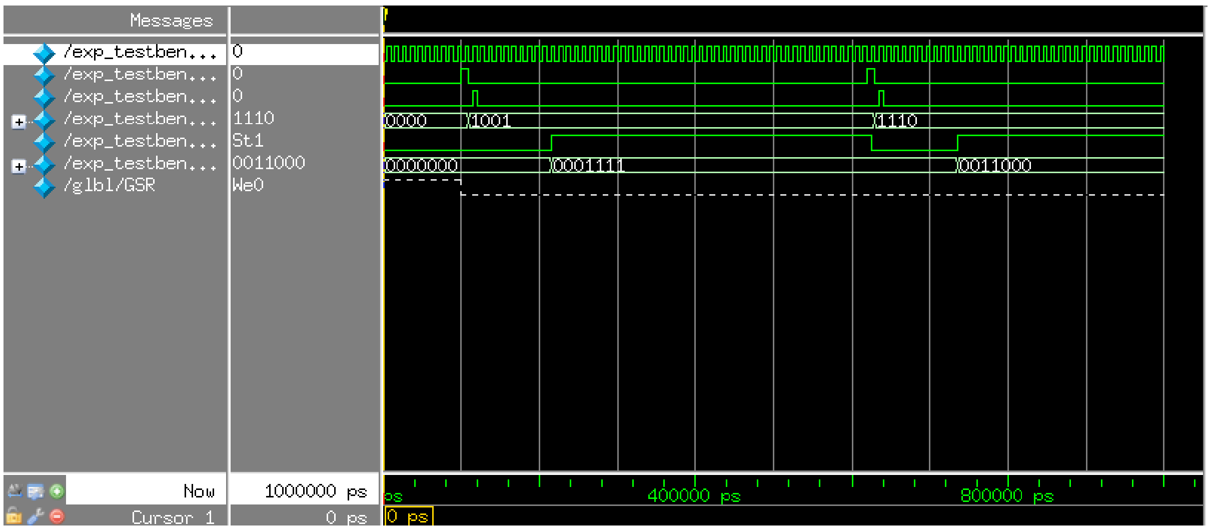

**Figure 4: GMM Test Results**. This figure shows the successful results of a test of an exponential module, the main component of the GMM PDF. The top signal is clock, followed downward by reset, input enable (telling the module to begin computation), the input data, output enable (telling the rest of the system that the module is done computing), and the output data. These results were obtained using known parameters, and the computation was done in parallel in Mathematica, after which the results were compared to ensure accuracy.

## Visualization Module - led by William Huffman

The visualization module is broken down into four major sub-components: the pixel logic module, the collision module, the VGA display module, and the pixel map modules (four cars - one for each direction, map, and done) (See Figure 5 on page 9 for the wiring diagram). The pixel logic module contains the simple state machine for transforming between commands from the speech recognition module and the state of the visual display. The collision module contains a set of rules for determining if a particular position of the car will result in a collision with a wall or the finish line, which is then sent back to the pixel logic module to be taken into account for determining the car's final position. The pixel logic module uses this position to offset the display of the car and the map in the appropriate places. The car, map, and done modules each contain a sub module which tells the pixel logic module what color should be displayed for each pixel on the screen. These sub modules are labeled ROMs, as it was originally intended that these would all be ROMs. The VGA Display module takes the output from the pixel logic module, incorporates vsync and hsync signals, and sends the entire frame buffer off to the display using the VGA connection.

In addition, two minor modules control the timing for the visualization component. A 65MHz clock module provides a sufficiently fast clock signal for the visualization component to produce XVGA signals. An input sanitization module synchronizes the input from the speech recognition module to the 65MHz clock for use in the visualization module.

### The Game Logic (pixel logic and collision modules)

The pixel logic module is a relatively simple FSM that was to change its state based on the speech recognition output, though by the completion of the project it could only change its state based on direct user input through the directional buttons on the FPGA. The state is then used to increment the position variables for the car in the x and y directions by an amount corresponding to a velocity constant.

The pixel logic FSM has six states: five of which correspond to the five input commands recognized by the speech recognition module ('up', 'down', 'left', 'right', and 'stop'), and one of which corresponds to the done screen to be displayed after the car crosses the finish line ('done') (see Figure 6 on page 10 for the state transition diagram). Each of the five directional states move freely into each other based on voice commands, and each can move to the 'done' state. The 'done' state triggers when the car crosses the finish line, and transitions to the 'stop' state after a short timeout. In the stop state, the car's offset does not change. In each of the four directional states, the car updates its position to have moved a number of pixels (the value 2 was used for testing and seemed to be fast enough, though faster values were possible) in that direction. This updated position is sent to the collision module, which responds with whether this new updated position is valid (if not, the updated position is reset to the previous position). Finally, the updated position is loaded into the pixel logic module to display the car in the appropriate position onscreen. Further, the sixth state is the done state, which sets set the done bit in the pixel logic module's state. The done state is triggered by the car crossing the finish line, and contains a counter which counts down two seconds, after which the pixel logic module resets to the car's initial position in the stop state.

The pixel logic module is also responsible for querying the car, map, and done modules to grab color data for each pixel to be drawn to the screen, and outputting a combination of these pixel colors to finally be drawn to the screen. As the car moves around the screen, the specific pixels queried for the car are offset from the pixels being drawn by an amount equal to the car's position. The car and map pixels were combined via a simple OR, as the car and map did not overlap. The done pixels were used to override the car and map pixels if the done bit was asserted in the pixel logic state output, otherwise they were be ignored.

Despite the pixel logic module doing two distinct functions, since these functions were both higher level functions which called other modules to perform most of the computation (such as the collision and image modules), it was easiest to peform these functions in one top-level pixel logic module. Further, this meant that the pixel logic module was actually quite basic to code and test, with few difficulties in the actual implementation.

The collision module takes as input a 19-bit position (encoding both the x and y position of the car) as well as a direction and velocity magnitude in order to compute the next position of the car, and gives a 1-bit output describing one of two possible outcomes: 0 is no collision, and 1 is a collision with a wall. The collision module is broken up into two parts, the first of which predicts the position of the car in the future based on its current position, direction, and velocity. This is used to query a second module, the collision ROM, in order to determine whether the new position will have collided with an object. The collision ROM was a mapping between addresses on the screen and single bits determining whether that area involved a collision. This was slightly different than the positions of the walls in the map, as the collision module worked in the configuration space of the car image. Because the car ROM addressed its position based on the top left corner of the image, the collision ROM's configuration space accounted for the possibilities of the image of the car colliding with a wall, while the addressed bit (top left corner) did not collide.

While the name of collision ROM submodule implies that the address to collision bit mapping was a ROM lookup, in the final implementation this ended up not being the case. Ordinary ROMs in verilog are compiled using case statements, however using a case statement for each of 1024x768 pixels caused a very long compile time for the project, slowing down debugging and testing. In order to solve this, the module switched to using Xilinx's proprietary ROM generation tool, which created a pre-written ROM for use on the FPGA. This caused the compilation time for the project to be fast, but the ROM was essentially a black box with no insight into how the ROM internals worked. Therefore, after struggling in a few different instances with attempting to debug the ROM, the Xilinx proprietary ROM was abandoned. Since the compile time issue involving case statements was only an issue with large ROMs such as the collision ROM, and since those large ROMs also happened to be made entirely of rectangular objects, it was decided that a ROM could be mimicked by a long sequence of conditionals strung together with the OR operator and assigned to the final bit output. Of course, writing that sequence of conditionals by hand would be difficult to do without typos, so a python script was created which generated this operation in verilog based on a more readable set of input rectangles.

## Drawing to the Screen (the VGA display and car, map, and done modules)

The car, map, and done modules are all image modules which contain information about pixel colors at different locations in the image. The car module contains pixel data for representing a car on screen (this is actually four sets of pixel data, one set for each direction that the car could be facing), the map module contains pixel data for representing the map on the screen, and the done module contains pixel data for representing a done screen upon crossing the finish line. Each pixel location contains 8 bits for each of red, green, and blue for the car image; and one bit for distinguishing image from transparent background for the done and map modules. It was originally planned that the car images would only use 4 bits for each color, but the python script that was created for transforming images into ROMs had difficulty selecting the correct downscaling of precision to use in order to preserve image fidelity. While the images could have simply been downscaled in an image manipulation program, since the ROMs were all made via a python script, it was easier to just change the number of bits used to 8, after which the ROMs worked perfectly. The smaller car and done modules were implemented as case statement ROMs, but the maze module ran into the same problems that were mentioned above in the collision module. The same solution was pursued as well, causing the maze ROM module to be a composition of conditionals and OR statements produced by a python script.

The VGA Display module is a basic module which takes in the combined pixel data from the pixel logic module, incorporates vsync and hsync signals for VGA communication, and outputs the final signal through the VGA Display. As this module is very similar to that used in lab three, testing was minimal and merely involved running the pixel logic module output through the VGA Display module to ensure that the correct image is displayed on the monitor.

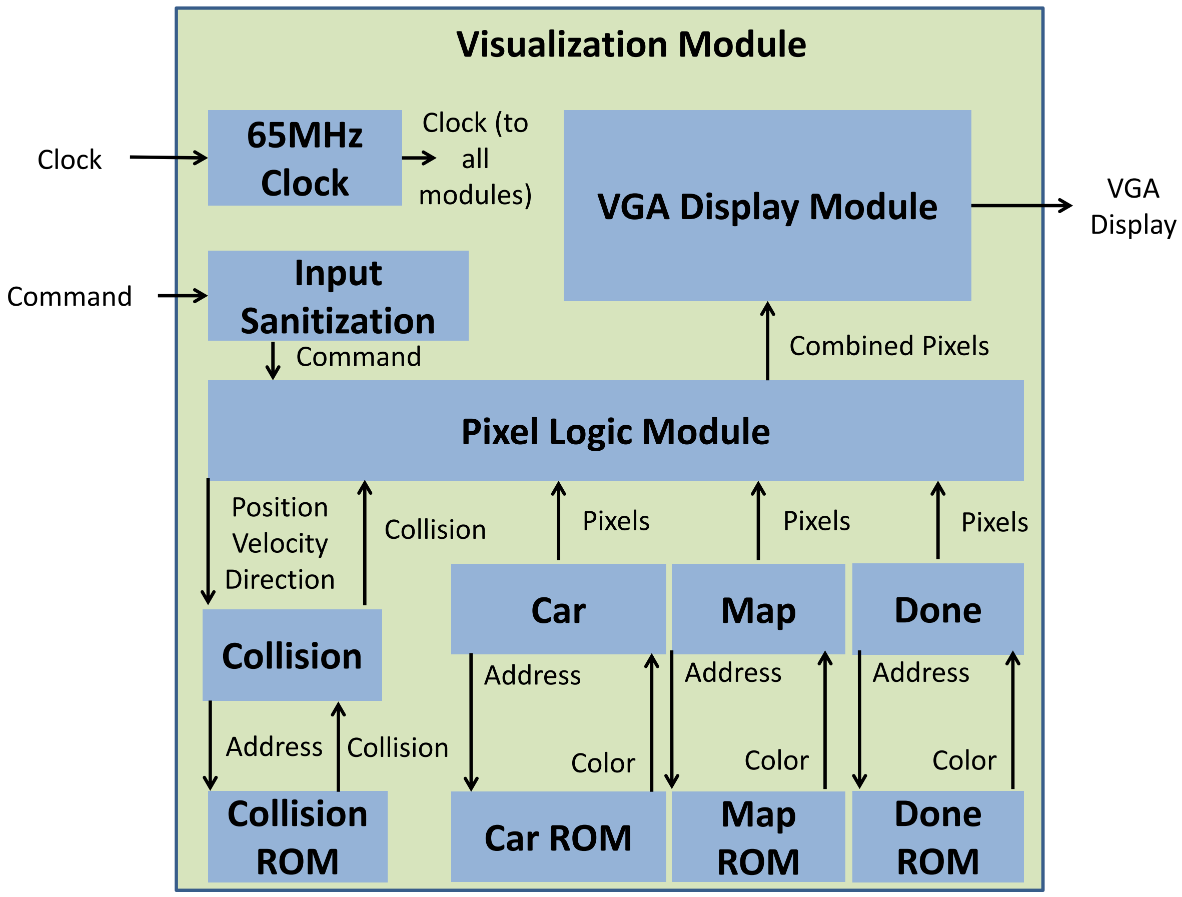

**Figure 5: Organization of the Visualization Component into Modules Handling Movement Logic and Display**. The input from the speech recognition component is sanitized and sent into the pixel logic module, which uses its internal FSM in conjunction with a collision detection module to determine where the car sprite should move. This position, in terms of an offset for the car sprite to be displayed on screen, is sent to the VGA Display modules to be rendered appropriately to the screen. The 65MHz clock module produces a sufficiently fast clock signal to output 1024x768 VGA frames.

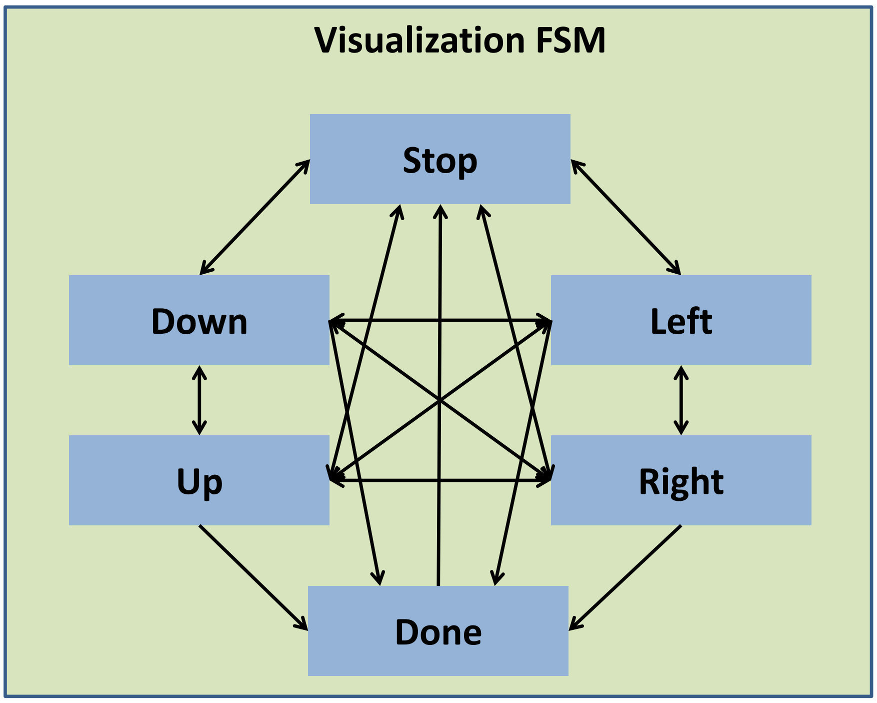

**Figure 6: Pixel Logic Module FSM**. The pixel logic module contains a simple FSM for determining the direction that the displayed car should travel. The FSM states correspond to the direction of the car, as well as whether or not the game is over. Each state can transition into any other state (in reality, the stop state will never transition into the done state, though it's not explicitly disallowed), except for the done state, which always returns to the stop state.

## Conclusion

The voice controlled game sprite project consisted of two main submodules: a speech recognition system and a visualization system. The visualization system was in charge of controlling the game logic and displaying the state of the system to the screen in order to give visual feedback that a voice command had been interpreted. The speech recognition system was in charge of taking input from the mic and translating it into one of five different possible commands. While the visualization system was able to be built and tested to completion, due to a delay in testing the speech recognition system in hardware the project as a whole was not able to be debugged in time for the project to finish. Therefore, an obvious improvement would be to begin testing the hardware of the speech recongition system on real data sooner, to allow more time to debug at a modular rather than a global level. Should this pursuit be accomplished, an extension of this project is to use the full HMM system in the inference submodule. This would allow many other commands from different speakers, with different accents, etc. to be added into the model relatively easily. As a whole, while this project fell just short of completion, it still provided many valuable lessons about digital programming, especially concerning dealing with hardware precision and data structure size; and due to the tough requirements of parallelization in speech processing for large vocabularies, coding a voice recognition system in hardware is still a worthwhile goal.

## Code Samples

(This section contains several samples of code used in the project, such as the main code file, the gmm pdf module, etc; as well as some of the python scripts we wrote to generate appropriate verilog modules for the project. For the full code (which is quite large), all of the python scripts, and/or the matlab simulation code, please contact us at whuffman@mit.edu.)

```
*******************************

module exponential #(parameter MUS = 100'd0, SIGMANUM = 500'd0, FINALSUB = 0,
WIDTH = 6, PRECISION = 8)
   (input clk, reset, input_enable,
    input [PRECISION*WIDTH-1:0]  data_point,
    output     output_enable,
    output [3*PRECISION:0] pdf);

 //PACKING PROCEDURE:
 //MUS = {mu_n, mu_n-1, ..., mu_1, mu_0}, each mu is two's complement
and
 // PRECISION+1 bits long, with n=WIDTH
 //SIGMANUM = {sigma_n,n; sigma_n,n-1;... sigma_n,0; sigma_n-1,n;...
sigma_0,0}
 //each sigma_row,column is two's complement and PRECISION+1 bits long,
with n=WIDTH
 //data_point = {x_n, x_n-1,... x_0}, each x_n is unsigned and PRECISION
bits long,
 //with n=WIDTH

   reg [9:0]     iter; //iterator for loops
   reg [9:0]     iter2; //iterator for loops (also used for state)
reg [9:0]     iter3;
reg [9:0]     iter4;
reg [9:0]     iter5;
reg [9:0]     iter6;
reg [9:0]     iter7;
reg [9:0]     iter8;
reg [9:0] iterreset;
reg [9:0] iterreset2;
   wire [PRECISION-1:0]    initdata [WIDTH-1:0]; //data for grabbing data_ponit
from input
   wire signed [PRECISION:0] initmu [WIDTH-1:0]; //data for grabbing mus from
parameter
   wire signed [PRECISION:0] initsigma [WIDTH*WIDTH-1:0]; //data for grabbing
sigmas from parameter
   reg signed [3*PRECISION:0] edata [WIDTH-1:0]; //precision-extended data point
   reg signed [3*PRECISION:0] emus [WIDTH-1:0]; //precision-extended mus
   reg signed [3*PRECISION:0] sigmas [WIDTH*WIDTH-1:0]; //precision-extended
sigmas

genvar i;
genvar j;
generate for(i = 0; i < WIDTH; i = i+1) begin:instmem
assign initdata[i] = data_point[(PRECISION*(i+1) - 1) -:
PRECISION];
assign initmu[i] = MUS[((PRECISION+1)*(i+1)-1) -: PRECISION+1];
for(j = 0; j < WIDTH; j = j+1) begin:instmem2
```

```
assign initsigma[i*WIDTH+j] =
SIGMANUM[((PRECISION+1)*(i*WIDTH+j+1)-1) -: PRECISION+1];
end
end endgenerate

always @(posedge input_enable) begin
for(iter = 0; iter < WIDTH; iter = iter + 1) begin
edata[iter] = \$unsigned(initdata[iter]);
emus[iter] = \$signed(initmu[iter]);
for(iter2 = 0; iter2 < WIDTH; iter2 = iter2 + 1) begin
sigmas[iter*WIDTH+iter2] =
\$signed(initsigma[iter*WIDTH+iter2]);
end
end
   end

   reg [9:0]        counter = 10'd0; //counter to keep track of state
   reg        done = 1'b0; //is high when the module is finished
   reg signed [3*PRECISION:0] subs [WIDTH-1:0]; //holds (x-mu)
   reg        matrix_state = 1'b0; //state for the matrix
multiplication
   reg signed [3*PRECISION:0] matrix_final [WIDTH-1:0]; //holds S*(x-mu)
   reg signed [3*PRECISION:0] matrix_interm [WIDTH*WIDTH-1:0]; //holds
intermediate values for matrix multiplication
   reg signed [3*PRECISION:0] dot_final; //holds (x-mu)^T*S*(x-mu)
   reg signed [3*PRECISION:0] exponent; //holds -1/2*(x-mu)^T*S*(x-mu)
   reg signed [3*PRECISION:0] taylor [2:0]; //holds {1/2*x^2, x, 1}
   reg signed [3*PRECISION:0] taylor_final; //holds COEFF*(1+x+1/2*x^2)
   reg [3*PRECISION:0]        out = 0; //holds the pdf

   assign pdf = out;
   assign output_enable = done;

   always @(posedge clk) begin
      if(reset) begin
 counter <= 10'd0;
 done <= 1'b0;
 for(iterreset = 0; iterreset < WIDTH; iterreset = iterreset+1) begin
matrix_final[iterreset] <= 0;
subs[iterreset] <= 0;
for(iterreset2 = 0; iterreset2 < WIDTH; iterreset2 =
iterreset2+1) begin
matrix_interm[iterreset*WIDTH+iterreset2] <= 0;
end
end
    end else if(input_enable) begin
counter <= 10'd6;
done <= 1'b0;
    end else if(done) begin
//Do nothing
    end else if(counter != 10'd0) begin
 case(counter)
   10'd6: begin //Subtraction: x-mu
      for(iter3 = 0; iter3 < WIDTH; iter3 = iter3 + 1) begin
 subs[iter3] <= edata[iter3] - emus[iter3];
      end
      counter <= counter - 10'd1;
      matrix_state <= 1'b0;
   end
   10'd5: begin //Matrix Multiplication: S*(x-mu)
      case(matrix_state)
1'b0: begin
for(iter4 = 0; iter4 < WIDTH; iter4 =
iter4 + 1) begin
for(iter5 = 0; iter5 < WIDTH;
iter5 = iter5 + 1) begin
matrix_interm[iter4*WIDTH+iter5] <=
sigmas[iter4*WIDTH+iter5]*subs[iter5];
end
end
matrix_state <= 1'b1;
iter6 <= 0;
for(iter7 = 0; iter7 < WIDTH; iter7 =
iter7 + 1) begin
matrix_final[iter7] <= 0;
end
end
1'b1: begin
if(iter6 < WIDTH) begin
for(iter8 = 0; iter8 < WIDTH;
iter8 = iter8 + 1) begin
matrix_final[iter8] <=
matrix_final[iter8] + matrix_interm[iter8*WIDTH+iter6];
end
iter6 <= iter6 + 1;
end else begin
iter6 <= 0;
counter <= counter - 10'd1;
dot_final <= 0;
end
end
      endcase
   end
   10'd4: begin //Dot Product: (x-mu)^T*(S*(x-mu))
      if(iter6 < WIDTH) begin
dot_final <= dot_final +
subs[iter6]*matrix_final[iter6];
iter6 <= iter6 + 1;
      end else counter <= counter - 10'd1;
   end
   10'd3: begin //Negation and Bit-Shift: -1/2*(x-mu)^T*(S*(x-mu))
      exponent = -(dot_final >>> 1);
      counter <= counter - 10'd2;
      iter6 <= 0;
   end
//    10'd2: begin //Taylor Series: COEFF*(1+x+1/2*x^2)
//       case(iter6)
// 0: begin
//    taylor[0] <= 1;
//    taylor[1] <= exponent;
//    taylor[2] <= (exponent*exponent) >> 1;
//    iter6 <= 1;
// end
// 1: begin
//    taylor_final <= COEFF*(taylor[0] + taylor[1] + taylor[2]);
//    counter <= counter - 10'd1;
// end
//       endcase
//    end
   10'd1: begin //output result
      out <= exponent - FINALSUB;
      done <= 1'b1;
      counter <= counter - 10'd1;
   end
   default: counter <= 10'd6;
 endcase // case (counter)
      end // if (counter != 10'd0)
   end // always @ (posedge clk)
endmodule

*******************************

'timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
//
// Company:
// Engineer:
//
// Create Date:    13:56:59 12/09/2013
// Design Name:
// Module Name:    gmm_pdf
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
//
module gmm_pdf #(parameter MUS1 = 100'd0, MUS2 = 100'd0, MUS3 = 100'd0, SIGMAS1
= 500'd0, SIGMAS2 = 500'd0, SIGMAS3 = 500'd0, SUB1 = 0, SUB2 = 0, SUB3 = 0,
WIDTH = 3, PRECISION = 16)
(input clk, reset, input_enable,
input [PRECISION*WIDTH-1:0] data_point,
output output_enable,
output [3*PRECISION:0] pdf);

wire [2:0] output_enables;
wire [3*PRECISION:0] pdfs [2:0];

reg out_en = 0;
reg [3*PRECISION:0] out;
```

```
assign output_enable = out_en;
assign pdf = out;

exponential #(.MUS(MUS1), .SIGMANUM(SIGMAS1), .FINALSUB(SUB1),
.WIDTH(WIDTH), .PRECISION(PRECISION))
exp1(
.clk(clk),
.reset(reset),
.input_enable(input_enable),
.data_point(data_point),
.output_enable(output_enables[0]),
.pdf(pdfs[0])
);

exponential #(.MUS(MUS2), .SIGMANUM(SIGMAS2), .FINALSUB(SUB2),
.WIDTH(WIDTH), .PRECISION(PRECISION))
exp2(
.clk(clk),
.reset(reset),
.input_enable(input_enable),
.data_point(data_point),
.output_enable(output_enables[1]),
.pdf(pdfs[1])
);

exponential #(.MUS(MUS3), .SIGMANUM(SIGMAS3), .FINALSUB(SUB3),
.WIDTH(WIDTH), .PRECISION(PRECISION))
exp3(
.clk(clk),
.reset(reset),
.input_enable(input_enable),
.data_point(data_point),
.output_enable(output_enables[2]),
.pdf(pdfs[2])
);

always @(posedge clk) begin
if(output_enables == 3'd7) begin
out_en = 1'b1;
if((pdfs[0] < pdfs[1]) && (pdfs[0] < pdfs[2]))
out = pdfs[0];
else if(pdfs[1] < pdfs[2])
out = pdfs[1];
else
out = pdfs[2];
end
end
endmodule

*******************************


`default_nettype none

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

```
// Switch Debounce Module
//
///////////////////////////////////////////////////////////////////////////////

module debounce (
  input wire reset, clock, noisy,
  output reg clean
);
  reg [18:0] count;
  reg new;

  always @(posedge clock)
    if (reset) begin
      count <= 0;
      new <= noisy;
      clean <= noisy;
    end
    else if (noisy != new) begin
      // noisy input changed, restart the .01 sec clock
      new <= noisy;
      count <= 0;
    end
    else if (count == 270000)
      // noisy input stable for .01 secs, pass it along!
      clean <= new;
    else
      // waiting for .01 sec to pass
      count <= count+1;

endmodule

///////////////////////////////////////////////////////////////////////////////
//
// bi-directional monaural interface to AC97
//
///////////////////////////////////////////////////////////////////////////////

module lab5audio (
  input wire clock_27mhz,
  input wire reset,
  input wire [4:0] volume,
  output wire [7:0] audio_in_data,
  input wire [7:0] audio_out_data,
  output wire ready,
  output reg audio_reset_b,   // ac97 interface signals
  output wire ac97_sdata_out,
  input wire ac97_sdata_in,
  output wire ac97_synch,
  input wire ac97_bit_clock
);

  wire [7:0] command_address;
  wire [15:0] command_data;
  wire command_valid;
  wire [19:0] left_in_data, right_in_data;
  wire [19:0] left_out_data, right_out_data;
  // wait a little before enabling the AC97 codec
  reg [9:0] reset_count;
  always @(posedge clock_27mhz) begin
    if (reset) begin
      audio_reset_b = 1'b0;
      reset_count = 0;
    end else if (reset_count == 1023)
      audio_reset_b = 1'b1;
    else
      reset_count = reset_count+1;
  end

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

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

  reg [7:0] out_data;
  always @ (posedge clock_27mhz)
    if (ready) out_data <= audio_out_data;
  assign audio_in_data = left_in_data[19:12];
  assign left_out_data = {out_data, 12'b000000000000};
  assign right_out_data = left_out_data;

  // generate repeating sequence of read/writes to AC97 registers
  ac97commands cmds(.clock(clock_27mhz), .ready(ready),
                    .command_address(command_address),
                    .command_data(command_data),
                    .command_valid(command_valid),
                    .volume(volume),
                    .source(3'b000));     // mic
endmodule

// assemble/disassemble AC97 serial frames
module ac97 (
  output reg ready,
  input wire [7:0] command_address,
  input wire [15:0] command_data,
  input wire command_valid,
  input wire [19:0] left_data,
  input wire left_valid,
  input wire [19:0] right_data,
  input wire right_valid,
  output reg [19:0] left_in_data, right_in_data,
  output reg ac97_sdata_out,
  input wire ac97_sdata_in,
  output reg ac97_synch,
  input wire ac97_bit_clock
);
  reg [7:0] bit_count;

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

  initial begin
    ready <= 1'b0;
    // synthesis attribute init of ready is "0";
    ac97_sdata_out <= 1'b0;
    // synthesis attribute init of ac97_sdata_out is "0";
    ac97_synch <= 1'b0;
    // synthesis attribute init of ac97_synch is "0";

    bit_count <= 8'h00;
    // synthesis attribute init of bit_count is "0000";
    l_cmd_v <= 1'b0;
    // synthesis attribute init of l_cmd_v is "0";
    l_left_v <= 1'b0;
    // synthesis attribute init of l_left_v is "0";
    l_right_v <= 1'b0;
    // synthesis attribute init of l_right_v is "0";

    left_in_data <= 20'h00000;
    // synthesis attribute init of left_in_data is "00000";
    right_in_data <= 20'h00000;
    // synthesis attribute init of right_in_data is "00000";
  end

  always @(posedge ac97_bit_clock) begin
    // Generate the sync signal
    if (bit_count == 255)
      ac97_synch <= 1'b1;
    if (bit_count == 15)
      ac97_synch <= 1'b0;

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

    // Latch user data at the end of each frame. This ensures that the
    // first frame after reset will be empty.
    if (bit_count == 255) begin
      l_cmd_addr <= {command_address, 12'h000};
      l_cmd_data <= {command_data, 4'h0};
      l_cmd_v <= command_valid;
      l_left_data <= left_data;
      l_left_v <= left_valid;
      l_right_data <= right_data;
      l_right_v <= right_valid;
    end

    if ((bit_count >= 0) && (bit_count <= 15))
      // Slot 0: Tags
      case (bit_count[3:0])
        4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
        4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
        4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
        4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
        4'h4: ac97_sdata_out <= l_right_v; // Right data valid
        default: ac97_sdata_out <= 1'b0;
      endcase
    else if ((bit_count >= 16) && (bit_count <= 35))
      // Slot 1: Command address (8-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;
    else if ((bit_count >= 36) && (bit_count <= 55))
      // Slot 2: Command data (16-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;
    else if ((bit_count >= 56) && (bit_count <= 75)) begin
      // Slot 3: Left channel
      ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
      l_left_data <= { l_left_data[18:0], l_left_data[19] };
    end
    else if ((bit_count >= 76) && (bit_count <= 95))
      // Slot 4: Right channel
      ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
    else
      ac97_sdata_out <= 1'b0;

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

  always @(negedge ac97_bit_clock) begin
    if ((bit_count >= 57) && (bit_count <= 76))
      // Slot 3: Left channel
      left_in_data <= { left_in_data[18:0], ac97_sdata_in };
    else if ((bit_count >= 77) && (bit_count <= 96))
      // Slot 4: Right channel
      right_in_data <= { right_in_data[18:0], ac97_sdata_in };
  end
endmodule

// issue initialization commands to AC97
module ac97commands (
  input wire clock,
  input wire ready,
  output wire [7:0] command_address,
  output wire [15:0] command_data,
  output reg command_valid,
  input wire [4:0] volume,
  input wire [2:0] source
);
```

```
  reg [23:0] command;

  reg [3:0] state;
  initial begin
    command <= 4'h0;
    // synthesis attribute init of command is "0";
    command_valid <= 1'b0;
    // synthesis attribute init of command_valid is "0";
    state <= 16'h0000;
    // synthesis attribute init of state is "0000";
  end

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

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

  always @(posedge clock) begin
    if (ready) state <= state+1;

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

///////////////////////////////////////////////////////////////////////////////
//
// generate PCM data for 750hz sine wave (assuming f(ready) = 48khz)
//
///////////////////////////////////////////////////////////////////////////////

module tone750hz (
  input wire clock,
  input wire ready,
  output reg [19:0] pcm_data
);
   reg [8:0] index;

   initial begin
      index <= 8'h00;
      // synthesis attribute init of index is "00";
      pcm_data <= 20'h00000;
      // synthesis attribute init of pcm_data is "00000";
   end

   always @(posedge clock) begin
      if (ready) index <= index+1;
   end

   // one cycle of a sinewave in 64 20-bit samples
   always @(index) begin
      case (index[5:0])
        6'h00: pcm_data <= 20'h00000;
        6'h01: pcm_data <= 20'h0C8BD;
        6'h02: pcm_data <= 20'h18F8B;
        6'h03: pcm_data <= 20'h25280;
        6'h04: pcm_data <= 20'h30FBC;
        6'h05: pcm_data <= 20'h3C56B;
        6'h06: pcm_data <= 20'h471CE;
        6'h07: pcm_data <= 20'h5133C;
        6'h08: pcm_data <= 20'h5A827;
        6'h09: pcm_data <= 20'h62F20;
        6'h0A: pcm_data <= 20'h6A6D9;
        6'h0B: pcm_data <= 20'h70E2C;
        6'h0C: pcm_data <= 20'h7641A;
        6'h0D: pcm_data <= 20'h7A7D0;
        6'h0E: pcm_data <= 20'h7D8A5;
        6'h0F: pcm_data <= 20'h7F623;
        6'h10: pcm_data <= 20'h7FFFF;
        6'h11: pcm_data <= 20'h7F623;
        6'h12: pcm_data <= 20'h7D8A5;
        6'h13: pcm_data <= 20'h7A7D0;
        6'h14: pcm_data <= 20'h7641A;
        6'h15: pcm_data <= 20'h70E2C;
        6'h16: pcm_data <= 20'h6A6D9;
        6'h17: pcm_data <= 20'h62F20;
        6'h18: pcm_data <= 20'h5A827;
        6'h19: pcm_data <= 20'h5133C;
        6'h1A: pcm_data <= 20'h471CE;
        6'h1B: pcm_data <= 20'h3C56B;
        6'h1C: pcm_data <= 20'h30FBC;
        6'h1D: pcm_data <= 20'h25280;
        6'h1E: pcm_data <= 20'h18F8B;
        6'h1F: pcm_data <= 20'h0C8BD;
        6'h20: pcm_data <= 20'h00000;
        6'h21: pcm_data <= 20'hF3743;
        6'h22: pcm_data <= 20'hE7075;
        6'h23: pcm_data <= 20'hDAD80;
        6'h24: pcm_data <= 20'hCF044;
        6'h25: pcm_data <= 20'hC3A95;
        6'h26: pcm_data <= 20'hB8E32;
        6'h27: pcm_data <= 20'hAECC4;
        6'h28: pcm_data <= 20'hA57D9;
        6'h29: pcm_data <= 20'h9D0E0;
        6'h2A: pcm_data <= 20'h95927;
        6'h2B: pcm_data <= 20'h8F1D4;
        6'h2C: pcm_data <= 20'h89BE6;
        6'h2D: pcm_data <= 20'h85830;
        6'h2E: pcm_data <= 20'h8275B;
        6'h2F: pcm_data <= 20'h809DD;
        6'h30: pcm_data <= 20'h80000;
        6'h31: pcm_data <= 20'h809DD;
        6'h32: pcm_data <= 20'h8275B;
        6'h33: pcm_data <= 20'h85830;
        6'h34: pcm_data <= 20'h89BE6;
        6'h35: pcm_data <= 20'h8F1D4;
        6'h36: pcm_data <= 20'h95927;
        6'h37: pcm_data <= 20'h9D0E0;
        6'h38: pcm_data <= 20'hA57D9;
        6'h39: pcm_data <= 20'hAECC4;
        6'h3A: pcm_data <= 20'hB8E32;
        6'h3B: pcm_data <= 20'hC3A95;
        6'h3C: pcm_data <= 20'hCF044;
        6'h3D: pcm_data <= 20'hDAD80;
        6'h3E: pcm_data <= 20'hE7075;
        6'h3F: pcm_data <= 20'hF3743;
      endcase // case(index[5:0])
   end // always @ (index)
endmodule

///////////////////////////////////////////////////////////////////////////////
/
////
//// 6.111 FPGA Labkit -- Template Toplevel Module
////
//// For Labkit Revision 004
//// Created: October 31, 2004, from revision 003 file
//// Author: Nathan Ickes, 6.111 staff
////
///////////////////////////////////////////////////////////////////////////////
/

module lab5   (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
       ac97_bit_clock,

       vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
       vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
       vga_out_vsync,

       tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
       tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
       tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

       tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
       tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
       tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
       tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

       ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
       ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

       ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
       ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

       clock_feedback_out, clock_feedback_in,

       flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
       flash_reset_b, flash_sts, flash_byte_b,

       rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

       mouse_clock, mouse_data, keyboard_clock, keyboard_data,

       clock_27mhz, clock1, clock2,

       disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
       disp_reset_b, disp_data_in,

       button0, button1, button2, button3, button_enter, button_right,
       button_left, button_down, button_up,

       switch,

       led,

       user1, user2, user3, user4,

       daughtercard,

       systemace_data, systemace_address, systemace_ce_b,
       systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

       analyzer1_data, analyzer1_clock,
         analyzer2_data, analyzer2_clock,
         analyzer3_data, analyzer3_clock,
         analyzer4_data, analyzer4_clock);

   output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
   input  ac97_bit_clock, ac97_sdata_in;

   output [7:0] vga_out_red, vga_out_green, vga_out_blue;
   output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
  vga_out_hsync, vga_out_vsync;

   output [9:0] tv_out_ycrcb;
   output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
  tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
  tv_out_subcar_reset;

   input  [19:0] tv_in_ycrcb;
```

```
  input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
 tv_in_hff, tv_in_aff;
  output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
 tv_in_reset_b, tv_in_clock;
  inout  tv_in_i2c_data;

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

  inout  [35:0] ram1_data;
  output [18:0] ram1_address;
  output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
  output [3:0] ram1_bwe_b;

  input  clock_feedback_in;
  output clock_feedback_out;

  inout  [15:0] flash_data;
  output [24:0] flash_address;
  output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
  input  flash_sts;

  output rs232_txd, rs232_rts;
  input  rs232_rxd, rs232_cts;

  input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

  input  clock_27mhz, clock1, clock2;

  output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
  input  disp_data_in;
  output  disp_data_out;

  input  button0, button1, button2, button3, button_enter, button_right,
 button_left, button_down, button_up;
  input  [7:0] switch;
  output [7:0] led;

  inout [31:0] user1, user2, user3, user4;

  inout [43:0] daughtercard;

  inout  [15:0] systemace_data;
  output [6:0]  systemace_address;
  output systemace_ce_b, systemace_we_b, systemace_oe_b;
  input  systemace_irq, systemace_mpbrdy;

  output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
analyzer4_data;
  output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

  ////////////////////////////////////////////////////////////////////////////
  //
  // I/O Assignments
```

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


   // Audio Input and Output
   assign beep= 1'b0;
   //lab5 assign audio_reset_b = 1'b0;
   //lab5 assign ac97_synch = 1'b0;
   //lab5 assign ac97_sdata_out = 1'b0;
   // ac97_sdata_in is an input

   // VGA Output
//   assign vga_out_red = 10'h0;
//   assign vga_out_green = 10'h0;
//   assign vga_out_blue = 10'h0;
//   assign vga_out_sync_b = 1'b1;
//   assign vga_out_blank_b = 1'b1;
//   assign vga_out_pixel_clock = 1'b0;
//   assign vga_out_hsync = 1'b0;
//   assign vga_out_vsync = 1'b0;

   // Video Output
   assign tv_out_ycrcb = 10'h0;
   assign tv_out_reset_b = 1'b0;
   assign tv_out_clock = 1'b0;
   assign tv_out_i2c_clock = 1'b0;
   assign tv_out_i2c_data = 1'b0;
   assign tv_out_pal_ntsc = 1'b0;
   assign tv_out_hsync_b = 1'b1;
   assign tv_out_vsync_b = 1'b1;
   assign tv_out_blank_b = 1'b1;
   assign tv_out_subcar_reset = 1'b0;

   // Video Input
   assign tv_in_i2c_clock = 1'b0;
   assign tv_in_fifo_read = 1'b0;
   assign tv_in_fifo_clock = 1'b0;
   assign tv_in_iso = 1'b0;
   assign tv_in_reset_b = 1'b0;
   assign tv_in_clock = 1'b0;
   assign tv_in_i2c_data = 1'bZ;
   // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
   // tv_in_aef, tv_in_hff, and tv_in_aff are inputs

   // SRAMs
   assign ram0_data = 36'hZ;
   assign ram0_address = 19'h0;
   assign ram0_adv_ld = 1'b0;
   assign ram0_clk = 1'b0;
   assign ram0_cen_b = 1'b1;
   assign ram0_ce_b = 1'b1;
   assign ram0_oe_b = 1'b1;
   assign ram0_we_b = 1'b1;
   assign ram0_bwe_b = 4'hF;
   assign ram1_data = 36'hZ;
   assign ram1_address = 19'h0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_clk = 1'b0;
   assign ram1_cen_b = 1'b1;
   assign ram1_ce_b = 1'b1;
   assign ram1_oe_b = 1'b1;
   assign ram1_we_b = 1'b1;
   assign ram1_bwe_b = 4'hF;
   assign clock_feedback_out = 1'b0;
   // clock_feedback_in is an input

   // Flash ROM
   assign flash_data = 16'hZ;
   assign flash_address = 24'h0;
   assign flash_ce_b = 1'b1;
   assign flash_oe_b = 1'b1;
   assign flash_we_b = 1'b1;
   assign flash_reset_b = 1'b0;
   assign flash_byte_b = 1'b1;
   // flash_sts is an input

   // RS-232 Interface
   assign rs232_txd = 1'b1;
   assign rs232_rts = 1'b1;
   // rs232_rxd and rs232_cts are inputs

   // PS/2 Ports
   // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

   // LED Displays
   assign disp_blank = 1'b1;
   assign disp_clock = 1'b0;
   assign disp_rs = 1'b0;
   assign disp_ce_b = 1'b1;
   assign disp_reset_b = 1'b0;
   assign disp_data_out = 1'b0;
   // disp_data_in is an input

   // Buttons, Switches, and Individual LEDs
   //lab5 assign led = 8'hFF;
   // button0, button1, button2, button3, button_enter, button_right,
   // button_left, button_down, button_up, and switches are inputs

   // User I/Os
   //assign user1 = 32'hZ;
   assign user2 = 32'hZ;
   assign user3 = 32'hZ;
   assign user4 = 32'hZ;

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

   // SystemACE Microprocessor Port
   assign systemace_data = 16'hZ;
   assign systemace_address = 7'h0;
   assign systemace_ce_b = 1'b1;
   assign systemace_we_b = 1'b1;
```

```
   assign systemace_oe_b = 1'b1;
   // systemace_irq and systemace_mpbrdy are inputs

   // Logic Analyzer
   //lab5 assign analyzer1_data = 16'h0;
   //lab5 assign analyzer1_clock = 1'b1;
   assign analyzer2_data = 16'h0;
   assign analyzer2_clock = 1'b1;
   //lab5 assign analyzer3_data = 16'h0;
   //lab5 assign analyzer3_clock = 1'b1;
   assign analyzer4_data = 16'h0;
   assign analyzer4_clock = 1'b1;

//   wire [7:0] from_ac97_data, to_ac97_data;
//   wire ready;

   ////////////////////////////////////////////////////////////////////////////
   //
   // Reset Generation
   //
   // A shift register primitive is used to generate an active-high reset
   // signal that remains high for 16 clock cycles after configuration finishes
   // and the FPGA's internal clocks begin toggling.
   //
   ////////////////////////////////////////////////////////////////////////////
   wire reset;
   SRL16 #(.INIT(16'hFFFF)) reset_sr(.D(1'b0), .CLK(clock_27mhz), .Q(reset),
                                  .A0(1'b1), .A1(1'b1), .A2(1'b1),
.A3(1'b1));

   wire [7:0] from_ac97_data, to_ac97_data;
   wire ready;

   // allow user to adjust volume
   wire vup,vdown;
   reg old_vup,old_vdown;
   debounce
bup(.reset(reset),.clock(clock_27mhz),.noisy(~button_up),.clean(vup));
   debounce
bdown(.reset(reset),.clock(clock_27mhz),.noisy(~button_down),.clean(vdown));
   reg [4:0] volume;
   always @ (posedge clock_27mhz) begin
     if (reset) volume <= 5'd8;
     else begin
if (vup & ~old_vup & volume != 5'd31) volume <= volume+1;
if (vdown & ~old_vdown & volume != 5'd0) volume <= volume-1;
     end
     old_vup <= vup;
     old_vdown <= vdown;
   end

   // AC97 driver
   lab5audio a(clock_27mhz, reset, volume, from_ac97_data, to_ac97_data, ready,
       audio_reset_b, ac97_sdata_out, ac97_sdata_in,
       ac97_synch, ac97_bit_clock);
   // push ENTER button to record, release to playback
   wire playback;
   debounce
benter(.reset(reset),.clock(clock_27mhz),.noisy(button_enter),.clean(playback));

   // switch 0 up for filtering, down for no filtering
   wire filter;
   debounce
sw0(.reset(reset),.clock(clock_27mhz),.noisy(switch[0]),.clean(filter));

   // light up LEDs when recording, show volume during playback.
   // led is active low
   //assign led = playback ? ~{filter,2'b00, volume} : ~{filter,7'hFF};

wire [15:0] addr;
wire [3:0] st;
wire [17:0] fo;
wire [7:0] fi;
   // record module
   //recorder r(.clock(clock_27mhz), .reset(reset), .ready(ready),
     //         .playback(playback), .filter(filter),
       //       .from_ac97_data(from_ac97_data),.to_ac97_data(to_ac97_data),
//   .a(addr), .state1(st), .filter_in(fi),
.filter_out(fo));
wire chunk_enable, chunks_done, ib_ce, ib_cd;
wire [23:0] chunk_out, ib_co;
wire record;
assign record = ~playback;
wire [2:0] state1;
wire [15:0] base_addr, record_addr;
wire [5:0] ib_count;

parameter WIDTH=8, VWIDTH=15, NPEAKS=3, INDEX=4, VINDEX=4, I=1;

 wire comp_reset, done, fft_reset, start, write_enable_in;
 wire comp_enable, he;
 wire read_enable_out;
 wire [WIDTH+1:0] addr_in_last;
 wire [WIDTH:0] addr_out;
 wire [VWIDTH:0] data_real_in, data_imag_in, next_data_in;
 wire [VWIDTH:0] data_real_out, data_imag_out;

feature_detector #(.WIDTH(WIDTH), .VWIDTH(VWIDTH), .NPEAKS(NPEAKS),
.INDEX(INDEX), .VINDEX(VINDEX), .I(I))f(.clk(clock_27mhz), .reset(reset),
.ready(ready),
.record(record), .from_ac97_data(from_ac97_data),
.chunk_enable(chunk_enable),
.chunk_out(chunk_out), .chunks_done(chunks_done),
.state(state1),
.base_addr(base_addr), .record_addr(record_addr),
.comp_enable(comp_enable), .he(he), .comp_reset(comp_reset),
 .read_enable_out(read_enable_out),
 .addr_in_last(addr_in_last), .addr_out(addr_out),
 .data_real_in(data_real_in), .data_imag_in(data_imag_in),
.next_data_in(next_data_in),
 .data_real_out(data_real_out), .data_imag_out(data_imag_out),
 .fft_reset(fft_reset), .start(start),
.write_enable_in(write_enable_in),
 .done(done)
);

inference_buffer ib (.clk(clock_27mhz), .record(record), .reset(reset),
.chunks_enable(chunk_enable),

.chunks_done(chunks_done), .chunk_in(chunk_out), .counter(ib_count),
.chunk_out(ib_co),
.chunk_enable(ib_ce), .chunk_done(ib_cd));



wire [4:0] command;
wire output_enable;

wire [3*(WIDTH+1):0] p1, p2, p3, p4, p5;

// Instantiate the Unit Under Test (UUT)
inference #(.WIDTH(WIDTH), .VWIDTH(VWIDTH), .NPEAKS(NPEAKS),
.INDEX(INDEX), .VINDEX(VINDEX), .I(I))
uut_inf (
.clk(clock_27mhz),
.reset(reset),
.chunk_enable(ib_ce),
.chunk(ib_co),
.command(command),
.output_enable(output_enable),
.p1(p1),
.p2(p2),
.p3(p3),
.p4(p4),
.p5(p5)
);


   wire clock_65mhz_unbuf,clock_65mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
   BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

   // power-on reset generation
   // ENTER button is user reset
//   wire reset,user_reset;
//   debounce
db1(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button_enter),
.clean(user_reset));
//   assign reset = power_on_reset;

   // UP, DOWN, LEFT, and RIGHT for car movement
   wire up,down,left,right,buttonzero;
assign {up, down, left, right, buttonzero} = command;
```

```
//   debounce
db2(.reset(reset),.clock(clock_65mhz),.noisy(~button_up),.clean(up));
//   debounce
db3(.reset(reset),.clock(clock_65mhz),.noisy(~button_down),.clean(down));
//   debounce
db4(.reset(reset),.clock(clock_65mhz),.noisy(~button_left),.clean(left));
//   debounce
db5(.reset(reset),.clock(clock_65mhz),.noisy(~button_right),.clean(right));
//   debounce
db6(.reset(reset),.clock(clock_65mhz),.noisy(~button0),.clean(buttonzero));

   // generate basic XVGA video signals
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;
   xvga xvga1(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
              .hsync(hsync),.vsync(vsync),.blank(blank));

   // feed XVGA signals to visualization
   wire [23:0] pixel;
   wire phsync,pvsync,pblank;
   visualization vis(.vclock(clock_65mhz),.reset(reset),
                .up(up),.down(down),.left(left),.right(right),
     .stop(buttonzero),
.hcount(hcount),.vcount(vcount),
                .hsync(hsync),.vsync(vsync),.blank(blank),
.phsync(phsync),.pvsync(pvsync),.pblank(pblank),.pixel(pixel));

   // switch[1:0] selects which video generator to use:
   //  00: visualization
   //  01: 1 pixel outline of active video area (adjust screen controls)
   //  10: color bars
   reg [23:0] rgb;
   wire border = (hcount==0 | hcount==1023 | vcount==0 | vcount==767);

   reg b,hs,vs;
   always @(posedge clock_65mhz) begin
      if (switch[1:0] == 2'b01) begin
 // 1 pixel outline of visible area (white)
 hs <= hsync;
 vs <= vsync;
 b <= blank;
 rgb <= {24{border}};
      end else if (switch[1:0] == 2'b10) begin
 // color bars
 hs <= hsync;
 vs <= vsync;
 b <= blank;
 rgb <= {{8{hcount[8]}}, {8{hcount[7]}}, {8{hcount[6]}}} ;
      end else begin
         // default: visualization
 hs <= phsync;
 vs <= pvsync;
 b <= pblank;
 rgb <= pixel;
      end
   end

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz.
   assign vga_out_red = rgb[23:16];
   assign vga_out_green = rgb[15:8];
   assign vga_out_blue = rgb[7:0];
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_blank_b = ~b;
   assign vga_out_pixel_clock = ~clock_65mhz;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;

//   assign led = ~{3'b000,up,down,reset,switch[1:0]};


assign led = {command, state1, 1'b0};
   // output useful things to the logic analyzer connectors
   assign analyzer1_clock = ac97_bit_clock;
   //assign analyzer1_data[0] = audio_reset_b;
   //assign analyzer1_data[1] = ac97_sdata_out;
   //assign analyzer1_data[2] = ac97_sdata_in;
   //assign analyzer1_data[3] = ac97_synch;
// assign analyzer1_data[1:0] = {playback, filter};
assign analyzer1_data[14:0] = base_addr;
assign analyzer1_data[15] = 1;
//   assign analyzer1_data[15:5] = fo[17:7];

   assign analyzer3_clock = clock_27mhz;
   assign analyzer3_data[4:0] = {chunk_enable, chunks_done, state1};
assign analyzer3_data[7:5] = 1;
assign analyzer3_data[15:8] = from_ac97_data;
endmodule

///////////////////////////////////////////////////////////////////////////////
//
// Record/playback
//
///////////////////////////////////////////////////////////////////////////////


module fir31s(
  input wire clock,reset,ready,
  input wire signed [7:0] x,
  output reg signed [17:0] accumulator,
  output reg signed [17:0] next_accumulator,
  output reg signed [17:0] y,
  output wire signed [9:0] coeff,
  output wire signed [7:0] sample2,
  output reg signed [7:0] s1,
  output reg signed [9:0] s2,
  output reg [4:0] index,
  output reg [4:0] offset);

   reg signed [7:0] sample [31:0];  // 32 element array each 8 bits wide
   reg [4:0] next_offset = 5'b0;
reg [5:0] next_index = 5'b0;
// wire signed [9:0] coeff;

coeffs31 c31(.index(index), .coeff(coeff));


always @(offset, index, accumulator, coeff, sample[0], sample[1],
sample[2], sample[3], sample[4], sample[5], sample[6], sample[7], sample[8],
sample[9], sample[10], sample[11], sample[12], sample[13], sample[14],
sample[15], sample[16], sample[17], sample[18], sample[19], sample[20],
sample[21], sample[22], sample[23], sample[24], sample[25], sample[26],
sample[27], sample[28], sample[29], sample[30], sample[31]) begin
   if (ready | reset) next_index = 0;
next_offset = offset + 1'b1;

if (next_index <= 30) begin
next_index = index + 1'b1;
s1 = sample[offset-index];
s2 = coeff;
next_accumulator = accumulator + (s1 * s2);
end
else begin
next_index = index;
next_accumulator = accumulator;
end
   end

  // for now just pass data through
  always @(posedge clock) begin
 if (reset) begin
y <= 18'sd0;
offset <= 5'b0;
index <= 5'b0;
accumulator <= 18'sd0;
sample[0] <= 8'sd0;
sample[1] <= 8'sd0;
sample[2] <= 8'sd0;
sample[3] <= 8'sd0;
sample[4] <= 8'sd0;
sample[5] <= 8'sd0;
sample[6] <= 8'sd0;
sample[7] <= 8'sd0;
sample[8] <= 8'sd0;
sample[9] <= 8'sd0;
sample[10] <= 8'sd0;
sample[11] <= 8'sd0;
sample[12] <= 8'sd0;
sample[13] <= 8'sd0;
sample[14] <= 8'sd0;
sample[15] <= 8'sd0;
sample[16] <= 8'sd0;
sample[17] <= 8'sd0;
sample[18] <= 8'sd0;
sample[19] <= 8'sd0;
sample[20] <= 8'sd0;
sample[21] <= 8'sd0;
```

```
sample[22] <= 8'sd0;
sample[23] <= 8'sd0;
sample[24] <= 8'sd0;
sample[25] <= 8'sd0;
sample[26] <= 8'sd0;
sample[27] <= 8'sd0;
sample[28] <= 8'sd0;
sample[29] <= 8'sd0;
sample[30] <= 8'sd0;
sample[31] <= 8'sd0;
 end
    if (ready) begin
\$display("ready");
y <= accumulator;
offset <= next_offset;
index <= 5'b0;
accumulator <= 18'sd0;
sample[next_offset] <= x;
 end
 else begin
index <= next_index;
accumulator <= next_accumulator;
 end
  end
  assign sample2 = sample[offset-index];
endmodule




////////////////////////////////////////////////////////////////////////////////
//
// Verilog equivalent to a BRAM, tools will infer the right thing!
// number of locations = 1<<LOGSIZE, width in bits = WIDTH.
// default is a 16K x 1 memory.
//
////////////////////////////////////////////////////////////////////////////////

module mybram #(parameter LOGSIZE=14, WIDTH=1)
              (input wire [LOGSIZE-1:0] addr,
               input wire clk,
               input wire [WIDTH-1:0] din,
               output reg [WIDTH-1:0] dout,
               input wire we);
   // let the tools infer the right number of BRAMs
   (* ram_style = "block" *)
   reg [WIDTH-1:0] mem[(1<<LOGSIZE)-1:0];
   always @(posedge clk) begin
     if (we) mem[addr] <= din;
     dout <= mem[addr];
   end
endmodule

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

module coeffs31(
  input wire [4:0] index,
  output reg signed [9:0] coeff
);
  // tools will turn this into a 31x10 ROM
  always @(index)
    case (index)
      5'd0:  coeff = -10'sd1;
      5'd1:  coeff = -10'sd1;
      5'd2:  coeff = -10'sd3;
      5'd3:  coeff = -10'sd5;
      5'd4:  coeff = -10'sd6;
      5'd5:  coeff = -10'sd7;
      5'd6:  coeff = -10'sd5;
      5'd7:  coeff = 10'sd0;
      5'd8:  coeff = 10'sd10;
      5'd9:  coeff = 10'sd26;
      5'd10: coeff = 10'sd46;
      5'd11: coeff = 10'sd69;
      5'd12: coeff = 10'sd91;
      5'd13: coeff = 10'sd110;
      5'd14: coeff = 10'sd123;
      5'd15: coeff = 10'sd128;
      5'd16: coeff = 10'sd123;
      5'd17: coeff = 10'sd110;
      5'd18: coeff = 10'sd91;
      5'd19: coeff = 10'sd69;
      5'd20: coeff = 10'sd46;
      5'd21: coeff = 10'sd26;
      5'd22: coeff = 10'sd10;
      5'd23: coeff = 10'sd0;
      5'd24: coeff = -10'sd5;
      5'd25: coeff = -10'sd7;
      5'd26: coeff = -10'sd6;
      5'd27: coeff = -10'sd5;
      5'd28: coeff = -10'sd3;
      5'd29: coeff = -10'sd1;
      5'd30: coeff = -10'sd1;
      default: coeff = 10'hXXX;
    endcase
endmodule

*******************************

#!/usr/bin/env python
from PIL import Image
import sys

def writeImage(img, numbits):
    width = img.size[0]
    length = img.size[1]
    data = img.tostring()
    R = data[::4]
    G = data[1::4]
    B = data[2::4]
    A = data[3::4]
    message = ""
    for y in range(length):
        for x in range(width):
            pixel = (R[x+y*width], G[x+y*width], B[x+y*width], A[x+y*width])
            rpix = format(ord(pixel[0]), '0%db'%(numbits))[:numbits]
            gpix = format(ord(pixel[1]), '0%db'%(numbits))[:numbits]
            bpix = format(ord(pixel[2]), '0%db'%(numbits))[:numbits]
            message = message + "10'd%d: temp_data <= %d'b%s%s%s;\n" %
(x+y*width, 3*numbits, rpix, gpix, bpix)
    print message[:-1]

def writeRom(name):
    img = Image.open(name+".png")
    my_file = open(name+"_rom_core.v",'w')
    sys.stdout = my_file
    numbits = 8
    print "module %s_rom_core(input [9:0] addr, input clk, output [%d:0] data);"
% (name, 3*numbits - 1)
    print "reg [%d:0] temp_data;" % (3*numbits - 1)
    print "assign data = temp_data;"
    print "always @(posedge clk) begin"
    print "case(addr)"
    writeImage(img, numbits)
    print "endcase"
    print "end"
    print "endmodule"
    sys.stdout = sys.__stdout__
    my_file.close()

writeRom("up")
writeRom("down")
writeRom("left")
writeRom("right")

*******************************

#!/usr/bin/env python
import sys

def fun(mus, sigmas, sub):
    precision = 17
    base = 10000

    muarray = mus.split(',')
    sigmaarray = sigmas.split(',')
    mult = 2**precision
    mutimes = [mult**i for i in range(0,len(muarray))]
    sigmatimes = [mult**i for i in range(0,len(sigmaarray))]
    finalmu = [int(a*float(b)) for a,b in zip(mutimes,muarray)]
    intermediate1 = [int(base*float(b)) for b in sigmaarray]
    intermediate2 = [int(bin(a%(1<<precision)),2) for a in intermediate1]
    finalsigma = [int(a*b) for a,b in zip(sigmatimes,intermediate2)]
    mu = sum(finalmu)
    sigma = sum(finalsigma)
    muvalue = "%d'd%d" % (precision*len(muarray), mu)
    sigmavalue = "%d'd%d" % (precision*len(sigmaarray), sigma)
    return [muvalue, sigmavalue, sub]

param_file = open("params.txt",'r')
data = []
for line in param_file:
    data.append(line)
param_file.close()
width = 3
prec = 16
my_file = open("test.txt",'w')
sys.stdout = my_file
for iter1 in range(0,5):
    values = []
    for i in range(0,3):
        values.append(fun(data[iter1*10+3*i+1], data[iter1*10+3*i+2],
data[iter1*10+3*i+3]))
    print "gmm_pdf #(.MUS1(%s), .MUS2(%s), .MUS3(%s), .SIGMAS1(%s),
.SIGMAS2(%s), .SIGMAS3(%s), .SUB1(%d), .SUB2(%d), .SUB3(%d), .WIDTH(%d),
.PRECISION(%d))" % (values[0][0], values[1][0], values[2][0], values[0][1],
values[1][1], values[2][1], int(float(values[0][2])), int(float(values[1][2])),
int(float(values[2][2])), width, prec)
    print "%spdf(.clk(clk), .reset(reset),
.input_enable(chunk_enable),.data_point(chunk),.output_enable(o%d),
.pdf(p%d));\n" % ((data[iter1*10])[:-1], iter1+1, iter1+1)
sys.stdout = sys.__stdout__
my_file.close()

******************************

#!/usr/bin/env python
import sys

def rect(x, y, width, height, maze, coll):
    offset = 32
    maze.append("(addr[9:0] >= 10'd%d && addr[9:0] <= 10'd%d && addr[19:10] >=
10'd%d && addr[19:10] <= 10'd%d)\n" % (x, x+width, y, y+height))
    coll.append("(addr[9:0] >= 10'd%d && addr[9:0] <= 10'd%d && addr[19:10] >=
10'd%d && addr[19:10] <= 10'd%d)\n" % (max(x-offset,0), x+width,
max(y-offset,0), y+height))

maze = []
coll = []
thickness = 32
pathsize = 128
rect(0, 0, 1023, thickness, maze, coll)
rect(0, 0, thickness, 768, maze, coll)
rect(0, 768-thickness, 1023, thickness, maze, coll)
rect(1023-thickness, 0, thickness, 768, maze, coll)
rect(pathsize+thickness, 0, 512-pathsize-thickness, 768-pathsize-thickness,
maze, coll)
```

```
rect(512+pathsize, thickness+pathsize, 512-2*pathsize-thickness,
768-thickness-pathsize, maze, coll)

maze_file = open("maze_rom_core.v",'w')
sys.stdout = maze_file
print "module maze_rom_core(input [19:0] addr, input clk, output imgbit);"
print "assign imgbit = %s;" % ("||".join(maze))
print "endmodule"
sys.stdout = sys.__stdout__
maze_file.close()

coll_file = open("collision_rom_core.v",'w')
sys.stdout = coll_file
print "module collision_rom_core(input [19:0] addr, input clk, output imgbit);"
print "assign imgbit = %s;" % ("||".join(coll))
print "endmodule"
sys.stdout = sys.__stdout__
coll_file.close()

*******************************
```