Voice-Controlled Chess Game on FPGA Using Dynamic Time Warping

Varun Chirravuri, Michael Kuo

Project Abstract

Most modern digital chess games employ a mouse and keyboard based user interface. We change this paradigm by designing an FPGA based, voice-controlled, chess game. We will train the game to recognize specified voice commands from the players. We will use dynamic time warping to compare real-time speech samples to the trained command templates to determine what move a player wishes to make. The game will be displayed on a VGA display with all the functionality of a standard chess game.

Table of Contents

Table of Figures

Overview

The complete system takes in voice commands and keyboard inputs to control a chess game displayed on an XVGA display. The system is divided into two components: audio recognition hardware and chess hardware.

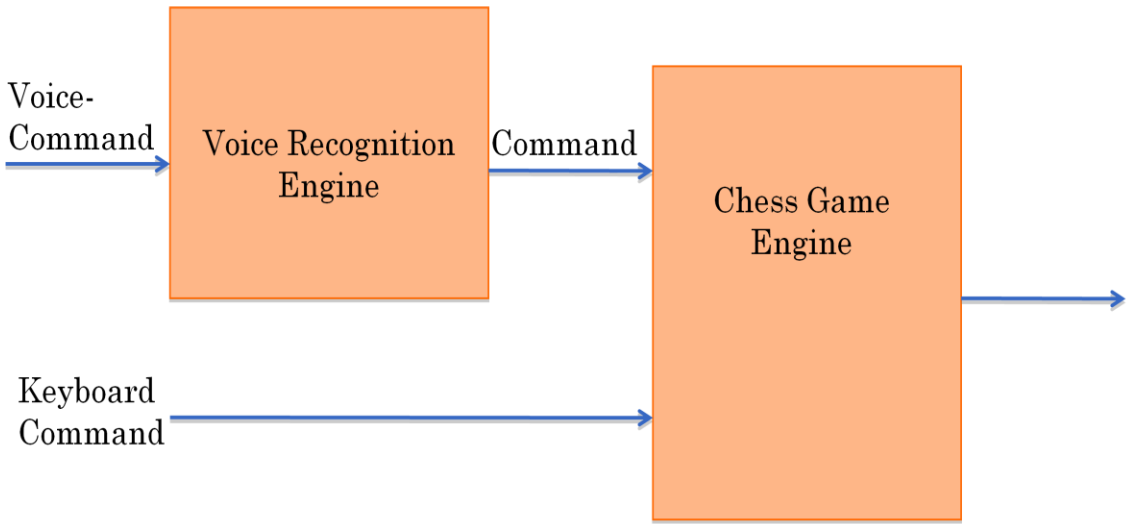

Figure 1. High Level Block Diagram Of Entire System

Audio Recognition Hardware (Varun)

The audio recognition hardware is in charge of handling all of the audio processing and calculations associated with the voice-recognition portion of the chess game. Employing a series of finite state machines and the dynamic time warping algorithm to compare audio samples, the system correctly determines which move the user wishes to make in the chess game, and sends that information to the chess engine to execute.

Incoming audio streams are down-sampled and filtered to reduce the aliasing effects caused by the down-sampling. The filtered audio stream is then sent to a valid-checking module that determines whether a valid word has indeed been spoken or if the incoming audio is simply background noise.
When the checking module detects a valid word, the system begins actively recording the audio stream. A one-half second audio clip is stored into memory, and then streamed down to the dynamic time warping engines.

There are a total of eight dynamic time warping (DTW) engines. Once trained, each engine contains audio-templates corresponding to one of the eight letters (A-H), and one of the eight numbers (1-8) that specify all possible positions on the board. Depending on where in the command sequence the user is when the valid sample is detected, the incoming audio streamed is compared to either all eight numbers or all eight numbers, once comparison per engine.

The DTW engines use the dynamic time warping algorithm to compare the samples. The algorithm uses dynamic programming to correct for temporal differences between stored samples and the valid sample, and returns a value corresponding to the error between the samples. The template belonging to the engine that returns the lowest error for a given audio input is determined to be the intended number or letter. Once four such commands are issued, the system concatenates the results as a complete command consisting of two numbers and two letters and sends the information to the chess engine.

Chess Hardware (Michael)

The chess system is an implementation of a two-player chess game on the 6.111 LabKit. It receives move instructions from either the voice recognition system or a keyboard, checks that instructed moves are permissible, and displays the chessboard and chess pieces on an XVGA display. The three main components of the chess system, as illustrated in the figure below, are the keyboard input, the chess engine, and the graphics engine.

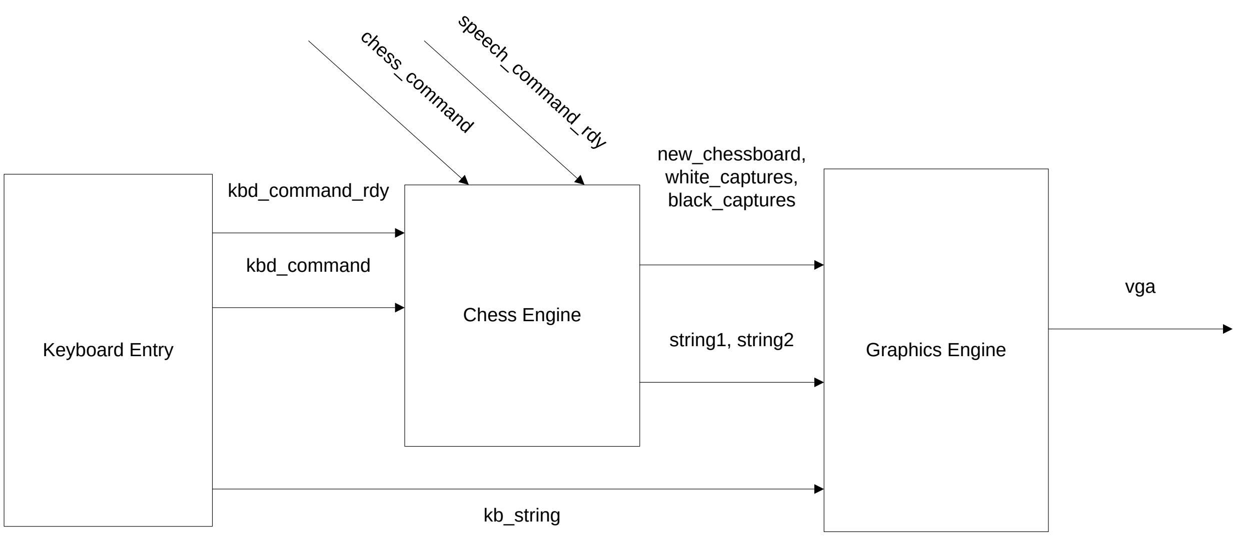

Figure 2. Top level block diagram of chess system.

Keyboard Input

The keyboard input component encodes keyboard input into a move command that is passed to the chess engine.

Chess Engine

The chess engine directs the flow of the chess game. It keeps track of the turn, determines if instructed moves from the voice recognition system or keyboard input are permissible, and manages the internal representation of the chessboard and the chess pieces on the board.

Graphics Engine

The graphics engine generates video images for the chess game to be displayed on the XVGA display. The video images include the chessboard and the chess pieces on the board, a grid of the pieces that each player has captured, as well as text from the chess engine and keyboard input.

Description

Input (Michael)

In order to modularize the voice recognition system and the chess system, a unified encoding scheme was defined for all move commands passed to the chess engine. The encoding of the move command is composed of the column and row (file and rank) of the square containing the piece being moved and the column and row of the square where the piece is being moved.

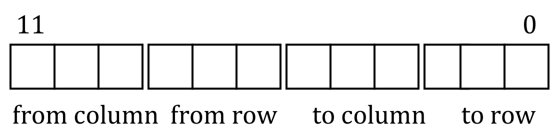

Figure 3. Move command encoding.

The chessboard is a square board of eight columns and eight rows of squares. The columns, which are lettered A through H are encoded with 0 through 7, and the rows, which are numbered 1 through 8 are also encoded with 0 through 7. The complete encoding of the move command is twelve bits – three bits for each of the columns and rows of the original square and destination square of the piece being moved.

Audio Recognition Hardware (Varun)

Audio is sent to the audio recognition hardware from the AC97 codec found in the 6.111 lab 4 documents. Audio is sampled from the headphones at 48 kHz and sent from the codec directly to the Audio Recognition Hardware. The hardware consists of five modules, listed in the order of complexity: the DTW System Controller, DTW Engines, Valid Detector, Finite Impulse Response Filter, and the Shift Connector. Because of the many modules involved in processing incoming audio, a complex signaling and handshaking system was used to ensure data was not lost, and was processed in the correct order.

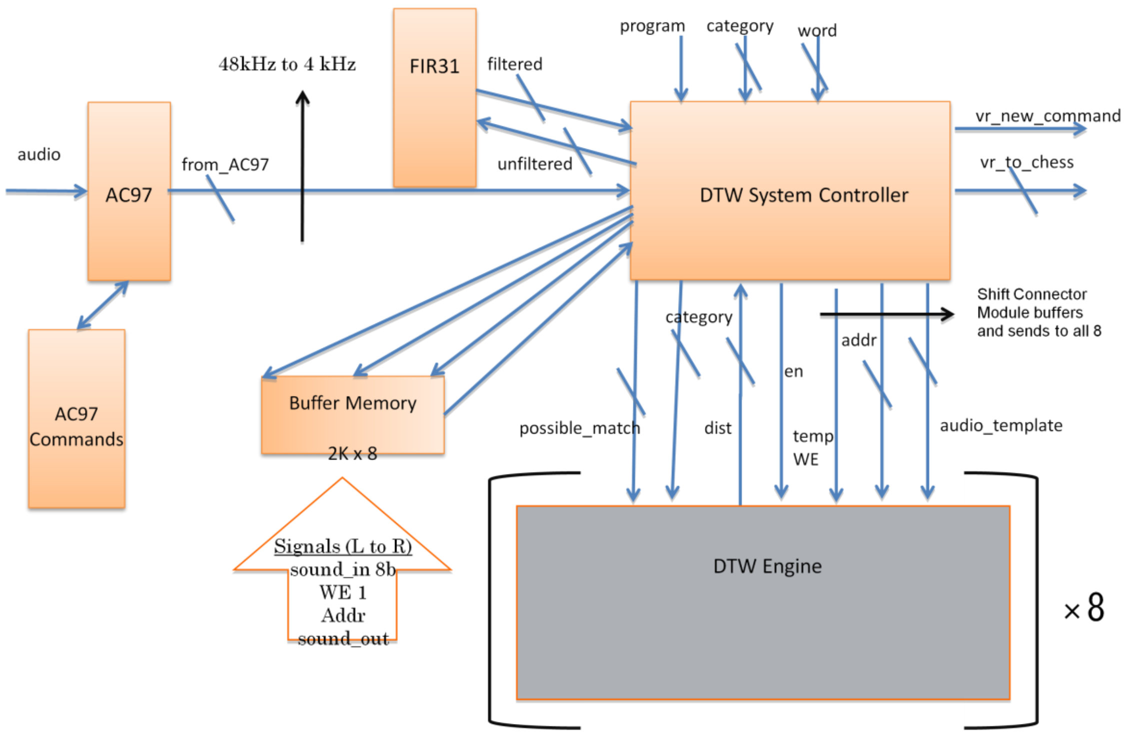

Figure 4. Block Diagram Of Audio Recognition Hardware.

DTW System Controller

The DTW System Controller controls the flow of data through the entire system, and therefore contains the most complexity. It is implemented as two nested finite state machines. The outer most FSM is in charge of capturing incoming data, sending valid samples to the DTW Engines, and then interpreting the error values returned by the DTW Engines to output the correct command to the chess module. The inner FSM is used to determine which section of the command the user is inputting (*from_letter, from_number, to_letter,* or *to_number*) and if a complete command has been issued and is ready to be outputted. The next state and next substate are determined combintationally, with state being updated to reflect the next state at the following clock edge.

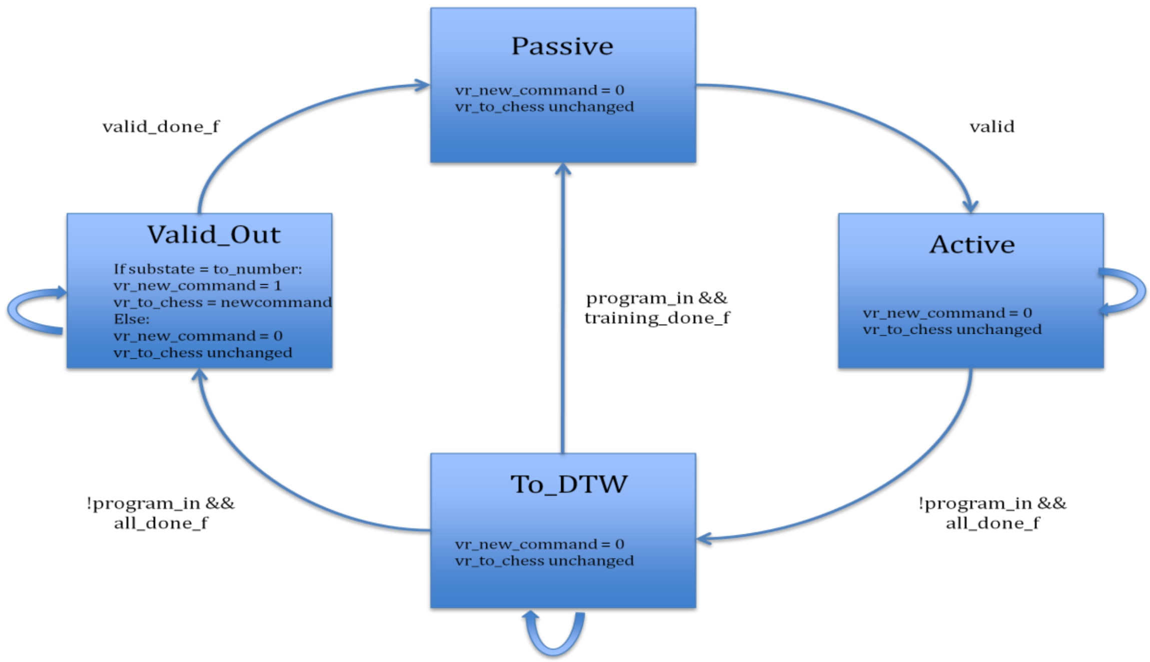

Figure 5. Outer FSM of the DTW System Controller.

Outer FSM

Passive State

The outer FSM has four states: *passive*, *active*, *to_DTW*, and *valid_out*. The default state of the machine is *passive*. In this state, audio is downsampled from the 48 kHz sampling rate of the AC97 codec to 4 kHz. Audio is only sampled when users press the record button, indicating that they wish to record audio. Incoming audio is immediately sent to the anti-aliasing filter to correct for the error induced by downsampling. The filter output is then stored in a 2048 x 8 bit BRAM (incoming audio is 8 bits wide). The BRAM stores approximately half a second of audio, and is used like a circular buffer when in the passive state. Whenever a sample is written to memory, an enable pulse is sent to the *Valid_Checker* module along with the current sample being written. The *Valid_Checker* outputs a *valid* signal whenever a valid word is spoken. Because the valid-checking module outputs a valid pulse only after there is a noticeable increase in the amplitudes of the previous 128 samples, a *start_val* pointer is updated to point 128 memory locations behind the current location being written to. As soon as a *valid* signal is outputted, the FSM switches to the *active* state.

Active State

The job of the *active* state is to continue downsampling and recording audio until a complete, half-second audio segment is written to memory. While downsampling, and filtering occur just as in the *passive* state, the *start_val* and *end_val* pointers are no longer updated. Enable signals are no longer sent to the *Valid_Checker* as we have already recording a valid word. Writing to memory continues and incrementing the address continues until a sample is written to the *end_val* address. At this point, the system asserts a flag (*end_record_f)* letting the system know a half-second word has been recorded. Once this flag goes high, *to_DTW* is assigned to next state, and becomes the state on the following clock cycle.

To_DTW State

Upon entering the *to_DTW* state, the end record flag is reset to zero and the memory address is set to the sample-starting pointer, *start_val*. Over the next 2048 clock cycles, all 2048 audio samples stored in the *Controller's* memory are outputted to one of two audio output channels depending on whether the system is training the DTW Engines or if it is sending then possible match audio to compare against their templates.

If the user is training the system and the *train_in* switch is set high, then a write enable signal (*temp_WE*) is outputted to the DTW Engines. When training, users use switches to choose which category they are training (letters or numbers) and which word (1-8 or A-H) they wish to train. This data is outputted via the 3 bit *word_out* and 1 bit *category_out* registers, which are combinationally assigned to the value selected by the user. The flag *training_done_f* is raised to indicate that training is complete, and nextstate is set to *passive*, so the system can accept a new training sample or a command.

If the user is not training the device, the match write enable is raised instead of the template write enable, signalling to the DTW engines that the incoming audio sample is a possible match and should be compared to the stored template. The system remains in the *to_DTW* state until all 8 of the DTW engines signal that they have finished their calculations and have returned a valid *distance* value. When this happens, the *all_done_f* flag is raised and the system moves the *valid out* state.

Valid_Out State

The *valid_out* state compares the *distance* values returned by the DTW Engines and determines which trained template most closely matched the inputted audio. To prevent any timing issues, and to reduce redundancy, the system sequentially calculates the minimum of the 8 returned value in 4 clock cycles. In the first three clock cycles, the system performs four, two, and one comparison respectively to determine which DTW Engine returned the lowest distance. The number of the DTW Engine with the lowest distance is stored. On the fourth clock cycle, a *valid_done_f* flag is raised, indicating all comparisons are done, and setting the state back to *passive*. The number of the register with the lowest distance is stored as the intended letter/number (1-8, A-H). If all four parts of the complete command have been determined, the complete command is sent to the chess engine along with a signal *vr_new_command* which signals that a new command has indeed been sent.

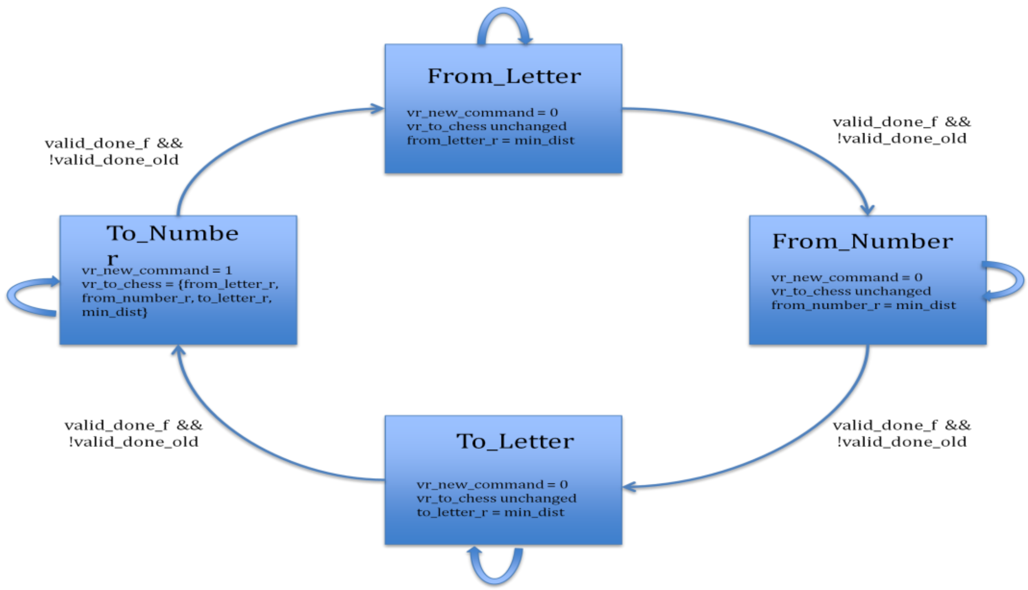

Figure 6. Inner FSM of the DTW System Controller.

Inner FSM

The inner FSM is in charge of determining which element of the complete command the current input corresponds to, and is much simpler than the outer FSM. All state changes occur at the positive edge of the *valid_out_f* flag—that is, once the previous section of the command, or the previous command has successfully been computed. The states correspond to each section of the complete command – the letter and number coordinates of the piece to be moved (*from_letter, from_number*), and the letter and number coordinates of the location the piece is to be moved to (*to_letter, to_number*). Because the last section of a command is the number of the square where the piece should be moved, commands are only outputted when the inner FSM is in the *to_letter* substate and the outer FSM is in the *valid_out* state. The sequential nature of the inner FSM ensures that the entire command is specified once the system reaches this overall state.

Dynamic Time Warping Engines

The Dynamic Time Warping Engines use a dynamic programming to remove temporal differences between audio samples and compare how closely they match. For instance, if a speaker says the word “book” twice, it is highly unlikely that the time-domain waveforms of the words will correspond—it is more likely that one of the two times the word will be spoken slower or faster than the other. Algorithms that just match time-domain samples cannot correct for this difference, and so an algorithm like dynamic time warping is used.

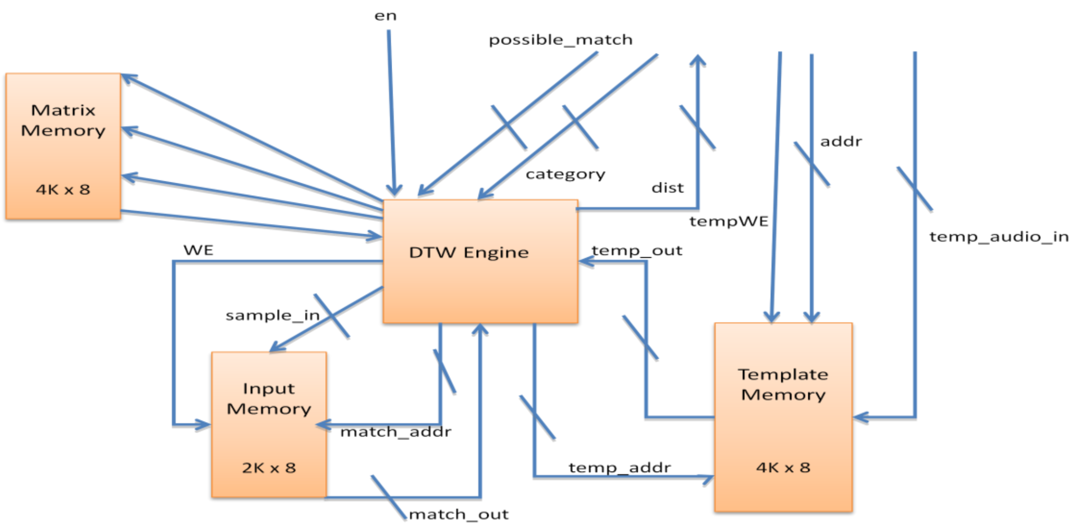

Figure 7. Diagram of the DTW Engine.

Dynamic Time Warping Algorithm

The DTW algorithm is fairly straightforward. Take two data series, X and Y, of length *m* and *n* respectively. Create an *m x n* matrix *D*, where the element D_{ij} represents the squared difference between the i^{th} sample of X and the j^{th} sample of Y. That is:

$$D_{ij} = (X_i - Y_j)^2 \text{ for } 0 \leq i \leq m \text{ and } 0 \leq j \leq n$$

From this matrix, we then solve the the dynamic programming problem of the shortest path from D_{00} to D_{mn} by creating a second *m+1 x n+1* matrix Υ. The values of Υ are computed as such:

$$\Upsilon_{ij} = D_{i-1,j-1} + \min(\Upsilon_{i-1,j}, \Upsilon_{i,j-1}, \Upsilon_{i-1,-1j}) \text{ for } 1 \leq i \leq m+1 \text{ and } 1 \leq j \leq n+1$$
$$\text{with } \Upsilon_{0,j} = \Upsilon_{i,0} = \infty, \ \Upsilon_{1,1} = D_{0,0} \text{ and } \Upsilon_{0,0} = 0$$

Thus, every value Υ_{ij} represents the shortest "error path" found from the first element of *D* to the current element. So it stands that, once the algorithm completes, $\Upsilon_{m+1,n+1}$ will be the shortest "error path" that exists between the two samples. So, for example, if the word "book" is said once at a normal pace, and once twice as slow, the algorithm will traverse Υ precisely on the diagonal with a slope of 2, accruing no error as it traverses, and will indicate that the words are an exact match.

Dynamic Time Warping Implementation

The main challenge with porting the DTW algorithm into hardware is that assuming that the samples are of length *m* and *n* with each sample having width *b*, the overall memory needed will be *m*n*b* bits. At 2048 8-bit samples per audio segment, running 8 DTW Engines on the FPGA would exceed the memory available. To work around this limitation, a simple solution was chosen. At each iteration of the algorithm, only two rows of Υ are ever needed to compute the current element. So instead of needing to store 2048*2048*8 bits of data, the algorithm would only need 2048*2*8 bits or memory, which is feasible. Also, the algorithm must keep 4 data registers to hold the 4 values needed to compute the next value of the matrix ($D_{i-1,j-1}$, $\Upsilon_{i-1,j}$, $\Upsilon_{i,j-1}$, $\Upsilon_{i-1,-1j}$).

To do this, the DTW Engine keeps two counters, one that is 11 bits and one that is 22 bits. At each iteration of the algorithm, both counters are incremented. The first counter rolls back to 0 once it has reached 2047, and so it is used to indicate where in the row the algorithm is. The other counter increments until it reaches the value 4,194,303, which indicates that the algorithm has computed all of the values of Υ.

The algorithm then moves sequentially, with each iteration taking 5 clock cycles. On the first clock, if the value of element *(i,j)* is being computed, the algorithm decrements the memory address for the DTW pseudo-matrix memory by 2047, so that it loads the value of the *(i,j-1)* on the third clock. On the second clock, it pulls the value for the i^{th} element of the template and the j^{th} element of the inputted sample and stores their difference to a register. On the third clock cycle, the previous value of *(i,j)* is loaded into the register for *(i,j-1)*, the previous value of *(i-1,j)* is loaded into the register for *(i-1,j-1)* , and the value coming out of the DTW memory is loaded into the *(i-1,j)* register. Essentially, the frame of reference for the computation shifts right. Also on the third clock cycle, the value D_{ij} is computed by squaring the difference found in the second clock cycle, and storing that back into the same register. The DTW memory address is also set to point to the next location in memory to store the *(i,j)* th value in two clock cycles.

A simple speedup that was used reduced the number of comparisons needed in the next clock cycle was to store the minimum of the *(i,j)* and *(i-1,j)* elements and storing that instead of shifting the values in those registers into the other registers and finding the minimum of three values on the next clock.

On the third clock cycle, the algorithm performs all of the comparisons, and computes the value for the next *(i,j)* value. If the 22 bit counter is at zero, it simply loads the D_{ij} value into the *(i,j)* register. If the 22 bit counter is less than 2047, the algorithm is still computing the first row of the matrix and so *(i,j)* is the sum of D_{ij} and *(i,j-1).* When the 11 bit counter is 0 but the 22 bit counter is not 0, the algorithm is computing the first column, so *(i,j)* is the sum of D_{ij} and *(i-1,j).* Otherwise *(i,j)* is computed as the sum of D_{ij} and the minimum of *(i-1,j)* and the "speedup" register mentioned in the previous paragraph. Once the 22 bit counter reaches its maximum value, the algorithm has finished computing, and the current value of D_{ij} + *(i,j)* represents the minimum "distance" between the samples. At each iteration both counters are incremented. At every pass, the address to the possible match BRAM is incremented, and only when the 11 bit counter is zero and the 22 bit is not zero is the address to the template BRAM incremented.

DTW Engine FSM

To allow for the DTW Engine to be trained, to take in template samples, and to compute the distance between samples and a template, an FSM is implemented. It has four states: *hold, training, transfer,* and *calculate.* When in the *hold* state, the FSM simply waits for either a training enable or a transfer enable from the *DTW System Controller*. If in training, the engine uses the category bit from the *System Controller* to determine whether to store the sample in the first 2048 memory locations, or in the last 2048 memory locations in its template BRAM—0 is for letter and is stored in the first 2048, 1 is for number and is stored in the last. Similarly, when in the *transfer* and *calculate* states, it uses the same category bit to determine which template to compare the input against. Once a template is fully transferred to the BRAM, the system depends on the train from the *Controller* to fall to move into the *hold* state. When in the transfer state, the system depends on the enable signal from the *Controller* to go low to move into the *calculate* state, where the above calculations are performed. Once completed, the system outputs a *DTW_done* signal, and outputs the calculated *distance* value. The DTW holds its outputs for 3 clocks cycles, and completely wipes the DTW pseudo-matrix memory after it has finished computing the *distance* value.

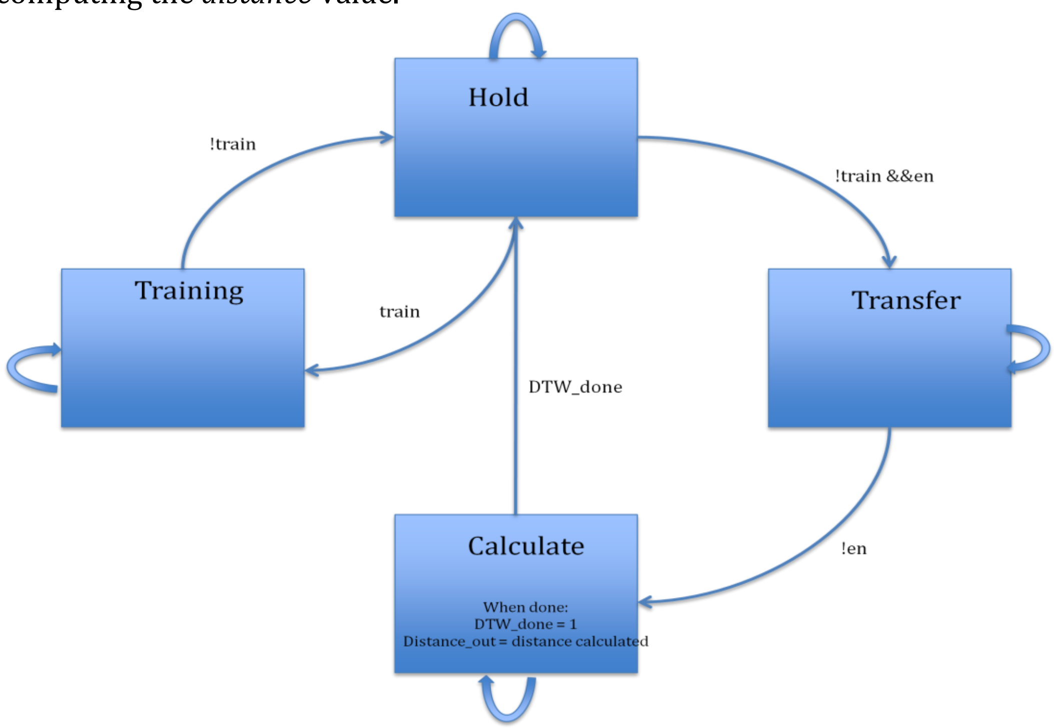

Figure 8. Dynamic Time Warping Engine FSM.

Valid Checker

The valid checker algorithm samples the audio stream coming into the *System Controller* to determine whether a word is being spoken. It uses a 256 x 8 bit register partitioned into two 128 halves. The register array is used as a FIFO buffer. At each clock, when an enable is issued from the *Controller*, the *valid checker* takes the audio input sent to it, calculates its absolute value, and then adds it to the tail of the array. Three values are always maintained – the index of the current sample, the 256th sample, and the 128th sample. Initially all registers in the array are set to zero. Two sum counters, first and last, are set to zero as well. When a new sample arrives, it is inserted into the array, and its absolute value is added to the last sum counter. The value of the old 128th sample is subtracted from that sum counter, and added to the other. The value of the old 256th sample is subtracted off of the first sum counter. The indices of the 128th and 256th values are moved back to indicate that the frame of reference as shifted. As this repeats and the buffer fills up, the values of the sum counters reflect the sum of the first 128th and last 128th samples. When the value of sum last exceeds the value of sum first by some margin that was empirically determined, the valid checker outputs a *valid* signal, and then clears the register array as well as the sum first and sum last counters in preparation for the next time the *System Controller* enters the *passive* state.

Finite Impulse Response Filter

The FIR filter used here was the same 31 tap FIR filter designed in 6.111 lab 4, with the filter coefficients modified to represent the change in downsampling rate. It is implemented with a circular register array to hold audio samples, and an accumulator that performs the necessary convolution multiplication between *ready* signals from the AC97 codec to output filtered data to the *DTW System Controller*.

Shift Connector

The *shift connector* module sits between the *DTW System Controller* and all of the DTW Engines. When the *System Controller* sends train signals to the engines, the shift connector uses the word outputted by the *System Controller* to determine which of the 8 engines should receive the training enable pulse and the template audio. The *shift connector* has the important function of allowing the *System Controller* and the *DTW Engine* to communicate as state machines, without having to assign outputs at different times to compensate for the one clock lag of changing state. There is a one-clock lag from when the DTW Engine receives an enable pulse and changes state to training or transfer, to when it actually accepts audio data in the new state. So without the *shift connector*, the DTW Engine would drop the first audio sample from the System Controller. To avoid this, the *shift connector* extends the enable/train from the *System Controller* by one clock cycle, and holds each audio sample being streamed down for one clock cycle, padding the first, dropped packet with a 0 value.

Chess Hardware (Michael)

Keyboard Input

The keyboard input component consists of two modules: keyboard entry and keyboard encoder. A block diagram containing the two modules is illustrated in the figure below.

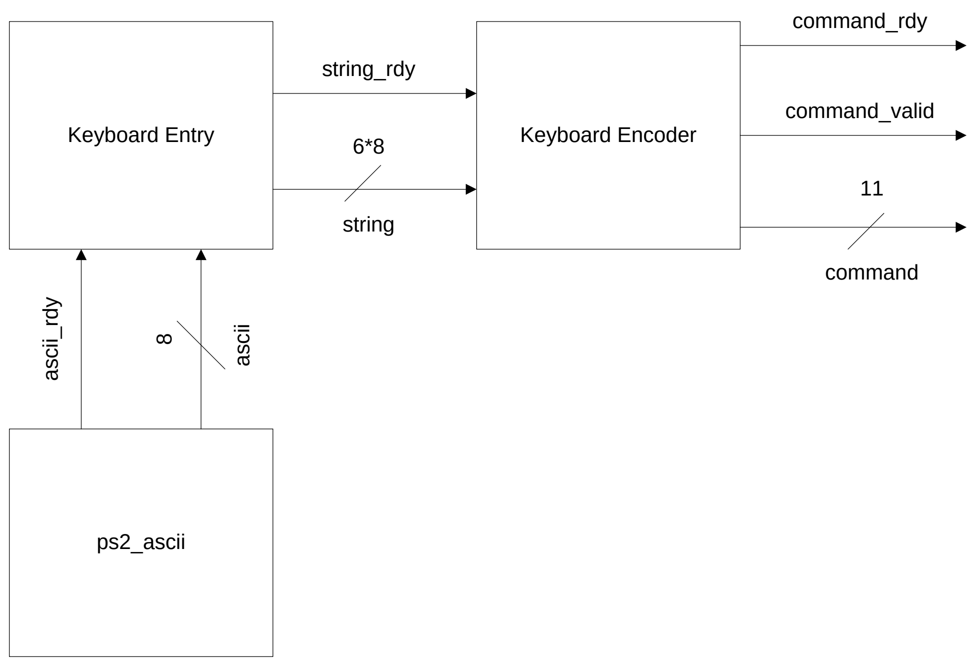

Figure 9. Block diagram of keyboard input component.

Keyboard Entry

The keyboard entry module packages inputs from the keyboard into strings of ASCII code to be encoded into move commands. Professors Ike Chuang's and Chris Terman's *ps2_kbd module*[1] is used to convert scan codes generated by the keyboard into ASCII codes. An array stores the ASCII codes as keys are pressed – up to five codes, as five ASCII characters is enough to issue a keyboard move instruction. Keyboard move instructions take the form

<from column><from row>_<to column><to row>

(e.g. "A2 A3"). Pressing the Backspace key deletes the last stored ASCII code stored in the array – functionally similar to pressing the Backspace key in a word processor. Pressing the Enter key locks the array of ASCII codes for one clock cycle and signals on the same clock cycle that a string of ASCII code is ready to be

[1] http://web.mit.edu/6.111/www/f2005/code/ps2_kbd.v

encoded. Afterwards, the array is cleared to allow new ASCII codes for new move commands to be stored.

Keyboard Encoder

The keyboard encoder module encodes the string of ASCII code received from the keyboard entry module into a 12-bit move instruction using the encoding scheme described previously. Each ASCII code is encoded if it corresponds to a valid letter or number, depending on its position in the string. The rightmost and forth-from-right ASCII codes in the string should each correspond to one of the letters from A through H, the second-from-right and leftmost ASCII codes in the string should each correspond to one of the numbers from 1 through 8, and the third-from-right ASCII code in the string should correspond to a whitespace. The overall encoding is valid if the above criteria are met. A one clock cycle ready signal is raised once encoding is complete or the string of ASCII code has been determined to be invalid – a valid signal is raised at the same time if the overall encoding is actually valid.

Chess Engine

The chess engine component consists of two modules: chess engine and move checker. A block diagram containing the two modules is illustrated in the figure below.

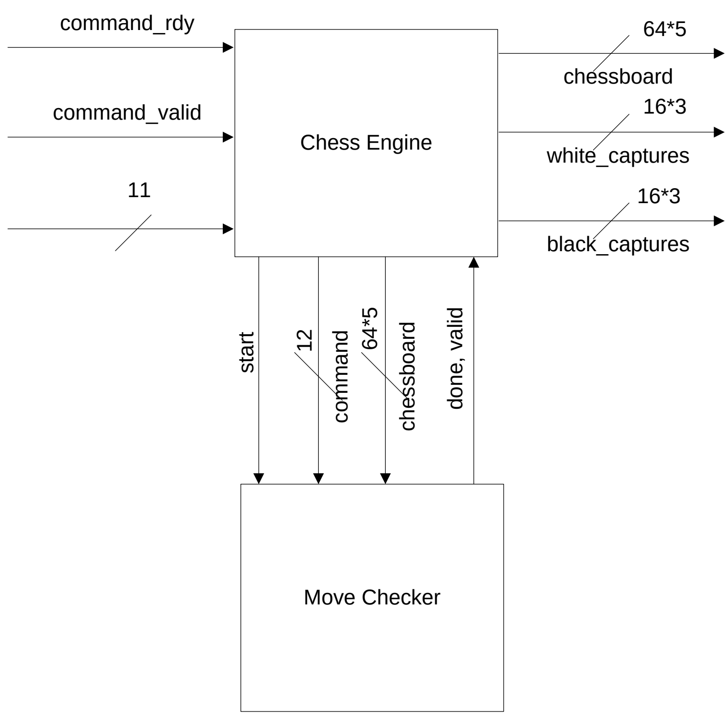

Figure 10. Block diagram of chess engine component.

Chess Engine

The chess engine module is an FSM that directs the play of the chess game and manages the internal representation of the chessboard. Its states and transitions are illustrated in the figure below.

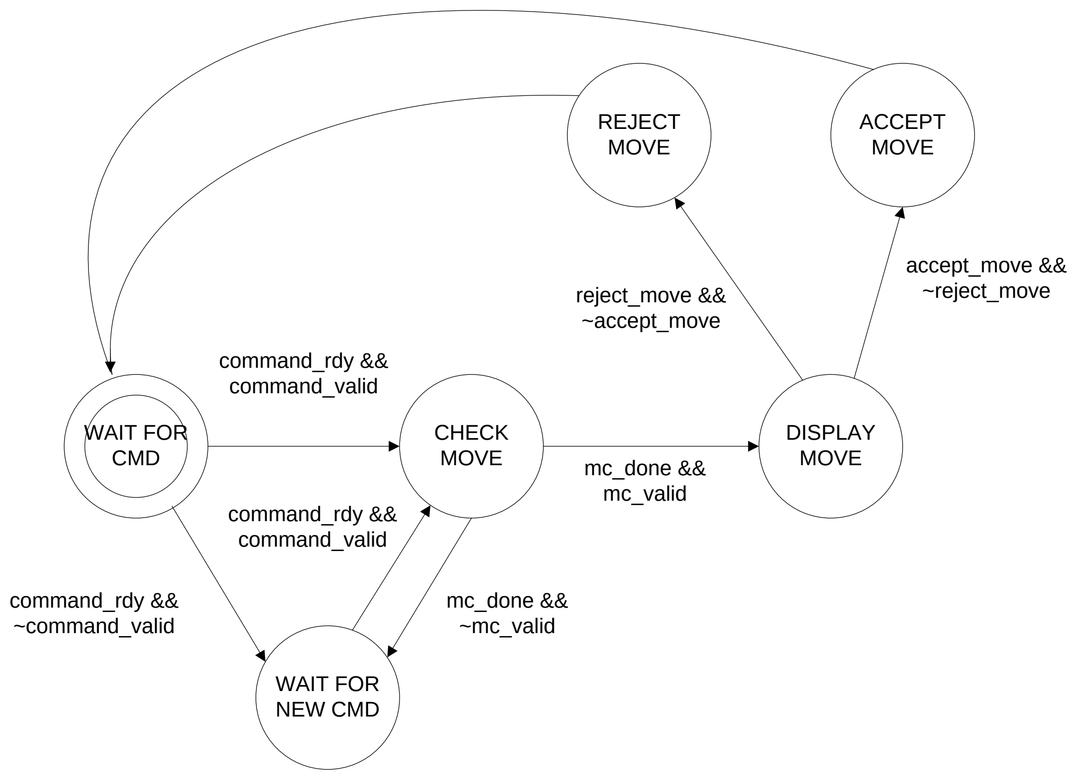

Figure 11. State diagram of chess engine FSM.

The FSM starts in the state WAIT FOR CMD, with a register indicating the current player initialized to WHITE. Two switches on the 6.111 LabKit are used to select the input signals for the two players. If the input is the voice recognition module, then the chess engine will wait for a ready signal from that. Otherwise it will wait for a ready signal from the keyboard encoder. On a ready signal from the voice recognition module, the FSM will direct the start of move checking to determine if the instructed move is permissible. On a ready signal from the keyboard encoder, the FSM will check that the encoding was valid before signaling the start of the move checker. An invalid encoding will land the FSM in the state WAIT FOR NEW CMD, which is also the state that the FSM transitions to if the move checker decided that an instructed move is impermissible. However, if the move checker determines that the instructed move is valid, the chess engine will update the internal representation of the chessboard – a multidimensional array of registers – to reflect the move in the state DISPLAY MOVE.

Because the voice recognition module is not expected to generate the correct move instruction all of the time, the chess engine keeps a second copy of the chessboard to allow the players (regardless of their input type) to review their moves. The two copies of the chessboard are designated old-chessboard and new-chessboard. Moves are always made to new-chessboard first. If a player rejects a move, then the two affected squares of new-chessboard are replaced with the values in the corresponding squares of old-chessboard. On the other hand, if a player accepts the move, then the two affected squares of new-chessboard are copied over to old-chessboard. Once a player accepts the move, his or her turn is over (the register indicating the current player switches to the other player), and the two chessboards are made identical by the start of the next player's turn.

The chess engine also handles pawn promotions, as well as outputs text to be displayed on the XVGA screen. When pawns have reached the other side of the chessboard, they are, due to design limitations, automatically promoted to queens. The chess engine outputs text to let the players know whose turn it is, as well as direct them as they play the game.

Move Checker

The move checker module determines whether an instructed move is valid according to the style of movement of the piece being moved. The move checker first determines the type of piece being moved by looking at the square in new-chessboard corresponding to the from-column and from-row specified in the move instruction. If there is a piece in the square, then the move checker verifies that it belongs to the current player, as well as verifies that the destination square, corresponding to the to-column and to-row specified in the move instruction, is empty or contains an opponent's piece. Only then does it check to see that the instructed move matches the style of movement of the piece being moved. Descriptions of each piece's style of movement are listed below.

Piece	Style of Movement
King	One square in any direction
Queen	Up to seven squares in any direction if its path is unhindered
Rook	Up to seven squares horizontally or vertically if its path is unhindered
Bishop	Up to seven squares diagonally if its path is unhindered
Knight	L-shaped – two squares horizontally or vertically, then one square vertically or horizontally, respectively
Pawn	Generally, one square up the rows for WHITE and down the rows for BLACK, but can advance two squares in the same manner on the piece's first move – captures are made one square in the forward diagonal direction

For queens, rooks, and bishops, the move checker directs a sub-module to iterate through the squares along its path of movement to determine if the path is clear or blocked by another piece. Otherwise, the move checker uses combinational logic to determine if a piece's movement matches its prescribed style of movement. The move checker signals for one cycle that it is done if a move instruction is determined to be valid or as soon as it is determined to be invalid. A valid signal is raised at the same time if the move instruction is valid.

Graphics Engine

The graphics engine produces the visual aspects of the chess game, pictured in the figure below.

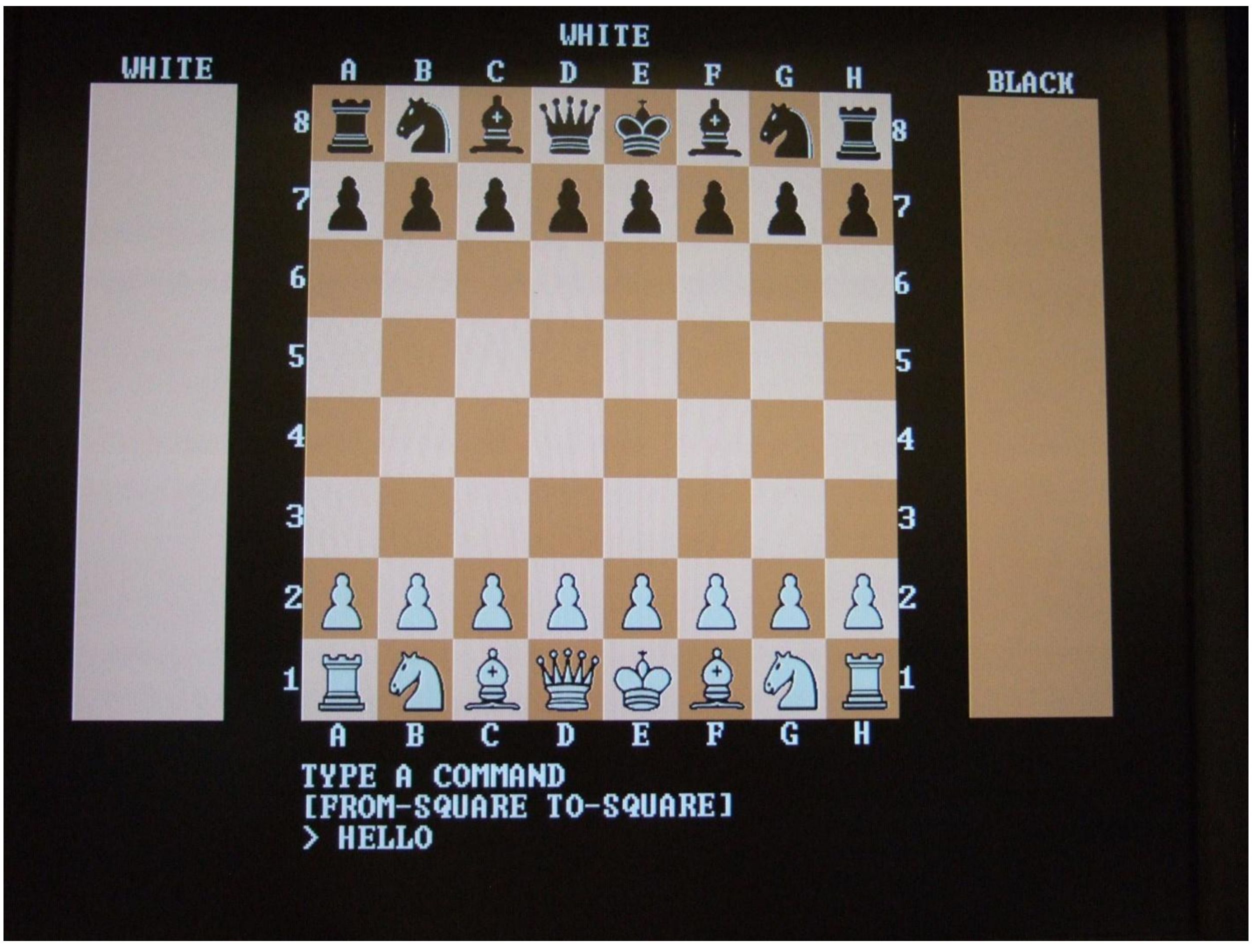

Figure 12. Graphics produced by the graphics engine.

Chessboard Drawer

Chessboard drawer generates the pixel values needed to render a chessboard on the XVGA display. The width of each square of the chessboard is 64 pixels, so within the bounds of the chessboard, the chessboard drawer uses only the higher order bits of normalized hcount and vcount signals to determine which of the two parameterized color values to output.

Chess Pieces Drawer

Chess pieces drawer generates the pixel values needed to render images of chess pieces on the XVGA display. Twelve images of the six types of pieces in white and black were obtained from Wikipedia. Using GIMP, the backgrounds of these images were filled with solid blue and the images were converted into .jpg files. A MATLAB script (Appendix P) was then used to generate .coe files from the .jpg files. The .coe files were then analyzed to determine what values the originally solid blue had become. Those values of blue are used as transparent colors.

Twelve ROMs for the chess piece images are instantiated in the chess pieces drawer. All twelve ROMs are addressed by the five lower order bits of normalized hcount and vcount signals. The data out from a specific ROM is selected if a corresponding piece needs to be drawn at the location specified by hcount and vcount. Otherwise, the module generates the pixel value corresponding to a blue transparent color. The pixel values from the ROM and the blue transparent color are 8-bits (3-bits red, 3-bits green, 2-bits blue), so they are upconverted into 24-bit values by padding with zeros.

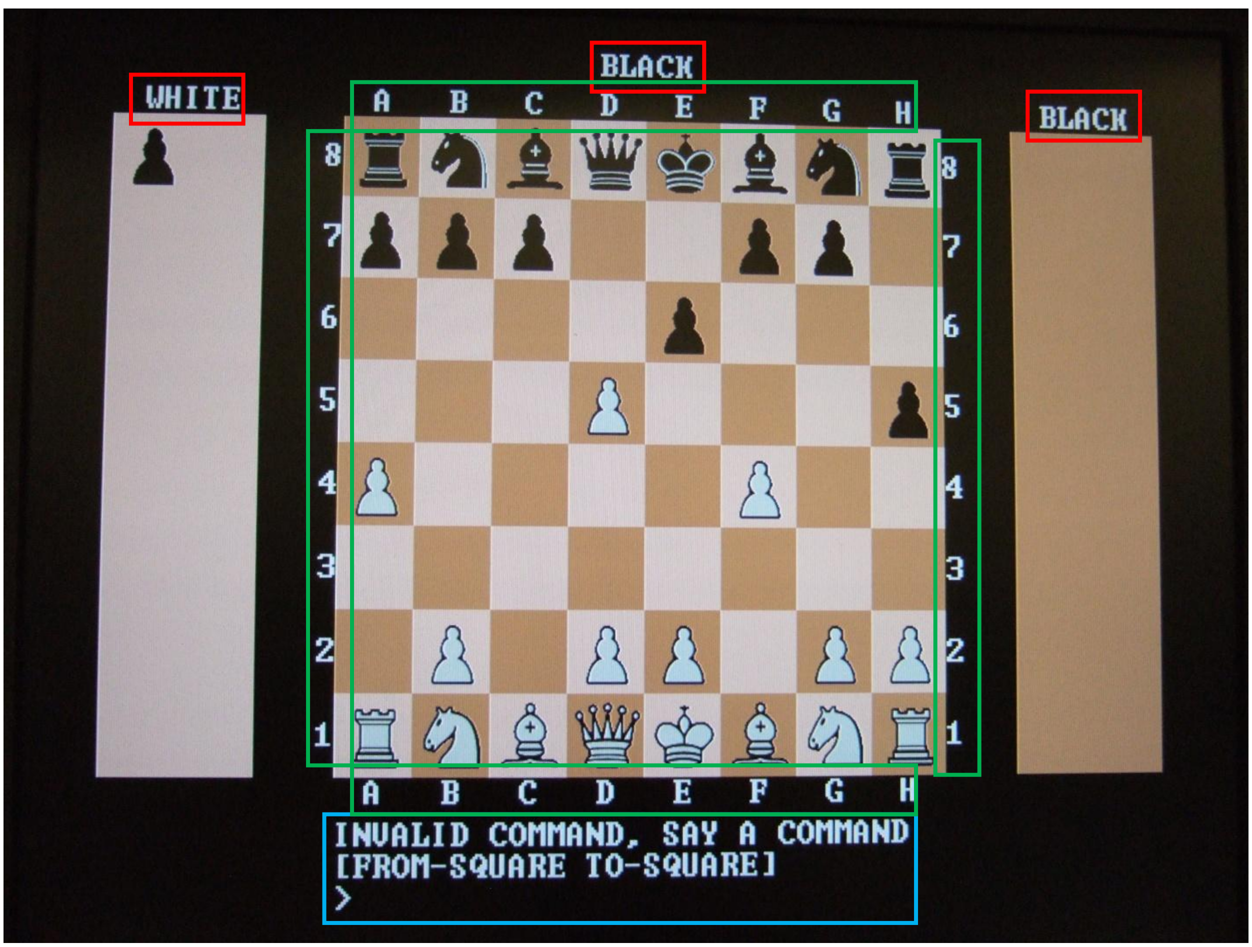

Figure 13. Text types. Labels are red. Column and row indicators are green.

Text Drawer
Text drawer generates all the pixel values for text to be rendered on the XVGA display. The text to be rendered is divided into three types: labels, column and row indicators for the chessboard, and body text. As indicated in the figure above, the labels include the headers for the grids of captured pieces as well as the turn indicator above the chessboard. Column and row indicators are the letters and numbers surrounding the perimeter of the chessboard. These indicators make it easier for the players to specify columns and rows in making moves. Body text consists of two lines of text from the chess engine and text from the keyboard entry module. The texts of each type are all the same length. Labels read either "WHITE" or "BLACK", which are five characters each. Column and row indicators are single characters. The actual lines of body text and text from the keyboard entry module vary, but are each padded to be 32 characters long. In order to render the three different types of text, three instances of Professors Ike Chuang's and Chris Terman's *cstringdisp*[2] module are instantiated with the appropriate parameter values. Since the text is designed to have no overlap, the outputs from the three *cstringdisp* modules are composed into one output by taking the bitwise OR of the three signals.

Chess Graphics
The chess graphics module combines the outputs from the chessboard drawer, the chess pieces drawer, the text drawer, as well as two solid-colored rectangles generated by the *blob*[3] module from Lab 5, to produce the final output to be displayed on the XVGA display. The chessboard, text, and solid-colored rectangles are designed so that they do not overlap. Thus, the outputs from chessboard drawer, text drawer, and two *blobs* can be combined simply by performing a bitwise OR on the four. The chess pieces produced by the chess pieces drawer, however, are intended to layer on top of the chessboard and the two solid colored rectangles. In terms of layering, the chess pieces make up the top layer and everything else make up the bottom layer. The graphics module displays the top layer when there are chess pieces to be displayed and the bottom layer otherwise. To accomplish this, the chess graphics module allows the bottom layer to filter through when pixel values in the top layer correspond to the blue transparent colors.

Testing and Debugging

Audio Recognition Hardware (Varun)
Debugging the Audio Recognition hardware initially done entirely in ModelSim, and once the ModelSim tests were satisfactory, the modules were debugged and tested on hardware. To develop solid testing practices and to build awareness of common mistakes that were being made, an informal bug log was kept for the first week of ModelSim testing. The shift connector, finite impulse response filter, and valid

[2] http://web.mit.edu/6.111/www/f2005/code/cstringdisp.v
[3] http://web.mit.edu/6.111/www/f2008/handouts/labs/lab5.html

checker modules were relatively easy to debug. They each required only ModelSim verification before being shown to work up to specification on hardware.
For the valid checker, additional testing on hardware was done to find the optimal threshold at which to acknowledge a valid word was spoken. It was integrated into the 6.111 lab4 audio recorder[4] and fed incoming audio streams with varying thresholds. The goal was to systematically determine what threshold provided the highest hit rate for spoken words while minimizing the rate of false positives (ie. prevent short audio bursts like claps from being acknowledged as valid words).

DTW Engine

The first stage of testing the DTW involved performing a proof of concept test on the DTW algorithm. For that, a version of the algorithm quickly scripted in Python was fed a series of audio test vectors and shown to perform satisfactorily. After this was shown, the algorithm was examined rigorously to find where improvements could be made to reduce memory requirements. At this stage, it would have been very beneficial to build a MATLAB or Python simulation of the hardware-optimized algorithm to ensure that no unforeseen errors could occur, but luckily careful planning was all that was necessary.

Once the frameworks for the algorithm were written in Verilog, they were run over and over again in ModelSim. Trying to pipeline the circuit for increased computational throughput created the biggest problem. This caused serious problems because of the necessary accesses to different parts of the DTW Memory. Eventually this was scrapped in favor of a sequential approach. While the sequential approach required almost six times as many clock cycles to complete as a working pipelined version would have, the incremental gain in time would have been imperceptible. Finally a version of the algorithm was shown to work on known test vectors in ModelSim and hardware testing began.

The first problem noticed when testing on hardware began almost immediately. When fed any audio samples, the algorithm would return a distance metric that hovered around a value that seemed linked to the template audio. That is, when the template audio was very loud, the distance returned would be very large, and vice versa. Using the Logic Analyzer, it was found to be a problem with state. The *DTW_done* signal was not being reset fast enough so as soon as the System Controller would move into the *to_DTW* state, it would see the done signal high, and only have one clock cycle to send data before its FSM would move it to the next state – meaning that the DTW Engine would compare the template against an audio sample with one valid input and the rest zeros.

The second major problem had to do with how the engine handled audio inputs. In simulation, test vectors were chosen to be unsigned integer values. As such, the algorithm code never specified that the audio samples should be treated as signed integers, and so seemingly small negative numbers were viewed as large positive

[4] http://web.mit.edu/6.111/www/f2008/handouts/labs/lab4.v

numbers. So while the algorithm continued to produce the proper values when hardwired to positive constants, when fed audio, the values that were being computed were absurd. Luckily, the fix was relatively simple—just adding the word signed to the Verilog.

DTW System Controller

The DTW System Controller was initially written in pseudo code on paper and then refined to be as robust as possible. Once the pseudo code was written, it was translated into Verilog and tested top to bottom. First, its ability to transition between states with proper input sequences was tested. At the same time, the system was shown not to glitch when many inputs arrived simultaneously, and instead dealt with them sequentially as it was supposed to.

After state transitions were tested, enable pulse timings were tested and debugged. Because the entire system has so many parts that depend on exact timing and transfer of data, this step was very important.

Next to be tested was the audio buffering and transfer procedure. The address incrementing and memory storage/output were examined to show that no data was being truncated, lost, or written improperly.

Once this was done, the system was ready to be tested on hardware, with DTW_done pulses controlled by switches and distance values set to by switches. The states, substates, and outputted commands were displayed on the hex display using the module provided in the 6.111 lab documentation. Because of the ModelSim testing, this phase only took a few hours to debug the entire controller system.

System Integration

Because the input/output and handshaking behavior of each module were understood from their individual ModelSim simulations, whole system integration was not too challenging. It was first done in ModelSim and shown to work. The ModelSim model was then compiled to hardware. After fixing the bug caused by not clearing the DTW_done signal before the *System Controller* reached the *to_DTW* state—which took three days to catch—and after losing a day to not specifying the proper bit width of an output, the system worked as hoped.

Proof of Concept Testing

Once the system was running, the performance of one single DTW Engine was tested. While the engine had trouble distinguishing between words like “alpha” and “beta” , it was shown to be able to distinguish between “funk” and “bridge” when trained on “funk”, and between “cat” and “dog” when trained on “dog” as can be seen in Figures 14 and 15(data for “cat v dog” and “funk v bridge”, see Appendix A). Further testing revealed on the entire system with all 8 DTW Engines showed that the system had between a 3% and 12% hit rate when matching complete chess commands to human voices—and had an almost 90% hit rate when trying to distinguish a spoken number 5 (See Appendix B).

The errors with the system are almost entirely due to the variable nature of human voice, as well as the environmental noise of the training and testing environment. When a computer generated voice was used to train the system and then inputted as test audio, the computer had a 100% hit rate – indicating that it could successfully match perfect signals. Furthermore, when the audio was sped up or slowed down by up to 12%, the algorithm still exhibited a 95% hit rate. This indicates that the DTW algorithm was, in fact, performing dynamic time warping successfully.

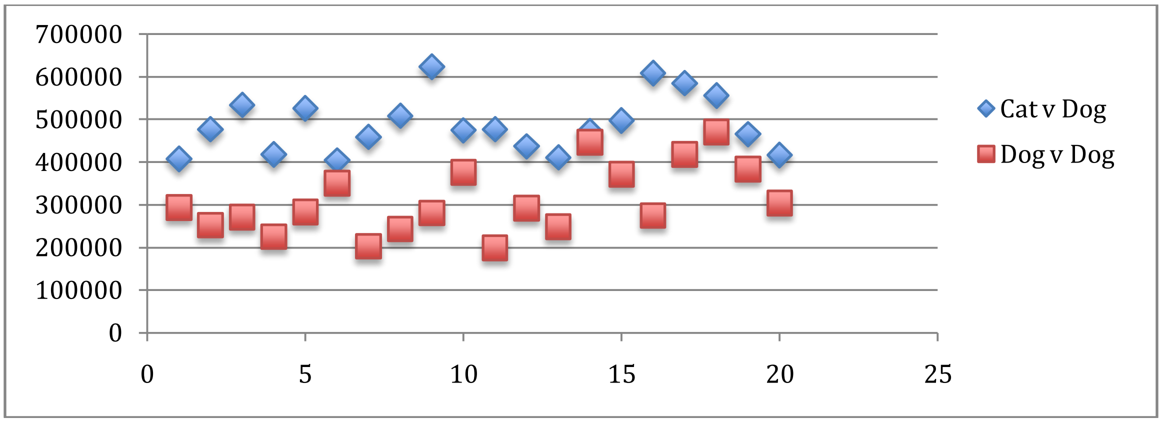

Figure 14. "Dog" and "Cat" when trained on "Dog".

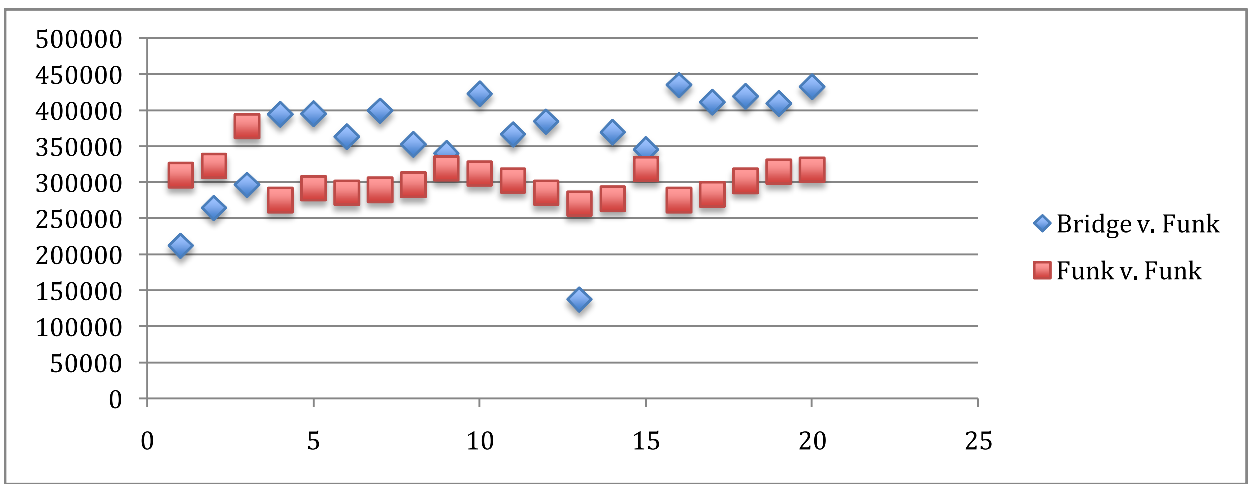

Figure 15. "Bridge" and "Funk" when trained on "Funk".

Chess Hardware (Michael)

Testing of the modules of the chess system was done either by visual inspection or by simulating using ModelSim. The keyboard entry and the graphics engine modules were tested by loading up the LabKit and checking the visual output on the XVGA display. Since the keyboard encoder and move checker had no visual components, they were tested in ModelSim. Problems were usually caught by checking the Warning messages in the Xilinx tool, as well as by carefully analyzing and reanalyzing the Verilog code that had been written. However, I did come upon an odd bug that took the help of Ben to resolve – at one point, my project file could not be opened by Xilinx, and any attempts to create a new project from the old Verilog file resulted in Xilinx closing unannounced. After numerous attempts at creating new projects and copying various fragments of code into to Xilinx, Ben noticed that a *localparam* had been assigned to itself. Sure enough, this circular assignment was the cause of the mysterious problem.

Conclusion

The Voice Controlled Chess Game built on the FPGA was successfully shown to demonstrate full chess visualizations and game play, as well as sufficient voice recognition capability.

The voice recognition hardware was shown to be a successful implementation of the Dynamic Time Warping algorithm. Controlling for the effects of environment noise and audio pitch, the system was able to detect and match input audio samples of varying rates of up to 15% from the trained sample. It also demonstrated that the DTW algorithm, which is normally thought to be memory inefficient, could be built in a way so as to preserve the functionality while greatly curbing memory usage.

The system's inability to reliably detect microphone inputs can be attributed to the shortcomings of comparing unfiltered time series of audio instead of converting audio streams features vectors that emphasize spoken voice and normalize for the speaker's pitch and amplitude. One such technique involves using Mel-cepstral coefficients to scale the Fourier Transform of incoming audio before passing these new scaled feature-vectors to the DTW Engine. This has been implemented on FPGAs and shown to perform better than the system implemented in this project.

The chess hardware was successfully implemented on the FPGA. Capable of taking both keyboard and speech commands, it can function both independently and as a part of the voice controlled chess system. The chess hardware features basic move checking – checking that moves match the style of movement of a piece. Thus, it does have a number of limitations, preventing it from functioning as a full-fledged chess system. Due to design limitations, the chess system does not allow a number of special moves, namely, en passant captures and castling. Pawn promotion is also limited to queens. Regardless of these shortcomings in the chess engine, the chess hardware is still complete, taking moves and displaying them on screen.

The Voice Controlled Chess Game is fully functional, although there are features that we would have liked to implement.

Appendices

Appendix A : Single DTW Test Data “Funk” v “Bridge” and “Cat” v “Dog”

Cat v Dog		Dog v Dog	
6360E	407054	47889	293001
74628	476712	3D6D3	251603
8255D	533853	420F0	270576
65E4C	417356	36CDE	224478
806ad	525997	44E5F	282207
628BA	403642	55812	350226
6FFAB	458667	3148A	201866
7C193	508307	3B28B	242315
9830C	623372	44706	280326
73B0A	473866	5B9CF	375247
7455F	476511	30770	198512
6A867	436327	47119	291097
6434D	410445	3C9A3	248227
731C7	471495	6CDF4	445940
796A4	497316	5A901	370945
94692	607890	43061	274529
8EC7E	584830	66072	417906
87AE8	555752	72C33	470067
7193C	465212	5D8AB	383147
65A2C	416300	4A219	303641

Mean	487545.2	308792.8
Median	475188.5	286652
Std Dev	67004.73487	79157.35594

Bridge v Funk		Funk v Funk	
33f16	212758	4be07	310791
40b71	265073	4efd1	323537
488a1	297121	5c625	378405
607a7	395175	436b9	276153
60b37	396087	476b1	292529
58dbf	363967	45de4	286180
61af2	400114	46e06	290310
564ad	353453	488d1	297169
53533	341299	4e272	320114
675e1	423393	4c6a9	313001
599e9	367081	49f7c	302972
5dfda	384986	45d59	286041
21afd	137981	423f5	271349
5a56a	370026	43d37	277815
548f7	346359	4de76	319094
6a5a0	435616	436cd	276173
64846	411718	4551a	283930
66826	419878	49e9d	302749
6423c	410172	4d0e9	315625
69d0d	433421	4dada	318170
33f16	212758	4be07	310791
40b71	265073	4efd1	323537
488a1	297121	5c625	378405
607a7	395175	436b9	276153

60b37	396087	476b1	292529
58dbf	363967	45de4	286180

Appendix B: Letter Hit Frequency Data

spoken	received	frequency
a	**a**	**0.31**
a	b	0.02
a	c	0.09
a	d	0.00
a	e	0.10
a	f	0.12
a	g	0.06
a	h	0.30
b	a	0.00
b	b	0.30
b	c	0.10
b	d	0.00
b	e	0.00
b	f	0.10
b	g	0.00
b	**h**	**0.40**
c	a	0.00
c	b	0.20
c	**c**	**0.00**
c	d	0.00
c	e	0.20
c	f	0.10
c	**g**	**0.30**
c	h	0.20
d	a	0.20
d	b	0.00
d	c	0.00
d	d	0.00
d	**e**	**0.50**
d	f	0.00
d	g	0.00
d	h	0.30
e	a	0.30
e	b	0.00
e	c	0.00
e	d	0.00
e	**e**	**0.40**
e	f	0.10
e	g	0.00
e	h	0.20
f	a	0.33
f	b	0.00
f	c	0.22
f	d	0.00
f	e	0.00
f	f	0.00
f	g	0.00

f		**h**		**0.44**
g		a		0.00
g		b		0.20
g		c		0.10
g		d		0.00
g		e		0.20
g		f		0.00
g		g		0.00
g		**h**		**0.40**
h		a		0.00
h		b		0.60
h		c		0.00
h		d		0.00
h		e		0.10
h		**f**		**0.30**
h		g		0.00
h		h		0.00
	1		**1**	**0.61**
	1		2	0.13
	1		3	0.01
	1		4	0.14
	1		5	0.08
	1		6	0.01
	1		7	0.01
	1		8	0.01
	2		1	0.00
	2		**2**	**0.70**
	2		3	0.00
	2		4	0.10
	2		5	0.00
	2		6	0.20
	2		7	0.00
	2		8	0.00
	3		1	0.00
	3		2	0.20
	3		**3**	**0.30**
	3		4	0.00
	3		5	0.10
	3		**6**	**0.30**
	3		7	0.10
	3		8	0.10
	4		1	0.00
	4		**2**	**0.60**
	4		3	0.10
	4		**4**	**0.30**
	4		5	0.00
	4		6	0.00
	4		7	0.00
	4		8	0.00
	5		1	0.00

5	2	0.00
5	3	0.00
5	4	0.10
5	**5**	**0.90**
5	6	0.00
5	7	0.00
5	8	0.00
6	1	0.00
6	2	0.00
6	3	0.00
6	4	0.30
6	5	0.10
6	**6**	**0.50**
6	7	0.00
6	8	0.00
7	1	0.00
7	**2**	**0.40**
7	3	0.00
7	4	0.20
7	5	0.20
7	6	0.00
7	**7**	**0.10**
7	8	0.00
8	1	0.10
8	2	0.20
8	3	0.00
8	4	0.10
8	5	0.10
8	**6**	**0.50**
8	7	0.00
8	**8**	**0.00**

Appendix C: Shift Connector Verilog

```
module shift_connector(     clock, reset, cat_in,word_in,
                                    train_in, train_audio_in,
                                    en_in, audio_in,
                                    en_out, audio_out,
                                    cat_out, train_audio_out,
                                    train_1, train_2, train_3, train_4,
                                    train_5, train_6, train_7, train_8);
                            input wire clock;
                            input wire reset;
                            input wire cat_in;
                            input wire [2:0] word_in;
                            input wire train_in;
                            input wire [7:0] train_audio_in;
                            input wire en_in;
                            input wire [7:0] audio_in;
                            output reg en_out;
                            output reg [7:0] audio_out;
                            output reg cat_out;
                            output reg [7:0] train_audio_out;
                            output reg train_1;
                            output reg train_2;
                            output reg train_3;
                            output reg train_4;
                            output reg train_5;
                            output reg train_6;
                            output reg train_7;
                            output reg train_8;

                            /*shift registers that hold things back two clock cycles*/
                            reg en_old;
                            reg en_old_old;
                            reg train_old;
                            reg train_old_old;
                            reg [7:0] audio_hold_1;
                            reg [7:0] train_hold_1;
                            reg cat_in_hold;
                            reg word_in_hold;

                            always @ (posedge clock) begin
                                    if (reset) begin
                                        audio_hold_1 <= 0;
                                        audio_hold_2 <= 0;
                                        train_hold_1 <= 0;
                                        train_hold_2 <= 0;
                                        en_old <= 0;
                                        en_old_old <= 0;
                                        train_old <= 0;
                                        train_old_old <= 0;
                                        en_out <= 0;
                                        audio_out <= 0;
                                        cat_out <= 0;
                                        train_audio_out <= 0;
                                        train_1 <= 0;
                                        train_2 <= 0;
                                        train_3 <= 0;
                                        train_4 <= 0;
                                        train_5 <= 0;
                                        train_6 <= 0;
                                        train_7 <= 0;
                                        train_8 <= 0;
                                    end
                                    else begin
                                        en_old <= en_in;
                                        en_old_old <= en_old;
```

```
                train_old <= train_in;
                train_old_old <= train_old;

                if (train_in || train_old_old) begin
                    en_out <= 0;
                    cat_in_hold <= (train_in) ? cat_in : cat_in_hold;
                    train_hold_1 <= (train_in) ? train_audio_in : 0;
                    train_audio_out <= train_hold_1;
                    cat_out <= cat_in;
                    train_1 <= (word_in == 0) ? 1 : 0;
                    train_2 <= (word_in == 1) ? 1 : 0;
                    train_3 <= (word_in == 2) ? 1 : 0;
                    train_4 <= (word_in == 3) ? 1 : 0;
                    train_5 <= (word_in == 4) ? 1 : 0;
                    train_6 <= (word_in == 5) ? 1 : 0;
                    train_7 <= (word_in == 6) ? 1 : 0;
                    train_8 <= (word_in == 7) ? 1 : 0;

                end
                else if (en_in || en_old_old) begin
                    train_1 <= 0;
                    train_2 <= 0;
                    train_3 <= 0;
                    train_4 <= 0;
                    train_5 <= 0;
                    train_6 <= 0;
                    train_7 <= 0;
                    train_8 <= 0;
                    audio_out <= audio_in;
                    cat_out <= cat_in;
                    en_out <= 1;
                end
                else begin
                    audio_hold_1 <= 0;
                    audio_hold_2 <= 0;
                    train_hold_1 <= 0;
                    train_hold_2 <= 0;
                    en_old <= 0;
                    en_old_old <= 0;
                    train_old <= 0;
                    train_old_old <= 0;
                    en_out <= 0;
                    audio_out <= 0;
                    train_audio_out <= 0;
                    train_1 <= 0;
                    train_2 <= 0;
                    train_3 <= 0;
                    train_4 <= 0;
                    train_5 <= 0;
                    train_6 <= 0;
                    train_7 <= 0;
                    train_8 <= 0;
                end
            end
        end
endmodule
```

Appendix D: FIR 31 Verilog

```
//////////////////////////////////////////////////////////////////////////
//////
//
// 31-tap FIR filter, 8-bit signed data, 10-bit signed coefficients.
// ready is asserted whenever there is a new sample on the X input,
// the Y output should also be sampled at the same time.  Assumes at
// least 32 clocks between ready assertions.  Note that since the
// coefficients have been scaled by 2**10, so has the output (it's
// expanded from 8 bits to 18 bits).  To get an 8-bit result from the
// filter just divide by 2**10, ie, use Y[17:10].
//
//////////////////////////////////////////////////////////////////////////
//////

module fir31(
 input wire clock,reset,ready,
 input wire signed [7:0] x,
 output reg signed [17:0] y
);
	reg signed [7:0] sample[31:0];	// buffer of 32 8-bit signed samples
	reg [4:0] offset;		// offset pointer for sample memory
	reg [4:0] index;

	wire signed [9:0] coeff;
	coeffs31 coeffs31(.index(index),.coeff(coeff));

	always @(posedge clock) begin
		if (reset) begin
			offset <= 0;
			index <= 0;
			y <= 0;
		end
		else if (ready) begin
			offset <= offset + 1;
			sample[offset] <= x;
			y <= 0;
			index <= 0;
		end
		else if (index < 31) begin
			y <= y + coeff * sample[(offset - index - 1) & 31] ;
			index <= index + 1;
		end
	end
endmodule
```

```
//////////////////////////////////////////////////////////////////////////////
//////
//
// Coefficients for a 31-tap low-pass FIR filter with Wn determined for a 4kHz
sampling rate.  Since we're doing integer arithmetic, we've scaled
// the coefficients by 2**10
// Matlab command: round(fir1(30,.2/24)*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)
          'd0:  coeff = -1;
     'd1:  coeff = -1;
     'd2:  coeff = -1;
     'd3:  coeff = 0;
     'd4:  coeff = 2;
     'd5:  coeff = 5;
     'd6:  coeff = 11;
     'd7:  coeff = 19;
     'd8:  coeff = 28;
     'd9:  coeff = 40;
     'd10:  coeff = 52;
     'd11:  coeff = 64;
     'd12:  coeff = 75;
     'd13:  coeff = 84;
     'd14:  coeff = 90;
     'd15:  coeff = 91;
     'd16:  coeff = 90;
     'd17:  coeff = 84;
     'd18:  coeff = 75;
     'd19:  coeff = 64;
     'd20:  coeff = 52;
     'd21:  coeff = 40;
     'd22:  coeff = 28;
     'd23:  coeff = 19;
     'd24:  coeff = 11;
     'd25:  coeff = 5;
     'd26:  coeff = 2;
```

```
    'd27:  coeff = 0;
    'd28:  coeff = -1;
    'd29:  coeff = -1;
    'd30:  coeff = -1;
     default: coeff = 10'hXXX;
   endcase
endmodule
```

Appendix E: DTW Engine Verilog

```
module dtw_engine2(input wire clock,
                                                          input wire reset,
                                                          input wire signed [7:0] train_in,//training audio
                                                          input wire train,
                                                          input wire en,
                                                          input wire signed [7:0] audio_in,
                                                          input wire category,    //1 bit toggle for category
                                                          output reg [25:0] distance,
                                                          output reg DTW_done);



        reg [11:0]a_temp;
        reg  we_temp;
        reg signed [7:0] mem_in_temp;
        wire signed [7:0] mem_out_temp;
        //Template Memory
        mybram #(.LOGSIZE(12),.WIDTH(8))
    template(.addr(a_temp),.clk(clock),.we(we_temp),.din(mem_in_temp),.dout(mem_out_temp));

  reg [10:0] a_match;
  reg we_match;
  reg signed [7:0] mem_in_match;
  wire signed [7:0] mem_out_match;
  //Match Memory
  mybram #(.LOGSIZE(11),.WIDTH(8))
    match(.addr(a_match),.clk(clock),.we(we_match),.din(mem_in_match),.dout(mem_out_match));

        reg [11:0] a_dtw;
        reg [11:0] a_dtw_store;
//stores the address of a_dtw so that we can go back to it while doing the address manipulation
        reg we_dtw;
        reg [19:0] mem_in_dtw;
        wire [19:0] mem_out_dtw;
        //DTW Memory
        mybram #(.LOGSIZE(12),.WIDTH(20))
    DTW(.addr(a_dtw),.clk(clock),.we(we_dtw),.din(mem_in_dtw),.dout(mem_out_dtw));



  /*miscellaneous */
  reg [2:0] state;
  reg [2:0] nextstate;
  reg [2:0] substate;
  reg train_old;
  reg train_old_old;
  reg en_old;
  reg en_old_old;
  reg we_dtw_old;
  reg a_dtw_toggle;
  reg clear_dtw_mem_f;
  reg [12:0] clear_count;

  /*pointers */
  reg [11:0] end_val;
  reg [11:0] start_val;
  reg [21:0] count;
  reg [10:0] rollcount;

  /*local storage */
  reg [25:0] ij;
//the four blocks needed for each calculation
  reg [25:0] i1j;
  reg [25:0] min_ij1_i1j1;
```

```
    reg [25:0] i1j1;
    reg [25:0] ij1;
    reg [17:0] Dij;
//difference between the two samples

    /*state*/
    localparam training = 1;
    localparam transfer = 2;
    localparam calculate = 3;
    localparam hold = 4;

    /*substate*/
    localparam write = 0;
    localparam add = 1;
    localparam read_dtw = 2;
    localparam read_mem = 3;
    localparam burn_clock = 4;

    always @ * begin
                        case (state)
                            training: nextstate <= (!train) ? hold : training;
                            transfer: nextstate <= (!en) ? calculate : transfer;
                            calculate: nextstate <= (DTW_done) ? hold : calculate;
                            hold: nextstate <= (train) ? training : ((en) ? transfer : hold);
                            default: nextstate <= training;
                        endcase

            end

            always @ (posedge clock) begin

                        if (reset) begin
                            a_dtw_toggle <= 0;
                                    state <= training;
                                    substate <= burn_clock;
                                    rollcount <= 0;
                                    count <= 0;
                           a_temp <= 0;
        we_temp <= 0;
        a_match <= 0;
        we_match <= 0;
        a_dtw <= 0;
                           we_dtw <= 0;
                           clear_dtw_mem_f <= 0;
                           clear_count <= 4100;
          a_dtw_store <= 0;
          count <= 0;
          rollcount <= 0;
          DTW_done <= 0;
          distance <= 0;
          ij <=0;
          Dij <= 0;
          i1j <= 0;
          ij1 <= 0;
          i1j1 <= 0;
          min_ij1_i1j1 <= 0;
                        end

                        else begin
                           end_val <= (category) ? 4095 : 2047;
                           train_old <= train;
                           train_old_old <= train_old;
                        // delays address incrementing
                           en_old <= en;
                           en_old_old <= en_old; //hack
                           we_dtw_old <= we_dtw; //hack -- resets a_match properly
                                    state <= nextstate;
                                    if (nextstate != state) begin
```

```
                    if (nextstate == training) begin
                            a_temp <= (category) ? 2048 : 0;
                            //set the start of the template recording
                            we_match <= 0;
                            we_dtw <= 0;
                    end
                    else if (nextstate == transfer) begin
                            a_match <= 0; //reset the match address
                            we_temp <= 0;
                            we_dtw <= 0;

                            count <= 0;
                            rollcount <= 0;
                            DTW_done <= 0;
                    end
                    else if (nextstate == calculate) begin
                      substate <= burn_clock;
                            a_dtw <= 0;
                            a_dtw_store <= 0;
                            a_match <= 0;
                            a_temp <= (category) ? 2048 : 0;
                            we_temp <= 0;
                            we_match <= 0;
                            Dij <= 0;
                            ij <= 0;
                            i1j <= 0;
                            ij1 <= 0;
                            i1j1 <= 0;
                            min_ij1_i1j1 <= 0;

                            distance <= 0;
                    end
            end

            else if (clear_dtw_mem_f == 1) begin
                if (clear_count == 0) begin
                    clear_dtw_mem_f <= 0;
                    we_dtw <= 1;
                    mem_in_dtw <= 0;

                             DTW_done <= 0;
                end

                    //explicitly hold the dtw_done on for 3 clock cycles and force a reset

                    else if (clear_count == 1 || clear_count == 2 || clear_count == 3) begin

                            DTW_done <= 1;
                  distance <= ij+Dij;

                            clear_count <= clear_count - 1;

                    end
                else begin
                    clear_count <= clear_count - 1;
                    a_dtw <= a_dtw - 1;
                    we_dtw <= 1;
                    mem_in_dtw <= 0;
                end

            end

            else if (state == training && (train || train_old_old) ) begin
                    //make sure we're in training state but not just defaulted
                    if (a_temp != end_val) begin
                    //to not load the previous train_in improperly when a shifts
if (train && train_old_old) begin
    we_temp <= 1;
```

```
                 mem_in_temp <= train_in;
                 a_temp <= a_temp + 1;
             end
             else if (train && !train_old_old) begin
                 we_temp <= 1;
                 mem_in_temp <= train_in;
             end
         end
         else begin
             we_temp <= 0;
         end
                                end

                                else if (state == transfer) begin
                                        if (a_match != 11'b11111111111) begin
                                            if (en && en_old_old) begin
                 we_match <= 1;
                 a_match <= a_match + 1;
                 mem_in_match <= audio_in;
             end
             else if (en && !en_old_old) begin
                 we_match <= 1;
                 mem_in_match <= audio_in;
             end
             end
         else if (a_match == 11'b11111111111) begin
                 we_match <= 0;

             end
                                end
                                else if (state == calculate) begin
                                    if (a_dtw_toggle == 0)begin
                                        a_match <= 0;
                                        a_temp <= (category) ? 2048 : 0;
                                        a_dtw_store <= 0;
                                    end

                                    if (substate == burn_clock)begin
                                        we_dtw <= 0;
                                        substate <= substate - 1;
                                        a_dtw <= a_dtw - 2047;
                                        end
                                        else if (substate == read_mem) begin
                                                //make Dij positive for posterity
Dij <= (mem_out_temp > mem_out_match) ? mem_out_temp - mem_out_match : mem_out_match - mem_out_temp;
                                                substate <= substate - 1;

                                        end

                                        else if (substate == read_dtw) begin
                                            if (a_dtw_toggle != 0) begin

                                                        Dij <= Dij*Dij; //square Dij
                                                  a_dtw_store <= a_dtw_store + 1;
                                                  i1j <= mem_out_dtw;
                                                  ij1 <= ij;
                                                  min_ij1_i1j1 <= (ij > i1j) ? i1j : ij;
                                                        //saves the two "left" squares as one

                                                end
                                            substate <= substate - 1;
                                        end

                                        else if (substate == add) begin
                                                if (count == 22'b1111111111111111111111) begin // last piece
                                                        a_dtw_toggle <= 0;
                                                        clear_dtw_mem_f <= 1; //clear the dtw memory!
                                                        clear_count <= 4100;
```

```
					end
					else begin
						substate <= substate - 1;
						if (count == 0 && a_dtw_toggle == 0) begin //first spot
							ij <= Dij;
						end
						else if (rollcount == 2047) begin //first column
							ij <= Dij + i1j;
							count <= count + 1;
						  rollcount <= rollcount + 1;
						end
						else if (count < 2048) begin //first row
							ij <= Dij + ij1;
							count <= count + 1;
						  rollcount <= rollcount + 1;
						end
						else if (count >= 2048) begin //normal pieces
							ij <= (min_ij1_i1j1 < i1j) ? Dij + min_ij1_i1j1 :  Dij + i1j;
							count <= count + 1;
						  rollcount <= rollcount + 1;
						end
					end

				end

				else if (substate == write) begin
				  substate <= burn_clock;
					if (count != 0 && rollcount == 0) begin
						a_temp <= a_temp + 1; end

					if (a_dtw_toggle == 0) begin
						  a_dtw_toggle <= 1;
						  a_dtw <= (category) ? 2048 : 0;
						  //a_dtw <= 0;
						  mem_in_dtw <= ij;
						  we_dtw <= 1;
					end
					 else begin
					      a_match <= a_match + 1;
						//increment everything
					      we_dtw <= 1;
					      a_dtw <= a_dtw_store;
					      mem_in_dtw <= ij;
					      a_dtw_toggle <= 1;
					  end

				  end

			end
		end

	end

endmodule
```

Appendix F: DTW System Controller + Valid Checker Verilog

```
module recorder(
  input wire clock,                         // 27mhz system clock
  input wire reset,                   // 1 to reset to initial state
  input wire playback,                // 1 for playback, 0 for record
  input wire ready,                   // 1 when AC97 data is available
  input wire [7:0] from_ac97_data, // 8-bit PCM data from mic
  input wire category_in,       //inputted category being trained
  input wire [2:0] word_in,           //which word is being trained
  input wire program_in,              //are we programming now?
  input wire [25:0] distance_1,    //distance calculated by DTW 1
  input wire [25:0] distance_2,    //distance calculated by DTW 2
  input wire [25:0] distance_3,    //distance calculated by DTW 3
  input wire [25:0] distance_4,    //distance calculated by DTW 4
  input wire [25:0] distance_5,    //distance calculated by DTW 5
  input wire [25:0] distance_6,    //distance calculated by DTW 6
  input wire [25:0] distance_7,    //distance calculated by DTW 7
  input wire [25:0] distance_8,    //distance calculated by DTW 8
  input wire DTW_done_1,              //done signal from DTW_1
  input wire DTW_done_2,              //done signal from DTW_2
  input wire DTW_done_3,              //done signal from DTW_3
  input wire DTW_done_4,              //done signal from DTW_4
  input wire DTW_done_5,              //done signal from DTW_5
  input wire DTW_done_6,              //done signal from DTW_6
  input wire DTW_done_7,              //done signal from DTW_7
  input wire DTW_done_8,              //done signal from DTW_8
  output reg [7:0] possible_match_out,          //audio sample to check against
  output reg [7:0] template_audio_out, //template audio
  output reg category_out,
  output reg [2:0] word_out,
  output reg en,                          //write enable for samples
  output reg temp_WE,                     //write enable for template memory
  output reg vr_new_command,               //enable signal for chess game
  output reg [11:0] vr_to_chess,          //output from VR to chess
  output reg LED_TO_RECORD,
  output reg [2:0] state_out,
  output reg [2:0] substate_out,
  output reg valid_disp,
  output reg [7:0] to_ac97_data);

            reg [10:0] a;                             //RAM address. initially zero
            reg [7:0] mem_in ;                                    //data to be written to RAM address a
            wire [7:0] mem_out;                                   //data outputted from RAM address a
            reg we;                                               //write enable for RAM
            //instantiate ram
   mybram #(.LOGSIZE(11),.WIDTH(8))
       ram(.addr(a),.clk(clock),.we(we),.din(mem_in),.dout(mem_out));
            reg [7:0] to_filter;
            wire [17:0] from_filter;
            fir31 fir(clock, reset, ready,to_filter,from_filter);

   reg vc_enable;
   wire valid;
   //instantiate the valid_checker
   valid_checker vc(.clk(clock), .reset(reset),.enable(vc_enable), .in(mem_in), .valid(valid));

            //counter used to determine when to sample
            reg [3:0] store_count;
            //the maximum memory address written to during a record cycle. so as not to play
            //previous recordings when in the playback mode
            reg [10:0] start_sample;
            reg [10:0] end_sample;
            reg [2:0] state;
            reg [2:0] next_state;
            reg [1:0] substate;
            reg [1:0] next_substate;
```

```
//major states of behavior
localparam training = 1;
localparam passive = 2;
localparam active = 3;
localparam to_dtw = 4; //also "to_template_memory"
localparam valid_out = 5;

//substates in determining what we're recording
localparam from_letter = 0;
localparam from_number = 1;
localparam to_letter = 2;
localparam to_number = 3;

//registers to hold values to be outputted
// no "to_number_r" because it's just concatenated to output, never saved
reg [2:0] from_letter_r; //A-F --> 0:7
reg [2:0] from_number_r; // 1-8 --> 0:7
reg [2:0] to_letter_r;

//flags and misc
reg end_record_f; //done recording
reg valid_done_f; //done outputting valid
reg valid_done_old; //holds old valid done
reg program_in_old;
reg all_done_f;   //all distances have returned
reg training_done_f; //done training the module
reg [1:0] valid_compare; //used to denote end of comparing distances
reg min1_2;             //represents DTW1 and DTW2 -- whichever is lesser
reg min3_4;
reg min5_6;
reg min7_8;
reg [2:0] min_so_far_l;    //min # of DTW's 1,2,3,4
reg [2:0] min_so_far_r;    //min # of DTW's 5,6,7
reg [2:0] min_dist;        //# of min DTW

//registers to hold DTW_done_i signals
reg dtwdun1;
reg dtwdun2;
reg dtwdun3;
reg dtwdun4;
reg dtwdun5;
reg dtwdun6;
reg dtwdun7;
reg dtwdun8;


always @ * begin
              substate_out = substate;
               state_out = state;
               valid_disp = valid;
    //all_done_f becomes a 1 clock long pulse that occurs when all DTW_done_i are 1
    all_done_f = (dtwdun1&&dtwdun2&&dtwdun3&&dtwdun4&&dtwdun5&&dtwdun6&&dtwdun7&&dtwdun8) ? 1 : 0;
    //training will be done clocked and override all!
    LED_TO_RECORD = (state == passive) ? 1 : 0;
    case (state)
        //determines next_state
        passive: next_state = (valid) ? active : passive;
        active: next_state = (end_record_f) ? to_dtw : active;
        to_dtw: next_state = (program_in) ? ( (training_done_f) ? passive : to_dtw) : ((all_done_f) ? valid_out : to_dtw);
        valid_out: next_state = (valid_done_f) ? passive : valid_out;
        default next_state = passive;
    endcase

    case(substate)
        //determines next_substate at the rising edge of valid_done_f
        from_letter: next_substate = (valid_done_f&&(~valid_done_old)) ? from_number : from_letter;
        from_number:next_substate = (valid_done_f&&(~valid_done_old)) ? to_letter : from_number;
        to_letter: next_substate = (valid_done_f&&(~valid_done_old)) ? to_number : to_letter;
```

```
            to_number:next_substate = (valid_done_f&&(~valid_done_old)) ? from_letter : to_number;
            default: next_substate = from_letter;
        endcase

        if (~program_in) begin
            if (next_substate == from_letter || next_substate == to_letter)
                category_out = 0;
            else if (next_substate == from_number || next_substate == to_number)
                category_out = 1;
        end
        else if (program_in) begin
             category_out = category_in;
             word_out = word_in;
                    end
       end

always @ (posedge clock) begin
        //ensure that all values are set to zero when the machine loads
                if (reset)begin
                    dtwdun1 <= 0;
                    dtwdun2 <= 0;
                    dtwdun3 <= 0;
                    dtwdun4 <= 0;
                    dtwdun5 <= 0;
                    dtwdun6 <= 0;
                    dtwdun7 <= 0;
                    dtwdun8 <= 0;
                    playback_old <= 0;
                                a <= 0;
                                start_sample <= 0;
                                end_sample <= 0;
                                en <= 0;
                                store_count <= 0;
                                we <= 1'b0;
                                state <= passive;
                                substate <= from_letter;
                                end_record_f <= 0;
                                valid_done_f <= 0;
                                valid_compare <= 3;
                                temp_WE <= 0;
                                end_record_f <=0;
        valid_done_f<=0;
        valid_done_old <=0;
        min1_2<=0;
        min3_4<=0;
        min5_6<=0;
        min7_8 <= 0;
        min_so_far_l<=0;
        min_so_far_r<=0;
        min_dist<=0;
        from_letter_r<=0;
        from_number_r<=0;
        to_letter_r<=0;
        vr_new_command <= 0;
        vr_to_chess <= 0;
        template_audio_out <= 0;
                end

                state <= next_state;
                substate <= (program_in) ? from_letter : next_substate;
                valid_done_old <= valid_done_f;

                 if (!all_done_f) begin
                     dtwdun1 <= (DTW_done_1) ? 1 : dtwdun1;
                     dtwdun2 <= (DTW_done_2) ? 1 : dtwdun2;
                     dtwdun3 <= (DTW_done_3) ? 1 : dtwdun3;
                     dtwdun4 <= (DTW_done_4) ? 1 : dtwdun4;
                     dtwdun5 <= (DTW_done_5) ? 1 : dtwdun5;
```

```
                                dtwdun6 <= (DTW_done_6) ? 1 : dtwdun6;
                                dtwdun7 <= (DTW_done_7) ? 1 : dtwdun7;
                                dtwdun8 <= (DTW_done_8) ? 1 : dtwdun8;
                            end
                            else if (all_done_f) begin
                                dtwdun1 <= 0;
                                dtwdun2 <= 0;
                                dtwdun3 <= 0;
                                dtwdun4 <= 0;
                                dtwdun5 <= 0;
                                dtwdun6 <= 0;
                                dtwdun7 <= 0;
                                dtwdun8 <= 0;
                            end

if (ready) begin
    if (state == passive) begin
        vr_new_command <= 0;
         valid_compare <= 3;
        valid_done_f <= 0;
        training_done_f <= 0;
        if (~playback) begin

            if (store_count == 11) begin //if we're on the 12th sample
                we <= 1;
                vc_enable <= 1;
                a <= a + 1;
                store_count <= 0;
                start_sample <= a - 127; //128 behind current sample
                end_sample <= start_sample; //129 behind current sample -- we will consider the last 128 samples as "valid"
                to_filter <= from_ac97_data;
          mem_in <= from_filter[17:10];
           to_ac97_data <= mem_out;
            end
            else begin
                we <= 0;
                vc_enable <= 0;
                store_count <= store_count + 1;
            end
        end
    end
    else if (state == active) begin
        vc_enable <= 0;
        if (~playback) begin
            if (store_count == 11)begin
                store_count <= 0;
                if(a == end_sample || a == start_sample) begin //handle the one clock of wait between changing states by allowing a to be end OR start_sample
                    end_record_f <= 1; //start sending to dtw
                    we <=0;
                    a <= start_sample; //move up one so we can access entire stored sample
                end
                else begin
                    we <= 1;
                    a <= a + 1;
         to_filter <= from_ac97_data;
                    mem_in <= from_filter[17:10];
         to_ac97_data <= mem_out;
                end
            end
            else begin
                we <= 0;
                store_count <= store_count + 1;
            end

        end
    end
end
```

```
 if (state == to_dtw) begin
        valid_compare <= 3;
        we <= 0;
        end_record_f <= 0;
     valid_done_f <= 0; //overspecification
        if (a != end_sample) begin
            if (program_in) begin
                a <= a + 1;
                template_audio_out <= mem_out;
                temp_WE <= 1;
            end
            else begin
                a <= a + 1;
                en <= 1;    //enable writing to DTW
                possible_match_out <= mem_out; //send the possible match down
            end
        end

        else if (!program_in && a == end_sample ) en <= 0;

        else if (program_in && a == end_sample) begin
            temp_WE <= 0;
            training_done_f <= 1;
        end
end

else if (state == valid_out) begin

    en <= 0;
    //build a 7->1 fan in comparator
   if (valid_compare == 3) begin
        min1_2 <= (distance_1 < distance_2) ? 0 : 1; //signifies which is lesser
        min3_4 <= (distance_3 < distance_4) ? 0 : 1;
        min5_6 <= (distance_5 < distance_6) ? 0 : 1;
        min7_8 <= (distance_7 < distance_8) ? 0 : 1;
        valid_compare <= 2;
    end
    else if (valid_compare == 2) begin
        case({min1_2, min3_4})
            2'b00: min_so_far_l <= (distance_1 < distance_3) ? 0 : 2;

            2'b11: min_so_far_l <= (distance_2 < distance_4) ? 1 : 3;

            2'b01: min_so_far_l <= (distance_1 < distance_4) ? 0 : 3;

            2'b10: min_so_far_l <= (distance_2 < distance_3) ? 1 : 2;
        endcase
        case ({min5_6, min7_8})
            2'b00: min_so_far_r <= (distance_5 < distance_7) ? 4 : 6;

            2'b11: min_so_far_r <= (distance_6 < distance_8) ? 5 : 7;

            2'b01: min_so_far_r <= (distance_5 < distance_8) ? 4 : 7;

            2'b10: min_so_far_r <= (distance_6 < distance_7) ? 5 : 6;
        endcase
        valid_compare <= 1;
    end
    else if (valid_compare == 1) begin
        case ({min_so_far_l, min_so_far_r})
            {3'b000,3'b100}: min_dist <= (distance_1 < distance_5) ? 0 : 4;
            {3'b001,3'b100}: min_dist <= (distance_2 < distance_5) ? 1 : 4;
            {3'b010,3'b100}: min_dist <= (distance_3 < distance_5) ? 2 : 4;
            {3'b011,3'b100}: min_dist <= (distance_4 < distance_5) ? 3 : 4;
            {3'b000,3'b101}: min_dist <= (distance_1 < distance_6) ? 0 : 5;
            {3'b001,3'b101}: min_dist <= (distance_2 < distance_6) ? 1 : 5;
            {3'b010,3'b101}: min_dist <= (distance_3 < distance_6) ? 2 : 5;
```

```
                    {3'b011,3'b101}: min_dist <= (distance_4 < distance_6) ? 3 : 5;
                    {3'b000,3'b110}: min_dist <= (distance_1 < distance_7) ? 0 : 6;
                    {3'b001,3'b110}: min_dist <= (distance_2 < distance_7) ? 1 : 6;
                    {3'b010,3'b110}: min_dist <= (distance_3 < distance_7) ? 2 : 6;
                    {3'b011,3'b110}: min_dist <= (distance_4 < distance_7) ? 3 : 6;
                    {3'b000,3'b111}: min_dist <= (distance_1 < distance_8) ? 0 : 7;
                    {3'b001,3'b111}: min_dist <= (distance_2 < distance_8) ? 1 : 7;
                    {3'b010,3'b111}: min_dist <= (distance_3 < distance_8) ? 2 : 7;
                    {3'b011,3'b111}: min_dist <= (distance_4 < distance_8) ? 3 : 7;
                endcase
                valid_compare <= 0;
            end

            else if (valid_compare == 0) begin
                if (substate == to_letter) begin
                    to_letter_r <= min_dist;
                    valid_done_f <= 1;
                end
                else if (substate == to_number) begin//hold the proper output high for 1 clocks
                    vr_new_command <= 1;
                    vr_to_chess <= {from_letter_r, from_number_r, to_letter_r, min_dist};
                    valid_done_f <= 1;
                end
                else if (substate == from_letter) begin
                    from_letter_r <= min_dist;
                    valid_done_f <= 1;
                end
                else if (substate == from_number) begin

             rom_number_r <= min_dist;
                   valid_done_f <= 1;
                end
            end


        end

             end
endmodule

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];
   integer i;

   initial begin
      for (i = 0; i < 32; i = i + 1) begin
        mem[i] = 8'd0;
  end
   end

   always @(posedge clk) begin
     if (we) mem[addr] <= din;
     dout <= mem[addr];
   end
endmodule
```

```
module valid_checker (input clk, input reset, input enable, input [7:0] in, output reg valid);

		//Throws an enable pulse if a valid audio sample is noted!
		reg [7:0] temp [255:0]; //holds data

		reg [7:0] temp_store;
    reg [19:0] sum_first; //sums the oldest 128 samples
    reg [19:0] sum_last; //sums the newest 128 samples
    reg [7:0] i;          //indexes up to 256
    reg [7:0] index;
    reg [7:0] top_index;
		reg clear_flag;                          //signals to hold valid high count
		reg [8:0] clear_count; //holds valid out for 2 clocks
		reg en_old;
		reg [7:0] abs_in;

    integer j;
    initial for(j = 0; j <= 255; j=j+1)
       temp[j] = 0;
    //use for loop to zero out things whenever program or train is hit!

    //One clock delay on valid --> please note
    always @ (posedge clk) begin
        if (reset)begin
           top_index <= 1;
           i <=0;
 index <= 128;
valid <= 0;
           sum_first <= 0;
           sum_last <= 0;
              clear_flag <= 0;
               clear_count <= 255;

        end
        else if (clear_flag) begin
             if (clear_count == 0) begin
                  clear_count <= 255;
                  clear_flag  <= 0;
                  temp[clear_count] <= 0;

             end
             else begin
                  valid <= 0;
                  temp[clear_count] <= 0;
                  clear_count <= clear_count -1;
             end
        end

        else begin
           en_old <= enable;
           if(enable && ~en_old) begin
                i<=i+1;
                top_index <= i+2;
              index <= i+129;
                temp[i] <= abs_in;
                sum_first <= (sum_first < temp[top_index]+temp[index]) ? 0 : sum_first - temp[top_index]+temp[index];//remove the oldest sample and shift over the 
middle sample
                sum_last <= (sum_last + abs_in < temp[index]) ? 0 : sum_last + abs_in - temp[index]; //add newest sample and remove the shifted one
             end
             if (sum_first + {5'b00100, 9'b000000000} < sum_last)begin //found empirically -- use 9'b000...
                sum_first <= 0;
                sum_last <= 0;
                top_index <= 1;
                i <=0;
              index <= 128;
                sum_first <= 0;
```

```
                sum_last <= 0;
                valid <= 1;
                clear_flag <= 1;
            end
            else begin
                valid <= 0;
           end
end
    end

                 always @ * begin
                 if (in[7] == 1)
                                abs_in = ~in + 1;//if negative, make positive
                 else
                                abs_in = in;
              end

endmodule
```

Appendix G: Modified Lab 4 W/ Instantiated Modules + Debouncer Verilog

```
`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 lab4audio (
  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 lab4(
  // Remove comment from any signals you use in your design!

  // AC97
  output wire /*beep,*/ audio_reset_b, ac97_synch, ac97_sdata_out,
  input wire ac97_bit_clock, ac97_sdata_in,

  // VGA
  /*
  output wire [7:0] vga_out_red, vga_out_green, vga_out_blue,
  output wire vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync, vga_out_vsync,
  */

  // NTSC OUT
  /*
  output wire [9:0] tv_out_ycrcb,
  output wire tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
  output wire tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
  output wire tv_out_subcar_reset;
  */
```

```
// NTSC IN
/*
input wire [19:0] tv_in_ycrcb,
input wire tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
output wire tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,
inout wire tv_in_i2c_data,
*/

// ZBT RAMS
/*
inout wire [35:0] ram0_data,
output wire [18:0] ram0_address,
output wire ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b,
output wire [3:0] ram0_bwe_b,
inout wire [35:0]ram1_data,
output wire [18:0]ram1_address,
output wire ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b,
output wire [3:0] ram1_bwe_b,
input  wire clock_feedback_in,
output wire clock_feedback_out,
*/

// FLASH
/*
inout wire [15:0] flash_data,
output wire [23:0] flash_address,
output wire flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b,
input wire flash_sts,
*/

// RS232
/*
output wire rs232_txd, rs232_rts,
input wire rs232_rxd, rs232_cts,
*/

// PS2
/*
input wire mouse_clock, mouse_data, keyboard_clock, keyboard_data,
*/

// FLUORESCENT DISPLAY

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


// BUTTONS, SWITCHES, LEDS
//input wire button0,
//input wire button1,
//input wire button2,
//input wire button3,
input wire button_enter,
//input wire button_right,
//input wire button_left,
input wire button_down,
input wire button_up,
input wire [7:0] switch,
output wire [7:0] led,

// USER CONNECTORS, DAUGHTER CARD, LOGIC ANALYZER
//inout wire [31:0] user1,
//inout wire [31:0] user2,
//inout wire [31:0] user3,
//inout wire [31:0] user4,
//inout wire [43:0] daughtercard,
output wire [15:0] analyzer1_data, output wire analyzer1_clock,
```

```
    //output wire [15:0] analyzer2_data, output wire analyzer2_clock,
    output wire [15:0] analyzer3_data, output wire analyzer3_clock,
    //output wire [15:0] analyzer4_data, output wire analyzer4_clock,

    // SYSTEM ACE
    /*
    inout wire [15:0] systemace_data,
    output wire [6:0] systemace_address,
    output wire systemace_ce_b, systemace_we_b, systemace_oe_b,
    input wire systemace_irq, systemace_mpbrdy,
    */

    // CLOCKS
    //input wire clock1,
    //input wire clock2,
    input wire clock_27mhz
);
     ////////////////////////////////////////////////////////////////////////////
     //
     // 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
     lab4audio 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 ledd;
              wire valid;
```

```
assign led[7:1] = 7'hFF;

assign led[0] = ~ledd;


// record module


wire [25:0] distance_1;

wire DTW_done_1;

wire [25:0] distance_2;
wire DTW_done_2;

wire [25:0] distance_3;
wire DTW_done_3;
wire [25:0] distance_4;
wire DTW_done_4;

wire [25:0] distance_5;
wire DTW_done_5;
wire [25:0] distance_6;
wire DTW_done_6;
wire [25:0] distance_7;
wire DTW_done_7;
wire [25:0] distance_8;
wire DTW_done_8;

wire [7:0] possible_match_out;

wire [7:0] template_audio_out;

wire en_out;

wire temp_WE;

wire vr_new_command;

wire [11:0] vr_to_chess;

wire [2:0] word_out;

wire category_in;

wire category_out;

wire [2:0] state_out;



wire en_out_sc;
wire [7:0] audio_out_sc;
wire cat_out_sc;
wire [7:0] train_audio_out;
wire train_1;
wire train_2;
wire train_3;
wire train_4;
wire train_5;
wire train_6;
wire train_7;
wire train_8;
```

```
wire [7:0] mem_in;

wire [7:0] mem_out;

wire we_debug;

wire [10:0] address_out;


wire [2:0] substate_out;

wire [11:0] vr_newcommandh = (vr_new_command) ? vr_to_chess  : vr_newcommandh;



recorder r(.clock(clock_27mhz), .reset(reset),.word_in(switch[2:0]), .LED_TO_RECORD(ledd), .playback(playback), .ready(ready), .from_ac97_data(from_ac97_data),

        .category_in(switch[5]), .program_in(switch[7]), .distance_3(distance_3),.DTW_done_3(DTW_done_3),

        .distance_2(distance_2),.DTW_done_2(DTW_done_2),.distance_1(distance_1),.DTW_done_1(DTW_done_1),

        .distance_4(distance_4),.DTW_done_4(DTW_done_4),.distance_5(distance_5),.DTW_done_5(DTW_done_5),

        .distance_6(distance_6),.DTW_done_6(DTW_done_6),.distance_8(distance_8),.DTW_done_8(DTW_done_8),

        .distance_7(distance_7),.DTW_done_7(DTW_done_7),


        .possible_match_out(possible_match_out), .template_audio_out(template_audio_out),

        .category_out(category_out), .word_out(word_out), .en(en_out), .temp_WE(temp_WE), .vr_new_command(vr_new_command),

        .vr_to_chess(vr_to_chess), .state_out(state_out), .substate_out(substate_out), .valid_disp(valid), .to_ac97_data(to_ac97_data)) ;




 shift_connector sc(        .clock(clock_27mhz), .reset(reset), .cat_in(category_out),.word_in(word_out),
                                        .train_in(temp_WE), .train_audio_in(template_audio_out),
                                        .en_in(en_out), .audio_in(possible_match_out),
                                        .en_out(en_out_sc), .audio_out(audio_out_sc),
                                        .cat_out(cat_out_sc), .train_audio_out(train_audio_out),
                                        .train_1(train_1), .train_2(train_2), .train_3(train_3),
.train_4(train_4),
                                        .train_5(train_5), .train_6(train_6), .train_7(train_7),
.train_8(train_8));



dtw_engine2 de0(.clock(clock_27mhz),

                    .reset(reset),

                    .train_in(train_audio_out),

                    .train(train_1),

                    .en(en_out_sc),
```

```
                    .audio_in(audio_out_sc),

                    .category(cat_out_sc) ,    //1 bit toggle for category

                    .distance(distance_1),

                    .DTW_done(DTW_done_1));


dtw_engine2 de1(.clock(clock_27mhz),
                    .reset(reset),
                    .train_in(train_audio_out),
                    .train(train_2),
                    .en(en_out_sc),
                    .audio_in(audio_out_sc),
                    .category(cat_out_sc) ,    //1 bit toggle for category
                    .distance(distance_2),
                    .DTW_done(DTW_done_2));


dtw_engine2 de2(.clock(clock_27mhz),
                    .reset(reset),
                    .train_in(train_audio_out),
                    .train(train_3),
                    .en(en_out_sc),
                    .audio_in(audio_out_sc),
                    .category(cat_out_sc) ,    //1 bit toggle for category
                    .distance(distance_3),
                    .DTW_done(DTW_done_3));

dtw_engine2 de3(.clock(clock_27mhz),
                    .reset(reset),
                    .train_in(train_audio_out),
                    .train(train_4),
                    .en(en_out_sc),
                    .audio_in(audio_out_sc),
                    .category(cat_out_sc) ,    //1 bit toggle for category
                    .distance(distance_4),
                    .DTW_done(DTW_done_4));


dtw_engine2 de4(.clock(clock_27mhz),
                    .reset(reset),
                    .train_in(train_audio_out),
                    .train(train_5),
                    .en(en_out_sc),
                    .audio_in(audio_out_sc),
                    .category(cat_out_sc) ,    //1 bit toggle for category
                    .distance(distance_5),
                    .DTW_done(DTW_done_5));

dtw_engine2 de5(.clock(clock_27mhz),
                    .reset(reset),
                    .train_in(train_audio_out),
                    .train(train_6),
                    .en(en_out_sc),
                    .audio_in(audio_out_sc),
                    .category(cat_out_sc) ,    //1 bit toggle for category
                    .distance(distance_6),
                    .DTW_done(DTW_done_6));

dtw_engine2 de6(.clock(clock_27mhz),
                    .reset(reset),
                    .train_in(train_audio_out),
                    .train(train_7),
```

```
                                                        .en(en_out_sc),
                                                        .audio_in(audio_out_sc),
                                                        .category(cat_out_sc) ,    //1 bit toggle for category
                                                        .distance(distance_7),
                                                        .DTW_done(DTW_done_7));

                        dtw_engine2 de7(.clock(clock_27mhz),
                                                        .reset(reset),
                                                        .train_in(train_audio_out),
                                                        .train(train_8),
                                                        .en(en_out_sc),
                                                        .audio_in(audio_out_sc),
                                                        .category(cat_out_sc) ,    //1 bit toggle for category
                                                        .distance(distance_8),
                                                        .DTW_done(DTW_done_8));

        display_16hex disp(.reset(reset), .clock_27mhz(clock_27mhz), .data({1'b0,state_out,1'b0, substate_out, 40'h0000000000,1'b0, vr_newcommandh[11:9], 1'b0,
vr_newcommandh[8:6], 1'b0, vr_newcommandh[5:3], 1'b0, vr_newcommandh[2:0]}),
                        .disp_blank(disp_blank), .disp_clock(disp_clock), .disp_rs(disp_rs), .disp_ce_b(disp_ce_b),
                        .disp_reset_b(disp_reset_b), .disp_data_out(disp_data_out));


   // output useful things to the logic analyzer connectors
   assign analyzer1_clock = clock_27mhz;
   //assign analyzer1_data[0] = valid;
   //assign analyzer1_data[7:0] = template_audio_out;
   assign analyzer1_data[15:0] = {3'b000, vr_new_command, vr_to_chess};
   //assign analyzer1_data[3] = ac97_synch;
   //assign analyzer1_data[15:3] = 0;
   assign analyzer3_clock = clock_27mhz;

//   assign analyzer3_clock = clock_27mhz;
   assign analyzer3_data = {state_out, address_out, 2'b00};

endmodule




///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Hex display driver
//
// File:    display_16hex.v
// Date:    24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove clear
// 28-Nov-06 CJT: fixed race condition between CE and RS (thanks Javier!)
//
// This verilog module drives the labkit hex dot matrix displays, and puts
// up 16 hexadecimal digits (8 bytes).  These are passed to the module
// through a 64 bit wire ("data"), asynchronously.
//
///////////////////////////////////////////////////////////////////////////////

module display_16hex (reset, clock_27mhz, data,
                        disp_blank, disp_clock, disp_rs, disp_ce_b,
                        disp_reset_b, disp_data_out);

   input reset, clock_27mhz;    // clock and reset (active high reset)
   input [63:0] data;                   // 16 hex nibbles to display

   output disp_blank, disp_clock, disp_data_out, disp_rs, disp_ce_b,
           disp_reset_b;
```

```
reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

////////////////////////////////////////////////////////////////////////////
//
// Display Clock
//
// Generate a 500kHz clock for driving the displays.
//
////////////////////////////////////////////////////////////////////////////

reg [4:0] count;
reg [7:0] reset_count;
reg clock;
wire dreset;

always @(posedge clock_27mhz)
  begin
        if (reset)
          begin
            count = 0;
            clock = 0;
          end
        else if (count == 26)
          begin
            clock = ~clock;
            count = 5'h00;
          end
        else
          count = count+1;
  end

always @(posedge clock_27mhz)
  if (reset)
    reset_count <= 100;
  else
    reset_count <= (reset_count==0) ? 0 : reset_count-1;

assign dreset = (reset_count != 0);

assign disp_clock = ~clock;

////////////////////////////////////////////////////////////////////////////
//
// Display State Machine
//
////////////////////////////////////////////////////////////////////////////

reg [7:0] state;                // FSM state
reg [9:0] dot_index;            // index to current dot being clocked out
reg [31:0] control;             // control register
reg [3:0] char_index;   // index of current character
reg [39:0] dots;                // dots for a single digit
reg [3:0] nibble;               // hex nibble of current character

assign disp_blank = 1'b0; // low <= not blanked

always @(posedge clock)
  if (dreset)
    begin
          state <= 0;
          dot_index <= 0;
          control <= 32'h7F7F7F7F;
    end
  else
    casex (state)
          8'h00:
            begin
```

```
      // Reset displays
      disp_data_out <= 1'b0;
      disp_rs <= 1'b0; // dot register
      disp_ce_b <= 1'b1;
      disp_reset_b <= 1'b0;
      dot_index <= 0;
      state <= state+1;
   end

8'h01:
   begin
      // End reset
      disp_reset_b <= 1'b1;
      state <= state+1;
   end

8'h02:
   begin
      // Initialize dot register (set all dots to zero)
      disp_ce_b <= 1'b0;
      disp_data_out <= 1'b0; // dot_index[0];
      if (dot_index == 639)
              state <= state+1;
      else
              dot_index <= dot_index+1;
   end

8'h03:
   begin
      // Latch dot data
      disp_ce_b <= 1'b1;
      dot_index <= 31;                  // re-purpose to init ctrl reg
      disp_rs <= 1'b1; // Select the control register
      state <= state+1;
   end

8'h04:
   begin
      // Setup the control register
      disp_ce_b <= 1'b0;
      disp_data_out <= control[31];
      control <= {control[30:0], 1'b0};    // shift left
      if (dot_index == 0)
              state <= state+1;
      else
              dot_index <= dot_index-1;
   end

8'h05:
   begin
      // Latch the control register data / dot data
      disp_ce_b <= 1'b1;
      dot_index <= 39;                  // init for single char
      char_index <= 15;                 // start with MS char
      state <= state+1;
      disp_rs <= 1'b0;                  // Select the dot register
   end

8'h06:
   begin
      // Load the user's dot data into the dot reg, char by char
      disp_ce_b <= 1'b0;
      disp_data_out <= dots[dot_index]; // dot data from msb
      if (dot_index == 0)
        if (char_index == 0)
          state <= 5;                              // all done, latch data
              else
              begin
```

```
                    char_index <= char_index - 1;          // goto next char
                    dot_index <= 39;
                  end
              else
                  dot_index <= dot_index-1; // else loop thru all dots
            end

      endcase

   always @ (data or char_index)
     case (char_index)
       4'h0:          nibble <= data[3:0];
       4'h1:          nibble <= data[7:4];
       4'h2:          nibble <= data[11:8];
       4'h3:          nibble <= data[15:12];
       4'h4:          nibble <= data[19:16];
       4'h5:          nibble <= data[23:20];
       4'h6:          nibble <= data[27:24];
       4'h7:          nibble <= data[31:28];
       4'h8:          nibble <= data[35:32];
       4'h9:          nibble <= data[39:36];
       4'hA:          nibble <= data[43:40];
       4'hB:          nibble <= data[47:44];
       4'hC:          nibble <= data[51:48];
       4'hD:          nibble <= data[55:52];
       4'hE:          nibble <= data[59:56];
       4'hF:          nibble <= data[63:60];
     endcase

   always @(nibble)
     case (nibble)
       4'h0: dots <= 40'b00111110_01010001_01001001_01000101_00111110;
       4'h1: dots <= 40'b00000000_01000010_01111111_01000000_00000000;
       4'h2: dots <= 40'b01100010_01010001_01001001_01001001_01000110;
       4'h3: dots <= 40'b00100010_01000001_01001001_01001001_00110110;
       4'h4: dots <= 40'b00011000_00010100_00010010_01111111_00010000;
       4'h5: dots <= 40'b00100111_01000101_01000101_01000101_00111001;
       4'h6: dots <= 40'b00111100_01001010_01001001_01001001_00110000;
       4'h7: dots <= 40'b00000001_01110001_00001001_00000101_00000011;
       4'h8: dots <= 40'b00110110_01001001_01001001_01001001_00110110;
       4'h9: dots <= 40'b00000110_01001001_01001001_00101001_00011110;
       4'hA: dots <= 40'b01111110_00001001_00001001_00001001_01111110;
       4'hB: dots <= 40'b01111111_01001001_01001001_01001001_00110110;
       4'hC: dots <= 40'b00111110_01000001_01000001_01000001_00100010;
       4'hD: dots <= 40'b01111111_01000001_01000001_01000001_00111110;
       4'hE: dots <= 40'b01111111_01001001_01001001_01001001_01000001;
       4'hF: dots <= 40'b01111111_00001001_00001001_00001001_00000001;
     endcase

endmodule
```

Appendix H: Labkit File for Chess System

```
///////////////////////////////////////////////////////////////////////
////////
//
// Pushbutton Debounce Module (video version)
//
///////////////////////////////////////////////////////////////////////
////////

module debounce (input reset, clock, noisy,
                 output reg clean);

   reg [19:0] count;
   reg new;

   always @(posedge clock)
     if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != new) begin new <= noisy; count <= 0; end
     else if (count == 650000) clean <= new;
     else count <= count+1;

endmodule

///////////////////////////////////////////////////////////////////////
////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////
////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is
an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
```

```
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated
into
//    the data bus, and the byte write enables have been combined into
the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is
now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////
////////
//
// Complete change history (including bug fixes)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb
devices
//              actually populated on the boards. (The boards support
up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a
default
//              value. (Previous versions of this file declared this
port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb
devices
//              actually populated on the boards. (The boards support
up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////
////////

module labkit   (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 [23:0] 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;
   assign audio_reset_b = 1'b0;
   assign ac97_synch = 1'b0;
   assign ac97_sdata_out = 1'b0;
   // ac97_sdata_in is an input

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


//////////////////////////////////////////////////////////////////////
/////
   //
   // chess chess chess
   //

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

   // use FPGA's digital clock manager to produce a
   // 65MHz clock (actually 64.8MHz)
   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
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
                  .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;

   // ENTER button is user reset
   wire reset,user_reset;
   debounce
db1(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button_enter),.c
lean(user_reset));
   assign reset = user_reset | power_on_reset;

   // UP and DOWN buttons for pong paddle
   wire accept_move, reject_move;
```

```
   debounce
db2(.reset(reset),.clock(clock_65mhz),.noisy(~button_up),.clean(accept_
move));
   debounce
db3(.reset(reset),.clock(clock_65mhz),.noisy(~button_down),.clean(rejec
t_move));

   // 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));

        // receive keyboard input as ascii
        wire [6*8-1:0] kb_string;
        wire kb_string_rdy;
        keyboard_entry kbe(clock_65mhz, reset, keyboard_clock,
                keyboard_data, kb_string, kb_string_rdy);

        // encode ascii into chess command
        wire [11:0] kb_command;
        wire kb_command_rdy;
        wire kb_command_valid;
        wire [6*8-1:0] kben_string;
        keyboard_encoder kben(clock_65mhz, reset, kb_string,
                kb_string_rdy, kb_command, kb_command_rdy,
kb_command_valid,
                kben_string);

        wire [63:0] data;
        // chess engine
        wire cvsync;
        assign user1[31:13] = 19'hZ;
        reg sync_rdy_1, sync_rdy_2, speech_command_rdy;
        reg [11:0] sync_command_1, sync_command_2, speech_command;
        always @(posedge clock_65mhz)
                begin
                        sync_rdy_1 <= user1[12];
                        sync_command_1 <= user1[11:0];
                        sync_rdy_2 <= sync_rdy_1;
                        sync_command_2 <= sync_command_1;
                        speech_command_rdy <= sync_rdy_2;
                        speech_command <= sync_command_2;
                end
        wire [64*5-1:0] flattened_chessboard;
        wire [16*3-1:0] flattened_white_captures;
        wire [16*3-1:0] flattened_black_captures;
        wire [5*8-1:0] player_string;
        wire [32*8-1:0] string_1, string_2;
        chess_engine ce(clock_65mhz, reset, cvsync, switch[3:2],
                speech_command_rdy, speech_command, kb_command_rdy,
                kb_command, kb_command_valid, accept_move, reject_move,
                flattened_chessboard, flattened_white_captures,
                flattened_black_captures, player_string, string_1,
                string_2, data[63:60]);
```

```
	// //*****dummy values for testing*****
	// reg [64*5-1:0] flattened_chessboard = 0;
	// wire [16*3-1:0] flattened_white_captures = 48'b001001;
	// wire [16*3-1:0] flattened_black_captures = 48'b001001;
	// wire [32*8-1:0] string_1 = "this really sucks";
	// wire [32*8-1:0] string_2 = "a lot";
	// wire [5*8-1:0] player_string = "WHITE";
	// wire [6*8-1:0] kb_string = "HELLO ";

	// generate graphics for chess game
	wire chsync,cblank;
	wire [23:0] cpixel;
	chess_graphics cg(clock_65mhz, hcount, vcount,
		hsync, vsync, blank, flattened_chessboard,
flattened_white_captures,
		flattened_black_captures, player_string, string_1,
string_2, kb_string,
		chsync, cvsync, cblank, cpixel, switch[7]);

	// hex display...fun
	// wire [63:0] data = 64'b0;
	// assign data[63:60] = 4'h0;
	assign data[59:48] = kb_command;
	assign data[47:0] = kben_string;
	display_16hex hex(reset, clock_65mhz, data, disp_blank,
disp_clock,
		disp_rs, disp_ce_b, disp_reset_b, disp_data_out);

	// //////////////////////////////////// GRAPHICS TEST
///////////////////////////////////
	// localparam [2:0] KING = 3'd6;
	// localparam [2:0] QUEEN = 3'd5;
	// localparam [2:0] ROOK = 3'd4;
	// localparam [2:0] BISHOP = 3'd3;
	// localparam [2:0] KNIGHT = 3'd2;
	// localparam [2:0] PAWN = 3'd1;

	// localparam WHITE = 1'b1;
	// localparam BLACK = 1'b0;

	// localparam MOVED = 1'b1;
	// localparam UNMOVED = 1'b0;

	// localparam [4:0] EMPTY = 5'b0;

	// reg [4:0] chessboard [7:0][7:0];

	// // link flattened representation of chessboard to
	// // multidimensional array of chessboard
	// integer c;
	// integer r;
	// always
		// begin
			// for (c = 0; c < 8; c = c + 1)
				// begin
					// for (r = 0; r < 8; r = r + 1)
						// begin
```

```
                                                            //
flattened_chessboard[((8*c+r+1)*5-1)-:5] = chessboard[c][r];
                                                    // end
                                    // end
                // end
        // integer co;
        // integer ro;
        // initial
                // begin
                        // for (co = 0; co < 8; co = co + 1)
                                // begin
                                // // place a pawn in every column in
rows 1 and 6
                                // chessboard[co][1] =
{PAWN,WHITE,UNMOVED};
                                // chessboard[co][6] =
{PAWN,BLACK,UNMOVED};
                                        // // empty every square between
rows 2 and 6 (inclusive)
                                        // for (ro = 2; ro < 6; ro = ro +
1)
                                                // begin
                                                        //
chessboard[co][ro] = EMPTY;
                                                // end
                                // end

                        // // set up main pieces
                        // chessboard[0][0] = {ROOK,WHITE,UNMOVED};
                        // chessboard[1][0] = {KNIGHT,WHITE,UNMOVED};
                        // chessboard[2][0] = {BISHOP,WHITE,UNMOVED};
                        // chessboard[3][0] = {QUEEN,WHITE,UNMOVED};
                        // chessboard[4][0] = {KING,WHITE,UNMOVED};
                        // chessboard[5][0] = {BISHOP,WHITE,UNMOVED};
                        // chessboard[6][0] = {KNIGHT,WHITE,UNMOVED};
                        // chessboard[7][0] = {ROOK,WHITE,UNMOVED};

                        // chessboard[0][7] = {ROOK,BLACK,UNMOVED};
                        // chessboard[1][7] = {KNIGHT,BLACK,UNMOVED};
                        // chessboard[2][7] = {BISHOP,BLACK,UNMOVED};
                        // chessboard[3][7] = {QUEEN,BLACK,UNMOVED};
                        // chessboard[4][7] = {KING,BLACK,UNMOVED};
                        // chessboard[5][7] = {BISHOP,BLACK,UNMOVED};
                        // chessboard[6][7] = {KNIGHT,BLACK,UNMOVED};
                        // chessboard[7][7] = {ROOK,BLACK,UNMOVED};
                // end

        // /////////////////////////////////// GRAPHICS TEST
///////////////////////////

   // switch[1:0] selects which video generator to use:
   //  00: chess
   //  01: 1 pixel outline of active video area (adjust screen
controls)
   //  10: color bars
   reg [23:0] rgb;
   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 <= (hcount==0 | hcount==1023 | vcount==0 | vcount==767) ?
24'hFF_FF_FF : 0;
      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: chess
         hs <= chsync;
         vs <= cvsync;
         b <= cblank;
         rgb <= cpixel;
      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;


endmodule

//////////////////////////////////////////////////////////////////////
/////////
//
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
//
//////////////////////////////////////////////////////////////////////
/////////

module xvga(input vclock,
            output reg [10:0] hcount,    // pixel number on current
line
            output reg [9:0] vcount,   // line number
            output reg vsync,hsync,blank);

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   reg hblank,vblank;
   wire hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount == 1023);
```

```
   assign hsyncon = (hcount == 1047);
   assign hsyncoff = (hcount == 1183);
   assign hreset = (hcount == 1343);

   // vertical: 806 lines total
   // display 768 lines
   wire vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount == 767);
   assign vsyncon = hreset & (vcount == 776);
   assign vsyncoff = hreset & (vcount == 782);
   assign vreset = hreset & (vcount == 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 @(posedge vclock) begin
      hcount <= hreset ? 0 : hcount + 1;
      hblank <= next_hblank;
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

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

      blank <= next_vblank | (next_hblank & ~hreset);
   end
endmodule
```

Appendix I: Keyboard Entry

```
module keyboard_entry (
        input clock_65mhz,
        input reset,
        input keyboard_clock,
        input keyboard_data,
        output reg [6*8-1:0] string,
        output reg string_rdy);

// ascii codes
localparam [7:0] ENTER = 8'h0D;
localparam [7:0] BACKSPACE = 8'h08;
localparam [7:0] NULL = 8'h00;

// keyboard input converted into ascii
wire [7:0] ascii;
wire char_rdy;
ps2_ascii_input ascii_input(clock_65mhz, reset,
                        keyboard_clock, keyboard_data, ascii, char_rdy);


// string_rdy goes high when enter key is pressed;
// old_string_rdy is 1-cycle delayed copy of string_rdy,
// and is used to determine if enter key was pressed
// one cycle ago (character array is held for one cycle
// after enter key is pressed before being cleared)
reg old_string_rdy;
always @(posedge clock_65mhz)
        begin
                string_rdy <= (char_rdy && (ascii == ENTER));
                old_string_rdy <= string_rdy;
        end

reg [7:0] char_array [5:0];    // character(ascii) array
reg [3:0] index = 5;           // index of character array

// character array - clears on reset or one cycle
// after enter key is pressed
integer i;
always @(posedge clock_65mhz)
        begin
                if (reset || (old_string_rdy && ~string_rdy))
                        begin
                                index <= 5;
                                for (i = 0; i < 6; i = i + 1)
                                        begin
                                                char_array[i] <= NULL;
                                        end
                        end
                else if (char_rdy && ~string_rdy)
                        begin
                                if (ascii == BACKSPACE)
                                        begin
                                                if (index < 5)
                                                        begin
```

```
        char_array[index+1] <= NULL;
                                                                  index <=
index + 1;
                                                          end
                                            end
                                    else if (index > 0)
                                            begin
                                                    char_array[index] <=
ascii;
                                                    index <= index - 1;
                                            end
                        end
        end

// tie string to character array
integer j;
always @(posedge clock_65mhz)
        begin
                for (j = 0; j < 6; j = j + 1)
                        begin
                                string[(7+(8*j))-:8] <= char_array[j];
                        end
        end

endmodule
```

Appendix J: Keyboard Encoder

```
module keyboard_encoder (
        input clock_65mhz,
        input reset,
        input [6*8-1:0] kb_string,
        input kb_string_rdy,
        output reg [11:0] command,
        output reg command_rdy,
        output reg command_valid,
        output reg [6*8-1:0] string);

// register to hold kb_string
//reg [6*8-1:0] string;

// instantiate encoder for from square
wire [2*8-1:0] from_substring = string[(6*8-1)-:(2*8)] ;
wire [5:0] from_command_values;
wire from_valid;
col_row_encoder from_square(from_substring, from_command_values,
from_valid);

// instantate encoder for to square
wire [2*8-1:0] to_substring = string[(3*8-1)-:(2*8)];
wire [5:0] to_command_values;
wire to_valid;
col_row_encoder to_square(to_substring, to_command_values, to_valid);

// instante validifier for whitespace
wire [7:0] whitespace_substring = string[(4*8-1)-:8];
wire whitespace_valid;
whitespace_validifier between_to_and_from(whitespace_substring,
whitespace_valid);

reg started;
always @(posedge clock_65mhz)
        begin
                if (reset)
                        begin
                                command_rdy <= 0;
                                command_valid <= 0;
                                started <= 0;
                        end
                else if (started)
                        begin
                                command_rdy <= 1;
                                command_valid <= (from_valid && to_valid
&& whitespace_valid);
                                command <=
{from_command_values,to_command_values};
                                started <= 0;
                        end
                else if (kb_string_rdy)
                        begin
                                string <= kb_string;
                                command_rdy <= 0;
```

```
                                command_valid <= 0;
                                started <= 1;
                        end
                else
                        begin
                                command_rdy <= 0;
                                command_valid <= 0;
                                started <= 0;
                        end
        end

endmodule

// encodes one letter (A-H) and one number (1-8)
// into column and row in chess command encoding
module col_row_encoder (
        input [2*8-1:0] substring,
        output reg [5:0] command_values,
        output reg valid);

reg valid_col;
reg valid_row;

always @(*)
        begin
                case(substring[(2*8-1)-:8])
                        "A":
                                begin
                                        command_values[5:3] = 3'd0;
                                        valid_col = 1;
                                end
                        "B":
                                begin
                                        command_values[5:3] = 3'd1;
                                        valid_col = 1;
                                end

                        "C":
                                begin
                                        command_values[5:3] = 3'd2;
                                        valid_col = 1;
                                end
                        "D":
                                begin
                                        command_values[5:3] = 3'd3;
                                        valid_col = 1;
                                end
                        "E":
                                begin
                                        command_values[5:3] = 3'd4;
                                        valid_col = 1;
                                end
                        "F":
                                begin
                                        command_values[5:3] = 3'd5;
                                        valid_col = 1;
                                end
```

```
        "G":
                begin
                        command_values[5:3] = 3'd6;
                        valid_col = 1;
                end
        "H":
                begin
                        command_values[5:3] = 3'd7;
                        valid_col = 1;
                end
        default: valid_col = 0;
endcase

case(substring[(8-1)-:8])
        "1":
                begin
                        command_values[2:0] = 3'd0;
                        valid_row = 1;
                end
        "2":
                begin
                        command_values[2:0] = 3'd1;
                        valid_row = 1;
                end
        "3":
                begin
                        command_values[2:0] = 3'd2;
                        valid_row = 1;
                end
        "4":
                begin
                        command_values[2:0] = 3'd3;
                        valid_row = 1;
                end
        "5":
                begin
                        command_values[2:0] = 3'd4;
                        valid_row = 1;
                end
        "6":
                begin
                        command_values[2:0] = 3'd5;
                        valid_row = 1;
                end
        "7":
                begin
                        command_values[2:0] = 3'd6;
                        valid_row = 1;
                end
        "8":
                begin
                        command_values[2:0] = 3'd7;
                        valid_row = 1;
                end
        default: valid_row = 0;
endcase
```

```
                valid = (valid_col && valid_row);
        end
endmodule

// validates a whitespace character
module whitespace_validifier (
        input [7:0] substring,
        output valid);

        localparam [7:0] SPACE = 8'h20;
        localparam [7:0] NULL = 8'h00;

        assign valid = (substring == SPACE || substring == NULL);
endmodule

`timescale 1ns / 100ps
module test_kbe ();

reg clk;
reg reset;
reg [5*8-1:0] kb_string;
reg kb_string_rdy;
wire [11:0] command;
wire command_rdy;
wire command_valid;
initial
        begin
                clk = 0;
                forever #7 clk = ~clk; // 14 ns period clock
        end

keyboard_encoder kbe(clk, reset, kb_string, kb_string_rdy,
                                        command, command_rdy,
command_valid);

initial
        begin
                #14
                $display("Keyboard Entry: 'A4 A5'");
                kb_string = "A4 A5";
                kb_string_rdy = 1;
                #14
                kb_string_rdy = 0;

                #28
                $display("Keyboard Entry: '6.111'");
                kb_string = "6.111";
                kb_string_rdy = 1;
                #14
                kb_string_rdy = 0;
        end

always @(posedge clk)
        begin
                if (command_rdy)
                        $display ("Command: %b \n Valid: %b", command,
command_valid);
```

```
        end
endmodule
```

Appendix K: Chess Engine

```
module chess_engine(
        input clock_65mhz,
        input reset,
        input cvsync,
        input [1:0] input_mode,              // switches[3:2]
        input speech_command_rdy,
        input [11:0] speech_command,
        input kb_command_rdy,
        input [11:0] kb_command,
        input kb_valid,
        input accept_move,                   // button up
        input reject_move,                   // button down
        output reg [64*5-1:0] flattened_chessboard,
        output reg [16*3-1:0] flattened_white_captures,
        output reg [16*3-1:0] flattened_black_captures,
        output reg [5*8-1:0] player_string,
        output reg [32*8-1:0] string_1,
        output reg [32*8-1:0] string_2,
        output reg [2:0] state);

localparam [2:0] KING = 3'd6;
localparam [2:0] QUEEN = 3'd5;
localparam [2:0] ROOK = 3'd4;
localparam [2:0] BISHOP = 3'd3;
localparam [2:0] KNIGHT = 3'd2;
localparam [2:0] PAWN = 3'd1;

localparam WHITE = 1'b1;
localparam BLACK = 1'b0;

localparam MOVED = 1'b1;
localparam UNMOVED = 1'b0;

localparam [4:0] EMPTY = 5'b0;
localparam [2:0] NONE = 3'b0;

localparam S_WAIT_FOR_CMD = 0;
localparam S_WAIT_FOR_NEW_CMD = 1;
localparam S_CHECK_MOVE = 2;
localparam S_DISPLAY_MOVE = 3;
localparam S_ACCEPT_MOVE = 4;
localparam S_REJECT_MOVE = 5;

reg delayed_cvsync;
wire negedge_cvsync = (~cvsync & delayed_cvsync);
always @(posedge clock_65mhz)
        begin
                delayed_cvsync <= cvsync;
        end

reg player;                     // current player (1 = WHITE, 0 = BLACK)

wire command_rdy = (input_mode[player]) ? speech_command_rdy :
kb_command_rdy;
```

```
wire [11:0] command = (input_mode[player]) ? speech_command :
kb_command;
wire command_valid = (input_mode[player]) ? 1 : kb_valid;

reg mc_start;
reg mc_color;
reg [11:0] mc_command;
wire [2:0] from_col = mc_command[11:9];
wire [2:0] from_row = mc_command[8:6];
wire [2:0] to_col = mc_command[5:3];
wire [2:0] to_row = mc_command[2:0];
wire mc_done;
wire mc_valid;

move_checker mc(clock_65mhz, reset, mc_start, mc_color, mc_command,
                               flattened_chessboard, mc_done, mc_valid);

//reg [2:0] state, next_state;
reg [2:0] next_state;
always @(*)
        begin
                case(state)
                        S_WAIT_FOR_CMD:         next_state = (command_rdy)
? ((command_valid) ? S_CHECK_MOVE : S_WAIT_FOR_NEW_CMD) :
S_WAIT_FOR_CMD;
                        S_WAIT_FOR_NEW_CMD:     next_state = (command_rdy)
? ((command_valid) ? S_CHECK_MOVE : S_WAIT_FOR_NEW_CMD) :
S_WAIT_FOR_NEW_CMD;
                        S_CHECK_MOVE:           next_state = (mc_done) ?
((mc_valid) ? S_DISPLAY_MOVE : S_WAIT_FOR_NEW_CMD) : S_CHECK_MOVE;
                        S_DISPLAY_MOVE:         next_state = (accept_move
&& ~reject_move) ? S_ACCEPT_MOVE :

                        S_ACCEPT_MOVE:          next_state =
S_WAIT_FOR_CMD;
                        S_REJECT_MOVE:          next_state =
S_WAIT_FOR_CMD;
                        default:                        next_state =
S_WAIT_FOR_CMD;
                endcase
        end


reg [4:0] old_chessboard[7:0][7:0];
reg [4:0] new_chessboard[7:0][7:0];

reg [2:0] white_captures[15:0];
reg [3:0] white_captures_index;
reg [2:0] black_captures[15:0];
reg [3:0] black_captures_index;

reg piece_captured;

integer c;
integer r;
integer i;
always @(posedge clock_65mhz)
```

```
        begin
                if (reset)
                        begin
                                player <= WHITE;
                                state <= S_WAIT_FOR_CMD;

                                white_captures_index <= 0;
                                black_captures_index <= 0;
                                for (i = 0; i < 16; i = i + 1)
                                        begin
                                                white_captures[i] <= NONE;
                                                black_captures[i] <= NONE;
                                        end

                                for (c = 0; c < 8; c = c + 1)
                                        begin
                                                // place a pawn in every
square in rows 1 and 6
                                                old_chessboard[c][1] <=
{PAWN,WHITE,UNMOVED};
                                                old_chessboard[c][6] <=
{PAWN,BLACK,UNMOVED};

                                                new_chessboard[c][1] <=
{PAWN,WHITE,UNMOVED};
                                                new_chessboard[c][6] <=
{PAWN,BLACK,UNMOVED};

                                        // empty every square between
rows 2 and 6 (inclusive)
                                        for (r = 2; r < 6; r = r + 1)
                                                begin

        old_chessboard[c][r] <= EMPTY;

        new_chessboard[c][r] <= EMPTY;
                                                end
                                        end

                                // set up main pieces on chessboards
                                old_chessboard[0][0] <=
{ROOK,WHITE,UNMOVED};
                                old_chessboard[1][0] <=
{KNIGHT,WHITE,UNMOVED};
                                old_chessboard[2][0] <=
{BISHOP,WHITE,UNMOVED};
                                old_chessboard[3][0] <=
{QUEEN,WHITE,UNMOVED};
                                old_chessboard[4][0] <=
{KING,WHITE,UNMOVED};
                                old_chessboard[5][0] <=
{BISHOP,WHITE,UNMOVED};
                                old_chessboard[6][0] <=
{KNIGHT,WHITE,UNMOVED};
                                old_chessboard[7][0] <=
{ROOK,WHITE,UNMOVED};
```

```
                              old_chessboard[0][7] <=
{ROOK,BLACK,UNMOVED};
                              old_chessboard[1][7] <=
{KNIGHT,BLACK,UNMOVED};
                              old_chessboard[2][7] <=
{BISHOP,BLACK,UNMOVED};
                              old_chessboard[3][7] <=
{QUEEN,BLACK,UNMOVED};
                              old_chessboard[4][7] <=
{KING,BLACK,UNMOVED};
                              old_chessboard[5][7] <=
{BISHOP,BLACK,UNMOVED};
                              old_chessboard[6][7] <=
{KNIGHT,BLACK,UNMOVED};
                              old_chessboard[7][7] <=
{ROOK,BLACK,UNMOVED};

                              new_chessboard[0][0] <=
{ROOK,WHITE,UNMOVED};
                              new_chessboard[1][0] <=
{KNIGHT,WHITE,UNMOVED};
                              new_chessboard[2][0] <=
{BISHOP,WHITE,UNMOVED};
                              new_chessboard[3][0] <=
{QUEEN,WHITE,UNMOVED};
                              new_chessboard[4][0] <=
{KING,WHITE,UNMOVED};
                              new_chessboard[5][0] <=
{BISHOP,WHITE,UNMOVED};
                              new_chessboard[6][0] <=
{KNIGHT,WHITE,UNMOVED};
                              new_chessboard[7][0] <=
{ROOK,WHITE,UNMOVED};

                              new_chessboard[0][7] <=
{ROOK,BLACK,UNMOVED};
                              new_chessboard[1][7] <=
{KNIGHT,BLACK,UNMOVED};
                              new_chessboard[2][7] <=
{BISHOP,BLACK,UNMOVED};
                              new_chessboard[3][7] <=
{QUEEN,BLACK,UNMOVED};
                              new_chessboard[4][7] <=
{KING,BLACK,UNMOVED};
                              new_chessboard[5][7] <=
{BISHOP,BLACK,UNMOVED};
                              new_chessboard[6][7] <=
{KNIGHT,BLACK,UNMOVED};
                              new_chessboard[7][7] <=
{ROOK,BLACK,UNMOVED};
                       end
               else
                       begin
                              // state actions
                              // (at the end of players turn/at the
start of new players turn,
                              //  new_chessboard = old_chessboard)
```

```
                              if (state == S_DISPLAY_MOVE)
                                   begin
                                        if (negedge_cvsync)
                                             begin
                                                  // empty
out from-square

        new_chessboard[from_col][from_row] <= EMPTY;

                                                  // move
piece to to-square
                                                  // if pawn
has reached an end-row, promote to queen
                                                  if
((old_chessboard[from_col][from_row][4:2] == PAWN)
                                                       &&
((player == WHITE && to_row == 7) ||
                                                            (player ==

        begin
                                                            new_chessbo
                                                       end
                                                  else

        begin
                                                            new_chessbo
                                                       end

                                                  // if a
piece is being captured, place it in the appropriate player's
                                                  // list of
captured pieces (and signal, internally, that a piece has
                                                  // been
captured)
                                                  if
((new_chessboard[to_col][to_row] != EMPTY) &&

        (player ^ new_chessboard[to_col][to_row][1]))

        begin
                                                            if (player
                                                                 beg

                                                                 end
                                                            else
                                                                 beg

                                                                 end
                                                            piece_captu
                                                       end

                                                  state <=
next_state;
                                             end
                                   end
```

```
                              else if (state == S_REJECT_MOVE)
                                   begin
                                        if (negedge_cvsync)
                                             begin
                                                  // revert
the affected squares

     new_chessboard[to_col][to_row] <=
old_chessboard[to_col][to_row];

     new_chessboard[from_col][from_row] <=
old_chessboard[from_col][from_row];

                                                  // if a
piece was captured, remove it from the
                                                  //
appropriate player's list of captured pieces
                                                  if
(piece_captured)

     begin
                                                            if (player
                                                                 beg


                                                                 end
                                                            else
                                                                 beg


                                                                 end
                                                       end


     piece_captured <= 0;
                                                  state <=
next_state;
                                             end
                                   end
                              else if (state == S_ACCEPT_MOVE)
                                   begin
                                        if (negedge_cvsync)
                                             begin

     old_chessboard[to_col][to_row] <=
new_chessboard[to_col][to_row];

     old_chessboard[from_col][from_row] <=
old_chessboard[from_row][from_col];

     piece_captured <= 0;
                                                  player <=
~player;
                                                  state <=
next_state;
                                             end
                                   end
```

```
                                // transition actions
                                else if ((state == S_WAIT_FOR_CMD ||
state == S_WAIT_FOR_NEW_CMD) &&
                                                 (next_state ==
S_CHECK_MOVE))
                                        begin
                                                mc_command <= command;
                                                mc_start <= 1;
                                                mc_color <= player;
                                                state <= next_state;
                                        end
                                else
                                        begin
                                                mc_start <= 0;
                                                piece_captured <= 0;
                                                state <= next_state;
                                        end
                        end
        end

always @(posedge clock_65mhz)
        begin
                // only update the strings when cvsync is low
                if (negedge_cvsync)
                        begin
                                if (state == S_WAIT_FOR_CMD)
                                        begin
                                                string_1 <=
(input_mode[player]) ?
                                                            "SAY A COMMAND
" :

        "TYPE A COMMAND                  ";
                                                string_2 <= "[FROM-SQUARE
TO-SQUARE]         ";
                                        end
                                else if (state == S_WAIT_FOR_NEW_CMD)
                                        begin
                                                string_1 <=
(input_mode[player]) ?

        "INVALID COMMAND, SAY A COMMAND  " :

        "INVALID COMMAND, TYPE A COMMAND ";
                                                string_2 <= "[FROM-SQUARE
TO-SQUARE]         ";
                                        end
                                else if (state == S_DISPLAY_MOVE)
                                        begin
                                                string_1 <= "ACCEPT MOVE
or REJECT MOVE      ";
                                                string_2 <= " UP BUTTON
DOWN BUTTON      ";
                                        end
                                else
                                        begin
                                                string_1 <= "";
```

```
                                                        string_2 <= "";
                                                end
                                        player_string <= (player) ? "WHITE" :
"BLACK";
                                end
                end

// link flattened representation of chessboard to
// multidimensional array of chessboard
// and flattned representation of captured pieces
// list to array representation
integer co;
integer ro;
integer in;
always
        begin
                for (co = 0; co < 8; co = co + 1)
                        begin
                                for (ro = 0; ro < 8; ro = ro + 1)
                                        begin

        flattened_chessboard[((8*co+ro+1)*5-1)-:5] =
new_chessboard[co][ro];
                                        end
                        end
                for (in = 0; in < 16; in = in + 1)
                        begin
                                flattened_white_captures[((in+1)*3-1)-:3]
= white_captures[in];
                                flattened_black_captures[((in+1)*3-1)-:3]
= black_captures[in];
                        end
        end

endmodule
```

Appendix L: Move Checker

```
// yozo
module move_checker (
        input clock_65mhz,
        input reset,
        input start,
        input color,
        input [11:0] command,
        input [64*5-1:0] flattened_chessboard,
        output reg done,
        output reg valid);

localparam [2:0] KING = 3'd6;
localparam [2:0] QUEEN = 3'd5;
localparam [2:0] ROOK = 3'd4;
localparam [2:0] BISHOP = 3'd3;
localparam [2:0] KNIGHT = 3'd2;
localparam [2:0] PAWN = 3'd1;
```

```
localparam WHITE = 1'b1;
localparam BLACK = 1'b0;

localparam MOVED = 1'b1;
localparam UNMOVED = 1'b0;

localparam [4:0] EMPTY = 5'b0;

// multidimensional array representation of chessboard
reg [4:0] chessboard [7:0][7:0];

// link flattened representation of chessboard to
// multidimensional array of chessboard
integer c;
integer r;
always @(*)
	begin
		for (c = 0; c < 8; c = c + 1)
			begin
				for (r = 0; r < 8; r = r + 1)
					begin
						chessboard[c][r] =
flattened_chessboard[((8*c+r+1)*5-1)-:5];
					end
			end
	end

reg [2:0] piece;
reg [2:0] from_col;
reg [2:0] from_row;
reg [2:0] to_col;
reg [2:0] to_row;

wire signed [3:0] delta_col = to_col - from_col;
wire signed [3:0] delta_row = to_row - from_row;

wire [3:0] mag_delta_col = (delta_col[3]) ? ~delta_col + 1 : delta_col;
wire [3:0] mag_delta_row = (delta_row[3]) ? ~delta_row + 1 : delta_row;

reg isc_start;
wire isc_done;
wire isc_valid;
intermediate_square_checker isc(clock_65mhz, isc_start,
flattened_chessboard,
	from_col, from_row, to_col, to_row, delta_col, delta_row,
isc_done, isc_valid);

reg started;	// indicates that checking has begun
reg waiting;	// indicates that move_checker is waiting for result
from intermediate_square_checker

always @(posedge clock_65mhz) begin
	if (reset)
		begin
			done <= 0;
			valid <= 0;
			started <= 0;
```

```
                        waiting <= 0;
                        isc_start <= 0;
                end
        else if (waiting)
                begin
                        if (isc_done)
                                begin
                                        done <= 1;
                                        valid <= isc_valid;
                                        started <= 0;
                                        waiting <= 0;
                                        isc_start <= 0;
                                end
                        else
                                begin
                                        done <= 0;
                                        valid <= 0;
                                        isc_start <= 0;
                                end
                end
        else if (started)
                begin
                        if ((chessboard[from_col][from_row][1] == color)
&&              // piece belongs to current player
                                !(to_col == from_col && to_row ==
from_row))              // from-square is not the same as to-square
                                begin
                                        case(piece)
                                                KING:
                                                        begin
                                                                if
((mag_delta_col == 1 && mag_delta_row == 0) ||

        (mag_delta_col == 0 && mag_delta_row == 1) ||

        (mag_delta_col == 1 && mag_delta_row == 1))

        begin
                                                                                valid <= (c

                                                                        end
                                                                else

        begin
                                                                                valid <= 0;
                                                                        end
                                                                done <= 1;
                                                                started <=
0;
                                                        end
                                                KNIGHT:
                                                        begin
                                                                if
((mag_delta_col == 2 && mag_delta_row == 1) ||

        (mag_delta_col == 1 && mag_delta_row == 2))
```

```
        begin
                                                                        valid <= (c

                                                                  end
                                                            else

        begin
                                                                        valid <= 0;
                                                                  end
                                                            done <= 1;
                                                            started <=
0;
                                                      end
                                                PAWN:
                                                      begin
                                                            if
((mag_delta_col == 0)                  // one square advance
                                                                  &&
((color == WHITE && delta_row == 1) ||
                                                                        (color == B

        begin
                                                                        valid <= ch
                                                                  end
                                                            else if
((mag_delta_col ==0)           // two square advance
                                                                         && ((color
                                                                               (c

        begin
                                                                        // check th
                                                                        valid <= ((

                                                                  end
                                                            else if
((mag_delta_col == 1)          // capture
                                                                         && ((color
                                                                               (c

        begin
                                                                        // check th
                                                                        // and that
                                                                        valid <= ((

                                                                  end
                                                            else

        begin
                                                                        valid <= 0;
                                                                  end
                                                            done <= 1;
                                                            started <=
0;
                                                      end
                                                QUEEN:
                                                      begin
```

```
                                                            if
(((mag_delta_col == 0 ^ mag_delta_row == 0) ||
(mag_delta_col == mag_delta_row))
                                                                   &&
((chessboard[to_col][to_row] == EMPTY) ^
                                                                         (color ^ c
      begin
                                                                        isc_start <
                                                                        waiting <=
                                                                  end
                                                            else
      begin
                                                                        done <= 1;
                                                                        valid <= 0;
                                                                        started <=
                                                                  end
                                                      end
                                                BISHOP:
                                                      begin
                                                            if
((mag_delta_col == mag_delta_row)
                                                                  &&
((chessboard[to_col][to_row] == EMPTY) ||
                                                                        (color ^ ch
      begin
                                                                        isc_start <
                                                                        waiting <=
                                                                  end
                                                            else
      begin
                                                                        done <= 1;
                                                                        valid <= 0;
                                                                        started <=
                                                                  end
                                                      end
                                                ROOK:
                                                      begin
                                                            if
((mag_delta_col == 0 ^ mag_delta_row == 0)
                                                                  &&
((chessboard[to_col][to_row] == EMPTY) ||
                                                                        (color ^ ch
      begin
                                                                        isc_start <
                                                                        waiting <=
                                                                  end
                                                            else
      begin
                                                                        done <= 1;
                                                                        valid <= 0;
```

```
                                                                        started <=
                                                                end
                                                end
                                        default:
                                                begin
                                                        done <= 1;
                                                        valid <=
0;
                                                        started <=
0;
                                                end
                                endcase
                        end
                else
                        begin
                                done <= 1;
                                valid <= 0;
                                started <= 0;
                        end
                end
        else if (start)
                begin
                        piece <=
chessboard[command[11:9]][command[8:6]][4:2];
                        from_col <= command[11:9];
                        from_row <= command[8:6];
                        to_col <= command[5:3];
                        to_row <= command[2:0];

                        done <= 0;
                        valid <= 0;
                        started <= 1;
                        waiting <= 0;
                        isc_start <= 0;
                end
        else
                begin
                        done <= 0;
                        valid <= 0;
                        started <= 0;
                        waiting <= 0;
                        isc_start <= 0;
                end
end

endmodule

// checks to see if there are any pieces along the line,
// striaght or diagonal, from the from-square to the
// to-square; does not to-square, as module was originally
// intended to do checking for castling as well
module intermediate_square_checker (
        input clock_65mhz,
        input start,
        input [64*5-1:0] flattened_chessboard,
        input [2:0] from_col,
        input [2:0] from_row,
```

```
        input [2:0] to_col,
        input [2:0] to_row,
        input signed [3:0] delta_col,
        input signed [3:0] delta_row,
        output reg done,
        output reg valid);

localparam [4:0] EMPTY = 5'b0;

// multidimensional array representation of chessboard
reg [4:0] chessboard [7:0][7:0];

// link flattened representation of chessboard to
// multidimensional array of chessboard
integer c;
integer r;
always @(*)
        begin
                for (c = 0; c < 8; c = c + 1)
                        begin
                                for (r = 0; r < 8; r = r + 1)
                                        begin
                                                chessboard[c][r] =
flattened_chessboard[((8*c+r+1)*5-1)-:5];
                                        end
                        end
        end

reg [2:0] col;
reg [2:0] row;
reg started;
always @(posedge clock_65mhz)
        begin
                if (started)
                        begin
                                if (col == to_col && row == to_row)
                                        begin
                                                done <= 1;
                                                valid <= 1;
                                                started <= 0;
                                        end
                                else if (chessboard[col][row] == EMPTY)
                                        begin
                                                col <= (delta_col < 0) ?
col - 1 : (delta_col == 0) ? col : col + 1;
                                                row <= (delta_row < 0) ?
row - 1 : (delta_row == 0) ? row : row + 1;
                                        end
                                else
                                        begin
                                                done <= 1;
                                                valid <= 0;
                                                started <= 0;
                                        end
                        end
                else if (start)
                        begin
```

```
                              done <= 0;
                              valid <= 0;
                              started <= 1;
                              col <= (delta_col < 0) ? from_col - 1 :
(delta_col == 0) ? from_col : from_col + 1;
                              row <= (delta_row < 0) ? from_row - 1 :
(delta_row == 0) ? from_row : from_row + 1;
                       end
               else
                       begin
                              done <= 0;
                              valid <= 0;
                              started <= 0;
                       end
        end
endmodule

`timescale 1ns / 100ps
module test();

localparam [2:0] KING = 3'd6;
localparam [2:0] QUEEN = 3'd5;
localparam [2:0] ROOK = 3'd4;
localparam [2:0] BISHOP = 3'd3;
localparam [2:0] KNIGHT = 3'd2;
localparam [2:0] PAWN = 3'd1;

localparam WHITE = 1'b1;
localparam BLACK = 1'b0;

localparam MOVED = 1'b1;
localparam UNMOVED = 1'b0;

localparam [4:0] EMPTY = 5'b0;

reg [4:0] chessboard [7:0][7:0];
reg [64*5-1:0] flattened_chessboard;

// link flattened representation of chessboard to
// multidimensional array of chessboard
integer c;
integer r;
always @(*)
        begin
               for (c = 0; c < 8; c = c + 1)
                       begin
                              for (r = 0; r < 8; r = r + 1)
                                     begin

        flattened_chessboard[((8*c+r+1)*5-1)-:5] = chessboard[c][r];
                                     end
                       end
        end

reg clk;
initial
        begin
```

```
		clk = 0;
		forever #7 clk = ~clk; // 14 ns period clock
	end

reg reset;
reg start;
reg color;
reg [11:0] command;
wire done;
wire valid;
move_checker mc(clk, reset, start, color, command,
flattened_chessboard, done, valid);


integer co;
integer ro;
initial
	begin
		$display("Setting up board with pieces...");
		for (co = 0; co < 8; co = co + 1)
			begin
			// place a pawn in every column in rows 1 and 6
			chessboard[co][1] <= {PAWN,WHITE,UNMOVED};
			chessboard[co][6] <= {PAWN,BLACK,UNMOVED};
				// empty every square between rows 2 and
6 (inclusive)
				for (ro = 2; ro < 6; ro = ro + 1)
					begin
						chessboard[co][ro] =
EMPTY;
					end
			end

		// set up main pieces
		chessboard[0][0] = {ROOK,WHITE,UNMOVED};
		chessboard[1][0] = {KNIGHT,WHITE,UNMOVED};
		chessboard[2][0] = {BISHOP,WHITE,UNMOVED};
		chessboard[3][0] = {QUEEN,WHITE,UNMOVED};
		chessboard[4][0] = {KING,WHITE,UNMOVED};
		chessboard[5][0] = {BISHOP,WHITE,UNMOVED};
		chessboard[6][0] = {KNIGHT,WHITE,UNMOVED};
		chessboard[7][0] = {ROOK,WHITE,UNMOVED};

		chessboard[0][7] = {ROOK,BLACK,UNMOVED};
		chessboard[1][7] = {KNIGHT,BLACK,UNMOVED};
		chessboard[2][7] = {BISHOP,BLACK,UNMOVED};
		chessboard[3][7] = {QUEEN,BLACK,UNMOVED};
		chessboard[4][7] = {KING,BLACK,UNMOVED};
		chessboard[5][7] = {BISHOP,BLACK,UNMOVED};
		chessboard[6][7] = {KNIGHT,BLACK,UNMOVED};
		chessboard[7][7] = {ROOK,BLACK,UNMOVED};

		$display("Finished setting up board...");

		#14
		command = {3'd2,3'd1,3'd2,3'd3};
		color = WHITE;
```

```
                start = 1;
                #14
                start = 0;
                $display("Pawn C2 to C4 is valid?");

                #14
                command = {3'd3,3'd7,3'd1,3'd5};
                color = BLACK;
                start = 1;
                #14
                start = 0;
                $display("Queen D8 to B6 is valid?");

                // #14
                // command = {000010000011};
                // color = BLACK;
                // start = 1;
                // #14
                // start = 0;
                // $display("No-piece A3 to A4 is valid?");
        end

always @(posedge clk)
        begin
                if (done)
                        $display("Valid: %b", valid);
        end

endmodule
```

Appendix L: Chessboard Drawer

```
module chessboard_drawer (
        input clock_65mhz,
        input [10:0] x,
        input [9:0] y,
        input [10:0] hcount,
        input [9:0] vcount,
        output reg [23:0] pixel);

localparam BOARD_WIDTH = 64*8;

localparam LIGHT = 24'hFF_CE_9E;
localparam DARK = 24'hD1_8B_47;

reg signed [11:0] norm_hcount;
reg signed [10:0] norm_vcount;

always @(posedge clock_65mhz)
        begin
                norm_hcount <= hcount - x;
                norm_vcount <= vcount - y;
        end

always @(posedge clock_65mhz)
        begin
                if ((norm_hcount >= 0 && norm_hcount < BOARD_WIDTH) &&
```

```
                        (norm_vcount >= 0 && norm_vcount < BOARD_WIDTH))
                        begin
                                if (norm_hcount[6] ^ norm_vcount[6])
                                        begin
                                                pixel <= LIGHT;
                                        end
                                else
                                        begin
                                                pixel <= DARK;
                                        end
                        end
                else
                        begin
                                pixel <= 24'h0;
                        end
        end

endmodule
```

Appendix M: Chess Pieces Drawer

```
module chess_pieces_drawer (
        input clock_65mhz,
        input [10:0] x,
        input [9:0] y,
        input [10:0] hcount,
        input [9:0] vcount,
        input [64*5-1:0] flattened_chessboard,
        input [16*3-1:0] flattened_white_captures,
        input [16*3-1:0] flattened_black_captures,
        output [23:0] pixel);

// chess parameters
localparam [2:0] KING = 3'd6;
localparam [2:0] QUEEN = 3'd5;
localparam [2:0] ROOK = 3'd4;
localparam [2:0] BISHOP = 3'd3;
localparam [2:0] KNIGHT = 3'd2;
localparam [2:0] PAWN = 3'd1;

localparam WHITE = 1'b1;
localparam BLACK = 1'b0;

// 8-bit pixel value for transparent color
localparam [7:0] TRANSPARENT = 8'h03;

// grid and board parameters
localparam BOARD_WIDTH = 64*8;
localparam GRID_WIDTH = 64*2;
localparam PAD_WIDTH = 64*1;
localparam HEIGHT = 64*8;
localparam BOARD_L_EDGE = GRID_WIDTH + PAD_WIDTH;
localparam BOARD_R_EDGE = BOARD_L_EDGE + BOARD_WIDTH;
localparam BLACK_L_EDGE = GRID_WIDTH + PAD_WIDTH + BOARD_WIDTH +
PAD_WIDTH;
localparam BLACK_R_EDGE = BLACK_L_EDGE + GRID_WIDTH;
```

```
// array representations of chessboard and pieces
// captured by white and black
reg [4:0] chessboard [7:0][7:0];
reg [2:0] white_captures [1:0][7:0];
reg [2:0] black_captures [1:0][7:0];

// convert from flat representations to array
// representations
integer c;
integer r;
integer i;
integer j;
always @(*)
	begin
		for (c = 0; c < 8; c = c + 1)
			begin
				for (r = 0; r < 8; r = r + 1)
					begin
						chessboard[c][r] =
flattened_chessboard[((8*c+r+1)*5-1)-:5];
					end
			end
		for (i = 0; i < 2; i = i + 1)
			begin
				for (j = 0; j < 8; j = j + 1)
					begin
						white_captures[i][j] =
flattened_white_captures[((i+2*j+1)*3-1)-:3];
						black_captures[i][j] =
flattened_black_captures[((i+2*j+1)*3-1)-:3];
					end
			end
	end


// normalized hcount and vcount and delayed copies
reg signed [11:0] norm_hcount;
reg signed [10:0] norm_vcount;
reg signed [11:0] norm_hcount_d1;
reg signed [10:0] norm_vcount_d1;
reg signed [11:0] norm_hcount_d2;
reg signed [10:0] norm_vcount_d2;

// calculate normalized hcount and vcount and rom address
reg [11:0] addr;
always @(posedge clock_65mhz)
	begin
		norm_hcount <= hcount - x;
		norm_vcount <= vcount - y;
		norm_hcount_d1 <= norm_hcount;
		norm_vcount_d1 <= norm_vcount;
		norm_hcount_d2 <= norm_hcount_d1;
		norm_vcount_d2 <= norm_vcount_d1;
		addr <= {norm_vcount[5:0],norm_hcount[5:0]};
	end
```

```
wire [7:0] white_king_out;
wire [7:0] white_queen_out;
wire [7:0] white_rook_out;
wire [7:0] white_bishop_out;
wire [7:0] white_knight_out;
wire [7:0] white_pawn_out;

wire [7:0] black_king_out;
wire [7:0] black_queen_out;
wire [7:0] black_rook_out;
wire [7:0] black_bishop_out;
wire [7:0] black_knight_out;
wire [7:0] black_pawn_out;

// instantiate roms for piece sprites
white_king wk(clock_65mhz, addr, white_king_out);
white_queen wq(clock_65mhz, addr, white_queen_out);
white_rook wr(clock_65mhz, addr, white_rook_out);
white_bishop wb(clock_65mhz, addr, white_bishop_out);
white_knight wkn(clock_65mhz, addr, white_knight_out);
white_pawn wp(clock_65mhz, addr, white_pawn_out);

black_king bk(clock_65mhz, addr, black_king_out);
black_queen bq(clock_65mhz, addr, black_queen_out);
black_rook br(clock_65mhz, addr, black_rook_out);
black_bishop bb(clock_65mhz, addr, black_bishop_out);
black_knight bkn(clock_65mhz, addr, black_knight_out);
black_pawn bp(clock_65mhz, addr, black_pawn_out);

// column and row of chessboard that corresponds with normalized hcount
and vcount
wire [2:0] col = norm_hcount_d2[8:6] - 3;
wire [2:0] row = 7 - norm_vcount_d2[8:6];

wire [2:0] white_col = norm_hcount_d2[8:6];
wire [2:0] black_col = norm_hcount_d2[8:6] - 4;
wire [2:0] grid_row = norm_vcount_d2[8:6];

// select which rom to get 8-bit pixel info from
reg [7:0] piece_pixel;
always @(posedge clock_65mhz)
        begin
                if ((norm_hcount_d2 >= 0 && norm_hcount_d2 < GRID_WIDTH)
&&
                        (norm_vcount_d2 >= 0 && norm_vcount_d2 < HEIGHT))
                        begin
                                case(white_captures[white_col][grid_row])
                                        KING:           piece_pixel <=
black_king_out;
                                        QUEEN:          piece_pixel <=
black_queen_out;
                                        ROOK:           piece_pixel <=
black_rook_out;
                                        BISHOP:         piece_pixel <=
black_bishop_out;
```

```
                                        KNIGHT:         piece_pixel <=
black_knight_out;
                                        PAWN:           piece_pixel <=
black_pawn_out;
                                        default:
        piece_pixel <= TRANSPARENT;
                                endcase
                        end
                else if ((norm_hcount_d2 >= BOARD_L_EDGE &&
norm_hcount_d2 < BOARD_R_EDGE) &&
                                 (norm_vcount_d2 >= 0 && norm_vcount_d2 <
HEIGHT))
                        begin
                                case(chessboard[col][row][4:1])
                                        {KING,WHITE}:   piece_pixel <=
white_king_out;
                                        {QUEEN,WHITE}:  piece_pixel <=
white_queen_out;
                                        {ROOK,WHITE}:   piece_pixel <=
white_rook_out;
                                        {BISHOP,WHITE}:piece_pixel <=
white_bishop_out;
                                        {KNIGHT,WHITE}:piece_pixel <=
white_knight_out;
                                        {PAWN,WHITE}:   piece_pixel <=
white_pawn_out;
                                        {KING,BLACK}:   piece_pixel <=
black_king_out;
                                        {QUEEN,BLACK}:  piece_pixel <=
black_queen_out;
                                        {ROOK,BLACK}:   piece_pixel <=
black_rook_out;
                                        {BISHOP,BLACK}:piece_pixel <=
black_bishop_out;
                                        {KNIGHT,BLACK}:piece_pixel <=
black_knight_out;
                                        {PAWN,BLACK}:   piece_pixel <=
black_pawn_out;
                                        default:
        piece_pixel <= TRANSPARENT;
                                endcase
                        end
                else if ((norm_hcount_d2 >= BLACK_L_EDGE &&
norm_hcount_d2 < BLACK_R_EDGE) &&
                                 (norm_vcount_d2 >= 0 && norm_vcount_d2 <
HEIGHT))
                        begin
                                case(black_captures[black_col][grid_row])
                                        KING:           piece_pixel <=
white_king_out;
                                        QUEEN:          piece_pixel <=
white_queen_out;
                                        ROOK:           piece_pixel <=
white_rook_out;
                                        BISHOP:         piece_pixel <=
white_bishop_out;
```

```
                                        KNIGHT:         piece_pixel <=
white_knight_out;
                                        PAWN:           piece_pixel <=
white_pawn_out;
                                        default:        piece_pixel <=
TRANSPARENT;
                                endcase
                        end
                else
                        begin
                                piece_pixel <= TRANSPARENT;
                        end
        end

// upconvert 8-bit pixel value to 24-bit value
assign pixel =
{piece_pixel[7:5],5'b0,piece_pixel[4:2],5'b0,piece_pixel[1:0],6'b0};

endmodule
```

Appendix N: Text Drawer

```
module text_drawer (
        input clock_65mhz,
        input [10:0] hcount,
        input [9:0] vcount,
        input [5*8-1:0] player_string,
        input [32*8-1:0] string_1,
        input [32*8-1:0] string_2,
        input [6*8-1:0] kb_string,
        output [23:0] pixel);

localparam CENTER_SCREEN_X = 512;

localparam CHAR_HEIGHT = 24;
localparam CHAR_WIDTH = 16;

localparam TOP_MARGIN = 32;
localparam BOTTOM_MARGIN = 64;
localparam PADDING = 8;

localparam SQUARE_SIZE = 64;
localparam BOARD_WIDTH = 64 * 8;

localparam LEFT_EDGE_BOARD = 64 * 4;
localparam RIGHT_EDGE_BOARD = LEFT_EDGE_BOARD + BOARD_WIDTH;
localparam TOP_EDGE_BOARD = TOP_MARGIN + CHAR_HEIGHT + PADDING +
CHAR_HEIGHT;
localparam BOTTOM_EDGE_BOARD = TOP_EDGE_BOARD + BOARD_WIDTH;

localparam CENTER_WHITE_GRID_X = 64 * 2;
localparam CENTER_BLACK_GRID_X = 64 * 14;

// text for headers
```

```
wire [2:0] hd_pixel;
reg [5*8-1:0] hd_string;
reg [10:0] hd_x;
reg [9:0] hd_y;
always @(posedge clock_65mhz)
        begin
                if (vcount < TOP_EDGE_BOARD - CHAR_HEIGHT - (PADDING /
2))
                        begin
                                hd_x <= CENTER_SCREEN_X - (CHAR_WIDTH *
2) - (CHAR_WIDTH / 2);
                                hd_y <= TOP_EDGE_BOARD - CHAR_HEIGHT -
PADDING - CHAR_HEIGHT;
                                hd_string <= player_string;
                        end
                else if (hcount < CENTER_SCREEN_X)
                        begin
                                hd_x <= CENTER_WHITE_GRID_X - (CHAR_WIDTH
* 2) - (CHAR_WIDTH / 2);
                                hd_y <= TOP_EDGE_BOARD - CHAR_HEIGHT;
                                hd_string <= "WHITE";
                        end
                else
                        begin
                                hd_x <= CENTER_BLACK_GRID_X - (CHAR_WIDTH
* 2) - (CHAR_WIDTH / 2);
                                hd_y <= TOP_EDGE_BOARD - CHAR_HEIGHT;
                                hd_string <= "BLACK";
                        end
        end
char_string_display hd(clock_65mhz, hcount, vcount,
                        hd_pixel, hd_string, hd_x, hd_y);
defparam hd.NCHAR = 5;
defparam hd.NCHAR_BITS = 3;

// text for col letters and row numbers
wire [2:0] col_row_pixel;
reg [7:0] col_row_string;
reg [10:0] col_row_x;
reg [9:0] col_row_y;
always @(posedge clock_65mhz)
        begin
                if (vcount < TOP_EDGE_BOARD || vcount >
BOTTOM_EDGE_BOARD)
                        begin
                                col_row_y <= (vcount < TOP_EDGE_BOARD) ?
(TOP_EDGE_BOARD - CHAR_HEIGHT) : BOTTOM_EDGE_BOARD;
                                if (hcount < LEFT_EDGE_BOARD +
SQUARE_SIZE)
                                        begin
                                                col_row_x <=
LEFT_EDGE_BOARD + (SQUARE_SIZE / 2) - (CHAR_WIDTH / 2);
                                                col_row_string <= "A";
                                        end
                                else if (hcount < LEFT_EDGE_BOARD +
(SQUARE_SIZE * 2))
                                        begin
```

```
                                        col_row_x <=
LEFT_EDGE_BOARD + (SQUARE_SIZE * 1) + (SQUARE_SIZE / 2) - (CHAR_WIDTH /
2);
                                        col_row_string <= "B";
                                end
                        else if (hcount < LEFT_EDGE_BOARD +
(SQUARE_SIZE * 3))
                                begin
                                        col_row_x <=
LEFT_EDGE_BOARD + (SQUARE_SIZE * 2) + (SQUARE_SIZE / 2) - (CHAR_WIDTH /
2);
                                        col_row_string <= "C";
                                end
                        else if (hcount < LEFT_EDGE_BOARD +
(SQUARE_SIZE * 4))
                                begin
                                        col_row_x <=
LEFT_EDGE_BOARD + (SQUARE_SIZE * 3) + (SQUARE_SIZE / 2) - (CHAR_WIDTH /
2);
                                        col_row_string <= "D";
                                end
                        else if (hcount < LEFT_EDGE_BOARD +
(SQUARE_SIZE * 5))
                                begin
                                        col_row_x <=
LEFT_EDGE_BOARD + (SQUARE_SIZE * 4) + (SQUARE_SIZE / 2) - (CHAR_WIDTH /
2);
                                        col_row_string <= "E";
                                end
                        else if (hcount < LEFT_EDGE_BOARD +
(SQUARE_SIZE * 6))
                                begin
                                        col_row_x <=
LEFT_EDGE_BOARD + (SQUARE_SIZE * 5) + (SQUARE_SIZE / 2) - (CHAR_WIDTH /
2);
                                        col_row_string <= "F";
                                end
                        else if (hcount < LEFT_EDGE_BOARD +
(SQUARE_SIZE * 7))
                                begin
                                        col_row_x <=
LEFT_EDGE_BOARD + (SQUARE_SIZE * 6) + (SQUARE_SIZE / 2) - (CHAR_WIDTH /
2);
                                        col_row_string <= "G";
                                end
                        else if (hcount < LEFT_EDGE_BOARD +
(SQUARE_SIZE * 8))
                                begin
                                        col_row_x <=
LEFT_EDGE_BOARD + (SQUARE_SIZE * 7) + (SQUARE_SIZE / 2) - (CHAR_WIDTH /
2);
                                        col_row_string <= "H";
                                end
                        else
                                begin
                                        col_row_x <= 0;
                                        col_row_string <= " ";
```

```
                                end
                        end
                else
                        begin
                                col_row_x <= (hcount < CENTER_SCREEN_X) ?
LEFT_EDGE_BOARD - CHAR_WIDTH : RIGHT_EDGE_BOARD;
                                if (vcount < TOP_EDGE_BOARD +
SQUARE_SIZE)
                                        begin
                                                col_row_y <=
TOP_EDGE_BOARD + (SQUARE_SIZE / 2) - (CHAR_HEIGHT / 2);
                                                col_row_string <= "8";
                                        end
                                else if (vcount < TOP_EDGE_BOARD +
(SQUARE_SIZE * 2))
                                        begin
                                                col_row_y <=
TOP_EDGE_BOARD + (SQUARE_SIZE * 1) + (SQUARE_SIZE / 2) - (CHAR_HEIGHT /
2);
                                                col_row_string <= "7";
                                        end
                                else if (vcount < TOP_EDGE_BOARD +
(SQUARE_SIZE * 3))
                                        begin
                                                col_row_y <=
TOP_EDGE_BOARD + (SQUARE_SIZE * 2) + (SQUARE_SIZE / 2) - (CHAR_HEIGHT /
2);
                                                col_row_string <= "6";
                                        end
                                else if (vcount < TOP_EDGE_BOARD +
(SQUARE_SIZE * 4))
                                        begin
                                                col_row_y <=
TOP_EDGE_BOARD + (SQUARE_SIZE * 3) + (SQUARE_SIZE / 2) - (CHAR_HEIGHT /
2);
                                                col_row_string <= "5";
                                        end
                                else if (vcount < TOP_EDGE_BOARD +
(SQUARE_SIZE * 5))
                                        begin
                                                col_row_y <=
TOP_EDGE_BOARD + (SQUARE_SIZE * 4) + (SQUARE_SIZE / 2) - (CHAR_HEIGHT /
2);
                                                col_row_string <= "4";
                                        end
                                else if (vcount < TOP_EDGE_BOARD +
(SQUARE_SIZE * 6))
                                        begin
                                                col_row_y <=
TOP_EDGE_BOARD + (SQUARE_SIZE * 5) + (SQUARE_SIZE / 2) - (CHAR_HEIGHT /
2);
                                                col_row_string <= "3";
                                        end
                                else if (vcount < TOP_EDGE_BOARD +
(SQUARE_SIZE * 7))
                                        begin
```

```
                                                col_row_y <=
TOP_EDGE_BOARD + (SQUARE_SIZE * 6) + (SQUARE_SIZE / 2) - (CHAR_HEIGHT /
2);
                                                col_row_string <= "2";
                                        end
                                else if (vcount < TOP_EDGE_BOARD +
(SQUARE_SIZE * 8))
                                        begin
                                                col_row_y <=
TOP_EDGE_BOARD + (SQUARE_SIZE * 7) + (SQUARE_SIZE / 2) - (CHAR_HEIGHT /
2);
                                                col_row_string <= "1";
                                        end
                                else
                                        begin
                                                col_row_y <= 0;
                                                col_row_string <= " ";
                                        end
                        end
        end
char_string_display col_row(clock_65mhz, hcount, vcount,
                        col_row_pixel, col_row_string, col_row_x,
                        col_row_y);
defparam col_row.NCHAR = 1;
defparam col_row.NCHAR_BITS = 1;

// text for body (from chess engine and keyboard input)
localparam TOP_BODY_TEXT = BOTTOM_EDGE_BOARD + CHAR_HEIGHT + PADDING;
localparam SPACE_24 = {"    ","    ","    ","    ","    ","    "};
wire [2:0] body_pixel;
reg [32*8-1:0] body_string;
reg [10:0] body_x;
reg [9:0] body_y;
always @(posedge clock_65mhz)
        begin
                body_x <= LEFT_EDGE_BOARD;
                if (vcount < TOP_BODY_TEXT + CHAR_HEIGHT)
                        begin
                                body_y <= TOP_BODY_TEXT;
                                body_string <= string_1;
                        end
                else if (vcount < TOP_BODY_TEXT + (CHAR_HEIGHT * 2))
                        begin
                                body_y <= TOP_BODY_TEXT + CHAR_HEIGHT;
                                body_string <= string_2;
                        end
                else
                        begin
                                body_y <= TOP_BODY_TEXT + (CHAR_HEIGHT *
2);
                                body_string <= {"> ", kb_string,
SPACE_24};
                        end

        end
char_string_display body(clock_65mhz, hcount, vcount,
                        body_pixel, body_string, body_x, body_y);
```

```
defparam body.NCHAR = 32;
defparam body.NCHAR_BITS = 5;

// combine pixel values
wire [2:0] pixel_3_bit = (hd_pixel | col_row_pixel | body_pixel);
// upconvert pixel values to 24 bits
assign pixel = {{8{pixel_3_bit[2]}},
                               {8{pixel_3_bit[1]}},
                               {8{pixel_3_bit[0]}}};
endmodule
```

Appendix O: Chess Graphics

```
module chess_graphics (
        input clock_65mhz,
        input [10:0] hcount,
        input [9:0] vcount,
        input hsync,
        input vsync,
        input blank,
        input [64*5-1:0] flattened_chessboard,
        input [16*3-1:0] flattened_white_captures,
        input [16*3-1:0] flattened_black_captures,
        input [5*8-1:0] player_string,
        input [32*8-1:0] string_1,
        input [32*8-1:0] string_2,
        input [6*8-1:0] kb_string,
        output reg chsync,
        output reg cvsync,
        output reg cblank,
        output [23:0] cpixel,
        input toggle);

// pixel values for "transparent" colors
localparam [7:0] X1 = 8'h03;
localparam [7:0] X2 = 8'h02;
localparam [23:0] TRANSPARENT_1 =
{X1[7:5],5'b0,X1[4:2],5'b0,X1[1:0],6'b0};
localparam [23:0] TRANSPARENT_2 =
{X2[7:5],5'b0,X2[4:2],5'b0,X2[1:0],6'b0};

// general layout
localparam TOP_MARGIN = 32;
localparam BOTTOM_MARGIN = 64;
localparam PADDING = 8;
localparam CHAR_HEIGHT = 24;

// board layout
localparam BOARD_WIDTH = 64 * 8;
localparam LEFT_EDGE_BOARD = 64 * 4;
localparam RIGHT_EDGE_BOARD = LEFT_EDGE_BOARD + BOARD_WIDTH;
localparam TOP_EDGE_BOARD = TOP_MARGIN + CHAR_HEIGHT + PADDING +
CHAR_HEIGHT;
localparam BOTTOM_EDGE_BOARD = TOP_EDGE_BOARD + BOARD_WIDTH;

// grids (of pieces that white and black have captured) layout
localparam TOP_EDGE_GRID = TOP_EDGE_BOARD;
localparam LEFT_EDGE_WHITE_GRID = 64;
localparam LEFT_EDGE_BLACK_GRID = 64 * 13;



// instantiate chessboard
wire [23:0] chessboard_pixel;
chessboard_drawer cbd(clock_65mhz, LEFT_EDGE_BOARD, TOP_EDGE_BOARD,
                       hcount, vcount, chessboard_pixel);

// delay chessboard pixel by 2 clock cycles
```

```
reg [23:0] chessboard_pixel_d1;
reg [23:0] chessboard_pixel_d2;
always @(clock_65mhz)
        begin
                chessboard_pixel_d1 <= chessboard_pixel;
                chessboard_pixel_d2 <= chessboard_pixel_d1;
        end


// instantiate chess pieces
wire [23:0] chess_pieces_pixel;
chess_pieces_drawer cpd(clock_65mhz, LEFT_EDGE_WHITE_GRID,
                        TOP_EDGE_BOARD, hcount, vcount,
flattened_chessboard,
                        flattened_white_captures,
flattened_black_captures,
                        chess_pieces_pixel);


// background for grid of pieces that white has captured
localparam LIGHT = 24'hFF_CE_9E;
wire [23:0] white_bg_pixel;
blob white_bg(LEFT_EDGE_WHITE_GRID, hcount, TOP_EDGE_GRID,
                        vcount, white_bg_pixel);
defparam white_bg.COLOR = LIGHT;

// background for grid of pieces that black has captured
localparam DARK = 24'hD1_8B_47;
wire [23:0] black_bg_pixel;
blob black_bg(LEFT_EDGE_BLACK_GRID, hcount, TOP_EDGE_GRID,
                        vcount, black_bg_pixel);
defparam black_bg.COLOR = DARK;


// draw text
wire [23:0] text_pixel;
text_drawer text(clock_65mhz, hcount, vcount, player_string,
                        string_1, string_2, kb_string, text_pixel);


// put everything together
reg [23:0] chess_pixel;
always @(*)
        begin
                if ((chess_pieces_pixel == TRANSPARENT_1) ||
                        (chess_pieces_pixel == TRANSPARENT_2))
                        begin
                                chess_pixel <= (chessboard_pixel_d2 |
text_pixel |

        black_bg_pixel | white_bg_pixel);
                        end
                else
                        begin
                                chess_pixel <= chess_pieces_pixel;
                        end
        end
```

```
assign cpixel = (toggle) ? chess_pieces_pixel : chess_pixel;


// delay hsync and vsync by 3 clock cycles to match
// delay of chess pieces graphics module
reg hsync_d1;
reg vsync_d1;
reg blank_d1;

reg hsync_d2;
reg vsync_d2;
reg blank_d2;

always @(posedge clock_65mhz)
        begin
                hsync_d1 <= hsync;
                vsync_d1 <= vsync;
                blank_d1 <= blank;

                hsync_d2 <= hsync_d1;
                vsync_d2 <= vsync_d1;
                blank_d2 <= blank_d1;

                chsync <= hsync_d2;
                cvsync <= vsync_d2;
                cblank <= blank_d2;
        end

endmodule

// blob module from lab5, modified to produce 24-bit pixel value
module blob
        #(parameter WIDTH = 128,              // default width: 128
pixels
                                HEIGHT = 64*8,                // default
height: 64*8 pixels
                                COLOR = 24'hFF_FF_FF)         // default
color: white
        (input [10:0] x, hcount,
         input [9:0] y, vcount,
     output reg [23:0] pixel);

        always @ (x or y or hcount or vcount)
                begin
                        if ((hcount >= x && hcount < (x+WIDTH)) &&
                                (vcount >= y && vcount < (y+HEIGHT)))
                                pixel = COLOR;
                        else
                                pixel = 0;
                end
endmodule
```

Appendix P: MATLAB JPG to COE

```
function img2 = IMG2coe8(imgfile, outfile)
% Create .coe file from .jpg image
% .coe file contains 8-bit words (bytes)
% each byte contains one 8-bit pixel
% color byte: [R2,R1,R0,G2,G1,G0,B1,B0]
% img2 = IMG2coe8(imgfile, outfile)
% img2 is 8-bit color image
% imgfile = input .jpg file
% outfile = output .coe file
% Example:
% img2 = IMG2coe8('loons240x160.jpg', 'loons240x160.coe');

img = imread(imgfile);
height = size(img, 1);
width = size(img, 2);
s = fopen(outfile,'wb');  %opens the output file
fprintf(s,'%s\n','; VGA Memory Map ');
fprintf(s,'%s\n','; .COE file with hex coefficients ');
fprintf(s,'; Height: %d, Width: %d\n\n', height, width);
fprintf(s,'%s\n','memory_initialization_radix=16;');
fprintf(s,'%s\n','memory_initialization_vector=');
cnt = 0;
img2 = img;
for r=1:height
   for c=1:width
       cnt = cnt + 1;
       R = img(r,c,1);
       G = img(r,c,2);
       B = img(r,c,3);
       Rb = dec2bin(R,8);
       Gb = dec2bin(G,8);
       Bb = dec2bin(B,8);
       img2(r,c,1) = bin2dec([Rb(1:3) '00000']);
       img2(r,c,2) = bin2dec([Gb(1:3) '00000']);
       img2(r,c,3) = bin2dec([Bb(1:2) '000000']);
       Outbyte = [ Rb(1:3) Gb(1:3) Bb(1:2) ];
       if (Outbyte(1:4) == '0000')
           fprintf(s,'0%X',bin2dec(Outbyte));
       else
           fprintf(s,'%X',bin2dec(Outbyte));
       end
       if ((c == width) && (r == height))
           fprintf(s,'%c',';');
       else
           if (mod(cnt,32) == 0)
               fprintf(s,'%c\n',',');
           else
                   fprintf(s,'%c',',');
           end
       end
   end
end
fclose(s);
```