Implementing a Sampling Synthesizing Keyboard on an FPGA

Damon Vander Lind, 2007
Mark Tobenkin, 2007

November 4th, 2005

Abstract

We have developed a sampling synthesizing piano keyboard, the Wumpus. This mechanism allows users to record, edit, and play back samples at higher or lower replay rates based on which electric piano key they hit. The attached MIDI keyboard allows users to play back samples at faster or slower rates (and thus frequencies), and to apply echo. Up to 8 "notes", or frequency shifted samples, may be played back simultaneously. Development was done in verilog, with a Xilinx Virtex II FPGA, set up on a prototyping board designed for MIT's introductory digital design class.

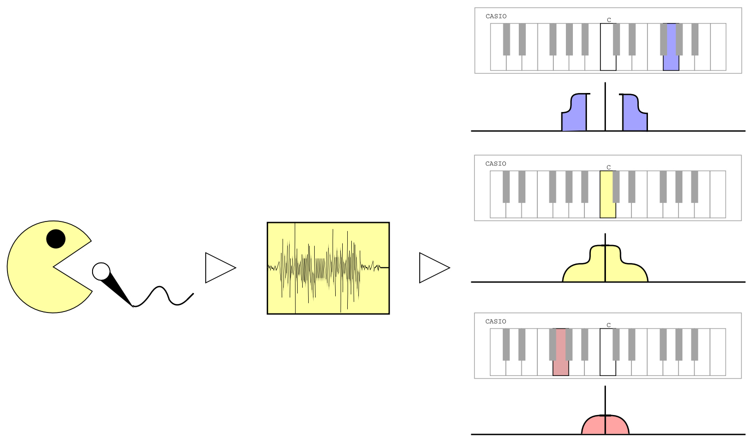


Figure 1: Diagram of Wumpus operation from the user perspective. A sample is recorded into memory, and then played back at different rates (and thus different frequencies) based on what key the user hits. Chords of up to 8 notes are possible.

1 Overview

The Wumpus, so named because of the preferred development test word, is a sampling synthesizing keyboard. Users may record in a sample, edit it's start, end, and hold positions, and play it back at faster or slower rates with one of two modes. In loop playback mode, the sample continually loops from the start position to the end position, allowing the user to adjust the above parameters. Alternately, the sample may be played back according to the frequencies and timing encoded in a MIDI keyboard's output, which is mapped to different rates of sample playback according to the usual correlation for musical notes. Thus, one can record in "Wumpus", and proceed to play simple piano pieces. Up to 8 notes can concurrently be playing, and notes pressed beyond this limit are ignored such that the system will not get backed up. When a note is held down, it cycles through a short (*window_hold_length* long) window of the sample at *window_hold_pos*, util the key is let up. A subsampled waveform is displayed on a VGA screen, with bars to show start, end, and hold positions, such that the user may better see where they are adjusting their values to. Inputs to the system are done primarily with greycode knobs for ease of use.

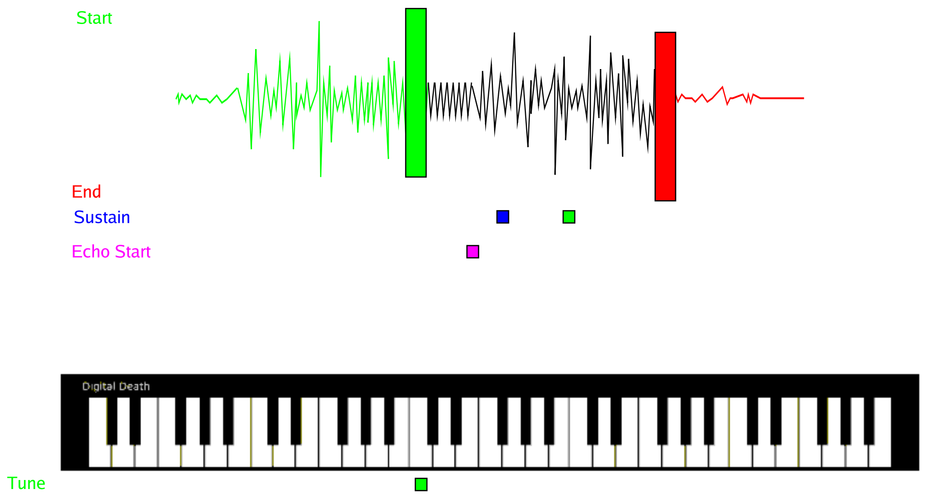


Figure 2: Mockup of the final user interface.

A user begins by pressing the enter button to capture their sample. This sample is painted on the screen as they record it. The user can flip switch zero to listen to their sample play back in a loop. Using the top most rotary encoder, the user can move the starting point of the sample. The green bar moves to indicate where on the sample the starting point is, and paints the unused portions of the sample green. The user can adjust the end position of the sample as well. The fourth rotary encoder down adjusts the large red bar on the screen, painting the unused end portions of the sample red. These parameters can also be adjusted in keyboard mode. When the loop switch is down, the user can play up to eight notes on the keyboard. Every full octave above Middle C the user presses will double again the speed of the playback. Every full octave below Middle C the user presses will halve the speed of the playback. Each note in between will vary the speed logarithmically between the octaves. The blue and green squares in the center of the waveform indicates the sustain point of the sample. When a key is pressed and held down, the point in the waveform between the blue and green squares will be repeated until the key is released. The second rotary encoder will move the position of both of these points, and the third encoder will change the distance between them. While playing in keyboard mode, the original sample initialy plays back at Middle C. By using the 6th encoder, the user can adjust the Tune parameter, and move the original sample to play back on any key on the keyboard.

2 Modules

The Wumpus consists of 5 top level modules: Control, Notes, Echo, Visualization, and Audio. We coded the first four, while the last was kindly provided by the 6.111 staff.

Figure 2 displays the topology of these modules. The design can be separated into two halves: the sound handling half, which includes *notes* and *echo* and the control half, which includes *visualizer* and *control*. Damon implemented the sound handling half while Mark implemented the control half and both worked on integration. The interface between the two consists of the echo controls, and of the parameters stored in a memory in *notes*, which *control* writes and sees return data from, to play and check allocation of notes respectively.

We should define three words before delving into module descriptions, for the sake of clarity. A **sample** is the entirety of a recording into the Wumpus. A **frame** is a single value of the sample at a given time, and a **window** is a small sequential subset of frames in the sample.

2.1 Notes

The *notes* module takes in audio data from *from_ac97_data* and outputs summed polyphonic and playback-rate-shifted audio data on *data_out*.Internally, this module consists of four submodules which work together to sequentially create and sum the desired frame for each individual note being output at a given time. Although the mechanism used to do this has very low overhead per added note, we decided to limit the number of notes to 8 to avoid degradation of the playback quality. The four submodules of the *notes* module are *taylor*, *notemem*, *frameprep*, and *accumulator*. The *notemem* module stores parameters (values which are constant throughout the playback of a single note) and variables (values which can change on any given frame) describing each note. The parametarized description of each of the 8 notes is fed in series to *frameprep*, which provides *taylor* with the current, previous, and next sample in relation to the note's current playback location. *taylor* uses these to do a taylor expansion of the sample and provide the desired frame, and feeds new variable values to notemem (signalling notemem to move onto the next note). *accumulator* sums the serially created frames for each note, and outputs this value on *data_out*. This module is fairly simple in terms of the number of lines of code, but proved to be very hard to debug due to the high level of interdependence between modules. All modules in *notes* signal off the previous module they depend on for input values. The way this is done, it creates a cycle which goes through all notes in the mimimum number of clock cycles given the delay of each module in the cycle.

This module construction differs significantly from the original plan, which involved ten copies of a single *note* module, which would all timeshare a top-level *sample* module for memory access. Parameterization of each note, and creation of *linemem*, described below, allowed for the much simpler, if somewhat less straightforward design used in the final implementation of the Wumpus.

2.1.1 Taylor

The *taylor* module takes, all in parallel, three concurrent frames, a *ready_frameprep* signal from *frameprep*, and all parameters and variables stored in *notemem*. At *ready_frameprep*, taylor uses the current and previous frames to linearly interpolate the value of a virtual frame *period* frames from the previous virtual frame for the playing note. *taylor* then determines the integer position, *n*, and fractional offset position, *delta*, from *period*, and returns these to *notemem* as it signals *taylor_done*. The *taylor_done* signal also

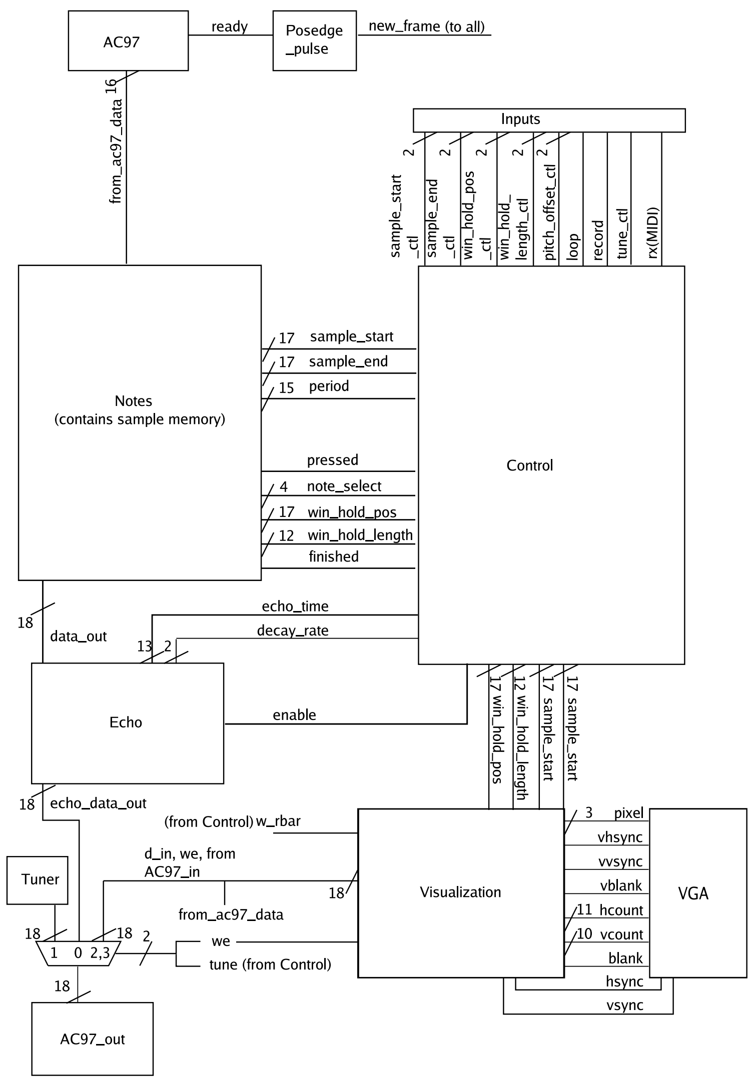


Figure 3: Main Block Diagram. The sample record and playback chain is displayed on the left, while the control system is pictured on the right. Samples are stored in *notes*, and played back according to the contents of a small memory in *notes* that is written by *control* and read by *notes*.

triggers *accumulator* to add the output *tframe_out* to it's running sum. A quadratic expansion was originally planned (and implemented), but it was found to have little or no effect on the sound of the original sample and therefore was removed.

In order to deal with playing ,pressed versus playing, nonpressed, and nonplaying, nonpressed notes, taylor runs a non-strict finite state machine with states for each combination of *going* and *pressed*. For non-playing, non-pressed keys, both are zero. For pressed, non-playing(and thus non-going) keys, *going* is set high. For pressed, playing notes, *going* is left high, and the sample plays until it reaches *win-*

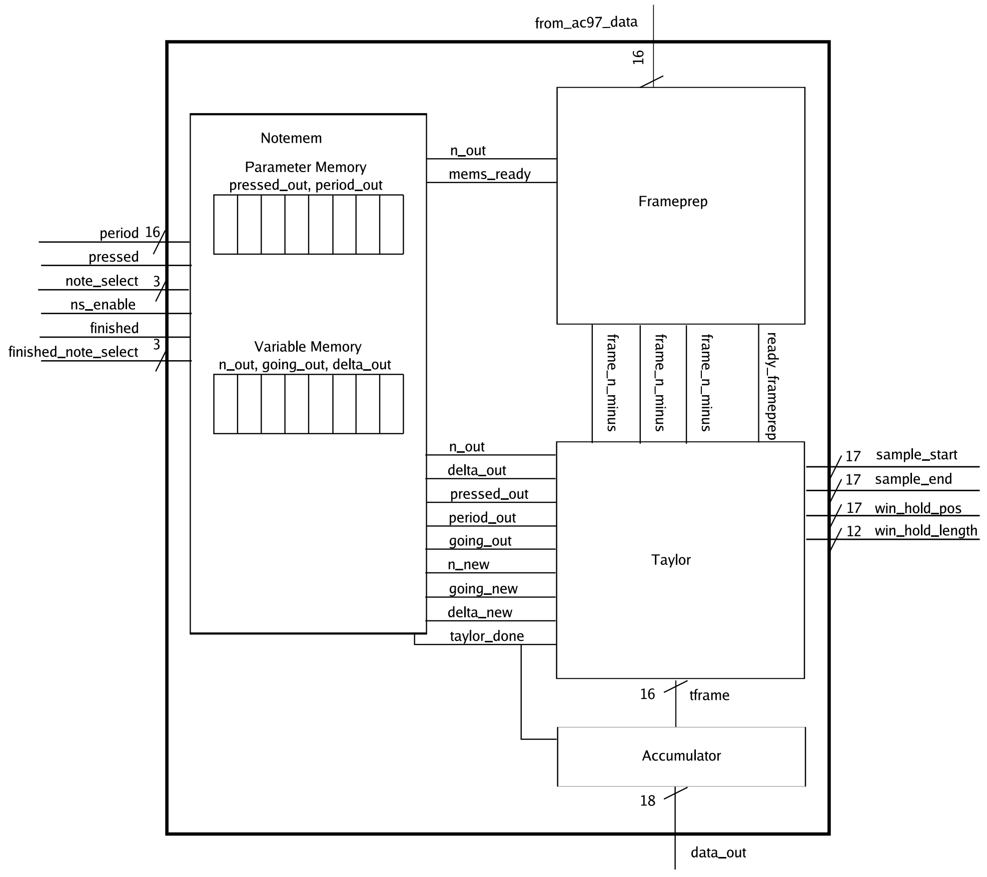


Figure 4: The *notes* module. This module consists of the *taylor*, *frameprep*, *notemem*, and *accumulator* modules. For every frame of audio data prepared by *notes*, *notemem*, *taylor*, and *frameprep* run serially through the notes to calculate their respective frames of audio data, and *accumulator* sums them together. At the signal of the next *new_frame*, this data is read by the next module in the chain before audio output.

dow_hold_pos, at which point it cycles over the previous *window_hold_length* samples. When the note is no longer pressed (regardless of wether it is cycling at *window_hold_pos*), *taylor* enters the final state, in which it plays until the end of the sample and resets all note variables to 0 upon reaching *sample_end*, including *going*. The signal of *going* low coming from *taylor* tells *notemem* to output a finished signal for the current note, telling *note_control* that a new note is free.

The *taylor* module not only performs the taylor expansion of the sample at the location in sample time of the desired frame, but also computes the new values of *going*, *n* and *delta*. *delta* is the offset from the current ram address, *n*, to where we want to create the current frame for this note.

2.1.2 Notemem

The *notemem* module stores the parameters and variables corresponding to each of 8 notes. These values are stored in two 8 long by 32 wide register based memories using the *line_mem* module. Notes are read out sequentially from both the variable and parameter memories concurently, with a new note being evaluated every time *taylor_done* signals it is done with the previous one, until all 8 have been played. Variables are written by *taylor* to the ram address of the current note, whenever *taylor* signals it is complete, and are read out to *frameprep* and *taylor*. Parameters are written by *note_control*, into whatever note it chooses. *note_control*, in turn, is signaled with notes that are finished.

2.1.3 Linemem

The *linemem* module implements an 8 long by 32 wide memory with one exclusive read port and one exclusive write port. The memory consists of a 256 bit long register and 32 bit wide output and input ports, each shifted by their respective addresses to select the right bits of the register. As read connections were made combinationally, the clock cycle delay normally present in BRAMS between asserting a read address and seeing data is not present. Since notemem simply implements a pass-through to it's linemem instantiations to read parameter and variable data, this design removed only about a shift register's delay from *notes*'s maximum clock speed (which is higher than that of *visualizer* and therefore unimportant.

2.1.4 Frameprep

The *frameprep* module contains the sample memory, which is 98304 samples long and 16 bits wide, due to BRAM memory constraints. On the *notemem_ready* signal, *frameprep* reads the frames at $n, n-1, n+1$ and stores them into *frame_n*, *frame_n_minus*, and *frame_n_plus* respectively. After the four clock cycles required for this to happen have passed, *ready_frameprep* is set high for a clock cycle to signal *taylor* to start it's calculations. For unpressed notes, *frameprep* still prepares sample frames. It is *taylor* that sets the output to 0 for these notes.

2.1.5 Accumulator

The *accumulator* module is quite simple. Upon the signal *taylor_done*, *accumulator* adds the current output from *taylor* to it's running sum. This running sum is cleared upon each *new_frame*, allowing a new polyphonic output frame to be built. Since there can be up to 8 notes playing concurrently, *accumulator* adds together all 8 16 bit signals, and shifts the output left one to make it 18 bit, which is the limit in accuracy afforded to us by the labkit.

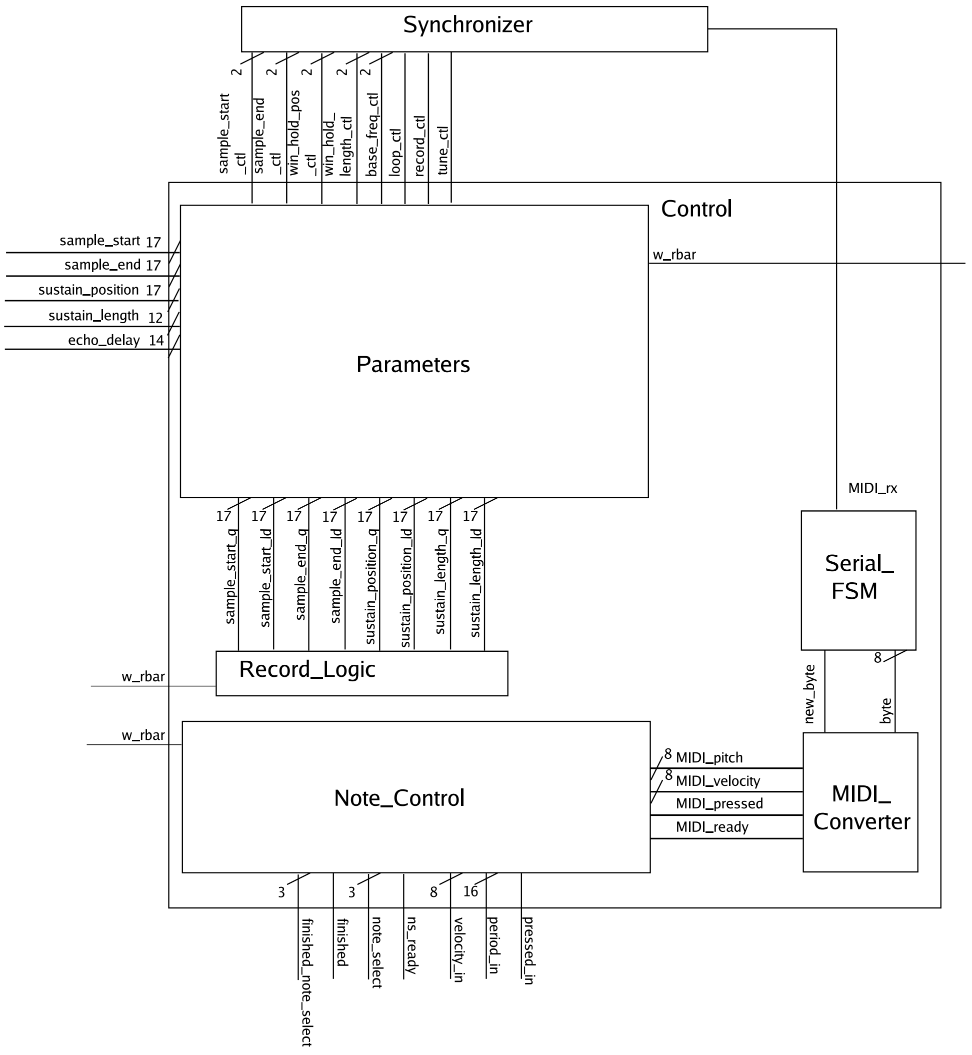
2.2 Echo

The *echo* module uses a BRAM to store the values of the previous *echo_time* frames from the output of *notes*, and adds the bit shifted least recent frame in it's memory to *data_in* to create *data_out*. This is done by way of a pointer to the BRAM. The BRAM is first read at the address of the pointer, and the output is shifted by *taylor_decay* and added to *data_in*.

The current value of *data_in* is then written into the BRAM at the pointer, and the pointer address is incremented. Length is adjusted by way of limiting the maximum pointer value possible before it is reset to 0. This function performs only a single echo, although recursive echo is quite easy to implement under this framework, involving only one or two lines of code. We found that the recursive echo sounds very cluttered and fuzzy when played back with multiple notes, and thus removed it from the final version.

The input data is buffered on new frame to avoid timing issues with the *sumframe_out* module in *notes*, as both as signalled on *new_frame* to start a new operation. This creates an extra one frame delay out the output, which has no effect on the keyboard playback experience.

Figure 5: The *control* module.

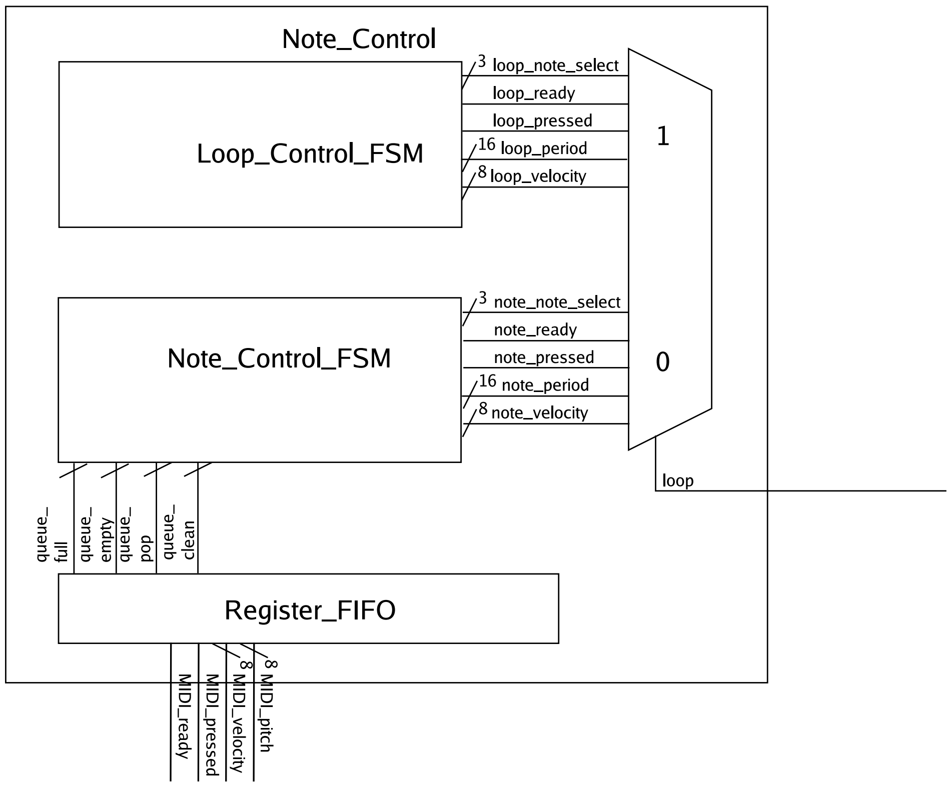


2.3 Control

2.3.1 Parameters

The *parameters* module controls the value of the *sample_start*, *sample_end*, *pitch_offset*, *echo_time*,*window_hold_pos*, and *window_hold_length* in response to user input. Each of these values is controlled by a rotary encoder knob. The six encoders' each provide a pair of control signals, which are decoded into increment/decrement requests by instances of the *grey_decode* module. Each of the controlled values is created by a *bounded_parameter* module, which bounds the output value, and changes it according to the corresponding increment and decrement requests. The boundaries of these signals are provided in Table 1.

Figure 6: The *note_control* module.



2.3.2 Grey Decode

The *grey_decode module* interprets two rotary encoder signals to produce increment and decrement requests. The increment and decrement requests are encoded on the wires. The *ready_pulse* output wire pulls high for one clock cycle when a new increment/decrement event occurs. A 1 on the *up_down* wire indicates an increment event, a 0 indicates a decrement event. Wires *a* and *b* carry the rotary encoder's control signals. When the encoders rotate, both lines *a* and *b* go low, with nearly a .01 second delay in between. Rotating clockwise causes line *a* to drop first, counter clockwise causes line *b* to drop first. A negative edge on either line triggers the *grey_decode* module to test if the other line is high.

2.3.3 Bounded Parameter

The *bounded_parameter* module provides an output bus of a parameterized size, keeps that value with variable upper and lower bounds, increments and decrements the value by a parameterized amount in response to requests, and on request loads a particular value into the bus. The bus *value* carries the output information. The buses *upper* and *lower* bound the value. When the wire *ld* is pulled high, the bus *q* is loaded into value, corrected for the current bounds. The parameter *SIZEOF* sets the width of the *upper*, *lower*, *value* and *q* buses. The parameter *INCREMENT* sets the step size of increment/decrement events. A one clock cycle pulse on the *incr* pin causes an increment/decrement event. A 1 on the *up_down* wire indicates an increment event, a 0 indicates a decrement event.

Table 1: Boundary Conditions for Parameters

Value	Upper	Lower
sample_start	*sample_end*	0x0
sample_end	0x1FFFF	*sample_start*
window_hold_pos	*sample_end-window_hold_length*	*sample_start*
window_hold_length	0xFFF	0x200
pitch_offset	0x3C	0x0
echo_time	0xFFF	0x0

Table 2: Values Loaded Into Parameters after Record

Bus	Value
sample_start	0x0
sample_end	*sample_length*
window_hold_pos	*sample_start*
window_hold_length	0x200
pitch_offset	0x18
echo_time	0xFFF

2.3.4 Record Logic

When a record event occurs, *control* module contains logic for setting new values in the parameters. An instance of the *level_to_pulse* module creates a pulse on the falling edge of the *record* signal. This triggers new values to load into *sample_start*, *sample_end*, *pitch_offset*, *echo_time*,*window_hold_pos*, and *window_hold_length*. The buses *sample_start_q*, *sample_end_q*, *pitch_offset_q*, *echo_time_q*,*window_hold_pos_q*, and *window_hold_length_q* are set to the default values, which are documented in Table 2. The record logic pulls the *sample_start_ld*, *sample_end_ld*, *pitch_offset_ld*, *echo_time_ld*,*window_hold_pos_ld*, and *window_hold_length_ld* lines high for two clock cycles. This delay allow for parameters whose boundaries rely on other parameters to settle to legal values.

2.3.5 Serial FSM/Sequencer

The *serial_fsm* module outputs bytes from an asyncronous serial input. The module expects a single start bit value of zero, followed by 8 bits and a stop bit. The baud rate of the incoming signal should be set in the parameter *BAUD_RATE*, and its width in bits in *BAUD_RATE_SIZEOF*. The clock rate of the system must also be set in parameters *CLOCK_RATE* and *CLOCK_RATE_SIZEOF*. These values are used to calculate delays in between reading bits from the input line *rx*. When the FSM/Sequencer has completed reading a byte, the *ready_pulse* line pulls high for a single clock cycle. The *byte* bus contains the new byte.

2.3.6 Delay

The delay module provides a single clock cycle pulse after one of two numbers of clock cycles. The pulse wire provides the output pulse after either parameter DELAY1_COUNT or parameter DELAY2_COUNT number of clock cycles. If the short signal is high, DELAY1_COUNT is used, else DELAY2_COUNT is used. The pulse occurs only once, and the module must be reset to create another pulse. The parameter COUNT_SIZEOF must specify the greater of the two bit widths between DELAY1_COUNT and DELAY2_COUNT.

2.3.7 MIDI FSM

The *midi_fsm* module interprets incoming MIDI messages for Note On/Off events, and outputs the pitch and velocity and pressed/depressed status of those events. The FSM receives bytes from the MIDI interface serially via the *byte* line. When a new byte is ready the *new_byte* line should be driven high for a single clock cycle. The FSM will ignore all MIDI events other than those which begin with the Note On (0x9) or Note Off (0x8) nibble. The *pitch* and *velocity* bytes for these events follow from the serial line. The *pitch* byte indicates which key on the keyboard was pressed, middle C being key 0x3C. The *velocity* byte indicates how hard the key was pressed, with a "normal" key press being placed at 0x40. There are other MIDI events which consist of multi byte frames similar to this. The MIDI FSM correctly ignores the data bytes of unsupported MIDI events. The MIDI module also generates a *pressed* signal. A Note Off event or Note On event with *velocity* zero will drive this line low, the line is driven high by a Note On event with non-zero velocity. When a midi event has been processed, the FSM generates a single clock cycle pulse on *ready*.

2.4 Note Control

The note control module manages note pressed and released events going to the Notes module for both the keyboard playback and loop modes. The module exports the *period*, *velocity* and *pressed* status of a note identified by the *note_select* bus to the *notes* module. The module drives the *note_ready* line high when the data is to be written. The *finished_note_select* line allows the *notes* module to identify a note, and then indicate its availability to play new requests with a high or low signal on the *finished* line. Note control contains a FIFO for handling incoming MIDI keyboard events. The FIFO is a Xilinx IP Core Syncronous FIFO with a width of 17 (*pitch*,*velocity* and *pressed*) and a depth of 32. The *note_control_fsm* handles this FIFO to generate a *keyboard_pitch*, *keyboard_velocity*, *keyboard_pressed* and *keyboard_ready* signal for MIDI events. A small BRAM is used to provide a lookup table between the MIDI keyboard values and the periods which *notes* expects. The *keyboard_pitch* combined with the *pitch_offset* input is an index into this rom, generating *keyboard_period*. This FSM is also passed the *finished_note_select* and finished wires to keep track of which notes are free. The FSM uses this information to correctly select notes with the *keyboard_note_select* and *keyboard_note_select* signals. A *loop_control_fsm* exists to control the outputs when in loop mode. The module generates a parallel set of control signals, named *loop_pressed*, *loop_note_select* and *loop_note_ready*. The *loop_period* and *loop_velocity* are fixed at "Middle C" and the "normal" velocity for MIDI of 0x40. The output of the *note_control_fsm* and the *loop_control_fsm* are multiplexed by the input signal *loop*.

2.4.1 Note Control FSM

The *note_control_fsm* creates appropriate control signals destined for *notes* based on events arriving at an external queue. When the external queue indicates it is no longer empty by pulling the *queue_empty* line low, the FSM/Sequencer signals a pop by pulling *queue_pop* high, then a clock cycle later latches the output of the queue to the output lines *period*, *velocity* and *pressed*. If a key was just pressed, the FSM enters a small sequencer to test if there are any notes available. The status of note availability is being constantly updated into the *finished_reg* array. The input *finished_note_select* indexes into this array, whose value is set to the input finished every clock cycle. The sequencer increments *note_select* as another index into *finished_reg*. If no finished notes are found, the FSM returns to testing for a non-empty queue. If a note is found, the FSM stores the new *period* the note will be playing into the array *note_periods*, then pulls *note_ready* high. After this the FSM enters a sequencer to wait a configurable number of *new_frame* pulses before testing the queue again. If a key released event arrives, the FSM follows an inverse process. The FSM enters a small sequencer to see if the *period* of the released key matches any entry in *note_period*, which is also not finished in *finished_reg*. If no note is found, the queue returns to testing for non-empty queues. If a note is found, *pressed* is again set appropriately, *ready* pulled high, and a sequencer is used to wait a certain number of *new_frame* pulses. Upon returning to testing for a non-empty queue, the FSM pulls *ready* low.

2.4.2 Loop Control FSM

The *loop_control_fsm* generates a press and release signal in short succession, destined for the *notes* module. The *loop_control_fsm* watches particularly for the note identified by *finished_note_select* equalling zero (the zeroth note) to be finished. When *finished* is high, and *finished_note_select* is zero, the *loop_control_fsm* generates a *pressed* signal, waits for a *new_frame* pulse and then pulls *pressed* low and *ready* high.

2.5 Visualizer

The visualizer module provides a 800x600, 60Hz stream of 3 bit values to *pixel*. The *hcount* and *vcount* buses indicate the position of the pixel stream on the screen currently. The graphic displays a space on the screen limited to 512 pixels wide and 128 pixels tall. The visualizer module combines a series of different pixel streams created by submodules to create its output. The submodules *start_text*, *sustain_text*, *end_text*, *echo_text* and *tune_text* each generate an ASCII string using the provided *char_string_display* module. The submodules *start_bar*, *end_bar*, *sustain_position_bar*, *sustain_length_bar*, *echo_bar*, and *tune_bar* each generate rectangles using the *visualizer_rectangle* module. The *waveform_g* element generates a waveform representation using the *waveform_graphic* module. The *my_waveform_background* element generates a multicolor rectangle using the *waveform_background* module. The *keyboard* element generates the image of a keyboard using the *keyboard_bitmap* module. Finally the pixel streams of all of these modules are bitwise or'd together and output to *pixel*.

2.5.1 Visualizer Rectangle

The *visualizer_rectangle* generates a pixel stream for a rectangle of a parameterized *COLOR*, *HEIGHT* and *WIDTH*, at a location specified by the busses *x* and *y*. The *hcount* and *vcount* buses indicate the position of the pixel stream on the screen currently.

2.5.2 Keyboard Bitmap

The *keyboard_bitmap* module reads an image of a keyboard, stored in a BRAM, based on positions encoded by *hcount* and *vcount*. The graphic displays a space on the screen limited to 512 pixels wide and 128 pixels tall. The memory is a single bit wide (black and white) BRAM with 56000 locations. This size is the product of the *HEIGHT* and *WIDTH* parameters. The *keyboard_bitmap* module resets the address into the memory when *hcount* and *vcount* arrive at the values in *x* and *y*. The *keyboard_bitmap* outputs a zero on the *pixel* bus whenever the the *hcount* and *vcount* indicate a point outside of the box designated by *x*,*y* and the parameter *HEIGHT* and *WIDTH*. Elsewise, the module outputs the value of the memory replicated three times on *pixel*, and increments the address.

2.5.3 Waveform Graphic

The *waveform_graphic* module provides a pixel stream of a visualization of the sample which is recorded. The *hcount* and *vcount* buses indicate the position of the pixel stream on the screen currently. The graphic displays a space on the screen limited to 512 pixels wide and 128 pixels tall. However, it has to display up to 98304 frames which are 16 bits wide. To accomplish this, the module has an FSM/sequencer for averaging frames, and storing them in a small memory. When the *record* signal goes high, the FSM begins averaging frames. A separate sum is calculated for the positive and negative values of 256 sequential frames. The 7 highest order bits of these sums are concatenated and then stored into memory. The memory is 512 deep an 14 bits wide. When *record* is released, the FSM sits in the WAIT state, which enables the output to pixel. When outside of the pixel locations specified by the *HEIGHT*, *WIDTH-sample_length_x*, *X_POSITION* and *Y_POSITION*, the module outputs zero to *pixel*. The address into memory is calculated based on the *X_POSITION* on screen. While inside the boundaries of these positions, the module creates a signed number ranging from (*HEIGHT*/2)-1 to -*HEIGHT*/2. For the top half of the rectangle, the *pixel* is output white if it is less than the signed positive average read from memory. For the bottom half of the rectangle, the *pixel* is output white if it is less than the negative average read from memory.

2.5.4 Waveform Background

The *visualizer_rectangle* generates a pixel stream for a rectangle of a parameterized *HEIGHT* and *WIDTH*, and position *X* and *Y*. The *hcount* and *vcount* buses indicate the position of the pixel stream on the screen currently. The color of the outputted pixel depends on the inputs *sample_start* and *sample_end*. If the current pixel is less than *X* plus *sample_start_x*, the output is green. If the current pixel is greater than *X* plus *sample_end_x* the output is red. Elsewise the output is white.

3 Physical Implementation

Beyond the provided Labkit, implementation of the Wumpus involved an electric piano keyboard, an optocoupler to isolate it's serial MIDI output, and a set of greycode knobs. The provided LCD screen was used to run VGA output for the visualizer module.

4 Debugging

The large amount of interdependence within the code made debugging more diffucult than previous labs, and aided in creating a large number of hard to track connections. The single largest problem encountered in the debugging process was misnamed wires, which were hard to detect due to complicated wiring and the poor debugging capabilities of the Xilinx software. The good part of a day was also lost due to the Xilinx project manager crashing. Perhaps with better compiler debugging outputs and more stable, reliable software, projects such as this could be done in half the time. Problems unrelated to debugging mistakes were nearly always related to a naming problem, such as the overlooking of a test wire when combining modules. Problems with asynchronosity were encountered in *note_control*, and were fixed by synchronizing the code.

In the case of the *notes* module, the interdependancy of all modules on one another as well as the intricate timing involved led to about a week of debugging before any audible output was heard, at which point nearly the entire module worked. This aspect of the module also made computer simulation of the verilog diffucult to perform accurately, and oversights in setting up the test waveforms was the cause of a perceived bug just as often as errors in verilog.

5 Conclusions

The design and implementation of the Wumpus demonstrated both the ease with which complicated systems can be designed in verilog, and the diffuculty of debugging simple naming problems or dealing with the ambiguity allow by verilog. This leads us to suggest that a more constrained language with less room for interpretation would lead to more efficient programming.

And finally, we must recoginize the fact that the more time one spends in lab, the more they smell like a monkey.

The following files have been reformatted to fit the page.

Listing 1: Verilog file accumulator.v

```
module accumulator(clock, reset, new_frame, taylor_done, tframe_out, sumframe_out, sum_done);
    input clock;
    input reset;
         input new_frame;      //signals the accumulator to start a new accumulation
         input taylor_done;    //signals that there is new data to be added to the accumulator
         input signed [15:0] tframe_out;    //data coming from taylor
         output signed [17:0] sumframe_out;  //accumulated frame. two bits wider than tframe_out,
                                        //meaning that we still have to shift one bit to fit 8
                            //notes. This still gives slightly higher accuracy during the addition.
         output sum_done;

         reg signed [19:0] running_sum; //unshifted accumulated frame

assign sumframe_out = running_sum[19:2];

always @ (posedge clock) begin
        if(reset|new_frame) begin
                running_sum <= 0;
        end
        else begin
                if(taylor_done) begin
                        running_sum<=running_sum + tframe_out;
                end
        end
end

endmodule
```

Listing 2: Verilog file audio.v

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

module audio (clock, reset, audio_in_data, audio_out_data, ready, new_frame,
              audio_reset_b, ac97_sdata_out, ac97_sdata_in,
              ac97_synch, ac97_bit_clock);

   input clock;
   input reset;
   output [17:0] audio_in_data;
   input [17:0] audio_out_data;
   output ready;
        output new_frame;

   //ac97 interface signals
   output audio_reset_b;
   output ac97_sdata_out;
   input ac97_sdata_in;
   output ac97_synch;
   input ac97_bit_clock;

   wire [4:0] volume;
   wire source;
   assign volume = 4'd44;//4'd22;   //a reasonable volume value
   assign source = 1;        //mic

   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;

   reg audio_reset_b;
   reg [10:0] reset_count;

   //wait a little before enabling the AC97 codec
   always @(posedge clock) begin
      if (reset) begin
         audio_reset_b = 1'b0;
         reset_count = 0;
      end else if (reset_count == 2047)
        audio_reset_b = 1'b1;
      else
        reset_count = reset_count+1;
   end

   ac97 ac97(xready, command_address, command_data, command_valid,
             left_out_data, 1'b1, right_out_data, 1'b1, left_in_data,
             right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,
             ac97_bit_clock);

   synchronize syncready(.clk(clock),.in(xready),.out(ready));
        level_to_pulse new_framemaker(.clock(clock), .reset(reset), .level(ready),.pulse(new_frame));

        ac97commands cmds(clock, ready, command_address, command_data,
                     command_valid, volume, source);


   assign left_out_data = {audio_out_data, 4'b00};
   assign right_out_data = left_out_data;

   //arbitrarily choose left input, get highest-order bits
   assign audio_in_data = left_in_data[19:2];

endmodule

// assemble/disassemble AC97 serial frames
module ac97 (ready,
             command_address, command_data, command_valid,
             left_data, left_valid,
             right_data, right_valid,
             left_in_data, right_in_data,
             ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

   output ready;
   input [7:0] command_address;
   input [15:0] command_data;
   input command_valid;
   input [19:0] left_data, right_data;
   input left_valid, right_valid;
   output [19:0] left_in_data, right_in_data;

   input ac97_sdata_in;
   input ac97_bit_clock;
   output ac97_sdata_out;
   output ac97_synch;

   reg ready;

   reg ac97_sdata_out;
   reg ac97_synch;

   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;
   reg [19:0] left_in_data, right_in_data;

   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 (clock, ready, command_address, command_data,
                     command_valid, volume, source);

   input clock;
   input ready;
   output [7:0] command_address;
   output [15:0] command_data;
   output command_valid;
   input [4:0] volume;
   input source;

   reg [23:0] command;
   reg command_valid;

   reg old_ready;
   reg done;
   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";
      done <= 1'b0;
      // synthesis attribute init of done is "0";
      old_ready <= 1'b0;
      // synthesis attribute init of old_ready 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;

   always @(posedge clock) begin
      if (ready && (!old_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 <= 24'h1A_0000; // microphone
        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)

      old_ready <= ready;

   end // always @ (posedge clock)

endmodule // ac97commands
```

Listing 3: Verilog file bounded_parameter.v

```
/**
 * bounded_parameter - Maintains a value with in particular boundaries, with parameterized steps.
 * @author mmt
 *
 * @parameter SIZEOF Number of bits for the value, its boundaries and force load.
 *
 * @input clock System clock.
 * @input reset Syncronous reset.
 * @input [SIZEOF-1:0] upper The upper bound of the value.
 * @input [SIZEOF-1:0] lower The lower bound of the value.
 * @input [SIZEOF-1:0] q     A bus which can be forceable loaded into the value.
 * @input incr A one clock cycle pulse indicating an increment/decrement event.
 * @input up_down High value indicates increment event, low value indicates decrement event.
 *
 * @output [SIZEOF-1:0] value The output of the parameter.
 */
module bounded_parameter(clock, reset, upper, lower, q, up_down, incr, ld, value);
        // the size in bits of the parameter
        parameter SIZEOF = 14;
        // the value to add/subtract on increment/decrement
        parameter INCREMENT = 1;

        //system clock
        input clock;
        //reset the module
        input reset;
        //upper bound on value
        input [SIZEOF-1:0] upper;
        //lower bound on value
        input [SIZEOF-1:0] lower;
        //force the load of q
        input ld;
        //value to force load
        input [SIZEOF-1:0] q;
        //indicates increment direction (+INCR or -INCR)
        input up_down;
        //single clock cycle increment signal
        input incr;
        //value begin maintained
        output [SIZEOF-1:0] value;

        reg [SIZEOF-1:0] value;


        always @ (posedge clock) begin
                if(reset) begin
                        value <= 0;
                end
                else begin
                        if(ld) begin
                                //boundary checks on load
                                if(q < lower)
                                        value <= lower;
                                else if(q > upper)
                                        value <= upper;
                                else
                                        value <= q;
                        end
                        else if(value < lower)
                                value <= lower;
                        else if(value > upper)
                                value <= upper;
                        else if (incr) begin
                                //boundary checks on increment
                                if(up_down)
                                        value <= (value + INCREMENT > upper ) ? upper : value + INCREMENT;
                                        else
                                                value <= (value < lower + INCREMENT) ? lower : value - INCREMENT;
                                end
                        end
                end
endmodule
```

Listing 4: Verilog file control.v

```
/**
 * control - Main module for controlling note events for keyboard and loop mode and maintaining parameters.
 *
 * @input clock System clock.
 * @input new_frame Pulse to indicate a new audio sample available.
 * @input [16:0] sample_length The length of the recording.
 * @input record A high indicates the system is currently recording.
 * @input midi_rx A midi serial line input (normally high)..
 * @input [1:0] sample_start_ctl Rotary encoder knob input for sample_start.
 * @input [1:0] sample_end_ctl Rotary encoder knob input for sample_end.
 * @input [1:0] sustain_position_ctl Rotary encoder knob input for sustain_position.
 * @input [1:0] sustain_length_ctl Rotary encoder knob input for sustain_length.
 * @input [1:0] pitch_offset_ctl Rotary encoder knob input for pitch_offset.
 * @input [1:0] echo_delay_ctl Rotary encoder knob input for echo_delay.
 * @input loop Indicates the system is in loop mode.
 * @input [16:0] sample_start
 * @input [16:0] sample_end
 * @input [16:0] sustain_position
 * @input [11:0] sustain_length
 * @input [13:0] echo_delay
 */
module control(clock, reset,
                new_frame,
                sample_length,
                record,
                midi_rx,
                sample_start_ctl, sample_end_ctl, sustain_position_ctl, sustain_length_ctl, echo_delay_ctl, pitch_offset_ctl,
                loop,
                sample_start, sample_end, sustain_position, sustain_length, echo_delay, pitch_offset,
                //sample_start_ld, sample_end_ld, sustain_position_ld, sustain_length_ld,
                //sample_start_q, sample_end_q, sustain_position_q, sustain_length_q,
                finished, finished_note_select,
                note_ready, note_select, period, velocity, pressed,
                //Debug outputs
                serial_byte, serial_ready);
        output serial_ready;
        output [7:0] serial_byte;

        input clock, reset;

        input new_frame;
        input [16:0] sample_length;


        input record;


        input [1:0] sample_start_ctl, sample_end_ctl, sustain_position_ctl, sustain_length_ctl, echo_delay_ctl, pitch_offset_ctl;

        output [16:0] sample_start, sample_end, sustain_position; //Actual values of parameters
        output [11:0] sustain_length;
        output [13:0] echo_delay;
        output [5:0]  pitch_offset;

        input [2:0] finished_note_select;
        input finished;
        output [2:0] note_select;
        output [15:0] period;
        output [7:0] velocity;
        output pressed;
        output note_ready;

        input midi_rx;
        input loop;

        reg sample_start_ld, sample_end_ld, sustain_position_ld, sustain_length_ld, echo_delay_ld, pitch_offset_ld; //Force load signals
        reg [16:0] sample_start_q, sample_end_q, sustain_position_q; //Data for forced load
        reg [11:0] sustain_length_q;
        reg [13:0] echo_delay_q;
        reg [5:0]  pitch_offset_q;


        parameters my_parameters(.clock(clock),.reset(reset),
                        .sample_length(sample_length),
                        .sample_start_ctl(sample_start_ctl),.sample_start_ld(stample_start_ld),
                        .sample_start_q(sample_start_q),.sample_start(sample_start),
                        .sample_end_ctl(sample_end_ctl),.sample_end_ld(sample_end_ld),
                        .sample_end_q(sample_end_q),.sample_end(sample_end),
                        .sustain_position_ctl(sustain_position_ctl),.sustain_position_ld(sustain_position_ld),
                        .sustain_position_q(sustain_position_q),.sustain_position(sustain_position),
                        .sustain_length_ctl(sustain_length_ctl),.sustain_length_ld(sustain_length_ld),
                        .sustain_length_q(sustain_length_q),.sustain_length(sustain_length),
                        .echo_delay_ctl(echo_delay_ctl),.echo_delay_ld(echo_delay_ld),
                        .echo_delay_q(echo_delay_q),.echo_delay(echo_delay),
                        .pitch_offset_ctl(pitch_offset_ctl),.pitch_offset_ld(pitch_offset_ld),
                        .pitch_offset_q(pitch_offset_q),.pitch_offset(pitch_offset));
        ///
        /// NOTE CONTROL
        ///



        //clock out serial information
        wire [7:0] serial_byte;
        wire serial_ready;

        serial_fsm my_serial_fsm(.clock(clock),.reset(reset),
                                .rx(midi_rx),.byte(serial_byte),.ready_pulse(serial_ready));


        //generate the midi commands
        wire [7:0] midi_pitch, midi_velocity;
        wire midi_pressed;
        wire midi_ready;

        midi_fsm my_midi_fsm(.clock(clock),.reset(reset),
                                .byte(serial_byte),.new_byte(serial_ready),
                                .pitch(midi_pitch),.velocity(midi_velocity),.pressed(midi_pressed),
                                .ready(midi_ready));

        //check for loop mode, which overrides the keyboard input


        //control the notes
        note_control my_note_control(.clock(clock),.reset(reset),
                        .pitch_offset(pitch_offset),
                        .finished_note_select(finished_note_select),.finished(finished),
                        .note_select(note_select),.note_ready(note_ready),
                        .pressed(pressed),.period(period),.velocity(velocity),.loop(loop),
                        .midi_velocity(midi_velocity),.midi_pitch(midi_pitch),
                        .midi_pressed(midi_pressed),.midi_ready(midi_ready),
                        .new_frame(new_frame));

        wire record_negedge;
        level_to_pulse ready_negedge(.clock(clock), .reset(reset), .level(~record),.pulse(record_negedge));

        reg [1:0] reset_extend;

        ///
        /// RECORD LOGIC
        ///
        always @ (posedge clock) begin
                if(reset || record || reset_extend) begin //on reset load reasonable values
                        //These values are determined after every record
                        sample_start_q <= 0;
                        sample_end_q <= sample_length;
                        sustain_position_q <= (sample_start+sample_end)>>1;
                        sustain_length_q <=     1024;
                        echo_delay_q <= 14'b11111111111111;
                        pitch_offset_q <= 24;
                        {sample_start_ld,sample_end_ld,sustain_position_ld,sustain_length_ld,echo_delay_ld,pitch_offset_ld}
                                <= 6'b111111; // Load values
                        if(reset)
                                reset_extend <= 1;
                        if(reset_extend > 0)
                                reset_extend <= reset_extend - 1;
                end
                else
                        {sample_start_ld,sample_end_ld,sustain_position_ld,sustain_length_ld,echo_delay_ld,pitch_offset_ld}
                                <= 6'b000000; // Allow values to change.

        end



endmodule
```

Listing 5: Verilog file cstringdisp.v

```
//
// File:    cstringdisp.v
// Date:    24-Oct-05
// Author: I. Chuang, C. Terman
//
// Display an ASCII encoded character string in a video window at some
// specified x,y pixel location.
//
// INPUTS:
//
//    vclock        - video pixel clock
//    hcount        - horizontal (x) location of current pixel
//    vcount        - vertical (y) location of current pixel
//    cstring       - character string to display (8 bit ASCII for each char)
//    cx,cy         - pixel location (upper left corner) to display string at
//
// OUTPUT:
//
//    pixel         - video pixel value to display at current location
```

```
//
// PARAMETERS:
//
//    NCHAR          - number of characters in string to display
//    NCHAR_BITS     - number of bits to specify NCHAR
//
// pixel should be OR'ed (or XOR'ed) to your video data for display.
//
// Each character is 8x12, but pixels are doubled horizontally and vertically
// so fonts are magnified 2x.  On an XGA screen (1024x768) you can fit
// 64 x 32 such characters.
//
// Needs font_rom.v and font_rom.ngo
//
// For different fonts, you can change font_rom.  For different string
// display colors, change the assignment to cpixel.


////////////////////////////////////////////////////////////////////////////
//
// video character string display
//
////////////////////////////////////////////////////////////////////////////

module char_string_display (vclock,hcount,vcount,pixel,cstring,cx,cy);

   parameter NCHAR = 8;  // number of 8-bit characters in cstring
   parameter NCHAR_BITS = 3; // number of bits in NCHAR
        parameter COLOR = 7;

   input vclock;          // 40MHz clock
   input [10:0] hcount;  // horizontal index of current pixel (0..799)
   input [9:0]  vcount;  // vertical index of current pixel (0..599)
   output [2:0] pixel;   // char display's pixel
   input [NCHAR*8-1:0] cstring;  // character string to display
   input [10:0] cx;
   input [9:0]  cy;

   // 1 line x 8 character display (8 x 12 pixel-sized characters)

   wire [10:0]  hoff = hcount-1-cx;
   wire [9:0]   voff = vcount-cy;
   wire [NCHAR_BITS-1:0] column = NCHAR-1-hoff[NCHAR_BITS-1+3:3];  // < NCHAR
   wire [2:0]   h = hoff[2:0];              // 0 .. 7
   wire [3:0]   v = voff[3:0];              // 0 .. 11

   // look up character to display (from character string)
   reg [7:0]  char;
   integer  n;
   always @(*)
     for (n=0 ; n<8 ; n = n+1 )          // 8 bits per character (ASCII)
       char[n] <= cstring[column*8+n];

   // look up raster row from font rom
   wire reverse = char[7];
   wire [10:0] font_addr = char[6:0]*12 + v;     // 12 bytes per character
   wire [7:0]  font_byte;
   font_rom f(font_addr,vclock,font_byte);

   // generate character pixel if we're in the right h,v area
   wire [2:0] cpixel = (font_byte[7 - h] ^ reverse) ? COLOR : 0;
   wire dispflag = ((hcount > cx) & (vcount >= cy) & (hcount <= cx+NCHAR*8)
                    & (vcount < cy + 12));
   wire [2:0] pixel = dispflag ? cpixel : 0;

endmodule
```

Listing 6: Verilog file debounce.v

```
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
module debounce (reset, clock, noisy, clean);
   parameter DELAY = 200000;   // .005 sec with a 40MHz clock
   input reset, clock, noisy;
   output clean;

   reg [18:0] count;
   reg new, clean;

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

```
        count <= count+1;

endmodule
```

Listing 7: Verilog file delay.v

```
/**
 * delay - Pulse generated after one of two selectable delay.
 *
 * @parameter DELAY2_COUNT The delay in clock cycles when short == 0;
 * @parameter DELAY1_COUNT The delay in clock cycles when short == 1;
 * @parameter COUNT_SIZEOF The number of bits necessary to represent max(DELAY1_COUNT,DELAY2_COUNT)
 *
 * @input clock System clock.
 * @input reset Reset the counter for pulse generation.
 * @input short Select between DELAY1_COUNT (1) and DELAY2_COUNT (0) for delays.
 *
 * @output pulse A pulse generated after a delay of (short ? DELAY1_COUNT : DELAY2_COUNT)
 *
 * @debug [COUNT_SIZEOF-1:0] counter Internal counter for pulse generation.
 */
module delay(clock,reset,pulse,counter, short);
   parameter DELAY2_COUNT   = 862;
   parameter COUNT_SIZEOF = 10;
   parameter DELAY1_COUNT = 430;

   input clock;
   input reset;
   input short;

   output pulse;
   output [COUNT_SIZEOF-1:0] counter;

   reg pulse;
   reg [COUNT_SIZEOF-1:0] counter;

   always @  (posedge clock) begin
        if(reset) begin
                counter <= 0;
                pulse    <= 0; //reset state of pulse low
        end
        else begin
                if((short && counter == DELAY1_COUNT-1) ||
                   (~short && counter == DELAY2_COUNT-1))
                        pulse <= 1; // we've waited long enough and no longer.
                else
                        pulse <= 0;
                 // increments until we are a bit past the delay, meaning pulse stays high for only one clock cycle
                if((short && counter < DELAY1_COUNT) ||
                   (~short && counter < DELAY2_COUNT)) begin
                        counter <= counter + 1;
                end
        end
   end
endmodule
```

Listing 8: Verilog file display 16hex.v

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Hex display driver
//
//
// File:    display_16hex.v
// Date:    24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// This module drives the labkit hex displays and shows the value of
// 8 bytes (16 hex digits) on the displays.
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove clear
// 02-Nov-05 Ike: updated to make it completely synchronous
//
// Inputs:
//
//    reset         - active high
//    clock_27mhz - the synchronous clock
//    data          - 64 bits; each 4 bits gives a hex digit
//
// Outputs:
//
//     disp_*       - display lines used in the 6.111 labkit (rev 003 & 004)
//
///////////////////////////////////////////////////////////////////////////////

module display_16hex (reset, clock_27mhz, data_in,
                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_in;              // 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 [5:0] count;
   reg [7:0] reset_count;
//   reg            old_clock;
   wire       dreset;
   wire       clock = (count<27) ? 0 : 1;

   always @(posedge clock_27mhz)
     begin
        count <= reset ? 0 : (count==53 ? 0 : count+1);
        reset_count <= reset ? 100 : ((reset_count==0) ? 0 : reset_count-1);
//      old_clock <= clock;
     end

   assign dreset = (reset_count != 0);
   assign disp_clock = ~clock;
   wire   clock_tick = ((count==27) ? 1 : 0);
//   wire   clock_tick = clock & ~old_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
   reg [63:0] data;

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

   always @(posedge clock_27mhz)
     if (clock_tick)
       begin
          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
                   state <= state+1;
                end
              8'h04:
                begin
                   // Setup the control register
                   disp_rs <= 1'b1; // Select 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
                   data <= data_in;
                   state <= state+1;
                end

              8'h06:
                begin
                   // Load the user's dot data into the dot reg, char by char
                   disp_rs <= 1'b0;                      // Select the dot register
                   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
                         data <= data_in;
                         dot_index <= 39;
                       end
                   else
                     dot_index <= dot_index-1;  // else loop thru all dots
                end

            endcase // casex(state)
       end

   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
```

Listing 9: Verilog file echo.v

```
module echo(clock, reset, new_frame, echo_time, decay_rate, enable, data_in,
                                data_out, ramoutwire, eramaddr, e_we, echostate);
        input clock;
        input reset;
        input new_frame;
        input [12:0] echo_time;  //time to echo for, in terms of frames.
        input [1:0] decay_rate;  //rate at which to decay, in terms of shift operations
        input enable;            //enable or disable echo

        input signed [17:0] data_in;     //input data
        output signed [17:0] data_out; //output data, with echo on it
        //debugging outputs:
        output signed [17:0] ramoutwire; //the output of the echo ram
        output [12:0] eramaddr;  //the address of the echo ram
        output e_we;               //write enable on the echo ram
        output [1:0] echostate; //state of the echo FSM

        reg [12:0] eramaddr;     //ram address of the current frame.
        reg [17:0] data_out;     //latched output frame.

        reg [1:0] echostate;     //state of echo operation for a given frame.
        reg e_we;                //write enable for the BRAM used in this module.
        reg [17:0] data_in_holdvalue;       //latched input data
        reg signed [18:0] big_data_out;     //output data without a bit cut off
                                 //(due to the addition, we end up with another bit).

        wire signed [17:0] ramoutwire;      //output wire of the BRAM.

        wire signed [17:0] nothingness;
        assign nothingness = 0;
        wire signed [17:0] muxed_data_in = enable? data_in_holdvalue:nothingness;
                        //mux on the output to determine if echo should be enabled.

eram14x18 eram1(.addr(eramaddr),.clk(clock),.din(data_in_holdvalue),.dout(ramoutwire),.we(e_we));
                                          //note here that data out is the frame output of the module.


//memory for echo

always @ (posedge clock) begin
        if(reset) begin
                eramaddr <= 0;
                //endpos <= 0;
                big_data_out <= 0;
                echostate <= 0;
                e_we <= 0;
                data_in_holdvalue <= 0;
                data_out <= 0;
        end
        else begin
                        if(new_frame) data_in_holdvalue <= data_in;
                        data_out <= big_data_out[18:1];
                        if(enable) begin
                                case(echostate)
                                        0: begin
                case(decay_rate)
                        0:      big_data_out <= ramoutwire + data_in_holdvalue;
                        1:      big_data_out <= {ramoutwire[17], ramoutwire[17], ramoutwire[16:1]}
                                                                          + data_in_holdvalue;
                        2: big_data_out <= {ramoutwire[17], ramoutwire[17], ramoutwire[17],
                                                          ramoutwire[16:2]} + data_in_holdvalue;
                        3: big_data_out <= {ramoutwire[17], ramoutwire[17], ramoutwire[17],
                                        ramoutwire[17], ramoutwire[16:3]} + data_in_holdvalue;
                endcase
                                                echostate <= 1;
                                                e_we <= 1;
                                                end
                                        1:begin
                                                echostate <= 2;
                                                e_we <= 0;
                                        end
                                        2: begin
                                                //data_out <= addvalue;
                                                if(eramaddr<echo_time) eramaddr <= eramaddr + 1;
                                                else eramaddr <= 0;
                                                echostate <= 3;
                                        end
                                        3:begin
                                                if(new_frame) echostate <= 0;
                                        end
                                endcase
                        end
                        else begin
                                data_out <= data_in;
                                eramaddr <= eramaddr + 1;
                                e_we <= 1;
                                echostate <= 0;
                        end
        end
end
endmodule
```

Listing 10: Verilog file final.v

```
///////////////////////////////////////////////////////////////////////////////
//
// 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 final   (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]  user2, user3;

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

   ////////////////////////////////////////////////////////////////////////////
   //
   // 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.
   //
   ////////////////////////////////////////////////////////////////////////////

   // Audio Input and Output
   assign beep= 1'b0;
```

```
//lab3 assign audio_reset_b = 1'b0;
//lab3 assign ac97_synch = 1'b0;
//lab3 assign ac97_sdata_out = 1'b0;
// ac97_sdata_in is an input

// VGA Output
     /*
assign vga_out_red = 10'h0;
assign vga_out_green = 10'h0;
assign vga_out_blue = 10'h0;
assign vga_out_sync_b = 1'b1;
assign vga_out_blank_b = 1'b1;
assign vga_out_pixel_clock = 1'b0;
assign vga_out_hsync = 1'b0;
assign vga_out_vsync = 1'b0;
     */
// Video Output
assign tv_out_ycrcb = 10'h0;
assign tv_out_reset_b = 1'b0;
assign tv_out_clock = 1'b0;
assign tv_out_i2c_clock = 1'b0;
assign tv_out_i2c_data = 1'b0;
assign tv_out_pal_ntsc = 1'b0;
assign tv_out_hsync_b = 1'b1;
assign tv_out_vsync_b = 1'b1;
assign tv_out_blank_b = 1'b1;
assign tv_out_subcar_reset = 1'b0;

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

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

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

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

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

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

// Buttons, Switches, and Individual LEDs
//lab3 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
//wumpus 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;

     ///
     /// CLOCK SYNTHESIS
     /// The clock is 40.5 MHz
     ///
     wire clock_unbuf,clock;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 2
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 3
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
     // synthesis attribute CLKIN_PERIOD of vclk1 is 37

BUFG vclk2(.O(clock),.I(clock_unbuf));

// power-on reset generation
wire power_on_reset;    // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock), .Q(power_on_reset),
                .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;


     ///
     /// AUDIO INTERFACING
     ///
     wire new_frame;
     wire [17:0] from_audio;
     wire [17:0] to_audio;
     audio bidirectional_audio(.clock(clock), .reset(reset),
                                                                        .audio_out_data(to_audio), .audio_in_data(from_audio), .ready(ready),
                                                            .audio_reset_b(audio_reset_b), .ac97_sdata_out(ac97_sdata_out), .ac97_sdata_in(ac97_sdata_in),
                                                      .ac97_synch(ac97_synch), .ac97_bit_clock(ac97_bit_clock));



     level_to_pulse new_frame_maker(.clock(clock), .reset(reset), .level(ready),.pulse(new_frame));



     ///
     /// SYNCRONIZATION
     /// loop - switch [0]
     /// record - button_enter
     /// user4[

     input [31:0] user1;
     input [31:0] user4;

     wire [1:0] sample_start_ctl,sample_end_ctl,sustain_position_ctl,sustain_length_ctl,echo_delay_ctl,pitch_offset_ctl;
     wire record;
     wire loop;
     wire echo_enable;
     wire [1:0] echo_decay;
     wire midi_rx;

     // Record
     debounce debounce_record(.reset(reset), .clock(clock), .noisy(~button_enter), .clean(record));
     // Loop Switch
     synchronize sync_loop(.clk(clock), .in(switch[0]), .out(loop));


     // Echo
     synchronize sync_echoe(.clk(clock), .in(switch[1]), .out(echo_enable));
     synchronize sync_echod0(.clk(clock), .in(switch[2]), .out(echo_decay[0]));
     synchronize sync_echod1(.clk(clock), .in(switch[3]), .out(echo_decay[1]));

     synchronize sync_c(.clk(clock), .in(switch[4]), .out(c));

     debounce debounce_control[9:0](.clock(clock),.reset(reset),
                                                        .noisy(user4[13:4]),
                                                        .clean({sample_start_ctl[1:0],sustain_position_ctl[1:0],sustain_length_ctl[1:0],sample_end_ctl[1:0],echo_delay_ctl[1:0]

     debounce deb_pitch_offset1(.clock(clock),.reset(reset),
                                                        .noisy(user4[3]),.clean(pitch_offset_ctl[0]));
     debounce deb_pitch_offset0(.clock(clock),.reset(reset),
                                                        .noisy(user4[2]),.clean(pitch_offset_ctl[1]));
```

```
    //MIDI Serial Input
    synchronize sync_rx(.clk(clock),.in(user1[1]),.out(midi_rx));

    ///
    /// CONTROL
    ///

    // Inputs from section NOTES
    wire [16:0] sample_length;
    wire finished,note_ready;
    wire [2:0] finished_note_select,note_select;

    // Outputted Parameters
    wire [16:0] sample_start,sample_end,sustain_position;
    wire [11:0] sustain_length;
    wire [13:0] echo_delay;
    wire [5:0]  pitch_offset;


    wire [15:0]period;
    wire [7:0] velocity;
    wire pressed;

    wire serial_ready;
    wire [7:0] serial_byte;


    control my_control(.clock(clock), .reset(reset),
                                    .new_frame(new_frame),
                                    .record(record),
                                    .sample_length(sample_length),
                                    .loop(loop),
                                    /// COMMUNICATION WITH NOTES
                                    .finished(finished),.finished_note_select(finished_note_select),
                                    .note_ready(note_ready),.note_select(note_select),.period(period),.velocity(velocity),.pressed(pressed),
                                    /// PARAMETERS
                                    .sample_start(sample_start),.sample_end(sample_end),
                                    .sustain_position(sustain_position),.sustain_length(sustain_length),
                                    .echo_delay(echo_delay),.pitch_offset(pitch_offset),
                                    .midi_rx(midi_rx),
                                    .sample_start_ctl(sample_start_ctl),.sample_end_ctl(sample_end_ctl),
                                    .sustain_position_ctl(sustain_position_ctl),.sustain_length_ctl(sustain_length_ctl),
                                    .echo_delay_ctl(echo_delay_ctl),.pitch_offset_ctl(pitch_offset_ctl)
                                    /// DEBUG
                                    ,.serial_ready(serial_ready),.serial_byte(serial_byte)
                                    );

    ///
    /// NOTES
    ///

    wire [17:0] notes_to_audio_play;
    wire [15:0] notes_to_audio_record;
    wire [17:0] echo_to_audio;

    notes notes1(.clock(clock),.reset(reset),
                                    .new_frame(new_frame),.w_rbar(record),
                                    .ns_write(note_ready),.note_select(note_select),.period_in(period),.velocity_in(velocity),.pressed_in(pressed),
                                    .finished_note(finished_note_select),.finished_state(finished),
                                    .sample_start(sample_start),.sample_end(sample_end),.window_hold_pos(sustain_position),.window_hold_length(sustain_length),
                                    .data_out(notes_to_audio_play),
                                    .sample_length(sample_length),
                                    .from_ac97_data(from_audio[17:2]),.to_ac97_record(notes_to_audio_record),
                                    .taylor_done(taylor_done),
                                    .current_note(current_note),
                                    .ready_frameprep(ready_frameprep),.mems_ready(mems_ready),
                                    .pressed_out(pressed_out),.period_out(period_out),
                                    .n_new(n_new),.going_out(going_out),.going_new(going_new),.n_null(n_null),
                                    .n_out(n_out),.frame_n(frame_n),.frame_n_plus(frame_n_plus),.frame_n_minus(frame_n_minus),.ram_addr(ram_addr)
                                    );


    ///
    /// ECHO
    ///

    echo my_echo(.clock(clock),.reset(reset),.new_frame(new_frame),
                                     .echo_time(echo_delay),.decay_rate(echo_decay),.enable(echo_enable),
                                     .data_in(notes_to_audio_play),.data_out(echo_to_audio));

    wire [17:0] c_to_audio;
    assign c_to_audio[0] = 0;
    middle_c my_middle_c(
.CLK(clock),
.SINE(c_to_audio[17:1])
);

    assign to_audio = c ? c_to_audio : ( record ? {2'b0,notes_to_audio_record} : echo_to_audio);

    ///
    /// VIDEO OUTPUT GENERATION
    ///

// generate basic XVGA video signals
wire [10:0] hcount;
wire [9:0]  vcount;
```

```
   wire hsync,vsync,blank;
   xvga xvga1(clock,hcount,vcount,hsync,vsync,blank);

   // feed XVGA signals to user's pong game
   wire [2:0] pixel;

   reg [2:0] rgb;
   reg b,hs,vs;

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz.
   assign vga_out_red = {8{rgb[2]}};
   assign vga_out_green = {8{rgb[1]}};
   assign vga_out_blue = {8{rgb[0]}};
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_blank_b = ~b;
   assign vga_out_pixel_clock = ~clock;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;


        visualizer viz(.clock(clock),.reset(reset),
                .hcount(hcount),.vcount(vcount),.hsync(hsync),.vsync(vsync),.blank(blank),
                .vhsync(vhsync),.vvsync(vvsync),.vblank(vblank),.pixel(pixel),
                .sample_start(sample_start),.sample_end(sample_end),
                .sample_length(sample_length),
                .sustain_position(sustain_position),.sustain_length(sustain_length),
                .echo_delay(echo_delay),
                .pitch_offset(pitch_offset),
                .audio(from_audio[15:0]),.record(record),.new_frame(new_frame));

        ///
        /// DEBUG OUTPUTS
        ///
        reg [7:0] lights;
        assign led = ~ lights;
        assign analyzer1_clock = clock;
        assign analyzer1_data = {serial_byte,6'b0,serial_ready,midi_rx};
        assign analyzer2_data = {period};

        always @ (posedge clock) begin
                if(reset) begin

                end else
                        lights <= sustain_length[11:4];
                hs <= vhsync;
                vs <= vvsync;
                b <= vblank;
                rgb <= pixel;
        end



endmodule
```

Listing 11: Verilog file grey_decode.v

```
/**
 * grey_decode -
 *
 * @input clock
 * @input reset
 * @input a
 * @input b
 * @output up_down
 * @output ready_pulse
 */
module grey_decode(clock,reset,a,b,up_down,ready_pulse);
    input clock;
    input reset;
    input a;
    input b;
    output up_down;
    output ready_pulse;

         parameter DELAY = 200000; //Delay for .005 ms

         reg up_down;
         reg ready_pulse;

         reg old_a,old_b;

         always @ (posedge clock) begin
                 if(reset) begin
                    up_down <= 0;
                         ready_pulse <= 0;
                 end
                 else begin
               if( (old_a && ~a && b) || (old_b && ~b && a)) begin // on falling edge
                                ready_pulse <= 1;
                                up_down <= b ? 1 : 0;
                         end
                         else
                                ready_pulse <= 0;
                 end
```

```
                old_a <= a;
                old_b <= b;
        end

endmodule
```

Listing 12: Verilog file level_to_pulse.v

```
/**
 * Converts a rising edge on level into a one clock cycle pulse.
 *
 * @clk The System Clock.
 * @level The trigger for the pulse.
 * @pulse A signal which pulls high for one clock cycle on the rising edge of level.
 */
module level_to_pulse(clock, reset, level,pulse);
        input clock;
                  input reset;
        input level;

        output pulse;

        reg old_level;
        reg pulse;

        always @ (posedge clock) begin
                     if(reset) begin
                             old_level <= 0;
                                  pulse <= 0;
                          end
                          else begin
                             old_level <= level;
                                  pulse <= level & ~ old_level;
           end
        end
endmodule
```

Listing 13: Verilog file linemem.v

```
module linemem(clock,reset,clear_all,write_enable,write_addr,
                                write_data,read_addr,read_data);
        input clock;
        input reset;
        input clear_all;        //clears all memory data. Does the same thing as reset,
                                         but is here to make the code in notemem clearer
        input write_enable;
        input [2:0] write_addr;         //address for writing memory
        input [31:0] write_data;        //data to write to memory
        input [2:0] read_addr;          //address for reading memory
        output [31:0] read_data;        //data read from memory

        reg [255:0] memory_line;    //this is all the data and addressed of the memory
                                                        aligned in a line of registers...

        wire[7:0] read_addr_appended; //shifted read address, to align the requested
                                                        address with the array of registers.

assign read_addr_appended = read_addr<<5;
assign read_data = memory_line>>read_addr_appended;



//assign write_MSB = {write_addr, 5'b11111};
//assign write_LSB = {write_addr, 5'b00000};

always @ (posedge clock) begin
        if(reset) begin
                memory_line <= 0;
        end
        else if(clear_all) begin
                memory_line <= 0;
        end
        else if(write_enable) begin
                case(write_addr)
                        0: memory_line[31:0] <= write_data[31:0];
                        1: memory_line[63:32] <= write_data[31:0];
                        2: memory_line[95:64] <= write_data[31:0];
                        3: memory_line[127:96] <= write_data[31:0];
                        4: memory_line[159:128] <= write_data[31:0];
                        5: memory_line[191:160] <= write_data[31:0];
                        6: memory_line[223:192] <= write_data[31:0];
                        7: memory_line[255:224] <= write_data[31:0];
                endcase
        end
end
endmodule
```

Listing 14: Verilog file loop_control_fsm.v

```
/**
 * loop_control_fsm —
 * @input clock System clock.
 * @input rest  Syncronous reset.
 * @input [2:0] finished_note_select Indicates which note finished describes the status of.
 * @input finished Describes whether a note is finished or not.

 * @output [2:0] note_select Indicates which note should receive an event.
 * @output ready Indicates that the note selecte by note_select should receive an event.
 * @output pressed Indicates whether an event is a press (high) or rlease (low).
 */
module loop_control_fsm(clock, reset, new_frame, finished, finished_note_select,
                           pressed, ready,
                                                                      //Debug outputs
                                                                      state);
    input clock;
    input reset;
    input new_frame;
    input finished;
    input [2:0] finished_note_select;
    output pressed;
         output ready;
         output [2:0] state;

         reg pressed;
         reg ready;


         parameter WAIT_FRAME0 = 0;      //wait for a new frame
         parameter WAIT_FINISH = 1;      //wait for to for the zeroth note to finish
         parameter PRESS = 2;            //press the note
         parameter WAIT_FRAME1 = 3;      //wait for another two frame
         parameter RELEASE = 4;          //release the note

         //Store the 5 states abov
         reg [2:0] state;
         reg [2:0] old_state; //compare

         reg [1:0] frame_count; //sequencer for wait_frame1

         always @ (posedge clock) begin
       if(reset) begin
                         state <= WAIT_FRAME0;
                    old_state <= RELEASE;
                         pressed <= 0;
                         ready <= 0;
                         frame_count <= 0;
                 end
                 else begin
                // Calculate state transitions
                 case(state)
                        WAIT_FRAME0:
                                if(new_frame)
                                        state <= WAIT_FINISH;
                        WAIT_FINISH:
                                if(finished_note_select == 0 && finished)
                                        state <= PRESS;
                        PRESS:
                                state <= WAIT_FRAME1;
                         WAIT_FRAME1:
                                if(frame_count == 2)
                                        state <= RELEASE;
                        RELEASE:
                                state <= WAIT_FRAME0;
                        default:
                                state <= WAIT_FRAME0;
                 endcase

                //Calculate new outputs
                if(state != old_state || state == WAIT_FRAME1) begin
                    case(state)
                           WAIT_FRAME0: begin
                                   ready <= 0;
                                   frame_count <= 0;
                           end
                           //WAIT_FINISH:
                           PRESS: begin
                                   pressed <= 1;
                                   ready <= 1;
                           end

                           WAIT_FRAME1: begin
                                    ready <= 0;
                                    if(new_frame) frame_count <= frame_count+1;
                           end

                           RELEASE: begin
                                    pressed <= 0;
                                    ready <= 1;
                           end
                         endcase

                         old_state <= state;

                 end

         end
```

```
        end

endmodule
```

Listing 15: Verilog file midi_fsm.v

```
/**
 * midi_fsm - Interprets MIDI messages for Pressed and Released events one byte at a time.
 *
 * @input clock System clock.
 * @input reset Syncronous reset.
 * @input new_byte Single clock second pulse to indicate a new byte is available.
 * @input [7:0]byte The most recent MIDI byte, must be stable at least two cycles after new_byte.
 *
 * @output pressed Whether the event is a Pressed (1) or Released (0) event.
 * @output [7:0]pitch The pitch of the event.
 * @output [7:0]velocity The velocity of the event.
 * @output ready A one clock cycle pulse indicatiing a new event has occured.
 */
module midi_fsm(clock,reset,byte,new_byte,pitch,velocity,ready,pressed,state);
    input clock;
    input reset;
    input [7:0] byte;
    input new_byte;
    output [7:0] pitch;
    output [7:0] velocity;
    output ready;
    output pressed;
    output [3:0] state;

    reg [7:0] pitch;
    reg [7:0] velocity;
    reg ready;
    reg pressed;
         reg [7:0]fbyte;

    //The states of the finite state machine
    parameter WAIT = 0;           //wait for the first byte to arrive
    parameter FUNCTION = 1;       //deterimine the function of the midi event
    parameter IGNORE_1 = 2;       //ignore 1 or
    parameter IGNORE_2 = 3;       //2 data bits for unsupported opcodes
    parameter ON_OFF = 4;         //determine pressed for the midi event
    parameter PITCH_WAIT = 5;     //wait for the next byte
    parameter PITCH = 6;          //store the pitch value
    parameter VELOCITY_WAIT = 7;//wait for the next byte
    parameter VELOCITY = 8;       //store the velocity value
    parameter READY = 9;          //indicate the midi message is complete

    reg [3:0] state; // four bits for the 9 states above
    reg [3:0] old_state; //state from previous clock cycle




    always @ (posedge clock) begin
        if(reset) begin
                ready <= 0;
                state <= WAIT;
                old_state <= 4'hF;
                pitch <= 0;
                velocity <= 0;
                pressed <=  0;
        end
        else begin

        //State transition calculations
        case(state)
                WAIT:
                        if(new_byte) begin //wait for the first byte
                                state <= FUNCTION;
                                fbyte <= byte;
                        end
                FUNCTION: begin

                        if(fbyte[7:4] == 8 || fbyte[7:4] == 9) //pressed or released events
                                state <= PITCH_WAIT;

                        //for all other events ignore the appropriate 2, 1, or 0 bytes that follow
                        else if(fbyte[7:4] == 4'hA || fbyte[7:4] == 4'hB || fbyte[7:4] == 4'hE || fbyte == 8'hF2)
                                state <= IGNORE_2;
                        else if(fbyte[7:4] == 4'hC || fbyte[7:4] == 4'hD || fbyte == 8'hF3 || fbyte == 8'hF5)
                                state <= IGNORE_1;
                        else
                                state <= WAIT;
                end
                IGNORE_1:
                        if(new_byte) //wait for a byte...
                                state <= WAIT; //... and ignore it
                IGNORE_2:
                        if(new_byte) //wait for a byte...
                                state <= IGNORE_1; //... and ignore the next one too

                PITCH_WAIT:
                        if(new_byte) //pitch value has arrived
                                state <= PITCH;
```

```
			PITCH:
				state <= VELOCITY_WAIT; //wait for velocity value
			VELOCITY_WAIT:
				if(new_byte)
					state <= VELOCITY; //velocity value has arrived
			VELOCITY:
				state <= ON_OFF;
			ON_OFF:
				state <= READY; //immediately wait for pitch
			READY:
				state <= WAIT;
			default:
				state <= WAIT;
		endcase

		//State output calculations
		if(old_state != state) //this comparison adds an extra clock cycle delay to outputs taking effect
		case(state)
			WAIT: ready <= 0;
			//FUNCTION:
			//IGNORE_1:
			//IGNORE_2:
			ON_OFF: pressed <= velocity == 0 ? 0 : 1;
			//PITCH_WAIT:
			PITCH:  pitch <= byte;
			//VELOCITY_WAIT:
			VELOCITY: velocity <= byte;
			READY: ready <= 1;
			default: ready <= 0;
		endcase
		old_state <= state;
		end
		end


endmodule
```

Listing 16: Verilog file newfp.v

```
module newfp(clock, reset, mems_ready, n_out, frame_n, frame_n_plus, frame_n_minus, ready_frameprep,
				w_rbar, new_frame, from_ac97_data, to_ac97_record, ramaddr, sample_length);
	input clock;
	input reset;
	input [16:0] n_out;
	input w_rbar;
	input new_frame;
	input mems_ready;
	input [15:0] from_ac97_data;
	output [15:0] frame_n, frame_n_plus, frame_n_minus;
	output ready_frameprep;
	output [15:0] to_ac97_record;
	output [16:0] ramaddr;
	output [16:0] sample_length;

	parameter MAXSIZE = 98304; //we reduced the maxsize from 2^17 down to three fourths
				//that in order to meet the chip's memory constraints. Still
				//plenty of sample time.

	reg ready_frameprep;	//signal that frameprep is done fetching data from the memory,
				//and that taylor can start calculating.
	reg [16:0] ramaddr;	//current address in the sample memory being used
	reg old_w_rbar;		//previous signal of the write button, used to determine when
				//the button was pressed and thus when to reset ramaddr
	reg [2:0] fstate;	//state of the frame fetch FSM. states are counted through linearly.
	reg ready_hold;		//holds the ready value from mems_ready, such that the FSM can
				//complete it's action
	reg [15:0] frame_n, frame_n_minus, frame_n_plus;  //frames fetched from memory and
				//prepared for taylor
	reg [16:0] sample_length;  //length of the last recorded sample, used by control to
				//determine a maximum sample_end value.

wire [15:0] ramout;

assign to_ac97_record = ramout;

memram memram1(.clk(clock),.addr(ramaddr),.we(old_w_rbar),.din(from_ac97_data),.dout(ramout));


always @ (posedge clock) begin
	if(reset) begin
		ready_frameprep <= 0;
		ramaddr <= 0;
		old_w_rbar <= 0;
		frame_n <= 0;
		frame_n_minus <= 0;
		frame_n_plus <= 0;
		ready_hold <= 0;
		fstate <= 0;
		sample_length <= 0;
	end
	else begin
		if(ready_frameprep) ready_frameprep <= 0;
		old_w_rbar <= w_rbar;
		if(w_rbar) begin
			if(~old_w_rbar) ramaddr <= 0;
```

```
                        else if(ramaddr < MAXSIZE) if(new_frame) begin
                                ramaddr <= ramaddr + 1;
                                sample_length <= ramaddr;
                        end
                        else    begin
                                        if(new_frame) ramaddr <= 0;
                        end
                end
                else begin
                        if(mems_ready|ready_hold) begin
                                case(fstate)
                                        0:begin
                                                ramaddr <= n_out;
                                                ready_hold <= 1;
                                                fstate <= 1;
                                        end
                                        1:begin
                                                ramaddr <= n_out+1;
                                                fstate <= 2;
                                        end
                                        2:begin
                                                ramaddr <= n_out-1;
                                                frame_n <= ramout;
                                                fstate <= 3;
                                        end
                                        3: begin
                                                frame_n_plus <= ramout;
                                                fstate <= 4;
                                        end
                                        4:begin
                                                frame_n_minus <= ramout;
                                                ready_hold <= 0;
                                                fstate <= 0;
                                                if(~ready_frameprep) ready_frameprep <= 1;
                                        end
                                 endcase
                        end
                end
        end
end
endmodule
```

Listing 17: Verilog file note_control_fsm.v

```
/**
 * note_control_fsm
 *
 */
module note_control_fsm(clock,reset,queue_empty,queue_pop,
                           finished,finished_note_select,note_ready,note_select,
                                                                queue_period, queue_pressed, queue_velocity,new_frame,
                                                                period,pressed,velocity,
                                                                //Debug Outputs
                                                                state,note_period,finished_reg_out);
    parameter NOTE_COUNT = 8; //The number of notes
         parameter NOTE_COUNT_SIZEOF = 3; //The bit width necessary to capture it
    input clock;
    input reset;
    input queue_empty;
         input [15:0] queue_period;
         input [7:0] queue_velocity;
         input queue_pressed;
         output [15:0] period;
         output [7:0] velocity;
         output pressed;
    output queue_pop;
    input finished;
         input new_frame;
    input [2:0] finished_note_select;
    output note_ready;
         output [2:0] note_select;
         output [2:0] state;
         output [15:0] note_period;
         output [7:0] finished_reg_out;

         reg [15:0] period;
         reg [7:0] velocity;
         reg pressed;

         reg queue_reset;
         reg queue_pop;

         reg [2:0] note_select;
         reg note_ready;


        reg finished_reg [7:0];
        assign finished_reg_out[7:0] =  {finished_reg[0],finished_reg[1],finished_reg[2],finished_reg[3],finished_reg[4],finished_reg[5],finished_reg[6],finished_reg[7]};
        reg [15:0] note_periods [7:0];

        assign note_period = note_periods[note_select];

        //Constants representing the different states of the FSM
        parameter WAIT   = 0      //Wait for the queue to not be empty;
        parameter LATCH = 1;     //Latch the queue inputs
        parameter CHECK_PRESSED = 2;     //if a pressed event, see if any notes are free
        parameter CHECK_RELEASED =3;     //if a released event, see if the particular note is playing (changes note_select)
        parameter FULL   = 4;            //if no notes are free (changes note_select)
        parameter NOTE_PRESSED = 5;      //press the note (pull ready high)
        parameter NOTE_RELEASED = 6;     //release the note (pull ready high)
        parameter NOTE_NOT_FOUND = 7;    //if a note being released wasn't found
        parameter WAIT_FRAME = 8;        //delay a variable number of frame pulses before checking the queue again (currently 9)

        reg [3:0] state;     //There are four states as listed above
        reg [3:0] wait_frames;

        reg [3:0] old_state;
        reg [3:0] old_wait_frames;
        reg [1:0] old_note_select;

        always @ (posedge clock) begin
                if(reset) begin
                        state <= WAIT;
                        note_select <= 0;
                        note_ready   <= 0;
                        finished_reg[0] <= 1;
                        finished_reg[1] <= 1;
                        finished_reg[2] <= 1;
                        finished_reg[3] <= 1;
                        finished_reg[4] <= 1;
                        finished_reg[5] <= 1;
                        finished_reg[6] <= 1;
                        finished_reg[7] <= 1;


                        queue_pop    <= 0;
                        note_periods[0] <= 22;
                        note_periods[1] <= 22;
                        note_periods[2] <= 22;
                        note_periods[3] <= 22;
                        note_periods[4] <= 22;
                        note_periods[5] <= 22;
                        note_periods[6] <= 22;
                        note_periods[7] <= 22;

                end
                else begin
                        case(state) //Transition between states if necessary
                        WAIT: begin
                        note_select <= 0;
                        if(~queue_empty) begin
                                //state <= queue_pressed ? CHECK_PRESSED : CHECK_RELEASED;
                                state <= LATCH;
                                queue_pop <= 1;
                        end
                        end
                        LATCH: begin
                                state <= queue_pressed ? CHECK_PRESSED : CHECK_RELEASED;
                                pressed <= queue_pressed;
                                velocity <= queue_velocity;
                                period <= queue_period;
                                queue_pop <= 0;
                        end

         CHECK_PRESSED:
                        if(finished_reg[note_select])
                                state <= NOTE_PRESSED;   //hah! this one isn't being used
                        else if(note_select == 7)
                                state <= FULL; //did we reach the last note?
                        else
                                note_select <= note_select + 1; //otherwise increment the note
         CHECK_RELEASED:
                        if(~finished_reg[note_select] && period == note_periods[note_select])
                                state <= NOTE_RELEASED;
         else if(note_select == 7)
                                state <= NOTE_NOT_FOUND;
                        else
                                note_select <= note_select + 1;
                        FULL:
                        state <= WAIT;
                        NOTE_PRESSED: begin
                                note_periods[note_select] <= period;
                                state <= WAIT_FRAME;
                                wait_frames <= 9;
                                note_ready <= 1;
                        end
                        NOTE_RELEASED: begin
                                state <= WAIT_FRAME;
                                wait_frames <= 9;
                                note_ready <= 1;
                        end
                        NOTE_NOT_FOUND:
                        state <= WAIT;
                        WAIT_FRAME:        begin
                        if(new_frame)
                                if(wait_frames == 0)
                                        state <= WAIT;
                                else
                                        wait_frames <= wait_frames -1;
                        end
                        endcase
```

```
                        finished_reg [finished_note_select] <= finished; //ignore finished if we just turned the note on
                end

        end

endmodule
```

Listing 18: Verilog file note_control.v

```
/**
 *      note_control - Manages loop and keyboard output to a set of notes.
 *
 * @parameter NOTE_COUNT The number of notes available.
 * @parameter NOTE_COUNT_SIZEOF The number of bits necessary to represent NOTE_COUNT
 *
 * @input clock The system clock.
 * @input reset A syncronous reset.
 * @input midi_ready A one clock cycle pulse indicating a new keyboard event.
 * @input midi_pressed Whether a keyboard event is press or release.
 * @input [7:0] midi_velocity The velocity of the keyboard event.
 * @input [7:0] midi_pitch The pitch of the keyboard event.
 * @input [NOTE_COUNT_SIZEOF-1:0]finished_note_select Selects a note to indicate the finished state of.
 * @input finished Indicates whether the note selected by finished_note_select is finished or not.
 * @input loop Indicates whether note output should be controlled in a loop (1) or by a keyboard (0)
 * @input new_frame Indicates that a new audio frame is expected.
 * @input base_period
 *
 * @output note_select Selects the target note for new note events.
 * @output pressed Indicates whether the note control event is pressed or released.
 * @output [15:0] period Indicates the period (relative to system clock) of a new note event.
 * @output [7:0] velocity Indicates the velocity of a new note event.
 * @output note_ready One clock cycle pulse indicating a new note control event is ready.
 *
 */
module note_control(clock, reset, pitch_offset, finished_note_select,
                        finished, note_select, note_ready, pressed, period, velocity, loop, new_frame,
                                                          midi_velocity, midi_pitch, midi_pressed, midi_ready, base_period,
                                                          //Debug outputs
                                                          note_fsm_state, queue_empty, queue_pop, keyboard_period, finished_reg, fifo_state,
                                                          note_period);
    parameter NOTE_COUNT = 8; //The number of notes
        parameter NOTE_COUNT_SIZEOF = 3; //The bit width necessary to capture it
    input clock;
    input reset;
    input midi_ready;
        input midi_pressed;
        input [7:0] midi_velocity;
        input [7:0] midi_pitch;
        input [5:0] pitch_offset;
    input [NOTE_COUNT_SIZEOF-1:0] finished_note_select;
    input finished;
        input loop;
        input new_frame;
        input base_period;
    output [NOTE_COUNT_SIZEOF-1:0] note_select;
    output note_ready;
        output pressed;
    output [15:0] period;
    output [7:0] velocity;

        output queue_empty;
        output queue_pop;
        output [15:0] keyboard_period;
        output [15:0] note_period;
        output [2:0] note_fsm_state;
        output [1:0] fifo_state;
        output [7:0] finished_reg;



        wire [NOTE_COUNT_SIZEOF-1:0] note_select;
        wire note_ready;

        wire keyboard_ready;
        wire [2:0] keyboard_note_select;
        wire keyboard_pressed;
        wire [7:0] keyboard_velocity;
        wire [15:0] keyboard_period;
        wire [7:0] keyboard_pitch;




        wire queue_full;
        wire queue_empty;
        wire queue_pop;

        wire [7:0] queue_pitch;
        wire [7:0] queue_velocity;
        wire queue_pressed;
        wire [15:0] queue_period;
        //Xilinx Syncronous FIFO, 32 deep
        note_fifo my_note_fifo(.clk(clock),
        .sinit(reset),
        .din({midi_pitch[7:0],midi_velocity[7:0],midi_pressed}),
        .wr_en(midi_ready),
        .rd_en(queue_pop),
        .dout({queue_pitch[7:0],queue_velocity[7:0],queue_pressed}),
        .full(queue_full),
        .empty(queue_empty));



        note_control_fsm keyboard_fsm(.clock(clock),.reset(reset),.queue_empty(queue_empty),
                                                        .queue_pop(queue_pop),
                                                        .queue_period(queue_period),.queue_velocity(queue_velocity),
                                                        .queue_pressed(queue_pressed),
                                                        .pressed(keyboard_pressed),
                                                        .velocity(keyboard_velocity),
                                                        .period(keyboard_period),
                                                        .finished(finished),.finished_note_select(finished_note_select),
                                                        .new_frame(new_frame),
                                                        .note_ready(keyboard_ready),.note_select(keyboard_note_select),
                                                        .state(note_fsm_state),.finished_reg_out(finished_reg),
                                                        .note_period(note_period));



        //adjust the period for the period offset
    period_lookup_table period_table(.clk(clock),.addr(queue_pitch + 24 - pitch_offset),
                                                        .dout(queue_period));


        wire loop_ready;
        wire loop_pressed;
        wire [2:0] loop_note_select;
        wire [7:0] loop_velocity;
        wire [15:0] loop_period;

        loop_control_fsm loop_fsm(.clock(clock),.reset(reset),
                                                                .new_frame(new_frame),.finished(finished),
                                                                .finished_note_select(finished_note_select),
                                                                .ready(loop_ready),.pressed(loop_pressed));

        assign loop_note_select = 0;
        assign loop_velocity    = 64;
        assign loop_period       = 12'b100000000000;

        assign note_select = loop ? loop_note_select : keyboard_note_select;
        assign note_ready  = loop ? loop_ready : keyboard_ready;
        assign pressed     = loop ? loop_pressed : keyboard_pressed;
        assign velocity    = loop ? loop_velocity : keyboard_velocity;
        assign period      = loop ? loop_period : keyboard_period;

endmodule
```

Listing 19: Verilog file notemem.v

```
module notemem(clock,reset,new_frame,mems_ready,w_rbar,ns_write,period_in,velocity_in,
        pressed_in,note_select,sample_start,sample_end,window_hold_pos,window_hold_length,
        n_new,delta_new,going_new,taylor_done,n_out,delta_out,going_out,velocity_out,period_out,
        pressed_out,finished_note,finished_state,current_note,incbutton,n_null);

        input clock;
        input reset;
        input new_frame;        //ac97 needs/has a new frame

        input w_rbar;           //signal from control indicating that the sound is writing or reading.

        //data written to parameter memory by control
        input ns_write;         //(note select write) indicating selected note should be written with
                                //the below data
        input pressed_in;       // value of pressed to read into note memory location from
                                        //controller. Presumably 1.
        input [15:0] period_in;         //period to write into note memory location from controller
        input [7:0] velocity_in;         //velocity to write into note memory location
        input [2:0] note_select;         //selection of note to write new values to from controller

        input [16:0] sample_start;       //start of sample read-off from controller
        input [16:0] sample_end;         //end of sample read-off from controller
        input [16:0] window_hold_pos;    //position of the window/vowel hold, from controller
        input [11:0] window_hold_length; //length of the window/vowel hold, from controller

        //data from taylor used to write the variable memory:
        input [16:0] n_new;    //new n from taylor
        input [10:0] delta_new; //new delta from taylor
        input going_new; //new statement on if note is going (that is, hasn't reached end of sample marker)
        input taylor_done;       //message from taylor saying it is done with the last note, and
                                //ready for frameprep to get a new one.

        //data output from either memory.
        output [16:0] n_out;            //n given to frameprep from the memory
        output [10:0] delta_out;        //delta given to taylor from the memory
        output going_out;                       //output of previous going state for this note.
        output [7:0] velocity_out; //velocity given to taylor from memory
        output [15:0] period_out;       //period given to taylor from memory
        output pressed_out;     //pressed given to frameprep from memory
        output mems_ready;              //notemem is ready for frameprep to read a new set of values
        //data output based on
        output [2:0] finished_note;
        output finished_state;   //{note memory location, finished}, for allocation of notes by control.
        output [2:0] current_note;

        //debugging outputs
        input incbutton;
        output [16:0] n_null;

        //registers:
        reg [2:0] current_note;
        reg [2:0] previous_note;
        reg finished_frame; //finished reading all notes for this frame (so we can stop feeding more
                                //notes into mems ready)
        reg mems_ready;
        reg taylor_done_delayed;
        reg [2:0] finished_note;   //data telling control which notes are finished.
        reg finished_state;

wire [31:0] p_data_write, p_data_read;
assign p_data_write = {7'b0000000,velocity_in, pressed_in, period_in};
assign {velocity_out, pressed_out, period_out} = p_data_read[24:0];

linemem params_mem(.clock(clock),.reset(reset),.clear_all(w_rbar),.write_enable(ns_write),.write_addr(note_select),
                                                .write_data(p_data_write),.read_addr(current_note),.read_data(p_data_read));

wire [31:0] v_data_write, v_data_read;
assign v_data_write = {3'b000, n_new, delta_new, going_new};
assign {n_out, delta_out, going_out} = v_data_read[28:0];

        //writes in vars_mem whenever taylor is done. writes to old address, so current note, actually.
linemem vars_mem(.clock(clock),.reset(reset),.clear_all(w_rbar),.write_enable(taylor_done),.write_addr(current_note),
                                                .write_data(v_data_write),.read_addr(current_note),.read_data(v_data_read));



always @ (posedge clock) begin
        if(reset) begin
                current_note <= 0;
                previous_note <= 0;
                finished_frame <= 0;
                mems_ready <= 0;
                taylor_done_delayed <= 0;
                finished_note <= 0;
                finished_state <= 0;
        end
        else begin

                taylor_done_delayed <= taylor_done;
                if(mems_ready) mems_ready <= 0;
                //count out notes upon new frame:
                if(new_frame) begin
                        current_note <= 0;
                        if((~mems_ready)&(~w_rbar)) mems_ready <= 1;//this starts off first note.
                end
                else if(taylor_done) begin
                        previous_note<=current_note;
                        finished_note <= current_note;
                        finished_state <= ~going_new;
                        if(current_note==7) begin
                                if(~finished_frame) finished_frame <= 1; //what does this do??
                        end
                        else begin
                                current_note <= current_note + 1;
                                if((~mems_ready)&(~w_rbar)) mems_ready <= 1;
                        end
                end
        end
end
endmodule
```

Listing 20: Verilog file notes.v

```
module notes(clock,reset,new_frame,w_rbar,ns_write,note_select,period_in,velocity_in,pressed_in,
        window_hold_pos,window_hold_length,finished_note,finished_state,sample_start,sample_end,
        data_out,from_ac97_data,taylor_done,current_note,ready_frameprep,mems_ready,pressed_out,
        period_out,n_out,frame_n,frame_n_plus,frame_n_minus,ram_addr,to_ac97_record,n_new,
        going_out,going_new,n_null,sample_length);

        //this module simply connects accumulator, taylor, notemem, and frameprep(newfp) together.
        input clock;
        input reset;
        input new_frame;
        input w_rbar;
        input ns_write;
        input [2:0] note_select;
        input [15:0] period_in;
        input [7:0] velocity_in;
        input pressed_in;
        input [16:0] window_hold_pos;
        input [11:0] window_hold_length;
        input [16:0] sample_start;
        input [16:0] sample_end;
        output [17:0] data_out;
        input [15:0] from_ac97_data;
        output [16:0] sample_length;

        //start of debugging outputs:
        output taylor_done;
        output [2:0] current_note;
        output ready_frameprep;
        output mems_ready;
        output pressed_out;
        output [15:0] period_out;
        output [16:0] n_out;
        output [15:0] frame_n, frame_n_plus, frame_n_minus;
        output [16:0] ram_addr;
        output [15:0] to_ac97_record;
        //new debugging ports
        output [16:0] n_new;
        output going_out;
        output going_new;
        output [16:0] n_null;

        output [2:0] finished_note;
        output finished_state;


        wire [2:0] finished_note;
        wire finished_state;

        wire mems_ready;
        wire [16:0] n_new;
        wire [10:0] delta_new;
        wire taylor_done;
        wire [16:0] n_out;
        wire [10:0] delta_out;
        wire [7:0] velocity_out;
        wire [15:0] period_out;
        wire pressed_out;

        wire [15:0] frame_n;
        wire [15:0] frame_n_plus;
        wire [15:0] frame_n_minus;
        wire ready_frameprep;
        wire [15:0] tframe_out;
        //wire [15:0] to_ac97_record;

        //make data_out either sumframe out or to_ac97_record, depending...
        wire [17:0] sumframe_out;
        wire sum_done;

        assign data_out = sumframe_out;

        //debugging outputs
        wire [16:0] ram_addr;
        wire [16:0] n_null;

        wire [16:0] sample_length;

notemem notemem1(.clock(clock),.reset(reset),.new_frame(new_frame),.mems_ready(mems_ready),
        .w_rbar(w_rbar),.ns_write(ns_write),.period_in(period_in),.velocity_in(velocity_in),
        .pressed_in(pressed_in),.note_select(note_select),.sample_start(sample_start),.sample_end(sample_end),
        .window_hold_pos(window_hold_pos),.window_hold_length(window_hold_length),.n_new(n_new),
        .delta_new(delta_new),.going_new(going_new),.taylor_done(taylor_done),.n_out(n_out),
        .delta_out(delta_out),.going_out(going_out),.velocity_out(velocity_out),.period_out(period_out),
        .pressed_out(pressed_out),.finished_note(finished_note),.finished_state(finished_state),
        .current_note(current_note),.n_null(n_null));

newfp frameprep1(clock, reset, mems_ready, n_out, frame_n, frame_n_plus, frame_n_minus, ready_frameprep,
        w_rbar, new_frame, from_ac97_data, to_ac97_record,ram_addr,sample_length);

taylor taylor1(clock,reset, period_out, pressed_out, sample_start,sample_end,window_hold_pos,window_hold_length,
        frame_n,frame_n_plus,frame_n_minus,ready_frameprep,n_out,delta_out,going_out,delta_new,n_new,going_new,
        tframe_out, taylor_done);

accumulator accumulator1(clock,reset,new_frame,taylor_done,tframe_out,sumframe_out,sum_done);



endmodule
```

Listing 21: Verilog file parameter_control.v

```
module parameter_control(clock,reset,sample_start_ctl,sample_end_ctl,sustain_pos_ctl,
                         sustain_length_ctl,echo_decay_ctl,echo_delay_ctl,
                         tuning_ctl,sample_start,sample_end,sustain_pos,sustain_length,
                         echo_delay,echo_decay,sample_length);
    input clock;
    input reset;
    input [1:0] sample_start_ctl;
    input [1:0] sample_end_ctl;
    input [1:0] sustain_pos_ctl;
    input [1:0] sustain_length_ctl;
    input [1:0] echo_decay_ctl;
    input [1:0] echo_delay_ctl;
    input [1:0] tuning_ctl;
```

```
     input [16:0] sample_length;
output [16:0] sample_start;
output [16:0] sample_end;
output [14:0] sustain_pos;
output [11:0] sustain_length;
output [10:0] echo_delay;
output [3:0] echo_decay;

    wire sample_end_up_down, sample_end_incr;
    reg sample_end_ld;
    reg [16:0]sample_end_q;

    grey_decode sample_end_decode(.clock(clock),.reset(reset),
                                                                        .a(sample_end_ctl[0]),.b(sample_end_ctl[1]),.up_down(sample_end_up_down),.ready_pulse(sample_end_incr)

    bounded_parameter sample_end_param(.clock(clock),.reset(reset),
                                                                                        .upper(20'b11111111111111111),.lower(0),
                                                                                        .q(sample_end_q),.ld(sample_end_ld),
                                                                                        .up_down(sample_end_up_down),.incr(sample_end_incr),
                                                                                        .value(sample_end));
    defparam sample_end_param.SIZEOF = 17;
    defparam sample_end_param.INCREMENT = 1024;

    wire sample_start_up_down, sample_start_incr, sample_start_ld;
    assign sample_start_ld = 0;

    grey_decode sample_start_decode(.clock(clock),.reset(reset),
                                                                        .a(sample_start_ctl[0]),.b(sample_start_ctl[1]),.up_down(sample_start_up_down),.ready_pulse(sample_sta

    bounded_parameter sample_start_param(.clock(clock),.reset(reset),
                                                                                        .upper(sample_end),.lower(0),
                                                                                        .q(0),.ld(sample_start_ld),
                                                                                        .up_down(sample_start_up_down),.incr(sample_start_incr),
                                                                                        .value(sample_start));
    defparam sample_start_param.SIZEOF = 20;
    defparam sample_start_param.INCREMENT = 1024;



    wire sustain_pos_up_down, sustain_pos_incr, sustain_pos_ld;
    assign sustain_pos_ld = 0;

    grey_decode sustain_pos_decode(.clock(clock),.reset(reset),
                                                                        .a(sustain_pos_ctl[0]),.b(sustain_pos_ctl[1]),.up_down(sustain_pos_up_down),.ready_pulse(sustain_pos_i

    bounded_parameter sustain_pos_param(.clock(clock),.reset(reset),
                                                                                        .upper(15'b111111111111111),.lower(0),
                                                                                        .q(0),.ld(sustain_pos_ld),
                                                                                        .up_down(sustain_pos_up_down),.incr(sustain_pos_incr),
                                                                                        .value(sustain_pos));
    defparam sustain_pos_param.SIZEOF = 15;
    defparam sustain_pos_param.INCREMENT = 1;

    wire sustain_length_up_down, sustain_length_incr, sustain_length_ld;
    assign sustain_length_ld = 0;

    grey_decode sustain_length_decode(.clock(clock),.reset(reset),
                                                                        .a(sustain_length_ctl[0]),.b(sustain_length_ctl[1]),.up_down(sustain_length_up_down),.ready_pulse(sust

    bounded_parameter sustain_length_param(.clock(clock),.reset(reset),
                                                                                        .upper(12'b111111111111),.lower(0),
                                                                                        .q(0),.ld(sustain_length_ld),
                                                                                        .up_down(sustain_length_up_down),.incr(sustain_length_incr),
                                                                                        .value(sustain_length));
    defparam sustain_length_param.SIZEOF = 12;
    defparam sustain_length_param.INCREMENT = 1;

    wire echo_delay_up_down, echo_delay_incr, echo_delay_ld;
    assign echo_delay_ld = 0;

    grey_decode echo_delay_decode(.clock(clock),.reset(reset),
                                                                        .a(echo_delay_ctl[0]),.b(echo_delay_ctl[1]),.up_down(echo_delay_up_down),.ready_pulse(echo_delay_incr)

    bounded_parameter echo_delay_param(.clock(clock),.reset(reset),
                                                                                        .upper(0),.lower(0),
                                                                                        .q(0),.ld(echo_delay_ld),
                                                                                        .up_down(echo_delay_up_down),.incr(echo_delay_incr),
                                                                                        .value(echo_delay));
    defparam echo_delay_param.SIZEOF = 11;
    defparam echo_delay_param.INCREMENT = 1;

    wire echo_decay_up_down, echo_decay_incr, echo_decay_ld;
    assign echo_decay_ld = 0;

    grey_decode echo_decay_decode(.clock(clock),.reset(reset),
                                                                        .a(echo_decay_ctl[0]),.b(echo_decay_ctl[1]),.up_down(echo_decay_up_down),.ready_pulse(echo_decay_incr)

    bounded_parameter echo_decay_param(.clock(clock),.reset(reset),
                                                                                        .upper(0),.lower(0),
                                                                                        .q(0),.ld(echo_decay_ld),
                                                                                        .up_down(echo_decay_up_down),.incr(echo_decay_incr),
                                                                                        .value(echo_decay));
    defparam echo_decay_param.SIZEOF = 4;
    defparam echo_decay_param.INCREMENT = 1;

    always @ (posedge clock) begin
            if(reset) begin
```

```
                        sample_end_ld <= 1;
                        sample_end_q <= 20'b11111111111111111111;
                end
                else begin
                        sample_end_ld <= 0;
                end
        end
endmodule
```

Listing 22: Verilog file parameters.v

```
/**
 * parameters Manages the parameters for the wumpus.
 */
module parameters(clock, reset,
                sample_length,
                sample_start_ctl, sample_start_ld, sample_start_q, sample_start,
                sample_end_ctl, sample_end_ld, sample_end_q, sample_end,
                window_hold_pos_ctl, window_hold_pos_ld, window_hold_pos_q, window_hold_pos,
                window_hold_length_ctl, window_hold_length_ld, window_hold_length_q, window_hold_length,
                echo_delay_ctl, echo_delay_ld, echo_delay_q, echo_delay,
                pitch_offset_ctl, pitch_offset_ld, pitch_offset_q, pitch_offset);
        input clock;
        input reset;

        //Sample length
        input [16:0] sample_length;


        //Control signals and output for sample_start
        //upper: sample_end
        //lower: 0
        //increment: 1024
        input [1:0] sample_start_ctl;
        input sample_start_ld;
        input [16:0] sample_start_q;
        output [16:0] sample_start;

        //Control signals and output for sample_end
        // lower: sample_start
        // upper: sample_length
        //increment: 1024
        input [1:0] sample_end_ctl;
        input sample_end_ld;
        input [16:0] sample_end_q;
        output [16:0] sample_end;

        //Control signals and output for window_hold_pos
        //lower: sample_start
        //upper: sample_end
        //increment: 1024
        input [1:0] window_hold_pos_ctl;
        input window_hold_pos_ld;
        input [16:0] window_hold_pos_q;
        output [16:0] window_hold_pos;

        //Control signals and output for window_hold_length
        //upper: 12'b111111111111
        //lower: 12'b0
        //increment: 16
        input [1:0] window_hold_length_ctl;
        input window_hold_length_ld;
        input [11:0] window_hold_length_q;
        output [11:0] window_hold_length;

        //Control signals and output for echo_delay
        //lower: 14'b0
        //upper: 14'b11111111111111
        //increment: 16
        input [1:0] echo_delay_ctl;
        input echo_delay_ld;
        input [13:0] echo_delay_q;
        output [13:0] echo_delay;


        //Control signals and output for pitch_offset
        //lower: 0
        //upper: 60
        //increment: 1
        input [1:0] pitch_offset_ctl;
        input pitch_offset_ld;
        input [11:0] pitch_offset_q;
        output [11:0] pitch_offset;

        /// Sample Start
        wire sample_start_ud;
        wire sample_start_ready;
        grey_decode sample_start_decode(.clock(clock),.reset(reset),.a(sample_start_ctl[0]),.b(sample_start_ctl[1]),
                                                .up_down(sample_start_ud),.ready_pulse(sample_start_ready));

        bounded_parameter sample_start_param(.clock(clock),.reset(reset),
                                .upper(sample_end),.lower(17'b0),
                                .q(sample_start_q),.ld(sample_start_ld),
                                .up_down(sample_start_ud),.incr(sample_start_ready),
                                .value(sample_start));
        defparam sample_start_param.SIZEOF = 17;
        defparam sample_start_param.INCREMENT = 1024;


        /// Sample End
        wire sample_end_ud;
        wire sample_end_ready;
        grey_decode sample_end_decode(.clock(clock),.reset(reset),.a(sample_end_ctl[0]),.b(sample_end_ctl[1]),
                                           .up_down(sample_end_ud),.ready_pulse(sample_end_ready));

        bounded_parameter sample_end_param(.clock(clock),.reset(reset),
                                      .upper(sample_length),.lower(sample_start),
                                      .q(sample_end_q),.ld(sample_end_ld),
                                      .up_down(sample_end_ud),.incr(sample_end_ready),
                                      .value(sample_end));
        defparam sample_end_param.SIZEOF = 17;
        defparam sample_end_param.INCREMENT = 1024;



        /// Sustain Position
        wire window_hold_pos_ud;
        wire window_hold_pos_ready;
        grey_decode window_hold_pos_decode(.clock(clock),.reset(reset),.a(window_hold_pos_ctl[0]),.b(window_hold_pos_ctl[1]),
                                           .up_down(window_hold_pos_ud),.ready_pulse(window_hold_pos_ready));

        bounded_parameter window_hold_pos_param(.clock(clock),.reset(reset),
                                      .upper(sample_end - 10),.lower(sample_start + 10),
                                      .q(window_hold_pos_q),.ld(window_hold_pos_ld),
                                      .up_down(window_hold_pos_ud),.incr(window_hold_pos_ready),
                                      .value(window_hold_pos));
        defparam window_hold_pos_param.SIZEOF = 17;
        defparam window_hold_pos_param.INCREMENT = 1024;



        /// Sustain Length
        wire window_hold_length_ud;
        wire window_hold_length_ready;
        grey_decode window_hold_length_decode(.clock(clock),.reset(reset),.a(window_hold_length_ctl[0]),.b(window_hold_length_ctl[1]),
                                           .up_down(window_hold_length_ud),.ready_pulse(window_hold_length_ready));

        bounded_parameter window_hold_length_param(.clock(clock),.reset(reset),
                                      .upper(12'b111111111111),.lower(12'b0),
                                      .q(window_hold_length_q),.ld(window_hold_length_ld),
                                      .up_down(window_hold_length_ud),.incr(window_hold_length_ready),
                                      .value(window_hold_length));
        defparam window_hold_length_param.SIZEOF = 12;
        defparam window_hold_length_param.INCREMENT = 16;

        wire echo_delay_ud;
        wire echo_delay_ready;
        grey_decode echo_delay_decode(.clock(clock),.reset(reset),.a(echo_delay_ctl[0]),.b(echo_delay_ctl[1]),
                                           .up_down(echo_delay_ud),.ready_pulse(echo_delay_ready));

        bounded_parameter echo_delay_param(.clock(clock),.reset(reset),
                                      .upper(13'b1111111111111),.lower(12'b0),
                                      .q(echo_delay_q),.ld(echo_delay_ld),
                                      .up_down(echo_delay_ud),.incr(echo_delay_ready),
                                      .value(echo_delay));
        defparam echo_delay_param.SIZEOF = 14;
        defparam echo_delay_param.INCREMENT = 16;

        wire pitch_offset_ud;
        wire pitch_offset_ready;
        grey_decode pitch_offset_decode(.clock(clock),.reset(reset),.a(pitch_offset_ctl[0]),.b(pitch_offset_ctl[1]),
                                           .up_down(pitch_offset_ud),.ready_pulse(pitch_offset_ready));

        bounded_parameter pitch_offset_param(.clock(clock),.reset(reset),
                                      .upper(6'd59),.lower(6'd0),
                                      .q(pitch_offset_q),.ld(pitch_offset_ld),
                                      .up_down(pitch_offset_ud),.incr(pitch_offset_ready),
                                      .value(pitch_offset));
        defparam pitch_offset_param.SIZEOF = 6;
        defparam pitch_offset_param.INCREMENT = 1;


endmodule
```

Listing 23: Verilog file serial fsm.v

```
/**
 * serial_fsm - Asynchronous 8 bit serial receiver, 1 start bit, one stop bit.
 * Currently designed for 31250 baud rate on a 27MHz clock.
 * TODO - Make baud rate and clock rate parameterized.
 * @author mmt
 *
 * @input clock System clock.
 * @input reset Syncronous reset of FSM.
 * @input rx Input wire with serial signal.
 *
 * @output [7:0]byte Serial word coming out.
 * @output ready_pulse Single clock cycle pulse indicating new output byte.
 *
 * @debug  read High the cycle after the receiver reads from the rx line.
 * @debug   counter Counter output from delay module instance.
 * @debug   [3:0]bit Which bit in the byte the receiver is writing to.
 * @debug   [1:0]state The state of the FSM.
 *
 */
module serial_fsm(clock,reset,rx,byte,ready_pulse,read,
                    bit,state,counter,state_change);
         parameter BAUD_RATE = 31250;
         parameter CLOCK_RATE = 40500000;
         parameter BAUD_RATE_SIZEOF = 15;
         parameter CLOCK_RATE_SIZEOF = 26;

    input clock;
    input reset;
    input rx;
    output [7:0] byte;
    output ready_pulse;
    output read;
    output [9:0] counter;
    output state_change;

    output [3:0] bit;
    output [1:0] state;

    reg state_change;

    //Four states of the FSM
    //WAIT and READ actually 8 separate states, one for each value of bit.
    parameter STOPPED = 0; //Waiting for a stop bit
    parameter WAIT   = 1; //Wait for one baud period
    parameter READ   = 2; //Read a bit from the rx line
    parameter READY = 3; //Completed reading a serial word

    reg [1:0] state; // 2 bits to store the above states
    reg [1:0] old_state; // state from previous cycle

    reg short;
    reg [3:0] bit; //the bit of the serial word, 4 bits wide to include start and stop bit

    reg ready_pulse;
    wire read;

    reg [9:0] word; //the full serial line output including start and stop bit
    wire [7:0] byte;
    assign byte [7:0] = word [9:1]; //wire only the actually serial message to the output

    wire wait_reset;
    reg wait_disable;

    assign wait_reset = reset | wait_disable; //reset the delay circuit on

    wire [9:0] counter;
    //Generates a pulse on read every Baud Period, or half that if short is high
    //Short is high only when reading the start bit into word, to place the sample times in the center of data on the rx line.
    delay delay_circuit(.clock(clock),.reset(wait_reset),.pulse(read),.short(short),.counter(counter));
         defparam delay_circuit.DELAY1_COUNT = CLOCK_RATE / BAUD_RATE / 2;
         defparam delay_circuit.DELAY2_COUNT = CLOCK_RATE / BAUD_RATE;
         defparam delay_circuit.COUNT_SIZEOF = CLOCK_RATE_SIZEOF - BAUD_RATE_SIZEOF;

    always @ (posedge clock) begin
        if(reset) begin
                state <= STOPPED; // state is stopped
                state_change <= 1;//force state outputs
                //other initial values
                short <= 1;
                word   <= 0;
                bit    <= 0;
                wait_disable <= 1;
                ready_pulse <= 0;
        end
        else begin

        case(state)
                STOPPED:
                        if(~rx) //start bit! get going
                                state <= WAIT;
                WAIT:
                        if(read) //delay circuit indicates a baud count has gone by
                                state <= READ;
                READ:
                        if(bit >= 9) //stop bit has been reached
                                state <= READY; //output is ready
                        else
                                state <= WAIT; //wait to read the next bit
                READY:
                             state <= STOPPED; //full cycle complete
        endcase

        if(state != old_state) //if the state has changed since the last clock cycle
                               //its worth noting this adds a delay of a clock cycle for the
                               // change of the state to be noticed, but such timing inefficiencies are not
                               // much of a concession at serial baud rates.
        case(state)
                STOPPED: begin
                        short <= 1;
                        bit    <= 0;
                        wait_disable <= 1;
                    ready_pulse <= 0;
                end
                WAIT:
                    wait_disable <= 0;

                READ: begin
                    bit <= bit + 1;
                    word[bit] <= rx;
                    wait_disable <= 1;
                    short <= 0;
                end
                READY:
                    ready_pulse <= 1;
        endcase
        old_state[1:0] <= state[1:0];
        end //end if(~reset)
        end
endmodule
```

Listing 24: Verilog file synchronize.v

```
// pulse synchronizer
module synchronize(clk,in,out);
  parameter WIDTH=1;
  parameter NSYNC = 2;  // number of sync flops.  must be >= 2
  input clk;
  input [WIDTH-1:0]in;
  output [WIDTH-1:0]out;

  reg [(WIDTH-1)*(NSYNC-2):0]sync;
  reg [WIDTH-1:0]out;

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

Listing 25: Verilog file taylor.v

```
module taylor(clock,reset, period_out, pressed_out, sample_start,sample_end,window_hold_pos,window_hold_length,
        frame_n,frame_n_plus,frame_n_minus,ready_frameprep,n_out,delta_out,going_out,delta_new,n_new,going_new,
        tframe_out, taylor_done);

//NOTE this has been formatted to fit your page. Normally my indents don't go inward like this.
        input clock;
        input reset;
        input [15:0] period_out; //the period of the sample, with the 5 MSB being integer.
        input pressed_out;

        input [16:0] sample_start;        //start of sample read-off from controller
        input [16:0] sample_end;                          //end of sample read-off from controller
        input [16:0] window_hold_pos;     //position of the window/vowel hold, from controller
        input [11:0] window_hold_length; //length of the window/vowel hold, from controller

        input signed [15:0] frame_n;             //frame at current n
        input signed [15:0] frame_n_plus;        //frame at next n
        input signed [15:0] frame_n_minus;       //frame at previous n
        input ready_frameprep;

        input [16:0] n_out;                      //current address.
        input [10:0] delta_out;                  //current offset to virtual frame we are creating.
        input going_out;                         //output from the variable ram to determine if the
                                                 //note is 'going' (running)

        output [10:0] delta_new;           //new offset to be given to notemem
        output [16:0] n_new;               //new address to be given to notemem
        output going_new;                  //new determination if note is going. turns to 0 if note has
                                         //reached end,1 if pressed_out==1 & n_out is sample_start

        output signed [15:0] tframe_out;         //output frame
        output taylor_done;                      //the taylor series is done computing a note-frame.
                                                 //(signals accumulator and notemem)



        reg [16:0] n_new;
        reg [10:0] delta_new;
        reg [15:0] tframe_out;
        reg taylor_done;
        reg going_new;

always @ (posedge clock) begin
        if(reset) begin
                n_new <= 0;
                delta_new <= 0;
                tframe_out <= 0;
                taylor_done <= 0;
                going_new <= 0;
        end
        else begin
                if(ready_frameprep) begin
```

```
                        if(pressed_out) begin
                                if(going_out) begin      //loop
        if(n_out<window_hold_pos) begin
                n_new <= n_out + ((delta_out+period_out)>>11);
                going_new <= 1;
        end
        else if (n_out>sample_end) begin
                n_new <= sample_start;
                going_new <= 0;
        end
        else begin
                n_new <= n_out - window_hold_length;
                going_new <= 1;
        end
        delta_new <= delta_out + period_out;
        taylor_done <= 1;
        tframe_out <= frame_n + (((frame_n_minus-frame_n)*delta_out)>>11);
                                end
                                else begin        //starting out (pressed but not yet going)
        going_new <= 1;
        n_new <= sample_start;//n_out + ((delta_out+period_out)>>11);//sample_start;
        delta_new <= delta_out + period_out;
        taylor_done <= 1;
        tframe_out <= frame_n + (((frame_n_minus-frame_n)*delta_out)>>11);
                                end
                        end
                        else begin //no longer pressed
                                if(going_out) begin      //finish
        if(n_out<sample_end) begin
                n_new <= n_out + ((delta_out+period_out)>>11);
                going_new <= 1;
        end
        else begin
                n_new <= sample_start;
                going_new <= 0;
        end
        delta_new <= delta_out + period_out;
        taylor_done <= 1;
        tframe_out <= frame_n + (((frame_n_minus-frame_n)*delta_out)>>11);
                                end
                                else begin //not going, not pressed. Reset values.
        going_new <= 0;
        n_new <= sample_start;
        delta_new <= 0;
        taylor_done <= 1;
        tframe_out <= 0;
                                end
                        end
                end
                else if(taylor_done) taylor_done <= 0;
        end
end
endmodule
```

Listing 26: Verilog file visualizer.v

```
/**
 * Visualizer
 * Include visualizer_rectangle and waveform_background and keyboard_bitmap at the bottom.
 *
 * Generates a 800x600 pixel stream based on the indexes provided by hcount and vcount.
 */
module visualizer (clock,reset,
                  hcount,vcount,hsync,vsync,blank,
                  vhsync,vvsync,vblank,pixel,
                  sample_start,sample_end,window_hold_pos,echo_delay,window_hold_length,pitch_offset,
                  sample_length,
                  record,audio,new_frame,

                  wf_state);
        input [16:0] sample_start;
        input [16:0] sample_end;
        input [16:0] sample_length;
        input [16:0] window_hold_pos;
        input [13:0] echo_delay;
        input [11:0] window_hold_length;
        input [5:0]  pitch_offset;

        parameter SCREEN_WIDTH = 800;
        parameter SCREEN_HEIGHT = 600;

        input clock;    // 40MHz clock
        input reset;            // 1 to initialize module
        input [10:0]  hcount;   // horizontal index of current pixel (0..1023)
        input [9:0] vcount; // vertical index of current pixel (0..767)
        input hsync;            // XVGA horizontal sync signal (active low)
        input vsync;            // XVGA vertical sync signal (active low)
        input blank;            // XVGA blanking (1 means output black pixel)
        input [15:0] audio;     // incoming audio data
        input record;           // system is recording
        input new_frame;        // new audio frame available pulse


        output vhsync;
        output vvsync;
        output vblank;
        output [2:0] pixel;

        output [1:0] wf_state;

        reg [2:0] pixel;

        assign vhsync = hsync;
        assign vvsync = vsync;
        assign vblank = blank;

        //Calculate Screen positions from parameters
        reg [9:0] sample_start_x;

        reg [9:0] sample_end_x;

        reg [9:0] window_hold_pos_x;

        reg [9:0] window_hold_length_x;
        reg [9:0] echo_delay_x;
        reg [9:0] sample_length_x;


        /*
         * Screen Layout:
         * 10 pixel top margin
         * 100 pixel left margin to text
         * Text left aligned
         * 188 pixel left margin to visualizer
         * 300 top margin to keyboard
         */
        ////////
        // Text Elements
        ////////
        wire [2:0] st_pixel;
        char_string_display start_text(.vclock(clock),.hcount(hcount),.vcount(vcount),.pixel(st_pixel),.cstring("Start"),
                                                                           .cx(11'd100),.cy(10'd20));
        defparam start_text.NCHAR = 5;
        defparam start_text.NCHAR_BITS = 3;
        defparam start_text.COLOR = 3'b010;


        wire [2:0] spt_pixel;
        char_string_display sustain_text(.vclock(clock),.hcount(hcount),.vcount(vcount),.pixel(spt_pixel),.cstring("Sustain"),
                                                                           .cx(11'd100),.cy(10'd168));

        defparam sustain_text.NCHAR = 7;
        defparam sustain_text.NCHAR_BITS = 3;
        defparam sustain_text.COLOR = 3'b001;

        wire [2:0] et_pixel;
        char_string_display end_text(.vclock(clock),.hcount(hcount),.vcount(vcount),.pixel(et_pixel),.cstring("End"),
                                                                           .cx(11'd100),.cy(10'd180));
        defparam end_text.NCHAR = 3;
        defparam end_text.NCHAR_BITS = 2;
        defparam end_text.COLOR = 3'b100;

        wire [2:0] ect_pixel;
        char_string_display echo_text(.vclock(clock),.hcount(hcount),.vcount(vcount),.pixel(ect_pixel),.cstring("Echo Start"),
                                                                           .cx(11'd100),.cy(10'd192));

        defparam echo_text.NCHAR = 10;
        defparam echo_text.NCHAR_BITS = 4;
        defparam echo_text.COLOR = 3'b101;

        wire [2:0] tt_pixel;
        char_string_display tune_text(.vclock(clock),.hcount(hcount),.vcount(vcount),.pixel(tt_pixel),.cstring("Tune"),
                                                                           .cx(11'd10),.cy(10'd380));

        defparam tune_text.NCHAR = 4;
        defparam tune_text.NCHAR_BITS = 3;
        defparam tune_text.COLOR = 3'b010;



        ////////
        //      Waveform
        ////////


        wire [2:0]wfbg_pixel;
        waveform_background my_waveform_background(.hcount(hcount),.vcount(vcount),.pixel(wfbg_pixel),.sample_start_x(sample_start_x+10),.sample_end_x(sample_end_x));
        defparam my_waveform_background.HEIGHT=128;
        defparam my_waveform_background.WIDTH=512;
        defparam my_waveform_background.X = 188;
        defparam my_waveform_background.Y = 40;



        wire [2:0] wf_pixel;
        waveform_graphic waveform_g(.clock(clock),.reset(reset),
                                                    .audio(audio[15:0]),
                                                    .record(record),
                                                    .new_frame(new_frame),
                                                    .hcount(hcount),.vcount(vcount),
                                                    .pixel(wf_pixel),
                                                    .sample_length_x(sample_length_x));
```

```
        defparam waveform_g.X_POSITION = 188;
        defparam waveform_g.Y_POSITION = 40;

        ////////
        //      Keyboard Element
        ////////
        wire [2:0] kbbg_pixel;
        visualizer_rectangle keyboard_bg(.x(11'd750),.y(10'd300),.hcount(hcount),.vcount(vcount),.pixel(kbbg_pixel));
        defparam keyboard_bg.HEIGHT=80;
        defparam keyboard_bg.WIDTH=2;
        defparam keyboard_bg.COLOR =3'b111;

        wire [2:0] keyboard_pixel;
        keyboard_bitmap keyboard(.clock(clock),.x(11'd50),.y(10'd300),.hcount(hcount),.vcount(vcount),.pixel(keyboard_pixel));

        ////////
        //      Vertical Bar Elements
        ////////

        //peaks above the graphic
        wire [2:0] sb_pixel;
        visualizer_rectangle start_bar(.x(sample_start_x),.y(10'd30),.hcount(hcount),.vcount(vcount),.pixel(sb_pixel));
        defparam start_bar.HEIGHT=138;
        defparam start_bar.WIDTH=10;
        defparam start_bar.COLOR=3'b010;

        //peaks below the graphic
        wire [2:0] eb_pixel;
        visualizer_rectangle end_bar(.x(sample_end_x),.y(10'd40),.hcount(hcount),.vcount(vcount),.pixel(eb_pixel));
        defparam end_bar.HEIGHT=150;
        defparam end_bar.WIDTH=10;
        defparam end_bar.COLOR=3'b100;



        wire [2:0] spb_pixel;
        visualizer_rectangle window_hold_pos_bar(.x(window_hold_pos_x),.y(10'd168),.hcount(hcount),.vcount(vcount),.pixel(spb_pixel));
        defparam window_hold_pos_bar.HEIGHT=10;
        defparam window_hold_pos_bar.WIDTH=10;
        defparam window_hold_pos_bar.COLOR=3'b001;

        wire [2:0] slb_pixel;
        visualizer_rectangle window_hold_length_bar(.x(window_hold_length_x),.y(10'd168),.hcount(hcount),.vcount(vcount),.pixel(slb_pixel));
        defparam window_hold_length_bar.HEIGHT=10;
        defparam window_hold_length_bar.WIDTH=10;
        defparam window_hold_length_bar.COLOR=3'b010;

        wire [2:0] ecb_pixel;
        visualizer_rectangle echo_bar(.x(echo_delay_x),.y(10'd192),.hcount(hcount),.vcount(vcount),.pixel(ecb_pixel));
        defparam echo_bar.HEIGHT=10;
        defparam echo_bar.WIDTH=10;
        defparam echo_bar.COLOR=3'b101;

        //This lookup table maps the pitch offset to the pixel of the key it represents.
        wire [10:0] tune_bar_x;
        keyboard_key_pixel_lut tune_position(.clk(clock),.addr(pitch_offset),.dout(tune_bar_x));

        wire [2:0] tb_pixel;
        visualizer_rectangle tune_bar(.x(11'd45 + tune_bar_x),.y(10'd381),.hcount(hcount),.vcount(vcount),.pixel(tb_pixel));
        defparam tune_bar.HEIGHT=10;
        defparam tune_bar.WIDTH=10;
        defparam tune_bar.COLOR=3'b010;

        always @ (posedge clock) begin

                //translate all fo the incoming parameters into their pixel equivalents.
                sample_start_x      <= (sample_start >>8) + 178;
                sample_end_x        <= (sample_end >>8) + 188;
                window_hold_pos_x <= (window_hold_pos >>8) + 183;
                window_hold_length_x    <= (window_hold_pos >>8) + 183 + (window_hold_length >>8);
                echo_delay_x              <= (echo_delay >>8) + (sample_start >>8) + 183;
                sample_length_x     <= (sample_length>>8);

                //bitwise or all the elements, except for the waveform background, which provides color to the shape
                // of the wf_pixel
                pixel <= ((wfbg_pixel & wf_pixel) | keyboard_pixel | kbbg_pixel |
                                        tb_pixel |  ecb_pixel | sb_pixel | eb_pixel | spb_pixel | slb_pixel |
                                        tt_pixel | ect_pixel | st_pixel | et_pixel | spt_pixel);
        end

endmodule


/**
 * visualizer_rectangle
 * Outputs a rectangle on the screen at the location provided by x,y as a pixel stream indexed by hcount vcount
 */
module visualizer_rectangle(x,y,hcount,vcount,pixel);
   parameter HEIGHT = 64;        //the height
   parameter WIDTH = 64;         //the width
   parameter COLOR = 3'b111;     //the color

   input  [10:0] x,hcount;
   input  [9:0] y,vcount;
   output [2:0] pixel;

   reg [2:0] pixel;
   always @ (x or y or vcount or hcount) begin
      if ((hcount >= x && hcount < (x+WIDTH)) &&
          (vcount >= y && vcount < (y+HEIGHT)))
          pixel = COLOR;
      else pixel = 0;    //black outside
   end

endmodule


/**
 * Generate a color coded rectangle based on the start and end of the sample.
 */
module waveform_background(hcount,vcount,sample_start_x,sample_end_x,pixel);
   //Parameterized position and dimensions on the screen
   parameter HEIGHT = 128;
   parameter WIDTH = 512;
        parameter X = 0;
        parameter Y = 0;


   input  [10:0] sample_start_x,sample_end_x,hcount;
   input  [9:0] vcount;
   output [2:0] pixel;

   reg [2:0] pixel;

   always @ (sample_start_x or sample_end_x or vcount or hcount) begin
      if ((hcount >= X && hcount < (X+WIDTH)) &&
          (vcount >= Y && vcount < (Y+HEIGHT)))
        //If we're left of the start pixel, be green, right of the stop pixel be red, else black
          pixel = (hcount < sample_start_x) ? 3'b010 : ((hcount > sample_end_x) ? 3'b100 : 3'b111);
      else pixel = 0;
   end

endmodule


/**
 * keyboard_bitmap - Outputs the pixel stream of a keyboard image.
 */
module keyboard_bitmap(clock,x,y,hcount,vcount,pixel);
        parameter HEIGHT = 80;
        parameter WIDTH = 700;
        input clock;

   input  [10:0] x,hcount;
   input  [9:0] y,vcount;
   output [2:0] pixel;

        reg [2:0] pixel;

        reg [10:0] old_hcount;
        reg [9:0] old_vcount;

        reg [15:0] addr;
        wire [7:0] dout;



        keyboard_sprite sprite(.clk(clock),.addr(addr),.dout(dout));
        always @ (posedge clock) begin
                if(hcount != old_hcount || vcount != old_vcount) begin
                        if(hcount + 3 == x && vcount == y) // reset the address when we get to top left

                                addr <= 1;
                        else if(hcount + 1  >= x && hcount +1   < x + WIDTH && vcount >= y && vcount  <= y + HEIGHT) begin
                                addr <= addr + 1; //each valid position we increment the address
                                pixel <= dout[2:0];
                        end
                        else
                                pixel <= 0;
                end
                old_hcount <= hcount;
                old_vcount <= vcount;
        end
endmodule
```

Listing 27: Verilog file waveform_graphic.v

```
/**
 * waveform_graphic - generate a pixel stream visualization of an audio sample.
 *
 * @parameter HEIGHT The height of the visualization
 * @parameter WIDTH The width of the visualization
 * @parameter X_POSITION The horizontal position of the graphic on the screen.
 * @parameter Y_POSITION The vertical position of the graphic on the screen.
 *
 * @input clock The video clock.
 * @input reset A syncronous reset.
 * @input signed [15:0] audio The audio stream to be sampled.
 * @input record Indicates
 *
 *
 */
module waveform_graphic(clock, reset, audio, record, new_frame, hcount,
                        vcount, pixel, sample_length_x,
                                                        state, we, addrin, din);
        parameter HEIGHT=128;
        parameter WIDTH=512;
        parameter X_POSITION=0;
        parameter Y_POSITION=0;
        parameter x = X_POSITION;
        parameter y = Y_POSITION;
        input clock;
        input reset;
        input signed [15:0] audio;
        input record;
        input new_frame;
        input [10:0] hcount;
        input [9:0]  vcount;
        output [2:0] pixel;
        input [9:0] sample_length_x; //length of the sample in pixels, not absolute to the screen

        output [1:0] state;
        output we;
        output [8:0]addrin;
        output [13:0]din;

        parameter WAIT = 0;
        parameter ACCUMULATE = 1;
        parameter STORE = 2;
        parameter STEP = 3;

        reg [2:0] pixel;

        reg [1:0] state;
        reg [7:0] frame;

        //watching for state and sequencer transitions
        reg [1:0] old_state;
        reg [7:0] old_frame;


        //two port memory, one for writing (syncronous) and one for reading (asyncronous)
        reg [13:0] din;
        wire [13:0] dout;
        reg we;
        reg [8:0] addrin, addrout;
        reg [8:0] length;
        reg signed [22:0] sum_pos, sum_neg;
        waveform_graphic_memory my_graphic_memory(
        .addra(addrin),
        .addrb(addrout),
        .clka(clock),
        .clkb(clock),
        .dina(din),
        .doutb(dout),
        .wea(we));

        // Writing logic
        // This FSM/Sequencer finds the average positive and average negative values of the
        //      incoming audio signal, in groups of 256.
        // The values are written to the memory.
        always @ (posedge clock) begin
                if(reset) begin
                        state <= WAIT;
                        frame <= 0;
                end
                else begin
                //State transitions
                case(state)
                WAIT:
                        if(record)
                                state <= ACCUMULATE;

                ACCUMULATE:
                        if(~record || frame == 8'hFF)
                                state <= STORE; //accumulation is done or cut short
                        else if(new_frame)
                                frame <= frame + 1; //otherwise, wait for another frame
                STORE:
                        if(record)
                                state <= STEP;  //step to the next address, and complete write operation
                        else
                                state <= WAIT;
                STEP: begin
                        frame <= 0; //reset the frame for the next accumulate
                        if(record)
                                state <= ACCUMULATE;
                        else
                                state <= WAIT;
                end
                endcase

                if(state != old_state || frame != old_frame)
                case(state)
                WAIT: begin
                        addrin <= 9'b0;
                        we <= 0;
                        sum_pos <= 0;
                sum_neg <= 0;
            end
            ACCUMULATE: begin
                if(audio > 0)
                    sum_pos <= sum_pos + audio;      //write the positive data one sum
                else
                    sum_neg <= sum_neg + audio;   //and the negative to the other
                we <= 0;
                //addr and data latched
            end
            STORE: begin
                din <= {sum_pos[22:16],sum_neg[22:16]}; //bit shift , note that sign is preserved)
                we <= 1;
                sum_pos <= 0;
                sum_neg <= 0;
            end
            STEP: begin
                we <= 0; //complete write
                addrin <= addrin + 1; //increment address
                sum_pos <= 0;
                sum_neg <= 0;
            end
            endcase


            end
            old_state <= state;
            old_frame <= frame;
        end






        wire signed [6:0] top_data = dout[13:8];
        wire signed [6:0] bot_data = dout[7:0];


        always @ (hcount or vcount) begin
            addrout = hcount - x;

            if(hcount >= x && hcount < x + WIDTH && hcount < x + sample_length_x) begin
                if(vcount >= y && vcount <= y + HEIGHT/2)
                    pixel = (HEIGHT/2  + y) > (top_data + vcount) ? 3'b000 : 3'b111; // output black if the pixel is less than the positive sum

                else if(vcount > y + HEIGHT/2 && vcount < y + HEIGHT)
                    pixel = (HEIGHT/2 + y) < (bot_data + vcount) ? 3'b000 : 3'b111; // output black if the pixel is greater than the negative sum

            end
            else
                pixel = 3'b000;
        end



endmodule
```

Listing 28: Verilog file xvga.v

```
///////////////////////////////////////////////////////////////////////////////
//
// xvga: Generate XVGA display signals (800 x 600 @ 60Hz)
//
///////////////////////////////////////////////////////////////////////////////

module xvga(vclock,hcount,vcount,hsync,vsync,blank);
   input vclock;
   output [10:0] hcount;
   output [9:0] vcount;
   output        vsync;
   output        hsync;
   output        blank;

   reg     hsync,vsync,hblank,vblank,blank;
   reg [10:0]    hcount;    // pixel number on current line
   reg [9:0] vcount;     // line number

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   wire       hsyncon,hsyncoff,hreset,hblankon;
   assign     hblankon = (hcount == 799);
   assign     hsyncon = (hcount == 839);
   assign     hsyncoff = (hcount == 967);
   assign     hreset = (hcount == 1055);

   // vertical: 806 lines total
   // display 768 lines
   wire       vsyncon,vsyncoff,vreset,vblankon;
   assign     vblankon = hreset & (vcount == 599);
   assign     vsyncon = hreset & (vcount == 600);
   assign     vsyncoff = hreset & (vcount == 604);
   assign     vreset = hreset & (vcount == 627);
   // 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
```