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, until 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 greyc ode knobs for ease of use.

![Figure 2: Mockup of the final user interface.](image)

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 parametrized 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 minimum 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
triggers accumulator to add the output $t_{frame_out}$ to its 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.

*window* _hold pos_, at which point it cycles over the previous *window* _hold length_ samples. When the note is no longer pressed (regardless of whether 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 concurrently, with a new note being
evaluated every time \texttt{taylor\_done} signals it is done with the previous one, until all 8 have been played. Variables are written by \texttt{taylor} to the ram address of the current note, whenever \texttt{taylor} signals it is complete, and are read out to \texttt{frameprep} and \texttt{taylor}. Parameters are written by \texttt{note\_control}, into whatever note it chooses. \texttt{note\_control}, in turn, is signaled with notes that are finished.

2.1.3 Linemem

The \texttt{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 \texttt{visualizer} and therefore unimportant.

2.1.4 Frameprep

The \texttt{frameprep} module contains the sample memory, which is 98304 samples long and 16 bits wide, due to BRAM memory constraints. On the \texttt{notemem\_ready} signal, \texttt{frameprep} reads the frames at \(n, n-1, n+1\) and stores them into \texttt{frame\_n, frame\_n\_minus, and frame\_n\_plus} respectively. After the four clock cycles required for this to happen have passed, \texttt{ready\_frameprep} is set high for a clock cycle to signal \texttt{taylor} to start it’s calculations. For unpressed notes, \texttt{frameprep} still prepares sample frames. It is \texttt{taylor} that sets the output to 0 for these notes.

2.1.5 Accumulator

The \texttt{accumulator} module is quite simple. Upon the signal \texttt{taylor\_done}, \texttt{accumulator} adds the current output from \texttt{taylor} to it’s running sum. This running sum is cleared upon each \texttt{new\_frame}, allowing a new polyphonic output frame to be built. Since there can be up to 8 notes playing concurrently, \texttt{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 \texttt{echo} module uses a BRAM to store the values of the previous \texttt{echo\_time} frames from the output of \texttt{notes}, and adds the bit shifted least recent frame in it’s memory to \texttt{data\_in} to create \texttt{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 \texttt{taylor\_decay} and added to \texttt{data\_in}.

The current value of \texttt{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 \texttt{sumframe\_out} module in \texttt{notes}, as both as signalled on \texttt{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.
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.
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.
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\_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, sample\_end, pitch\_offset, echo\_time, window\_hold\_pos, and window\_hold\_length are set to the default values, which are documented in Table 2. The record logic pulls the sample\_start, sample\_end, pitch\_offset, echo\_time, window\_hold\_pos, and window\_hold\_length lines high for two clock cycles. This delay allows 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 asynchronous 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.

### Table 1: Boundary Conditions for Parameters

<table>
<thead>
<tr>
<th>Value</th>
<th>Upper</th>
<th>Lower</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample_start</td>
<td>sample_end</td>
<td>0x0</td>
</tr>
<tr>
<td>sample_end</td>
<td>0x1FFFF</td>
<td>sample_start</td>
</tr>
<tr>
<td>window_hold_pos</td>
<td>sample_end-window_hold_length</td>
<td>sample_start</td>
</tr>
<tr>
<td>window_hold_length</td>
<td>0xFFF</td>
<td>0x200</td>
</tr>
<tr>
<td>pitch_offset</td>
<td>0x3C</td>
<td>0x0</td>
</tr>
<tr>
<td>echo_time</td>
<td>0xFFF</td>
<td>0x0</td>
</tr>
</tbody>
</table>

### Table 2: Values Loaded Into Parameters after Record

<table>
<thead>
<tr>
<th>Bus</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample_start</td>
<td>0x0</td>
</tr>
<tr>
<td>sample_end</td>
<td>sample_length</td>
</tr>
<tr>
<td>window_hold_pos</td>
<td>sample_start</td>
</tr>
<tr>
<td>window_hold_length</td>
<td>0x200</td>
</tr>
<tr>
<td>pitch_offset</td>
<td>0x18</td>
</tr>
<tr>
<td>echo_time</td>
<td>0xFFF</td>
</tr>
</tbody>
</table>
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 Synchronous 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 equaling 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 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 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, sampleLength, 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 its 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 difficult 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 interdependency 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 difficult 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 difficulty 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 recognize 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

```verilog
module accumulator(clock, reset, new_frame, taylور done, tframe_out, sum_out, sum_done);
  input clock;
  input reset;
  input new_frame; // signals the accumulator to start a new accumulation
  input taylор done; // signals that there is new data to be added to the accumulator
  input signed [15:0] tframe_out; // data coming from taylор
  output signed [17:0] sum_out; // accumulated frame two bits wider than tfraim'e_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 sum_frame_out = running_sum[19:2];
always @ (posedge clock) begin
  if (reset | new_frame) begin
    running_sum = 0;
  end else begin
    if (taylور done) begin
      running_sum = running_sum + tframe_out;
    end
  end
endmodule
```

Listing 2: Verilog file audio.v

```verilog
module audio (clock, reset, audio_in_data, audio_out_data, ready, new_frame, audio_reset_b, ac97_data_out, ac97_data_in, ac97_sync, 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_data_out;
  input ac97_data_in;
  output ac97_sync;
  input ac97_bit_clock;
  wire [4:0] volume;
  wire source;
  assign volume = 4'd44; // 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(trready, command_address, command_data, command_valid, left_in_data, 1'b1, right_in_data, 1'b1, left_out_data,
```
right\_data = ac97\_data\_out; ac97\_data\_in = ac97\_data\_in; ac97\_synch, ac97\_bi\_clock);
synchronize syncread, clk(clock), in(ready), out(ready);
level - pulse newframe, clock(clock), reset(reset), level(ready), pulse(newframe);
ac97commands cmd\_out, cmd\_address, cmd\_data, command\_valid, volume, source;

assign left\_out\_data = \{ audio\_out\_data, 4'h00\};
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,
    // ac97\_data\_in, ac97\_data\_out, ac97\_bi\_clock);
output ready;
input [7:0] command\_address;
input [15:0] command\_data;
input command\_valid;
input [19:0] left\_data, right\_data;
output left\_in\_data, right\_in\_data;
input ac97\_data\_in;
input ac97\_bi\_clock;
output ac97\_data\_out;
reg ready;
reg ac97\_data\_out;
reg ac97\_bi\_clock;
reg [7:0] bit\_count;
reg [19:0] cmd\_adr;
reg [19:0] cmd\_data;
reg [19:0] left\_data, right\_data;
reg [19:0] lcmd\_data, lright\_data;
reg [19:0] left\_in\_data, right\_in\_data;

initial begin
ready <= 'b0;
// synthesis attribute init of ready is "0";
ac97\_data\_out <= 'b0;
// synthesis attribute init of ac97\_data\_out is "0";
ac97\_synch <= 'b0;
// synthesis attribute init of ac97\_synch is "0";
bit\_count <= 8'h00;
// synthesis attribute init of bit\_count is "0000";
left\_data <= 'b0;
// synthesis attribute init of left\_data is "00000000";
left\_valid <= 'b0;
// synthesis attribute init of left\_valid is "0";
right\_data <= 'b0;
// synthesis attribute init of right\_data is "0";
right\_valid <= 'b0;
// synthesis attribute init of right\_valid is "0";
end
always @posedge ac97\_bi\_clock begin
// Generate the sync signal
if (bit\_count == 255) ac97\_synch <= 'b1;
if (bit\_count == 15) ac97\_synch <= 'b0;
if (bit\_count == 128) ac97\_synch <= 'b1;
if (bit\_count == 2) ac97\_synch <= 'b0;
// Generate the ready signal
if (bit\_count == 128) ready <= 'b1;
if (bit\_count == 2) ready <= '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
  cmd\_adr <= [command\_address, 12'h0000];
  cmd\_data <= [command\_data, 4'b0];
  cmd\_valid <= command\_valid;
  left\_data <= left\_data;
  left\_valid <= left\_valid;
  // Latch user data at the end of each frame. This ensures that the
  // first frame after reset will be empty
end

```verilog
module a97_commands (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 <= 4'0; // synthesis attribute init of command is "0";
  command_valid <= 1'0; // 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)
    4'0: // Read ID
      begin
        command <= 24'h0000;
        command_valid <= 1'b1;
      end
    ```
Listing 3: Verilog file bounded parameter.v

```verilog
module bounded_parameter(clk, 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 clk;
  // 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 (+INC or -INC)
  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 clk) 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;
      end
    end
  end
endmodule
```

// 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 Synchronous 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.

// the bounded parameter in Verilog

// usage example

// module bounded_parameter(clk, reset, upper, lower, q, up_down, incr, ld, value);

// always @(posedge clk) begin

//   if (reset) begin
//     value <= 0;
//   end

//   else begin

//     if (ld) begin

//       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

//       if (up_down)
//         value <= (value + INCREMENT) > upper ? upper : value + INCREMENT;

//     end

//   end

// end
```
Listing 4: Verilog file control.v

module control(clock, reset, new_name, sample/length, record, mid/wx, sample/start, sample/end, sustain/position, sustain/length, echo/delay, pitch/offset, loop, sample/start, sample/end, sustain/position, sustain/length, echo/delay, pitch/offset, // sample/start, sample/end, sustain/position, sustain/length, echo/offset, finished, finished/not/finish, note/ready, not/finish, period, velocity, pressed, // Debug outputs serial/byte, serial/ready, output serial/length, /@input [16:0] sample/length; // The length of the recording. @input [1:0] sample/position; // Rotary encoder knob input for sample/position. @input [1:0] sustain/position; // Rotary encoder knob input for sustain/position. @input [1:0] sustain/length; // Rotary encoder knob input for sustain/length. @input [1:0] pitch/offset; // Rotary encoder knob input for pitch/offset. @input [1:0] echo/delay; // Rotary encoder knob input for echo/delay. @input clock System clock. @input new_name Pulse to indicate a new audio sample available. @input record A high indicates the system is currently recording. @input mid/wx A mid serial line input (normally high). @input [1:0] sample/start Rotary encoder knob input for sample/start. @input [1:0] sustain/position Rotary encoder knob input for sustain/position. @input [1:0] sustain/length Rotary encoder knob input for sustain/length. @input [1:0] pitch/offset Rotary encoder knob input for pitch/offset. @input [1:0] echo/delay 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. @input module control(clock, reset, new_name, sample/length, record, mid/wx, sample/start, sample/end, sustain/position, sustain/length, echo/delay, pitch/offset, loop, sample/start, sample/end, sustain/position, sustain/length, echo/delay, pitch/offset, // sample/start, sample/end, sustain/position, sustain/length, echo/offset, finished, finished/not/finish, note/ready, not/finish, period, velocity, pressed, // Debug outputs serial/byte, serial/ready, output serial/length, /@input [16:0] sample/length; // The length of the recording. @input [1:0] sample/position; // Rotary encoder knob input for sample/position. @input [1:0] sustain/position; // Rotary encoder knob input for sustain/position. @input [1:0] sustain/length; // Rotary encoder knob input for sustain/length. @input [1:0] pitch/offset; // Rotary encoder knob input for pitch/offset. @input [1:0] echo/delay; // Rotary encoder knob input for echo/delay. @input clock System clock. @input new_name Pulse to indicate a new audio sample available. @input record A high indicates the system is currently recording. @input mid/wx A mid serial line input (normally high). @input [1:0] sample/start Rotary encoder knob input for sample/start. @input [1:0] sustain/position Rotary encoder knob input for sustain/position. @input [1:0] sustain/length Rotary encoder knob input for sustain/length. @input [1:0] pitch/offset Rotary encoder knob input for pitch/offset. @input [1:0] echo/delay 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.

else value < (value < lower + INCREMENT) ? lower : value - INCREMENT;
end
endmodule

parameters
parameters [clock, clock, reset, reset],
...sample/length, sample/length...,
...sample/position, sample/position...,
...sustain/position, sustain/position...,
...sustain/length, sustain/length...,
...echo/delay, echo/delay...,
...pitch/offset, pitch/offset...;

| 17 |
wires [7:0] serial_byte;

wire [7:0] serial_ready;

serial fsm my_serial_fsm( .clock(clock), .reset(reset),
 .byte(serial_byte), .ready(serial_ready));

wire [7:0] midi_pitch, midi_velocity;
wire midiread;

midi fsm my_midi_fsm( .clock(clock), .reset(reset),
 .byte(serial_byte), .new_byte(serial_ready), .pitch(midi_pitch), .velocity(midi_velocity), .pressed(midiread), .ready(midiread));

wire record_edge;

level <= midiread & record_edge;
reg [1:0] resetextend;

record begin
always @(posedge clock) begin
if (reset) begin
if (resetextend) begin
// reset load reasonable values
samplestart <= 0;
samplelength <= samplestart + random(1, samplelength);
sustainposition <= samplelength;
echodelay <= samplelength;
pitchoffset <= 24;
end
end
else begin
{samplestart, sampleend, sustainposition, sustainlength, echodelay, pitchoffset} <= 0;
end
end
endmodule

Listing 5: Verilog file cstringdisp.v
// PARAMETERS:
// NCHAR — number of characters in string to display
// NCHARBITS — 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 and font.rom ngo.
// For different fonts, you can change font.rom. For different string
display colors, change the assignment to cpixel.

mod cstring_display (vclk, hcount, vcount, pixel, cstring, cx, cy);

parameter NCHAR = 8; // number of 8-bit characters in string
parameter NCHARBITS = 3; // number of bits in NCHAR
parameter COLOR = 7; //
inpu vclk; // 40 MHz clock
inpu [10:0] hcount; // horizontal index of current pixel (0...799)
inpu [9:0] vcount; // vertical index of current pixel (0...599)
ou [2:0] pixel; // char display's pixel
inpu [NCHAR*8-1:0] cstring; // character string to display
inpu [10:0] cx;
inpu [9:0] cy;

// 1 line x 8 character display (8 x 12 pixel-sized characters)
wire [10:0] hoff = hcount - cx;
wire [9:0] voff = vcount - cy;
wire [NCHARBITS-1:0] column = NCHAR - 1 - hoff [NCHARBITS-1 + 3:3]; //
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*n + n];

// look up raster row from font rom
wire reverse = char[7];
wire [7:0] font_byte;
font_addr /= (font_byte, vclk, font_byte);

// generate character pixel if we're in the right h,v area
wire [2:0] ddisplay = ((hcount >= cx) & (vcount >= cy) & (hcount <= cx + NCHAR*8) & (vcount <= cy + 12));
wire [2:0] pixel = ddisplay ? 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; // .003 sec with a 40MHz clock
input reset, clock, noisy, clean;

reg [18:0] count;
reg new, clean;
always @posedge clock
begin
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
begin
count <= count + 1;
end
end
count <= count+1;
endmodule

Listing 7: Verilog file delay.v

// delay − Pulse generated after one of two selectable delay.
* @param DELAY2_COUNT The delay in clock cycles when short == 0.
* @param DELAY1_COUNT The delay in clock cycles when short == 1.
* @param 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 DELAY1_COUNT = 430;
parameter COUNT_SIZEOF = 10;
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; //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: 26−Sept−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−Sept−05 like: updated to use new reset−once state machine, remove clear
// 02−Nov−05 like: updated to make it completely synchronous
//
// Inputs:
//  reset − active high
// clock27mhz − the synchronous clock
// data − 64 bits; each 4 bits gives a hex digit
//
// Outputs:
//
// display − display lines used in the 6.111 labkit (rev 003 & 004)
//
// module display_16hex (reset, clock27mhz, data_in,
// dispblank, dispclock, disp0−15, dispreset, dispdata_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_ax, disp_ce_b, disp_reset_b;

reg disp_data_in, disp_ax, 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;
wire old_clock;
wire reset = (reset_count != 0);
assign disp_blank = 1'b0;  // low <= not blanked
always @(posedge clock @ 27mhz)
begin
  if (reset)
  begin
    state <= 0;
    dot_index <= 0;
    control <= 32'b0FFFF7FF;
  end
  else
  case (state)
    8'0b0:
      begin
        // Reset displays
        disp_data_in <= 1'b0;
        disp_ax <= 1'b0; // dot register
        disp_ce_b <= 1'b0;
        disp_data_in <= 1'b0;
        dot_index <= 0;
        state <= state+1;
      end
    8'0b1:
      begin
        // End reset
        disp_ce_b <= 1'b1;
        state <= state+1;
      end
    8'0b2:
      begin
        // Initialize dot register (set all dots to zero)
        disp_data_in <= 1'b0;
        disp_data_out <= 1'b0; // dot_index[0];
        if (dot_index == 63)
          state <= state+1;
        else
          dot_index <= dot_index+1;
      end
    8'0b3:
      begin
        // Latch dot data
        disp_ax <= 1'b1;
        dot_index <= 31;  // re-purpose to init ctrl reg
        state <= state+1;
      end
  endcase
end

always @(posedge clock @ 27mhz)
begin
  count <= reset ? 0 : (count = = 53 ? 0 : count + 1);
  old_clock <= clock;
end
assign disp_reset = (reset_count != 0);
assign disp_ce = ~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 [3:0] dots;  // dots for a single digit
reg [3:0] nibble;  // hex nibble of current character
reg [63:0] data;

always @(posedge clock @ 27mhz)
begin
  if (clock_tick)
  begin
    case x(state)
      8'0b0:
        begin
          // Reset displays
          disp_data_in <= 1'b0;
          disp_ax <= 1'b0; // dot register
          disp_ce_b <= 1'b0;
          disp_data_in <= 1'b0;
          dot_index <= 0;
          state <= state+1;
        end
      8'0b1:
        begin
          // End reset
          disp_ce_b <= 1'b1;
          state <= state+1;
        end
      8'0b2:
        begin
          // Initialize dot register (set all dots to zero)
          disp_data_in <= 1'b0;
          disp_data_out <= 1'b0; // dot_index[0];
          if (dot_index == 63)
            state <= state+1;
          else
            dot_index <= dot_index+1;
        end
      8'0b3:
        begin
          // Latch dot data
          disp_ax <= 1'b1;
          dot_index <= 31;  // re-purpose to init ctrl reg
          state <= state+1;
        end
  endcase
end
module echo (clock, reset, new_frame, echo_frame, decode, enable, data_in);
    input clock;
    input reset;
    input new_frame;

Listing 9: Verilog echo.v
input [12:0] echotime; // time to echo for, in terms of frames
input [11:0] decayrate; // rate at which to decay, in terms of shift operations
input enable; // enable or disable echo
input signed [17:0] datain; // input data
output signed [17:0] dataout; // 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 [1:0] echostate; // state of the echo FSM
reg [12:0] eramaddr; // ram address of the current frame.
reg [17:0] dataout; // latched output frame.
reg [1:0] echostate; // state of echo operation for a given frame.
reg [17:0] datainholdvalue; // latched input data
reg signed [18:0] bigdataout; // 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;
wires signed [17:0] muxeddatain = enable? datain: nothingness;
// mux on the output to determine if echo should be enabled.
eram14x18 ram14 ( .addr( eramaddr ) , .clk( clock ) , .din(datainholdvalue) , .dout(ramoutwire) , .we(eче) ) ;
// note here that data out is the frame output of the module.

always @ (posedge clock) begin
if(reset) begin
ramaddr <= 0;
//endres <= 0;
bigdataout <= 0;
ewe <= 0;
datainholdvalue <= 0;
datain <= 0;
deramaddr <= 0;
enewframe <= 0;
end
else begin
if(newframe) datainholdvalue <= datain;
dataout <= bigdataout[18:1];
if(enable) begin
case(eчostate)
0: begin
bigdataout <= ramoutwire + datainholdvalue;
1: begin
bigdataout <= {ramoutwire[17], ramoutwire[17], ramoutwire[16:1]}
+ datainholdvalue;
2: begin
bigdataout <= {ramoutwire[17], ramoutwire[17], ramoutwire[17],
ramoutwire[16:2]} + datainholdvalue;
3: begin
bigdataout <= {ramoutwire[17], ramoutwire[17], ramoutwire[17],
ramoutwire[17], ramoutwire[16:3]} + datainholdvalue;
endcase
eчostate <= 1;
ewe <= 1;
end
1: begin
eчostate <= 2;
ewe <= 0;
end
2: begin
if(dataout <= addvalue)
eчostate <= eramaddr <= eramaddr + 1;
else eramaddr <= 0;
eчostate <= 3;
end
3: begin
if(newframe) echostate <= 0;
end
eчostate <= 0;
end
end
endelse begin
dataout <= datain;
ramaddr <= eramaddr + 1;
ewe <= 1;
eчostate <= 0;
endendendmodule

Listing 10: Verilog file final.v
// CHANGES FOR BOARD REVISION 004
// 1) Added some signals for logic analyzer pods 2-4.
// 2) Expanded "tv/rgb/clock" to 20 bits.
// 3) Renamed "[2]-vga-out" to "tv/rgb-data" and "[2]-clock" to
//     "tv/rgb-clock".
// 4) Reversed displayout and dispdataout signals, so that "out" is an
//     output of the FPGA, and "in" is an input.

// CHANGES FOR BOARD REVISION 005
// 1) Combined flash chip enables into a single signal, flash_ea.

// CHANGES FOR BOARD REVISION 006
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mousedata_"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//     the data bus, and the byte write enable has been combined into the
// 4) Removed the "systemace/lock" net, since the SystemACE clock is now
//     hardwired on the PCB to the oscillator.

// Complete change history (including bug fixes)

// 2003-Sep-09 Added missing default assignments to "ac97_data_out", "disp_data_out", "analyzer[2]-clock" and
//     "analyzer[2]-data".
// 2003-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/win" 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 buses to 19 bits, to match 16Mb 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, ac97_data_out, ac97_data_in, ac97_sync, ac97_clock,
             vga_in_red, vga_in_green, vga_in_blue, vga_in_sync,
             vga_in_clk, vga_out_red, vga_out_green, vga_out_blue, vga_out_sync,
             vga_out_clk, tv_out_red, tv_out_green, tv_out_blue, tv_out_sync,
             tv_out_clk, tv_out_data, tv_out_hsync, tv_out_vsync,
             tv_out_hsync, tv_out_feedback, tv_out_hblank, tv_out_vblank,
             tv_out_vblank, vi_out[24], vi_out_data, vi_out[24], vi_out_data,
             vi_out[24], vi_out_vsync, vi_out[24], vi_out_hsync, vi_out[24], vi_out_hsync,
             vi_out[24], vi_out_feedback, vi_out[24], vi_out_hblank, vi_out[24], vi_out_vblank,
             vi_out[24], vi_out_vblank, vi_out[24], vi_out_feedback,
             ram_data, ram_address, ram[1:0], raml[1:0], raml[1:0], ram[1:0],
             ram[1:0], ram_address, ram[1:0], raml[1:0], ram[1:0], raml[1:0], ram[1:0],
             raml[1:0], raml[1:0], ram[1:0], raml[1:0], ram[1:0],
             clock_feedback_out, clock_feedback_in,
             flash_data, flash_address, flash[2:0], flash[2:0], flash[2:0], flash[2:0],
             flash_data, flash_address, flash[2:0], flash[2:0], flash[2:0], flash[2:0],
             sv2[14], sv2[14], sv2[14], sv2[14], sv2[14], sv2[14],
             mouse_clock, mouse_data, keyboard_clock, keyboard_data,
             clock[27], clock1, clock2,
             disp[2:0], disp[2:0], disp[2:0], disp[2:0], disp[2:0], disp[2:0], disp[2:0],
             buttons, buttons, buttons, buttons, buttons, buttons, buttons, buttons,
             switch, led,
user1, user2, user3, user4, daughtercard,
system1[6:0], system2[6:0], system3[6:0], system4[6:0],
system5[6:0], system6[6:0], system7[6:0], system8[6:0],
analyzerr[6:0], analyzerr[6:0], analyzerr[6:0], analyzerr[6:0],

output beep, audio_in1[6:0], audio_in2[6:0], audio_out;
input audio_in1[6:0], audio_in2[6:0], audio_out;

output [7:0] vga_in[6:0], vga_out[6:0], vga_out[6:0],
input vga_out[6:0], vga_out[6:0], vga_out[6:0], vga_out[6:0];

output [9:0] tv_out[8:0], tv_out[8:0], tv_out[8:0],
input tv_out[8:0], tv_out[8:0], tv_out[8:0], tv_out[8:0];

output [19:0] vga_in[14:0],
input vga_in[14:0], vga_in[14:0], vga_in[14:0];

output [35:0] ram0[32:0],
input [35:0] ram0[32:0], ram0[32:0], ram0[32:0];

output [18:0] ram1[16:0], ram1[16:0], ram1[16:0], ram1[16:0],
input [18:0] ram1[16:0], ram1[16:0], ram1[16:0], ram1[16:0];

output [3:0] ram0[3:0],
input [3:0] ram0[3:0], ram0[3:0], ram0[3:0], ram0[3:0];

input clock_feedback_in,
output clock_feedback_out;

input [15:0] flash[14:0],
output [23:0] flash[22:0], flash[22:0], flash[22:0],
input flash[22:0],

output [23:0] r23[22:0],
input [23:0] r23[22:0], r23[22:0];

input mon0[32:0], mon0[32:0], keyboard_in[32:0],
input [32:0] mon0[32:0], mon0[32:0], mon0[32:0];

input clock[32], clock[32], clock[32],
output disp[31:0], disp[31:0], disp[31:0], disp[31:0];

output disp[31:0],
input disp[31:0], disp[31:0], disp[31:0];

input button[4], button[4], button[4],
output button[4], button[4], button[4],
input [4] button[4],
output [7:0] switch;

output [7:0] led;
input [31:0] user1, user2, user3;

input [43:0] daughtercard;

input [15:0] system1[14:0],
output [6:0] system2[5:0], system3[5:0],
input system4[5:0], system5[5:0],
output [15:0] analyzer1[14:0], analyzer2[14:0],
input analyzer3[14:0], analyzer4[14:0],
output analyzer1[15:0], analyzer2[15:0], analyzer3[15:0], analyzer4[15:0];

//utures

// Audio Input and Output
assign beep = 1'b0;
// lab3 assign audio_in = 1'b0;
// lab3 assign audio_out = 1'b0;
// lab3 assign c97dataout = 1'b0;
// c97_data_in is an input

// VGA Output
/*
assign rgb_red = 10'h0;
assign rgb_green = 10'h0;
assign rgb_blue = 10'h0;
assign rgb_sync = 1'b1;
assign rgb_blank = 1'b1;
assign rgb_pixelclock = 1'b0;
assign rgb_hsync = 1'b0;
assign rgb_vsync = 1'b0;
*/

// Video Output
assign tvc_out = 10'h0;
assign tvc_reset = 1'b0;
assign tvc_clock = 1'b0;
assign tvc_i2c = 1'b0;
assign tvc_pal = 1'b0;
assign tvc_hsync = 1'b1;
assign tvc_vsync = 1'b1;
assign tvc_blank = 1'b1;
assign tvc_subreset = 1'b0;

// Video Input
assign tvi_in = 10'h0;
assign tvi_fifo = 1'b0;
assign tvi_iso = 1'b0;
assign tvi_reset = 1'b0;
assign tvi_i2c = 1'bZ;

// SRAM
assign rami_data = 36'hZ;
assign rami_address = 19'h0;
assign rami_clk = 1'b0;
assign rami_i2c = 1'b1;
assign rami_w = 1'b1;
assign rami_r = 1'b1;
assign rami_c = 1'b1;
assign rami_we = 4'hF;

assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_clk = 1'b0;
assign ram1_i2c = 1'b1;
assign ram1_w = 1'b1;
assign ram1_r = 1'b1;
assign ram1_c = 1'b1;
assign ram1_we = 4'hF;

assign rami_feedbackout = 1'b0;

// Flash ROM
assign flash_data = 16'hZ;
assign flash_address = 24'h0;
assign flash_i2c = 1'b1;
assign flash_w = 1'b1;
assign flash_r = 1'b1;
assign flash_c = 1'b1;

// RS-232 Interface
assign rs232_txd = 1'b1;
assign rs232_rxd = 1'b1;

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

// LED Displays
//assign disp_blank = 1'b1;
//assign disp_lock = 1'b0;
//assign disp_reset = 1'b0;
//assign disp_dataout = 1'b0;
//disp_dataout is an input

// Buttons, Switches, and Individual LEDs
// lab3 assign led = 8'WF;
// button0, button1, button2, button3, button Harding, button_right,
// button_left, button_down, button_up, and switches are inputs

// User LEDs
// wumpus assign user1 = 32'hZ;
assign user2 = 32'hZ;
assign user3 = 32'hZ;
// MIDI Serial Input
synchronize syncr (clk(clock), in(user[1]), out(midin));

// CONTROL

// Inputs from section NOTES
wire [16:0] sample_length;
wire [0:0] finished_note, note_length;
wire [16:0] sustain_length;

// Outputs Parameters
wire [16:0] sample_start, sample_end, sustain_position;
wire [13:0] echo_length;
wire [5:0] delay;
wire [15:0] pitch_offset;
wire [15:0] period;
wire [7:0] velocity;
wire pressed;
wire serial_ready;
wire serial_byte;

control my_control (clock(clk), reset(reset),
    new_frame (new_frame),
    record (record),
    sample_length (sample_length),
    finished_note (finished_note),
    note_length (note_length),
    note_select (note_select),
    period (period),
    velocity (velocity),
    pressed (pressed),
    // COMMUNICATION WITH NOTES
    finished_note (finished_note),
    finished_state (finished_state),
    sustain_state (sustain_state),
    sustain_length (sustain_length),
    echo_delay (echo_delay),
    pitch_offset (pitch_offset),
    midin (midin),
    sample_start (sample_start),
    sample_end (sample_end),
    sustain_position (sustain_position),
    sustain_length (sustain_length),
    echo_delay (echo_delay),
    pitch_offset (pitch_offset);

// READY
    serial_ready (serial_ready),
    serial_byte (serial_byte);

/// NOTES

wire [17:0] audio_delay;
wire [15:0] note_length;
wire [17:0] echo_audio;

notes notes (clock(clk), reset(reset),
    new_frame (new_frame),
    note_length (note_length),
    echo_delay (echo_delay),
    record (record),
    sample_start (sample_start),
    sample_end (sample_end),
    finished_state (finished_state),
    finished_note (finished_note),
    period (period),
    velocity (velocity),
    pressed (pressed),
    finished_note (finished_note),
    finished_state (finished_state),
    sustain_state (sustain_state),
    sustain_length (sustain_length),
    echo_delay (echo_delay),
    pitch_offset (pitch_offset);

// DEBUG
    current_note (current_note),
    current_frame (current_frame),
    ready (ready),
    frame_prep (frame_prep),
    mem_ready (mem_ready),
    pressed (pressed),
    period (period),
    time (time),
    audio_delay (audio_delay),
    frame (frame),
    frame_length (frame_length),
    frame_address (frame_address),
    frame_address (frame_address),
    ram_address (ram_address);

/// ECHO

echo my_echo (clock(clk), reset(reset),
    new_frame (new_frame),
    record (record),
    decay (echo_decay),
    enable (echo_enable),
    data_in (data_in),
    data_out (data_out),
    audio (audio);

wire [17:0] echo_audio;
wire [15:0] audio_delay;
wire [16:0] delay;
middle [15:0] delay (delay);

// SERIAL PORT
    clk(clk),
    audio_delay (audio_delay);

// ASSIGN
    assign audio = c ? echo_audio : t record ? 2'b0, notes_audio, record : echo_audio;

/// VIDEO OUTPUT GENERATION

// generate basic XYG video signals
wire [18:0] hcounter;
wire [9:0] vcounter;
wire hsync, vsync, blank;
vga vfga(clock, hcount, vcount, hsync, vsync, blank);

// feed VGA signals to user's pong game
wire [2:0] pixel;
reg [2:0] rgb;
reg b, v, vblank;

// VGA Output: In order to meet the setup and hold times of the
// ADV7125, we send ~clock 65mhz.
assign vgaout_red = {8{rgb[2]}},
assign vgaout_green = {8{rgb[1]}},
assign vgaout_blue = 1'b1; // not used
assign vgaout_black = ~b;
assign vgaout_clock = ~clock;
assign vgaout_hsync = hsync;
assign vgaout_vsync = vs;

visualizer.vi(clock(clock), reset(reset),
    hcount(hcount), vcount(vcount), hsync(hsync), vsync(vsync), blank(blank),
    vblank(vblank), vcenter(vcenter), hcenter(hcenter), 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(pixel_offset),
    audio(from_audio[15:0]), record(record), new_frame(new_frame));

// DEBUG OUTPUTS
reg [7:0] lights;
assign led = ~lights;
assign analyzer2_clock = clock;
assign analyzer2_data = {serialbyte, 6'b0, serialready, midline};
assign analyzer2_data = {period};
always @(posedge clock) begin
    if (reset) begin
        updown = 0;
        readypulse = 0;
    end else begin
        if (((olda && ~a && ~b) || (oldb && ~b && a)) begin // on falling edge
            updown <= ~1;
            readypulse <= 1;
            updown <= b ? 1'0;
        end else
            readypulse <= 0;
    end
endmodule

Listing 11: Verilog file grey_decode.v
Listing 12: Verilog file level_to_pulse.v

```verilog
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

```verilog
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 notmem 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 addresses 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];
        default: memory_line <= memory_line;
      endcase
    end
  end
endmodule
```

Listing 14: Verilog file loop_control_fsm.v

```verilog
```
module loop_control fsm (clock, reset, new frame, finished, finished_note select, pressed, ready, state);

// Debug outputs

input clock;
input reset;
input new frame;
input finished;
input finished_note select;
output pressed;
output ready;
output [2 : 0] state;

reg pressed;
reg ready;

reg [2 : 0] state;
reg [2 : 0] old state;

reg [1 : 0] frame count;

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 FRAME1;
            WAIT FRAME1:
                if (frame count == 2) state <= RELEASE;
            default:
                state <= WAIT FRAME0;
        endcase
        // Calculate new outputs
        if (state != old state || state == WAIT FRAME0) begin
            case (state)
                WAIT FRAME0:
                    begin
                        pressed <= 0;
                        frame count <= 0;
                    end
                // WAIT FRAME1:
                    if (new frame) frame count <= frame count + 1;
                    // Wait for another two frame
                    if (finished_note select == 0 & & finished)
                        state <= PRESS;
                    endcase
                pressed <= 0;
                pressed <= 0;
                if (pressed) pressed <= pressed + 1;
                if (pressed) pressed <= pressed + 1;
                if (pressed) pressed <= pressed + 1;
                if (pressed) pressed <= pressed + 1;
            endcase
                old state <= state;
        end
        end
    end
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 [3:0] state;
  reg [3:0] old_state;

  // The states of the finite state machine
  parameter WAIT = 0; // wait for the first byte to arrive
  parameter FUNCTION = 1; // determine the function of the midi event
  parameter IGNORE = 2; // ignore 1 or 2
  parameter IGNORE1 = 3; // 12 data bits for unsupported opcodes
  parameter IGNORE2 = 4; // determine pressed for the midi event
  parameter PITCHWAIT = 5; // wait for the next byte
  parameter PITCH = 6; // store the pitch value
  parameter VELOCITYWAIT = 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;
            byte <= byte;
          end
          FUNCTION:
          begin
            if (byte[7:4] == 8 || (byte[7:4] == 9)) // pressed or released events
              state <= PITCHWAIT;
              state <= IGNORE2;
              state <= IGNORE1;
            else
              state <= WAIT;
          end
        IGNORE1:
          if (new_byte) // wait for a byte...
            state <= WAIT; // ... and ignore it
        IGNORE2:
          if (new_byte) // wait for a byte...
            state <= IGNORE2; // ... and ignore the next one too
        PITCHWAIT:
          if (new_byte) // pitch value has arrived
            state <= PITCH;
      endcase
    end
  end
module newfp(clock, reset, mems_ready, wait, frame_n, frame_n_plus, frame_n_minus, ready_frameprep, w_r_bar, mem_frame, from_ac97_data, to_ac97_record, ramaddr, sample_length);

input clock;
input reset;
input [16:0] w_r_bar;
input [16:0] 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.
reg [16:0] ramaddr; // current address in the sample memory being used
reg old_w_r_bar; // previous signal of the write button, used to determine when // the button was pressed and thus when to reset ramaddr
reg [2:0] f_state; // 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_plus, frame_n_minus; // 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 length value.

wire [15:0] ramout;
assign to_ac97_record = ramout;

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

always @ (posedge clock) begin
if(reset) begin
ready_frameprep <= 0;
ramaddr <= 0;
old_w_r_bar <= 0;
frame_n <= 0;
frame_n_plus <= 0;
frame_n_minus <= 0;
ready_hold <= 0;
f_state <= 0;
sample_length <= 0;
end
else begin
if(ready_frameprep) ready_frameprep <= 0;
old_w_r_bar <= w_r_bar;
if(w_r_bar) begin
if( old_w_r_bar ) ramaddr <= 0;
end

endmodule
else if (ramaddr < MAXSIZE) if (newframe) begin
  ramaddr <= ramaddr + 1;
sample_length <= ramaddr;
end else begin
  if (newframe) ramaddr <= 0;
end else begin
  if (memready | readyhold) begin
    case (fstate)
      0:
        begin
          ramaddr <= nout;
          readyhold <= 1;
          fstate <= 1;
        end
      1:
        begin
          ramaddr <= nout + 1;
          fstate <= 2;
        end
      2:
        begin
          ramaddr <= nout - 1;
          frame <= ramout;
          fstate <= 3;
        end
      3:
        begin
          frame <= ramout;
          fstate <= 4;
        end
      4:
        begin
          if (!readyframeprep) readyframeprep <= 1;
        end
    endcase
  end
end endmodule

Listing 17: Verilog note_control_fsm.v

```verilog
module note_control_fsm (clock, reset, queue_empty, queue_pop, finished, finished_note_select, note_ready, note_select, queue_period, queue_pressed, queue_velocity, newframe, 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 newframe;
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[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_period [note_select];
assign note_period = note_period[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_up <= 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
        state <= WAIT;
        note_select <= 0;
      end
    endcase
      LATCH: begin
        state <= queue_empty ? CHECK_PRESSED : CHECK_RELEASED;
        state <= LATCH;
        queue_up <= 1;
      end
    end
    if (queue_empty) begin
      if (state <= queue_pressed) begin
        CHECK_PRESSED: CHECK_RELEASED:
        // has this one isn’t being used
        if (note_select == 7)
          state <= FULL;  // did we reach the last note?
        else if (note_select == 7)
          state <= FULL;  // did we reach the last note?
      end
      else begin
        note_select <= note_select + 1;  // otherwise increment the note
      end
      end
    endcase
    CHECK_PRESSED: if (finished_reg[note_select])
      state <= NOTE_PRESSED;  // hasn’t this one isn’t being used
    endcase
      CHECK_RELEASED: if (finished_reg[note_select] & period == note_periods[note_select])
      state <= NOTE_RELEASED;
    endcase
      NOTE_PRESSED: begin
        state <= WAIT;
        note_periods[note_select] <= period;
        state <= WAIT_FRAME;
        wait_frames <= 9;
        note_ready <= 1;
      end
    endcase
      CHECK_PRESSED: begin
        state <= WAIT;
        wait_frames <= 9;
        note_ready <= 1;
      end
    endcase
      NOTE_NOTFOUND: begin
        state <= WAIT;
        if (queue_empty) begin
          state <= WAIT;
        end
      end
    endcase
  end
end
Listing 18: Verilog file note_control.v

```verilog
/*
 * note_control — Manages loop and keyboard output to a set of notes.
 */
parameter NOTE_COUNT = 8; //The number of notes
parameter NOTE_COUNT/16 = 3; //The bit width necessary to capture it

input clock;
input reset;
input midiready;
input [7:0] midivelocity;
input [7:0] midipitch;
input [5:0] pitchoffset;
input [NOTE_COUNT/16-1:0] finished; //one note select;
input finished; //one note select;
input [5:0] pitch;
input [7:0] notevelocity;
input [7:0] notecontrolstate;
input [7:0] basemoji;
output [NOTE_COUNT/16-1:0] note_select;
output note_select; //one note select;
output [15:0] period; //The period (relative to system clock) of a new note event.
output [7:0] velocity; //The velocity of a new note event.
output note_ready; //One clock cycle pulse indicating a new note control event is ready.
output note_pressed; //Whether note control event is pressed or released.
output finished; //ignore finished if we just turned the note on
output midiready; //A one cycle pulse indicating a new keyboard event.
output midivelocity; //The velocity of the keyboard event.
output midipitch; //The pitch of the keyboard event.
//Debug outputs
output note_select,
output baseemoji,
output finished,
output period,
output velocity,
output loop,
output note Velocity,
output note select,
output keyboard select,
output Fifo empty,
output Fifo full,
output Fifo read,
output Fifo write,
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
output Fifo read;
output Fifo state;
output Fifo offset,
output Fifo end;
output Fifo reset;
output Fifo write,
Listing 19: Verilog file notemem.v

module notemem(clk, reset, new_frame, mem_ready, w_bar, r_write, period_in, velocity_in, pressed_out, note_select, finished_note, finished_state, erase_note, inc_button, n_still);

input clk;
input reset;
input new_frame; //a? needs has a new frame
input w_bar; //signal from control indicating that the sound is writing or reading.

//data written to parameter memory by control
input w_write;  //a? select write) indicating selected note should be written with
input pressed_in;  //value of pressed to read into note memory location from

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 [11:0] window_len;  //length of the window (vowel hold from controller

//data from taylor used to write the variable memory:
input [16:0] new;  //new from taylor
input [10:0] delta_t;  //new delta from taylor
input going[3];  //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] w_out;  //w from 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

endmodule

```verilog
//Xilinx Synchronous FIFO, 32 deep
notemem(clk, reset, mem_ready, w_bar, n_write, period_in, velocity_in, pressed_out, note_select, finished_note, finished_state, erase_note, inc_button, n_still);

notemem sm_keyboard(sm_clock(clk), .reset(reset), .pressed(pressed), .velocity(velocity), .queue(queue), .frame(frame), .finished(finished), .note_select(finished_note_select), .new_frame(new_frame), .note_ready(keyboard_ready), .note_select(keyboard_note_select), .state(note_state), .finished_note(finished_note_select), .note_period(note_period));

notemem fifo (notemem clk, reset, .write(mid_pitch[7:0], midi_velocity[7:0], midi_pressed), .read(mid_pitch[7:0], midi_velocity[7:0], midi_pressed), .full(queue_full));

//adjust the period for the period offset
period_lookup_table(period_table(clk), .add(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 FSM loop FSM(clk, reset, .q0(queue_pitch + 24 - pitch_offset), .q1(queue_period));

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 pressed = loop ? loop_pressed : keyboard_pressed;
assign velocity = loop ? loop_velocity : keyboard_velocity;
assign period = loop ? loop_period : keyboard_period;
endmodule
```
module notes(clock, reset, nframe, mbar, enable, note select, period_in, velocity_in, pressed_in, window, hold, length, finished_note, finished_state, sample_start, sample_end, dataout, frame, sframe, sframes, frame_alive, mem_ready, mem_status, mem_add, mem_write, mem_read, mems, rbar, new, note ready, note pready, note select; for allocation of notes by control.

output pressed; //pressed given to framemem from memory
output mem_ready; //memm is ready for framemem to read a new set of values
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 inclusion;
output [16:0] new;

//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 note
//notes into mem ready)
reg mem_ready;
reg taylor_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 = (? '6000000, velocity_in, pressed_in, period_in);
assign {velocity_in, pressed_in, period_in} = p_data_read[24:0];

linemem params: clock(clock), reset(reset), clear(0), writeenable(1), writeaddr(note select),
write(data), read(data), readaddr(current_note), read(data[t_data_read]);

wire [31:0] v_data_write, v_data_read;
assign v_data_write = (? 3'6000, sframe, delay_new, delay_grow);
assign {v_data, delay_new, delay_grow} = v_data_read[24:0];

always @(posedge clock) begin
if(reset) begin
  current_note <= 0;
  previous_note <= 0;
  finished_frame <= 0;
  mem_ready <= 0;
  taylor_delayed <= 0;
  finished_note <= 0;
  finished_state <= 0;
end
else begin
  taylor_delayed <= taylor_delayed;
  if(multi_note) mem_ready <= 0;
  //count out notes upon new frame:
  if(new_note) begin
    current_note <= 0;
    if(!!mem_ready & !mbar) mem_ready <= 1;//this starts off first note.
  end
  else if(taylor_delay) begin
    previous_note <= previous_note;
    finished_note <= finished_note;
    finished_state <= finished_state;
    if(!!current_note) begin
      finished_frame <= 1; //what does this do??
    end
    else begin
      current_note <= current_note + 1;
      if(!!mem_ready & !mbar) mem_ready <= 1;
    end
  end
endendmodule

Listing 20: Verilog file notes.v
Listing 21: Verilog file parameter_control.v

```verilog
module parameter_control(clock, reset, sample_start, tframe, nframe, tayl, frame, sumframe, sumdone);

    input clock;
    input reset;
    input [1:0] sample_start;
    input [1:0] sample;
    input [1:0] sustain;
    input [1:0] echa;
    input [1:0] echo_delay;
    input [1:0] scanlength;
    input [1:0] sample_pos;
    input [1:0] sustain_pos;
    input [1:0] scan_pos;
    input [1:0] echo_pos;
    input [1:0] echo_delay_pos;
    input [1:0] tun_length;

endmodule
```
wire sample\_length, sample\_start, sample\_end, incr;
reg sample\_length,
reg [16:0] sample\_end;
grey\_decode sample\_end, sample\_incr;

bound\_parameter sample\_end, param (clock, clock), reset(reset);
defparam sample\_end, param SIZEOF = 17;
defparam sample\_end, param INCREMENT = 1024;
wire sample\_length, sample\_start, incr, sample\_start\_ld;
assign sample\_length = 0;
grey\_decode sample\_start, decode (clock, clock), reset(reset);
bound\_parameter sample\_start, param (clock, clock), reset(reset);
defparam sample\_start, param SIZEOF = 20;
defparam sample\_start, param INCREMENT = 1024;
wire sustain\_length, sustain\_start, incr, sustain\_length\_ld;
assign sustain\_length\_ld = 0;
grey\_decode sustain\_length, decode (clock, clock), reset(reset);
bound\_parameter sustain\_length, param (clock, clock), reset(reset);
defparam sustain\_length, param SIZEOF = 15;
defparam sustain\_length, param INCREMENT = 1;
wire sustain\_length\_ld, sustain\_length\_incr, sustain\_length\_ld;
assign sustain\_length\_ld = 0;
grey\_decode sustain\_length\_ld, decode (clock, clock), reset(reset);
bound\_parameter sustain\_length\_ld, param (clock, clock), reset(reset);
defparam sustain\_length\_ld, param SIZEOF = 12;
defparam sustain\_length\_ld, param INCREMENT = 1;
wire echo\_delay, echo\_delay\_ld, echo\_delay\_incr, echo\_delay\_ld;
assign echo\_delay\_ld = 0;
grey\_decode echo\_delay\_ld, decode (clock, clock), reset(reset);
bound\_parameter echo\_delay\_ld, param (clock, clock), reset(reset);
defparam echo\_delay\_ld, param SIZEOF = 11;
defparam echo\_delay\_ld, param INCREMENT = 1;
wire echo\_delay\_ld, echo\_delay\_incr, echo\_delay\_ld;
assign echo\_delay\_ld = 0;
grey\_decode echo\_delay\_ld, decode (clock, clock), reset(reset);
bound\_parameter echo\_delay\_ld, param (clock, clock), reset(reset);
defparam echo\_delay\_ld, param SIZEOF = 4;
defparam echo\_delay\_ld, param INCREMENT = 1;
always @(posedge clock) begin
  if(reset) begin

    a(sample\_length[0]), b(sample\_length[1]), up(down(sample\_length, up),
      ready\_pulse(sample\_length, incr);
    up(down(sample\_length, up),
      lower(0),
      incr(sample\_length, incr);
    value(sample\_length);

    a(sample\_start[0]), b(sample\_start[1]), up(down(sample\_start, up),
      ready\_pulse(sample\_start);
    up(down(sample\_start, up),
      lower(0),
      incr(sample\_start, incr);
    value(sample\_start);

    a(sustain\_length[0]), b(sustain\_length[1]), up(down(sustain\_length, up),
      ready\_pulse(sustain\_length);
    up(down(sustain\_length, up),
      lower(0),
      incr(sustain\_length, incr);
    value(sustain\_length);

    a(sustain\_length\_ld[0]), b(sustain\_length\_ld[1]), up(down(sustain\_length\_ld, up),
      ready\_pulse(sustain\_length\_ld);
    up(down(sustain\_length\_ld, up),
      lower(0),
      incr(sustain\_length\_ld, incr);
    value(sustain\_length\_ld);

    a(echo\_delay[0]), b(echo\_delay[1]), up(down(echo\_delay, up),
      ready\_pulse(echo\_delay);
    up(down(echo\_delay, up),
      lower(0),
      incr(echo\_delay, incr);
    value(echo\_delay);

    a(echo\_delay\_ld[0]), b(echo\_delay\_ld[1]), up(down(echo\_delay\_ld, up),
      ready\_pulse(echo\_delay\_ld);
    up(down(echo\_delay\_ld, up),
      lower(0),
      incr(echo\_delay\_ld, incr);
    value(echo\_delay\_ld);

    a(echo\_delay\_ld[0]), b(echo\_delay\_ld[1]), up(down(echo\_delay\_ld, up),
      ready\_pulse(echo\_delay\_ld);
    up(down(echo\_delay\_ld, up),
      lower(0),
      incr(echo\_delay\_ld, incr);
    value(echo\_delay\_ld);
  end
end
module parameters(clock, reset, sample_start_end, sample_start, sample_end, sample_width, sample_length, window_hold_length, window_hold_period, window_hold_length_end, window_hold_length_start, 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_start
// lower: 0
// increment: 1024
input [1:0] sample_start_ctl;
input sample_start_ctl;
input [16:0] sample_start;
output [16:0] sample_start;

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

// Control signals and output for window_hold_length
// upper: 12'b111111111111
// lower: 12'00
// increment: 16
input [1:0] window_hold_length_ctl;
input window_hold_length_ctl;
input [11:0] window_hold_length;
output [11:0] window_hold_length;

// Control signals and output for echo_delay
// lower: 14'00
// upper: 14'b111111111111
// increment: 16
input [1:0] echo_delay_ctl;
input echo_delay_ctl;
input [13:0] echo_delay;
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_ctl;
input [11:0] pitch_offset;
output [11:0] pitch_offset;

// Sample Start
wire sample_start_ctl;
wire sample_start_ready;
generic decode sample_start.decode: (clock, clock, reset, reset), a(sample_start_ctl[0]), b(sample_start_ctl[1]), updown(sample_start_ctl), ready, value(sample_start_ready);

bound parameter sample_start_ctl param (clock, clock, reset, reset),
updown(sample_start_ctl), ready, value(sample_start_ready),
value(sample_start CTL);
Listing 23: Verilog file serial fsm.v
module serial_fsm (clock, reset, rx, byte, ready, pulse, read, r, bit, state, counter, state_change);

parameter BAUD_RATE = 31250;
parameter CLOCK_RATE = 40000000;
parameter BAUD_RATE_SIZEOF = 15;
parameter CLOCK_RATE_SIZEOF = 26;

input clock;
input reset;
input rx;
output [7:0] byte;
output read, pulse;
output ready, 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 READY = 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 [2:0] bit; // the bit of the serial word, 4 bits wide to include start and stop bit
reg read, pulse;
reg byte;
wire 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 | 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
// Input 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));
delay delay_circuit DELAY1 = CLOCK_RATE / BAUD_RATE / 2;
delay delay_circuit DELAY2 = CLOCK_RATE / BAUD_RATE;
delay delay_circuit COUNT = CLOCK_RATE / BAUD_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, else <= 0;
end
else begin

case(state)

STOPPED: begin

if (* rx) // start bit, get going
state <= WAIT;

if(read) // delay circuit indicates a baud count has gone by
state <= READY;
// output is ready
else
state <= STOPPED; // wait to read the next bit

READY: begin

if (bit >= 9) // stop bit has been reached
state <= READY;
// output is ready
else
state <= READY; // wait to read the next bit

endcase

endcase

if (state != old), state) // if the state has changed since the last clock cycle
// it's 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.

endcase

end

case(state)

STOPPED: begin
short <= 1;
bit <= 0;
wait, disable <= 1;

endcase

always @(posedge clock) begin
if (reset) begin
state <= STOPPED;
state_change <= 1;
wait, disable <= 1;

end
else begin

case(state)

STOPPED: begin
short <= 1;
bit <= 0;
wait, disable <= 1;

endcase

if (state != old), state) // if the state has changed since the last clock cycle
// it's 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.

endcase

end

case(state)

STOPPED: begin
short <= 1;
bit <= 0;
wait, disable <= 1;

endcase

always @(posedge clock) begin
if (reset) begin
state <= STOPPED;
state_change <= 1;
wait, disable <= 1;

end
else begin

case(state)

STOPPED: begin
short <= 1;
bit <= 0;
wait, disable <= 1;

endcase

if (state != old), state) // if the state has changed since the last clock cycle
// it's 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.

endcase

end

case(state)
Listing 24: Verilog file synchronize.v

```verilog
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-1:0] out;
reg [NSYNC-1:0] sync;
reg [WIDTH-1:0] state;
always @ (posedge clk)
begin
    out = {sync[NSYNC-2:0], in};
end
endmodule
```

Listing 25: Verilog file taylor.v

```verilog
module taylor(clock, reset, period, pressed, sample_start, sample_end, window_hold_pos, window_hold_length, frame_n, frame_n_plus, frame_n_minus, ready_frame_prep, frame_n_out, delta_out, going_out, delta_new, frame_n_new, going_new, t_frame_out, taylor_done);

input clock;
input reset;
input [15:0] period; // the period of the sample, with the 5 MSB being integer.
input pressed;
input [16:0] sample_start; // start of sample offset from controller.
input [16:0] sample_end; // end of sample offset from controller.
input [11:0] 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.
in 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_frame_prep;
input [16:0] frame_n_out; // current address.
input [10:0] delta_out; // current offset to virtual frame we are creating.
input going_out; // output from taylor to virtual frame we are creating.
input [10:0] delta_new; // new offset to be given to next frame.
input [15:0] frame_n_new; // new address to be given to next frame.
output signed [15:0] t_frame_out; // output frame offset.
output taylor_done; // the taylor series is done computing a note - frame.

reg signed [15:0] t_frame_out;
reg taylor_done;
reg going_new;
always @ (posedge clock)
begin
    if (reset)
        begin
            t_frame_out < = 0;
            taylor_done < = 0;
            going_new < = 0;
        end
    else begin
        if (ready_frame_prep)
            begin
                frame_n_out < = 0;
                delta_out < = 0;
                t_frame_out < = 0;
                taylor_done < = 0;
                going_new < = 0;
            end
    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,
                 hsync, vsync, sblank, pixel,
                 sample_start, sample_end, window:hold:pos, echo:delay, window:hold:length, pitch:offset,
                 sample:length, record:audio, new:frame.

    //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 [8:0] vcount; // vertical index of current pixel (0..787)
input hsync; // XGA horizontal sync signal (active low)
input vsync; // XGA vertical sync signal (active low)
input blank; // XGA 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 vsync;
output vblank;
output [2:0] pixel;
output [1:0] wstate;
reg [2:0] pixel;
assign vhsync = vhsync;
assign vsync = vsync;
assign vblank = vblank;

// Calculate Screen positions from parameters
reg [9:0] samplesx;
reg [9:0] samplesy;
reg [9:0] windowholdx;
reg [9:0] windowholdd;w
reg [9:0] xlength;

// 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
/

// Test Elements
/
wire [2:0] stop_pixel;
char_string display start_text (.vclock(clock), .hcount(hcount), .vcount(vcount), .pixel(stop_pixel), .cstring("Start"),
.cx(1'd400), .cy(10'd20));
defparam start_text.NCHAR = 5;
defparam start_text.NCHARBits = 3;
defparam start_text.COLOR = 3'b010;
wire [2:0] sustain_pixel;
char_string display sustain_text (.vclock(clock), .hcount(hcount), .vcount(vcount), .pixel(sustain_pixel), .cstring("Sustain"),
.cx(1'd400), .cy(10'd180));
defparam sustain_text.NCHAR = 7;
defparam sustain_text.NCHARBits = 3;
defparam sustain_text.COLOR = 3'b010;
wire [2:0] ecchodelete_pixel;
char_string display echo_text (.vclock(clock), .hcount(hcount), .vcount(vcount), .pixel(ecchodelete_pixel), .cstring("Echo Start"),
.cx(1'd400), .cy(10'd192));
defparam echo_text.NCHAR = 10;
defparam echo_text.NCHARBits = 4;
defparam echo_text.COLOR = 3'b010;
wire [2:0] tune_pixel;
char_string display tune_text (.vclock(clock), .hcount(hcount), .vcount(vcount), .pixel(tune_pixel), .cstring("Tune"),
.cx(1'd400), .cy(10'd180));
defparam tune_text.NCHAR = 4;
defparam tune_text.NCHARBits = 3;
defparam tune_text.COLOR = 3'b010;
/

// Waveform
/
wire [2:0] wbg_pixel;
waveform background wbg (background), .hcount(hcount), .vcount(vcount), .pixel(wbg_pixel), .samplesx(samplesx), .samplesy(samplesy), .xlength(xlength);
defparam wbg_waveform.background.BRIGHT=128;
defparam wbg_waveform.background.WIDTH=512;
defparam wbg_waveform.background.x=188;
defparam wbg_waveform.background.y=40;
wire [2:0] wbg_pixel;
waveform graphic waveform (.clock(clock), .reset(reset),
.audio(audio[15:0]),
.record(record);
.newframe(newframe),
.hcount(hcount), .vcount(vcount),
.pixel(wbg_pixel),
samplesx(samplesx), .samplesy(samplesy));
defparam waveform.ePOSITION = 188;
defparam waveform.ePOSITION = 40;

// Keyboard Element
wire [2:0] keyb_pix;
visualizer_rectangle keyboard_pix (x(11'd270), y(10'd300), hcount(hcount), vcount(vcount), pixel(keyb_pix));
defparam keyboard_pix WIDTH=40;
defparam keyboard_pix HEIGHT=40;
defparam keyboard_pix COLOR=3' b111;

wire [2:0] keyboard_pix;
keyboarditmap keyboard_pix, clock (clock), x(11'd50), y(10'd500), hcount(hcount), vcount(vcount), pixel(keyboard_pix);

// Vertical Bar Elements
// peaks above the graphic
wire [2:0] sh_pix;
visualizer_rectangle start_bar (x(sample_start), y(10'd50), hcount(hcount), vcount(vcount), pixel(sh_pix));
defparam start_bar HEIGHT=138;
defparam start_bar WIDTH=10;
defparam start_bar COLOR=3' b010;
// peaks below the graphic
wire [2:0] sb_pix;
visualizer_rectangle end_bar (x(sample_end), y(10'd50), hcount(hcount), vcount(vcount), pixel(sb_pix));
defparam end_bar HEIGHT=150;
defparam end_bar WIDTH=10;
defparam end_bar COLOR=3' b001;

wire [2:0] sp_pix;
visualizer_rectangle windowhold_bar (x(windowhold), y(10'd168), hcount(hcount), vcount(vcount), pixel(sp_pix));
defparam windowhold_bar WIDTH=10;
defparam windowhold_bar COLOR=3' b010;

wire [2:0] shb_pix;
visualizer_rectangle windowholdlength_bar (x(windowholdlength), y(10'd168), hcount(hcount), vcount(vcount), pixel(shb_pix));
defparam windowholdlength_bar WIDTH=10;
defparam windowholdlength_bar COLOR=3' b001;

wire [2:0] ech_pix;
visualizer_rectangle echoset (echo_delay, y(10'd492), hcount(hcount), vcount(vcount), pixel(ech_pix));
defparam echoset HEIGHT=10;
defparam echoset WIDTH=10;
defparam echoset COLOR=3' b001;

// This lookup table maps the pitch offset to the pixel of the key it represents.
wire [10:0] tune_bar;
keyboard_pix[tune_bar] = tune_position (clk (clock), add (pitch_offset, dot (tune_bar)));
wire [2:0] tmb_pix;
visualizer_rectangle tune_bar (x(11'd45 + tune_bar), y(10'd381), hcount(hcount), vcount(vcount), pixel(tmb_pix));
defparam tune_bar HEIGHT=10;
defparam tune_bar WIDTH=10;
defparam tune_bar COLOR=3' b010;
always @ (posedge clock) begin
  // translate all the incoming parameters into their pixel equivalents.
sample_start = (sample_start > 8) + 188;
sample_offset = (sample_offset > 8) + 188;
windowhold = (windowhold > 8) + 183 + (windowholdlength > 8);
windowholdlength = (windowholdlength > 8) + 183 + (windowholdlength > 8);
ecochannel = (echo_delay > 8) + (sample_start > 8) + 183;
sample_length = (sample_length > 8);

  // bitwise or all the elements except for the waveform background, which provides color to the shape
  // of the waveform pixel
  pixel = (tmb_pix & w_pix) | (keyboard_pix | keyb_pix | sp_pix | sb_pix | spb_pix | sb_pix | ech_pix | sb_pix) sp_pix;
end

endmodule
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

//
// 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_start_y, 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 < (n+HEIGHT)))
    // If we're left of the start pixel, be green, right of the top 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

//
// Indicate a key press
module keyboard_bitmap (cclk, x, y, hcount, vcount, pixel);
  parameter HEIGHT = 80;
  parameter WIDTH = 700;
  input cclk;
  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(cclk), .addr(addr), .dout(dout));

  always @(posedge cclk) begin
    if (hcount ! = old_hcount || vcount ! = old_vcount) begin
      if (hcount + 3 >= x && vcount >= y) addr =  addr + 1; // each valid position we increment the address
      else begin
        if (hcount + 1 >= x && hcount + 1 < x + WIDTH) if (vcount + 1 >= y && vcount + 1 < y + HEIGHT) begin
          if (addr < 1) begin
            if (addr < 1) addr =  addr + 1; // if it's not a key press
          end
          else pixel <= 0;
        end
        else pixel <= 0;
      end
    end
    end
end

endmodule
module waveform_graphic (clock, reset, audio, record, new_frame, hcount, vcount, pixel, sample_length, state, we, addrin, din);

parameter HEIGHT=128;
parameter WIDTH=512;
parameter XPOSITION=0;
parameter YPOSITION=0;
parameter x = XPOSITION;
input clock;
input reset;
input signed [15:0] audio;
input record;
input new_frame;
input [10:0] hcount;
input [9:0] vcount;
input [9:0] sample_length; // 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 (synchronous) and one for reading (asynchronous)
reg [15:0] dim;
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(
  addrin(addrin),
  addrb(addrout),
  clk(a clock),
  clkb(clock),
  din(din),
  dout(dout),
  we(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 : begin
        addrin <= 9'h0;
        we <= 0;
        sum_pos <= 0;
        // Data processing
      end
      // More state transitions...
    endcase
  end
end
ACCUmulate: begin
    if (audio > 0)
        sumpos <= sumpos + audio;  // write the positive data one sum
    else
        sumneg <= sumneg + audio;   // and the negative to the other
    we <= 0;  // add and data latched
end

STORE: begin
    dis <= {sumpos[22:16], sumneg[22:16]};  // bit shift, note that sign is preserved
    we <= 1;
    sumpos <= 0;
    sumneg <= 0;
end

STEP: begin
    we <= 0;  // complete write
    addrin <= addrin + 1;  // increment address
    sumpos <= 0;
    sumneg <= 0;
end
case
    oldstate <= state;
    oldframe <= frame;
end

Listing 28: Verilog file xvgav
// sync and blanking
wire next_blank, next_blank;
assign next_blank = hreset ? 0 : hblank ? 1 : hblank;
assign next_blank = vreset ? 0 : vblank ? 1 : vblank;
always @ (posedge vclock) begin
  hcount <= hreset ? 0 : hcount + 1;
  hblank <= next_blank;
  hsync <= hsync ? 0 : hsyncoff ? 1 : hsync; // active low
  vcount <= vreset ? (vreset ? 0 : vcount + 1) : vcount;
  vblank <= next_vblank;
  vsync <= vsyncoff ? 0 : vsync; // active low
  blank <= next_vblank | (next_blank & hreset);
endmodule