The Sound of Music – Gone Digital!

Sarah-Jean Cunningham
Anne Romeo

6.111 Introductory Digital Systems Laboratory

14th December 2005

Abstract

This report describes the implementation in Verilog of a Digital Musical Tutorial Guide. The goal of the project is to design and implement an animated music tutorial for young children. The user will be able to associate a certain sound/note, within a range of a couple of octaves, with its name with the help of an associated video figure. The sequence of notes inputted by the user is stored in a memory unit, and can be played back on request. The notes are sent to a module which synthesises the notes of a piano and outputs the sound. A directory was created containing 14 different images, each corresponding to a note from Do to Ti. There will also be a game which tests perfect pitch. Here, the system will output a random note and ask the user to guess which note it is. A message is then displayed informing the user whether they got it right or not. The final

Contents

List of Figures

List of Tables

1 INTRODUCTION

1.1 Basic Overview

The idea behind the project came from a discussion on what helps people to learn how to read/recognise music. It is also true that this is a skill that like many things is a lot more easily obtained during childhood. So the idea is to produce a scheme that teaches children how to read, play and recognise musical notes.

A large portion of the project will be the actual synthesis of the notes. The aim is to produce a piano like sound. The secondary part of the project will be the visual display. This will play a key part in getting the children to form a connection between sound and sheet music.

1.2 Basic Architecture

The system must accept player inputs via a keyboard and act accordingly. The basic top level architecture is shown below in Figure 1.

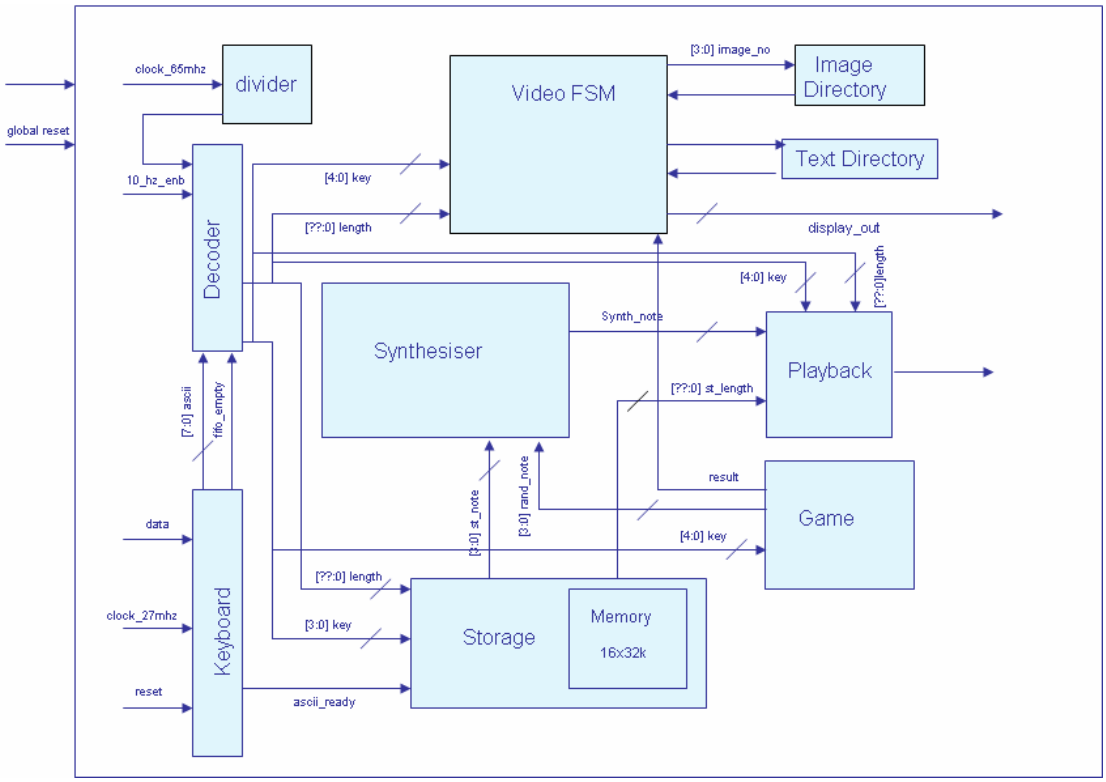

Figure 1: System Block Diagram

2 MODULE DESCRIPTION AND IMPLEMENTATION

2.1 Synchronizer

This module synchronizes the system 27MHz signals with the 65MHz clock used for the Video Modules, using two consecutive MUX's

2.2 Keyboard (AR)

The keyboard module is the interface between the PS/2 keyboard used to input the notes and various functional commands and our system. It outputs an 8 bit ascii code, which is then translated into a 4 bit number corresponding to the note being played, or to a 1 bit signal by the decoder module, depending on which key has been pressed.
This module has been implemented using the *ps2_kbd.v* code available on the course webpage.

2.3 Decoder (SJC)

The decoder is responsible for taking the ASCII code inputs from the keyboard and converting them into the required, corresponding signals e.g. notes, record and playback options. The other function of the decoder is to count the length of time a note has been played for.

As part of the original design, fifo_empty (a signal produced by the keyboard which is low all the while a key is being pressed) would be used by the decoder as a flag to alert us when a key has just been pressed and for how long it has been pressed. The decoder converts all the 'notes' into continuous signals (as long as they are being played). It converts the functional

inputs into one clock pulse long signals. A ready signal is also produced to flag when a note and its length are ready to be taken and stored by the memory unit.

2.4 AC97 Interface Modules (AR)

These modules are used to send the notes synthesized by the synthesizer module to the ac97 for playback. They are the same ones as the ones used in Lab 3.

2.5 Ten Hz Divider (AR)

In order to determine the length of the notes in $1/10^{th}$ of a second, a divider is used to send an enable signal every ten hertz to count the time between two different notes, via a *count* register.

2.6 Recorder/Playback or Storage Module (AR)

The recorder module (storage.v) is used in Recording and Playback Modes. All the notes the user plays after having pressed the Record button ('Enter' on the keyboard) are stored in a BRAM, as well as the length of time the note has been played for. The user exits the Recording Mode by pressing the Record button a second time.
The user then enters the Playback Mode by pressing the spacebar once on the keyboard. The content of the memory is played back, at the stored note length, until all the memory contents have been played.
This way of recording doesn't allow for the pauses between the notes to be taken into consideration (see 2.5). In order to do so, the Keyboard Module would have to be modified to output a 'key release' signal that would trigger a 'silence' note from the decoder.

2.7 32kx16 BRAM (AR)

The first 4 bits of each memory address are used to store the 4-bit binary number corresponding to the played note. The remaining 12 bits store the length during which that note has been played, in $1/10^{th}$ sec. Each time a different note is played, the memory address is incremented, storing in a register *addr_max* the maximum address used while recording.

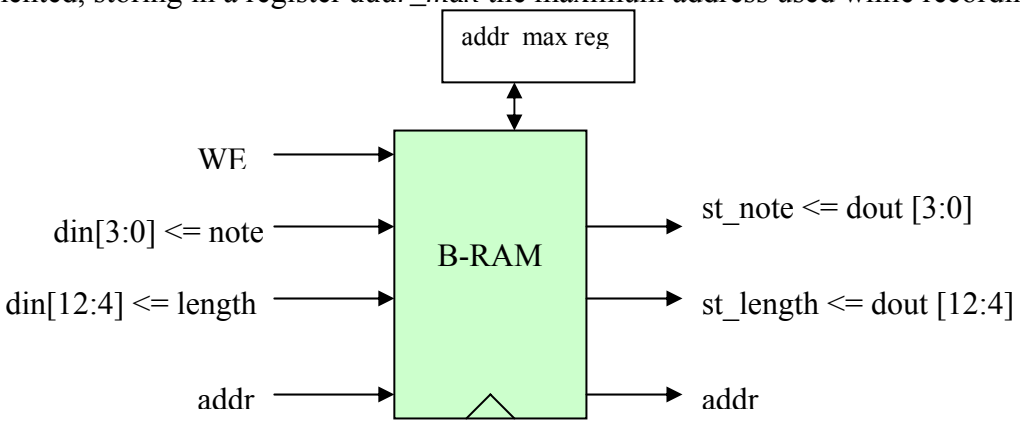

Figure 2: Recorder RAM inputs and outputs

2.8 Finite State Machine (AR)

To implement this Recorder/Playback module, a 3-state FSM is used (see table 1 and figure 2). An additional state has been added to clean the memory on Reset, by looping through the memory addresses up to addr_max and writing 0 in them.
A *ready* signal of duration one clock cycle is sent by the Decoder Module each time a different note is pressed on the keyboard, triggering the increment of the memory address.

State	Memory address	Write Enable	Note	Length
WAIT	addr <= 0	we <= 0	din[3:0] <= 4'b1111 // no sound	
REC	if (ready) addr <= addr + 1	we <= 1	if (ready) din[3:0] <= note	if (ready) din[12:4] <= 1 if (ten_hz_enb) din[12:4] <= din[12:4] + 1
PLAY	when st_length expires, addr <= addr + 1	we <= 0	dout[3:0] <= st_note	if (ten_hz_enb) count <= count + 1 // until count == st_length

Table 1: Recorder FSM state details

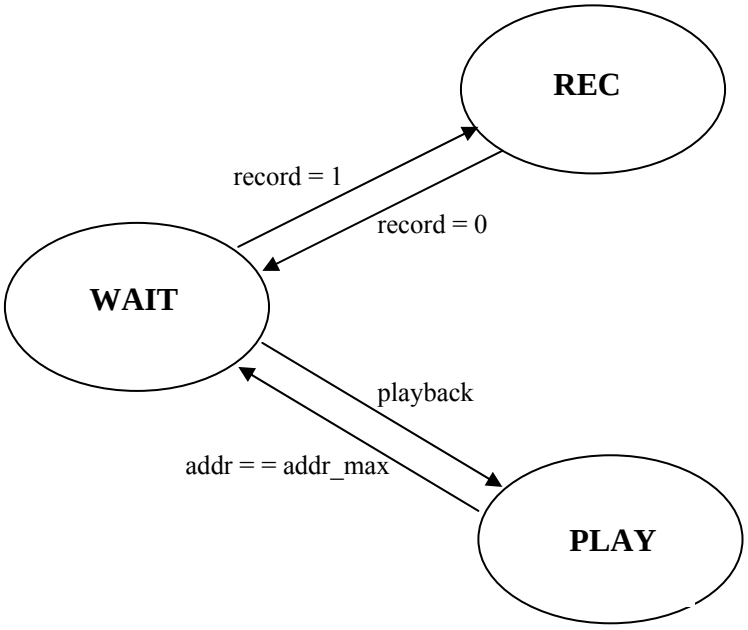

Figure 3: Recorder State Diagram

2.9 On/Off Button Module (AR)

In order for the Storage Module to work, we need 'record' to be high for the whole duration of the recording, and similarly with 'playback' during the playback. To do so, this On/Off button module was used: the 1st time the button is pressed, its corresponding signal's value is set to high. The next time the button is pressed, the signal is set to low. The outputs from this module are the signals *record_new* and *playback_new*.

2.10 Synthesizer (AR)

This module, *synthesizer.v*, takes in the note values sent by the Decoder Module (Live Playback), the Storage Module (Playback Mode) and the Game Module (Easy or Hard Game Modes) and creates the corresponding music note. This is done using Direct Digital Synthesis, or DDS.

2.10.1 DDS IPCore Module

The IPCore module provided by Xilinx is as follows:

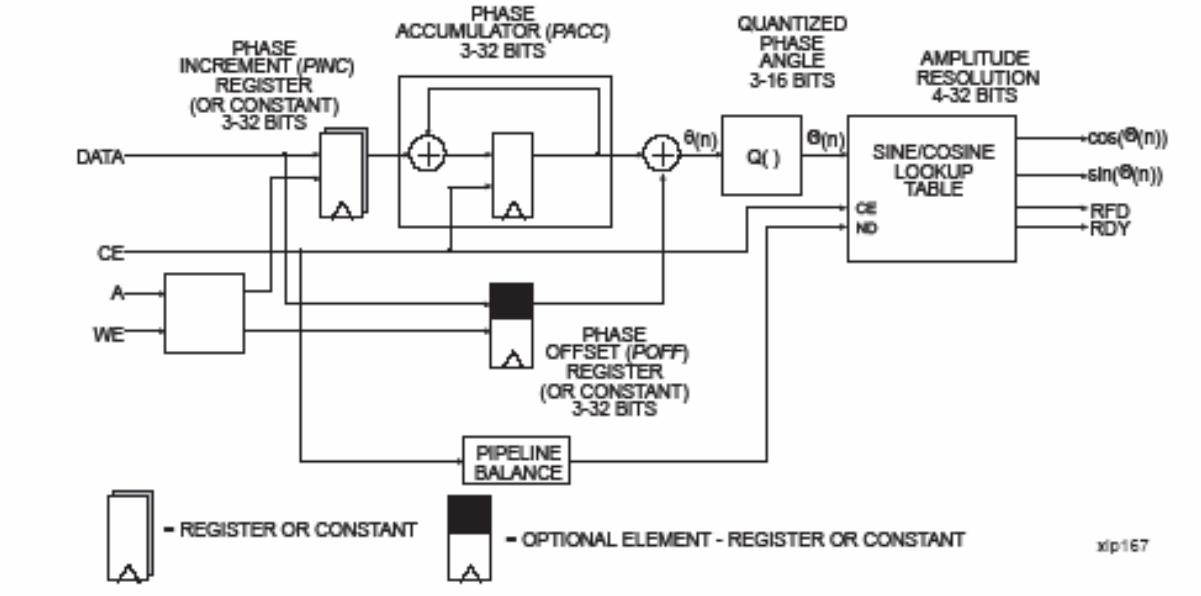

Figure 4: DDS Core (detailed view), Xilinx LogiCore Product Specification, fig 16

A musical note is composed of a linear combination of several signals: one at the fundamental frequency (the Fundamental), as well as its scaled versions at multiple integers of this frequency: the Harmonics. The aim was originally to use the multi-channel DDS to create the Harmonics of a note as well as its Fundamental, and then appropriately scale the Harmonics until a sound closer to that of a keyboard could be outputted. For time reasons however, we unfortunately had to limit ourselves to the single-channel, Fundamental only, case.

For efficiency, instead of storing the different signals in a look-up table style memory, only one DDS module is used. Its frequency is therefore changed depending on the inputted note. The output frequency f_{out} is defined as follows for the Xilinx IPCore:

$$f_{out} = \frac{f_{clk}\Delta\theta}{2^{B_{\Theta(n)}}} \text{ Hz}$$

with the clock enable rate f_{clk}, the phase increment $\Delta\theta$ and the number of bits in the phase accumulator $B_{\theta(n)}$.

The Phase Increment (PInc) however is the programmable parameter of the DDS IPCore module, and has to be determined in order to set the desired output frequency f_{out}. We have, for the phase increment $\Delta\theta$:

$$\Delta\theta = \frac{f_{out} 2^{B_{\theta(n)}}}{f_{clk}}$$

with notation as defined above.

In order to program the phase increment into the DDS module, the needed phase increment value *pinc* is sent to a channel address in the DDS via a Data Bus. The *dds_chan* port is used to write the phase increment or the phase offset into the right channel, using the memory map below (for our purposes, the phase offset is kept at zero). Without the harmonics (single channel), *dds_chan* therefore has to be set at 10000 at all times.

Address	Location Description
00000 - 01111	Phase Increment (PINC) values for channel 0 through 15
10000 - 11111	Phase Offset (POFF) values for channel 0 through 15

Table 2: Phase Increment and Phase Offset Memory Map (*dds_chan* value)

The DDS module needs a different sampling clock than the 27MHz one used for the rest of the system. In order to have a reasonable sound quality even at high frequency notes (and harmonics), we settled on a 0.1MHz clock, produced once again using a divider, of $1/100^{th}$ of second this time.

4.1 Synthesizer logic

With this f_{clk} = 0.1MHz, the corresponding $B_{\theta(n)}$ is given to be 27 bits. With these values, the notes' phase increments are as in table 3:

Note played	Corresponding 4-bit number	Frequency (Hz)	Phase increment (*pinc*)
Low DO (C)	0000	262	351650
Low RE (D)	0001	294	394600
Low MI (E)	0010	330	442918
Low FA (F)	0011	349	468419
Low SO (G)	0100	392	526133
Low LA (A)	0101	440	590558
Low TI (B)	0110	494	663035
Middle DO (C)	0111	524	701958
Middle RE (D)	1000	588	789200
Middle MI (E)	1001	660	885837
Middle FA (F)	1010	698	936839

Middle SO (G)	1011	784	1052266
Middle LA (A)	1100	880	1181116
Middle TI (B)	1101	988	1326071
High DO (C)	1110	1048	1406600

Table 3: Notes Value and Phase Increment

The notes needed during the three different modes (live playback, recorder playback and game) are grouped into a single register *played_note*. The notes are outputted continuously by the different modules while being played, for the right amount of time. Choosing which phase increment to select is thus determined by a simple `case` statement, with 'no output' or 'silence' as a default (when no input or when note is 4'b1111).

2.11 Video (SJC)

The function of the VGA interface is to output the valid pixel data to the screen, interfacing with the Video DAC and the monitor itself. A block diagram of the system is shown in Figure 5.

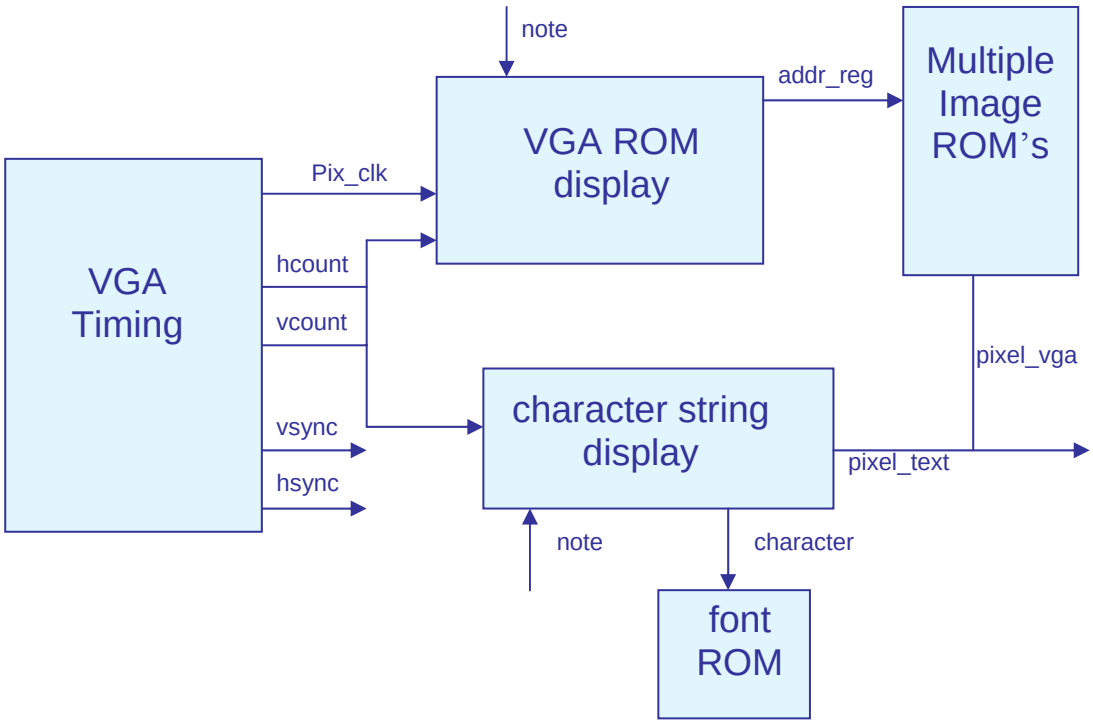

Figure 5: A Block Diagram of the VGA Interface

The outputs to the DACs require a Pixel Clk (whose rate is governed by the required resolution) in addition to three 8-bit colour lines (red, green and blue [rbg) for a 24-bit spectrum of colours.

The video interface must also time the horizontal and vertical sync signals to ensure a correct picture is displayed. A 320x240 resolution was used and since the lowest resolution on the monitor was 640x480 the counts were run as normal but displaying the horizontal lines twice for the vertical resolution. For a normal clock rate of 50MHz for a 640x480 display this

meant that a pixel clock of 25MHz was required for the timing. This was achieved using a pulse every four clock cycles (since this is being run at the system clock rate of 50MHz).

2.11.1 Images

The VGA display was used for the output of text as well as images (obtained from ROM's). In order to store an image in a ROM, it needs to be a COE file. As a result, initially all of the required images had to firstly be converted into .pgm files and then by running a script on this file using PYTHON, convert each into a COE file. Then each COE file was converted into a 1-bit wide memory. This was achieved by replacing all of the hex 0's with 1's and the FF's with 0's (working with black and white images only). The VGA timing supplies an x and y pixel at each count. In order to double the size of the image stored, a register called raddr was created with the function to pick out every other pixel vertically, and assign it a value w (width of image), 2w, 3w respectively. Every other horizontal pixel is picked out also.

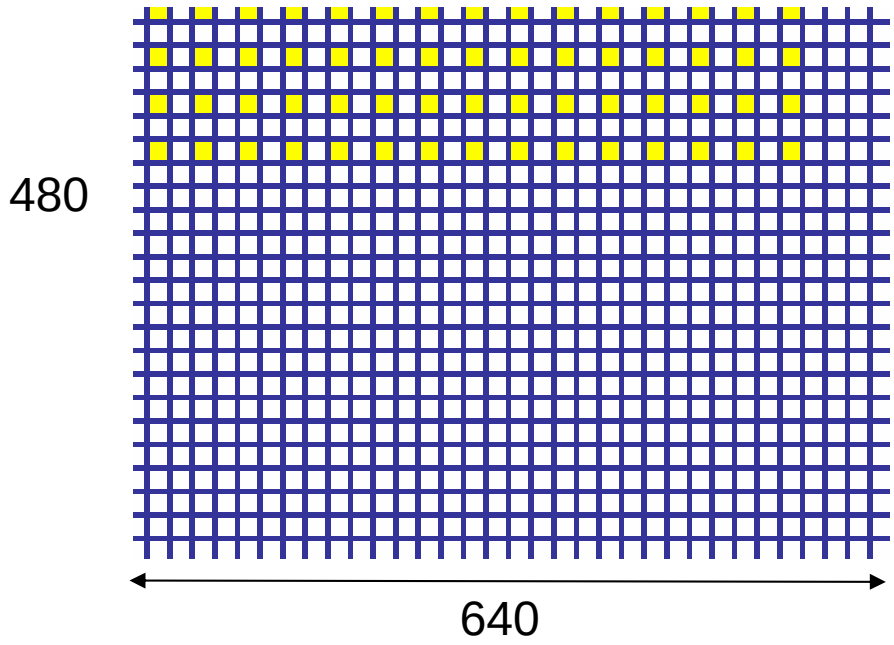

Figure 6: Only assigns alternate pixels, thus doubling the size of the image

2.11.2 Text

The text module converts a character string into a pixel display. Depending on what note was input, the corresponding Note character would be displayed. Text was also used for the Welcome Screen, and potentially would have been used to display instructions to different modes of the project

2.12 Game (SJC)

As described previously, the idea of the game is to test pitch recognition. The module simply generates a random note, which it sends to the synthesizer. It then compares the user's input (answer) and calculates whether it is correct or not. This result is then sent to the Video Module which then outputs a corresponding display.

3 TESTING AND DEBUGGING

3.1 The General Strategy

Testing and debugging are carried out in two main stages, namely Simulation and Hardware Testing.

In simulation, the computer is used to simulate the behaviour of individual modules and groups of modules. Theses simulations allow the removal of many of the bugs in simple to medium complexity modules. Unfortunately, simulations are not very useful for certain modules, especially those with large amounts of data variables or those with very long running times.

After all the suitable modules have been simulated and their bugs removed, simple collections of modules can be tested using the labkit itself. It is important to test them in small groups so that bugs can quickly be found.

3.2 Debugging Methods Used

3.2.1 Simulation, Test Bench, Logic Analyser

The first step to testing is usually by creating a Test bench Waveform and carrying out a simulation. This is a very useful way of checking whether the module performs the functions it is supposed to. Although it is not rare for a module to work fine during simulation and then not to function as it should once it has been loaded into the labkit, or vice versa.

3.2.2 Progressive Implementation

When a bug is found in a complex test, first make sure that those modules that are thought to be working correctly actually are. The second step is to remove modules and disable features until the module behaves as expected, and then to add them back one by one until it no longer works. Thus the last feature that is added is the one that is causing the problem.

4 PROBLEMS ENCOUNTERED

We encountered quite a few problems during this project, most of them while putting all the modules together. We initially lost a lot of time trying to use the files already provided on the course WebPages, such as the ones for the keyboard. This only worked as a separate project but wouldn't when incorporated into our project. Most of the problems in this particular case were solved by putting all the files into a new project. This simple manoeuvre was a solution to a lot of problems that made us waste several days…

We also had a few problems with the different clocks used for synchronising the modules. We indeed initially only wanted to use a single 65MHz clock, but a lot of clock cycles were wasted in that way as all the audio modules only need 27MHz. This was solved by using two different clocks: 27MHz for all the modules except the Video, and 50 MHz for the Video, more appropriate than 65MHz. To avoid glitches between the two clocks and timing discrepancies with the different inputs, we added double MUX's to filter and synchronise those of the inputs that were shared with the Video (synchronise module).

The main problem encountered while implementing the Video modules was due to the bad original conversion of the .bmp files to .coe. These images were indeed not entirely monochrome, which caused the script not to run properly.

Several problems came from the fact that the signal *fifo_empty* coming from the keyboard module didn't do what we had expected it to (see section 2.3). This was solved by replacing this signal by a *trigger* signal, high for one clock pulse when a different key is pressed on the keyboard. This has the major inconvenient of not enabling the same note to be played twice in a row, but the lack of time didn't permit us to find an alternative.

For the demonstration, all the modules worked independently. Unfortunately, we weren't able to fix all the timing problems that arose, especially with the Game Module and the Recorder/Playback Module. These problems might be due to a bad timing between the *ten_hz_enb* signal, the *ready* signal, and the exact clock pulse the recording was done.

5 CONCLUSION

The outcome of the project was much mitigated. On one hand, we managed to create a very interesting project that created sound and images from scratch and that was beyond anything we had done before using Hardware. On the other hand though, the multitude of problems encountered, the frustration arising from them as well as the lateness taken on our originally tight schedule prevented us from completing the implementation of all the different modules and forced us to sacrifice some of our main aims, such as the incorporation of harmonics in the sound to make them more like a keyboard, or the creation of a welcome and instruction screen. Better time management would be key to solving a lot of these issues.

APPENDIX

A

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

```
module audio (clock_27mhz, reset, audio_in_data, audio_out_data, ready,
            audio_reset_b, ac97_sdata_out, ac97_sdata_in,
              ac97_synch, ac97_bit_clock);

   parameter VOLUME = 4'd28;  //a reasonable volume value

   input clock_27mhz;
   input reset;
   output [19:0] audio_in_data;
   input [19:0] audio_out_data;
   output ready;

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

   wire [4:0] volume;
   wire source;
   assign volume = VOLUME;
   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 [9:0] reset_count;

   //wait a little before enabling the AC97 codec
   always @(posedge clock_27mhz) begin
      if (reset) begin
         audio_reset_b = 1'b0;
         reset_count = 0;
      end else if (reset_count == 1023)
        audio_reset_b = 1'b1;
      else
        reset_count = reset_count+1;
   end

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

   ac97commands cmds(clock_27mhz, ready, command_address, command_data,
                     command_valid, volume, source);

   // assign left_out_data = {audio_out_data, 12'b000000000000};
   assign left_out_data = audio_out_data;
   assign right_out_data = left_out_data;

   //arbitrarily choose left input
   // assign audio_in_data = left_in_data[19:12];
   assign audio_in_data = left_in_data;
```

```
endmodule

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

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

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

   reg ready;

   reg ac97_sdata_out;
   reg ac97_synch;

   reg [7:0] bit_count;

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

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

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

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

   always @(posedge ac97_bit_clock) begin
      // Generate the sync signal
```

```
      if (bit_count == 255)
        ac97_synch <= 1'b1;
      if (bit_count == 15)
        ac97_synch <= 1'b0;

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

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

      if ((bit_count >= 0) && (bit_count <= 15))
        // Slot 0: Tags
        case (bit_count[3:0])
          4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
          4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
          4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
          4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
        4'h4: ac97_sdata_out <= l_right_v; // Right data valid
          default: ac97_sdata_out <= 1'b0;
        endcase

      else if ((bit_count >= 16) && (bit_count <= 35))
        // Slot 1: Command address (8-bits, left justified)
        ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;

      else if ((bit_count >= 36) && (bit_count <= 55))
        // Slot 2: Command data (16-bits, left justified)
        ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;

      else if ((bit_count >= 56) && (bit_count <= 75))
        begin
           // Slot 3: Left channel
           ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
           l_left_data <= { l_left_data[18:0], l_left_data[19] };
        end
      else if ((bit_count >= 76) && (bit_count <= 95))
        // Slot 4: Right channel
           ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] :
1'b0;
      else
        ac97_sdata_out <= 1'b0;

      bit_count <= bit_count+1;

   end // always @ (posedge ac97_bit_clock)

   always @(negedge ac97_bit_clock) begin
```

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

endmodule

// issue initialization commands to AC97
module ac97commands (clock, ready, command_address, command_data,
                     command_valid, volume, source);

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

   reg [23:0] command;
   reg command_valid;

   reg old_ready;
   reg done;
   reg [3:0] state;

   initial begin
      command <= 4'h0;
      // synthesis attribute init of command is "0";
      command_valid <= 1'b0;
      // synthesis attribute init of command_valid is "0";
      done <= 1'b0;
      // synthesis attribute init of done is "0";
      old_ready <= 1'b0;
      // synthesis attribute init of old_ready is "0";
      state <= 16'h0000;
      // synthesis attribute init of state is "0000";
   end

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

   wire [4:0] vol;
   assign vol = 31-volume;

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

      case (state)
        4'h0: // Read ID
          begin
             command <= 24'h80_0000;
             command_valid <= 1'b1;
          end
        4'h1: // Read ID
          command <= 24'h80_0000;
        4'h3: // headphone volume
```

```
          command <= { 8'h04, 3'b000, vol, 3'b000, vol };
        4'h5: // PCM volume
          command <= 24'h18_0808;
        4'h6: // Record source select
          command <= 24'h1A_0000; // microphone
        4'h7: // Record gain = max
        command <= 24'h1C_0F0F;
        4'h9: // set +20db mic gain
          command <= 24'h0E_8048;
        4'hA: // Set beep volume
          command <= 24'h0A_0000;
        4'hB: // PCM out bypass mix1
          command <= 24'h20_8000;
        default:
          command <= 24'h80_0000;
      endcase // case(state)

      old_ready <= ready;

   end // always @ (posedge clock)

endmodule // ac97commands

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

module tone750hz (clock, ready, pcm_data);

   input clock;
   input ready;
   output [19:0] pcm_data;

   reg rdy, old_ready;
   reg [8:0] index;
   reg [19:0] pcm_data;

   initial begin
      old_ready <= 1'b0;
      // synthesis attribute init of old_ready is "0";
      index <= 8'h00;
      // synthesis attribute init of index is "00";
      pcm_data <= 20'h00000;
      // synthesis attribute init of pcm_data is "00000";
   end

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

   // one cycle of a sinewave in 64 20-bit samples
   always @(index) begin
      case (index[5:0])
        6'h00: pcm_data <= 20'h00000;
```

```
6'h01: pcm_data <= 20'h0C8BD;
6'h02: pcm_data <= 20'h18F8B;
6'h03: pcm_data <= 20'h25280;
6'h04: pcm_data <= 20'h30FBC;
6'h05: pcm_data <= 20'h3C56B;
6'h06: pcm_data <= 20'h471CE;
6'h07: pcm_data <= 20'h5133C;
6'h08: pcm_data <= 20'h5A827;
6'h09: pcm_data <= 20'h62F20;
6'h0A: pcm_data <= 20'h6A6D9;
6'h0B: pcm_data <= 20'h70E2C;
6'h0C: pcm_data <= 20'h7641A;
6'h0D: pcm_data <= 20'h7A7D0;
6'h0E: pcm_data <= 20'h7D8A5;
6'h0F: pcm_data <= 20'h7F623;
6'h10: pcm_data <= 20'h7FFFF;
6'h11: pcm_data <= 20'h7F623;
6'h12: pcm_data <= 20'h7D8A5;
6'h13: pcm_data <= 20'h7A7D0;
6'h14: pcm_data <= 20'h7641A;
6'h15: pcm_data <= 20'h70E2C;
6'h16: pcm_data <= 20'h6A6D9;
6'h17: pcm_data <= 20'h62F20;
6'h18: pcm_data <= 20'h5A827;
6'h19: pcm_data <= 20'h5133C;
6'h1A: pcm_data <= 20'h471CE;
6'h1B: pcm_data <= 20'h3C56B;
6'h1C: pcm_data <= 20'h30FBC;
6'h1D: pcm_data <= 20'h25280;
6'h1E: pcm_data <= 20'h18F8B;
6'h1F: pcm_data <= 20'h0C8BD;
6'h20: pcm_data <= 20'h00000;
6'h21: pcm_data <= 20'hF3743;
6'h22: pcm_data <= 20'hE7075;
6'h23: pcm_data <= 20'hDAD80;
6'h24: pcm_data <= 20'hCF044;
6'h25: pcm_data <= 20'hC3A95;
6'h26: pcm_data <= 20'hB8E32;
6'h27: pcm_data <= 20'hAECC4;
6'h28: pcm_data <= 20'hA57D9;
6'h29: pcm_data <= 20'h9D0E0;
6'h2A: pcm_data <= 20'h95927;
6'h2B: pcm_data <= 20'h8F1D4;
6'h2C: pcm_data <= 20'h89BE6;
6'h2D: pcm_data <= 20'h85830;
6'h2E: pcm_data <= 20'h8275B;
6'h2F: pcm_data <= 20'h809DD;
6'h30: pcm_data <= 20'h80000;
6'h31: pcm_data <= 20'h809DD;
6'h32: pcm_data <= 20'h8275B;
6'h33: pcm_data <= 20'h85830;
6'h34: pcm_data <= 20'h89BE6;
6'h35: pcm_data <= 20'h8F1D4;
6'h36: pcm_data <= 20'h95927;
6'h37: pcm_data <= 20'h9D0E0;
6'h38: pcm_data <= 20'hA57D9;
6'h39: pcm_data <= 20'hAECC4;
6'h3A: pcm_data <= 20'hB8E32;
6'h3B: pcm_data <= 20'hC3A95;
6'h3C: pcm_data <= 20'hCF044;
6'h3D: pcm_data <= 20'hDAD80;
```

```
        6'h3E: pcm_data <= 20'hE7075;
        6'h3F: pcm_data <= 20'hF3743;
      endcase // case(index[5:0])
   end // always @ (index)
endmodule

//////////////////////////////////////////////////////////////////////////////
/////
//
// Pushbutton DebounceVideo Module (video version)
//
//////////////////////////////////////////////////////////////////////////////
/////

module debounceVideo1 (reset, clock_65mhz, noisy, clean);
   input reset, clock_65mhz, noisy;
   output clean;

   reg [19:0] count;
   reg new, clean;

   always @(posedge clock_65mhz) begin
     if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != new) begin new <= noisy; count <= 0; end
     else if (count == 650000) clean <= new;
     else count <= count+1;
   end
endmodule


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

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


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

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

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

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

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

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

              clock_feedback_out, clock_feedback_in,

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

              rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

              mouse_clock, mouse_data, keyboard_clock, keyboard_data,

              clock_27mhz, clock1, clock2,

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

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

              switch,

              led,

              user1, user2, user3, user4,

              daughtercard,

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

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

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

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

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

   input  [19:0] tv_in_ycrcb;
   input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
tv_in_aef,
        tv_in_hff, tv_in_aff;
   output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
        tv_in_reset_b, tv_in_clock;
   inout  tv_in_i2c_data;

   inout  [35:0] ram0_data;
```

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

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

   input  clock_feedback_in;
   output clock_feedback_out;

   inout  [15:0] flash_data;
   output [23:0] flash_address;
   output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
   input  flash_sts;

   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;

   input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

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

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

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

   inout [43:0] daughtercard;

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

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


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

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

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

```
   //
   // reset Generation
   //
   // A shift register primitive is used to generate an active-high reset
   // signal that remains high for 16 clock cycles after configuration
finishes
   // and the FPGA's internal clocks begin toggling.
   //

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

   wire initial_reset;
   SRL16 reset_sr (.D(1'b0), .CLK(clock_27mhz), .Q(initial_reset),
            .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;

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

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

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

   // SRAMs
   assign ram0_data = 36'hZ;
   assign ram0_address = 19'h0;
   assign ram0_adv_ld = 1'b0;
```

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

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

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

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

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

   // Buttons, Switches, and Individual LEDs
   // button0, button1, button2, button3, button_enter, button_right,
   // button_left, button_down, button_up, and switches are inputs

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

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

   // SystemACE Microprocessor Port
   assign systemace_data = 16'hZ;
```

```
   assign systemace_address = 7'h0;
   assign systemace_ce_b = 1'b1;
   assign systemace_we_b = 1'b1;
   assign systemace_oe_b = 1'b1;
   // systemace_irq and systemace_mpbrdy are inputs

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

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

// AC97 driver
/*   audio a(clock_27mhz, reset, from_ac97_data, to_ac97_data, ready,
         audio_reset_b, ac97_sdata_out, ac97_sdata_in,
         ac97_synch, ac97_bit_clock);
   */


   // use FPGA's digital clock manager to produce a 50 Mhz clock from 27
Mhz
   // actual frequency: 49.85 MHz
   wire clock_50mhz_unbuf,clock_50mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_50mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 13
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
   BUFG vclk2(.O(clock_50mhz),.I(clock_50mhz_unbuf));



////////////////////////////////////////////////////////////////////////
/////////
 // Keyboard implementation
////////////////////////////////////////////////////////////////////////
/////////
  /*
   // power-on reset generation
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_srk (.D(1'b0), .CLK(clock_27mhz), .Q(power_on_reset),
               .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_srk.INIT = 16'hFFFF;
   */

   // ENTER button is user reset
   wire reset,user_reset;
   debounce db5(initial_reset, clock_27mhz, ~button1, user_reset);

   // reset is Kreset (from Keyboard), initial_reset and user_reset
(button1)
   wire Kreset;
   wire out;
   wire out_new;
```

```
   assign reset = (Kreset | initial_reset | user_reset);

   buttonOnOff out1(clock_27mhz,reset,out,out_new);


   // keyboard input
   wire [7:0] ascii;
   wire       char_rdy;
   ps2_ascii_input kbd(clock_27mhz, reset, keyboard_clock,
                       keyboard_data, ascii, char_rdy);

   // route keyboard input to display
   reg [3:0] count = 3;//15;
   reg [7:0] last_ascii;
  // reg [16*8-1:0] cstring;
   //reg [4*8-1:0] cstring;

   always @(posedge clock_27mhz)
     begin
      count <= reset ? 15 : (char_rdy ? count-1 : count);
      last_ascii <= char_rdy ? ascii : last_ascii;
     end

    /*always @(posedge clock_27mhz)
      begin
       cstring[7:0] <= (count==0) ? last_ascii : cstring[7:0];
       cstring[7+'o10:'o10] <= (count==1) ? last_ascii:
cstring[7+'o10:'o10];
       cstring[7+'o20:'o20] <= (count==2) ? last_ascii:
cstring[7+'o20:'o20];
       cstring[7+'o30:'o30] <= (count==3) ? last_ascii:
cstring[7+'o30:'o30];
       cstring[7+'o40:'o40] <= (count==4) ? last_ascii:
cstring[7+'o40:'o40];
       cstring[7+'o50:'o50] <= (count==5) ? last_ascii:
cstring[7+'o50:'o50];
       cstring[7+'o60:'o60] <= (count==6) ? last_ascii:
cstring[7+'o60:'o60];
       cstring[7+'o70:'o70] <= (count==7) ? last_ascii:
cstring[7+'o70:'o70];

       cstring[7+'o100:'o100] <= (count==8) ? last_ascii:
cstring[7+'o100:'o100];
       cstring[7+'o110:'o110] <= (count==9) ? last_ascii:
cstring[7+'o110:'o110];
       cstring[7+'o120:'o120] <= (count==10) ? last_ascii:
cstring[7+'o120:'o120];
       cstring[7+'o130:'o130] <= (count==11) ? last_ascii:
cstring[7+'o130:'o130];
       cstring[7+'o140:'o140] <= (count==12) ? last_ascii:
cstring[7+'o140:'o140];
       cstring[7+'o150:'o150] <= (count==13) ? last_ascii:
cstring[7+'o150:'o150];
       cstring[7+'o160:'o160] <= (count==14) ? last_ascii:
cstring[7+'o160:'o160];
       cstring[7+'o170:'o170] <= (count==15) ? last_ascii:
cstring[7+'o170:'o170];
      end     */

  // declare Record button module (AR)
```

```
    wire record;
    wire record_new;

   buttonOnOff Record1(clock_27mhz,reset,record,record_new);

    // declare Playback button module (AR)
    wire playback;
    wire playback_new;

   buttonOnOff Playback1(clock_27mhz,reset,playback,playback_new);

    // declare 10 Hz enable divider
    wire ten_hz_enb;
    wire trigger;

    divider ten_hz_enb1(clock_27mhz, reset, ten_hz_enb);

    // declare Decoder module (SJC)
    wire [3:0] note;
    wire [12:0] length;
    wire Ready;
    wire game_enb_hard;
    wire game_enb_easy;

   decoder
decoder1(ten_hz_enb,ascii,trigger,note,length,Ready,record,playback,
                 Kreset,game_enb_hard,game_enb_easy,clock_27mhz,reset);

    // declare 32kx16 storage Ram module (AR)
    wire [14:0] addr;       // memory address
    wire [15:0] din;        // data written to the ram
    wire [15:0] dout;       // data read from the ram
    wire        we;              // memory write enable, write = 1, read
= 0.

   ram_32kx16 recorderRam(addr,clock_27mhz,din,dout,we);

   // declare Storage module (AR)
    wire [3:0] st_note;


   storage
storage1(clock_27mhz,reset,note,Ready,record_new,playback_new,st_note,
                       addr,din,dout,we,ten_hz_enb);

   // AC97 driver

   wire [19:0] from_ac97_data, to_ac97_data;
   wire audio_ready;

   audio myaudio(clock_27mhz, initial_reset, from_ac97_data, to_ac97_data,
             audio_ready, audio_reset_b, ac97_sdata_out, ac97_sdata_in,
             ac97_synch, ac97_bit_clock);
   defparam myaudio.VOLUME = 4'd10;

   // declare test tone (750hz) module
   wire [19:0] pcm_data;

   tone750hz tone(clock_27mhz,audio_ready,pcm_data);

   // declare game module
```

```
   wire game = (game_enb_hard || game_enb_easy);
   wire result;
   wire [3:0] rand_note;

   game game1(clock_27mhz,rand_note,note,result,game);

   // declare synthesizer module
   wire [19:0] synth_note;
   wire [26:0] pinc;
   wire [4:0] dds_chan;
   wire [3:0] played_note;

   synthesizer
synthesizer1(clock_27mhz,reset/*,length*/,note,st_note,rand_note,synth_not
e,played_note,
                        playback_new,game,pinc,dds_chan/*,audio_ready*/);

   // if switch[7] is 0: test tone, if 1: synthesized note.
   assign to_ac97_data = switch[7] ? synth_note : pcm_data;

   ///// display ascii and other variables on hex dot LEDs


   //assign hex_data = cstring;
   wire [63:0] hex_data;
   display_16hex disp(reset, clock_27mhz, hex_data,
            disp_blank, disp_clock, disp_rs, disp_ce_b,
            disp_reset_b, disp_data_out);

   assign hex_data =
{played_note,4'b0,ascii,3'b0,playback_new,3'b0,record_new,2'b0,addr,3'b0,r
esult,18'b0,2'b0};//state};

   // display count of keyboard on LEDs,as well as reset and result
   assign led = {~reset,~record_new,~played_note,count[1:0]};
   assign analyzer2_data =
{Ready,we,din[3:0],dout[3:0],din[6:4],dout[6:4]};
   assign analyzer2_clock = clock_27mhz;
   assign analyzer1_data = {addr,1'b1};
   assign analyzer1_clock = clock_27mhz;

////////////////////////////////////////////////////////////////////////
//////////////
// transition between 27MHz clock and 65MHz
////////////////////////////////////////////////////////////////////////
//////////////
wire [3:0] noteV;
wire  resultV;
wire  game_enb_hardV;
wire game_enb_easyV;

synchroniser syncNote3(clock_50mhz,played_note[3],noteV[3]);
synchroniser syncNote2(clock_50mhz,played_note[2],noteV[2]);
synchroniser syncNote1(clock_50mhz,played_note[1],noteV[1]);
synchroniser syncNote0(clock_50mhz,played_note[0],noteV[0]);

synchroniser result1(clock_50mhz,result,resultV);
synchroniser reset1(clock_50mhz,reset,resetV);
synchroniser game_hard1(clock_50mhz,game_enb_hard,game_enb_hardV);
synchroniser game_easy1(clock_50mhz,game_enb_easy,game_enb_easyV);
```

```
//////////////////////////////////////////////////////////////////////////
/////////////
// VIDEO modules  (SJC)
//////////////////////////////////////////////////////////////////////////
/////////////
  wire [10:0] hcount;
 wire [9:0]  vcount;
  wire hsync,vsync,blank;

  // 640x480 VGA display
   wire [7:0] string_pixel;
   wire [7:0] img_pixel;
   wire [2:0] cdpixel;// cdpixel2;
   wire [7:0] pixel = string_pixel | img_pixel;
   wire       pix_clk;

   vga_sync vga1(clock_50mhz,hsync,vsync,hcount,vcount,pix_clk,blank);

   assign     vga_out_red = pixel;  // black and white display (for now)
   assign     vga_out_green = pixel;
   assign     vga_out_blue = pixel;
   assign     vga_out_blank_b = ~blank;
   assign     vga_out_sync_b = 1'b1;    // not used
   assign     vga_out_hsync = hsync;
   assign     vga_out_vsync = vsync;
   assign     vga_out_pixel_clock = pix_clk;     // vga pixel clock




   // vga rom image display
   wire [7:0] rpix;


   vga_romdisp vr(noteV, clock_50mhz,hcount,vcount,pix_clk,rpix);


   // video output

   wire       fpixel = (hcount==0 | hcount==639 | vcount==0 |
vcount==479);
   assign     img_pixel = blank ? 8'b0
                : ( switch[0] ? {hcount[7:5],5'b0}
                  : rpix | (fpixel ? 8'hff : 8'h00) );
   assign     string_pixel = (cdpixel) ? 8'hFF : 8'h00;




    // generate basic XVGA video signals

//  xvga xvga1(clock_65mhz,hcount,vcount,hsync,vsync,blank);


   // character display module: sample string in middle of screen
   // ="DO";

  // wire [2:0] cdpixel2;
  // wire [3:0] note;
```

```
   wire reset_game;
   //assign note = switch [7:4];
   reg [63:0] phrase=0;
  // reg [63:0] phrase2=0;
    always @(posedge clock_50mhz) begin
   case (noteV)
      4'b0000: begin //middlec
            phrase <= "DO";
            end
      4'b0001: begin
            phrase <= "RE";
            end
      4'b0010: begin
            phrase <= "MI"; //E
            end
      4'b0011: begin
            phrase <= "FA"; //F
            end
      4'b0100: begin
            phrase <= "SO";//G
            end
      4'b0101: begin
            phrase <= "LA"; //A
            end
      4'b0110: begin
            phrase <= "TI";//B
            end
      4'b0111: begin
            phrase <= "DO";//low C
            end
      4'b1000: begin
            phrase <= "RE"; //D
            end
      4'b1001: begin
            phrase <= "MI";//E
            end
      4'b1010: begin
            phrase <= "FA";//F
            end
      4'b1011: begin
            phrase <= "SO"; //G
            end
      4'b1100: begin
            phrase <= "LA";//A
            end
      4'b1101: begin
            phrase <= "TI"; //B
            end
      default: phrase <= "reset";

      endcase
      //if (reset_game) phrase2 <= "SOUND OF MUSIC - GONE DIGITAL!";
//    else phrase2<=0;
      end
      wire [15:0] cstring = phrase;
//    wire [239:0] cstring2 = "SOUND OF MUSIC - GONE DIGITAL!";

   char_string_display cd(clock_50mhz,hcount,vcount,
                  cdpixel,cstring,11'd20,10'd20);
      defparam cd.NCHAR = 2;
      defparam cd.NCHAR_BITS = 1;
```

```
 //char_string_display cd2(clock_50mhz,hcount,vcount,
//                  cdpixel2,cstring2,11'd512,10'd384);
//    defparam cd.NCHAR = 30;
//    defparam cd.NCHAR_BITS = 5;


  // assign game_enb_hard = switch [3];
  // assign game_enb_easy = switch [2];
   assign reset_game = ~button0;
  // assign result = ~button1;

  /* char_string_display cd(clock_65mhz,hcount,vcount,
                  cdpixel,cstring,11'd512,10'd384);
      defparam cd.NCHAR = 2;
      defparam cd.NCHAR_BITS = 1;          */

  /* // character display module: moving text
   wire [12*8-1:0] cstring2 = "ABCDEFGHIJKL";
   wire [2:0]  cdpixel2;
   reg [10:0]  cx2;
   reg [9:0]   cy2;
   char_string_display cd2(clock_65mhz,
                    hcount,vcount,cdpixel2,cstring2,cx2,cy2);
   defparam    cd2.NCHAR = 12;
   defparam    cd2.NCHAR_BITS = 4;

   // text movement
   parameter  PADMOVERATE = 060000;
   reg [31:0] pcount;
   wire       padflag = (pcount == PADMOVERATE) ? 1 : 0;

   always @(posedge clock_65mhz)
     begin
      pcount <= reset ? 0 : (padflag ? 0 : pcount + 1);
      cy2 <= reset ? 0 :
           (up & padflag) ? (cy2>0 ? cy2-1 : cy2) :
           ((down & padflag) ? (cy2+24<767 ? cy2+1 : cy2) : cy2);
      cx2 <= reset ? 0 :
           (left & padflag) ? (cx2>0 ? cx2-1 : cx2) :
           ((right & padflag) ? (cx2+16*12<1024 ? cx2+1 : cx2) : cx2);
     end
             */
   // switch[0] selects which video generator to use:
   //  00: text
   //  01: 1 pixel outline of active video area (adjust screen controls)
   //  10: color bars


  // assign led = ~{1'b0,up,down,left,right,reset,switch[1:0]};

   // display module for debugging

/*
    wire clock_27mhz_buf;
   BUFG vclk3(.O(clock_27mhz_buf),.I(clock_27mhz));

   wire [63:0] dispdata = {1'b0,cx2,2'b0,cy2,40'b0};
   display_16hex hexdisp1(reset, clock_27mhz_buf, dispdata,
            disp_blank, disp_clock, disp_rs, disp_ce_b,
            disp_reset_b, disp_data_out);
```

```
*/

endmodule

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

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

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

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   wire      hsyncon,hsyncoff,hreset,hblankon;
   assign    hblankon = (hcount == 1023);
   assign    hsyncon = (hcount == 1047);
   assign    hsyncoff = (hcount == 1183);
   assign    hreset = (hcount == 1343);

   // vertical: 806 lines total
   // display 768 lines
   wire      vsyncon,vsyncoff,vreset,vblankon;
   assign    vblankon = hreset & (vcount == 767);
   assign    vsyncon = hreset & (vcount == 776);
   assign    vsyncoff = hreset & (vcount == 782);
   assign    vreset = hreset & (vcount == 805);

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

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

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

// File:   char_string_display.v
// Date:   05 Dec 2005
// Author: SJ Cunningham
//
```

```
// Display an ASCII encoded character string in a video window at some
// specified x,y pixel location.
//////////////////////////////////////////////////////////////////////
////
//
// video character string display
//
//////////////////////////////////////////////////////////////////////
////

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

   parameter NCHAR = 8; // number of characters in string to display
   parameter NCHAR_BITS = 3; //number of bits to specify NCHAR

   input vclock;  // 65MHz clock
   input [10:0] hcount; // horizontal (x) location of current pixel
(0..1023)
   input [9:0]    vcount; //vertical (y) location of current pixel
(0..767)
   output [2:0] pixel;  // video pixel value to display at current
location
   input [NCHAR*8-1:0] cstring;     // character string to display (8 bit
ASCII for each char)
   input [10:0] cx;
   input [9:0]    cy;   // pixel location (upper left corner) to display
string at

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

   wire [10:0] hoff = (hcount-1-cx)>>2;   //shift hoff by 2 = multiply the
text to make it 4 times as large as the original font
   wire [9:0]     voff = (vcount-cy)>>2;
   wire [NCHAR_BITS-1:0] column = NCHAR-1-hoff[NCHAR_BITS-1+4:4];  // <
NCHAR
   wire [2:0]     h = hoff[3:1];            // 0 .. 7
   wire [3:0]     v = voff[4:1];            // 0 .. 11

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

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

   // generate character pixel if we're in the right h,v area
   wire [2:0] cpixel = (font_byte[7 - h] ^ reverse) ? 7 : 0;
   wire dispflag = ((hcount > cx) & (vcount >= cy) & (hcount <=
cx+NCHAR*64)      //16
                & (vcount < cy + 96));  //24
   wire [2:0] pixel = dispflag ? cpixel : 0;

endmodule
```

```
// File:   game.v
// Date:   07 Dec 2005
// Author: SJ Cunningham

module game(clk, rand_note,note, result);

    input clk;
    input [3:0] note; //input by user
    output result;      //result of game i.e. right or wrong
    output [3:0] rand_note;   //random not generated

    wire rand_note;
    reg [3:0] count; //reg to create continous count of 1-13
    reg result;

    always @ (posedge clk)
    begin
    //generate a random note between 0 and 13
      count <= (count>= 4'b1101)? 0:count + 1;

    //compare note with the random note and say whether it is correct or
not
    result <= (note == rand_note)? 1:0;

    end

//only look at the count when the game mode is selected, that way
//note is more or less random
    assign rand_note = count;


endmodule


// File:   vga_romdisp.v
// Date:   05 December 2005
// Author: SJ Cunningham
//
/
// This module reads data from ROM and creates the proper 8-bit pixel
// value for a pixel position defined by (hcount,vcount).  Note that
// there is a one cycle delay in reading from memory, so we may
// have a single pixel offset error here, to be fixed.  But the displayed
// image looks respectable nevertheless.
//
/
// The COE file is generated from an 8-bit PGM format image file (Black
and White)
// using pgm2coe.py

module vga_romdisp(note, clk,hcount,vcount,pix_clk,pixel);

   input [3:0] note;
   input clk;           // video clock
   input [9:0] hcount;  // current x,y location of pixel
   input [9:0] vcount;
   input pix_clk; // pixel clock
   output [7:0]pixel;   // pixel value output

   parameter middlec =  4'b0000;
   parameter middled = 4'b0001;
```

```
   parameter middlee = 4'b0010;
   parameter middlef http://web.mit.edu/6.111/www/f2004/index.html=
4'b0011;
   parameter middleg = 4'b0100;
   parameter middlea = 4'b0101;
   parameter middleb = 4'b0110;
   parameter lowc = 4'b0111;
   parameter lowd = 4'b1000;
   parameter lowe = 4'b1001;
   parameter lowf = 4'b1010;
   parameter lowg = 4'b1011;
   parameter lowa = 4'b1100;
   parameter lowb = 4'b1101;

  // reg [8:0] pixel;

   wire pixel_1;
   wire pixel_2;
   wire pixel_3;
   wire pixel_4;
   wire pixel_5;
   wire pixel_6;
   wire pixel_7;
   wire pixel_8;
   wire pixel_9;
   wire pixel_10;
   wire pixel_11;
   wire pixel_12;
   wire pixel_13;
   wire pixel_14;

   // the memory address is hcount/2 + vcount/2 * 320
   // (4 pixels per memory location, since image is 320x240, and
   // display is 640x480).

   reg [7:0] width = 0; //width of image to be used
   reg pixel_note;
   reg [15:0] raddr;


      // Only use every other pixel to output data, both Horizonatally and
Vertically
always @(posedge clk)
     raddr <= (hcount==0 & vcount==0) ? 0
            : (hcount==0 & pix_clk & ~vcount[0]) ? raddr + 320 : raddr;



wire [15:0]addr = {6'b0,hcount[9:0]} + raddr[15:0];

   always @(posedge clk)      begin

//assign pixel and width depending on what note is being used
   case (note)
   middlec: begin  //middlec
            width <= 219;
            pixel_note<=pixel_1;
            end

   middled: begin //middled
            width <= 213;
```

```
            pixel_note<=pixel_2;
            end

   middlee: begin  //middle e
            width <= 207;
            pixel_note<=pixel_3;
            end

   middlef:  begin//middle f
            width <= 212;
            pixel_note<=pixel_4;
            end

   middleg: begin //middle g
            width <= 211;
            pixel_note<=pixel_5;
            end

   middlea:  begin //middle a
            width <= 215;
            pixel_note<=pixel_6;
            end

   middleb:  begin //middle b
            width <= 214;
            pixelhttp://web.mit.edu/6.111/www/f2004/index.html_note <=
pixel_7;
            end

   lowc:  begin//low c
            width<=216;
            pixel_note<=pixel_8;
            end

   lowd:  begin//low d
            width <= 214;
            pixel_note<=pixel_9;
            end

   lowe:  begin//low e
            width <= 221;
            pixel_note<=pixel_10;
            end

   lowf:  begin//low f
            width <= 217;
            pixel_note<=pixel_11;
            end

   lowg: begin //low g
            width <= 212;
            pixel_note<=pixel_12;
            end

   lowa:  begin//low a
            width <= 217;
            pixel_note<=pixel_13;
            end

   lowb: begin//low b
            width <= 216;
```

```
            pixel_note<=pixel_14;
            end

   default: begin
            width <=215;
            pixel_note<=1;
            end

      endcase

raddr <= (hcount==0 & vcount==0) ? 0: (hcount==0 & pix_clk /*&
~vcount[0]*/) ? raddr + width : raddr;

   end

   reg [15:0]     addr_reg;
   always @(posedge clk) addr_reg <= addr;

// assign pixel video output
assign pixel = (hcount>=423 & hcount <=639 & vcount>=0 & vcount<=226)?
(pixel_note? 8'b00000011: 8'b11111111): 8'b00000000;


  // instantiate the image rom
   middlec img10(addr_reg,clk,pixel_1);
   middled img11(addr_reg, clk, pixel_2);
   middlee img12(addr_reg,clk,pixel_3);
   middlef img13(addr_reg,clk,pixel_4);
   middleg img14(addr_reg,clk,pixel_5);
   middlea img9(addr_reg,clk,pixel_6);
   middleb img8(addr_reg,clk,pixel_7);
   lowc img3(addr_reg,clk,pixel_8);
   lowd img4(addr_reg,clk,pixel_9);
   lowe img5(addr_reg,clk,pixel_10);
   lowf img6(addr_reg,clk,pixel_11);
   lowg img7(addr_reg,clk,pixel_12);
   lowa imgrom(addr_reg,clk,pixel_13);
   lowb img2(addr_reg,clk,pixel_14);
//

endmodule // vga_romdisp

// module to get signals such as record and playback to go to 1 when
pressed the 1st
//time and stay 1 until pressed a 2nd time - when it will go to 0 until
pressed again.

module buttonOnOff(clk,reset,signal_in,signal_out);
    input clk;
    input reset;
    input signal_in;
    output signal_out;

    reg signal_out = 0;

  always @ (posedge clk) begin
    if (reset) signal_out <=0;
    else if (signal_in) signal_out <= ~signal_out;
  end

endmodule
```

```
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
module debounce (clock, reset, noisy, clean);
   parameter DELAY = 270000;   // .01 sec with a 27Mhz clock
   input reset, clock, noisy;
   output clean;

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

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

endmodule

////////////////////////////////////////////////////////////////////////
/////
//
// 6.111 FPGA Labkit -- Hex display driver
//
// File:   display_16hex.v
// Date:   24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// This module drives the labkit hex displays and shows the value of
// 8 bytes (16 hex digits) on the displays.
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove
clear
//
// This verilog module drives the labkit hex dot matrix displays, and puts
// up 16 hexadecimal digits (8 bytes).  These are passed to the module
// through a 64 bit wire ("data"), asynchronously.
//
////////////////////////////////////////////////////////////////////////
/////

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

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

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

   reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;


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

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

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

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

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

   assign dreset = (reset_count != 0);

   assign disp_clock = ~clock;


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

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

   reg [7:0] state;           // FSM state
   reg [9:0] dot_index;       // index to current dot being clocked out
```

```
reg [31:0] control;        // control register
reg [3:0] char_index;      // index of current character
reg [39:0] dots;           // dots for a single digit
reg [3:0] nibble;          // hex nibble of current character

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

always @(posedge clock)
  if (dreset)
    begin
     state <= 0;
     dot_index <= 0;
     control <= 32'h7F7F7F7F;
    end
  else
    casex (state)
    8'h00:
      begin
         // Reset displays
         disp_data_out <= 1'b0;
         disp_rs <= 1'b0; // dot register
         disp_ce_b <= 1'b1;
         disp_reset_b <= 1'b0;
         dot_index <= 0;
         state <= state+1;
      end

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

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

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

    8'h04:
      begin
         // Setup the control register
         disp_rs <= 1'b1; // Select the control register
         disp_ce_b <= 1'b0;
         disp_data_out <= control[31];
         control <= {control[30:0], 1'b0};   // shift left
         if (dot_index == 0)
```

```
          state <= state+1;
          else
          dot_index <= dot_index-1;
        end

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

      8'h06:
        begin
          // Load the user's dot data into the dot reg, char by char
          disp_rs <= 1'b0;              // Select the dot register
          disp_ce_b <= 1'b0;
          disp_data_out <= dots[dot_index]; // dot data from msb
          if (dot_index == 0)
            if (char_index == 0)
              state <= 5;               // all done, latch data
          else
          begin
            char_index <= char_index - 1;     // goto next char
            dot_index <= 39;
          end
          else
          dot_index <= dot_index-1;     // else loop thru all dots
        end

      endcase

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

  always @(nibble)
    case (nibble)
      4'h0: dots <= 40'b00111110_01010001_01001001_01000101_00111110;
      4'h1: dots <= 40'b00000000_01000010_01111111_01000000_00000000;
      4'h2: dots <= 40'b01100010_01010001_01001001_01001001_01000110;
      4'h3: dots <= 40'b00100010_01000001_01001001_01001001_00110110;
      4'h4: dots <= 40'b00011000_00010100_00010010_01111111_00010000;
```

```
       4'h5: dots <= 40'b00100111_01000101_01000101_01000101_00111001;
       4'h6: dots <= 40'b00111100_01001010_01001001_01001001_00110000;
       4'h7: dots <= 40'b00000001_01110001_00001001_00000101_00000011;
       4'h8: dots <= 40'b00110110_01001001_01001001_01001001_00110110;
       4'h9: dots <= 40'b00000110_01001001_01001001_00101001_00011110;
       4'hA: dots <= 40'b01111110_00001001_00001001_00001001_01111110;
       4'hB: dots <= 40'b01111111_01001001_01001001_01001001_00110110;
       4'hC: dots <= 40'b00111110_01000001_01000001_01000001_00100010;
       4'hD: dots <= 40'b01111111_01000001_01000001_01000001_00111110;
       4'hE: dots <= 40'b01111111_01001001_01001001_01001001_01000001;
       4'hF: dots <= 40'b01111111_00001001_00001001_00001001_00000001;
     endcase

endmodule

//module to synchronise the 27MHz inputs to the 65MHz clock (for video):
double MUX

module synchroniser(clock_65mhz,old,new);

    input clock_65mhz;
    input old;
    output new;

    reg olg, new, mid;

    always@(posedge clock_65mhz) begin

        new <= mid;
        mid <= old;

    end

endmodule

// synthesizes required sound from dds module



module
synthesizer(clock_27mhz,reset,note,st_note,rand_note,synth_note,played_not
e,

                        playback_new,game,pinc,dds_chan);



    input clock_27mhz;          // system clock

    input [3:0] note;           // note from decoder module (live
playback)

    input [3:0] st_note;          // stored note from storage module
(playback mode)

    input [3:0] rand_note;      // random note requested by game module
(game mode)

    input playback_new;         // high when playback mode

    input reset;          // reset (on Kreset or reset_labkit)
```

```
    input game;                 // high when game_enb_hard or
game_enb_easy

    output [19:0] synth_note;   // to be assigned to to_ac97_data:
synnthesized note

    output [26:0] pinc;         // phase increment, outputted for
debugging

    output [4:0] dds_chan;      // outputted for debugging

    output [3:0] played_note;     // note, st_note or rand_note, depending
on mode


    parameter ldo = 4'b0000;      // note values (from [3:0]note)

    parameter lre = 4'b0001;

    parameter lmi = 4'b0010;

    parameter lfa = 4'b0011;

    parameter lsol = 4'b0100;

    parameter lla = 4'b0101;

    parameter lsi = 4'b0110;

    parameter hdo = 4'b0111;

    parameter hre = 4'b1000;

    parameter hmi = 4'b1001;

    parameter hfa = 4'b1010;

    parameter hsol = 4'b1011;

    parameter hla = 4'b1100;

    parameter hsi = 4'b1101;

    parameter vhdo = 4'b1110;


    parameter fldo = 262;           // note fundamental frequency

    parameter flre = 294;

    parameter flmi = 330;

    parameter flfa = 349;

    parameter flsol = 392;

    parameter flla = 440;
```

```
    parameter flsi = 494;

    parameter fhdo = 524;

    parameter fhre = 588;

    parameter fhmi = 660;

    parameter fhfa = 698;

    parameter fhsol = 784;

    parameter fhla = 880;

    parameter fhsi = 988;

    parameter fvhdo = 1048;


    parameter pincldo = 351650;       // phase increment for each note

    parameter pinclre = 394600;       // pinc = freq * (2^bit_size) /
clk_dds

    parameter pinclmi = 442918;

    parameter pinclfa = 468419;

    parameter pinclsol = 526133;

    parameter pinclla = 590558;

    parameter pinclsi = 663035;

    parameter pinchdo = 701958;

    parameter pinchre = 789200;

    parameter pinchmi = 885837;

    parameter pinchfa = 936839;

    parameter pinchsol = 1052266;

    parameter pinchla = 1181116;

    parameter pinchsi = 1326071;

    parameter pincvhdo = 1406600;


   reg [26 : 0] pinc;           // phase increment. defines output
frequency: freq = 0.1Mhz * pinc /(2^36)

   reg [26 : 0]  old_pinc;

   reg we;               // write enable for DDS channels(active high)

   reg [4 : 0] dds_chan;       // dds channel address
```

```
   wire RFD;                   // not used

   wire RDY;                   //not used

   wire [5 : 0] sine;          // dds output

   wire [5 : 0] cosine;        // dds output

   wire clk_dds;         // dds clock enable    (0.1MHz)


// declare multi channel dds module

fundamentals
dds(pinc,we,dds_chan,clock_27mhz,clk_dds,RFD,RDY,sine,cosine);



// declare 0.1 MHz CE module

dds_divider CE(clock_27mhz,reset,clk_dds/*,count*/);



// find negedge of playback_new

   wire neg_playback;

   reg old_playback;
   always @ (posedge clock_27mhz) old_playback <= (reset) ? 0 :
playback_new;
   assign neg_playback = ~playback_new & old_playback;         //when 1 ->
0 transition



   assign played_note = (game) ? rand_note :

                               (playback_new) ? st_note :

                               (neg_playback) ? 4'b1111 :            //no
sound when playback mode stopped

                                note;



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

// note synthesis (FSM)                                              /

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



    always @ (posedge clock_27mhz) begin
```

```
    old_pinc <= pinc;         // WE high for one clock pulse when pinc is
changed.
    we <= ~(pinc == old_pinc);



    dds_chan <= 5'b00000;    // writes into phase increment of channel 0
(fundamental only).

    pinc <= 32'b0;



    if (reset) begin

            we<=0;

            end



    else case (played_note)

    ldo: begin

             pinc <= pincldo;

             end

    lre: begin

             pinc <= pinclre;

             end



    lmi: begin

             pinc <= pinclmi;

             end



    lfa: begin

             pinc <= pinclfa;

             end



    lsol: begin

              pinc <= pinclsol;

              end
```

```
lla: begin
       pinc <= pinclla;
         end

lsi: begin
         pinc <= pinclsi;
         end

hdo: begin
        pinc <= pinchdo;
          end

hre: begin
        pinc <= pinchre;
          end

hmi: begin
        pinc <= pinchmi;
          end

hfa: begin
        pinc <= pinchfa;
          end

hsol: begin
        pinc <= pinchsol;
          end

hla: begin
        pinc <= pinchla;
          end
```

```
    hsi: begin
            pinc <= pinchsi;
              end

    vhdo: begin
              pinc <= pincvhdo;
              end

    default: begin                    // if no input, no sound (hopefully!)
            pinc <= 32'b0;
              end

    endcase
end // always

   assign synth_note = {sine,12'b0};        // output of the dds is
synthesized note.

endmodule

//////////////////////////////////////////////////////////////////////////
///
////
//// dds divider module, to get 0.1MHz clock enable for DDS module
////             (for 27MHz system clock)
//////////////////////////////////////////////////////////////////////////
///

module dds_divider(clk,reset,clk_enb);

    input clk;            // 27MHz clock
```

```
    input reset;    // system reset

    output clk_enb;       // 0.1MHz CE



    reg [8:0] count = 0;

    reg clk_enb = 0;



    parameter end_count = 270;



    always @ (posedge clk) begin



      count <= (reset) ? 0 :                           // counts up to
end_count and loops back to zero

                (count == end_count) ? 0 :

                  (count + 1);



       clk_enb <= (count == end_count) ? 1 : 0; // always low except for
one clock cycle when end_count reached



    end // always

endmodule   // dds_divider


module divider(clk, reset, ten_hz_enb);

    input clk;            // 27MHz clock
    input reset;    // system reset
    output ten_hz_enb;    // 0.1MHz CE

    reg [8:0] count = 0;
    reg ten_hz_enb = 0;

    parameter end_count = 2700000;

    always @ (posedge clk) begin

      count <= (reset) ? 0 :                           // counts up to
end_count and loops back to zero
                (count == end_count) ? 0 :
                  (count + 1);

       ten_hz_enb <= (count == end_count) ? 1 : 0;    // always low except
for one clock cycle when end_count reached
```

```
    end // always
endmodule
```